3-Е ИЗДАНИЕ

Python

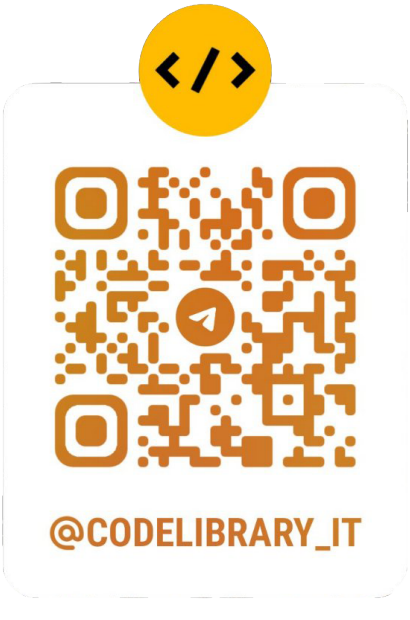
Экспресс-курс

Наоми Седер

MANNING

The Quick Python Book

THIRD EDITION



NAOMI CEDER
FOREWORD BY NICHOLAS TOLLERVEY

MANNING
SHELTER ISLAND

Наоми Седер

Python

Экспресс-курс

3-Е ИЗДАНИЕ

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2019

Наоми Седер

Python. Экспресс-курс

3-е издание

Серия «Библиотека программиста»

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Бульченко*
Художественный редактор	*С. Заматевская*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

ББК 32.973.2-018.1
УДК 004.43

Седер Наоми

С28 Python. Экспресс-курс. 3-е изд. — СПб.: Питер, 2019. — 480 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0908-1

Вы уже умеете кодить на одном или нескольких языках программирования? Тогда настала пора пройти экспресс-курс Python.

Впервые на русском языке выходит новое издание одной из самых популярных книг издательства Manning.

С помощью этой книги вы можете быстро перейти от основ к управлению и структурам данных, чтобы создавать, тестировать и развертывать полноценные приложения. Наоми Седер рассказывает не только об основных особенностях языка Python, но и его объектно-ориентированных возможностях, которые появились в Python 3. Данное издание учитывает все изменения, которые произошли с языком за последние 5 лет, а последние 5 глав рассказывают о работе с большими данными.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1617294037 англ.
ISBN 978-5-4461-0908-1

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 11.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 31.10.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 38,700. Тираж 1700. Заказ 0000.

Оглавление

Отзывы о втором издании

Самый быстрый способ изучить основы Python.

— Массимо Перга (Massimo Perga), Microsoft

Это моя любимая книга по Python. Грамотный подход к серьезному программированию на Python.

— Эдмон Беголи (Edmon Begoli), Oak Ridge National Laboratory

Превосходная книга, охватывает новое воплощение Python.

— Уильям Кайн-Грин (William Kahn-Greene),
Participatory Culture Foundation

Как и в самом Python, акцент в книге делается на читаемость и быстрое развитие.

— Дэвид МакВиртер (David McWhirter), Cranberryink

Эту книгу определенно стоит прочитать, и я бы рекомендовал вам купить ее, если вы новичок в Python.

— Джером Ланиг (Jerome Lanig), BayPiggies User Group

Кодерам-питонистам понравится эта замечательная книга.

— Сумит Пал (Sumit Pal), Leapfrogrx

Если вы когда-либо хотели изучать Python или иметь удобный справочник, то эта книга для вас. Автор кратко описывает синтаксис языка и функциональные возможности, а затем исследует все аспекты языка, а также библиотеки и модули, расширяющие Python в пространстве практических приложений.

— Джим Коли (Jim Kohli), Dzone

Это лучшая книга для изучения Python, которая подойдет профессиональным программистам или людям, которые уже знают, как программировать на другом языке... Она не станет вашей единственной книгой по Python, но определенно должна быть первой!

— Отзыв читателя на Amazon

Введение

Я знаю Наоми Седер уже много лет — как соавтора и как друга. В сообществе Python у нее репутация преподавателя-энтузиаста, опытного программиста и классного организатора сообщества. Каждому будет полезно прислушаться к ее мудрым словам.

Но не верьте мне на слово! Наоми в качестве преподавателя помогла очень многим людям в изучении Python. Многие участники сообщества Python, в том числе и я, пользовались результатами ее работы. Благодаря столь обширному опыту она знает, какие аспекты языка важны для начинающих питонистов и какие потребуют особого внимания от студентов. Эта мудрость была искусно переработана в страницы книги.

Как известно, язык Python соблюдает принцип «батарейки в комплекте»: вы можете сделать очень многое, а обширная экосистема модулей Python распространяется на очень многие практические области. Пришло время заняться изучением этого мощного, простого и успешно развивающегося языка.

«Экспресс»-характер книги выражается в лаконичном стиле преподавания Наоми и гарантирует, что вся основная информация о Python будет у вас под рукой. Кроме того, эта основная информация закладывает прочный фундамент для дальнейшего развития ваших навыков программирования на Python. Но что самое важное, после прочтения этой книги вы получите достаточно знаний и деталей для того, чтобы действовать независимо и эффективно: вы будете знать, что делать, где искать информацию и на что обращать внимание, когда вы неизбежно столкнетесь с препятствиями на своем пути к цели — карьере разработчика Python.

Книга Наоми воплощает суть «питонического» стиля: красивое лучше, чем уродливое, простое лучше, чем сложное, читаемость имеет значение.

Вы держите в руках замечательное руководство, которое поможет вам сделать первые шаги в изучении Python. Желаю удачи на предстоящем пути — и не забудьте получить удовольствие!

Николас Толлерви (Nicholas Tollervey),
сотрудник Python Software Foundation

Предисловие

Я программирую на Python уже 16 лет — намного больше, чем на любом другом языке, на котором я когда-либо работала. За эти годы я использовала Python для системного администрирования, для веб-приложений, для управления базами данных и для анализа данных, но что самое важное, я стала использовать Python просто для того, чтобы более наглядно представлять себе задачи.

На основании моего предыдущего опыта можно было бы ожидать, что к настоящему моменту я уже могла бы увлечься другим языком — более быстрым, крутым, эффектным и т. д. Думаю, этого не произошло по двум причинам. Во-первых, хотя другие языки не стоят на месте, ни один не помог мне делать то, что мне нужно, с той же эффективностью, что и Python. Даже после всех этих лет чем больше я использую Python, чем лучше понимаю его, тем сильнее я чувствую, что моя квалификация программиста растет и крепнет.

Вторая причина, по которой я остаюсь приверженцем языка, — сообщество Python. Это одно из самых доброжелательных, активных и дружественных сообществ, которые я видела; в него входят ученые, специалисты по анализу данных, веб-разработчики, системные администраторы и специалисты по теории данных на каждом континенте. Для меня работа с участниками этого сообщества была и честью, и удовольствием, и я рекомендую всем присоединяться к нему.

Написание этой книги было своего рода путешествием. Хотя сейчас мы все еще имеем дело с Python 3, сегодняшняя версия Python 3 прошла заметный путь по сравнению с 3.1; изменился и подход людей к использованию Python. И хотя я всегда стремилась сохранить лучшие фрагменты предыдущего издания, в этой книге вы найдете многочисленные дополнения, удаления и перестановки, благодаря которым (как я надеюсь) это издание стало как более полезным, так и более актуальным. Я постаралась сохранить четкий и сдержанный стиль изложения, не впадая в занудство.

Что касается меня, в этой книге я хотела поделиться положительным опытом, полученным мной в ходе обучения других людей Python 3 — последней и, на мой взгляд, лучшей из современных версий Python. Пусть ваше путешествие будет таким же увлекательным, каким было мое!

Благодарности

Я хочу поблагодарить Дэвида Фугейта (David Fugate) из LaunchBooks за то, что он когда-то привел меня к идее этой книги, и за всю поддержку и советы, полученные от него за прошедшие годы. Не могу представить себе лучшего агента и друга. Также хочу поблагодарить Майкла Стивенса (Michael Stephens) из Manning за продвижение идеи третьего издания книги и за поддержку моих усилий сделать ее такой же хорошей, как первые две. Кроме того, я хочу поблагодарить всех сотрудников Manning, работавших над проектом, и выразить особую благодарность Мариану Бэйсу (Marjan Bace) за поддержку; Кристине Тейлор (Christina Taylor) за направляющую работу в разных фазах разработки; Дженет Вейл (Janet Vail) за помощь в продвижении книги в производственном процессе; Кэти Симпсон (Kathy Simpson) за терпение в процессе редактирования; Элизабет Мартин (Elizabeth Martin) за правку текста. Кроме того, я сердечно благодарю многочисленных рецензентов, чьи наблюдения и личные мнения нам очень помогли: это Андре Филипе де Ассунсо э Брито (André Filipe de Assunção e Brito), научный редактор этого издания, а также Аарон Дженсен (Aaron Jensen), Эл Норман (Al Norman), Брукс Изольди (Brooks Isoldi), Карлос Фернандес Манзано (Carlos Fernández Manzano), Кристос Паисиос (Christos Paisios), Эрос Педрини (Eros Pedrini), Фелипе Эстебан Вилдосо Кастильо (Felipe Esteban Vildoso Castillo), Джулиано Латини (Giuliano Latini), Иэн Стерк (Ian Stirk), Негмат Муллоджанов (Negmat Mullodzhanov), Рик Оллер (Rick Oller), Роберт Траусмут (Robert Trausmuth), Руслан Видерт (Ruslan Vidert), Шобха Айер (Shobha Iyer) и Уильям Э. Уилер (William E. Wheeler).

Я должна поблагодарить авторов первого издания, Дарила Хармса (Daryl Harms) и Кеннета Макдоналда (Kenneth MacDonald); они написали книгу настолько выдающуюся, что она продолжает печататься далеко за пределами среднего срока актуальности большинства технических книг, а также предоставили мне возможность обновить второе (а сейчас и третье) издание. Спасибо всем, кто купил второе издание и оставил положительные отзывы о нем. Надеюсь, эта версия продолжит успешные и давние традиции первого и второго издания.

Также спасибо Николасу Толлерви (Nicholas Tollervey) за доброту (не говоря уже о скорости), с которой он написал предисловие к этому изданию, за наши годы дружбы и за все, что он сделал для сообщества Python. Я также приношу свою благодарность всемирному сообществу Python — безотказному источнику поддержки, мудрости, дружбы и радости в течение многих лет. Спасибо вам, друзья. Спасибо и моему четвероногому другу Эрин, которая преданно составляла мне компанию и помогала сохранить адекватность во время работы над этим изданием (и над вторым тоже).

Но самое важное, как обычно, — спасибо моей жене Бекки (Becky), которая уговорила меня взяться за этот проект и поддерживала меня на протяжении всего процесса. Без нее я бы не справилась.

О книге

Эта книга предназначена для людей, которые уже обладают опытом работы на одном или нескольких языках программирования и хотят по возможности быстро и просто изучить основы Python 3. Хотя в книге рассмотрены некоторые основные концепции, я не пытаюсь учить читателя фундаментальным понятиям программирования. Предполагается, что читатель уже знаком с управляющими конструкциями, ООП, работой с файлами, обработкой исключений и т. д. Книга также пригодится пользователям более ранних версий Python, которым нужен компактный справочник по Python 3.1.

Как использовать книгу

В части 1 приводится общая информация о Python. Вы узнаете, как загрузить и установить Python в вашей системе. Также здесь приводится общий обзор языка, который будет полезен прежде всего для опытных программистов, желающих получить высокоуровневое представление о Python.

Часть 2 содержит основной материал книги. В ней рассматриваются ингредиенты, необходимые для получения практических навыков использования Python как языка программирования общего назначения. Материал глав был спланирован так, чтобы читатели, только начинающие изучать Python, могли последовательно двигаться вперед, осваивая ключевые моменты языка. В этой части также присутствуют более сложные разделы, чтобы вы могли потом вернуться и найти всю необходимую информацию о некоторой конструкции или теме в одном месте.

В части 3 представлены расширенные возможности Python — элементы языка, которые не являются абсолютно необходимыми, но, безусловно, очень пригодятся любому серьезному программисту Python.

Часть 4 посвящена специализированным темам, выходящим за рамки простого синтаксиса языка. Вы можете читать эти главы или пропустить их в зависимости от ваших потребностей.

Начинающим программистам Python рекомендуется начать с главы 3, чтобы составить общее впечатление, а затем перейти к интересующим главам части 2. Вводите интерактивные примеры, чтобы немедленно закрепить концепции. Вы также можете выйти за рамки примеров, приведенных в тексте, и искать ответы на любые вопросы, оставшиеся неясными. Такой подход повысит скорость обучения и углубит понимание. Если вы еще не знакомы с ООП или оно не требуется для вашего приложения, вы можете пропустить большую часть главы 15.

Читателям, уже знакомым с Python, также стоит начать с главы 3. Она содержит хороший вводный обзор и описание различий между Python 3 и более знакомыми

версиями. По ней также можно оценить, готовы ли вы перейти к более сложным главам частей 3 и 4 этой книги.

Возможно, некоторые читатели, не имеющие опыта работы с Python, но имеющие достаточный опыт в других языках программирования, смогут получить большую часть необходимой информации, прочитав главу 3 и просмотрев модули стандартной библиотеки Python (глава 19) и справочное руководство по библиотеке Python в документации Python.

Структура книги

В главе 1 обсуждаются сильные и слабые стороны Python, а также объясняется, почему Python 3 хорошо подходит на роль языка программирования во многих практических ситуациях.

В главе 2 рассматривается загрузка, установка и запуск интерпретатора Python и IDLE, его интегрированной среды разработки.

Глава 3 представляет собой краткий обзор языка Python. Она дает представление об основах философии, синтаксиса, семантики и возможностей языка.

В главе 4 начинается изложение основ Python. В ней представлены переменные Python, выражения, строки и числа, а также синтаксис блочной структуры Python.

В главах 5, 6 и 7 описаны пять мощных встроенных типов данных Python: списки, кортежи, множества, строки и словари.

Глава 8 посвящена синтаксису и использованию управляющих конструкций Python (циклы и команды `if-else`).

В главе 9 описаны определения функций в Python и гибкие средства передачи параметров.

В главе 10 рассматриваются модули Python, которые предоставляют удобный механизм сегментирования пространств имен программы.

Глава 11 посвящена созданию автономных программ Python (сценариев) и их выполнению на платформах Windows, macOS и Linux. В этой главе также рассматривается поддержка параметров командной строки, аргументов и перенаправления ввода/вывода.

Из главы 12 вы узнаете, как работать с файлами и каталогами файловой системы и как перемещаться между ними. Она показывает, как написать код, по возможности независимый от операционной системы, в которой вы работаете.

В главе 13 представлены механизмы чтения и записи файлов в Python, включая основные средства чтения и записи строк (или потоков байтов), механизм чтения двоичных записей и средства чтения/записи произвольных объектов Python.

Глава 14 посвящена исключениям — механизму обработки ошибок, используемому в Python. Глава не требует знания исключений, хотя если вы уже пользовались ими в C++ или Java, они покажутся вам знакомыми.

В главе 15 описаны средства Python для написания объектно-ориентированных программ.

В главе 16 рассматривается поддержка регулярных выражений в Python.

В главе 17 представлены расширенные средства ООП, включая механизмы специальных методов-атрибутов, метаклассов и абстрактных базовых классов.

В главе 18 представлена концепция пакетов в Python и ее роль в организации кода больших проектов.

Глава 19 содержит краткий обзор стандартной библиотеки. В ней также рассказано о том, где найти другие модули и как установить их.

В главе 20 тема работы с файлами в Python рассматривается более подробно.

В главе 21 рассматриваются различные стратегии чтения, очистки и записи различных типов файлов данных.

Глава 22 содержит обзор основных процессов, проблем и инструментов, применяемых при загрузке данных по Сети.

В главе 23 обсуждаются средства работы с реляционными базами данных и базами данных NoSQL в Python.

Глава 24 содержит краткое введение в анализ данных с применением Python, Jupyter Notebook и `pandas`.

Практический пример проведет вас по основным фазам применения Python для загрузки данных, их очистки и графического представления. Проект объединяет сразу несколько возможностей языка, рассмотренных в предшествующих главах, и дает возможность понаблюдать за работой над проектом от начала до конца.

Из приложения А вы узнаете, как получить полную документацию Python и как лучше работать с ней. Здесь также приведено руководство по стилю Python, PEP 8 и Дзен Python — ироничное краткое содержание философии Python.

В приложении Б приведены ответы на большинство упражнений в книге. В отдельных случаях упражнение предполагает самостоятельные эксперименты. Для таких упражнений ответы не приводятся.

Правила оформления кода

Приведенные в книге примеры кода и результаты их выполнения оформлены `моноширинным шрифтом` и часто сопровождаются выносками. Примеры кода намеренно сделаны настолько простыми, насколько это возможно, потому что они не задумывались как полезные фрагменты, которые вы сможете включать в свой код. Вместо этого примеры кода предельно усечены, чтобы вы могли сосредоточиться на демонстрируемых принципах.

В соответствии с этой идеей простоты примеры кода по возможности представляются в виде интерактивных сеансов; вы можете ввести эти примеры и как угодно экспериментировать с ними. В интерактивных примерах вводимые команды

размещаются в строках, начинающихся с приглашения >>>, а результаты их выполнения (если они есть) выводятся со следующей строки.

В некоторых случаях требуется более длинный фрагмент кода. Такие случаи представлены в тексте листингами. Сохраните эти фрагменты кода в файлах с именами, совпадающими с приведенными в тексте, и запустите их как автономные сценарии.

Упражнения

Начиная с главы 4, в книге приводятся упражнения трех типов. Упражнения «Быстрая проверка» представляют собой очень короткие вопросы, которые помогут вам ненадолго остановиться и убедиться в том, что вы хорошо поняли только что представленную идею. Упражнения «Попробуйте сами» потребуют от читателя чуть большего — ему придется самостоятельно написать код Python. Многие главы завершаются упражнениями «Практическая работа», которые дают возможность объединить концепции текущей и предыдущих глав в один полноценный сценарий.

Ответы к упражнениям

Ответы ко многим упражнениям приводятся в приложении Б. Кроме того, они включены в отдельный каталог в архиве исходного кода книги. Помните, что ответы не могут считаться *единственно* правильными; возможны и другие решения. Лучший способ оценить ваши ответы — понять, как работает предлагаемое решение, и решить, достигает ли ваш вариант тех же целей.

Исходный код

Исходный код примеров этой книги можно загрузить на сайте издателя по адресу www.manning.com/books/the-quick-python-book-third-edition.

Системные требования

Примеры и весь код в этой книге были написаны с учетом особенностей Windows (Windows 7–10), macOS и Linux. Так как Python является кроссплатформенным языком, примеры и код *должны* работать на других платформах почти всегда — кроме таких платформенно-зависимых аспектов, как операции с файлами, путями и графическим интерфейсом.

Программные требования

Эта книга написана на базе Python 3.6, и все примеры должны работать во всех последующих версиях Python 3. (Большинство примеров было протестировано в предварительной версии Python 3.7.) За редкими исключениями эти примеры также работают в Python 3.5, но я настоятельно рекомендую использовать версию 3.6;

у предыдущих версий нет никаких преимуществ, а в версию 3.6 был внесен ряд незаметных усовершенствований. Учтите, что версия Python 3 является обязательной, и код из книги не будет работать в более ранних версиях Python.

Об авторе

Наоми Седер, автор третьего издания книги, занимается программированием на различных языках в течение почти 30 лет. Она работала системным администратором Linux, преподавателем программирования, разработчиком и системным архитектором. Она начала использовать Python в 2001 году, и с тех пор преподавала Python пользователям всех уровней, от 12-летних до профессионалов. Она рассказывает о Python и о достоинствах Python-сообщества каждому, кто готов слушать. В настоящее время Наоми руководит группой разработки для Dick Blick Art Materials и является председателем Python Software Foundation.

Об иллюстрации на обложке

Иллюстрация на обложке книги позаимствована из четырехтомной энциклопедии национальных костюмов, написанной Сильвеном Марешалем (Sylvain Maréchal) и опубликованной во Франции в конце XVIII века. Каждая иллюстрация была тщательно прорисована и раскрашена вручную. Разнообразие коллекции Марешаля ярко напоминает нам, насколько непохожими в культурном отношении были города и области мира всего 200 лет назад. Люди, жившие в изоляции друг от друга, говорили на разных диалектах и языках. На улицах городов и в сельской местности можно было легко узнать, где живут люди и чем они занимаются, — достаточно было взглянуть на их одежду.

С тех пор стандарты внешнего вида изменились, и региональные различия, столь богатые в то время, остались в прошлом. Сейчас уже трудно отличить друг от друга жителей разных континентов, не говоря уже о разных городах или областях. Возможно, мы пожертвовали культурным разнообразием ради более разнообразной личной жизни и, безусловно, ради более разнообразной и стремительной технологической жизни.

В наше время, когда одну книгу о компьютерах трудно отличить от другой, издательство Manning воздает должное изобретательности и инициативности компьютерной отрасли обложками своих книг, отражающими разнообразие региональной жизни двухсотлетней давности. Эти обложки снова вернулись к жизни благодаря иллюстрациям Марешаля.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ЧАСТЬ 1

Первые шаги

Первые три главы немного расскажут вам о Python, его сильных и слабых сторонах и о том, почему вам стоит заняться изучением Python 3. В главе 2 вы узнаете, как установить Python на платформах Windows, macOS и Linux и как написать простую программу. В главе 3 приведен краткий высокоуровневый обзор синтаксиса и основных возможностей Python.

А если вы хотите как можно быстрее взяться за изучение Python, начинайте с главы 3.

1 Знакомство с Python

Эта глава охватывает следующие темы:

- ✓ Почему стоит использовать Python
- ✓ Чем хорош Python
- ✓ В чем Python не очень хорош
- ✓ Почему следует изучать Python 3

Прочитайте эту главу, если вы хотите знать, чем Python отличается от других языков и какое место он занимает в общей картине. Если же вы хотите немедленно взяться за изучение Python, эту главу можно пропустить и перейти прямо к главе 3. Информация в этой главе является существенной частью книги, но она безусловно не является необходимой для программирования на Python.

1.1. Почему мне стоит использовать Python?

В современном мире существуют сотни языков программирования, от проверенных временем (таких, как C и C++) до недавно появившихся Ruby, C# и Lua и колоссов коммерческой разработки типа Java. Выбрать язык для изучения не так просто. Хотя ни один язык не может считаться идеальным вариантом для каждой возможной ситуации, я думаю, что Python хорошо подходит для многих задач программирования; кроме того, он может стать хорошим кандидатом для изучения программирования. Сотни тысяч программистов по всему миру используют Python, и их число растет с каждым годом.

Python продолжает привлекать новых пользователей по разным причинам. Это полноценный кроссплатформенный язык, который одинаково хорошо работает на платформах Windows, Linux/UNIX и Macintosh, а также многих других, от суперкомпьютеров до сотовых телефонов. Он может использоваться для разработки маленьких приложений и прототипов, но также хорошо масштабируется для разработки больших программ. В поставку Python входит мощный и удобный

инструментарий построения графических интерфейсов (GUI), библиотеки для веб-программирования и многое другое. *И все это* бесплатно.

1.2. Сильные стороны Python

Python — современный язык программирования, созданный Гвидо ван Россумом (Guido van Rossum) в 1990-е годы (и получивший название в честь знаменитой комедийной труппы «Монти Пайтон»). Хотя Python нельзя назвать идеальным кандидатом для каждого приложения, благодаря своим сильным сторонам он хорошо подходит для многих ситуаций.

1.2.1. Python прост в использовании

У программистов, знакомых с традиционными языками программирования, не будет трудностей с изучением Python. В нем поддерживаются все знакомые конструкции — циклы, условные команды, массивы и т. д., однако многие из них проще в использовании. И вот почему:

- *Типы связываются с объектами, а не с переменными.* Переменной можно присвоить значение любого типа, а список может содержать объекты многих типов. Это также означает, что преобразование типа обычно оказывается лишним, а ваш код не скован кандалами заранее объявленных типов.
- *Python обычно работает на более высоком уровне абстракции.* Отчасти это связано с тем, как построен язык, а отчасти объясняется обширной библиотекой стандартного кода, включенной в поставку Python. Программа для загрузки веб-страницы займет всего две-три строки!
- *Правила синтаксиса очень просты.* Чтобы стать экспертом Python, потребуется время и немалые усилия, но даже новичок может усвоить синтаксис Python в достаточной мере для написания полезного кода.

Python хорошо подходит для быстрой разработки приложений. Программирование приложения на Python нередко занимает в пять раз меньше времени, чем для его реализации на C или Java, а приложение занимает впятеро меньше строк, чем эквивалентная программа на C. Конечно, это зависит от конкретного приложения; для числовых алгоритмов, выполняющих в основном целочисленные операции в циклах `for`, прирост производительности будет куда менее заметным. Но для среднего приложения выигрыш может оказаться весьма значительным.

1.2.2. Выразительность Python

Язык Python чрезвычайно выразителен. Под *выразительностью* в данном контексте понимается то, что одна строка кода Python может сделать намного больше, чем одна строка кода в других языках. Преимущества более выразительного языка очевидны: чем меньше строк кода вам придется написать, тем быстрее вы завершите проект. Чем меньше строк кода содержит программа, тем меньше проблем будет с сопровождением и отладкой.

Чтобы понять, как выразительность Python упрощает код, возьмем задачу перестановки значений двух переменных, `var1` и `var2`. В таком языке, как Java, для этого потребуются три строки кода и лишняя переменная:

```
int temp = var1;
var1 = var2;
var2 = temp;
```

Переменная `temp` необходима для хранения значения переменной `var1` в то время, когда в ней хранится `var2`, и последующего его сохранения в `var2`. Процесс не особенно сложен, но чтобы прочитать эти три строки и понять, что произошла перестановка, даже опытному программисту придется немного поразмыслить.

С другой стороны, Python позволяет выполнить ту же перестановку в одной строке, причем так, что читатель кода сразу понимает, что значения меняются местами:

```
var2, var1 = var1, var2
```

Конечно, это очень простой пример, но аналогичные возможности постоянно встречаются в языке.

1.2.3. Удобочитаемость кода Python

Еще одно преимущество кода Python заключается в том, что он легко читается. Казалось бы, язык программирования предназначен для компьютера, но людям тоже приходится читать ваш код: тем, кто занимается отладкой вашего кода (возможно, это будете вы), тем, кто занимается сопровождением вашего кода (опять-таки это можете быть вы), тем, кто будет изменять этот код в будущем. Во всех этих ситуациях чем понятнее будет код и чем проще он читается, тем лучше.

Чем понятнее код, тем проще он в отладке, сопровождении и модификации. Главное преимущество Python в этом отношении — использование отступов. В отличие от многих других языков, Python *требует*, чтобы блоки кода снабжались отступами. Кому-то это требование может показаться странным, но оно гарантирует, что ваш код всегда будет отформатирован в очень простом и удобочитаемом стиле.

Ниже приведены две короткие программы: одна написана на Perl, а другая на Python. Обе программы получают списки чисел одинакового размера и возвращают парную сумму этих списков. На мой взгляд, код Python читается лучше, чем код Perl; он визуально чище и содержит меньше невразумительных знаков:

```
# Версия Perl.
sub pairwise_sum {
    my($arg1, $arg2) = @_;
    my @result;
    for(0 .. $#$arg1) {
        push(@result, $arg1->[$_] + $arg2->[$_]);
    }
    return(\@result);
}
```

```
# Версия Python.
def pairwise_sum(list1, list2):
    result = []
    for i in range(len(list1)):
        result.append(list1[i] + list2[i])
    return result
```

Оба фрагмента делают одно и то же, но код Python побеждает в отношении удобочитаемости. (Конечно, на Perl то же можно сделать и другими способами, многие из которых гораздо компактнее приведенного, — но в моем представлении они читаются хуже.)

1.2.4. Полнота Python — «батарейки в комплекте»

Еще одно преимущество Python — его философия «батарейки в комплекте» относительно библиотек. Идея заключается в том, что при установке Python вы получаете все необходимое для реальной работы, и устанавливать дополнительные библиотеки уже не потребуется. Вот почему стандартная библиотека Python поставляется с модулями для работы с электронной почтой, веб-страницами, базами данных, функциями операционной системы, построения графического интерфейса и т. д.

Например, на языке Python веб-сервер для обеспечения совместного доступа к файлам в каталоге состоит всего из двух строк кода:

```
import http.server
http.server.test(HandlerClass=http.server.SimpleHTTPRequestHandler)
```

Нет необходимости устанавливать библиотеки для обработки сетевых подключений и поддержки HTTP, вся функциональность уже доступна в Python.

1.2.5. Кроссплатформенность

Python также является превосходным кроссплатформенным языком. Python работает на многих платформах: Windows, Mac, Linux, UNIX и т. д. Так как язык является интерпретируемым, один код может выполняться на любой платформе с интерпретатором Python, а сейчас он есть практически на всех современных платформах. Существуют даже версии Python, работающие на базе Java (Jython) и .NET (IronPython), что дополнительно расширяет круг возможных платформ для запуска Python.

1.2.6. Свободное распространение

Наконец, за Python не нужно платить. Python изначально разрабатывался (и продолжает разрабатываться) на базе модели открытого исходного кода и свободного распространения. Вы можете загрузить и установить практически любую версию Python, использовать ее для разработки коммерческих или личных приложений, и вам ни копейки не придется платить за это.

Хотя обстановка постепенно меняется, некоторые люди до сих пор опасаются бесплатных продуктов и недостаточного уровня поддержки по сравнению с платной моделью. Тем не менее для многих авторитетных компаний Python стал ключевой частью бизнеса. Google, Rackspace, Industrial Light & Magic, Honeywell — и это лишь несколько примеров. Эти и многие другие компании справедливо считают Python очень надежным, стабильным и хорошо поддерживаемым продуктом с активным и знающим сообществом пользователей. В интернет-группах Python даже на самый трудный вопрос можно получить ответ быстрее, чем на большинстве телефонных линий технической поддержки, причем ответ будет правильным и бесплатным.

PYTHON И ПРОДУКТЫ С ОТКРЫТЫМ КОДОМ

Python не только бесплатно распространяется, но и его исходный код находится в открытом доступе; при желании вы можете изменять, улучшать и расширять его. Вы можете заняться этим самостоятельно или нанять кого-нибудь, кто сделает это за вас. В любых программных продуктах с закрытым исходным кодом возможность модификации при сколько-нибудь разумных затратах обычно отсутствует.

Если вы впервые сталкиваетесь с миром продуктов с открытым исходным кодом, следует понимать, что вы можете не только свободно использовать и изменять Python, но и вносить свой вклад в его совершенствование (и это даже приветствуется). В зависимости от ваших обстоятельств, интересов и квалификации это может быть финансовый вклад (например, пожертвование для PSF — Python Software Foundation), участие в одной из специальных групп (SIG), тестирование и обратная связь по выпускам базовой версии Python или одного из вспомогательных модулей или применение разработок (ваших или вашей компании) в сообществе. Конечно, уровень участия зависит только от вас; но если вы можете что-то сделать для других — это определенно стоит того. Здесь общими усилиями создается нечто весьма полезное, и у вас есть возможность внести свой вклад.

Итак, в пользу Python есть масса доводов: выразительность, удобочитаемость, богатый набор библиотек, кроссплатформенность. И распространение с открытым кодом. В чем же подвох?

1.3. Не самые сильные стороны Python

Хотя Python обладает многими преимуществами, ни один язык не решает всех проблем, так что Python не станет идеальным решением на все случаи жизни. Чтобы решить, подходит ли Python для вашей ситуации, также нужно отметить те области, в которых Python проявляет себя не лучшим образом.

1.3.1. Python не самый быстрый язык

Один из возможных недостатков Python — скорость выполнения кода. Python не является компилируемым языком. Вместо этого код сначала компилируется во внутренний байт-код, который затем выполняется интерпретатором Python. В некоторых областях (например, обработке строк с использованием регулярных выражений) у Python существуют эффективные реализации, не уступающие по скорости любым программам C, а то и превосходящие их. Тем не менее в большинстве

случаев при использовании Python получаются программы более медленные по сравнению с такими языками, как C. Впрочем, на это следует взглянуть здраво: современные компьютеры обладают такой вычислительной мощностью, что для большинства приложений скорость разработки важнее скорости выполнения, а программы на Python обычно пишутся намного быстрее. Кроме того, Python легко расширяется модулями, написанными на C или C++; они могут использоваться для выполнения частей программы, создающих интенсивную нагрузку на процессор.

1.3.2. Python не является лидером по количеству библиотек

Хотя Python включает превосходную подборку библиотек и еще много библиотек находится в свободном доступе, Python не является лидером в этом отношении. Для таких языков, как C, Java и Perl, доступны еще бóльшие подборки библиотек. Иногда они предоставляют решение в тех случаях, когда в Python его нет, или предлагают несколько вариантов там, где Python предлагает только один вариант. Тем не менее такие ситуации обычно оказываются узкоспециализированными, а Python при необходимости легко расширяется — либо кодом Python, либо существующими библиотеками на C и других языках. Практически для всех повседневных вычислительных задач в библиотеке Python реализована превосходная поддержка.

1.3.3. Не проверяет тип переменных во время компиляции

В отличие от некоторых языков, переменные в Python не служат контейнерами для своих значений, скорее они больше похожи на метки для разных объектов: целых чисел, строк, экземпляров классов и т. д. Это означает, что хотя сами объекты обладают типом, ссылающиеся на них переменные не привязаны к этому конкретному типу. Возможно использовать переменную `x` для хранения строки в одной точке программы и целого числа в другой (впрочем, это не значит, что так стоит поступать):

```
>>> x = "2"
>>> x
'2  <-- x содержит строку "2"
>>> x = int(x)
>>> x
2  <-- теперь x содержит целое число 2
```

Тот факт, что Python связывает типы с объектами, а не с переменными, означает, что интерпретатор не поможет с выявлением несовпадений типов. Если вы включили в программу переменную для хранения целого числа, Python не будет протестовать, если присвоить ей строку `"two"`. Программисты с опытом работы на традиционных языках считают это недостатком, потому что вы лишаетесь дополнительной проверки в коде. Однако поиск и исправление таких ошибок обычно не создают особых проблем, а средства тестирования Python помогают устранять ошибки несоответствия типов. Многие программисты Python считают, что гибкость динамической типизации с лихвой перевешивает ее недостатки.

1.3.4. Слабая поддержка мобильных устройств

За последнее десятилетие появилось великое множество всевозможных мобильных устройств: смартфоны, планшеты, планшетофоны, хромбуки... Новые мобильные устройства повсюду, и на них работают самые разные операционные системы. Python не принадлежит к числу сильных игроков в этом секторе. Варианты запуска Python на мобильных устройствах не всегда просты (хотя они и существуют), а при попытках использования Python для написания и распространения коммерческих приложений начинаются проблемы.

1.3.5. Слабая многопроцессорная поддержка

В наши дни многоядерные процессоры встречаются повсюду, и во многих случаях они обеспечивают значительный прирост производительности. Однако стандартная реализация Python не рассчитана на использование нескольких ядер из-за механизма GIL (Global Interpreter Lock). За дополнительной информацией обращайтесь к видеороликам с обсуждениями GIL и сообщениями Дэвида Бизли (David Beazley), Ларри Гастингса (Larry Hastings) и других специалистов или посетите страницу GIL в вики Python по адресу https://wiki.python.org/moin/GlobalInterpreterLock. Хотя выполнение параллельных процессов с использованием Python возможно, если вам нужны встроенные средства параллелизации, Python вряд ли будет лучшим кандидатом.

1.4. Почему нужно изучать Python 3?

Язык Python появился достаточно давно, и он развивался со временем. Первое издание этой книги было написано для Python 1.5.2, потом несколько лет доминировала версия Python 2.*x*. Эта книга написана на основе Python 3.6, хотя материал также тестировался в альфа-версии Python 3.7.

Python 3, в исходном варианте по какой-то прихоти названный Python 3000, заслуживает внимания уже потому, что это первая версия Python в истории языка, в которой была нарушена обратная совместимость. Это означает, что код, написанный для более ранних версий Python, скорее всего, не будет работать в Python 3 без некоторых изменений. Например, в более ранних версиях Python аргументы команды `print` можно было не заключать в круглые скобки:

```
print "hello"
```

В Python 3 `print` является функцией, поэтому круглые скобки обязательны:

```
print("hello")
```

Возможно, вы думаете: «Зачем изменять такие мелочи, если это нарушит работоспособность старого кода?» Потому что такие изменения станут большим шагом вперед для любого языка, и разработчики Python тщательно продумали этот вопрос. Хотя изменения в Python 3 нарушают совместимость со старым кодом, эти изменения относительно невелики и направлены к лучшему; с ними язык становится более последовательным, более удобочитаемым и однозначным. Python 3 не является

кардинальной переработкой языка, скорее это хорошо продуманный этап эволюции. Разработчики языкового ядра также постарались предоставить стратегию и инструментарий безопасной и эффективной миграции старого кода на Python 3 (об этом будет рассказано в одной из последующих глав). Также существуют библиотеки Six и Future, упрощающие переход.

Почему нужно изучать Python 3? Потому что это лучшая версия Python на данный момент. Кроме того, поскольку проекты переходят на использование усовершенствований этой версии, эта версия Python станет доминирующей на ближайшие годы. Портирование библиотек для Python 3 неуклонно идет с момента выхода версии, и в настоящее время многие популярнейшие библиотеки поддерживают Python 3. По данным Python Readiness (http://py3readiness.org), 319 из 360 самых популярных библиотек уже были портированы для Python 3. Если вам понадобится библиотека, которая еще не была конвертирована, или если вы работаете над уже сформированной кодовой базой, написанной на Python 2, — пожалуйста, используйте Python 2.*x*. Но если вы только начинаете изучать Python или открываете новый проект, выбирайте Python 3. Эта версия не просто лучше — за ней будущее.

Итоги

- Python — современный высокоуровневый язык с динамической типизацией, простым логичным синтаксисом и семантикой.
- Python — многоплатформенный язык с высокой модульностью, хорошо подходящий как для ускоренной разработки, так и для крупномасштабного программирования.
- Python работает достаточно быстро и может легко расширяться модулями C или C++ для повышения скорости.
- Python обладает такими встроенными нетривиальными возможностями, как долгосрочное хранение объектов, мощные хеш-таблицы, расширяемый синтаксис классов и универсальные функции сравнения.
- Python включает подборку библиотеки для обработки числовых данных, обработки графики, построения пользовательских интерфейсов и веб-сценариев.
- Язык поддерживается динамическим сообществом Python.

2 Первые шаги

Эта глава охватывает следующие темы:

- ✓ Установка Python
- ✓ Использование IDLE и базового интерактивного режима
- ✓ Написание простой программы
- ✓ Использование окна оболочки IDLE в Python

В этой главе описан процесс загрузки, установки и запуска Python и IDLE — интегрированной среды разработки для Python. На момент написания книги Python 3.6 был самой последней версией, а версия 3.7 находилась в разработке. После нескольких лет доработки Python 3 стал первой версией языка, которая не обладает полной обратной совместимостью с предыдущими версиями, поэтому обязательно установите версию Python 3. Вероятно, до следующего столь же кардинального изменения пройдет несколько лет, и при всех будущих усовершенствованиях разработчики постараются избежать последствий для уже значительной существующей кодовой базы, а это значит, что материал, представленный в этой главе, вряд ли потеряет актуальность в обозримом будущем.

2.1. Установка Python

Процесс установки Python несложен, на какой бы платформе вы ни работали. Прежде всего следует найти последний дистрибутив для вашей машины; самую свежую версию всегда можно найти на сайте www.python.org. Эта книга написана на основе Python 3.6. Если вы используете Python 3.5 или даже 3.7 — ничего страшного. Материал книги может без особых хлопот использоваться с любой версией Python 3.

УСТАНОВКА НЕСКОЛЬКИХ ВЕРСИЙ PYTHON

Возможно, на вашем компьютере уже установлена более ранняя версия Python. Многие дистрибутивы Linux и macOS включают Python 2.*x* как часть операционной системы. Так как Python 3

не обладает полной совместимостью с Python 2, возникает резонный вопрос: не приведет ли установка обеих версий на одном компьютере к конфликту?

Не беспокойтесь, несколько версий Python вполне могут существовать на одном компьютере. В системах на базе UNIX, таких как OS X и Linux, Python 3 устанавливается параллельно со старой версией и не заменяет ее. Когда ваша система ищет команду «python», она найдет именно ту версию, которая вам нужна, а если вы захотите обратиться к Python 3, выполните команду `python3` или `idle`. В системе Windows разные версии устанавливаются в разных каталогах и для них создаются разные команды меню.

Ниже приводятся описания установки Python для конкретных платформ. Подробности могут значительно изменяться в зависимости от платформы, поэтому обязательно прочитайте инструкции на странице загрузки для разных версий. Скорее всего, вы уже умеете устанавливать программы на вашей машине, поэтому описания будут достаточно короткими:

- *Microsoft Windows* — в большинстве версий Windows поддержка Python может устанавливаться программой установки Python, которая в настоящее время называется python-3.6.1.exe. Загрузите программу, выполните ее и действуйте по инструкциям программы. Возможно, для запуска программы вам понадобятся права администратора. Если вы работаете по сети и у вас нет пароля администратора, попросите системного администратора провести установку за вас.
- *Macintosh* — установите версию Python 3, соответствующую вашей версии OS X и вашему процессору. После того как вы определите правильную версию, загрузите файл с образом диска, смонтируйте его двойным щелчком и запустите программу установки. Программа для OS X настраивает все параметры автоматически, и Python 3 будет установлен во вложенную папку внутри папки Applications с указанием номера версии. macOS поставляется с разными версиями Python, но вам не нужно беспокоиться об этом; Python 3 будет установлен *в дополнение* к системной версии. Если у вас установлена система `brew`, вы также можете воспользоваться ею для установки Python командой `brew install python3`. Вы найдете дополнительную информацию об использовании Python в OS X по ссылкам на домашней странице Python.
- *Linux/UNIX* — поддержка Python присутствует в большинстве дистрибутивов Linux. Тем не менее версии Python могут быть разными и нет гарантий, что установлена будет именно версия 3; для материала этой книги необходимо убедиться в том, что у вас установлены пакеты Python 3. Также может оказаться, что среда IDLE не установлена по умолчанию, и этот пакет нужно будет установить отдельно. И хотя Python можно построить из исходного кода, доступного на сайте www.python.org, вам потребуются дополнительные библиотеки, а процесс построения не для новичков. Если существует заранее откомпилированная версия для вашего дистрибутива Linux, я рекомендую воспользоваться ею. Используйте систему управления пакетами для вашего дистрибутива, чтобы найти и установить нужные пакеты для Python 3 и IDLE. Также доступны версии для запуска Python во многих операционных системах. Актуальный список поддерживаемых платформ с подробными описаниями установки доступен на сайте www.python.org.

ANACONDA: АЛЬТЕРНАТИВНЫЙ ДИСТРИБУТИВ PYTHON

Кроме дистрибутива Python, который можно загрузить прямо с Python.org, сейчас набирает популярность дистрибутив Anaconda, особенно в среде ученых и специалистов по обработке данных. Anaconda — открытая платформа на базе Python. При установке Anaconda вы получаете не только Python, но и язык R с обширной подборкой заранее установленных пакетов анализа и обработки данных, к которой можно добавить много других пакетов при помощи встроенного менеджера пакетов conda. Также возможно установить версию miniconda, включающую только Python и conda, а затем добавить к ней необходимые пакеты.

Anaconda или miniconda можно загрузить по адресу www.anaconda.com/download/. Загрузите версию программы установки Python 3 для вашей операционной системы и запустите ее в соответствии с инструкциями. Когда все будет готово, на вашей машине появится полная версия Python.

Если ваши интересы лежат в области Data Science, возможно, Anaconda станет самым быстрым и простым способом начать работу с Python.

2.2. Базовый интерактивный режим и IDLE

Интерактивный доступ к интерпретатору Python можно получить двумя основными способами: исходный базовый режим (командная строка) и IDLE. Среда IDLE доступна на многих платформах, включая Windows, Mac и Linux, но на других платформах ее может не быть. Возможно, для запуска IDLE вам придется немного потрудиться и установить дополнительные программные пакеты, но эти усилия окупятся — работать с IDLE удобнее, чем в базовом интерактивном режиме. С другой стороны, даже если вы обычно используете IDLE, время от времени бывает нужно запустить базовый режим. Вы должны быть достаточно подготовлены, чтобы запустить и использовать любой из этих режимов.

2.2.1. Базовый интерактивный режим

Базовый интерактивный режим — достаточно примитивная среда, но интерактивные примеры, приведенные в книге, обычно невелики. Позднее вы узнаете, как легко подключить к сеансу код, хранящийся в файле (при помощи механизма модулей). Вот как запускается базовый интерактивный сеанс в Windows, macOS и UNIX:

- *Запуск базового сеанса в Windows* — для Python версии 3.x найдите команду запуска 32-разрядной версии Python 3.6 (32-разрядной) из подменю Python 3.6 папки Программы (Programs) меню Пуск (Start) и щелкните на ней. Также возможно найти исполняемый файл Python.exe (например, в папке C:\Users\myuser\AppData\Local\Programs\Python\Python35-32) и сделать на нем двойной щелчок. В результате открывается окно, показанное на рис. 2.1.

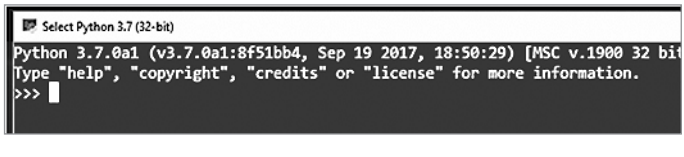


Рис. 2.1. Базовый интерактивный режим в Windows 10

- *Запуск базового сеанса в macOS* — откройте окно терминала и введите команду `python3`. Если вы получите ошибку типа «Команда не найдена», запустите сценарий `Update Shell Profile` из папки Python3 в папке Applications.
- *Запуск базового сеанса в UNIX* — введите команду `python3` в командной строке. В текущем окне выводится сообщение с указанием версии наподобие показанного на рис. 2.1, а за ним следует приглашение Python `>>>`.

ВЫХОД ИЗ ИНТЕРАКТИВНОГО СЕАНСА

Чтобы завершить базовый сеанс, нажмите Ctrl+Z (в системе Windows) или Ctrl+D (в Linux/UNIX) или введите `exit()` в командной строке.

На большинстве платформ поддерживается механизм редактирования командной строки и истории командной строки. Вы можете использовать клавиши со стрелками ↑ и ↓, а также клавиши Home, End, Page Up и Page Down для прокрутки списка старых команд; нажатие клавиши Enter повторяет ранее выполненную команду. Собственно, это все, что необходимо для работы с примерами в процессе изучения Python по этой книге. Еще один вариант — превосходный режим Python для Emacs, который среди прочего предоставляет доступ к интерактивному режиму Python через интегрированный буфер.

2.2.2. Интегрированная среда разработки IDLE

IDLE — встроенная среда разработки для Python. Название является сокращением от «*integrated development environment*», то есть «интегрированная среда разработки» (хотя, конечно, на него могла повлиять фамилия одного из участников небезызвестной комедийной британской группы[1]). IDLE объединяет интерактивный интерпретатор со средствами редактирования кода и отладки; в результате вы получаете все необходимое для создания кода Python. Благодаря разнообразию своего инструментария IDLE становится хорошей отправной точкой для изучения Python. Процедура запуска IDLE в Windows, macOS и Linux выглядит так:

- *Запуск IDLE в Windows* — для Python версии 3.6 найдите команду запуска IDLE в подменю Python 3.6 из папки Все приложения (All Apps) меню Windows и щелкните на ней. В результате открывается окно, показанное на рис. 2.2.

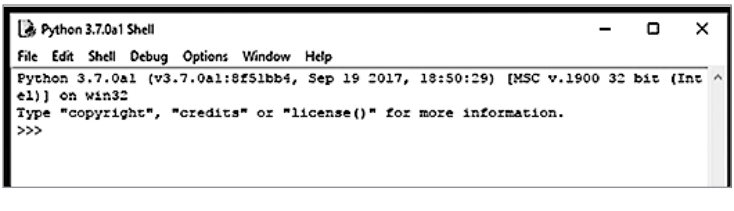


Рис. 2.2. IDLE в Windows

[1] Имеется в виду Эрик Айдл (Eric Idle) из «Монти Пайтон».

- *Запуск IDLE в macOS* — перейдите во вложенную папку Python 3.x в папке Applications и запустите IDLE.
- *Запуск IDLE в Linux или UNIX* — введите команду `idle3` в командной строке. На экране появляется окно, похожее на изображенное на рис. 2.2. Если вы установили IDLE при помощи менеджера пакетов своего дистрибутива, в подменю Programming (или что-нибудь в этом роде) должна присутствовать команда для запуска IDLE.

2.2.3. Выбор между базовым интерактивным режимом и IDLE

Что же использовать: IDLE или базовый интерактивный режим? На первых порах используйте IDLE или окно оболочки Python. Оба инструмента обладают всем необходимым для выполнения примеров кода книги до того, как мы дойдем до главы 10. С этого момента мы будем рассматривать написание собственных модулей, и IDLE предоставит удобные средства для создания и редактирования файлов. Но если вы предпочитаете другой редактор, возможно, вам хватит окна базового интерактивного режима и вашего любимого редактора. Если же у вас нет особых предпочтений, я рекомендую использовать IDLE с самого начала.

2.3. Использование окна оболочки Python в IDLE

При запуске IDLE открывается окно оболочки Python (рис. 2.3). IDLE обеспечивает автоматическую расстановку отступов и цветовое выделение синтаксиса во время редактирования кода в зависимости от типа синтаксиса Python.

Для перемещения по буферу используется мышь, клавиши управления курсором, клавиши Page Up и Page Down и/или некоторые стандартные привязки клавиш Emacs. За подробностями обращайтесь к меню Help.

Все данные вашего сеанса буферизуются. Вы можете прокручивать список или проводить поиск. Если установить курсор в любой строке и нажать Enter, эта строка будет скопирована в нижнюю часть экрана, где вы сможете отредактировать ее и передать интерпретатору повторным нажатием Enter. Или же, пока курсор остается в нижней части, вы можете перебирать ранее введенные команды комбинациями клавиш Alt+P и Alt+N; при этом внизу последовательно появляются копии строк. Обнаружив нужную строку, вы снова сможете отредактировать ее и передать интерпретатору клавишей Enter. Чтобы просмотреть список возможных вариантов завершения ключевых слов Python или значений, определенных пользователем, нажмите клавишу Tab.

Если вам покажется, что программа вроде бы зависла и не выводит новое приглашение, скорее всего, интерпретатор оказался в состоянии, когда он ждет ввода каких-то конкретных данных. Комбинация клавиш Ctrl+C прерывает программу, а на экране снова должно появиться приглашение. Она также может использоваться

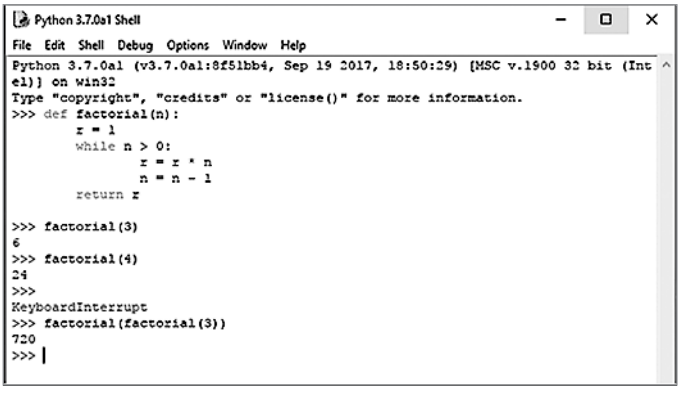


Рис. 2.3. Использование оболочки Python в IDLE. Код автоматически выделяется цветом (в соответствии с синтаксисом Python) в процессе ввода. Если подвести курсор к любой предыдущей команде и нажать Enter, команда и курсор перемещаются в нижнее поле; здесь вы можете отредактировать команду, а потом нажать Enter, чтобы передать ее интерпретатору. При размещении курсора в нижнем поле можно перемещаться по истории команд вверх и вниз клавишами Alt+P и Alt+N. Когда вы перейдете к нужной команде, отредактируйте ее так, как считаете нужным, и нажмите Enter. Команда будет передана интерпретатору

для прерывания выполняемой команды. Чтобы выйти из IDLE, выберите команду Exit из меню File.

Скорее всего, на первых порах чаще всего будет использоваться меню Edit. Как и любое другое меню, его можно отсоединить: сделайте двойной щелчок на верхней пунктирной линии и разместите его рядом с окном.

2.4. Hello, World

Каким бы способом вы ни вошли в интерактивный режим Python, вы увидите приглашение из трех угловых скобок: `>>>`. Это приглашение командной строки Python, в котором вы вводите команду для выполнения или выражение для обработки. Начнем с обязательной программы «Hello, World», которая в Python состоит из одной строки (каждая вводимая строка завершается нажатием Enter):

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

Здесь в командной строке вводится функция `print`, а результат появляется на экране. При выполнении функции `print` ее аргумент направляется в стандартный вывод — обычно на экран. Если бы команда была выполнена в то время, когда интерпретатор Python выполнял программу Python из файла, произошло бы ровно то же самое: на экран была бы выведена строка «Hello, World».

Поздравляю! Вы только что написали свою первую программу на языке Python, а ведь я еще даже не начала рассказывать об этом языке.

2.5. Использование интерактивного приглашения для изучения Python

Где бы вы ни работали, в IDLE или в стандартном интерактивном приглашении, в вашем распоряжении оказывается пара полезных инструментов, которые помогут вам в исследовании Python. Первый инструмент — функция `help()`, которая работает в двух режимах. Если ввести команду `help()` в приглашении, вы перейдете в справочный режим, в котором можно получить информацию о модулях, ключевых словах и темах. В справочном режиме выводится приглашение `help>`, а при вводе имени модуля (например, `math`) или другой темы будет выведена документация Python по данной теме.

Обычно функцию `help()` удобнее использовать целенаправленно. Если передать тип или имя переменной в параметре `help()`, вы сразу получите документацию по запрошенной теме:

```
>>> x = 2
>>> help(x)
Help on int object:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the...
(continues with the documentation for an int)
```

Этот вариант использования функции `help()` удобен для проверки синтаксиса метода или поведения объекта.

Функция `help()` входит в библиотеку `pydoc`, которая поддерживает несколько способов обращения к документации, встроенной в библиотеки Python. Так как каждая установка Python включает полную документацию, вся официальная документация всегда находится у вас под рукой даже без подключения к интернету. За дополнительной информацией о работе с документацией Python обращайтесь к приложению А.

Другая полезная функция — `dir()` — выводит список объектов в конкретном пространстве имен. Без параметров она выводит текущие глобальные переменные, но также может использоваться для вывода компонентов модуля и даже типа:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
     '__package__', '__spec__', 'x']
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
```

```
     '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__',
     '__float__', '__floor__', '__floordiv__', '__format__', '__ge__',
     '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__',
     '__init__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
     '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',
     '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__',
     '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
     '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__',
     '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__',
     '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
     '__subclasshook__', '__truediv__', '__trunc__', '__xor__',   'bit_length',
     'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real',
     'to_bytes']
>>>
```

При помощи функции `dir()` можно просмотреть набор определенных методов и данных, а также припомнить все компоненты, принадлежащие объекту или модулю. Наконец, функция пригодится в ходе отладки, потому что вы видите, что где определяется.

В отличие от `dir`, функции `globals` и `locals` выводят значения, связанные с объектами. В текущей ситуации обе функции возвращают одно и то же, поэтому ниже приводится вывод только для `globals()`:

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
     <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
     '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
     'x': 2}
```

Обе функции будут более подробно рассмотрены в главе 10, а пока достаточно знать, что вы можете несколькими способами просмотреть информацию о текущем состоянии сеанса Python.

Итоги

- Чтобы установить Python 3 в системе Windows, достаточно загрузить новейшую программу установки с сайта www.python.org и запустить ее. Процедура установки в Linux, UNIX и Mac зависит от конкретной системы.
- Инструкции по установке доступны на сайте Python. Там, где это возможно, используйте систему установки пакетов.
- Другой способ установки заключается в установке дистрибутива Anaconda (или miniconda) с сайта https://www.anaconda.com/download/.
- После установки Python можно использовать базовую интерактивную оболочку (а позднее ваш любимый редактор) или интегрированную среду разработки IDLE.

3 Краткий обзор Python

Эта глава охватывает следующие темы:

- ✓ Общее описание Python
- ✓ Использование встроенных типов данных
- ✓ Управляющие конструкции
- ✓ Создание модулей
- ✓ Использование объектно-ориентированного программирования

Эта глава дает первое представление о синтаксисе, семантике, возможностях и философии языка Python. Она была написана для того, чтобы создать у читателя исходную перспективу или концептуальную основу, которая будет постепенно обрастать подробностями по мере того, как вы будете знакомиться с ними в других главах книги.

При первом чтении вам не нужно досконально разбирать фрагменты кода во всех подробностях. Достаточно получить хотя бы общее представление о происходящем. В последующих главах многие аспекты языка будут рассмотрены более подробно, при этом никакие предыдущие познания от вас не потребуются. Вы всегда можете вернуться к этой главе и просмотреть примеры в соответствующих разделах, чтобы освежить память после того, как прочтете следующие главы.

3.1. Общее описание Python

Python содержит ряд встроенных типов данных: целые числа, числа с плавающей точкой, комплексные числа, строки, списки, кортежи, словари, объекты файлов

и т. д. Для работы с этими типами данных используются операторы языка, встроенные функции, библиотечные функции и собственные методы типа данных.

Программисты также могут определять собственные классы и создавать экземпляры[1] этих классов. Для работы с экземплярами классов используются методы, определенные программистом, а также операторы языка и встроенные функции, для которых программист определил соответствующие атрибуты методов.

Python поддерживает условные и циклические управляющие конструкции в форме команд `if-elif-else`, циклов `while` и `for`. Это позволяет определять функции с гибкими схемами передачи аргументов. Исключения (ошибки) инициируются командой `raise`, а для их перехвата и обработки используется конструкция `try-except-else-finally`.

Переменные (или идентификаторы) объявлять не нужно. Они могут ссылаться на любой встроенный тип данных, пользовательский объект, функцию или модуль.

3.2. Встроенные типы данных

В Python поддерживаются различные встроенные типы данных, от скалярных (например, числа и логические значения) до более сложных структур, таких как списки, словари и файлы.

3.2.1. Числовые типы

Четыре числовых типа Python — целые числа, числа с плавающей запятой (с плавающей точкой), комплексные числа и логические значения:

- целые числа: 1, –3, 42, 355, 888888888888888, –7777777777 (размер целых чисел ограничивается только объемом доступной памяти);
- числа с плавающей точкой: 3,0, 31e12, –6e-4;
- комплексные числа: 3 + 2j, –4- 2j, 4,2 + 6,3j;
- логические значения: True, False.

Для работы с числами используются арифметические операторы: `+` (сложение), `–` (вычитание), `*` (умножение), `/` (деление), `**` (возведение в степень) и `%` (остаток от деления.)

[1] В документации Python и в этой книге термин «объект» используется для обозначения экземпляра любого типа данных Python, а не только того, что во многих других языках называется *экземпляром класса*. Дело в том, что любой объект Python является экземпляром того или иного класса.

В следующих примерах используются целые числа:

```
>>> x = 5 + 2 - 3 * 2
>>> x
1
>>> 5 / 2
2.5 ❶
>>> 5 // 2
2 ❷
>>> 5 % 2
1
>>> 2 ** 8
256
>>> 1000000001 ** 3
1000000003000000003000000001 ❸
```

При делении целых чисел оператором `/` ❶ будет получен результат с плавающей точкой (новое поведение Python 3.*x*), а при делении целых чисел оператором `//` ❷ происходит отсечение дробной части. Следует заметить, что целые числа имеют неограниченный размер ❸, они увеличиваются по мере необходимости, а их размер ограничивается только объемом доступной памяти.

В следующей группе примеров используются числа с плавающей точкой, основанные на вещественных числах двойной точности языка C:

```
>>> x = 4.3 ** 2.4
>>> x
33.13784737771648
>>> 3.5e30 * 2.77e45
9.695e+75
>>> 1000000001.0 ** 3
1.000000003e+27
```

Примеры с комплексными числами:

```
>>> (3+2j) ** (2+3j)
(0.6817665190890336-2.1207457766159625j)
>>> x = (3+2j) * (4+9j)
>>> x ❶
(-6+35j)
>>> x.real
-6.0
>>> x.imag
35.0
```

Комплексные числа состоят из двух частей, вещественной и мнимой (снабженной суффиксом `j`). В приведенном фрагменте переменной `x` присваивается комплексное число ❶. Для получения вещественной части используется синтаксис атрибута `x.real`, а для получения мнимой части — `x.imag`.

Для работы с числовыми типами могут использоваться некоторые встроенные функции. Также в вашем распоряжении библиотечный модуль `cmath` (функции для работы с комплексными числами) и библиотечный модуль `math` (функции для трех других типов):

```
>>> round(3.49)  ❶
3
>>> import math
>>> math.ceil(3.49)  ❷
4
```

Встроенные функции доступны всегда, а для их вызова используется стандартный синтаксис вызова функций. В предшествующем коде функция `round` вызывается с передачей аргумента с плавающей точкой ❶.

Для получения доступа к функциям библиотечных модулей используется команда `import`. В точке ❷ импортируется библиотечный модуль `math`, а его функция `ceil` вызывается с использованием синтаксиса атрибута: `модуль.функция(аргументы)`.

В следующих примерах используются логические значения:

```
>>> x = False
>>> x
False
>>> not x
True
>>> y = True * 2  ❶
>>> y
2
```

Если не считать представления в виде `True` и `False`, логические значения ведут себя как числа 1 (`True`) и 0 (`False`) ❶.

3.2.2. Списки

В Python реализован мощный встроенный тип, представляющий списки:

```
[]
[1]
[1, 2, 3, 4, 5, 6, 7, 8, 12]
[1, "two", 3, 4.0, ["a", "b"], (5,6)]  ❶
```

Элементами списка могут быть другие типы в произвольном сочетании: строки, кортежи, списки, словари, функции, объекты файлов и любые числовые типы ❶.

Список может индексироваться как от начала, так и от конца. Также из списка можно выделить сегмент, или *срез*, с использованием следующего синтаксиса:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[0]
'first'                    ❶
>>> x[2]
'third'
>>> x[-1]
'fourth'
>>> x[-2]                  ❷
'third'
>>> x[1:-1]
['second', 'third']
>>> x[0:3]
['first', 'second', 'third']                    ❸
>>> x[-2:-1]
['third']
>>> x[:3]
['first', 'second', 'third']
>>> x[-2:]                 ❹
['third', 'fourth']
```

Индексирование от начала списка ❶ использует положительные значения (0 соответствует первому элементу). Индексирование от конца списка ❷ использует отрицательные индексы (–1 соответствует последнему элементу). Сегмент создается записью вида `[m:n]` ❸, где `m` — индекс начального элемента (включительно), а `n` — индекс конечного элемента (не включая его) (табл. 3.1). Сегмент `[:n]` ❹ начинается от начала списка, а сегмент `[m:]` продолжается до конца списка.

Таблица 3.1. Индексы в списках

x=	[	"first" ,	"second" ,	"third" ,	"fourth"	]
Положительные индексы		0	1	2	3	
Отрицательные индексы		–4	–3	–2	–1	

Эта форма записи может использоваться для добавления, удаления и замены элементов списка, а также для получения отдельных элементов или новых списков, которые представляют собой сегменты существующих списков:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[1] = "two"
>>> x[8:9] = []
>>> x
[1, 'two', 3, 4, 5, 6, 7, 8]
>>> x[5:7] = [6.0, 6.5, 7.0] ❶
>>> x
```

```
[1, 'two', 3, 4, 5, 6.0, 6.5, 7.0, 8]
>>> x[5:]
[6.0, 6.5, 7.0, 8]
```

Если новый сегмент больше или меньше заменяемого ❶, то размер списка увеличивается или уменьшается.

Также для работы со списками используются некоторые встроенные функции (`len`, `max` и `min`), некоторые операторы (`in`, `+` и `*`), команда `del` и методы списков (`append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse` и `sort`):

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(x)
9
>>> [-1, 0] + x ❶
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x.reverse() ❷
>>> x
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Каждый из операторов `+` и `*` создает новый список, оставляя исходный список без изменений ❶. Для вызова методов списка используется синтаксис атрибутов, применяемый к самому списку: `x.метод(аргументы)` ❷.

Некоторые из этих операций повторяют функциональность, которая может быть реализована в синтаксисе сегментов, но при этом код становится более понятным.

3.2.3. Кортежи

Кортежи (tuples) похожи на списки, но они являются *неизменяемыми* — другими словами, эти объекты невозможно изменить после создания. Операторы (`in`, `+` и `*`) и встроенные функции (`len`, `max` и `min`) работают с кортежами так же, как со списками, потому что они не изменяют оригинал. Индексы и синтаксис сегментов работают аналогично для получения элементов или сегментов, но не могут использоваться для добавления, удаления или замены элементов. У кортежей всего два метода: `count` и `index`. Одна из важных областей применения кортежей — использование в качестве ключей в словарях. Также кортежи более эффективно работают в тех ситуациях, в которых изменяемость не нужна.

```
()
(1,) ❶
(1, 2, 3, 4, 5, 6, 7, 8, 12)
(1, "two", 3L, 4.0, ["a", "b"], (5, 6)) ❷
```

Кортеж из одного элемента ❶ должен содержать запятую. Кортеж, как и список, может содержать произвольные комбинации других типов в своих элементах: строки, кортежи, списки, словари, функции, объекты файлов и любые разновидности чисел ❷.

Список преобразуется в кортеж при помощи встроенной функции `tuple`:

```
>>> x = [1, 2, 3, 4]
>>> tuple(x)
(1, 2, 3, 4)
```

И наоборот, кортеж преобразуется в список при помощи встроенной функции `list`:

```
>>> x = (1, 2, 3, 4)
>>> list(x)
[1, 2, 3, 4]
```

3.2.4. Строки

Работа со строками — одна из сильных сторон Python. Существует много способов определения строк с разными ограничителями:

```
"A string in double quotes can contain 'single quote' characters."
'A string in single quotes can contain "double quote" characters.'
'''\tA string which starts with a tab; ends with a newline character.\n'''
"""This is a triple double quoted string, the only kind that can
    contain real newlines."""
```

Строки могут ограничиваться одинарными (`' '`), двойными (`" "`), утроенными одинарными (`''' '''`) или утроенными двойными (`""" """`) кавычками; они могут содержать символы табуляции (`\t`) и символы новой строки (`\n`).

Строки также являются неизменяемыми. Операторы и функции, которые работают с ними, возвращают новые строки, полученные на основе оригинала. Операторы (`in`, `+` и `*`) и встроенные функции (`len`, `max` и `min`) работают со строками так же, как они работают со списками и кортежами. Синтаксис индексирования и сегментирования работает аналогичным образом для получения элементов и сегментов, но он не может использоваться для добавления, удаления или замены элементов.

Строки поддерживают ряд методов для работы с их содержимым; библиотечный модуль `re` также содержит функции для работы со строками:

```
>>> x = "live and     let \t   \tlive"
>>> x.split()
['live', 'and', 'let', 'live']
>>> x.replace("    let \t   \tlive", "enjoy life")
'live and enjoy life'
>>> import re ❶
>>> regexpr = re.compile(r"[\t ]+")
>>> regexpr.sub(" ", x)
'live and let live'
```

Модуль `re` ❶ предоставляет функциональность регулярных выражений — более сложные и мощные средства поиска по шаблону и замены по сравнению с модулем `string`.

Функция `print` выводит строки. Другие типы данных Python легко преобразуются в строки и форматируются функцией `print`:

```
>>> e = 2.718
>>> x = [1, "two", 3, 4.0, ["a", "b"], (5, 6)]
>>> print("The constant e is:", e, "and the list x is:", x) ❶
The constant e is: 2.718 and the list x is: [1, 'two', 3, 4.0,
['a', 'b'], (5, 6)]
>>> print("the value of %s is: %.2f" % ("e", e)) ❷
the value of e is: 2.72
```

Объекты автоматически преобразуются в строковое представление для вывода ❶. Оператор `%` ❷ предоставляет возможности форматирования, сходные с возможностями функции `sprintf` языка C.

3.2.5. Словари

Встроенный тип данных словаря (dictionary) в языке Python предоставляет функциональность ассоциативных массивов, реализованную на базе хеш-таблиц. Встроенная функция `len` возвращает количество пар «ключ–значение» в словаре. Команда `del` используется для удаления пары «ключ–значение». Как и в случае со списками, доступны различные методы для выполнения операций со словарями (`clear`, `copy`, `get`, `items`, `keys`, `update` и `values`).

```
>>> x = {1: "one", 2: "two"}
>>> x["first"] = "one"  ◄——  Связывает с ключом «first» значение «one»
>>> x[("Delorme", "Ryan", 1995)] = (1, 2, 3) ❶
>>> list(x.keys())
['first', 2, 1, ('Delorme', 'Ryan', 1995)]
>>> x[1]
'one'
>>> x.get(1, "not available")
'one'
>>> x.get(4, "not available") ❷
'not available'
```

Ключи должны относиться к неизменяемому типу ❶ — числа, строки, кортежи и т. д. Значениями могут быть объекты любого типа, включая такие изменяемые типы, как списки и словари. При попытке обратиться к значению ключа, отсутствующего в словаре, произойдет ошибка `KeyError`. Чтобы избежать этой ошибки, при отсутствии ключа в словаре метод словаря `get` ❷ может возвращать значение, определяемое пользователем.

3.2.6. Множества

Множество (set) в Python представляет собой неупорядоченный набор объектов, используемый в ситуациях, когда вас интересует лишь сам факт принадлежности объекта к множеству и уникальность в множестве. Множество ведет себя как коллекция ключей словаря без ассоциированных значений:

```
>>> x = set([1, 2, 3, 1, 3, 5]) ❶
>>> x
{1, 2, 3, 5} ❷
>>> 1 in x
True                ❸
>>> 4 in x
False
>>>
```

Множество создается вызовом `set` для последовательности — например, для списка ❶. При преобразовании последовательности в множество дубликаты удаляются ❷. Ключевое слово `in` ❸ используется для проверки принадлежности объекта к множеству.

3.2.7. Объекты файлов

Для работы с файлами в Python используются объекты файлов:

```
>>> f = open("myfile", "w") ❶
>>> f.write("First line with necessary newline character\n")
44
>>> f.write("Second line to write to the file\n")
33
>>> f.close()
>>> f = open("myfile", "r") ❷
>>> line1 = f.readline()
>>> line2 = f.readline()
>>> f.close()
>>> print(line1, line2)
First line with necessary newline character
Second line to write to the file
>>> import os ❸
>>> print(os.getcwd())
c:\My Documents\test
>>> os.chdir(os.path.join("c:\\", "My Documents", "images")) ❹
>>> filename = os.path.join("c:\\", "My Documents",
"test", "myfile") ❺
>>> print(filename)
c:\My Documents\test\myfile
>>> f = open(filename, "r")
>>> print(f.readline())
First line with necessary newline character
>>> f.close()
```

Команда `open` ❶ создает объект файла. В данном случае файл myfile в текущем рабочем каталоге открывается в режиме записи (`"w"`). После записи двух строк в файл и его закрытия ❷ файл открывается снова, на этот раз в режиме для чтения (`"r"`). Модуль `os` ❸ предоставляет несколько функций для перемещения по файловой системе и работы с именами файлов и каталогов. В данном примере происходит перемещение в другой каталог ❹. Тем не менее к файлу все равно можно обратиться по абсолютному имени ❺.

Также в Python доступны другие средства ввода/вывода. Например, встроенная функция `input` запрашивает и вводит строку. Библиотечный модуль `sys` открывает доступ к потокам `stdin`, `stdout` и `stderr`. Библиотечный модуль `struct` предоставляет поддержку чтения и записи файлов, которые генерируются (или должны использоваться) программами на C. Библиотечный модуль Pickle обеспечивает возможность долгосрочного хранения данных посредством простой записи и чтения из файлов типов данных Python.

3.3. Управляющие конструкции

Python поддерживает обширный набор конструкций для управления выполнением кода, к числу которых относятся стандартные структуры условного выбора и циклов.

3.3.1. Логические значения и выражения

В Python предусмотрено несколько возможных способов выражения логических значений; логическая константа `False`, `0`, неопределенное значение Python `None` и пустые значения (например, пустой список `[]` или пустая строка `""`) — все эти значения интерпретируются как `False`. Логическая константа `True` и все остальные значения интерпретируются как `True`.

Для создания логических условий используются операторы сравнения (`<`, `<=`, `==`, `>`, `>=`, `!=`, `is`, `is not`, `in`, `not in`) и логические операторы (`and`, `not`, `or`); все они возвращают либо `True`, либо `False`.

3.3.2. Команда if-elif-else

Выполняется блок кода после первого истинного условия (в `if` или `elif`). Если ни одно условие не равно `True`, то выполняется блок кода после `else`:

```
x = 5
if x < 5:
    y = -1
    z = 5
elif x > 5:
    y = 1
    z = 11
else:
    y = 0
    z = 10
print(x, y, z)
```

❶

❷

Присутствие секций `elif` и `else` не обязательно ❶, а количество секций `elif` не ограничено. Для ограничения блоков используются отступы ❷. Включение явных ограничителей (таких, как квадратные или фигурные скобки) не обязательно. Каждый блок состоит из одной или нескольких команд, разделенных символами

новой строки. Все эти команды должны снабжаться отступами одного уровня. В приведенном примере будет выведен результат `5 0 10`.

3.3.3. Цикл while

Цикл `while` продолжает выполняться, пока условие (в следующем примере `x > y`) остается истинным (`True`):

```
u, v, x, y = 0, 0, 100, 30 ❶
while x > y:                 |
    u = u + y                |
    x = x - y                |
    if x < y + 2:            |
        v = v + x            | ❷
        x = 0                |
    else:                    |
        v = v + y + 2        |
        x = x - y - 2        |
print(u, v)
```

В первой строке используется сокращенный синтаксис присваивания. Здесь `u` и `v` присваивается значение 0, `x` присваивается 100, а `y` присваивается значение 30 ❶. Затем идет блок цикла ❷. Цикл может содержать команды `break` (прерывание цикла) и `continue` (отмена текущей итерации цикла). Пример выводит результат `60 40`.

3.3.4. Цикл for

Цикл `for` — простая, но мощная конструкция для перебора любого итерируемого типа (например, списка или кортежа). В отличие от многих языков, цикл `for` в Python перебирает элементы последовательности (например, списка или кортежа), так что он больше напоминает циклы `foreach`. Следующий цикл находит в списке первое вхождение целого числа, кратного 7:

```
item_list = [3, "string1", 23, 14.0, "string2", 49, 64, 70]
for x in item_list: ❶
    if not isinstance(x, int):
        continue ❷
    if not x % 7:
        print("found an integer divisible by seven: %d" % x)
        break ❸
```

Переменной `x` последовательно присваивается каждое значение из списка ❶. Если `x` не является целым числом, то оставшаяся часть итерации отменяется командой `continue` ❷. Выполнение цикла продолжается присваиванием `x` следующего элемента списка. После того как будет найдено подходящее целое число, цикл завершается командой `break` ❸. Программа выводит

```
целое число, кратное 7: 49
```

3.3.5. Определение функции

Python поддерживает гибкий механизм передачи аргументов функциям:

```
>>> def funct1(x, y, z): ❶
...     value = x + 2*y + z**2
...     if value > 0:
...         return x + 2*y + z**2 ❷
...     else:
...         return 0
...
>>> u, v = 3, 4
>>> funct1(u, v, 2)
15
>>> funct1(u, z=v, y=2) ❸
23
>>> def funct2(x, y=1, z=1): ❹
...     return x + 2 * y + z ** 2
...
>>> funct2(3, z=4)
21
>>> def funct3(x, y=1, z=1, *tup): ❺
...     print((x, y, z) + tup)
...
>>> funct3(2)
(2, 1, 1)
>>> funct3(1, 2, 3, 4, 5, 6, 7, 8, 9)
(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> def funct4(x, y=1, z=1, **kwargs): ❻
...     print(x, y, z, kwargs)
>>> funct4(1, 2, m=5, n=9, z=3)
1 2 3 {'n': 9, 'm': 5}
```

Функции определяются командой `def` ❶. Команда `return` ❷ определяет значение, возвращаемое функцией. Это значение может относиться к любому типу. Если команда `return` не была обнаружена в ходе выполнения функции, возвращается значение Python `None`. Аргументы функции могут передаваться по позиции или по имени (ключевому слову). В данном примере аргументы `z` и `y` передаются по имени ❸. Для параметров функции могут определяться значения по умолчанию, которые будет использованы в том случае, если значение не указано при вызове функции ❹. Также можно определить специальный параметр, который объединяет все лишние позиционные аргументы при вызове функции в кортеж ❺. Аналогичным образом возможно определить специальный параметр, который объединяет все лишние именованные аргументы, указанные при вызове функции, в словарь ❻.

3.3.6. Исключения

Исключения (ошибки) перехватываются и обрабатываются сложной командой `try-except-else-finally`. Эта команда также может перехватывать и обрабатывать

исключения, которые вы определяете и инициируете самостоятельно. Любое неперехваченное исключение приводит к выходу из программы. В листинге 3.1 продемонстрирована базовая обработка исключений.

Листинг 3.1. Файл exception.py

```
class EmptyFileError(Exception): ❶
    pass
filenames = ["myfile1", "nonExistent", "emptyFile", "myfile2"]
for file in filenames:
    try: ❷
        f = open(file, 'r')
        line = f.readline() ❸
        if line == "":
            f.close()
            raise EmptyFileError("%s: is empty" % file) ❹
    except IOError as error:
        print("%s: could not be opened: %s" % (file, error.strerror)
    except EmptyFileError as error:
        print(error)
    else: ❺
        print("%s: %s" % (file, f.readline()))
    finally:
        print("Done processing", file) ❻
```

Здесь мы определяем собственный тип исключения, наследующий от базового типа `Exception` ❶. Если исключение `IOError` или `EmptyFileError` произойдет во время выполнения команд в блоке `try`, выполняется соответствующий блок `except` ❷. Исключение `IOError` может инициироваться в точке ❸, а исключение `EmptyFileError` — в точке ❹. Секция `else` не является обязательной ❺, она выполняется в том случае, если выполнение блока `try` обошлось без исключений (кстати, в этом примере в блоках `except` можно использовать команды `continue`). Секция `finally` также не обязательна ❻, она будет выполнена в конце блока независимо от того, было выдано исключение или нет.

3.3.7. Обработка контекста с ключевым словом with

Более элегантный способ реализации паттерна `try-except-finally` основан на использовании ключевого слова `with` и менеджера контекста. Python определяет менеджеров контекста для таких операций, как работа с файлами, но разработчик может определять собственных менеджеров контекста.

Одно из преимуществ менеджеров контекста заключается в том, что они могут определять (и обычно определяют) завершающие действия по умолчанию, которые выполняются всегда независимо от того, происходило исключение или нет.

В листинге 3.2 показано открытие и чтение файла с использованием `with` и менеджера контекста.

Листинг 3.2. Файл with.py

```
filename = "myfile.txt"
with open(filename, "r") as f:
    for line in f:
        print(f)
```

Здесь ключевое слово `with` создает менеджера контекста, который инкапсулирует функцию `open` и следующий за ней блок. В данном случае заранее определенное завершающее действие менеджера контекста закроет файл, даже если произошло исключение, поэтому при условии, что выражение в первой строке будет выполнено без исключения, файл всегда будет закрыт. Этот код эквивалентен следующему:

```
filename = "myfile.txt"
try:
    f = open(filename, "r")
    for line in f:
        print(f)
except Exception as e:
    raise e
finally:
    f.close()
```

3.4. Создание модуля

Вы можете легко создавать собственные модули, которые импортируются и используются точно так же, как модули встроенных библиотек Python. В листинге 3.3 создается простой модуль с одной функцией, которая предлагает пользователю ввести имя файла и подсчитывает количество вхождений слов в этом файле.

Листинг 3.3. Файл wo.py

```
"""Модуль wo. Содержит функцию: words_occur()""" ❶
# Функции интерфейса ❷
def words_occur():
    """words_occur() - подсчитывает вхождения слов в файле."""
    # Запросить у пользователя имя файла.
    file_name = input("Enter the name of the file: ")
    # Открыть файл, прочитать его и сохранить слова в списке.
    f = open(file_name, 'r')
    word_list = f.read().split() ❸
    f.close()
    # Подсчитать количество вхождений каждого слова в файле.
    occurs_dict = {}
    for word in word_list:
        # Увеличить счетчик для данного слова.
        occurs_dict[word] = occurs_dict.get(word, 0) + 1
    # Вывести результаты.
    print("File %s has %d words (%d are unique)" \ ❹
      % (file_name, len(word_list), len(occurs_dict)))
    print(occurs_dict)
if __name__ == '__main__': ❺
    words_occur()
```

Строки документации — стандартный способ документирования модулей, функций, методов и классов ❶. Комментарий состоит из всех символов, начинающихся с # ❷. Функция `read` возвращает строку, содержащую все символы в файле ❸, а функция `split` возвращает список слов строки, «разбитой» по символам-пропускам. Символ \ позволяет продолжить длинную команду в нескольких строках программы ❹. Эта команда `if` позволяет выполнить программу как сценарий, для чего следует ввести команду `python wo.py` в командной строке ❺.

Если разместить файл в одном из каталогов, входящих в путь поиска модулей из переменной `sys.path`, его можно будет импортировать командой `import` точно так же, как любой встроенный библиотечный модуль:

```
>>> import wo
>>> wo.words_occur() ❶
```

Функция вызывается ❶ с тем же синтаксисом атрибутов, который используется для функций библиотечных модулей.

Обратите внимание: если изменить файл wo.py на диске, команда `import` не отразит эти изменения в том же интерактивном сеансе. В таких ситуациях можно воспользоваться функцией `reload` из библиотеки `imp`:

```
>>> import imp
>>> imp.reload(wo)
<module 'wo'>
```

Для больших проектов существует обобщение концепции модуля — *пакеты* позволяют легко группировать модули в каталоге или дереве подкаталогов, чтобы затем импортировать и использовать для обращения к ним иерархические ссылки вида `пакет.субпакет.модуль`. В сущности, для этого потребуется лишь создать (возможно, пустой) инициализационный файл для каждого пакета или субпакета.

3.5. Объектно-ориентированное программирование

В Python реализована полноценная поддержка ООП. В листинге 3.4 показана заготовка простого модуля для графического редактора. Листинг всего лишь дает представление о возможностях Python для читателей, уже знакомых с ООП. Выноски поясняют связь синтаксиса и семантики Python со стандартными возможностями, присутствующими в других языках.

Листинг 3.4. Файл sh.py

```
"""Модуль sh. Содержит классы Shape, Square и Circle"""
class Shape: ❶
    """Класс Shape: содержит метод move"""
    def __init__(self, x, y): ❷
        self.x = x
        self.y = y          ❸
    def move(self, deltaX, deltaY): ❹
        self.x = self.x + deltaX
```

```
        self.y = self.y + deltaY
class Square(Shape):
    """Класс Square: наследует от Shape"""
    def __init__(self, side=1, x=0, y=0):
        Shape.__init__(self, x, y)
        self.side = side
class Circle(Shape): ❺
    """Класс Circle: наследует от Shape и содержит метод area"""
    pi = 3.14159 ❻
    def __init__(self, r=1, x=0, y=0):
        Shape.__init__(self, x, y) ❼
        self.radius = r
    def area(self):
        """метод area класса Circle: возвращает площадь круга."""
        return self.radius * self.radius * self.pi
    def __str__(self): ❽
        return "Circle of radius %s at coordinates (%d, %d)"\
               % (self.radius, self.x, self.y)
```

Классы определяются ключевым словом `class` ❶. Метод-инициализатор экземпляра (конструктор) класса всегда называется `__init__` ❷. Здесь создаются и инициализируются переменные экземпляров `x` и `y` ❸. Методы, как и функции, определяются ключевым словом `def` ❹. Первый аргумент любого метода по соглашению называется `self`. При вызове метода `self` присваивается ссылка на экземпляр, для которого был вызван метод. Класс `Circle` наследует от класса `Shape` ❺, а в точке ❻ определяется переменная класса. Класс в своем инициализаторе должен явно вызвать инициализацию базового класса ❼. Метод `__str__` используется функцией `print` ❽. Другие специальные атрибуты методов обеспечивают перегрузку операторов или используются встроенными методами, такими как функция вычисления длины (`len`).

Импортирование этого файла открывает доступ к этим классам:

```
>>> import sh
>>> c1 = sh.Circle() ❶
>>> c2 = sh.Circle(5, 15, 20)
>>> print(c1)
Circle of radius 1 at coordinates (0, 0)
>>> print(c2) ❷
Circle of radius 5 at coordinates (15, 20)
>>> c2.area()
78.539749999999998
>>> c2.move(5,6) ❸
>>> print(c2)
Circle of radius 5 at coordinates (20, 26)
```

Инициализатор вызывается неявно, а в программе создается экземпляр круга ❶. Функция `print` неявно использует специальный метод `__str__` ❷. Как видно из листинга, в программе доступен метод `move` класса `Shape` (родительского по отношению к `Circle`) ❸. Метод вызывается применением синтаксиса атрибутов к экземпляру объекта: `объект.метод()`. Значение первого параметра (`self`) задается неявно.

Итоги

- В этой главе приведен краткий и чрезвычайно общий обзор Python, в следующих главах эти вопросы будут рассмотрены более подробно. Эта глава завершает общее описание Python.
- Возможно, вам будет полезно вернуться к этой главе и еще раз рассмотреть подходящие примеры после того, как вы прочитаете изложение соответствующих тем в последующих главах.
- Если вы читали эту главу как обзор или вам хотелось бы узнать больше о некоторых возможностях Python, перейдите к соответствующим главам.
- Прежде чем переходить к части 4, необходимо четко понимать возможности Python, представленные в этой главе.

ЧАСТЬ 2

Основной материал

В следующих главах изложены основные темы, относящиеся к языку Python. Мы начнем с азов построения программ Python, а потом перейдем к встроенным типам данных и управляющим структурам, а также определению функций и использованию модулей.

В последней главе этой части я покажу, как писать автономные программы Python, работать с файлами, обрабатывать ошибки и пользоваться классами.

4 Основы

Эта глава охватывает следующие темы:

- ✓ Отступы и структурирование блоков
- ✓ Дифференцирование комментариев
- ✓ Назначение переменных
- ✓ Оценка выражений
- ✓ Использование общих типов данных
- ✓ Получение пользовательского ввода
- ✓ Использование правильного питонического стиля

В этой главе описаны фундаментальные концепции Python: вы узнаете, как использовать присваивание и выражения, как ввести число или строку, как определить комментарии в коде и т. д. Глава начинается с пояснения способа оформления программных блоков Python, который отличается от всех популярных языков.

4.1. Отступы и оформление блоков

Python отличается от многих других языков программирования тем, что он использует символы-пропуски (whitespace) и отступы для определения структуры блоков (то есть для определения того, какой код образует тело цикла, секцию `else` условной конструкции и т. д.). В большинстве языков для этой цели используются фигурные скобки. Приведенный ниже код на C вычисляет факториал 9 и сохраняет результат в переменной `r`:

```
/* Код на языке C */
int n, r;
n = 9;
r = 1;
while (n > 0) {
```

```
    r *= n;
    n--;
}
```

Фигурные скобки ограничивают тело цикла `while`, то есть код, выполняемый с каждым повторением цикла. Обычно в коде расставляются отступы большего или меньшего размера, которые четко показывают, что происходит в программе, хотя код также можно записать в следующем виде:

```
/* А это код C с произвольными отступами */
    int n, r;
        n = 9;
        r = 1;
    while (n > 0) {
r *=  n;
n--;
}
```

Такой код будет работать правильно, хотя читать его будет намного труднее. Эквивалентный код на Python выглядит так:

```
# Код на Python.
n = 9
r = 1
while n > 0:
    r = r * n  <-- Python также поддерживает конструкции в стиле C r *= n
    n = n - 1  <-- Python также поддерживает n -= 1
```

Python не использует фигурные скобки для обозначения структуры кода, вместо них используются сами отступы. Последние две строки предыдущего фрагмента являются телом цикла `while`, потому что они следуют сразу же за командой `while`, а их отступ на один уровень больше отступа команды `while`. Если бы эти строки не имели отступов, то они не считались бы частью тела `while`.

Возможно, вы не сразу привыкнете к структурированию кода с использованием отступов вместо фигурных скобок, но у этого способа есть значительные преимущества:

- Невозможно пропустить или поставить лишнюю фигурную скобку. Вам не придется шарить по своему коду и разыскивать завершающую фигурную скобку, которая соответствует открывающей фигурной скобке в самом начале файла.
- Визуальная структура кода отражает его реальную структуру, что позволяет с первого взгляда составить представление о его строении.
- Стиль оформления кода в Python становится более или менее единым. Иначе говоря, чужие представления об эстетически приятном коде вряд ли вызовут у вас особое раздражение. Код других разработчиков будет более или менее похож на ваш.

Возможно, вы уже используете постоянную схему расстановки отступов в вашем коде, так что это вряд ли станет большим изменением для вас. Если вы работаете

в IDLE, отступы будут расставляться автоматически. При желании вы всегда можете сократить уровень отступов нажатием клавиши Backspace. Многие редакторы для программистов и интегрированные среды, например Emacs, VIM и Eclipse, также предоставляют эту функциональность. Одна из вещей, на которых вы споткнетесь раз-другой, прежде чем привыкнете, заключается в том, что интерпретатор Python возвращает сообщение об ошибке, если перед командами, вводимыми в приглашении, стоит пробел (или несколько пробелов).

4.2. Комментарии

Как правило, все символы, следующие за знаком # в файле Python, образуют комментарий и игнорируются языком. Очевидным исключением из правила является символ # в строке, который просто присутствует в тексте:

```
# Присвоить x значение 5.
x = 5
x = 3                    # Теперь переменная x содержит 3.
x = "# This is not a comment"
```

Комментарии достаточно часто встречаются в коде Python.

4.3. Переменные и присваивание

Пожалуй, самая распространенная команда языка Python — присваивание — очень похожа на конструкции присваивания, которые вы использовали в других языках. Код Python для создания переменной с именем x и присваивания ей значения 5 выглядит так:

```
x = 5
```

В Python, в отличие от многих других компьютерных языков, не нужно ни объявлять тип переменной, ни включать ограничитель конца строки. Строка программы завершается там, где она завершается. Переменные создаются автоматически при первом присваивании.

ПЕРЕМЕННЫЕ В PYTHON: ВЕДРА ИЛИ ЯРЛЫКИ?

Термин «переменная» в Python несколько неточен; точнее было бы назвать переменную «*именем*» или «*ярлыком*». Однако похоже, все программисты в какой-то момент все равно называют переменные *переменными*. Как бы ни назывались переменные, вы должны знать, как они работают в Python.

Распространенное, хотя и неточное объяснение — что переменная является контейнером для хранения значения, чем-то вроде ведра, в котором лежит значение. Такая аналогия справедлива для многих языков программирования (например, C).

С другой стороны, переменные Python на ведра не похожи. Они ближе к биркам или ярлыкам, которые ссылаются на объекты из пространства имен интерпретатора Python. На один объект может ссылаться любое количество ярлыков (или переменных), и при изменении этого объекта значение, на которое ссылаются *все* эти переменные, тоже изменяется.

Чтобы понять, что это означает, рассмотрим следующий простой фрагмент:

```
>>> a = [1, 2, 3]
>>> b = a
>>> c = b
>>> b[1] = 5
>>> print(a, b, c)
[1, 5, 3] [1, 5, 3] [1, 5, 3]
```

Если рассматривать переменные как контейнеры, такой результат выглядит странно. Как изменение содержимого одного контейнера приводит к одновременному изменению содержимого двух других? Но если понимать, что переменные — всего лишь ярлыки, ссылающиеся на объекты, вполне логично, что изменение объекта, к которому относятся все три ярлыка, отразится сразу во всех местах.

Если переменные ссылаются на константы или неизменяемые значения, различие уже не столь очевидно:

```
>>> a = 1
>>> b = a
>>> c = b
>>> b = 5
>>> print(a, b, c)
1 5 1
```

Так как объекты, на которые они ссылаются, изменяться не могут, поведение переменных в данном случае соответствует любой аналогии. Собственно, после третьей строки a, b и c ссылаются на один неизменяемый объект целого числа со значением 1. Следующая строка b = 5 заставляет b ссылаться на объект целого числа 5, но не изменяет ссылки a или c.

Переменным Python могут присваиваться любые объекты, тогда как в C и многих других языках переменная может хранить значения только того типа, с которым она была объявлена. Ниже приведен абсолютно нормальный код Python:

```
>>> x = "Hello"
>>> print(x)
Hello
>>> x = 5
>>> print(x)
5
```

Переменная `x` сначала ссылается на строковый объект `"Hello"`, а потом на объект целого числа `5`. Конечно, и этой возможностью можно злоупотреблять, потому что произвольное присваивание одному имени переменной разных типов данных усложняет понимание кода.

Новое присваивание переопределяет все предыдущие. Команда `del` удаляет переменную. При попытке вывести содержимое переменной после ее удаления происходит ошибка, как если бы переменная никогда не создавалась:

```
>>> x = 5
>>> print(x)
5
```

```
>>> del x
>>> print(x)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

Здесь мы видим *трассировку*, которая выводится при возникновении ошибки, называемой *исключением*. Последняя строка сообщает, какое исключение было обнаружено; в данном случае это исключение `NameError` для `x`. После удаления `x` уже не является действительным именем переменной. В данном случае трассировка возвращает только `line 1, in <модуль>`, потому что в интерактивном режиме была отправлена только одна строка. В общем случае возвращается полная динамическая структура вызовов существующей функции на момент возникновения ошибки.

Если вы работаете в IDLE, то получаете ту же информацию с незначительными отличиями. Код может выглядеть примерно так:

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

В главе 14 этот механизм описан более подробно. Полный список возможных исключений и их причин содержится в документации стандартной библиотеки Python. Используйте алфавитный список для поиска описаний любых полученных конкретных исключений (таких, как `NameError`).

В именах переменных учитывается регистр символов; они могут содержать любые алфавитно-цифровые символы, а также символы подчеркивания, но должны начинаться с буквы или символа подчеркивания. За дополнительной информацией о создании имен переменных в стиле Python обращайтесь к разделу 4.10.

4.4. Выражения

Python поддерживает арифметические и другие математические выражения, которые покажутся знакомыми большинству читателей. Следующий фрагмент вычисляет среднее арифметическое чисел 3 и 5, сохраняя результат в переменной `z`:

```
x = 3
y = 5
z = (x + y) / 2
```

Следует помнить, что арифметические операторы, использующие только целые числа, *не всегда* возвращают целое число. Хотя все значения являются целыми числами, оператор деления (начиная с Python 3) возвращает число с плавающей точкой, так что дробная часть не теряется. Если же вы хотите выполнить традиционное целочисленное деление, возвращающее округленное целое число, используйте оператор `//`.

В выражениях действуют стандартные правила приоритета операций. Если не включить круглые скобки в последнюю строку, то код будет вычисляться в порядке `x + (y / 2)`.

Выражения не обязаны включать только цифровые значения; строки, логические значения и многие другие типы объектов также могут использоваться в выражениях. Эти объекты будут более подробно рассмотрены по мере использования.

ПОПРОБУЙТЕ САМИ: ПЕРЕМЕННЫЕ И ВЫРАЖЕНИЯ

Создайте в оболочке Python несколько переменных. Что произойдет, если вы попробуете включить пробелы, дефисы или другие неалфавитные символы в имя переменной? Поэкспериментируйте с более сложными выражениями — например, `x = 2 + 4 * 5 – 6/3`. Используйте круглые скобки для группировки чисел и посмотрите, как изменяется результат по сравнению с исходным выражением без группировки.

4.5. Строки

Вы уже видели, что в Python, как и в большинстве других языков программирования, строки заключаются в двойные кавычки. Следующий пример сохраняет строку `"Hello, World"` в переменной `x`:

```
x = "Hello, World"
```

Символ `\` (обратный слеш) используется для экранирования символов, то есть для назначения им специального смысла. `\n` — символ новой строки, `\t` — символ табуляции, `\\` — один обычный символ обратного слеша, а `\"` — обычный символ двойной кавычки (такой символ строки не завершает):

```
x = "\tThis string starts with a \"tab\"."
x = "This string contains a single backslash(\\)."
```

Вместо двойных кавычек можно использовать одинарные. Следующие две строки делают одно и то же:

```
x = "Hello, World"
x = 'Hello, World'
```

Единственное различие заключается в том, что в строках, заключенных в одинарные кавычки, не обязательно экранировать символы `"`, а в строках, заключенных в двойные кавычки, — символы `'`:

```
x = "Don't need a backslash"
x = 'Can\'t get by without a backslash'
x = "Backslash your \" character!"
x = 'You can leave the " alone'
```

Обычные строки не могут разбиваться символами новой строки. Следующий фрагмент работать не будет:

```
# При выполнении этого кода Python произойдет ОШИБКА -- простая разбивка строк не
работает.
x = "This is a misguided attempt to
put a newline into a string without using backslash-n"
```

Но в Python поддерживаются строки в утроенных кавычках, которые позволяют выполнять разбивку, а также включать в текст одинарные и двойные кавычки без экранирования символом \:

```
x = """Starting and ending a string with triple " characters
permits embedded newlines, and the use of " and ' without
backslashes"""
```

Теперь `x` содержит весь текст, заключенный между ограничителями `"""`. (Вместо утроенных двойных кавычек можно использовать утроенные одинарные кавычки `'''` — результат будет тем же.)

Python предлагает достаточно богатую функциональность работы со строками; этой теме посвящена глава 6.

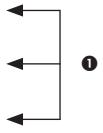
4.6. Числа

Скорее всего, читатели уже знакомы со стандартными числовыми операциями по другим языкам программирования, поэтому в книге не будет отдельной главы с описанием возможностей Python по работе со строками. В этой главе описаны уникальные возможности чисел Python, а все доступные функции перечислены в документации Python.

В Python поддерживаются четыре разновидности чисел: *целые числа*, *числа с плавающей точкой*, *комплексные числа* и *логические значения*. Целочисленные константы записываются в стандартном формате целых чисел: 0, –11, +33, 123456 — и обладают неограниченным диапазоном, который ограничивается только ресурсами вашего компьютера. Число с плавающей точкой может записываться либо в дробном формате, либо в экспоненциальной (научной) записи: 3.14, –2E-8, 2.718281828. Точность этих значений определяется архитектурой машины, но обычно она соответствует точности типа double (64-разрядного) в языке C. Вероятно, комплексные числа представляют интерес лишь для узкого круга читателей; они рассматриваются позднее в этом разделе. Логические значения принимают значения `True` и `False`, а по своему поведению идентичны 1 и 0 (если не считать строковых представлений).

Арифметические операции во многом напоминают язык C. Операции с двумя целыми числами дают целое число, кроме операции деления (`/`), которая дает число с плавающей точкой. Если использовать знак `//`, то результатом будет целое число (дробная часть отсекается). Операции с числами с плавающей точкой всегда дают результат с плавающей точкой. Несколько примеров:

```
>>> 5 + 2 - 3 * 2
1
>>> 5 / 2          # При обычном делении результат с плавающей точкой
2.5
>>> 5 / 2.0        # Также результат с плавающей точкой
2.5
>>> 5 // 2         # Целочисленное деление с оператором '//'
2
>>> 30000000000    # Во многих языках такое значение слишком велико для int
30000000000
>>> 30000000000 * 3
90000000000
>>> 30000000000 * 3.0
90000000000.0
>>> 2.0e-8         # Результат с плавающей точкой
2e-08
>>> 3000000 * 3000000
9000000000000
>>> int(200.2)
200
>>> int(2e2)
200
>>> float(200)
200.0
```



❶

Здесь продемонстрированы явные преобразования между типами ❶. Вызов `int` округляет дробные значения.

У чисел Python есть два преимущества по сравнению с C или Java: целые числа могут быть произвольно большими, а при делении двух целых чисел получается результат с плавающей точкой.

4.6.1. Встроенные числовые функции

Python предоставляет следующие числовые функции, входящие в базовый набор встроенной функциональности:

```
abs, divmod, float, hex, int, max, min, oct,
pow, round
```

За подробностями обращайтесь к документации.

4.6.2. Сложные числовые функции

Нетривиальные числовые функции, например тригонометрические и гиперболические тригонометрические функции, а также ряд полезных констант, не встроены в Python, а предоставляются стандартным модулем `math`. Модули будут более подробно рассмотрены позднее, а пока достаточно знать, что для получения доступа к функциям math из этого раздела следует начать программу Python или интерактивный сеанс командой

```
from math import *
```

Модуль `math` предоставляет следующие функции и константы:

```
acos, asin, atan, atan2, ceil, cos, cosh, e, exp, fabs, floor, fmod,
frexp, hypot, ldexp, log, log10, mod, pi, pow, sin, sinh, sqrt, tan,
tanh
```

За подробностями обращайтесь к документации.

4.6.3. Числовые расчеты

Базовая функциональность Python не очень хорошо подходит для интенсивных числовых расчетов из-за ограничений скорости. Однако мощное расширение Python `NumPy` предоставляет чрезвычайно эффективные реализации многих расширенных числовых операций. Особое внимание уделяется операциям с массивами, включая многомерные матрицы и такие нетривиальные функции, как быстрое преобразование Фурье. Вы найдете пакет `NumPy` (или ссылки на него) на сайте www.scipy.org.

4.6.4. Комплексные числа

Комплексные числа создаются автоматически каждый раз, когда в программе создается выражение вида `nj`, где `n` записывается в форме целого числа или числа с плавающей точкой Python. Конечно, `j` здесь соответствует стандартному обозначению мнимого числа, равного квадратному корню из –1, например:

```
>>> (3+2j)
(3+2j)
```

Обратите внимание: Python выводит полученное комплексное число в круглых скобках, чтобы показать, что выводимые данные представляют значение одного объекта:

```
>>> 3 + 2j - (4+4j)
(-1-2j)
>>> (1+2j) * (3+4j)
(-5+10j)
>>> 1j * 1j
(-1+0j)
```

Выражение `j * j` дает ожидаемый ответ –1, но результат остается объектом комплексного числа Python. Комплексные числа никогда не преобразуются автоматически в эквивалентный объект вещественного или целого числа. Тем не менее вы можете легко получить их вещественную и чисто мнимую часть при помощи функций `real` и `imag`:

```
>>> z = (3+5j)
>>> z.real
3.0
>>> z.imag
5.0
```

Обратите внимание: вещественная и мнимая части комплексного числа всегда возвращаются в формате чисел с плавающей точкой.

4.6.5. Расширенные функции комплексных чисел

Функции из модуля `math` не работают с целыми числами; в конце концов, большинство пользователей предпочитает, чтобы при извлечении квадратного корня из –1 программа выдавала ошибку, а не ответ!

Вместо этого модуль `cmath` предоставляет аналогичные функции для работы с комплексными числами:

```
acos, acosh, asin, asinh, atan, atanh, cos, cosh, e, exp, log, log10,
pi, sin, sinh, sqrt, tan, tanh
```

Чтобы наглядно показать в программе, что функции являются специализированными версиями для комплексных чисел, и чтобы избежать конфликтов имен с более привычными эквивалентами, лучше импортировать модуль `cmath` командой

```
import cmath
```

а затем явно указывать пакет `cmath` при использовании функции:

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
```

О НЕЖЕЛАТЕЛЬНОСТИ IMPORT *

Перед вами хороший пример того, почему лучше свести к минимуму использование формы `from <module> import *` команды `import`. Скажем, если использовать команду для импортирования сначала модуля `math`, а затем модуля `cmath`, функции `cmath` заменят одноименные функции `math`. Кроме того, читателю вашего кода будет сложнее определить источник конкретных используемых функций. Некоторые модули специально проектировались для использования этой формы импортирования.

За подробной информацией об использовании модулей и имен модулей обращайтесь к главе 10.

Важно помнить, что при импортировании модуля `cmath` можно сделать практически все, что делается с другими числами.

ПОПРОБУЙТЕ САМИ: РАБОТА СО СТРОКАМИ И ЧИСЛАМИ

В оболочке Python создайте несколько строковых и числовых переменных (целые числа, числа с плавающей точкой *и* комплексные числа). Поэкспериментируйте с различными операциями, в том числе и между типами. Можно ли, например, умножить строку на число? А умножить ее на число с плавающей точкой или комплексное число? Загрузите модуль `math` и опробуйте некоторые из его функций; затем загрузите модуль `cmath` и сделайте то же самое. Что произойдет, если вы попытаетесь вызвать одну из этих функций для целого числа или числа с плавающей точкой после загрузки модуля `cmath`? Как снова получить доступ к функциям модуля `math`?

4.7. Значение None

Кроме стандартных типов (таких, как строки и числа), в Python существует специальный базовый тип данных, определяющий один специальный объект данных с именем `None`. Как подсказывает имя, `None` используется для представления неопределенных значений. В Python оно неоднократно встречается в разных обличиях. Например, процедура в Python представляет собой функцию, которая не возвращает явное значение, а это означает, что по умолчанию она возвращает `None`.

Значение `None` часто используется в повседневном программировании Python в качестве заместителя; оно показывает, что значение некоторого поля структуры данных будет получено со временем, хотя в настоящее время оно еще не вычислено. Проверка присутствия `None` выполняется легко, потому что во всей системе Python существует только один экземпляр `None` (все упоминания `None` относятся к одному объекту), и значение `None` эквивалентно только самому себе.

4.8. Получение данных от пользователя

Функция `input()` предназначена для получения данных от пользователя. В ее параметре передается строка запроса, которая должна быть выведена для пользователя:

```
>>> name = input("Name? ")
Name? Jane
>>> print(name)
Jane
>>> age = int(input("Age? "))  <-- Преобразует введенное значение из строки в целое число
Age? 28
>>> print(age)
28
>>>
```

Этот способ получения входных данных относительно прост. Единственная загвоздка заключается в том, что ввод поступает в виде строки, так что если вы захотите использовать его как число, придется преобразовать данные вызовом функции `int()` или `float()`.

ПОПРОБУЙТЕ САМИ: ПОЛУЧЕНИЕ ВХОДНЫХ ДАННЫХ

Поэкспериментируйте с функцией `input()` для получения строковых и целочисленных данных. Если вы используете код вроде приведенного выше, что получится, если не применять `int()` к вызову `input()` для ввода целого числа? Сможете ли вы изменить этот код, чтобы программа получала число с плавающей точкой — скажем, 28.5? Что произойдет, если намеренно ввести значение неправильного типа — например, число с плавающей точкой вместо целого, строку вместо числа или наоборот?

4.9. Встроенные операторы

Python предоставляет разнообразные встроенные операторы, от стандартных (+, * и т. д.) до более экзотических (например, операторы поразрядного сдвига, поразрядные логические функции и т. д.). Большинство этих операторов наверняка знакомо вам по другим языкам программирования, поэтому я не буду подробно объяснять их в тексте. Полный список встроенных операторов Python приведен в документации.

4.10. Основной стиль программирования на Python

В Python относительно немного ограничений на стиль программирования, если не считать очевидного исключения — обязательного применения отступов для разделения кода на блоки. Даже в этом случае величина отступов и их тип (табуляция или пробелы) не задаются жестко. Тем не менее существуют рекомендации по стилю Python, которые сформулированы в предложении по улучшению Python, или PEP (Python Enhancement Proposal) 8. Сводка этих рекомендаций приведена в приложении А и доступна в по адресу www.python.org/dev/peps/pep-0008/. Подборка рекомендаций Python приведена в табл. 4.1, но чтобы полностью усвоить стиль программирования на Python, следует периодически перечитывать PEP 8.

Таблица 4.1. Рекомендации по стилю программирования Python

Ситуация	Рекомендация	Пример
Имена модулей/пакетов	Короткие, в нижнем регистре, подчеркивания только при необходимости	`imp, sys`
Имена функций	В нижнем регистре, подчеркивания_для_удобочитаемости	`foo(), my_func()`
Имена переменных	В нижнем регистре, подчеркивания_для_удобочитаемости	`my_var`
Имена классов	КаждоеСловоСПрописнойБуквы	`MyClass`
Имена констант	ВЕРХНИЙ_РЕГИСТР_С_ПОДЧЕРКИВАНИЯМИ	`PI, TAX_RATE`
Отступы	Четыре пробела на уровень, без табуляций	
Сравнения	Явные сравнения с True и False нежелательны	`if my_var:` `if not my_var:`

Я настоятельно рекомендую следовать рекомендациям PEP 8. Они были разумно выбраны и прошли проверку временем; если вы будете применять их, вам и другим программистам Python будет проще понять код.

БЫСТРАЯ ПРОВЕРКА: СТИЛЬ ПРОГРАММИРОВАНИЯ PYTHON

Какие из следующих имен переменных и функций не относятся к хорошему стилю программирования на Python? Почему?

```
bar(, varName, VERYLONGVARNAME, foobar, longvarname,
foo_bar(), really_very_long_var_name
```

Итоги

- Основного синтаксиса, кратко описанного в этой главе, достаточно для того, чтобы начать писать код Python.
- Синтаксис Python логичен и предсказуем.
- Так как синтаксис не преподносит особых сюрпризов, многие программисты неожиданно быстро переходят к написанию кода Python.

5 Списки, кортежи и множества

Эта глава охватывает следующие темы:

- ✓ Управление списками и индексами списка
- ✓ Изменение списков
- ✓ Сортировка
- ✓ Использование операций общего списка
- ✓ Обработка вложенных списков и глубоких копий
- ✓ Использование кортежей
- ✓ Создание и использование наборов

В этой главе рассматриваются две важнейшие разновидности последовательностей Python: списки и кортежи. На первый взгляд списки напоминают массивы во многих других языках, но не обманывайтесь; списки намного превосходят обычные массивы своей гибкостью и мощью.

Кортежи, как и списки, не могут изменяться; их можно рассматривать как ограниченную разновидность списков или как базовый тип записи данных. Необходимость в таких ограниченных типах данных будет объяснена позднее в этой главе. Кроме того, в этой главе обсуждается более новый тип коллекций Python: множества. Множества полезны тогда, когда для вас факт принадлежности объекта к коллекции важнее его позиции в этой коллекции.

Большая часть главы посвящена спискам, потому что если вы понимаете списки, то вы в значительной степени понимаете и кортежи. В последней части главы рассматриваются различия между списками и кортежами — в отношении как функциональности, так и строения.

5.1. Сходство между списками и массивами

Списки Python имеют много общего с массивами Java, C или любого другого языка; они также представляют собой упорядоченные последовательности объектов.

Чтобы создать список, заключите перечень элементов, разделенных запятыми, в квадратные скобки:

```
# Переменной x присваивается список из трех элементов
x = [1, 2, 3]
```

Вам не нужно объявлять список или заранее фиксировать его размер. Эта строка программы создает список, а также присваивает его переменной, причем этот список автоматически увеличивается или сокращается по мере необходимости.

МАССИВЫ В PYTHON

В Python доступен модуль `array`, который предоставляет поддержку массивов на базе типов данных C. Информацию об использовании этого модуля можно найти в справочнике *Python Library Reference*. Я рекомендую обращаться к этому модулю только в том случае, если вам действительно необходимо повышение быстродействия. Если ситуация требует интенсивных расчетов, рассмотрите возможность использования пакета `NumPy`, упомянутого в главе 4 (он доступен на сайте www.scipy.org).

В отличие от списков во многих языках программирования, списки Python также могут содержать элементы разных типов; элементом списка может быть любой объект Python. Следующий список содержит разнородные элементы:

```
# Первый элемент - число, второй - строка, третий - другой список.
x = [2, "two", [1, 2, 3]]
```

Пожалуй, основной встроенной функцией списков является функция `len`, которая возвращает количество элементов в списке:

```
>>> x = [2, "two", [1, 2, 3]]
>>> len(x)
3
```

Обратите внимание: функция `len` не учитывает элементы внутреннего вложенного списка.

БЫСТРАЯ ПРОВЕРКА: LEN()

Что вернет функция `len()` для каждого из следующих списков:

`[0]; []; [[1, 3, [4, 5], 6], 7]`?

5.2. Индексы списков

Если вы будете понимать, как работают индексы списков, вы сможете извлечь намного больше пользы из программирования на языке Python. Пожалуйста, внимательно прочитайте весь раздел!

Для извлечения элементов из списка Python используется синтаксис, сходный с синтаксисом индексирования массивов C. Как в C и во многих других языках, отсчет индексов в Python начинается с 0; при запросе элемента 0 вы получаете первый элемент списка, при запросе элемента 1 — второй элемент и т. д. Несколько примеров:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[0]
'first'
>>> x[2]
'third'
```

Однако механизм индексирования Python заметно превосходит индексирование C по гибкости. Отрицательные индексы обозначают позиции элементов от конца списка; –1 соответствует последней позиции списка, –2 — предпоследней и т. д. Продолжим пример для того же списка `x`:

```
>>> a = x[-1]
>>> a
'fourth'
>>> x[-2]
'third'
```

Для операций, в которых задействован только один индекс списка, обычно можно представлять индекс как указатель на конкретный элемент списка. Для более сложных операций индексы правильнее представлять как обозначения позиций *между* элементами. Так, для списка `["first", "second", "third", "fourth"]` индексы можно представлять так:

`x = [`		`"first",`		`"second",`		`"third",`		`"fourth"`		`]`
Положительные индексы	0		1		2		3			
Отрицательные индексы	–4		–3		–2		–1			

Это не так важно при извлечении одного элемента, но Python может извлекать или выполнять присваивание сразу для целой части списка — такая операция называется *сегментированием* (slicing). Вместо того чтобы вводить `list[index]` для извлечения элемента после позиции `index`, введите `list[index1:index2]` для извлечения всех элементов от `index1` (включительно) до `index2` (не включая) в новый список. Несколько примеров:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[1:-1]
['second', 'third']
>>> x[0:3]
['first', 'second', 'third']
>>> x[-2:-1]
['third']
```

Казалось бы, если второй индекс обозначает позицию списка, *предшествующую* позиции первого индекса, код должен вернуть элементы между этими индексами в обратном порядке, однако на практике этого не происходит. Вместо этого код возвращает пустой список:

```
>>> x[-1:2]
[]
```

При сегментировании списка также можно опустить `index1` или `index2`. Если отсутствует `index1`, сегмент начинается от начала списка, а если отсутствует `index2`, сегмент продолжается до конца списка:

```
>>> x[:3]
['first', 'second', 'third']
>>> x[2:]
['third', 'fourth']
```

Если опущены оба индекса, новый список распространяется от начала до конца исходного списка, то есть список копируется. Этот прием может пригодиться для создания копии, которую можно изменять без модификации исходного списка:

```
>>> y = x[:]
>>> y[0] = '1 st'
>>> y
['1 st', 'second', 'third', 'fourth']
>>> x
['first', 'second', 'third', 'fourth']
```

ПОПРОБУЙТЕ САМИ: СЕГМЕНТЫ И ИНДЕКСЫ

Используя то, что вы знаете о функции `len()` и сегментах списков, как бы вы получили вторую половину списка неизвестного размера? Поэкспериментируйте в сеансе Python и убедитесь в том, что ваше решение работает.

5.3. Модификация списков

Синтаксис индексирования может использоваться как для модификации списков, так и для извлечения из них отдельных элементов. Укажите индекс в левой части оператора присваивания:

```
>>> x = [1, 2, 3, 4]
>>> x[1] = "two"
>>> x
[1, 'two', 3, 4]
```

Синтаксис сегментов может использоваться и в этом случае. Команда вида `lista[index1:index2] = listb` заменяет все элементы `lista` между `index1` и `index2` элементами из `listb`. Список `listb` может содержать больше или меньше элементов, чем удаляется из `lista`; в этом случае длина `lista` изменяется. Сегментное присваивание может использоваться для выполнения разных операций:

```
>>> x = [1, 2, 3, 4]
>>> x[len(x):] = [5, 6, 7]  <-- Присоединяет список к концу списка
>>> x
[1, 2, 3, 4, 5, 6, 7]
>>> x[:0] = [-1, 0]  <-- Присоединяет список к началу списка
>>> x
[-1, 0, 1, 2, 3, 4, 5, 6, 7]
>>> x[1:-1] = []  <-- Удаляет элементы из списка
>>> x
[-1, 7]
```

Присоединение одного элемента к списку — операция настолько распространенная, что для нее был определен специальный метод `append`:

```
>>> x = [1, 2, 3]
>>> x.append("four")
>>> x
[1, 2, 3, 'four']
```

При попытке присоединения одного списка к другому может возникнуть проблема — список будет присоединен как один элемент основного списка:

```
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7]
>>> x.append(y)
>>> x
[1, 2, 3, 4, [5, 6, 7]]
```

Метод `extend` похож на метод `append`, но он предназначен для добавления элементов одного списка к другому:

```
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7]
>>> x.extend(y)
>>> x
[1, 2, 3, 4, 5, 6, 7]
```

Также существует специальный метод `insert`, который вставляет новый элемент списка между двумя существующими элементами или в начало списка. Метод `insert` используется как метод списков и получает два дополнительных аргумента. Первый дополнительный аргумент определяет индекс позиции списка, в которую будет вставлен новый элемент, а второй — сам новый элемент:

```
>>> x = [1, 2, 3]
>>> x.insert(2, "hello")
>>> print(x)
[1, 2, 'hello', 3]
>>> x.insert(0, "start")
>>> print(x)
['start', 1, 2, 'hello', 3]
```

Метод `insert` интерпретирует индексы так, как описано в разделе 5.2, но в большинстве случаев проще представлять вызов `list.insert(n, elem)` как вставку `elem` перед

n-м элементом списка `list`. Метод `insert` создан исключительно для удобства. Все, что можно сделать с методом `insert`, также можно сделать посредством присваивания сегментов. А именно конструкция `list.insert(n,elem)` эквивалентна `list[n:n] = [elem]`, где `n` — неотрицательное число. Использование `insert` несколько упрощает чтение кода, к тому же метод `insert` поддерживает отрицательные индексы:

```
>>> x = [1, 2, 3]
>>> x.insert(-1, "hello")
>>> print(x)
[1, 2, 'hello', 3]
```

Для удаления элементов списков или сегментов рекомендуется использовать команду `del`. Эта команда тоже не делает ничего такого, чего нельзя было бы сделать при помощи присваивания сегментов, но она лучше запоминается и упрощает чтение кода:

```
>>> x = ['a', 2, 'c', 7, 9, 11]
>>> del x[1]
>>> x
['a', 'c', 7, 9, 11]
>>> del x[:2]
>>> x
[7, 9, 11]
```

В общем случае команда `del list[n]` эквивалентна `list[n:n+1] = []`, тогда как команда `del list[m:n]` делает то же самое, что `list[m:n] = []`.

Метод `remove` не является обратным по отношению к `insert`. Если метод `insert` вставляет элемент в заданную позицию, `remove` ищет первый экземпляр заданного значения в списке и удаляет это значение из списка:

```
>>> x = [1, 2, 3, 4, 3, 5]
>>> x.remove(3)
>>> x
[1, 2, 4, 3, 5]
>>> x.remove(3)
>>> x
[1, 2, 4, 5]
>>> x.remove(3)
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

Если метод `remove` не находит удаляемых элементов, он выдает ошибку. Вы можете перехватить ошибку, используя средства обработки исключений Python, или же предотвратить саму проблему, проверяя наличие элементов в списке оператором `in` перед тем, как пытаться их удалить.

Метод `reverse` — специализированный метод изменения списка — эффективно переставляет элементы списка в обратном порядке «на месте»:

```
>>> x = [1, 3, 5, 6, 7]
>>> x.reverse()
>>> x
[7, 6, 5, 3, 1]
```

ПОПРОБУЙТЕ САМИ: МОДИФИКАЦИЯ СПИСКОВ

Допустим, список состоит из 10 элементов. Как переместить три последних элемента из конца в начало списка без нарушения их исходного порядка?

5.4. Сортировка списков

Для сортировки списков используется встроенный метод Python `sort`:

```
>>> x = [3, 8, 4, 0, 2, 1]
>>> x.sort()
>>> x
[0, 1, 2, 3, 4, 8]
```

Метод выполняет сортировку «на месте», то есть с изменением сортируемого списка. Если вы хотите отсортировать список без изменения исходного списка, возможны два варианта: либо использовать встроенную функцию `sorted()` (раздел 5.4.2), либо создать копию списка и отсортировать ее.

```
>>> x = [2, 4, 1, 3]
>>> y = x[:]
>>> y.sort()
>>> y
[1, 2, 3, 4]
>>> x
[2, 4, 1, 3]
```

Сортировка также работает со строками:

```
>>> x = ["Life", "Is", "Enchanting"]
>>> x.sort()
>>> x
['Enchanting', 'Is', 'Life']
```

Метод `sort` может отсортировать почти всё, потому что Python умеет сравнивать почти всё. Однако при сортировке возникает одна загвоздка: ключевой метод, используемый по умолчанию при сортировке, требует, чтобы все элементы списка имели совместимые типы. Это означает, что при использовании метода `sort` для списка, содержащего как числа, так и строки, будет выдано исключение:

```
>>> x = [1, 2, 'hello', 3]
>>> x.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

И наоборот, можно спокойно отсортировать список списков:

```
>>> x = [[3, 5], [2, 9], [2, 3], [4, 1], [3, 2]]
>>> x.sort()
>>> x
[[2, 3], [2, 9], [3, 2], [3, 5], [4, 1]]
```

В соответствии со встроенными правилами сравнения сложных объектов в Python субсписки сортируются сначала по возрастанию первого элемента, а затем по возрастанию второго элемента.

Метод `sort` обладает еще большей гибкостью; он поддерживает необязательный параметр `reverse`. При `reverse=True` сортировка производится в обратном порядке, и вы можете задать собственную ключевую функцию, определяющую способ сортировки элементов списка.

5.4.1. Нестандартная сортировка

Чтобы применить пользовательский критерий сортировки, необходимо уметь определять функции — тема, которая пока еще не рассматривалась. В этом разделе мы также обсудим тот факт, что `len(string)` возвращает количество символов в строке. Операции со строками более подробно рассмотрены в главе 6.

По умолчанию `sort` использует для определения порядка встроенные функции сравнения Python; в большинстве случаев этого достаточно. Однако время от времени требуется отсортировать список способом, отличным от порядка по умолчанию. Допустим, вы хотите отсортировать список слов по количеству символов в слове, а не по критерию лексикографической сортировки, который обычно применяет Python.

Для этого следует написать функцию, которая возвращает значение (или ключ) для проведения сортировки, и использовать его с методом `sort`. В контексте `sort` эта функция должна получать один аргумент и возвращать ключ или значение, используемое функцией `sort`.

Чтобы строки упорядочивались по количеству символов, ключевая функция может иметь вид

```
def compare_num_of_chars(string1):
    return len(string1)
```

Эта ключевая функция тривиальна — она передает методу сортировки длину строк (вместо самих строк).

После того как вы определите ключевую функцию, остается передать ее методу `sort` с именем `key`. Так как функции являются объектами Python, они могут передаваться функциям как любые другие объекты Python. Ниже приведена небольшая программа, демонстрирующая различия между стандартной и нестандартной сортировкой:

```
>>> def compare_num_of_chars(string1):
...     return len(string1)
>>> word_list = ['Python', 'is', 'better', 'than', 'C']
>>> word_list.sort()
>>> print(word_list)
['C', 'Python', 'better', 'is', 'than']
>>> word_list = ['Python', 'is', 'better', 'than', 'C']
>>> word_list.sort(key=compare_num_of_chars)
>>> print(word_list)
['C', 'is', 'than', 'Python', 'better']
```

Первый список упорядочен в лексикографическом порядке (верхний регистр предшествует нижнему), а второй список упорядочен по возрастанию количества символов.

Нестандартная сортировка чрезвычайно полезна, но она может быть медленнее стандартной. Обычно эффект минимален, и все же с особенно сложными ключевыми функциями он становится заметным, особенно если в сортировке задействованы сотни тысяч и миллионы элементов.

В частности, нестандартной сортировки стоит избегать при упорядочении списка по убыванию (а не по возрастанию). В этом случае параметру `reverse` метода `sort` задается значение `True`. Если по какой-либо причине это решение оказывается неприемлемым, все равно будет лучше отсортировать список обычным способом, а затем воспользоваться методом `reverse` для перестановки элементов полученного списка в обратном порядке. Эти две операции — стандартная сортировка и обратная перестановка — будут выполняться намного быстрее, чем нестандартная сортировка.

5.4.2. Функция sorted()

У списков имеется встроенный метод для сортировки, но у других итерируемых типов Python (например, у ключей словаря) метода `sort` нет. У Python также имеется встроенная функция `sorted()`, которая возвращает отсортированный список, построенная на базе любого итерируемого типа. Функция `sorted()` использует те же параметры `key` и `reverse`, что и метод `sort`:

```
>>> x = (4, 3, 1, 2)
>>> y = sorted(x)
>>> y
[1, 2, 3, 4]
>>> z = sorted(x, reverse=True)
>>> z
[4, 3, 2, 1]
```

ПОПРОБУЙТЕ САМИ: СОРТИРОВКА СПИСКОВ

Имеется список, каждый элемент которого также является списком: `[[1, 2, 3], [2, 1, 3], [4, 0, 1]]`. Допустим, вы хотите отсортировать этот список по второму элементу каждого списка, чтобы получить результат `[[4, 0, 1], [2, 1, 3], [1, 2, 3]]`. Какую функцию вы бы написали для передачи в параметре `key` метода `sort()`?

5.5. Другие распространенные операции со списками

Также у списков есть несколько других часто используемых методов, которые не относятся ни к какой конкретной категории.

5.5.1. Проверка принадлежности оператором in

Чтобы легко определить, присутствует ли некоторое значение в списке, воспользуйтесь оператором `in`, возвращающим логическое значение. Также можно выполнить обратную проверку с оператором `not in`:

```
>>> 3 in [1, 3, 4, 5]
True
>>> 3 not in [1, 3, 4, 5]
False
>>> 3 in ["one", "two", "three"]
False
>>> 3 not in ["one", "two", "three"]
True
```

5.5.2. Конкатенация списков оператором +

Чтобы создать список слиянием двух существующих списков, используйте оператор `+` (оператор конкатенации списков), который оставляет исходные списки-аргументы без изменений:

```
>>> z = [1, 2, 3] + [4, 5]
>>> z
[1, 2, 3, 4, 5]
```

5.5.3. Инициализация списков оператором *

Оператор * создает список заданного размера, инициализированный заданным значением. Эта операция типична для больших списков, размер которых известен заранее. И хотя вы можете использовать `append` для добавления элементов и автоматического расширения списка по мере необходимости, эффективнее будет использовать * для определения правильного размера списка в начале работы программы. Если размер списка остается неизменным, это позволяет избежать затрат на перемещение данных в памяти:

```
>>> z = [None] * 4
>>> z
[None, None, None, None]
```

При таком использовании списков оператор * (который в данном контексте называется *оператором умножения списков*) повторяет список заданное количество раз и объединяет все копии для формирования нового списка. Это стандартный способ опережающего определения списков заданного размера в языке Python. В умножении списков часто используется список, содержащий один экземпляр `None`, но содержимое списка может быть любым:

```
>>> z = [3, 1] * 2
>>> z
[3, 1, 3, 1]
```

5.5.4. Получение наименьшего или наибольшего элемента функциями min и max

Функции `min` и `max` используются для нахождения наименьшего и наибольшего элемента в списке. Вероятно, чаще всего вы будете использовать функции `min` и `max` с числовыми списками, но они могут применяться к спискам, содержащим элементы любых типов. Попытка определить наименьший или наибольший объект в множестве объектов разных типов приведет к ошибке, если сравнение этих типов не имеет смысла:

```
>>> min([3, 7, 0, -2, 11])
-2
>>> max([4, "Hello", [1, 2]])
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    max([4, "Hello",[1, 2]])
TypeError: '>' not supported between instances of 'str' and 'int'
```

5.5.5. Поиск в списках и метод index

Если вы хотите узнать, в какой позиции списка находится некоторое значение (то есть недостаточно знать, присутствует значение в списке или нет), используйте метод `index`. Этот метод ищет в списке элемент, эквивалентный заданному значению, и возвращает позицию этого элемента:

```
>>> x = [1, 3, "five", 7, -2]
>>> x.index(7)
3
>>> x.index(5)
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: 5 is not in list
```

Как видно из этого примера, при попытке определить позицию несуществующего элемента происходит ошибка. Эта ошибка обрабатывается точно так же, как и аналогичная ошибка, которая может происходить при вызове `remove` (то есть проверкой списка оператором `in` перед вызовом `index`).

5.5.6. Подсчет вхождений методом count

Метод `count` также проводит поиск заданного значения по списку, но вместо позиционной информации возвращает количество вхождений значения в список:

```
>>> x = [1, 2, 2, 3, 5, 2, 5]
>>> x.count(2)
3
>>> x.count(5)
2
>>> x.count(4)
0
```

5.5.7. Сводка операций со списками

Как видите, списки представляют собой чрезвычайно мощные структуры данных, которые по своим возможностям далеко превосходят старые добрые массивы. Операции со списками играют настолько важную роль в программировании на Python, что я для удобства приведу их краткую сводку в табл. 5.1.

Таблица 5.1. Операции со списками

Операция	Объяснение	Пример
`[]`	Создает пустой список	`x = []`
`len`	Возвращает длину списка	`len(x)`
`append`	Добавляет один элемент в конец списка	`x.append('y')`
`extend`	Добавляет список в конец списка	`x.extend(['a', 'b'])`
`insert`	Вставляет новый элемент в произвольную позицию списка	`x.insert(0, 'y')`
`del`	Удаляет элемент или сегмент из списка	`del(x[0])`
`remove`	Находит и удаляет заданное значение из списка	`x.remove('y')`
`reverse`	Переставляет элементы списка в обратном порядке «на месте»	`x.reverse()`
`sort`	Сортирует список «на месте»	`x.sort()`
`+`	Объединяет два списка	`x1 + x2`
`*`	Создает несколько копий списка	`x = ['y'] * 3`
`min`	Возвращает наименьший элемент списка	`min(x)`
`max`	Возвращает наибольший элемент списка	`max(x)`
`index`	Возвращает позицию значения в списке	`x.index['y']`
`count`	Подсчитывает количество вхождений значения в списке	`x.count('y')`
`sum`	Суммирует элементы списка (если они поддерживают суммирование)	`sum(x)`
`in`	Сообщает, присутствует ли элемент в списке	`'y' in x`

Если вы будете знать основные операции со списками, ваша жизнь как программиста Python заметно упростится.

БЫСТРАЯ ПРОВЕРКА: ОПЕРАЦИИ СО СПИСКАМИ

Какой результат вернет вызов len([[1,2]] * 3)?

Опишите два различия между оператором `in` и методом `index()` списков?

Какой из следующих вызовов приведет к выдаче исключения: `min(["a", "b", "c"])`; `max([1, 2, "three"])`; `[1, 2, 3].count("one")`?

ПОПРОБУЙТЕ САМИ: ОПЕРАЦИИ СО СПИСКАМИ

Имеется список `x`. Напишите код безопасного удаления элемента в том — и только в том! — случае, если значение присутствует в списке.

Измените код так, чтобы элемент удалялся только в том случае, если элемент присутствует в списке более чем в одном экземпляре.

5.6. Вложенные списки и глубокое копирование

В этом разделе рассматривается еще одна нетривиальная тема, которую можно пропустить, если вы только изучаете язык.

Как упоминалось ранее, списки могут содержать вложенные списки. В частности, вложение списков может использоваться для представления двумерных матриц. К элементам этих матриц можно обращаться по двумерным индексам. Индексы матриц работают так:

```
>>> m = [[0, 1, 2], [10, 11, 12], [20, 21, 22]]
>>> m[0]
[0, 1, 2]
>>> m[0][1]
1
>>> m[2]
[20, 21, 22]
>>> m[2][2]
22
```

Этот механизм естественным образом масштабируется для большего количества измерений.

В большинстве случаев ни о чем больше вам беспокоиться не придется. Однако у вас могут возникнуть проблемы с вложенными списками — они возникают из-за того, как переменные связываются с объектами, и из-за возможности изменения некоторых объектов (например, списков). Следующий пример демонстрирует эти проблемы:

```
>>> nested = [0]
>>> original = [nested, 1]
>>> original
[[0], 1]
```

Рисунок 5.1 поясняет суть происходящего. Теперь значение во вложенном списке можно изменить как через переменную `nested`, так и через `original`:

```
>>> nested[0] = 'zero'
>>> original
[['zero'], 1]
>>> original[0][0] = 0
>>> nested
```

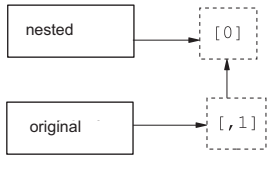


Рис. 5.1. Список, первый элемент которого ссылается на вложенный список

```
[0]
>>> original
[[0], 1]
```

Но если присвоить `nested` другой список, связь между списками будет разорвана:

```
>>> nested = [2]
>>> original
[[0], 1]
```

Эта ситуация изображена на рис. 5.2.

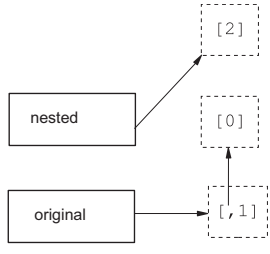


Рис. 5.2. Первый элемент исходного списка все еще остается вложенным списком, но переменная nested ссылается на другой список

Вы уже видели, что для создания копии списка можно воспользоваться полным сегментом (то есть `x[:]`). Также копию списка можно получить при помощи оператора `+` или `*` (например, `x + []` или `x * 1`). Эти способы несколько уступают по эффективности способу с сегментом. Все три способа создают *поверхностную* копию списка — вероятно, в большинстве случаев это именно то, что вам нужно. Но если ваш список содержит другие вложенные списки, возможно, вы предпочтете создать *глубокую* копию. Для этого можно воспользоваться функцией `deepcopy` модуля `copy`:

```
>>> original = [[0], 1]
>>> shallow = original[:]
>>> import copy
>>> deep = copy.deepcopy(original)
```

Рисунок 5.3 поясняет суть глубокого копирования.

Списки, на которые указывают переменные original или shallow, связаны. Изменение значения во вложенном списке через одну переменную повлияет на другую переменную:

```
>>> shallow[1] = 2
>>> shallow
[[0], 2]
>>> original
[[0], 1]
>>> shallow[0][0] = 'zero'
>>> original
[['zero'], 1]
```

Глубокая копия не зависит от оригинала, и изменения в ней не отражаются на исходном списке:

```
>>> deep[0][0] = 5
>>> deep
[[5], 1]
>>> original
[['zero'], 1]
```

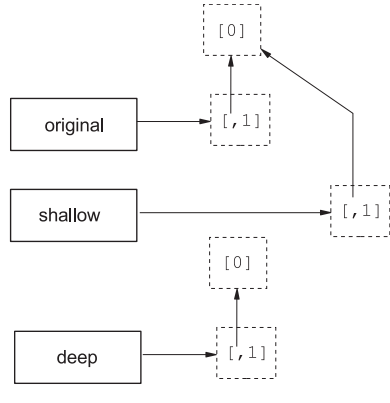


Рис. 5.3. При поверхностном копировании не копируются вложенные списки

Это поведение присуще всем остальным вложенным объектам в изменяемых списках (например, словарях).

Итак, теперь вы знаете, что можно сделать при помощи списков, и мы можем обратиться к кортежам.

ПОПРОБУЙТЕ САМИ: КОПИРОВАНИЕ СПИСКОВ

Имеется следующий список: `x = [[1, 2,3], [4, 5, 6], [7, 8, 9]]`. Какой код вы бы использовали для создания копии у этого списка, в которой элементы можно было бы изменять *без* побочного эффекта с изменением содержимого `x`?

5.7. Кортежи

Кортеж как структура данных очень похож на список, но кортежи не могут изменяться; их можно только создавать. Кортежи настолько похожи на списки, что у вас даже может возникнуть вопрос, для чего они были включены в Python. Дело в том, что у кортежей есть важные роли, которые не могут эффективно выполняться списками (например, роль ключей в словарях).

5.7.1. Знакомство с кортежами

Создание кортежа практически не отличается от создания списка: переменной присваивается последовательность значений. Список представляет собой последовательность, заключенную в квадратные скобки `[` и `]`; кортеж представляет собой последовательность, заключенную в круглые скобки `(` и `)`:

```
>>> x = ('a', 'b', 'c')
```

Эта строка создает кортеж из трех элементов.

После того как кортеж будет создан, операции с ним очень похожи на операции со списками — настолько, что разработчику легко забыть, что кортежи и списки являются разными типами данных:

```
>>> x[2]
'c'
>>> x[1:]
('b', 'c')
>>> len(x)
3
>>> max(x)
'c'
>>> min(x)
'a'
>>> 5 in x
False
>>> 5 not in x
True
```

Главное различие между кортежами и списками заключается в том, что кортежи неизменяемы. Попытка изменить кортеж приводит к выдаче невразумительного сообщения об ошибке. Так Python пытается сказать, что он не знает, как задать значение элемента кортежа:

```
>>> x[2] = 'd'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Кортежи можно создавать на основе существующих списков при помощи операторов + и *:

```
>>> x + x
('a', 'b', 'c', 'a', 'b', 'c')
>>> 2 * x
('a', 'b', 'c', 'a', 'b', 'c')
```

Копии кортежей создаются теми же способами, что и копии списков:

```
>>> x[:]
('a', 'b', 'c')
>>> x * 1
('a', 'b', 'c')
>>> x + ()
('a', 'b', 'c')
```

Если вы не прочитали раздел 5.6, оставшуюся часть этого абзаца можно пропустить. Сами кортежи изменяться не могут, но если они содержат изменяемые объекты (например, списки или словари), такие объекты могут изменяться, если они все еще связаны со своими собственными переменными. Кортежи, содержащие изменяемые объекты, не могут использоваться в качестве ключей в словарях.

5.7.2. Одноэлементные кортежи должны содержать запятую

С использованием кортежей связана одна синтаксическая тонкость. Так как квадратные скобки, в которые заключаются списки, в Python больше нигде не используются, понятно, что `[]` обозначает пустой список, а `[1]` обозначает список из одного элемента. О круглых скобках, в которые заключаются кортежи, этого сказать нельзя. Круглые скобки также могут использоваться для группировки элементов выражений с целью обеспечения определенного порядка вычисления. Допустим, в программе Python встречается запись `(x + y)`, что это должно означать? Что `x` и `y` нужно сложить и поместить в кортеж из одного элемента или же что `x` и `y` следует сложить до обработки любых других частей выражения?

Эта ситуация создает проблемы только для кортежей из одного элемента, потому что кортежи с несколькими элементами всегда содержат запятые, которыми разделяются элементы. По наличию запятых Python понимает, что круглые скобки обозначают кортеж, а не способ группировки. Для устранения неоднозначности в случае одноэлементного кортежа Python требует, чтобы за элементом кортежа

следовала запятая. В случае кортежа с нулем элементов (пустым кортежем) проблем нет. Пустая пара круглых скобок должна быть кортежем, потому что в противном случае она не имеет смысла:

```
>>> x = 3
>>> y = 4
>>> (x + y)   # Эта строка суммирует x и y.
7
>>> (x + y,)  # Запятая показывает, что круглые скобки обозначают кортеж.
(7,)
>>> ()        # Пустая пара скобок создает пустой кортеж.
()
```

5.7.3. Упаковка и распаковка кортежей

Для удобства Python позволяет размещать кортежи в левой части оператора присваивания. В этом случае переменным из кортежа присваиваются соответствующие значения из кортежа в правой части оператора. Простой пример:

```
>>> (one, two, three, four) =  (1, 2, 3, 4)
>>> one
1
>>> two
2
```

Этот пример можно записать еще проще, потому что Python распознает кортежи в контексте присваивания даже без круглых скобок. Значения в правой части упаковываются в кортеж, а затем распаковываются в переменные из левой части:

```
one, two, three, four = 1, 2, 3, 4
```

Одна строка кода заменила целых четыре строки:

```
one = 1
two = 2
three = 3
four = 4
```

Это удобный способ перестановки значений двух переменных. Вместо того чтобы использовать запись

```
temp = var1
var1 = var2
var2 = temp
```

достаточно написать

```
var1, var2 = var2, var1
```

Чтобы разработчику было еще удобнее, в Python 3 появился расширенный синтаксис распаковки, который позволяет элементу с пометкой * поглотить любое количество элементов, не имеющих парного элемента. И снова примеры будут более понятными, чем описание:

```
>>> x = (1, 2, 3, 4)
>>> a, b, *c = x
>>> a, b, c
(1, 2, [3, 4])
>>> a, *b, c = x
>>> a, b, c
(1, [2, 3], 4)
>>> *a, b, c = x
>>> a, b, c
([1, 2], 3, 4)
>>> a, b, c, d, *e = x
>>> a, b, c, d, e
(1, 2, 3, 4, [])
```

Обратите внимание: все лишние элементы сохраняются в элементе со звездочкой в виде списка, а при отсутствии лишних элементов элементу со звездочкой присваивается пустой список.

Упаковка и распаковка также могут выполняться с ограничителями списков:

```
>>> [a, b] = [1, 2]
>>> [c, d] = 3, 4
>>> [e, f] = (5, 6)
>>> (g, h) = 7, 8
>>> i, j = [9, 10]
>>> k, l = (11, 12)
>>> a
1
>>> [b, c, d]
[2, 3, 4]
>>> (e, f, g)
(5, 6, 7)
>>> h, i, j, k, l
(8, 9, 10, 11, 12)
```

5.7.4. Преобразования между списками и кортежами

Кортежи легко преобразуются в списки функцией `list`, которая получает в аргументе любую последовательность и строит новый список с элементами из исходной последовательности. Аналогичным образом список можно преобразовать в кортеж функцией `tuple`, которая делает то же самое, но создает новый кортеж вместо нового списка:

```
>>> list((1, 2, 3, 4))
[1, 2, 3, 4]
>>> tuple([1, 2, 3, 4])
(1, 2, 3, 4)
```

Попутно отметим, что список предоставляет удобный способ разбиения строк на символы:

```
>>> list("Hello")
['H', 'e', 'l', 'l', 'o']
```

Этот прием работает, потому что метод `list` (и `tuple`) может применяться к любой последовательности Python, а строка является обычной последовательностью символов. (Строки более подробно рассматриваются в главе 6.)

БЫСТРАЯ ПРОВЕРКА: КОРТЕЖИ

Объясните, почему следующие операции недопустимы для кортежа `x = (1, 2, 3, 4)`:

```
x.append(1)
x[1] = "hello"
del x[2]
```

Если у вас имеется кортеж `x = (3, 1, 4, 2)`, как можно отсортировать элементы `x`?

5.8. Множества

Множество (set) в Python представляет собой неупорядоченную коллекцию объектов, которая используется в том случае, если вас интересует прежде всего факт принадлежности объекта к коллекции и его уникальность. Как и ключи словарей (глава 7), элементы множества должны быть неизменяемыми и хешируемыми. Это означает, что целые числа, числа с плавающей точкой, строки и кортежи могут быть элементами множества, а списки, словари и сами множества — нет.

5.8.1. Операции с множествами

Кроме операций, применимых к коллекциям вообще (например, `in`, `len` и перебор для циклов `for`), множества поддерживают ряд операций, специфических для множеств:

```
>>> x = set([1, 2, 3, 1, 3, 5]) ❶
>>> x
{1, 2, 3, 5} ❷
>>> x.add(6) ❸
>>> x
{1, 2, 3, 5, 6}
>>> x.remove(5) ❹
>>> x
{1, 2, 3, 6}
>>> 1 in x ❺
True
>>> 4 in x
False
>>> y = set([1, 7, 8, 9])
>>> x | y ❻
{1, 2, 3, 6, 7, 8, 9}
>>> x & y ❼
{1}
>>> x ^ y ❽
{2, 3, 6, 7, 8, 9}
>>>
```

Множество можно создать вызовом `set` для последовательности — например, для списка ❶. При преобразовании последовательности в множество дубликаты исключаются ❷. После создания множества функцией `set` вы можете использовать методы `add` ❸ и `remove` ❹ для изменения элементов множества. Ключевое слово `in` используется для проверки принадлежности объекта к множеству ❺. Оператор `|` ❻ вычисляет объединение двух множеств, оператор `&` — их пересечение ❼, а оператор `^` ❽ — их симметрическую разность (то есть элементы, входящие только в одно из двух множеств).

Эти примеры не содержат полной сводки операций с множествами, но и они дают представление о том, как работают множества. За дополнительной информацией обращайтесь к официальной документации Python.

5.8.2. Фиксированные множества

Поскольку множества не являются неизменяемыми и хешируемыми, они не могут быть элементами других множеств. Для решения этой проблемы в Python существует еще один тип множества `frozenset`, который ведет себя как множество, но не может изменяться после создания. Поскольку фиксированные множества обладают свойствами неизменяемости и хешируемости, они могут быть элементами других множеств:

```
>>> x = set([1, 2, 3, 1, 3, 5])
>>> z = frozenset(x)
>>> z
frozenset({1, 2, 3, 5})
>>> z.add(6)
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    z.add(6)
AttributeError: 'frozenset' object has no attribute 'add'
>>> x.add(z)
>>> x
{1, 2, 3, 5, frozenset({1, 2, 3, 5})}
```

БЫСТРАЯ ПРОВЕРКА: МНОЖЕСТВА

Если бы вам потребовалось построить множество на базе следующего списка, то сколько элементов будет содержать это множество: `[1, 2, 5, 1, 0, 2, 3, 1, 1, (1, 2, 3)]`?

ПРАКТИЧЕСКАЯ РАБОТА 5: АНАЛИЗ СПИСКА

В этой лабораторной работе ваша задача — прочитать из файла множество температурных данных (ежемесячные температурные максимумы аэропорта Хитроу с 1948 по 2016 год), а затем вычислить некоторые статистические характеристики: максимальной и минимальной температуры, средней температуры

и медианной температуры (то есть температуры, которая будет занимать центральную позицию при сортировке температур).

Температурные данные находятся в файле lab_05.txt в каталоге исходного кода этой главы. Так как чтение файлов еще не рассматривалось, приведу готовый код чтения файла в список:

```
temperatures = []
with open('lab_05.txt') as infile:
     for row in infile:
        temperatures.append(int(row.strip())
```

Определите самую высокую и самую низкую температуру, среднюю и медианную температуру. Вероятно, вам для этого понадобятся функции/методы `min()`, `max()`, `sum()`, `len()` и `sort()`.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ

Определите, сколько уникальных температур содержит список.

Итоги

- Списки и кортежи — структуры, воплощающие идею последовательности элементов (например, строки).
- Списки похожи на массивы других языков программирования, но они поддерживают автоматическое изменение размеров, синтаксис сегментов и различные вспомогательные функции.
- Кортежи похожи на списки, но они не могут изменяться, поэтому они расходуют меньше памяти и могут использоваться в качестве ключей словаря (глава 7).
- Множества представляют собой итерируемые коллекции, но они не упорядочены и не могут содержать повторяющиеся элементы.

6 Строки

Эта глава охватывает следующие темы:

- ✓ Строки как последовательности символов
- ✓ Использование основных операций со строками
- ✓ Вставка специальных символов и экранированных последовательностей
- ✓ Преобразование из объектов в строки
- ✓ Форматирование строк
- ✓ Использование типа byte

Обработка текста — от пользовательского ввода и имен файлов до фрагментов текста — относится к числу типичных задач программирования. В языке Python реализованы мощные инструменты обработки и форматирования текста. В этой главе рассматриваются стандартные строки и строковые операции в Python.

6.1. Строки как последовательности символов

В контексте извлечения символов и подстрок строки могут рассматриваться как последовательности символов, что означает, что с ними можно использовать индексы и синтаксис сегментов:

```
>>> x = "Hello"
>>> x[0]
'H'
>>> x[-1]
'o'
>>> x[1:]
'ello'
```

В частности, синтаксис сегментов часто применяется для отсечения символа новой строки в конце строки (обычно только что прочитанной из файла):

```
>>> x = "Goodbye\n"
>>> x = x[:-1]
>>> x
'Goodbye'
```

Этот код приведен всего лишь для примера. У строк Python есть другие, более удобные методы отсечения нежелательных символов, а этот пример демонстрирует полезность сегментов.

Чтобы определить, сколько символов содержит строка, можно воспользоваться функцией `len` — точно так же, как для определения количества элементов в списке:

```
>>> len("Goodbye")
7
```

Однако строки не являются списками символов. Самое принципиальное различие между строками и символами заключается в том, что, в отличие от списков, *строки не могут изменяться*. При попытке использовать выражение вида `string.append('c')` или `string[0] = 'H'` произойдет ошибка. Из предыдущего примера видно, что для отсечения символа новой строки создается строка, которая является сегментом предыдущей, а не прямой модификацией предыдущей строки. Это базовое ограничение Python, установленное по соображениям эффективности.

6.2. Основные операции со строками

Простейший (и пожалуй, самый распространенный) способ объединения строк Python основан на использовании оператора конкатенации строк `+`:

```
>>> x = "Hello " + "World"
>>> x
'Hello World'
```

В Python также существует аналогичный оператор умножения строк, который иногда (но не так часто) оказывается полезным:

```
>>> 8 * "x"
'xxxxxxxx'
```

6.3. Специальные символы и экранированные последовательности

Вы уже видели некоторые последовательности, которые Python особым образом интерпретирует при работе со строками: `\n` представляет символ новой строки, а `\t` — символ табуляции. Последовательности символов, начинающиеся с обратного слеша и используемые для представления других символов, называются *экранированными последовательностями*. Экранированные последовательности обычно используются для представления *специальных символов*, то есть символов (например, табуляций и новых строк), не имеющих стандартного односимвольного

печатного представления. В этом разделе экранированные последовательности, специальные символы и сопутствующие темы будут рассмотрены более подробно.

6.3.1. Основные экранированные последовательности

Python предоставляет короткий перечень двухсимвольных экранированных последовательностей для использования в строках (табл. 6.1). Эти последовательности также могут использоваться с объектами байтовых строк, которые будут представлены в конце главы.

Таблица 6.1. Экранированные последовательности для строковых и байтовых литералов

Последовательность	Представляемый символ
`\'`	Одинарная кавычка
`\"`	Двойная кавычка
`\\`	Обратный слеш
`\a`	Звуковой сигнал
`\b`	Backspace
`\f`	Прогон страницы
`\n`	Новая строка
`\r`	Возврат курсора (не то же, что `\n`)
`\t`	Табуляция
`\v`	Вертикальная табуляция

Набор символов ASCII, используемый Python и являющийся стандартным почти на всех компьютерах, определяет еще несколько специальных символов. Для обращения к этим символам используются числовые экранированные последовательности, описанные в следующем разделе.

6.3.2. Числовые экранированные последовательности (восьмеричные и шестнадцатеричные) и Юникод

Вы можете включить в строку любой ASCII-символ, указав восьмеричную (основание 8) или шестнадцатеричную (основание 16) экранированную последовательность, соответствующую этому символу. Восьмеричная экранированная последовательность состоит из обратного слеша, за которым следуют три цифры, определяющие восьмеричное число; восьмеричная последовательность заменяется ASCII-символом, соответствующим этому восьмеричному коду. Шестнадцатеричная экранированная последовательность начинается с `\x` вместо `\` и может содержать произвольное количество шестнадцатеричных цифр. Экранированная последовательность завершается с обнаружением символа, который не является шестнадцатеричной цифрой. Например, в таблице ASCII-символов символу *m* соответствует десятичное значение 109, которое соответствует восьмеричному значению 155 и шестнадцатеричному значению 6D:

```
>>> 'm'
'm'
>>> '\155'
'm'
>>> '\x6D'
'm'
```

В результате обработки всех трех выражений будет получена строка, содержащая один символ *m*. Тем не менее эти формы также могут использоваться для вставки символов, не имеющих печатного представления. Например, символу новой строки `\n` соответствует восьмеричное значение 012 и шестнадцатеричное значение 0A:

```
>>> '\n'
'\n'
>>> '\012'
'\n'
>>> '\x0A'
'\n'
```

Так как все строки в Python 3 являются строками Юникода, они могут содержать практически любые символы из любых существующих языков. Обсуждение системы Юникод выходит далеко за рамки книги, но следующие примеры показывают, что экранирование может применяться к любому символу Юникода — по числовому коду (как показано выше) или по имени символа в Юникоде:

```
>>> unicode_a ='\N{LATIN SMALL LETTER A}'  <-- Экранирование по имени в Юникоде
>>> unicode_a
'a' ❶
>>> unicode_a_with_acute = '\N{LATIN SMALL LETTER A WITH ACUTE}'
>>> unicode_a_with_acute
'á'
>>> "\u00E1"  <-- Экранирование по коду символа с \u
'á'
>>>
```

В набор символов Юникода входят все стандартные ASCII-символы ❶.

6.3.3. Вывод и обработка строк со специальными символами

Ранее я уже говорила о различиях между интерактивным вычислением результата выражения Python и выводом результата того же выражения функцией `print`. И хотя в обоих случаях используется одна и та же строка, две операции могут выдавать на экран результаты, которые выглядят по-разному. Строка, которая обрабатывается на верхнем уровне интерактивного сеанса Python, выводится со всеми специальными символами в виде восьмеричных экранированных последовательностей, которые ясно показывают, что содержит строка. С другой стороны, функция `print` передает строку прямо программе терминала, которая может интерпретировать специальные символы особым образом. Вот что произойдет со строкой, состоящей из буквы `a`, за которой следует символ новой строки, табуляция и `b`:

```
>>> 'a\n\tb'
'a\n\tb'
>>> print('a\n\tb')
a
    b
```

В первом случае символы новой строки и табуляции явно выводятся в строке; во втором случае они интерпретируются как управляющие коды новой строки и табуляции.

Обычная функция `print` также добавляет символ новой строки в конец выводимого текста. В некоторых случаях (например, при работе со строками из файла, которые уже содержат завершающие символы новой строки) такое поведение может оказаться нежелательным. Если передать функции `print` параметр `end` со значением `""`, символ новой строки присоединяться не будет:

```
>>> print("abc\n")
abc

>>> print("abc\n", end="")
abc
>>>
```

6.4. Методы строк

Многие методы строк Python встроены в стандартный класс строк Python, поэтому они автоматически поддерживаются всеми объектами строк. Стандартный модуль `string` также содержит некоторые полезные константы. Модули более подробно рассматриваются в главе 10.

Для понимания этого раздела вам достаточно запомнить, что многие методы строк соединяются с объектом строки, с которым они работают, точкой (`.`) — например, `x.upper()`. Иначе говоря, перед именем метода указывается объект строки, за которым следует точка. Так как строки являются неизменяемыми, методы строк используются только для получения своего возвращаемого значения, они никак не изменяют объект строки, с которым они связаны.

Начнем с самых полезных и распространенных операций со строками; затем будут описаны некоторые реже встречающиеся, но все равно полезные операции. В конце этого раздела будут рассмотрены некоторые нюансы, связанные со строками. Не все методы строк документированы в этом разделе. За полным списком методов строк обращайтесь к документации.

6.4.1. Методы split и join

Любой разработчик, которому доводилось работать со строками, наверняка посчитает методы `split` и `join` бесценными. По сути они противоположны друг другу: `split` возвращает список подстрок, а `join` берет список строк и объединяет их в одну строку, вставляя исходную строку между каждой парой элементов. Обычно `split`

использует символы-пропуски в качестве разделителя при разбиении, но это поведение можно изменить при помощи необязательного аргумента.

Конкатенация строк оператором + полезна, но она неэффективна для объединения большого количества строк в одну строку, потому что при каждом применении + создается новый объект строки. В предыдущем примере «Hello, World» создаются три строковых объекта, один из которых будет немедленно отброшен. Лучше воспользоваться функцией `join`:

```
>>> " ".join(["join", "puts", "spaces", "between", "elements"])
'join puts spaces between elements'
```

Изменяя строку, используемую для вызова `join`, можно разместить между объединяемыми строками практически любые символы:

```
>>> "::".join(["Separated", "with", "colons"])
'Separated::with::colons'
```

Для объединения элементов списка может использоваться даже пустая строка:

```
>>> "".join(["Separated", "by", "nothing"])
'Separatedbynothing'
```

Пожалуй, на практике метод `split` чаще всего используется в качестве простого механизма разбора разделенных данных, хранящихся в текстовых файлах. По умолчанию `split` осуществляет разбивку по пропускам, а не по одиночным пробелам, но вы также можете приказать методу разбивать текст по конкретной последовательности символов, которая передается в необязательном аргументе:

```
>>> x = "You\t\t can have tabs\t\n \t and newlines \n\n " \
               "mixed in"
>>> x.split()
['You', 'can', 'have', 'tabs', 'and', 'newlines', 'mixed', 'in']
>>> x = "Mississippi"
>>> x.split("ss")
['Mi', 'i', 'ippi']
```

Иногда бывает полезно разрешить, чтобы последнее поле объединенной строки содержало произвольный текст — возможно, даже с подстроками, которые могли бы совпасть с разделителями при чтении этих данных. Для этого можно указать, сколько разбиений должен выполнить метод `split` при генерировании результатов при помощи второго необязательного аргумента. Если передать в этом аргументе n, метод `split` будет обрабатывать входную строку, пока не проведет n разбиений (и сгенерирует список, элементами которого являются n+1 подстрок) или пока не кончится строка. Несколько примеров:

```
>>> x = 'a b c d'
>>> x.split(' ', 1)
['a', 'b c d']
>>> x.split(' ', 2)
['a', 'b', 'c d']
>>> x.split(' ', 9)
['a', 'b', 'c', 'd']
```

При использовании `split` со вторым необязательным аргументом необходимо задать первый аргумент. Чтобы метод `split` выполнял разбиение по пропускам и при этом использовал второй аргумент, передайте `None` в первом аргументе.

Я очень часто использую методы `split` и `join` — чаще всего при работе с текстовыми файлами, сгенерированными другими программами. Если вы захотите генерировать более стандартные выходные файлы в своих программах, хорошими кандидатами станут модули `csv` и `json` из стандартной библиотеки Python.

БЫСТРАЯ ПРОВЕРКА: SPLIT И JOIN

Как использовать методы `split` и `join` для замены всех пропусков в строке `x` дефисами — например, преобразовать `"this is a test"` в `"this-is-a-test"`?

6.4.2. Преобразование строк в числа

Функции `int` и `float` преобразуют строки в целые числа или числа с плавающей точкой соответственно. Если функциям передается строка, которая не может интерпретироваться как число заданного типа, эти функции выдают исключение `ValueError`. Исключения рассматриваются в главе 14.

Кроме того, `int` можно передать необязательный второй аргумент с основанием системы счисления, которая должна использоваться при интерпретации входной строки:

```
>>> float('123.456')
123.456
>>> float('xxyy')
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: could not convert string to float: 'xxyy'
>>> int('3333')
3333
>>> int('123.456')  <-- Целое число не может содержать точку
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: invalid literal for int() with base 10: '123.45
>>> int('10000', 8)  <-- 10000 интерпретируется как восьмеричное число
4096
>>> int('101', 2)
5
>>> int('ff', 16)
255
>>> int('123456', 6)              <-- 123456 не может интерпретироваться как число
Traceback (innermost last):          в шестеричной системе счисления
 File "<stdin>", line 1, in ?
ValueError: invalid literal for int() with base 6: '123456'
```

А вы поняли причину последней ошибки? Я затребовала, чтобы строка интерпретировалась как число в системе счисления с основанием 6, но цифра 6 в шестеричном числе присутствовать не может. Хитро!

БЫСТРАЯ ПРОВЕРКА: ПРЕОБРАЗОВАНИЕ СТРОК В ЧИСЛА

Какая из следующих строк не будет преобразована в число и почему?

```
int('a1')
int('12G', 16)
float("12345678901234567890")
int("12*2")
```

6.4.3. Удаление лишних пропусков

На удивление полезна тройка простых методов: `strip`, `lstrip` и `rstrip`. Метод `strip` возвращает новую строку, которая получается из исходной после удаления всех пропусков в *начале и в конце* строки. Методы `lstrip` и `rstrip` работают аналогично, но они удаляют пропуски только в начале или в конце исходной строки соответственно:

```
>>> x = "  Hello,    World\t\t "
>>> x.strip()
'Hello,    World'
>>> x.lstrip()
'Hello,    World\t\t '
>>> x.rstrip()
'  Hello,    World'
```

В этом примере символы табуляции относятся к пропускам. Точное значение этого термина может зависеть от операционной системы, но вы всегда можете узнать, какие символы Python относит к категории пропусков, обратившись к константе `string.whitespace`. В моей системе Windows Python возвращает следующий результат:

```
>>> import string
>>> string.whitespace
' \t\n\r\x0b\x0c'
>>> " \t\n\r\v\f"
' \t\n\r\x0b\x0c'
```

Символы в шестнадцатеричном формате (`\xnn`) представляют символы вертикальной табуляции и прогона страницы. Пробел означает сам себя. Возможно, вам захочется изменить значение этой переменной, чтобы повлиять на работу `strip` и других методов, но делать этого не стоит. Нет никаких гарантий, что это приведет к желаемому результату.

Однако вы можете изменить состав символов, удаляемых `strip`, `rstrip` и `lstrip`, передавая константу с удаляемыми символами в дополнительном параметре:

```
>>> x = "www.python.org"
>>> x.strip("w") <-- Отсекает все буквы w
'.python.org'
>>> x.strip("gor") <-- ❶ Отсекает все буквы g, o и r
'www.python.'
>>> x.strip(".gorw") <-- Отсекает все точки, буквы g, o, r и w
'python'
```

Обратите внимание: `strip` удаляет все символы, содержащиеся в дополнительном параметре, независимо от порядка их следования ❶.

Чаще всего эти функции используются для быстрой очистки только что прочитанных строк. Они особенно полезны при чтении строк из файлов (глава 13), потому что Python всегда читает всю строку, включая завершающий символ новой строки, если он существует. Когда вы переходите к обработке прочитанных данных, завершающий символ новой строки обычно не нужен. Метод `rstrip` позволяет легко избавиться от него.

БЫСТРАЯ ПРОВЕРКА: STRIP

Если строка `x` равна `"(name, date),\n"`, какой из следующих вызовов вернет строку `"name, date"`?

```
x.rstrip("),")
x.strip("),\n")
x.strip("\n)(,")
```

6.4.4. Поиск в строках

Объекты строк предоставляют методы для выполнения простого поиска. Тем не менее, прежде чем описывать их, я хочу поговорить о другом модуле Python: `re` (этот модуль подробно рассматривается в главе 16).

ДРУГОЙ СПОСОБ ПОИСКА В СТРОКАХ: МОДУЛЬ RE

Модуль `re` также позволяет выполнять поиск в строках, но делает это намного более гибко с использованием *регулярных выражений*. Вместо того чтобы искать одну конкретную подстроку, модуль `re` проводит поиск по шаблону — например, можно искать подстроки, состоящие только из цифр.

Почему я упоминаю об этом сейчас, хотя модуль `re` будет рассматриваться позднее? По моему опыту, базовые средства поиска часто используются неподходящим образом. Разработчику стоило бы воспользоваться более мощными средствами поиска, но он не знает об их существовании и даже не ищет чего-то лучшего. Возможно, вы работаете над первоочередным проектом, в котором используются строки, и у вас нет времени читать всю книгу. Если базовые средства поиска решают вашу проблему — отлично. Но знайте, что существует и более мощная альтернатива.

Существуют четыре базовых метода поиска в строках: `find`, `rfind`, `index` и `rindex`. Сопутствующий метод `count` подсчитывает, сколько раз подстрока встречается в другой строке. Я подробно опишу метод `find`, а затем объясню, чем другие методы отличаются от него.

Метод `find` получает один обязательный аргумент: искомую подстроку. Он возвращает позицию первого символа в первом вхождении подстроки в строке или `-1`, если `подстрока` не встречается в строке:

```
>>> x = "Mississippi"
>>> x.find("ss")
```

```
2
>>> x.find("zz")
-1
```

Метод find также может получать один или два дополнительных необязательных аргумента. Первый аргумент start (если он присутствует) определяет начальную позицию поиска; он заставляет find игнорировать все символы, предшествующие позиции start, при поиске подстроки. Второй необязательный аргумент end (если он присутствует) определяет конечную позицию поиска; все символы в позиции end строки и после нее игнорируются:

```
>>> x = "Mississippi"
>>> x.find("ss", 3)
5
>>> x.find("ss", 0, 3)
-1
```

Метод rfind очень похож на find, не считая того, что он начинает поиск от конца строки и возвращает позицию первого символа последнего вхождения подстроки в строке:

```
>>> x = "Mississippi"
>>> x.rfind("ss")
5
```

Метод rfind также может получать один или два необязательных аргумента, смысл которых аналогичен аргументам find.

Методы index и rindex идентичны find и rfind соответственно, кроме одного различия: если index или rindex не могут найти вхождение подстроки в строке, метод не возвращает -1, а инициирует исключение ValueError. О том, что именно это означает, вы поймете после прочтения главы 14.

Метод count используется по тем же принципам, что и четыре предыдущие функции, но он возвращает количество неперекрывающихся вхождений заданной подстроки в строке:

```
>>> x = "Mississippi"
>>> x.count("ss")
2
```

Для поиска в строках также могут использоваться еще два метода: startswith и endswith. Эти методы возвращают результат True или False в зависимости от того, начинается ли (или заканчивается) строка, для которой они вызываются, одной из строк, переданных в параметрах:

```
>>> x = "Mississippi"
>>> x.startswith("Miss")
True
>>> x.startswith("Mist")
False
>>> x.endswith("pi")
True
>>> x.endswith("p")
False
```

Методы `startswith` и `endswith` позволяет искать более одной строки одновременно. Если параметр представляет собой кортеж строк, то оба метода проверяют все строки в кортеже и возвращают `True`, если будет найдена хотя бы одна из них:

```
>>> x.endswith(("i", "u"))
True
```

Методы `startswith` и `endswith` хорошо подходят для простого поиска, когда вы уверены в том, что именно ищется в начале или конце строки.

БЫСТРАЯ ПРОВЕРКА: ПОИСК В СТРОКАХ

Допустим, вы хотите проверить, завершается ли строка подстрокой `"rejected"`. Какой строковый метод вы для этого используете? Можно ли добиться того же результата другими способами?

6.4.5. Изменение строк

Строки являются неизменяемыми, но объекты строк поддерживают ряд методов, которые возвращают новую строку — модифицированную версию исходной строки. Такой результат, по сути, приводит к тому же результату, что и прямое изменение. За полным описанием этих методов обращайтесь к документации.

Метод `replace` может использоваться для замены вхождений подстроки `substring` (первого аргумента) другой строкой `newstring` (второй аргумент). Этот метод также получает необязательный третий аргумент (за подробностями обращайтесь к документации):

```
>>> x = "Mississippi"
>>> x.replace("ss", "+++")
'Mi+++i+++ippi'
```

Как и в случае с функциями поиска, модуль `re` предоставляет гораздо более мощные средства замены подстрок.

Функции `string.maketrans` и `string.translate` могут использоваться для преобразования символов в строках другими символами. Эти функции используются относительно редко, но в этих отдельных случаях они могут упростить вам жизнь.

Предположим, вы работаете над программой, которая преобразует строковые выражения с одного компьютерного языка на другой. В первом языке для обозначения логического отрицания используется оператор `~`, а в другом оператор `!`; в первом языке для обозначения операции «логическое ИЛИ» используется оператор `^`, а во втором `&`; в первом языке используются круглые скобки `(` и `)`, а во втором — квадратные скобки `[` и `]`. В заданном строковом выражении необходимо заменить все вхождения `~` на `!`, все вхождения `^` на `&`, все вхождения `(` на `[`, и все вхождения `)` на `]`. Это можно сделать многократными вызовами `replace`, но проще и эффективнее действовать иначе:

```
>>> x = "~x ^ (y % z)"
>>> table = x.maketrans("~^()", "!&[]")
```

```
>>> x.translate(table)
'!x & [y % z]'
```

Во второй строке метод `maketrans` создает таблицу преобразования по двум своим строковым аргументам. Два аргумента должны содержать одинаковое количество символов, а таблица устроена так, что для `n`-го символа первого аргумента она возвращает *n*-й символ второго аргумента.

Таблица, созданная вызовом `maketrans`, передается методу `translate`. Затем метод `translate` перебирает все символы своего объекта `строки` и проверяет, присутствуют ли они в таблице, заданной вторым аргументом. Если символ встречается в таблице преобразования, `translate` заменяет этот символ соответствующим преобразованным символом для получения преобразованной строки.

При вызове `translate` также можно передать необязательный аргумент с символами, которые должны быть удалены из строки. За подробностями обращайтесь к документации.

Другие функции модуля `string` решают более специализированные задачи. Метод `string.lower` преобразует все алфавитные символы строки к нижнему регистру, а метод `upper` решает обратную задачу. `capitalize` преобразует к верхнему регистру первый символ строки, а `title` делает это со всеми словами в строке. `swapcase` преобразует символы нижнего регистра одной строки к верхнему, а символы верхнего регистра к нижнему. `expandtabs` удаляет символы табуляции в строке, заменяя каждую табуляцию заданным количеством пробелов. `ljust`, `rjust` и `center` дополняют строку пробелами для ее выравнивания в поле определенной ширины.`zfill` дополняет числовую строку нулями слева. За подробной информацией обо всех этих методах обращайтесь к документации.

6.4.6. Изменение строк и операции со списками

Так как строки являются неизменяемыми объектами, вы не можете манипулировать с ними напрямую по аналогии с тем, как вы манипулируете со списками. И хотя операции, создающие новые строки (оставляющие исходные строки без изменения), полезны во многих ситуациях, иногда бывает удобно работать со строкой так, словно она представляет собой список символов. В таких ситуациях следует преобразовать строку в список символов, выполнить нужные операции, а затем преобразовать полученный список обратно в строку:

```
>>> text = "Hello, World"
>>> wordList = list(text)
>>> wordList[6:] = []  <-- Удаляет все после запятой
>>> wordList.reverse()
>>> text = "".join(wordList)
>>> print(text)  <-- Объединение без пробелов
,olleH
```

Также можно преобразовать строку в кортеж символов встроенной функцией `tuple`. Для преобразования строки обратно в список используется функция `"".join()`.

Не увлекайтесь этим методом, потому что он требует создания и уничтожения новых объектов строк, что обходится относительно дорого. Обработка сотен и тысяч строк не повлияет на вашу программу, но с обработкой миллионов строк ситуация может измениться.

БЫСТРАЯ ПРОВЕРКА: ИЗМЕНЕНИЕ СТРОК

Как быстро заменить все знаки препинания в строке пробелами?

6.4.7. Полезные методы и константы

У объектов `строк` также есть несколько полезных методов для получения разных характеристик строк — например, содержит ли строка только цифры или алфавитные символы либо только символы верхнего или нижнего регистра:

```
>>> x = "123"
>>> x.isdigit()
True
>>> x.isalpha()
False
>>> x = "M"
>>> x.islower()
False
>>> x.isupper()
True
```

За списком всех возможных методов string обращайтесь к соответствующему разделу официальной документации Python.

Наконец, модуль `string` определяет ряд полезных констант. Вы уже видели константу `string.whitespace` — строку, которая состоит из всех символов, которые Python относит к категории пропусков в вашей системе. Константа `string.digits` содержит строку `'0123456789'`. Константа `string.hexdigits` включает все символы `string.digits`, а также `'abcdefABCDEF'` — дополнительные символы, используемые в шестнадцатеричных числах. Константа `string.octdigits` содержит `'01234567'` — цифры, допустимые в восьмеричных числах. Константа `string.ascii_lowercase` содержит все алфавитные символы ASCII нижнего регистра; `string. ascii_uppercase` содержит все алфавитные символы ASCII верхнего регистра; `string. ascii_letters` содержит все символы `string. ascii_lowercase` и `string. ascii_uppercase`. Возможно, вам захочется присвоить новое значение этим константам, чтобы изменить поведение языка. Python разрешит выполнить это действие, но скорее всего, добром это не кончится.

Помните, что строки являются последовательностями символов. Это позволяет использовать удобный оператор `in` языка Python для проверки принадлежности символа к любой из этих строк, хотя обычно существующие методы строк проще и удобнее. Самые распространенные операции со строками перечислены в табл. 6.2.

Таблица 6.2. Распространенные операции со строками

Операция	Описание	Пример
`+`	Объединяет две строки	`x = "hello " + "world"`
`*`	Дублирует строку	`x = " " * 20`
`upper`	Преобразует строку к верхнему регистру	`x.upper()`
`lower`	Преобразует строку к нижнему регистру	`x.lower()`
`title`	Преобразует первую букву каждого слова в строке к верхнему регистру	`h x.title()`
`find, index`	Ищет заданную подстроку в строке	`x.find(y)` `x.index(y)`
`rfind, rindex`	Ищет заданную подстроку в строке, начиная с конца строки	`x.rfind(y)` `x.rindex(y)`
`startswith, endswith`	Проверяет начало или конец строки на совпадение с заданной подстрокой	`x.startswith(y)` `x.endswith(y)`
`replace`	Заменяет подстроку новой строкой	`x.replace(y, z)`
`strip, rstrip, lstrip`	Удаляет пропуски или другие символы в концах строки	`x.strip()`
`encode`	Преобразует строку Юникода в объект bytes	`x.encode("utf_8")`

Помните, что эти методы не изменяют саму строку; они возвращают либо позицию в строке, либо новую строку.

ПОПРОБУЙТЕ САМИ: ОПЕРАЦИИ СО СТРОКАМИ

Допустим, имеется список строк, в котором некоторые (но не обязательно все) строки начинаются и завершаются символом двойной кавычки:

```
x = ['"abc"', 'def', '"ghi"', '"klm"', 'nop']
```

Какой код вы бы использовали для удаления только двойных кавычек из каждого элемента?

Какой код вы бы использовали для нахождения позиции последней буквы `p` в слове `Mississippi`? А после того, как эта позиция будет найдена, какой код вы бы использовали для удаления только этой буквы?

6.5. Преобразование объектов в строки

В языке Python почти любой тип можно преобразовать в строковое представление вызовом встроенной функции `repr`. Списки — единственный сложный тип данных Python, который вам встречался ранее, поэтому мы преобразуем несколько списков в их строковые представления:

```
>>> repr([1, 2, 3])
'[1, 2, 3]'
```

```
>>> x = [1]
>>> x.append(2)
>>> x.append([3, 4])
>>> 'the list x is ' + repr(x)
'the list x is [1, 2, [3, 4]]'
```

Этот пример использует функцию `repr` для преобразования списка `x` в строковое представление, которое затем объединяется с другой строкой для формирования итоговой строки. Без `repr` этот код работать не будет. Что именно суммируется в выражении вида `"string" + [1, 2] + 3` — строки, списки или числа? Python не знает ваших намерений в такой ситуации, поэтому он выбирает безопасный вариант (выдает ошибку) вместо того, чтобы делать предположения. В предыдущем примере все элементы необходимо преобразовать в строковые представления, чтобы сработала конкатенация строк.

Списки — единственная разновидность сложных объектов Python, рассмотренных до настоящего момента, но `repr` может использоваться для получения строкового представления практически любых объектов Python. Чтобы убедиться в этом, попробуйте вызвать `repr` для встроенного сложного объекта — функции Python:

```
>>> repr(len)
'<built-in function len>'
```

Python не выдает строку с кодом реализации функции `len`, но по крайней мере возвращает строку — `<built-in function len>` — с описанием функции. Если вы опробуете функцию `repr` для каждого типа данных Python (словари, кортежи, классы и т. д.), упоминаемого в книге, вы увидите, что независимо от типа объекта Python вы получите строку, которая содержит некоторую информацию об объекте.

Эта возможность очень полезна для отладки программ. Если вы сомневаетесь в том, какие данные хранятся в переменной в определенной точке программы, используйте функцию `repr` и выведите содержимое этой переменной.

Итак, теперь вы знаете, как Python может преобразовать любой объект в строку с описанием этого объекта. По правде говоря, Python может сделать это двумя способами. Функция `repr` всегда возвращает то, что можно назвать *формальным строковым представлением* объекта Python. Если говорить конкретнее, `repr` возвращает строковое представление объекта Python, по которому можно восстановить исходный объект. Для больших сложных объектов это может быть не тот результат, который вам хотелось бы видеть в отладочном выводе или в отчетах состояния.

Python также предоставляет встроенную функцию `str`. В отличие от `repr`, `str` выводит *печатные* строковые представления и может применяться к любым объектам Python. `str` возвращает то, что можно назвать *неформальным строковым представлением* объекта. Строка, возвращаемая `str`, не обязана определять объект полностью; она предназначена для чтения человеком, а не кодом Python.

Когда вы начнете пользоваться `repr` и `str`, вы не заметите никаких различий между ними, потому что до того, как вы начнете пользоваться объектно-ориентированными возможностями Python, никаких различий нет. При вызове для любого встроенного объекта Python `str` всегда вызывает `repr` для получения результата. Только когда

вы начнете определять собственные классы, различия между `str` и `repr` начинают играть важную роль (глава 15).

Зачем говорить об этом сейчас? Я хочу, чтобы вы знали, что вызов `repr` делает нечто большее, чем простой отладочный вывод (`print`). Возьмите в привычку использовать `str` вместо `repr` при создании строк для вывода информации — этот вариант предпочтителен по соображениям стиля программирования.

6.6. Использование метода format

Форматирование строк в Python 3 может осуществляться двумя способами. Более новый способ основан на использовании метода `format` класса строки. Метод `format` объединяет форматную строку, содержащую поля-заменители в фигурных скобках `{ }`, со значениями, взятыми из параметров команды `format`. Если потребуется включить в строку литерал `{` или `}`, удвойте его (`{{` или `}}`). Команда `format` предоставляет мощный мини-язык форматирования строк, дающий почти бесконечные возможности для управления форматированием строк. С другой стороны, в большинстве стандартных ситуаций он достаточно прост в использовании, поэтому в этом разделе будут рассмотрены некоторые шаблоны. Если же вам потребуются нетривиальные возможности, обращайтесь к разделу, посвященному форматированию строк, в документации стандартной библиотеки.

6.6.1. Метод format и позиционные параметры

Простой способ использования строкового метода `format` связан с заменой нумерованных полей, соответствующих параметрам, переданным функции `format`:

```
>>> "{0} is the {1} of {2}".format("Ambrosia", "food", "the gods") ❶
'Ambrosia is the food of the gods'
>>> "{{Ambrosia}} is the {0} of {1}".format("food", "the gods") ❷
'{Ambrosia} is the food of the gods'
```

Следует заметить, что метод `format` применяется к форматной строке, которая также может быть строковой переменной ❶. Символы `{ }` экранируются удваиванием, чтобы они не интерпретировались как признак поля-заменителя ❷.

Пример содержит три поля-заменителя — `{0}`, `{1}` и `{2}`, — которые последовательно заменяются первым, вторым и третьим параметрами. Где бы в форматной строке ни размещалось поле `{0}`, оно всегда замещается первым параметром, и т. д.

Также можно использовать именованные параметры.

6.6.2. Метод format и именованные параметры

Метод `format` также распознает именованные параметры и поля замены:

```
>>> "{food} is the food of {user}".format(food="Ambrosia",
...       user="the gods")
'Ambrosia is the food of the gods'
```

В этом случае параметр выбирается по совпадению имени поля-заменителя с именем параметра, переданного команде `format`.

Допускается использование позиционных параметров вместе с именованными; в этих параметрах можно даже обращаться к атрибутам и элементам:

```
>>> "{0} is the food of {user[1]}".format("Ambrosia",
...          user=["men", "the gods", "others"])
'Ambrosia is the food of the gods'
```

В данном случае первый параметр является позиционным, тогда как обозначение `user[1]` относится ко второму элементу именованного параметра `user`.

6.6.3. Спецификаторы формата

Спецификаторы формата позволяют задать результат форматирования с еще большей точностью и широтой возможностей, чем форматные последовательности в старом стиле форматирования строк. Спецификатор формата позволяет задать символ-заполнитель, тип выравнивания, знак, ширину, точность и тип данных при подстановке на место поля-заменителя. Как упоминалось ранее, синтаксис спецификаторов формата представляет отдельный мини-язык, слишком сложный для того, чтобы полностью описывать его здесь. Тем не менее несколько примеров дадут вам представление о его возможностях:

```
>>> "{0:10} is the food of gods".format("Ambrosia") ❶
'Ambrosia   is the food of gods'
>>> "{0:{1}} is the food of gods".format("Ambrosia", 10) ❷
'Ambrosia   is the food of gods'
>>> "{food:{width}} is the food of gods".format(food="Ambrosia", width=10)
'Ambrosia   is the food of gods'
>>> "{0:>10} is the food of gods".format("Ambrosia") ❸
'  Ambrosia is the food of gods'
>>> "{0:&>10} is the food of gods".format("Ambrosia") ❹
'&&Ambrosia is the food of gods'
```

`:10` — спецификатор, который определяет поле шириной в 10 пробелов, дополняемое пробелами ❶. `:{1}` получает ширину из второго параметра ❷. `:>10` включает выравнивание поля по правому краю с дополнением пробелами ❸. `:&>10` включает выравнивание по правому краю с дополнением символами `&` вместо пробелов ❹.

БЫСТРАЯ ПРОВЕРКА: МЕТОД FORMAT()

Что будет содержать переменная `x` при выполнении следующих фрагментов кода?

```
x = "{1:{0}}".format(3, 4)
x = "{0:$>5}".format(3)
x = "{a:{b}}".format(a=1, b=5)
x = "{a:{b}}:{0:$>5}".format(3, 4, a=1, b=5, c=10)
```

6.7. Форматирование строк с символом %

В этом разделе рассматривается форматирование строк с использованием оператора %. Этот оператор используется для объединения значений Python в отформатированные строки для печати или иного применения. Пользователи C отметят неожиданное сходство с семейством функций printf. Применение % для форматирования строк относится к старому стилю форматирования, но я рассматриваю его здесь, потому что этот стиль считался стандартным в предыдущих версиях Python. Вы с большой вероятностью увидите его в коде, портированном из более ранних версий Python или написанном программистами, знакомыми с этими версиями. В новом коде этот стиль форматирования не должен использоваться, потому что он обречен на вымирание и в будущем будет удален из языка. Пример:

```
>>> "%s is the %s of %s" % ("Ambrosia", "food", "the gods")
'Ambrosia is the food of the gods'
```

Строковый оператор % (выделенный жирным шрифтом знак % в середине, а не три предшествующих экземпляра %s) работает с двумя частями: в левой части размещается строка, а в правой кортеж. Оператор % ищет в левой строке специальные *форматные последовательности* и строит новую строку, заменяя эти форматные последовательности значениями из правой части. В этом примере форматными последовательностями в левой части являются три экземпляра %s, которые означают «Здесь вставляется строка».

При передаче разных значений в правой части будут получены разные строки:

```
>>> "%s is the %s of %s" % ("Nectar", "drink", "gods")
'Nectar is the drink of gods'
>>> "%s is the %s of the %s" % ("Brussels Sprouts", "food",
...        "foolish")
'Brussels Sprouts is the food of the foolish'
```

К элементам кортежа в правой части, соответствующим спецификаторам %s, автоматически применяется str, так что они не обязаны быть строками:

```
>>> x = [1, 2, "three"]
>>> "The %s contains: %s" % ("list", x)
"The list contains: [1, 2, 'three']"
```

6.7.1. Использование форматных последовательностей

Все форматные последовательности представляют собой подстроки, содержащиеся в строке в левой части от центрального оператора %. Каждая форматная последовательность начинается со знака %, и за ней следует один или несколько символов, которые показывают, что должно быть подставлено на место форматной последовательности и как должна выполняться подстановка. Форматная последовательность %s, использованная выше, является простейшей форматной последовательностью, она означает, что на место %s должна быть подставлена соответствующая строка из кортежа в правой части.

Возможны и другие, более сложные форматные последовательности. Следующая последовательность задает ширину поля (общее количество символов) выводимого числа равной 6, задает количество символов в дробной части равным 2 и выравнивает число по левому краю в пределах поля. В следующем примере эта форматная последовательность заключается в угловые скобки, чтобы вы видели, где в форматной строке вставляются дополнительные пробелы:

```
>>> "Pi is <%-6.2f>" % 3.14159 # Форматная последовательность %–6.2f
'Pi is <3.14  >'
```

Полное описание символов, допустимых в форматных последовательностях, приводится в документации. Вариантов достаточно много, но все они относительно просты в использовании. Помните, что форматную последовательность всегда можно протестировать в интерактивном режиме Python; это поможет вам понять, работает ли она так, как вы ожидали.

6.7.2. Именованные параметры и форматные последовательности

Наконец, у оператора `%` есть еще одна дополнительная возможность, которая может оказаться полезной в определенных обстоятельствах. К сожалению, чтобы описать ее, мне придется воспользоваться возможностью Python, которая еще подробно не рассматривалась, — *словарями*, которые в других языках программирования часто называются *хеш-таблицами* или *ассоциативными массивами*. Словари рассматриваются в главе 7. Вы можете пропустить этот раздел и вернуться к нему позднее или же читать дальше в расчете на то, что из примеров все станет ясно.

Форматные последовательности могут определять подставляемое значение по имени, а не по позиции. В этом случае сразу же после знака `%` в форматной последовательности указывается имя, заключенное в круглые скобки:

```
"%(pi).2f"  ◄——  Обратите внимание: имя заключается в круглые скобки
```

Кроме того, в аргументе справа от оператора `%` указывается уже не отдельное значение или кортеж значений для вывода, а словарь, в котором для каждой именованной форматной последовательности содержится ключ с соответствующим именем. Так, с приведенной выше форматной последовательностью код может выглядеть так:

```
>>> num_dict = {'e': 2.718, 'pi': 3.14159}
>>> print("%(pi).2f - %(pi).4f - %(e).2f" % num_dict)
3.14 - 3.1416 - 2.72
```

Такой код особенно удобен при использовании форматных строк с большим количеством подстановок, потому что вам не приходится следить за позиционным соответствием кортежа в правой части и форматных последовательностей в форматной строке. Порядок определения элементов в аргументе `dict` не важен, а строка шаблона может использовать значения из `dict` более одного раза (как это делается выше для элемента `'pi'`).

УПРАВЛЕНИЕ ВЫВОДОМ С ФУНКЦИЕЙ PRINT

Встроенная функция Python `print` также поддерживает некоторые возможности, упрощающие простой строковый вывод. С одним параметром print выводит значение и символ новой строки, поэтому следующая серия вызовов `print` выводит каждое значение в отдельной строке:

```
>>> print("a")
a
>>> print("b")
b
```

Однако функция `print` способна на большее. Ей можно передать несколько аргументов, и эти аргументы будут выведены в одной строке, разделенные пробелами и завершенные символом новой строки:

```
>>> print("a", "b", "c")
a b c
```

Если это не совсем то, что вам нужно, передайте функции `print` дополнительные параметры, которые задают разделители элементов и завершитель строки:

```
>>> print("a", "b", "c", sep="|")
a|b|c
>>> print("a", "b", "c", end="\n\n")
a b c
>>>
```

Наконец, функция `print` может использоваться для вывода как в файлы, так и на консоль.

```
>>> print("a", "b", "c", file=open("testfile.txt", "w"))
```

Возможности функции print достаточны для простого вывода текста, но в более сложных ситуациях лучше воспользоваться методом `format`.

БЫСТРАЯ ПРОВЕРКА: ФОРМАТИРОВАНИЕ СТРОК С СИМВОЛОМ %

Что будет содержать переменная `x` при выполнении следующих фрагментов кода?

```
x = "%.2f" % 1.1111
x = "%(a).2f" % {'a':1.1111}
x = "%(a).08f" % {'a':1.1111}
```

6.8. Строковая интерполяция

В Python 3.6 появился механизм создания строковых констант, содержащих произвольные значения; он называется *строковой интерполяцией*. Строковая интерполяция позволяет включать значения выражений Python в строковые литералы. Эти f-строки (как они часто называются из-за префикса `f`) используют синтаксис, сходный с синтаксисом метода format, но с несколько большей эффективностью. Следующие примеры дают некоторое представление о том, как работают `f`-строки:

```
>>> value = 42
>>> message = f"The answer is {value}"
>>> print(message)
The answer is 42
```

Как и в случае с методом format, можно добавить спецификаторы формата:

```
>>> pi = 3.1415
>>> print(f"pi is {pi:{10}.{2}}")
pi is        3.1
```

Так как строковая интерполяция появилась в Python совсем недавно, пока неясно, как она будет использоваться. За полной документацией по f-строкам и форматным спецификаторам обращайтесь к PEP-498 в электронной документации Python.

6.9. Байтовые строки

Объект байтовой строки `bytes` напоминает объект строки `string` с одним важным различием: `строка` является неизменяемой последовательностью символов Юникода, тогда как объект `bytes` является последовательностью целых чисел со значениями от 0 до 256. Байтовые строки могут пригодиться при работе с двоичными данными, например при чтении из двоичного файла.

Главное — запомнить, что объекты `bytes` внешне похожи на строки, но они не могут использоваться точно так же, как строки, или объединяться с ними:

```
>>> unicode_a_with_acute = '\N{LATIN SMALL LETTER A WITH ACUTE}'
>>> unicode_a_with_acute
'á'
>>> xb = unicode_a_with_acute.encode() ❶
>>> xb
b'\xc3\xa1' ❷
>>> xb += 'A' ❸
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    xb += 'A'
TypeError: can't concat str to bytes
>>> xb.decode() ❹
'á'
```

Первое, что видно из этого фрагмента, — для преобразования обычной строки (в Юникоде) в `bytes` необходимо вызвать метод `encode` для строки ❶. После того как строка будет закодирована в объект `bytes`, символ занимает 2 байта и выводится не так, как выводился в строковом виде ❷. Более того, при попытке просуммировать объект `bytes` с объектом строки вы получите сообщение об ошибке типа из-за несовместимости двух типов ❸. Наконец, чтобы преобразовать объект `bytes` обратно в строку, необходимо вызвать метод `decode` этого объекта ❹.

В большинстве случаев вам вообще не придется задумываться о Юникоде и байтах. Но в тех случаях, когда вы работаете с международными наборами символов (а сейчас эта задача встречается все чаще), необходимо понимать различия между обычными строками и `bytes`.

БЫСТРАЯ ПРОВЕРКА: БАЙТОВЫЕ СТРОКИ

Для каких из следующих разновидностей данных вы бы использовали обычные строки? В каких случаях можно использовать байтовые строки?

❶ Файл данных с двоичными данными.

❷ Текст на языке, содержащем символы с диакритическими знаками.

❸ Текст, состоящий только из букв латинского алфавита в верхнем и нижнем регистре.

❹ Серия целых чисел, не превышающих 255.

ПРАКТИЧЕСКАЯ РАБОТА 6: ПРЕДВАРИТЕЛЬНАЯ ОБРАБОТКА ТЕКСТА

При обработке текста часто требуется почистить и нормализовать текст перед тем, как делать с ним что-то еще. Например, если вы хотите подсчитать количество вхождений слов в тексте, для упрощения задачи перед началом подсчета можно позаботиться о том, чтобы весь текст был записан в нижнем регистре (или в верхнем, если предпочитаете) и из него были удалены все знаки препинания. Также для упрощения задачи можно разбить текст на серии слов. В этой практической работе вы должны прочитать первую часть первой главы «Моби Дика» (присутствует в исходном коде книги), позаботиться о том, чтобы все символы относились к одному регистру, удалить все знаки препинания и записать слова по одному на строку во второй файл. Так как операции чтения и записи файлов в книге еще не рассматривались, я приведу код для выполнения этих операций:

```
with open("moby_01.txt") as infile, open("moby_01_clean.txt", "w") as outfile:
    for line in infile:
        # Привести к одному регистру
        # Удалить знаки препинания
        # Разбить на слова
        # Записать все слова по одному на строку файла
        outfile.write(cleaned_words)
```

Итоги

- Строки Python поддерживают мощные средства обработки текста, включая поиск и замену, отсечение символов и изменение регистра.
- Строки являются неизменяемыми, то есть они не могут изменяться «на месте».
- Операции, которые на первый взгляд изменяют строки, в действительности возвращают копию с изменениями.
- Модуль `re` содержит еще более мощные средства работы со строками (регулярные выражения), которые будут рассмотрены в главе 16.

7 Словари

Эта глава охватывает следующие темы:

- ✓ Определение словаря
- ✓ Использование операций со словарем
- ✓ Определение того, что можно использовать в качестве ключа
- ✓ Создание разреженных матриц
- ✓ Использование словарей в качестве кэшей
- ✓ Доверие к эффективности словарей

В этой главе рассматриваются словари — этим термином в Python обозначаются ассоциативные массивы или карты, реализуемые на базе хеш-таблиц. Словари чрезвычайно полезны даже в простых программах.

Поскольку словари знакомы многим программистам в меньшей степени, чем другие базовые структуры данных (такие, как списки и строки), некоторые примеры этой главы получились чуть сложнее соответствующих примеров для других встроенных структур данных. Возможно, для полного понимания примеров этой главы вам стоит прочитать отдельные части главы 8.

7.1. Что такое словарь?

Если вы никогда не работали с ассоциативными массивами или хеш-таблицами в других языках, словари можно сравнить со списками:

- Для обращения к значениям в словарях используются целые числа, называемые *индексами*. Они определяют позицию списка, в которой находится заданное значение.
- У словарей для обращения к значениям могут использоваться целые числа, строки или другие объекты Python, называемые *ключами*. Другими словами, и списки и словари предоставляют индексируемый доступ к произвольным

значениям, но множество элементов, которые могут использоваться в качестве индексов словарей, намного больше множества значений, которые могут использоваться в качестве индексов списков. Кроме того, механизм, используемый словарями для индексирования, сильно отличается от аналогичного механизма списков.

- Как в списках, так и в словарях могут храниться объекты любого типа.
- Значения, хранящиеся в списке, неявно *упорядочиваются* по своей позиции, потому что индексы, используемые для обращения к ним, представляют собой целые числа. Возможно, порядок элементов для вас неважен, но при желании его можно использовать. Для значений, хранящихся в словаре, *неявный* порядок относительно друг друга не определен, потому что ключи не ограничиваются числами. Если вы работаете со словарем, но вас также интересует порядок элементов (то есть порядок их добавления), используйте *упорядоченный словарь* — субкласс словаря, который можно импортировать из модуля `collections`. Также для элементов словаря можно определить порядок при помощи другой структуры данных (часто это список), в которой порядковая информация хранится в явном виде; это никак не отменяет того факта, что у базовых словарей неявный (встроенный) порядок не определен.

Несмотря на все различия, операции со словарями и списками часто выглядят одинаково. Сначала программа создает пустой словарь почти так же, как пустой список, но вместо квадратных скобок используются круглые:

```
>>> x = []
>>> y = {}
```

Первая строка создает новый пустой список и присваивает его переменной `x`. Вторая строка создает новый пустой словарь и присваивает его переменной `y`.

После того как словарь будет создан, в нем можно сохранять значения так, как если бы это был список:

```
>>> y[0] = 'Hello'
>>> y[1] = 'Goodbye'
```

Даже в этих командах присваивания проявляются принципиальные различия между практическим использованием словарей и списков. Попытка проделать то же самое со списком приведет к ошибке, потому что в Python присваивание в несуществующей позиции списка недопустимо. Например, при попытке присвоить значение *0*-му элементу списка `x` произойдет ошибка:

```
>>> x[0] = 'Hello'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

Со словарями такой проблемы нет; новые позиции в словаре создаются по мере необходимости.

После того как в словаре будут сохранены значения, программа сможет обращаться и работать с ними:

```
>>> print(y[0])
Hello
>>> y[1] + ", Friend."
'Goodbye, Friend.'
```

И все же на первый взгляд словари кажутся похожими на списки. Теперь перейдем к серьезным различиям. Словари позволяют связать значения с ключами, которые не являются целыми числами:

```
>>> y["two"] = 2
>>> y["pi"] = 3.14
>>> y["two"] * y["pi"]
6.28
```

Со списками такое определенно невозможно! Если индексы списков могут быть только целыми числами, ключи словарей не ограничиваются; это могут быть числа, строки или один из объектов широчайшего диапазона Python. Таким образом словари становятся естественными кандидатами для тех задач, которые не под силу спискам. Например, приложение телефонного справочника удобнее реализовать на базе словарей, чем на базе списков, потому что телефонный номер может индексироваться по фамилии абонента.

Словарь позволяет отобразить одно множество произвольных объектов на другое — логически связанное, но столь же произвольное множество объектов. Хорошим аналогом словарей Python служат реальные словари или алфавитные справочники. Чтобы увидеть, насколько естественно выглядит это соответствие, возьмем простейший переводчик названий цветов с английского языка на французский:

```
>>> english_to_french = {}   <-- Создает пустой словарь
>>> english_to_french['red'] = 'rouge'   <-- Сохраняет в нем три слова
>>> english_to_french['blue'] = 'bleu'
>>> english_to_french['green'] = 'vert'
>>> print("red is", english_to_french['red'])   C   <-- Получает значение для ключа 'red'
red is rouge
```

ПОПРОБУЙТЕ САМИ: СОЗДАНИЕ СЛОВАРЯ

Напишите код, который запрашивает у пользователя три имени и три возраста. После ввода имен и возрастов запросите у пользователя одно из имен и выведите соответствующий возраст.

7.2. Другие операции со словарями

Кроме простого присваивания и обращения к элементам, словари поддерживают и другие операции. Словарь также можно явно определить как серию пар «ключ–значение», разделенных запятыми:

```
>>> english_to_french = {'red': 'rouge', 'blue': 'bleu', 'green': 'vert'}
```

Функция `len` возвращает количество элементов в словаре:

```
>>> len(english_to_french)
3
```

Для получения всех ключей в словаре используется метод `keys`. Этот метод часто используется для перебора содержимого словаря в цикле Python `for` (глава 8):

```
>>> list(english_to_french.keys())
['green', 'blue', 'red']
```

В Python 3.5 и более ранних версиях порядок ключей в списке, возвращаемом `keys`, не имеет смысла, ключи не обязательно отсортированы и они не обязательно следуют в порядке их создания. На вашем компьютере код Python может вывести ключи в порядке, отличном от порядка в моем примере. Если вам нужно, чтобы ключи были отсортированы, сохраните их в списковой переменной и отсортируйте полученный список. Однако начиная с Python 3.6, словари сохраняют порядок создания ключей и возвращают их в этом порядке.

Также возможно получить все значения, хранящиеся в словаре, методом `values`:

```
>>> list(english_to_french.values())
 ['vert', 'bleu', 'rouge']
```

Этот метод используется намного реже, чем метод `keys`.

Метод `items` возвращает все ключи и связанные с ними значения в виде последовательности кортежей:

```
>>> list(english_to_french.items())
[('green', 'vert'), ('blue', 'bleu'), ('red', 'rouge')]
```

Как и метод `keys`, этот метод часто используется в циклах `for` для перебора содержимого словаря.

Команда `del` может использоваться для удаления элементов (пар «ключ–значение») из словаря:

```
>>> list(english_to_french.items())
[('green', 'vert'), ('blue', 'bleu'), ('red', 'rouge')]
>>> del english_to_french['green']
>>> list(english_to_french.items())
 [('blue', 'bleu'), ('red', 'rouge')]
```

ОБЪЕКТЫ ПРЕДСТАВЛЕНИЯ СЛОВАРЯ

Методы `keys`, `values` и `items` возвращают не списки, а представления (*views*); они ведут себя как последовательности, которые динамически обновляются при изменении словаря. Вот почему в этих примерах нам приходится использовать функцию `list` для того, чтобы интерпретировать их как списки. В остальном они ведут себя как последовательности, что позволяет программам перебирать их в циклах `for`, использовать `in` для проверки принадлежности и т. д.

Представление, возвращаемое методом `keys` (а в некоторых случаях представление, возвращаемое методом `items`), также ведет себя как множество с операциями объединения, пересечения и разности.

Попытки обратиться к ключу, отсутствующему в словаре, приводит к ошибке в Python. Чтобы избежать этой ошибки, можно проверить словарь на присутствие ключа оператором `in`, который возвращает `True`, если в словаре есть значение, ассоциированное с заданным ключом, и `False` в противном случае:

```
>>> 'red' in english_to_french
True
>>> 'orange' in english_to_french
False
```

Также можно воспользоваться функцией `get`. Эта функция возвращает значение, связанное с ключом, если этот ключ присутствует в словаре, или возвращает второй аргумент, если ключ отсутствует:

```
>>> print(english_to_french.get('blue', 'No translation'))
bleu
>>> print(english_to_french.get('chartreuse', 'No translation'))
No translation
```

Второй аргумент не является обязательным. Если этот аргумент отсутствует, `get` возвращает `None`, если словарь не содержит ключа.

Кроме того, если вы хотите безопасно получить значение, связанное с ключом, *и* определить в словаре значение по умолчанию, используйте метод `setdefault`:

```
>>> print(english_to_french.setdefault('chartreuse', 'No translation'))
No translation
```

Различия между `get` и `setdefault` заключаются в том, что после вызова `setdefault` в словаре появляется ключ `'chartreuse'` со значением `'No translation'`.

Для получения копии словаря используется метод `copy`:

```
>>> x = {0: 'zero', 1: 'one'}
>>> y = x.copy()
>>> y
{0: 'zero', 1: 'one'}
```

Этот метод создает поверхностную копию словаря; скорее всего, этого вам будет достаточно в большинстве ситуаций. Для словарей, содержащих изменяемые объекты в значениях (например, списки или другие словари), можно создать глубокую копию функцией `copy.deepcopy`. Концепции поверхностного и глубокого копирования были описаны в главе 5.

Метод `update` обновляет первый словарь всеми парами «ключ–значение» из второго словаря. Для ключей, общих для обоих словарей, значения из второго словаря замещают значения из первого словаря:

```
>>> z = {1: 'One', 2: 'Two'}
>>> x = {0: 'zero', 1: 'one'}
>>> x.update(z)
>>> x
{0: 'zero', 1: 'One', 2: 'Two'}
```

Методы словарей предоставляют в ваше распоряжение полный набор инструментов для работы со словарями. Важнейшие функции словарей перечислены в табл. 7.1.

Таблица 7.1. Операции со словарями

Операция	Описание	Пример
`{}`	Создает пустой словарь	`x = {}`
`len`	Возвращает количество элементов в словаре	`len(x)`
`keys`	Возвращает представление, содержащее все ключи в словаре	`x.keys()`
`values`	Возвращает представление, содержащее все значения в словаре	`x.values()`
`items`	Возвращает представление, содержащее все элементы в словаре	`x.items()`
`del`	Удаляет элемент из словаря	`del(x[key])`
`in`	Проверяет присутствие ключа в словаре	`'y' in x`
`get`	Возвращает значение ключа или выбранное значение по умолчанию	`x.get('y', None)`
`setdefault`	Возвращает значение, если ключ присутствует в словаре; в противном случае ассоциирует с ключом заданное значение по умолчанию и возвращает его	`x.setdefault('y', None)`
`copy`	Создает поверхностную копию словаря	`y = x.copy()`
`update`	Проводит слияние элементов двух словарей	`x.update(z)`

Таблица не содержит полного списка операций со словарями. За полным списком обращайтесь к документации стандартной библиотеки Python.

БЫСТРАЯ ПРОВЕРКА: ОПЕРАЦИИ СО СЛОВАРЯМИ

Допустим, имеются словари `x = {'a':1, 'b':2, 'c':3, 'd':4}` и `y = {'a':6, 'e':5, 'f':6}`. Что будет содержать переменная `x` при выполнении следующих фрагментов кода?

```
del x['d']
z = x.setdefault('g', 7)
x.update(y)
```

7.3. Подсчет слов

Предположим, имеется файл со списком слов — по одному слову на строку. Требуется узнать, сколько раз каждое слово встречается в файле. Словари позволяют легко решить эту задачу:

```
>>> sample_string = "To be or not to be"
>>> occurrences = {}
>>> for word in sample_string.split():
...     occurrences[word] = occurrences.get(word, 0) + 1 ❶
>>> for word in occurrences:
...     print("The word", word, "occurs", occurrences[word], \
...             "times in the string")
...
The word To occurs 1 times in the string
The word be occurs 2 times in the string
The word or occurs 1 times in the string
The word not occurs 1 times in the string
The word to occurs 1 times in the string
```

Счетчик вхождений каждого слова увеличивается в процессе перебора ❶. Это хороший пример выдающихся возможностей словарей: код прост, но поскольку операции со словарями в Python сильно оптимизированы, он также достаточно быстро работает. Эта схема настолько удобна, что она была стандартизирована в классе `Counter` модуля `collections` стандартной библиотеки.

7.4. Что может использоваться в качестве ключа?

В предыдущих примерах в качестве ключей использовались строки, но Python здесь вас никак не ограничивает. В качестве ключа словаря может использоваться любой объект Python, который является неизменяемым и хешируемым.

Как упоминалось ранее, в Python любой объект, который может быть изменен «на месте», называется *изменяемым*. Списки изменяемы, потому что вы можете добавлять, изменять и удалять их элементы. Словари изменяемы по тем же причинам. Числа неизменяемы. Если переменная `x` ссылается на число 3, то после присваивания `x` значения 4 переменная будет ссылаться на другое число (4), но число 3 при этом никак не изменится. Строки тоже неизменны. Запись `list[n]` возвращает *n*-й элемент списка, `string[n]` возвращает *n*-й символ строки, а `list[n] = value` изменяет *n*-й элемент строки; тем не менее запись `string[n] = character` в Python недопустима и порождает ошибку, потому что строки в Python неизменяемы.

К сожалению, требование неизменяемости и хешируемости ключей означает, что списки не могут быть ключами словарей, но во многих случаях было бы удобно работать с ключами, напоминающими списки. Например, было бы удобно хранить информацию о человеке с ключом, состоящим из имени и фамилии, что можно было бы легко сделать при использовании двухэлементного списка в качестве ключа.

В Python эта проблема решается за счет кортежей, которые, по сути, представляют собой неизменяемые списки. Кортежи создаются и используются так же, как списки, но после создания они уже не могут изменяться. Также есть еще одно ограничение: ключи словарей должны быть хешируемыми, что выводит ситуацию за грань простой неизменяемости. Чтобы быть хешируемым, значение должно иметь хеш-код (предоставляемый методом `__hash__`), который никогда не изменяется на протяжении срока жизни значения. Это означает, что кортежи, содержащие изменяемые

значения, хешируемыми *не являются*, хотя сами кортежи формально остаются неизменяемыми. Только кортежи, которые не содержат изменяемых объектов, являются хешируемыми и могут использоваться в качестве ключей в словарях.

В табл. 7.2 показано, какие из встроенных типов Python являются неизменяемыми, хешируемыми и пригодными для использования в качестве ключей словаря.

Таблица 7.2. Значения Python, пригодные для использования в качестве ключей словаря

Тип Python	Неизменяемый?	Хешируемый?	Ключ словаря?
`int`	Да	Да	Да
`float`	Да	Да	Да
`boolean`	Да	Да	Да
`complex`	Да	Да	Да
`str`	Да	Да	Да
`bytes`	Да	Да	Да
`bytearray`	Нет	Нет	Нет
`list`	Нет	Нет	Нет
`tuple`	Да	Иногда	Иногда
`set`	Нет	Нет	Нет
`frozenset`	Да	Да	Да
`dictionary`	Нет	Нет	Нет

В следующих разделах приводятся примеры, демонстрирующие совместное использование кортежей и словарей.

БЫСТРАЯ ПРОВЕРКА: ЧТО МОЖЕТ ИСПОЛЬЗОВАТЬСЯ В КАЧЕСТВЕ КЛЮЧА?

Решите, какие из следующих выражений могут быть ключами словаря: `1`; `'bob'`; `('tom', [1, 2, 3])`; `["file-name"]`; `"filename"`; `("filename", "extension")`.

7.5. Разреженные матрицы

В математике *матрица* представляет собой двумерную числовую таблицу, которая в учебниках обычно заключается в квадратные скобки:

$$\begin{bmatrix} 3 & 0 & -2 & 11 \\ 0 & 9 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & -5 \end{bmatrix}$$

Относительно стандартным способом представления таких матриц является список списков. В Python матрица представляется так:

```
matrix = [[3, 0, -2, 11], [0, 9, 0, 0], [0, 7, 0, 0], [0, 0, 0, -5]]
```

К элементам матрицы можно обращаться по номерам строк и столбцов:

```
element = matrix[rownum][colnum]
```

Однако в некоторых областях, например в метеорологическом прогнозировании, матрицы бывают очень большими. Количество строк и столбцов исчисляется тысячами, а следовательно, матрица может содержать миллионы элементов. Кроме того, такие матрицы обычно содержат очень много нулевых элементов. В некоторых приложениях все элементы матрицы, кроме небольшого подмножества, равны нулю. Для экономии памяти такие матрицы обычно хранятся в форме, в которой фактически в памяти находятся только ненулевые элементы. Подобные представления называются *разреженными матрицами*.

Разреженные матрицы легко реализуются в виде словарей с индексами-кортежами. Например, предыдущая разреженная матрица может быть записана следующим образом:

```
matrix = {(0, 0): 3, (0, 2): -2, (0, 3): 11,
          (1, 1): 9, (2, 1): 7, (3, 3): -5}
```

Теперь для обращения к отдельному элементу матрицы с заданными номерами столбца и строки можно воспользоваться следующим фрагментом кода:

```
if (rownum, colnum) in matrix:
    element = matrix[(rownum, colnum)]
else:
    element = 0
```

Несколько менее понятный (хотя и более эффективный) способ решения этой задачи основан на методе `get`, которому можно приказать вернуть `0`, если он не находит ключ в словаре, в противном случае возвращается значение, связанное с этим ключом, что предотвращает один из поисков по словарю:

```
element = matrix.get((rownum, colnum), 0)
```

Если вы собираетесь много работать с матрицами, вам стоит поближе познакомиться с `NumPy` — пакетом числовых вычислений.

7.6. Словари как кэши

В этом разделе показано, как словари могут использоваться в качестве *кэшей* — структур данных, в которых хранятся вычисленные результаты, чтобы их не приходилось пересчитывать заново. Предположим, вам нужна функция с именем `sole`, которая получает в аргументах три целых числа и возвращает результат. Функция может выглядеть примерно так:

```
def sole(m, n, t):
    # . . . Очень долгие вычисления. . .
    return(result)
```

Но если выполнение функции занимает много времени, а сама функция вызывается десятки тысяч раз, это может очень сильно замедлить работу программы.

Теперь представьте, что функция `sole` во время выполнения программы вызывается для 200 разных комбинаций аргументов. Таким образом, вызов `sole(12, 20, 6)` может встретиться 50 и более раз во время выполнения программы — и то же самое можно сказать о многих других комбинациях аргументов. Если удастся избежать повторного вычисления `sole` для того же набора аргументов, это позволит сэкономить много времени. Вы можете использовать словарь с кортежами в качестве ключей:

```
sole_cache = {}
def sole(m, n, t):
    if (m, n, t) in sole_cache:
        return sole_cache[(m, n, t)]
    else:
        # . . . Очень долгие вычисления . . .
        sole_cache[(m, n, t)] = result
        return result
```

Измененная функция `sole` использует глобальную переменную для хранения предыдущих результатов. Глобальная переменная представляет собой словарь, ключами которого являются кортежи, соответствующие комбинациям аргументов, передававшимся `sole` в прошлом. Каждый раз, когда функция `sole` будет получать комбинацию аргументов, для которой результат уже был вычислен, она вернет сохраненный результат вместо того, чтобы пересчитывать его заново.

ПОПРОБУЙТЕ САМИ: РАБОТА СО СЛОВАРЯМИ

Предположим, вы пишете программу, которая должна выполнять функции электронной таблицы. Как использовать словарь для хранения содержимого таблицы? Напишите код для хранения значения и чтения значения конкретной ячейки. Какими недостатками может обладать такое решение?

7.7. Эффективность словарей

Если у вас есть опыт использования традиционных компилируемых языков, возможно, вы предпочтете держаться подальше от словарей, потому что они уступают по эффективности спискам (массивам). Однако в действительности реализация словарей Python работает достаточно быстро. Многие внутренние аспекты языка зависят от словарей, поэтому была проведена серьезная работа по их оптимизации. Так как все структуры данных Python были основательно оптимизированы, вам не придется тратить много времени, беспокоясь о том, какой способ более или менее эффективен. Если проблема проще и элегантнее решается с применением словаря,

чем со списком, выбирайте это решение и рассматривайте альтернативы только в том случае, если словари безусловно приводят к неприемлемому снижению быстродействия.

ПРАКТИЧЕСКАЯ РАБОТА 7: ПОДСЧЕТ СЛОВ

В предыдущей практической работе вы взяли текст первой главы «Моби Дика», нормализовали регистр, удалили знаки препинания и записали разделенные слова в файл. В этой практической работе прочитайте этот файл, используйте словарь для подсчета вхождений каждого слова, а затем выведите самые частые и самые редкие слова.

Итоги

- Словари — мощные структуры данных, используемые для различных целей даже во внутренней реализации Python.
- Ключи словарей должны быть неизменяемыми, но любой неизменяемый объект может стать ключом словаря.
- Использование ключей означает более прямое обращение к коллекциям данных с меньшим объемом кода по сравнению с другими решениями.

8 Управляющие конструкции

Эта глава охватывает следующие темы:

- ✓ Повторение кода с циклом `while`
- ✓ Принятие решений: конструкция `if-elif-else`
- ✓ Итерация по списку с циклом `for`
- ✓ Использование перечней списков и словарей
- ✓ Разграничение операторов и блоков с отступом
- ✓ Оценка булевых значений и выражений

Python предоставляет полный набор конструкций, управляющих последовательностью выполнения команд, с циклами и условными командами. В этой главе все элементы рассматриваются подробно.

8.1. Цикл while

Базовый цикл `while` уже неоднократно встречался вам в книге. Полный цикл `while` выглядит так:

```
while условие:
    тело
else:
    завершение
```

Здесь `условие` является логическим выражением (то есть в результате которого будет получено `True` или `False`). Пока `условие` равно `True`, то `тело` цикла повторяется раз за разом. Если же `условие` принимает значение `False`, то цикл `while` выполняет секцию `завершение`, а затем прекращает выполнение. Если `условие` изначально равно `False`, `тело` не будет выполнено ни разу — только секция `завершение`. `Тело` и `завершение` представляют собой последовательности из одной или нескольких команд Python, разделенных символами новой строки и снабженных отступами

одного уровня. Интерпретатор Python использует уровень отступа как ограничитель блока. Никакие другие ограничители (например, квадратные или фигурные скобки) не нужны.

Часть `else` в циклах `while` не является обязательной и применяется нечасто. Это объясняется тем, что при отсутствии команды `break` в теле цикла этот цикл:

```
while условие:
    тело
else:
    завершение
```

и этот цикл:

```
while условие:
    тело
завершение
```

делают одно и то же — но вторая форма более понятна. Вероятно, я бы вообще не упоминала о секции `else`; с другой стороны, если вы не знаете о ней, она может привести вас в замешательство, если вы обнаружите ее в чужом коде. Кроме того, она может оказаться полезной в некоторых ситуациях.

В `теле` цикла `while` могут использоваться две специальные команды — `break` и `continue`. Команда `break` немедленно завершает цикл `while` даже без выполнения завершающей части (при наличии секции `else`). Команда `continue` пропускает оставшуюся часть `тела` цикла; условие проверяется снова, и цикл продолжается как обычно.

8.2. Команда if-elif-else

Самая общая форма конструкции `if-then-else` в Python выглядит так:

```
if условие1:
    тело1
elif условие2:
    тело2
elif условие3:
    тело3
.
.
.
elif условие(n-1):
    тело(n-1)



else:
    тело(n)
```

Это означает: если `условие1` равно `True`, выполняется `тело1`; в противном случае, если `условие2` равно `True`, выполняется `тело2`; в противном случае... и так далее, пока не будет найдено условие, равное `True`, или не будет достигнута секция `else` (тогда выполняется `тело(n)`). Как и в случае с циклом `while`, секции `тело` представляют

собой последовательности из одной или нескольких команд Python, разделенных символами новой строки и находящихся на одном уровне отступов.

Конечно, весь этот балласт нужен не для каждой проверки. Вы можете опустить части `elif` и/или часть `else`. Если условная команда не может найти тело для выполнения (ни одно условие не дает результат `True` или часть `else` отсутствует), она не делает ничего.

Часть `тело` после команды `if` является обязательной. Впрочем, в нее можно включить команду `pass` (как и в любой точке Python, где нужна команда). Команда `pass` размещается там, где должна находиться команда, но никаких действий она не выполняет:

```
if x < 5:
    pass
else:
    x = 5
```

В Python не существует конструкции case (или switch).

КУДА ПРОПАЛА КОМАНДА CASE В PYTHON?

Как упоминалось ранее, в Python нет команды case. Там, где в других языках использовалась бы команда case или switch, Python обычно прекрасно обходится цепочкой if… elif… elif… else. Если же цепочка становится слишком громоздкой, обычно можно воспользоваться словарем функций, как в следующем примере:

```
def do_a_stuff():
    #Обработка a
def do_b_stuff():
    #Обработка b
def do_c_stiff():
    #Обработка c

func_dict = {'a' : do_a_stuff,
             'b' : do_b_stuff,
             'c' : do_c_stuff }

x = 'a'
func_dict[x]()  <-- Выполнить функцию из словаря
```

В действительности выдвигались предположения по добавлению команды case в Python (см. PEP 275 и PEP 3103), но сообщество сошлось на том, что она не является необходимой и не оправдывает хлопоты по ее внедрению.

8.3. Цикл for

Цикл `for` в Python отличается от циклов `for` в некоторых других языках. В традиционном варианте при каждой итерации происходит увеличение и проверка переменной (как, например, обычно происходит в циклах C). В Python цикл `for` перебирает значения, возвращаемые любым итерируемым объектом, то есть любым объектом, который может сгенерировать последовательность значений. Так, цикл `for` может перебрать элементы списка, кортежа или строки. Однако

итерируемый объект также может быть специальной функцией с именем `range` или специальной разновидностью функций — так называемым *генератором*, или генераторным выражением, которые могут быть достаточно мощными. Обобщенная форма цикла `for` выглядит так:

```
for элемент in последовательность:
    тело
else:
    завершение
```

`тело` выполняется один раз для каждого элемента `последовательности`. Переменной `элемент` присваивается первый элемент `последовательности`, и выполняется `тело` цикла; затем переменной `элемент` присваивается второй элемент `последовательности`, выполняется `тело` цикла, и так далее для каждого оставшегося элемента `последовательности`.

Часть `else` не является обязательной. Как и часть `else` цикла `while`, она используется достаточно редко. Команды `break` и `continue` делают в циклах `for` то же самое, что и в циклах `while`.

Следующий цикл выводит обратные величины для всех чисел из `x`:

```
x = [1.0, 2.0, 3.0]
for n in x:
    print(1 / n)
```

8.3.1. Функция range

Иногда требуется выполнить цикл с явно заданными индексами (например, позициями значений в списке). Используйте команду `range` с вызовом `len` для списка, чтобы сгенерировать последовательность индексов для цикла `for`. Следующий код выводит позиции всех элементов списка, в которых будут обнаружены отрицательные числа:

```
x = [1, 3, -7, 4, 9, -5, 4]
for i in range(len(x)):
    if x[i] < 0:
        print("Found a negative number at index ", i)
```

Для заданного числа n вызов `range(n)` возвращает последовательность 0, 1, 2, ..., $n - 2$, $n - 1$. Таким образом, при передаче длины списка (вычисленной функцией `len`) будет получена последовательность индексов для элементов этого списка. Функция `range` не строит список целых чисел языка Python, как может показаться на первый взгляд. Вместо этого она создает объект диапазона, который выдает целые числа по запросу. В частности, это может быть полезно при использовании циклов для перебора очень больших списков. Например, вместо того чтобы строить список из 10 миллионов элементов, что потребует серьезных затрат памяти, можно воспользоваться вызовом `range(10000000)`, который займет лишь малую часть памяти, и сгенерировать последовательность целых чисел от 0 до (но *не* включая) 10 000 000, как того требует цикл `for`.

8.3.2. Управление диапазоном с использованием начального значения и приращения

Две разновидности функции `range` предоставляют еще больше возможностей для контроля значения создаваемой последовательностью. В форме `range` с двумя числовыми аргументами первый аргумент задает начальное число создаваемой последовательности, а второй аргумент — конечное число (которое не включается в последовательность). Несколько примеров:

```
>>> list(range(3, 7)) ❶
[3, 4, 5, 6]
>>> list(range(2, 10)) ❶
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 3))
[]
```

Функция `list()` здесь включена только для того, чтобы элементы, генерируемые `range`, выглядели как список. В реальном коде она обычно не используется ❶.

Возможность генерирования чисел в обратном порядке не поддерживается, поэтому значение `list(range(5, 3))` представляет собой пустой список. Чтобы вести отсчет в обратном порядке или с любым приращением, отличным от 1, необходимо передать `range` необязательный третий аргумент — приращение:

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
```

Последовательности, возвращаемые `range`, всегда включают начальное значение, переданное в аргументе, и никогда не включают конечное значение, переданное в другом аргументе.

8.3.3. Команды break и continue в циклах for

Две специальные команды — `break` и `continue` — также могут использоваться в теле цикла `for`. Команда `break` немедленно завершает цикл `for` даже без выполнения завершающей части (при наличии секции `else`). Команда `continue` пропускает оставшуюся часть тела цикла, и цикл продолжается со следующего элемента, как обычно.

8.3.4. Цикл for и распаковка кортежей

Распаковка кортежей позволит сделать некоторые циклы `for` более элегантными. Следующий код получает список двухэлементных кортежей и вычисляет значение суммы произведений двух чисел каждого кортежа (распространенная математическая операция в некоторых областях):

```
somelist = [(1, 2), (3, 7), (9, 5)]
result = 0
for t in somelist:
    result = result + (t[0] * t[1])
```

То же самое, но более элегантно:

```
somelist = [(1, 2), (3, 7), (9, 5)]
result = 0

for x, y in somelist:
    result = result + (x * y)
```

В этом коде сразу же за ключевым словом `for` вместо обычной одиночной переменной идет кортеж `x, y`. При каждой итерации цикла `for x` содержит элемент `0` текущего кортежа из `list`, а `y` содержит элемент `1` текущего кортежа из `list`. Такое использование кортежей реализовано в Python для удобства; оно показывает Python, что каждый элемент списка должен быть кортежем соответствующего размера для распаковки в переменные, имена которых указаны в кортеже после `for`.

8.3.5. Функция enumerate

Распаковку кортежа можно совместить с функцией `enumerate` для перебора как элементов, так и их индексов. Такое решение напоминает решение с `range`, но у него есть преимущество: код получается более элегантным и понятным. Как и в предыдущем примере, следующий фрагмент выводит все позиции списка, в которых будут обнаружены отрицательные числа:

```
x = [1, 3, -7, 4, 9, -5, 4]
for i, n in enumerate(x): ❶
    if n < 0: ❷
        print("Found a negative number at index ", i) ❸
```

Функция `enumerate` возвращает кортежи (индекс, элемент) ❶. При этом вы можете обратиться к элементу без индекса ❷, но индекс также доступен ❸.

8.3.6. Функция zip

Иногда бывает полезно объединить два и более итерируемых объекта до того, как выполнять перебор. Функция `zip` берет соответствующие элементы из одного или нескольких итерируемых объектов и объединяет их в кортежи, пока не будет достигнут конец более короткого итерируемого объекта:

```
>>> x = [1, 2, 3, 4]
>>> y = ['a', 'b', 'c']  <-- y содержит 3 элемента; x содержит 4 элемента
>>> z = zip(x, y)
>>> list(z)
[(1, 'a'), (2, 'b'), (3, 'c')]  <-- z содержит только 3 элемента
```

ПОПРОБУЙТЕ САМИ: ЦИКЛЫ И КОМАНДЫ IF

Допустим, имеется список `x = [1, 3, 5, 0, -1, 3, -2]`, из которого нужно удалить все отрицательные числа. Напишите код для решения этой задачи.

Как бы вы подсчитали общее количество отрицательных чисел в списке `y = [[1, -1, 0], [2, 5, -9], [-2, -3, 0]]`?

Какой код вы бы использовали для вывода описания: `very low`, если значение `x` меньше −5, `low`, если оно лежит в диапазоне от −5 до 0; `neutral`, если оно равно 0; `high`, если оно лежит в диапазоне от 0 до 5; и `very high`, если оно больше 5?

8.4. Генераторы строк и словарей

Паттерн с использованием цикла `for` для перебора списка, изменения или выбора отдельных элементов и создания нового списка или словаря чрезвычайно популярен. Такие циклы часто выглядят примерно так:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = []
>>> for item in x:
...     x_squared.append(item * item)
...
>>> x_squared
[1, 4, 9, 16]
```

Подобные ситуации встречаются настолько часто, что в Python для них существует специальная сокращенная запись, называемая *генератором* (comprehension). Генератор списка или словаря можно представить как однострочную запись цикла `for`, которая создает новый список или словарь из последовательности. Синтаксис генератора списка выглядит так:

```
новый_список = [выражение1 for переменная in старый_список if выражение2]
```

Генератор словаря выглядит так:

```
новый_словарь = {выражение1:выражение2 for переменная in список if выражение3}
```

В обоих случаях основная часть выражения напоминает начало цикла `for`: `for переменная in список`, также присутствует некое выражение, использующее эту переменную для создания нового ключа или значения, и необязательное условное выражение, которое на основании значения переменной принимает решение о ее включении в новый список или словарь. Следующий код делает то же самое, что и предыдущий, но в формате генератора списка:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = [item * item for item in x]
>>> x_squared
[1, 4, 9, 16]
```

Команда `if` может использоваться для выбора элементов из исходного списка:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = [item * item for item in x if item > 2]

>>> x_squared
[9, 16]
```

Генераторы словарей работают почти так же, но задать нужно как ключ, так и значение. Если вам нужно сделать нечто аналогичное предыдущему примеру, но так, чтобы число было ключом, а квадрат числа — значением в словаре, вы можете воспользоваться генератором словарей:

```
>>> x = [1, 2, 3, 4]
>>> x_squared_dict = {item: item * item for item in x}
>>> x_squared_dict
{1: 1, 2: 4, 3: 9, 4: 16}
```

Генераторы списков и словарей обладают исключительной гибкостью и мощью, и когда вы к ним привыкнете, операции обработки списков заметно упростятся. Старайтесь экспериментировать с ними и применять на практике каждый раз, когда вы пишете цикл `for` для обработки списка элементов.

8.4.1. Выражения-генераторы

Выражения-генераторы напоминают генераторы списков. Выражение-генератор похоже на генератор списка, но вместо квадратных скобок используются круглые. Следующий пример представляет собой версию генератора списка, описанного выше, реализованную с использованием выражений-генераторов:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = (item * item for item in x)
>>> x_squared
<generator object <genexpr> at 0x102176708>
>>> for square in x_squared:
...     print(square,)
...
1 4 9 16
```

Обратите внимание: различия не ограничиваются заменой квадратных скобок; это выражение не возвращает список. Вместо этого оно возвращает объект-генератор, который может использоваться в качестве итератора в цикле `for`, очень похоже на то, что делает функция `range()`. Преимущество выражений-генераторов заключается в том, что список не генерируется полностью в памяти, что позволяет генерировать очень большие последовательности с минимальными затратами памяти.

ПОПРОБУЙТЕ САМИ: ГЕНЕРАТОРЫ

Какой генератор списков вы бы использовали для обработки списка `x` с удалением всех отрицательных выражений?

Создайте генератор, возвращающий только нечетные числа от 1 до 100. (Подсказка: нечетные числа можно отличить по наличию остатка от деления на 2; чтобы узнать остаток от деления на 2, используйте операцию `%2`.)

Напишите код создания словаря, содержащего числа от 11 до 15 и их кубы.

8.5. Команды, блоки и отступы

Так как в управляющих конструкциях, упоминавшихся в этой главе, впервые использовались блоки и отступы, сейчас стоит вернуться к этой теме.

Python использует отступы для определения границ блоков (или тел) управляющих конструкций. Блок состоит из одной или нескольких команд, обычно разделенных символами новой строки. Примерами команд Python могут служить команда присваивания, вызовы функций, функция `print`, пустая команда `pass` и команда `del`. Управляющие конструкции (`if-elif-else`, циклы `while` и `for`) являются составными командами:

```
секция составной команды:
    блок
секция составной команды:
    блок
```

Составная команда состоит из одной или нескольких секций, за каждой из которых следует блок с отступом. Составные команды также могут находиться в блоках, как и любые другие команды. В этом случае они создают вложенные блоки.

Также существует пара особых случаев. В одной строке можно разместить сразу несколько команд, разделив их символом «точка с запятой» (`;`). Блок, содержащий одну строку, может быть размещен в той же строке после двоеточия (`:`), завершающего секцию составной команды:

```
>>> x = 1; y = 0; z = 0
>>> if x > 0: y = 1; z = 10
... else: y = -1
...
>>> print(x, y, z)
1 1 10
```

Неправильные отступы в коде приводят к выдаче исключения. Вам могут встретиться две формы этого исключения. Первая:

```
>>>
>>>   x = 1
File "<stdin>", line 1
    x = 1
    ^
   IndentationError: unexpected indent
>>>
```

В этом коде отступом снабжена строка, в которой отступа быть не должно. В базовом интерактивном режиме место, в котором возникла проблема, помечается символом ^ («крышка»). В оболочке Python среды IDLE (рис. 8.1) неправильный отступ выделяется цветом. То же сообщение будет выдано при отсутствии отступа в коде там, где он необходим (то есть в первой строке после секции составной команды).

Одна из ситуаций, в которой может возникнуть эта ошибка, особенно коварна. Если вы работаете в редакторе, в котором табуляции отображаются в виде четырех

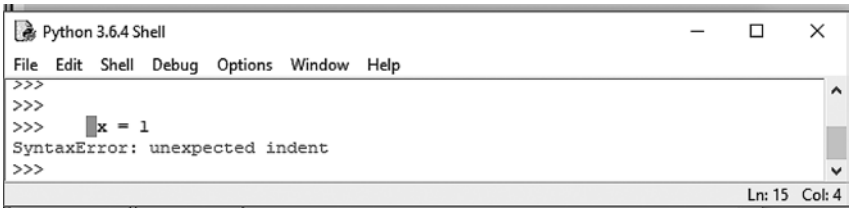


Рис. 8.1. Неправильные отступы

пробелов (или в интерактивном режиме Windows, в котором первая табуляция отделяется только четырьмя пробелами от приглашения), и создадите одну строку с четырьмя пробелами, а другую с табуляцией, внешне будет казаться, что эти две строки имеют одинаковые отступы. Однако вы получите исключение, потому что Python расширяет табуляцию до восьми пробелов. Лучший способ избежать этой проблемы — использовать только пробелы в коде Python. Если вы вынуждены использовать табуляции для создания отступов или же имеете дело с кодом, в котором применяются табуляции, никогда не смешивайте их с пробелами.

Что касается базового интерактивного режима и оболочки Python в IDLE, вероятно, вы заметили, что после внешнего уровня отступов необходима дополнительная строка:

```
>>> x = 1
>>> if x == 1:
...     y = 2
...     if v > 0:
...         z = 2
...         v = 0
...
>>> x = 2
```

После строки `z = 2` дополнительная строка не нужна, а после строки `v = 0` она необходима. Дополнительная строка не нужна при размещении кода в файле модуля.

Вторая разновидность исключения происходит в том случае, если команда в блоке имеет отступ меньше положенного:

```
>>> x = 1
>>> if x == 1:
           y = 2
         z = 2
File "<stdin>", line 3
       z = 2
       ^
    IndentationError: unindent does not match any outer indentation level
```

В этом примере строка с командой `z = 2` неправильно выровнена под строкой с командой `y = 2`. Эта форма встречается редко, но я снова упоминаю ее, потому что в похожих ситуациях она может привести вас в замешательство.

Python позволяет использовать отступы любого размера и не жалуется, если вы последовательно назначаете их в пределах одного блока. Пожалуйста, не

злоупотребляйте этой гибкостью. Рекомендуемым стандартом отступов считаются четыре пробела на каждый уровень.

Прежде чем завершать тему отступов, стоит поговорить о разбиении команд на несколько строк — конечно, необходимость в этом возрастает с увеличением уровня отступов. Для явного разбиения строки программы используется символ \. Вы также можете неявно разбить любую команду между лексемами внутри ограничителей (), {} или [] (то есть при вводе множества значений в списке, кортеже или словаре; наборе аргументов при вызове функции; или любом выражении в квадратных скобках). Отступ в строке продолжения можно выбрать произвольно:

```
>>> print('string1', 'string2', 'string3' \
...    , 'string4', 'string5')
string1 string2 string3 string4 string5
>>> x = 100 + 200 + 300 \
...    + 400 + 500
>>> x
1500
>>> v = [100, 300, 500, 700, 900,
...    1100, 1300]
>>> v
[100, 300, 500, 700, 900, 1100, 1300]
>>> max(1000, 300, 500,
...        800, 1200)
1200
>>> x = (100 + 200 + 300
...          + 400 + 500)
>>> x
1500
```

Строковые литералы также можно разбивать символом \. Однако при этом любые табуляции или пробелы в отступах становятся частью текста, а строка программы *должна* завершаться символом \. Чтобы избежать подобных ситуаций, помните, что интерпретатор Python автоматически объединяет строковые литералы, разделенные пропусками:

```
>>> "strings separated by whitespace "    \
...    """are automatically"""  ' concatenated'
'strings separated by whitespace are automatically concatenated'
>>> x = 1
>>> if x > 0:
...        string1 = "this string broken by a backslash will end up \
...                with the indentation tabs in it"
...
>>> string1
'this string broken by a backslash will end up \t\t\twith
     the indentation tabs in it'
>>> if x > 0:
...        string1 = "this can be easily avoided by splitting the " \
...            "string in this way"
...
>>> string1
'this can be easily avoided by splitting the string in this way'
```

8.6. Логические значения и выражения

В предыдущих примерах управления последовательностью выполнения команд использовались довольно очевидные условные проверки, но они никогда не объясняли, что же считается истинным или ложным условием в Python или какие выражения могут использоваться там, где должно размещаться условие. В данном разделе описаны эти аспекты Python.

В Python существует объект логического типа, которому может присваиваться значение `True` или `False`. Любое выражение с логической операцией возвращает `True` или `False`.

8.6.1. Использование объектов Python как логических значений

Кроме того, в интерпретации логических значений Python следует примеру языка C. Как известно, в C целое число `0` интерпретируется как ложное значение, а все остальные целые числа интерпретируются как истинные. Python обобщает эту идею: `0` и пустые значения интерпретируются как `False`, а все остальные значения интерпретируются как `True`. На практике это означает следующее:

- Числа `0`, `0.0` и `0+0j` интерпретируются как `False`; все остальные числа интерпретируются как `True`.
- Пустая строка `""` интерпретируется как `False`; любая другая строка интерпретируется как `True`.
- Пустой список `[]` интерпретируется как `False`; любой другой список интерпретируется как `True`.
- Пустой словарь `{}` интерпретируется как `False`; любой другой словарь интерпретируется как `True`.
- Пустое множество `set()` интерпретируется как `False`; любое другое множество интерпретируется как `True`.
- Специальное значение Python `None` всегда интерпретируется как `False`.

Мы еще не рассматривали некоторые структуры данных Python, но в общем случае действуют те же правила. Если структура данных пуста или содержит `0`, она интерпретируется как ложное значение в логическом контексте, в противном случае она всегда интерпретируется как истинное значение. Некоторые объекты (например, объекты файлов и объекты кода) не имеют осмысленного определения `0` или пустого элемента, поэтому они не должны использоваться в логическом контексте.

8.6.2. Сравнения и логические операторы

Для сравнения объектов можно использовать обычные операторы: `<`, `<=`, `>`, `>=` и т. д. Оператор `==` проверяет равенство, а оператор `!=` означает «не равно». Также существуют операторы `in` и `not in` для проверки принадлежности к последовательностям

(списки, кортежи, строки и словари) и операторы `is` и `is not` для проверки тождественности двух объектов.

Выражения, возвращающие логическое значение, могут объединяться в более сложные выражения операторами `and`, `or` и `not`. Следующий фрагмент кода проверяет, входит ли переменная в заданный диапазон:

```
if 0 < x and x < 10:
    ...
```

Для подобных составных условий Python предлагает удобную сокращенную запись. Диапазон записывается так, как он обычно записывается в учебниках математики:

```
if 0 < x < 10:
    ...
```

При этом действуют различные правила приоритета: если вы не уверены, добавьте круглые скобки, чтобы Python гарантированно интерпретировал выражение именно так, как вам нужно. Вероятно, круглые скобки уместно использовать в сложных выражениях независимо от того, необходимо это или нет, потому что при будущем сопровождении вашего кода разработчик будет четко понимать, что в нем происходит. За дополнительной информацией о приоритетах обращайтесь к документации Python.

Оставшаяся часть этого раздела содержит более сложный материал. Если вы читаете эту книгу для изучения языка, ее пока можно пропустить.

Операторы `and` и `or` возвращают объекты. Оператор `and` возвращает либо первый ложный объект (который будет получен при вычислении выражения), либо последний объект. Аналогичным образом оператор `or` возвращает либо первый истинный объект, либо последний объект. На первый взгляд такое описание выглядит невразумительно, но оно работает правильно; если выражение с `and` содержит хотя бы один ложный элемент, то под воздействием этого элемента для всего выражения вычисляется ложный результат, возвращается значение `False`. Если все элементы равны `True`, то результат всего выражения тоже равен `True` и возвращается последнее значение, которое тоже должно быть равно `True`. Для оператора `or` истинно обратное: даже одного элемента `True` достаточно для того, чтобы все выражение интерпретировалось как `True` и первое значение `True` возвращалось. Если ни одного значения `True` не будет найдено, возвращается последнее значение (`False`). Другими словами, как и в случае с многими другими языками, вычисление останавливается сразу же при обнаружении истинного выражения для оператора `or` или при обнаружении ложного значения для оператора `and`:

```
>>> [2] and [3, 4]
[3, 4]
>>> [] and 5
[]
>>> [2] or [3, 4]
[2]
>>> [] or 5
5
>>>
```

Операторы `==` и `!=` проверяют, содержат ли их операнды одинаковые значения. Операторы `==` и `!=` используются намного чаще операторов `is` и `is not`, которые проверяют, представляют ли их операнды один и тот же объект:

```
>>> x = [0]
>>> y = [x, 1]
>>> x is y[0]  <-- Они представляют один объект
True
>>> x = [0]  <-- x присваивается другой объект
>>> x is y[0]
False
>>> x == y[0]
True
```

Если этот пример вам недостаточно понятен, вернитесь к разделу 5.6 «Вложенные списки и глубокое копирование».

БЫСТРАЯ ПРОВЕРКА: ЛОГИЧЕСКИЕ ЗНАЧЕНИЯ И ИСТИННОСТЬ

Решите, будет ли каждое из следующих выражений интерпретировано как истинное или ложное: `1`, `0`, `-1`, `[0]`, `1 and 0`, `1 > 0 or []`.

8.7. Простая программа для анализа текстового файла

Чтобы вы лучше поняли, как работают программы Python, в этом разделе будет рассмотрен небольшой пример, который приблизительно повторяет функциональность утилиты UNIX `wc`: он выводит количество строк, слов и символов в файле. Пример в этом листинге написан специально для программистов, которые только осваивают Python, и написан насколько возможно просто.

Листинг 8.1. word_count.py

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

""" Читает файл и возвращает количество строк, слов
    и символов - по аналогии с утилитой UNIX wc
"""

infile = open('word_count.tst')  <-- Открывает файл

lines = infile.read().split("\n")  <-- Читает файл с разбивкой по строкам

line_count = len(lines)  <-- Определяет количество строк функцией len()

word_count = 0
char_count = 0  | Инициализирует другие счетчики

for line in lines:  <-- Перебирает строки файла
```

```
    words = line.split()  <-- Выполняет разбивку по словам
    word_count += len(words)

    char_count += len(line)  <-- Возвращает количество символов

print("File has {0} lines, {1} words, {2} characters".format
                              (line_count, word_count, char_count))  Выводит результаты
```

Для проверки можно выполнить этот пример для файла, содержащего первый абзац краткого описания этой главы:

Листинг 8.2. word_count.tst

```
Python provides a complete set of control flow elements,
including while and for loops, and conditionals.
Python uses the level of indentation to group blocks
of code with control elements.
```

При выполнении word_count.py будет получен следующий результат:

```
naomi@mac:~/quickpythonbook/code $ python3.1 word_count.py
File has 4 lines, 30 words, 189 characters
```

Этот код дает представление о программе Python. Код получился достаточно компактным, и большая часть работы выполняется в трех строках кода в цикле `for`. Более того, эту программу можно сделать еще короче и выразительнее. Многие программисты Python считают эту компактность одной из самых сильных сторон Python.

ПРАКТИЧЕСКАЯ РАБОТА 8: РЕФАКТОРИНГ WORD_COUNT

Перепишите программу word-count из раздела 8.7, чтобы сделать ее короче. Возможно, вам стоит вспомнить рассмотренные выше операции со строками и списками, а также продумать различные способы организации кода.

Также попробуйте сделать программу «умнее», чтобы словами считались только алфавитные строки (но не знаки препинания или специальные знаки).

Итоги

- Отступы используются в Python для группировки блоков кода.
- В Python существуют циклы `while` и `for`, а также условные конструкции с `if-elif-else`.
- В Python используются логические значения `True` и `False`, с которыми могут связываться переменные.
- Python интерпретирует 0 и пустые значения как `False`, а любые ненулевые и непустые значения как `True`.

9 Функции

Эта глава охватывает следующие темы:

- ✓ Определение функций
- ✓ Использование параметров функции
- ✓ Передача изменяемых объектов в качестве параметров
- ✓ Понимание локальных и глобальных переменных
- ✓ Создание и использование функций генератора
- ✓ Создание и использование лямбда-выражений
- ✓ Использование декораторов

Эта глава предполагает, что вы знакомы с определениями функций хотя бы в еще одном компьютерном языке, а также с концепциями определения функций, аргументов, параметров и т. д.

9.1. Базовые определения функций

Базовый синтаксис определения функций Python выглядит так:

```
def name(параметр1, параметр2, . . .):
    тело
```

Как и в случае с управляющими структурами, Python использует отступы для ограничения тела определения функции. В следующем простом примере код факториала из предыдущего раздела размещается в теле функции, чтобы вы могли вызвать функцию `fact` для получения факториала числа:

```
>>> def fact(n):
...     """Возвращает факториал заданного числа."""  ❶
...     r = 1
...     while n > 0:
...         r = r * n
...         n = n - 1
...     return r  ❷
...
```

Вторая строка ❶ содержит необязательную *строку документации*. Чтобы просмотреть ее значение, выведите переменную `fact.__doc__`. Строки документации предназначены для описания внешнего поведения функции и получаемых ею параметров, тогда как комментарии должны содержать внутренние сведения о работе кода. Строки документации следуют сразу же после определений функций; обычно они заключаются в тройные кавычки, что позволяет создавать многострочные описания. Существуют программы просмотра, извлекающие начало строк документации. Чаще всего в начале многострочных строк документации следует сводное описание функции, далее идет пустая вторая строка, и после нее следует остальная информация. Значение после `return` возвращается коду, вызывающему функцию ❷.

ПРОЦЕДУРА ИЛИ ФУНКЦИЯ?

В некоторых языках функция, не возвращающая значение, называется *процедурой*. Хотя вы можете (и наверняка будете) писать функции, не содержащие команды `return`, такие функции не будут процедурами. Все процедуры Python являются функциями. Если в теле процедуры не выполняется команда `return`, возвращается специальное значение `None`, а при выполнении команды `return arg` немедленно возвращается значение `arg`. После выполнения `return` никакие другие команды в теле функции не выполняются. Так как в Python нет полноценных процедур, я буду в обоих случаях использовать термин «*функция*».

Хотя все функции Python возвращают значения, вы сами решаете, использовать полученное значение или нет:

```
>>> fact(4) ❶
24 ❷
>>> x = fact(4) ❸
>>> x
24
>>>
```

Возвращаемое значение не связывается с переменной ❶. Значение функции `fact` выводится только в интерпретаторе ❷. Возвращаемое значение связывается с переменной `x` ❸.

9.2. Параметры функций

Большинство функций получает параметры при вызове; в каждом языке используется собственная спецификация определения параметров функций. Python гибок в этом отношении: в языке предусмотрено три способа определения параметров функций, которые будут кратко описаны в этом разделе.

9.2.1. Позиционные параметры

Проще всего передавать параметры функциям в Python в соответствии с позицией. В первой строке функции вы указываете имена переменных для всех параметров; при вызове функции параметры, указанные в коде вызова, сопоставляются с переменными параметров функции в соответствии с их порядком. Следующая функция вычисляет результат возведения `x` в степень `y`:

```
>>> def power(x, y):
...     r = 1
...     while y > 0:
...         r = r * x
...         y = y - 1
...     return r
...
>>> power(3, 3)
27
```

Этот вариант требует, чтобы количество параметров в точке вызова точно соответствовало количеству параметров в определении функции; в противном случае выдается исключение `TypeError`:

```
>>> power(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required positional argument: 'y'
>>>
```

ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

Параметрам функций могут назначаться значения по умолчанию; они присваиваются в первой строке определения функции:

```
def fun(арг1, арг=умолч2, арг3=умолч3, . . .)
```

Значения по умолчанию могут присваиваться любому количеству параметров. Параметры со значениями по умолчанию должны определяться последними в списке параметров, потому что Python, как и многие языки, сопоставляет аргументы с параметрами на основании их позиций. Количество передаваемых аргументов должно быть достаточным для того, чтобы последнему параметру в списке, не имеющему значения по умолчанию, соответствовал передаваемый аргумент. Более гибкий механизм передачи аргументов описан в разделе 9.2.2 «Передача аргументов по имени параметра».

Следующая функция также вычисляет результат возведения `x` в степень `y`. Но если значение `y` не указано при вызове функции, по умолчанию используется значение 2, и функция просто возводит аргумент в квадрат:

```
>>> def power(x, y=2):
...     r = 1
...     while y > 0:
...         r = r * x
...         y = y - 1
...     return r
...
```

Эффект аргумента по умолчанию продемонстрирован в следующем интерактивном сеансе:

```
>>> power(3, 3)
27
>>> power(3)
9
```

9.2.2. Передача аргументов по имени параметра

Аргументы также могут передаваться функциям по имени соответствующего параметра функции (вместо позиции). Продолжая предыдущий интерактивный пример, вы можете ввести

```
>>> power(2, 3)
8
>>> power(3, 2)
9
>>> power(y=2, x=3)
9
```

Поскольку для аргументов `power` в последнем вызове указаны имена, их порядок может быть произвольным; аргументы связываются с одноименными параметрами из определения `power`, поэтому в результате вы получаете `3^2`. Такой способ передачи аргументов называется *передачей по ключевым словам*.

Передача по ключевым словам в сочетании с возможностью определения аргументов по умолчанию может быть чрезвычайно полезной, когда вы определяете функции с множеством возможных аргументов, большинство из которых имеет значения по умолчанию.

Представьте функцию, которая должна строить список с информацией о файлах в текущем каталоге; эта функция использует логические аргументы для обозначения того, должен ли список включать информацию о размере файла, дате последнего изменения и т. д. для каждого файла. Такая функция может определяться по следующей схеме:

```
def list_file_info(size=False, create_date=False, mod_date=False, ...):
    ...получить имена файлов...
    if size:
        # код для получения размеров файлов
    if create_date:
        # код для получения дат создания
    # и т. д. для любых других атрибутов

    return fileinfostructure
```

Вызовите ее с передачей аргументов по ключевым словам, которые будут определять нужную информацию (в данном примере запрашивается размер файла и дата изменения, но *не* дата создания):

```
fileinfo = list_file_info(size=True, mod_date=True)
```

Такой способ передачи аргументов особенно хорошо подходит для функций с очень сложным поведением; в частности, такие функции встречаются при программировании графических интерфейсов (GUI). Если вы когда-либо будете работать с пакетом Tkinter для построения графических интерфейсов в Python, вы поймете, что подобные необязательные аргументы, обозначаемые ключевыми словами, могут быть чрезвычайно удобными.

9.2.3. Переменное количество аргументов

Функции Python также могут определяться для получения переменного количества аргументов; это можно сделать двумя способами. Первый способ предназначен для относительно знакомого случая, при котором неизвестное количество аргументов в конце списка аргументов собирается в список. В другом способе произвольное количество аргументов, передаваемых по ключевым словам и не имеющих параметров с соответствующим именем в списке параметров функции, объединяется в словарь. Эти два механизма рассматриваются ниже.

Неопределенное количество позиционных аргументов

Если имени последнего параметра функции предшествует символ *, все лишние аргументы без ключевых слов в вызове функции (то есть позиционные аргументы, не связанные с другим параметром) объединяются и присваиваются указанному параметру в формате кортежа. Ниже продемонстрирован простой способ реализации функции для нахождения наибольшего значения в списке чисел.

Сначала реализуйте функцию:

```
>>> def maximum(*numbers):
...     if len(numbers) == 0:
...         return None
...     else:
...         maxnum = numbers[0]
...         for n in numbers[1:]:
...             if n > maxnum:
...                 maxnum = n
...         return maxnum
...
```

Затем протестируйте поведение функции:

```
>>> maximum(3, 2, 8)
8
>>> maximum(1, 5, 9, -2, 2)
9
```

Неопределенное количество аргументов, передаваемых по ключевым словам

Также можно обработать произвольное количество аргументов, передаваемых по ключевым словам. Если последний параметр в списке снабжен префиксом **, все лишние аргументы, передаваемые *по ключевым словам*, объединяются в словарь. Ключом для каждого элемента словаря является ключевое слово (имя параметра) «избыточного» аргумента, а значением — сам аргумент. Аргумент, передаваемый по ключевому слову, считается избыточным, если ключевое слово, по которому он передавался, не соответствует ни одному из имен параметров в определении функции.

Пример:

```
>>> def example_fun(x, y, **other):
...     print("x: {0}, y: {1}, keys in 'other': {2}".format(x,
...           y, list(other.keys())))
...     other_total = 0
...     for k in other.keys():
...         other_total = other_total + other[k]
...     print("The total of values in 'other' is {0}".format(other_total))
```

Тестирование этой функции в интерактивном сеансе показывает, что функция успешно обрабатывает аргументы с ключевыми словами `foo` и `bar`, несмотря на то что `foo` и `bar` не являются именами параметров в определении функции:

```
>>> example_fun(2, y="1", foo=3, bar=4)
x: 2, y: 1, keys in 'other': ['foo', 'bar']
The total of values in 'other' is 7
```

9.2.4. Совмещение способов передачи аргументов

Формально все возможности передачи аргументов функций Python могут применяться одновременно, хотя если не принять меры, разобраться в таком коде будет непросто. Общее правило смешанной передачи аргументов гласит, что сначала идут позиционные аргументы, затем именованные аргументы, за ними следует неопределенный позиционный аргумент с одним символом * и, наконец, неопределенный аргумент ключевых слов с именем **. За полной информацией обращайтесь к документации.

БЫСТРАЯ ПРОВЕРКА: ФУНКЦИИ И ПАРАМЕТРЫ

Как вы напишете функцию, которая получает любое количество неименованных аргументов, а затем выводит их значения в обратном порядке?

Что нужно сделать, чтобы создать процедуру, то есть функцию без возвращаемого значения?

Что произойдет, если сохранить возвращаемое значение функции в переменной?

9.3. Изменяемые объекты в качестве аргументов

Аргументы передаются в виде ссылки на объект. Параметр становится новой ссылкой на объект. Для неизменяемых объектов (таких, как кортежи, строки и числа) любые манипуляции с параметром не имеют никакого эффекта за пределами функции. Но если вы передадите изменяемый объект (например, список, словарь или экземпляр класса), любые изменения в таком объекте будут отражены за пределами функции. Присваивание параметра не влияет на аргумент, как видно из рис. 9.1 и 9.2:

```
>>> def f(n, list1, list2):
...     list1.append(3)
...     list2 = [4, 5, 6]
...     n = n + 1
...
>>> x = 5
>>> y = [1, 2]
>>> z = [4, 5]
>>> f(x, y, z)
>>> x, y, z
(5, [1, 2, 3], [4, 5])
```

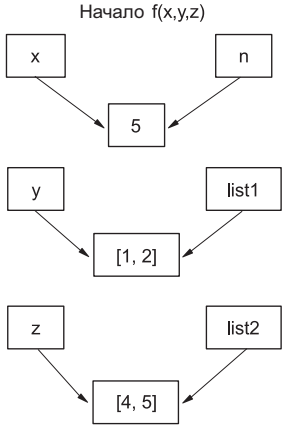


Рис. 9.1. В начале функции f() как исходные переменные, так и параметры функции ссылаются на одни и те же объекты

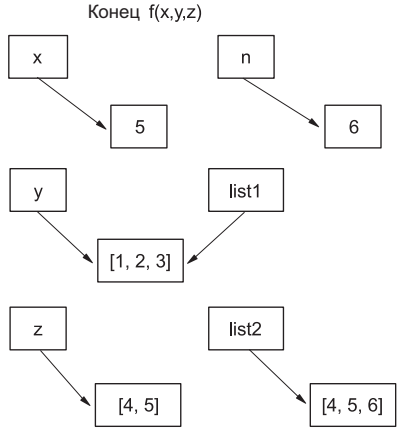


Рис. 9.2. В конце функции f() переменная y (list1 внутри функции) была изменена «на месте», а n и list2 ссылаются на другие объекты

Рисунки 9.1 и 9.2 показывают, что происходит при вызове функции `f`. Переменная `x` не изменяется, потому что она является неизменной. Вместо этого в результате присваивания параметр функции `n` ссылается на новое значение 6. Аналогичным образом переменная `z` остается неизменной, потому что внутри функции `f` ее соответствующему параметру `list2` присваивается ссылка на новый объект `[4, 5, 6]`. Только в `y` видны изменения, потому что изменился список, на который указывает переменная.

БЫСТРАЯ ПРОВЕРКА: ИЗМЕНЯЕМЫЕ ПАРАМЕТРЫ ФУНКЦИЙ

Что произойдет при изменении списка или словаря, который был передан функции как значение параметра? Какие операции с большой вероятностью породят изменения, которые будут видны за пределами функции? Что можно сделать, чтобы свести риск к минимуму?

9.4. Локальные, нелокальные и глобальные переменные

Вернемся к определению `fact`, приведенному в начале главы:

```
def fact(n):
    """Возвращает факториал заданного числа."""
    r = 1
    while n > 0:
        r = r * n
        n = n - 1
    return r
```

Переменные `r` и `n` являются *локальными* для любого конкретного вызова функции факториала; изменения в них, внесенные во время выполнения функции, не отразятся ни на каких переменных за пределами этой функции. Любые переменные в списке параметров функции и любые переменные, созданные в функции командой присваивания (например, `r = 1`), являются локальными для данной функции.

Переменную можно явно объявить глобальной перед ее использованием, для чего применяется команда `global`. Функция может обращаться к глобальным переменным и изменять их. Такие переменные существуют за пределами функций; обращаться к ним и изменять их также могут другие функции, если эти переменные будут объявлены в них глобальными, или код, не находящийся внутри функции. Следующий пример демонстрирует различия между локальными и глобальными переменными:

```
>>> def fun():
...     global a
...     a = 1
...     b = 2
...
```

В этом примере определяется функция, для которой `a` является глобальной переменной, а `b` — локальной. Эта функция пытается изменить `a` и `b`.

А теперь протестируем эту функцию:

```
>>> a = "one"
>>> b = "two"
>>> fun()
>>> a
1
>>> b
'two'
```

Команда присваивания `a` внутри `fun` является присваиванием глобальной переменной `a`, также существующей за пределами `fun`. Так как переменная `a` объявлена глобальной в `fun`, в результате присваивания глобальная переменная будет содержать значение `1` вместо значения `"one"`. С переменной `b` дело обстоит иначе; локальная переменная с именем `b` внутри `fun` сначала ссылается на то же значение,

что и переменная `b` за пределами `fun`, но в результате присваивания `b` начинает указывать на новое значение, локальное для функции `fun`.

Также существует команда `nonlocal`, близкая по смыслу к команде `global`, — она заставляет идентификатор обратиться к ранее связанной переменной в ближайшей внешней области видимости. Области видимости и пространства имен более подробно рассматриваются в главе 10, но суть в том, что `global` используется для переменной верхнего уровня, а `nonlocal` может ссылаться на любую переменную во внешней области видимости, как показывает пример в листинге 9.1.

Листинг 9.1. Файл nonlocal.py

```
g_var = 0
nl_var = 0
print("top level-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
def test():
    nl_var = 2
    print("in test-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
    def inner_test():
        global g_var
        nonlocal nl_var
        g_var = 1
        nl_var = 4
        print("in inner_test-> g_var: {0} nl_var: {1}".format(g_var,
                                                              nl_var))

    inner_test()
    print("in test-> g_var: {0} nl_var: {1}".format(g_var, nl_var))

test()
print("top level-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
```

g_var в inner_test связывается с g_var верхнего уровня

nl_var в inner_test связывается с nl_var в test

g_var в inner_test связывается с g_var верхнего уровня

nl_var в inner_test связывается с nl_var в test

При выполнении этот код выводит следующий результат:

```
top level-> g_var: 0 nl_var: 0
in test-> g_var: 0 nl_var: 2
in inner_test-> g_var: 1 nl_var: 4
in test-> g_var: 1 nl_var: 4
top level-> g_var: 1 nl_var: 0
```

Обратите внимание: значение `nl_var` верхнего уровня не изменилось, что произошло бы, если бы функция `inner_test` содержала строку `global nl_var`.

Итак, если вы хотите присвоить значение переменной, существующей за пределами функции, эту переменную следует явно объявить командой `nonlocal` или `global`. Но если вы обращаетесь к переменной, существующей внутри функции, объявлять ее командой `nonlocal` или `global` не нужно. Если Python не находит имя переменной в локальной области видимости функции, он пытается искать имя в глобальной области видимости. Таким образом, обращения к глобальным переменным автоматически переадресуются правильной глобальной переменной. Лично я не рекомендую использовать этот прием. Ваш код будет намного понятнее, если все глобальные переменные будут явно объявлены глобальными. Кроме

того, глобальные переменные в функциях лучше использовать только в отдельных редких случаях.

ПОПРОБУЙТЕ САМИ: ГЛОБАЛЬНЫЕ И ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

Если предположить, что `x = 5`, чему будет равно значение `x` после выполнения приведенной ниже функции `funct_1()`? А после выполнения `funct_2()`?

```
def funct_1():
    x = 3
def funct_2():
    global x
    x = 2
```

9.5. Присваивание функций переменным

Функции, как и другие объекты Python, могут присваиваться переменным:

```
>>> def f_to_kelvin(degrees_f):  <-- Определяет функцию f_to_kelvin
...     return 273.15 + (degrees_f - 32) * 5 / 9
...
>>> def c_to_kelvin(degrees_c):  <-- Определяет функцию c_to_kelvin
...     return 273.15 + degrees_c
...
>>> abs_temperature = f_to_kelvin  <-- Присваивает функцию переменной
>>> abs_temperature(32)
273.15
>>> abs_temperature = c_to_kelvin  <-- Присваивает функцию переменной
>>> abs_temperature(0)
273.15
```

Функции могут храниться в списках, кортежах или словарях:

```
>>> t = {'FtoK': f_to_kelvin, 'CtoK': c_to_kelvin} ❶
>>> t['FtoK'](32)  <-- Обращается к функции f_to_kelvin как к значению в словаре
273.15
>>> t['CtoK'](0)  <-- Обращается к функции c_to_kelvin как к значению в словаре
273.15
```

Переменная, ссылающаяся на функцию, может использоваться точно так же, как и функция ❶. Последний пример демонстрирует использование словаря для вызова разных функций в зависимости от значений строк, используемых в качестве ключей. Этот паттерн часто встречается в ситуациях, в которых требуется выбирать разные функции в зависимости от строкового значения; во многих случаях он заменяет структуру `switch` из таких языков, как C и Java.

9.6. Лямбда-выражения

Некоторые функции вроде только что приведенной также могут определяться с использованием лямбда-выражений в форме

```
lambda параметр1, параметр2, . . .: выражение
```

Лямбда-выражения представляют собой анонимные маленькие функции, которые могут быстро определяться «на месте». Часто такие маленькие функции требуется передавать другим функциям, как в случае с ключевой функцией, используемой методом сортировки списка. В таких случаях большие функции обычно не нужны, и было бы неудобно определять функцию где-то отдельно от места ее использования. Словарь из предыдущего подраздела может определяться в одном месте с определениями функций:

```
>>> t2 = {'FtoK': lambda deg_f: 273.15 + (deg_f - 32) * 5 / 9,
...       'CtoK': lambda deg_c: 273.15 + deg_c} ❶
>>> t2['FtoK'](32)
273.15
```

Этот пример определяет лямбда-выражения как значения в словаре ❶. Обратите внимание: лямбда-выражения не содержат команды `return`, потому что они автоматически возвращают значение выражения.

9.7. Функции-генераторы

Функция-*генератор* представляет собой особую разновидность функции, которую можно использовать для определения собственных итераторов. При определении функций-генераторов значение каждой итерации возвращается ключевым словом `yield`. Генератор перестает возвращать значения, когда итераций больше нет, при достижении пустой команды `return` или конца функции. Локальные переменные в функции-генераторе сохраняются между вызовами (в отличие от обычных функций):

```
>>> def four():
...     x = 0  <-- Присваивает x начальное значение 0
...     while x < 4:
...         print("in generator, x =", x)
...         yield x  <-- Возвращает текущее значение x
...         x += 1  <-- Увеличивает значение x
...
>>> for i in four():
...     print(i)
...
in generator, x = 0
0
in generator, x = 1
1
in generator, x = 2
2
in generator, x = 3
3
```

Заметим, что эта функция-генератор содержит цикл `while`, ограничивающий количество выполнений генератора. В зависимости от способа использования генератор, в котором не предусмотрено условие остановки, может создать бесконечный цикл в программе.

YIELD И YIELD FROM

Начиная с Python 3.3, наряду с ключевым словом `yield` появилось новое ключевое слово для генераторов — `yield from`. Фактически `yield` позволяет объединять генераторы в цепочку. `yield from` ведет себя так же, как `yield`, за исключением того, что вся механика генерирования делегируется субгенератору. Таким образом, в простейшем случае можно поступить так:

```
>>> def subgen(x):
...     for i in range(x):
...         yield i
...
>>> def gen(y):
...     yield from subgen(y)
...
>>> for q in gen(6):
...     print(q)
...
0
1
2
3
4
5
```

Этот пример позволяет вынести выражение `yield` из основного генератора, что упрощает его рефакторинг.

Функцию-генератор также можно использовать с оператором `in` для проверки того, входит ли значение в серию, созданную генератором.

```
>>> 2 in four()
in generator, x = 0
in generator, x = 1
in generator, x = 2
True
>>> 5 in four()
in generator, x = 0
in generator, x = 1
in generator, x = 2
in generator, x = 3
False
```

БЫСТРАЯ ПРОВЕРКА: ФУНКЦИИ-ГЕНЕРАТОРЫ

Что бы вы изменили в предыдущем коде функции `four()`, чтобы она работала для любого числа? Что нужно изменить в коде, чтобы начальное число последовательности тоже могло задаваться при вызове?

9.8. Декораторы

Так как функции являются полноценными объектами в Python, их можно присваивать переменным, как вы уже видели. Функции также могут передаваться в аргументах другим функциям и в возвращаемых значениях из других функций.

Например, можно написать функцию Python, которая получает другую функцию в параметре, «упаковывает» ее в другую функцию, которая делает нечто связанное, после чего возвращает новую функцию. Новая комбинация может использоваться вместо исходной функции:

```
>>> def decorate(func):
...     print("in decorate function, decorating", func.__name__)
...     def wrapper_func(*args):
...         print("Executing", func.__name__)
...         return func(*args)
...     return wrapper_func
...
>>> def myfunction(parameter):
...     print(parameter)
...
>>> myfunction = decorate(myfunction)
in decorate function, decorating myfunction
>>> myfunction("hello")
Executing myfunction
hello
```

Декоратор — синтаксическое удобство для таких процессов, позволяющее «упаковать» одну функцию внутри другой всего в одной строке. При этом эффект будет абсолютно таким же, как в приведенном коде, но полученный код является намного более элегантным и удобочитаемым.

Проще говоря, использование декоратора состоит из двух частей: определить функцию, которая будет включать, или «декорировать», другие функции, а затем поставить символ `@` с декоратором немедленно перед определением упаковываемой функции. Функция-декоратор должна получать функцию в параметре и возвращать функцию:

```
>>> def decorate(func):
...     print("in decorate function, decorating", func.__name__) ❶
...     def wrapper_func(*args):
...         print("Executing", func.__name__)
...         return func(*args)
...     return wrapper_func ❷
...
>>> @decorate ❸
... def myfunction(parameter):
...     print(parameter)
...
in decorate function, decorating myfunction
>>> myfunction("hello") ❹
Executing myfunction
hello
```

Функция `decorate` выводит имя упаковываемой функции при определении функции ❶. При завершении декоратор возвращает упакованную функцию ❷. `myfunction` декорируется маркером `@decorate` ❸. Упакованная функция вызывается после завершения функции-декоратора ❹.

Использование декоратора для упаковки одной функции внутри другой может быть удобно в нескольких отношениях. В веб-фреймворках (таких, как Django) декораторы используются для проверки того, что пользователь ввел регистрационные данные, перед выполнением функции, а в графических библиотеках декораторы могут использоваться для регистрации функции в графическом фреймворке.

ПОПРОБУЙТЕ САМИ: ДЕКОРАТОРЫ

Как бы вы изменили код функции-декоратора, чтобы она не выдавала лишние сообщения и заключала возвращаемое значение упакованной функции в теги `"<html>"` и `"</html>"` и чтобы вызов `myfunction("hello")` возвращал `"<html>hello<html>"`?

ПРАКТИЧЕСКАЯ РАБОТА 9: ПОЛЕЗНЫЕ ФУНКЦИИ

Вернитесь к практическим работам глав 6 и 7 и проведите рефакторинг, выделив код очистки и обработки данных в отдельные функции. В результате большая часть логики должна размещаться в функциях. Выбирайте функции и типы параметров на свое усмотрение, но помните, что функции должны решать только одну задачу без побочных эффектов, выходящих за границы функции.

Итоги

- К внешним переменным можно легко обращаться из функций при помощи команды `global`.
- Аргументы могут передаваться по позиции или по именам параметров.
- Для параметров функций могут определяться значения по умолчанию.
- Функции могут объединять аргументы в кортежи, что позволяет разработчику определять функции, получающие произвольное количество аргументов.
- Функции могут объединять аргументы в словари, что позволяет разработчику определять функции, получающие произвольное количество аргументов с передачей по имени параметра.
- Функции являются полноценными объектами в Python; это означает, что они могут присваиваться переменным, к ним можно обращаться через переменные и применять декораторы.

10 Модули и правила областей видимости

Эта глава охватывает следующие темы:

- ✓ Определение модуля
- ✓ Написание первого модуля
- ✓ Использование оператора импорта
- ✓ Изменение пути поиска модуля
- ✓ Создание имен в модулях
- ✓ Импорт стандартной библиотеки и сторонних модулей
- ✓ Понимание правил и пространств имен Python

Модули предназначены для структурирования больших проектов Python. Стандартная библиотека Python разбивается на модули, чтобы упростить работу с кодом. Разбивать код на модули не обязательно, но если вы пишете программы, занимающие больше нескольких страниц, или собираетесь использовать свой код повторно, вам стоит задуматься над этим.

10.1. Что такое модуль?

Модуль представляет собой файл с программным кодом. Он определяет группу функций Python или других объектов, а имя модуля определяется именем файла.

Модули часто содержат исходный код Python, но они также могут содержать откомпилированные объектные файлы C и C++. Откомпилированные модули и исходные модули Python используются одинаково.

Помимо группировки взаимосвязанных объектов Python, модули способствуют предотвращению конфликтов имен. Вы пишете для своей программы модуль с именем `mymodule`, который определяет функцию `reverse`. В той же программе может использоваться модуль `othermodule`, который также определяет функцию

с именем `reverse`, которая делает нечто отличное от вашей функции `reverse`. На языке без модулей невозможно использовать две разные функции с именем `reverse`. В Python это делается тривиально: вы просто обращаетесь в программе к функциям с именами `mymodule.reverse` и `othermodule.reverse`.

Использование имен модулей помогает отделить две функции `reverse` друг от друга, потому что Python использует пространства имен. *Пространство имен* фактически представляет собой словарь идентификаторов, доступных в блоке, функции, классе, модуле и т. д. Пространства имен более подробно рассматриваются в конце главы, а пока запомните, что каждый модуль имеет собственное пространство имен, предотвращающее конфликты имен.

Кроме того, модули упрощают работу с Python. Многие стандартные функции Python не встроены в основное ядро языка, а предоставляются конкретными модулями, загружаемыми по мере необходимости.

10.2. Первый модуль

Пожалуй, модули проще всего изучать на примере создания собственного модуля.

Создайте текстовый файл с именем mymath.py. Включите в этот текстовый файл код из листинга 10.1. (Если вы работаете в IDLE, выберите команду File ▸ New Window и начинайте вводить код, как показано на рис. 10.1.)

Листинг 10.1. Файл mymath.py

```
"""mymath - our example math module"""
pi = 3.14159
def area(r):
    """area(r): return the area of a circle with radius r."""
    global pi
    return(pi * r * r)
```

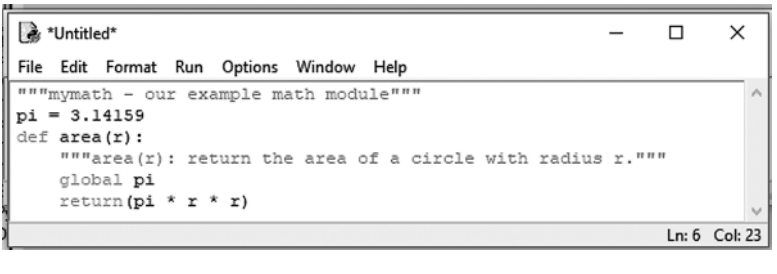


Рис. 10.1. Окно редактора IDLE предоставляет ту же функциональность редактирования, что и окно оболочки Python, включая автоматическую расстановку отступов и цветовое выделение

Сохраните этот код в каталоге, в котором находится исполняемый файл Python. Этот код просто присваивает значение переменной `pi` и определяет функцию. Всем файлам с кодом Python настоятельно рекомендуется присваивать суффикс .py;

по этому суффиксу интерпретатор Python опознает файлы с исходным кодом Python. Как и в случае с функциями, в первой строке модуля можно разместить строку документации.

Теперь запустите оболочку Python и введите следующие команды:

```
>>> pi
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'pi' is not defined
>>> area(2)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'area' is not defined
```

Иначе говоря, в Python нет встроенной константы `pi` или функции `area`.

Теперь введите следующие команды:

```
>>> import mymath
>>> pi
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'pi' is not defined
>>> mymath.pi
3.14159
>>> mymath.area(2)
12.56636
>>> mymath.__doc__
'mymath - our example math module'
>>> mymath.area.__doc__
'area(r): return the area of a circle with radius r.'
```

Вы подключили определения `pi` и `area` из файла mymath.py командой `import` (которая автоматически добавляет суффикс .py при поиске файла, определяющего модуль с именем `mymath`). Но новые определения недоступны для прямого обращения; при попытке ввести имя `pi` происходит ошибка, как и при попытке ввести `area(2)`. Вместо этого *перед* именами `pi` и `area` указывается имя содержащего их модуля, что гарантирует безопасность имен. Возможно, в программе загружен другой модуль, который тоже определяет `pi` (и возможно, автор этого модуля считает, что значение `pi` равно 3,14 или 3,14159265), но этот модуль ничему не мешает. Даже если этот модуль импортируется, к его версии `pi` придется обращаться по имени `othermodulename.pi`, которое отличается от `mymath.pi`. Такая форма доступа часто называется *уточнением* (то есть переменная `pi` уточняется именем модуля `mymath`). Также можно рассматривать `pi` как *атрибут* `mymath`.

Определения внутри модуля могут обращаться к другим определениям в этом модуле без указания имени модуля. Функция `mymath.area` обращается к константе `mymath.pi` по короткому имени `pi`.

При желании вы также можете явно импортировать имена из модуля так, что перед ними не нужно будет указывать имя модуля. Введите:

```
>>> from mymath import pi
>>> pi
3.14159
>>> area(2)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'area' is not defined
```

Имя `pi` становится непосредственно доступным, потому что вы конкретно запросили его командой `mymath import pi`. Тем не менее функцию `area` все равно придется вызывать в форме `mymath.area`, потому что она не была явно импортирована.

Возможно, вам захочется провести инкрементное тестирование модуля в процессе его создания в базовом интерактивном режиме или в оболочке Python среды IDLE. Однако при изменении модуля на диске повторное выполнение команды `import` не приведет к его повторной загрузке. Для этой цели нужно будет использовать функцию `reload` из модуля `importlib`. Модуль `importlib` предоставляет интерфейс к механизмам, лежащим в основе импортирования модулей:

```
>>> import mymath, importlib
>>> importlib.reload(mymath)
<module 'mymath' from '/home/doc/quickpythonbook/code/mymath.py'>
```

Когда модуль перезагружается (или импортируется в первый раз), Python разбирает весь его код. При обнаружении ошибок выдается синтаксическое исключение. С другой стороны, если все прошло нормально, создается файл .pyc (например, mymath.pyc) с байт-кодом Python.

При перезагрузке модуля программа не возвращается точно к такой же ситуации, как при запуске нового сеанса и первом импортировании модуля. Однако различия обычно не создают никаких проблем. Если эта тема вас заинтересует, за подробностями обращайтесь к описанию `reload` из раздела модуля `importlib` в справочнике *Python Language Reference* (https://docs.python.org/3/reference/import.html).

Конечно, модули могут использоваться не только из интерактивной оболочки Python. Их (и другие модули, если уж на то пошло) также можно импортировать в сценарии; вставьте соответствующие команды `import` в начало файла программы. Во внутренних механизмах Python интерактивный сеанс и сценарий тоже считаются модулями.

Подведем итог:

- Модуль представляет собой файл, определяющий объекты Python.
- Модуль `modulename` хранится в файле с именем modulename.py.
- Модуль с именем `modulename` подключается для использования командой `import modulename`. После выполнения команды к объектам, определенным в модуле, можно обращаться в форме `modulename.имя_объекта`.
- Также возможно подключить к программе конкретные имена из модуля командой `modulename import имя_объекта`. Эта команда позволяет напрямую обращаться к имени `имя_объекта` без указания префикса `modulename`; в частности, она удобна для часто используемых имен.

10.3. Команда import

Команда `import` существует в трех разных формах. Простейшая форма выглядит так:

```
import modulename
```

Она ищет модуль Python с заданным именем, разбирает его содержимое и делает его доступным для программы. Импортирующий код может использовать содержимое модуля, но любые обращения из этого кода к именам из модуля должны снабжаться префиксом с именем модуля. Если модуль с указанным именем не найден, происходит ошибка. О том, где именно Python ищет модули, будет рассказано в разделе 10.4.

Вторая форма разрешает явно импортировать имена из модуля в код:

```
from modulename import name1, name2, name3, . . .
```

Каждое из имен `name1`, `name2` и т. д. из модуля `modulename` становится доступным в импортирующем коде; код после команды `import` может использовать имена `name1`, `name2`, `name3` и т. д. без указания префикса с именем модуля.

Наконец, существует обобщенная форма `from… import…`:

```
from modulename import *
```

Символ `*` обозначает все экспортируемые имена из `modulename`. Команда `from modulename import *` импортирует все общедоступные имена из `modulename` (то есть имена, не начинающиеся с символа подчеркивания) и позволяет использовать их в импортирующем коде без указания префикса с именем модуля. Но если в модуле существует список имен `__all__` (или в файле `__init__.py` пакета), то эти имена будут импортироваться независимо от того, начинаются они с подчеркивания или нет.

Будьте осторожны при использовании этой конкретной формы импортирования. Если некоторое имя определяется в двух модулях и вы импортируете оба модуля в этой форме, возникнет конфликт имен и имя из второго модуля заместит имя из первого модуля. Кроме того, с этой формой читателю вашего кода будет труднее определить, откуда взялись используемые имена. С любой из двух предшествующих форм команды `import` вы передаете читателю конкретную информацию об их происхождении.

Впрочем, некоторые модули (например, `tkinter`) присваивают своим функциям имена, с которыми их происхождение становится очевидным, а риск конфликтов имен сводится к минимуму. Также обобщенная форма `import` нередко используется для того, чтобы избежать лишних нажатий клавиш в интерактивной оболочке.

10.4. Путь поиска модулей

Каталоги, в которых Python ищет модули, определяются в переменной с именем `path`, к которой можно обратиться через модуль с именем `sys`. Введите следующие команды:

```
>>> import sys
>>> sys.path
_список каталогов в пути поиска_
```

Значение, которое выводится в последней строке, зависит от конфигурации вашей системы. В любом случае строка содержит список каталогов, в которых Python проводит поиск (в указанном порядке) при выполнении команды `import`. Используется первый найденный модуль, удовлетворяющий запросу на импортирование. Если найти подходящий модуль не удается, инициируется исключение `ImportError`.

Если вы работаете в IDLE, путь поиска и содержащиеся в нем модули можно просмотреть в графическом виде в окне Path Browser, которое открывается из меню File окна оболочки Python.

Переменная `sys.path` инициализируется значением переменной среды (операционной системы) `PYTHONPATH`, если она существует, или же значением по умолчанию, зависящим от вашей установки. Кроме того, при выполнении сценария Python в начало переменной `sys.path` для этого сценария включается каталог, в котором находится сценарий; это позволяет удобно определить, в каком каталоге находится выполняемая программа Python. В интерактивном сеансе первому элементу `sys.path` присваивается пустая строка, что Python воспринимает как указание начать поиск модулей с текущего каталога.

10.4.1. Где следует размещать модули

В примере, приведенном в начале главы, модуль `mymath` доступен для Python, потому что (1) при интерактивном выполнении Python `sys.path` начинается с элемента `""`, приказывающего Python искать модули в текущем каталоге, и (2) Python был запущен из каталога, содержащего файл mymath.py. В условиях реальной эксплуатации ни одно из этих условий обычно не выполняется. Вы не будете запускать Python в интерактивном режиме, а файлы с кодом Python не будут находиться в текущем каталоге. Чтобы ваши программы могли использовать написанные вами модули, необходимо:

- Разместить модули в одном из каталогов, в которых Python обычно ищет модули.
- Разместить все модули, используемые программой Python, в одном каталоге с программой.
- Создать каталог (или каталоги) для хранения модулей и изменить переменную `sys.path`, чтобы она включала этот новый каталог (или каталоги).

Из всех трех вариантов первый, разумеется, реализуется проще всего; тем не менее этот вариант следует выбирать *только* в том случае, если ваша версия Python включает локальные каталоги с кодом в путь поиска модулей по умолчанию. Такие каталоги специально предназначены для кода, привязанного к вашей машине; они не будут перезаписаны при новой установке Python, потому что они не являются частью установки Python. Если ваша переменная `sys.path` включает такие каталоги, вы можете разместить свои модули в них.

Второй вариант хорошо подходит для модулей, связанных с конкретной программой. Просто храните такие модули вместе с программой.

Третий вариант выбирается для модулей, привязанных к конкретной площадке, которые будут использоваться более чем в одной программе на этой площадке. Переменную `sys.path` можно изменить разными способами. Ей можно присвоить значение в коде; сделать это несложно, но тогда местонахождение каталогов жестко фиксируется в программном коде. Можно задать переменную среды `PYTHONPATH`, что тоже относительно просто, но такое изменение может затронуть лишь часть пользователей; наконец, можно добавить значение в путь поиска по умолчанию при помощи файла .pth.

Примеры присваивания `PYTHONPATH` приведены в документации Python в разделе «Python Setup and Usage» (подраздел «Command line and environment»). Каталог или каталоги, заданные в этой переменной, подставляются в начало переменной `sys.path`. Если вы используете `PYTHONPATH`, будьте внимательны и избегайте определения модуля с таким же именем, как у одного из существующих библиотечных модулей, которые вы используете в программе. Если это произойдет, ваш модуль будет найден до библиотечного модуля. Иногда это именно то, что нужно, но скорее всего, такие случаи относительно редки.

Проблемы можно избежать при помощи файла .pth. В этом случае каталог или каталоги, добавленные вами, присоединяются к `sys.path`. Последний из этих механизмов лучше всего пояснить на примере. В Windows файл .pth можно поместить в каталог, на который указывает `sys.prefix`. Допустим, у вас `sys.prefix` указывает на каталог c:\program files\python; разместите файл из листинга 10.2 в этом каталоге.

Листинг 10.2. Файл myModules.pth

```
mymodules
c:\Users\naomi\My Documents\python\modules
```

При следующем запуске интерпретатора Python в переменную `sys.path` будут добавлены каталоги c:\program files\python\mymodules и c:\Users\naomi\My Documents\python\modules (если они существуют). Теперь вы можете размещать свои модули в этих каталогах. Обратите внимание: каталог mymodules все еще может быть заменен при новой установке. Каталог modules безопаснее для использования. Возможно, вам также придется переместить или создать файл mymodules.pth при обновлении Python. Если вам нужна более подробная информация об использовании файлов .pth, обращайтесь к описанию модуля `site` в справочнике «*Python Library Reference*».

10.5. Приватные имена в модулях

Ранее в этой главе я уже упоминала о том, что команда `from module import *` может импортировать *почти* все имена из модуля. Исключение составляют идентификаторы в модуле, начинающиеся с символа `_`: они не будут импортированы командой `from module import *`. Разработчики могут написать модули, предназначенные для импортирования командой `from module import *`, но при этом ограничить импортирование

некоторых функций или переменных. Если все внутренние имена (то есть имена, которые должны быть доступны только в пределах модуля) будут начинаться с символа подчеркивания, вы тем самым гарантируете, что команда `from module import *` импортирует только те имена, которые должны быть доступны для пользователя.

Чтобы увидеть, как работает этот механизм, представьте, что у вас имеется файл modtest.py со следующим кодом:

Листинг 10.3. Файл modtest.py

```
"""modtest: our test module"""
def f(x):
    return x
def _g(x):
    return x
a = 4
_b = 2
```

Теперь запустите интерактивный сеанс и введите следующие команды:

```
>>> from modtest import *
>>> f(3)
3
>>> _g(3)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name '_g' is not defined
>>> a
4
>>> _b
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name '_b' is not defined
```

Как видите, имена `f` и `a` импортируются, а имена `_g` и `_b` остаются скрытыми за пределами `modtest`. Помните, что это поведение встречается только с командой `from ... import *`. Для обращения к `_g` или `_b` можно поступить так:

```
>>> import modtest
>>> modtest._b
2
>>> from modtest import _g
>>> _g(5)
5
```

Схема с начальными подчеркиваниями, обозначающими приватные имена, применяется в Python, а не только в модулях.

10.6. Библиотечные и сторонние модули

В начале этой главы я уже упоминала о том, что стандартная поставка Python разбивается на модули для того, чтобы с ней было удобнее работать. После установки Python вы получаете доступ ко всей функциональности библиотечных модулей.

Все, что для этого потребуется, — импортировать подходящие модули, функции, классы и т. д. перед использованием.

В этой книге упоминаются многие часто используемые и полезные стандартные модули. Однако стандартная поставка Python включает гораздо больше модулей, чем описано в этой книге. Как минимум стоит просмотреть оглавление справочника «*Python Library Reference*».

В IDLE вы можете легко просмотреть документацию и найти информацию о модулях Python в окне Path Browser. Также можно найти примеры использования модулей в диалоговом окне Find in Files, открываемом из меню Edit окна оболочки Python. Этот способ также позволит провести поиск по вашим собственным модулям.

Доступные сторонние модули и ссылки на них приведены в каталоге Python Package Index (pyPI), который будет рассматриваться в главе 19. Чтобы эти модули стали доступными для импортирования в ваших программах, необходимо загрузить эти модули и установить их в каталоге, входящем в путь поиска модулей.

БЫСТРАЯ ПРОВЕРКА: МОДУЛИ

Предположим, имеется модуль с именем `new_math`, содержащий функцию с именем `new_divide`. Какими способами можно импортировать и использовать эту функцию? Какими достоинствами и недостатками обладает каждый способ?

Предположим, модуль `new_math` содержит вызов функции `_helper_math()`. Как начальный символ подчеркивания влияет на импортирование функции `_helper_math()`?

10.7. Правила областей видимости и пространств имен Python

Тема правил областей видимости и пространств имен Python станет более интересной по мере роста вашего опыта программирования на Python. Если вы только начинаете программировать на Python, вероятно, вам будет достаточно быстро просмотреть текст, чтобы понять основные идеи. За более подробной информацией обращайтесь к описанию *пространств имен* в «*Python Language Reference*».

Центральное место здесь занимает концепция пространства имен. *Пространство имен* в Python представляет собой отображение между идентификаторами и объектами, то есть способ хранения в Python информации об активных переменных и идентификаторах и о тех объектах, на которые они указывают. Таким образом, команда `x = 1` добавляет `x` в пространство имен (если переменная еще не входит в него) и связывает его со значением `1`. У блока кода, выполняемого в Python, есть три пространства имен: *локальное*, *глобальное* и *встроенное* (рис. 10.2).

Когда во время выполнения программы обнаруживается идентификатор, Python сначала ищет его в *локальном пространстве имен*. Если идентификатор не будет

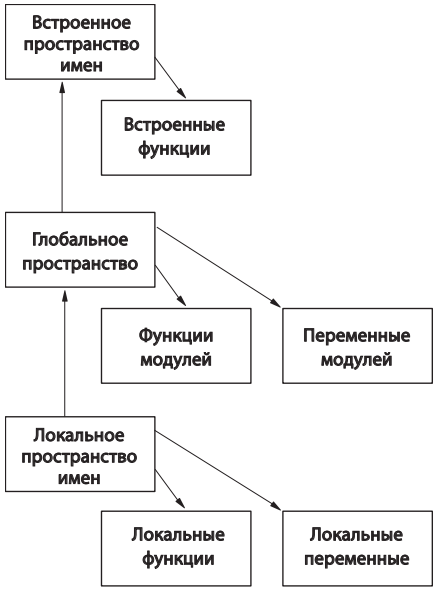


Рис. 10.2. Порядок проверки пространств имен при поиске идентификаторов

найден, поиск продолжается в *глобальном пространстве имен*. Если идентификатор и здесь не обнаружен, проверяется *встроенное пространство имен*. Если идентификатор не существует, Python решает, что произошла ошибка, и выдает исключение `NameError`.

Для модуля, команды, выполненной в интерактивном сеансе или в сценарии, запущенном из файла, глобальные и локальные пространства имен совпадают. Создание любой переменной или функции, или импортирование чего-либо из другого модуля приводит к появлению нового элемента (или *привязки*) в этом пространстве имен.

Но при вызове функции создается локальное пространство имен, в котором для каждого параметра вызова создается отдельная привязка (binding). Затем новые привязки вводятся в локальное пространство имен каждый раз, когда внутри функции создается переменная. Глобальное пространство имен функции представляет собой глобальное пространство имен вмещающего блока функции (то есть модуля, файла сценария или интерактивного сеанса). Оно не зависит от динамического контекста, в котором совершается вызов.

Во всех перечисленных ситуациях встроенным пространством имен является пространство имен модуля `__builtins__`. Среди прочего, этот модуль содержит все встроенные функции, уже встречавшиеся вам в книге (такие, как `len`, `min`, `max`, `int`, `float`, `list`, `tuple`, `range`, `str` и `repr`), и другие встроенные классы Python, например `NameError`.

Начинающих программистов Python иногда сбивает с толку тот факт, что вы можете переопределять элементы во встроенном модуле. Например, если создать

в программе список и сохранить его в переменной с именем `list`, вы в дальнейшем не сможете использовать встроенную функцию `list` — при поиске сначала будет обнаружена привязка для вашей переменной `list`. Имена функций, модулей и других объектов в этом отношении не различаются. Используется самая новая привязка для заданного идентификатора.

Хватит разговоров — пора рассмотреть несколько примеров. В примерах используются две встроенные функции: `locals` и `globals`. Эти функции возвращают словари, содержащие привязки локального и глобального пространства имен соответственно.

Запустите новый интерактивный сеанс:

```
>>> locals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__',
   '__doc__': None, '__package__': None}
>>> globals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__',
   '__doc__': None, '__package__': None}>>> >>> locals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__',
   '__doc__': None, '__package__': None}
>>> globals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__',
    '__doc__': None, '__package__': None}>>>
```

Локальное и глобальное пространства имен нового интерактивного сеанса совпадают. Они содержат три исходные пары «ключ–значение», предназначенные для внутреннего использования: (1) пустая строка документации `__doc__`, (2) имя основного модуля `__name__` (которое для интерактивных сеансов и сценариев, запущенных из файлов, всегда содержит `__main__`) и (3) модуль, используемый для встроенного пространства имен `__builtins__` (модуль `__builtins__`).

Если теперь создать переменную и импортировать из модулей, вы увидите, что в пространствах имен было создано несколько привязок:

```
>>> z = 2
>>> import math
>>> from cmath import cos
>>> globals()
{'cos': <built-in function cos>, '__builtins__': <module 'builtins'
  (built-in)>, '__package__': None, '__name__': '__main__', 'z': 2,
  '__doc__': None, 'math': <module 'math' from
  '/usr/local/lib/python3.0/libdynload/math.so'>}
>>> locals()
{'cos': <built-in function cos>, '__builtins__':
  <module 'builtins' (built-in)>, '__package__': None, '__name__':
  '__main__', 'z': 2, '__doc__': None, 'math': <module 'math' from
  '/usr/local/lib/python3.0/libdynload/math.so'>}
>>> math.ceil(3.4)
4
```

Как и ожидалось, локальное и глобальное пространство имен остаются эквивалентными. В них были добавлены элементы для `z` как числа, `math` как модуля и `cos` из модуля `cmath` как функции.

Вы можете воспользоваться командой `del` для удаления этих новых привязок из пространства имен (включая привязки модулей, созданные командами `import`):

```
>>> del z, math, cos
>>> locals()
{'__builtins__': <module 'builtins' (built-in)>, '__package__': None,
'__name__': '__main__', '__doc__': None}
>>> math.ceil(3.4)
Traceback (innermost last):
  File "<stdin>", line 1, in <module>
NameError: math is not defined
>>> import math
>>> math.ceil(3.4)
4
```

Ничего критичного в удалении нет, потому что вы можете снова импортировать модуль `math` и использовать его. Такое использование `del` может быть особенно удобным в интерактивном режиме[1].

Для самых отчаянных: да, команда `del` может использоваться для удаления элементов `__doc__`, `__main__` и `__builtins__`. Не надо так делать, для вашего сеанса это добром не кончится!

А теперь присмотримся к функции, созданной в интерактивном сеансе:

```
>>> def f(x):
...     print("global: ", globals())
...     print("Entry local: ", locals())
...     y = x
...     print("Exit local: ", locals())
...
>>> z = 2
>>> globals()
{'f': <function f at 0xb7cbfeac>, '__builtins__': <module 'builtins'
  (built-in)>, '__package__': None, '__name__': '__main__', 'z': 2,
  '__doc__': None}
>>> f(z)
global:  {'f': <function f at 0xb7cbfeac>, '__builtins__': <module
  'builtins' (built-in)>, '__package__': None, '__name__': '__main__',
  'z': 2, '__doc__': None}
Entry local:  {'x': 2}
Exit local:  {'y': 2, 'x': 2}
>>>
```

Если как следует разобраться в этой путанице, вы увидите, что, как и предполагалось, при входе параметр `x` входит в число начальных элементов локального пространства имен `f`, но переменная `y` добавляется позднее. Глобальное пространство имен соответствует глобальному пространству имен вашего интерактивного сеанса,

[1] Вызов `del` с последующим повторным импортированием не отразит изменения, внесенные в модуль на диске. При этом модуль не стирается из памяти с последующей повторной загрузкой с диска: привязка сначала удаляется, а затем возвращается в пространство имен. Если вы хотите отразить изменения, внесенные в файл, используйте importlib.reload.

в котором была определена функция `f`. Обратите внимание: пространство имен также содержит переменную `z`, которая была определена после `f`.

В среде реальной эксплуатации обычно вызываются функции, определенные в модулях. Их глобальное пространство имен соответствует глобальному пространству имен модуля, в котором определяются функции. Создайте файл с кодом из листинга 10.4.

Листинг 10.4. Файл scopetest.py

```
"""scopetest: тестовый модуль для области видимости"""
v = 6
def f(x):
    """f: тестовая функция"""
    print("global: ", list(globals().keys()))
    print("entry local:", locals())
    y = x
    w = v
    print("exit local:", locals().keys())
```

Обратите внимание: мы выводим только ключи (идентификаторы) словаря, возвращаемого `globals`, для того чтобы не загромождать результаты. Выводятся только ключи, потому что модули оптимизируются для хранения всего словаря `__builtins__` как значения ключа `__builtins__`:

```
>>> import scopetest
>>> z = 2
>>> scopetest.f(z)
global:  ['__name__', '__doc__', '__package__', '__loader__', '__spec__',
     '__file__', '__cached__', '__builtins__', 'v', 'f']
entry local: {'x': 2}
exit local: dict_keys(['x', 'w', 'y'])
```

Теперь глобальное пространство имен соответствует пространству имен модуля `scopetest`, и оно включается в функцию `f` и переменную `v` (но не переменную `z` из интерактивного сеанса). Таким образом, при создании модуля вы можете полностью контролировать пространства имен его функций.

Мы рассмотрели локальные и глобальные пространства имен. Пора перейти к встроенному пространству имен. В этом примере встречается другая встроенная функция — `dir`, которая для заданного модуля возвращает список определенных в нем имен:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
     'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
     'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
     'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
     'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
     'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
     'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
     'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
     'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
     'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
```

```
'NotADirectoryError', 'NotImplemented', 'NotImplementedError',
'OSError', 'OverflowError', 'PendingDeprecationWarning',
'PermissionError', 'ProcessLookupError', 'RecursionError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'ZeroDivisionError', '__build_class__', '__debug__', '__doc__',
'__import__', '__loader__', '__name__', '__package__', '__spec__',
'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes',
'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright',
'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr',
'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals',
'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open',
'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed',
'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str',
'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Список получился довольно длинным. Элементы, заканчивающиеся суффиксами `Error` и `Exit`, — имена исключений, встроенных в Python. Они будут рассмотрены в главе 14.

Последнюю группу (от `abs` до `zip`) образуют встроенные функции Python. Многие из этих функций уже встречались вам в книге, другие еще встретятся, но я не буду описывать все эти функции здесь. При необходимости вы найдете остальные описания в справочнике «*Python Library Reference*». Также можно легко получить строку документации для любого из них при помощи функции `help()` (или вывести ее напрямую):

```
>>> print(max.__doc__)
max(iterable[, key=func]) -> value
max(a, b, c, ...[, key=func]) -> value

With a single iterable argument, return its largest item.
With two or more arguments, return the largest argument.
```

Как упоминалось ранее, начинающие программисты Python иногда случайно переопределяют встроенные функции:

```
>>> list("Peyto Lake")
['P', 'e', 'y', 't', 'o', ' ', 'L', 'a', 'k', 'e']
>>> list = [1, 3, 5, 7]
>>> list("Peyto Lake")
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: 'list' object is not callable
```

Интерпретатор Python прекращает поиск после новой привязки `list`, хотя вы используете синтаксис встроенной функции `list`.

Конечно, то же самое произойдет при попытке использовать один идентификатор дважды в одном пространстве имен. Предыдущее значение будет заменено независимо от его типа:

```
>>> import mymath
>>> mymath = mymath.area
>>> mymath.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'function' object has no attribute 'pi'
```

Если вы знаете об этой ситуации, особых проблем не будет. В конце концов, повторное использование идентификаторов даже для объектов разных типов все равно не помогает чтению кода. Если же вы непреднамеренно совершите подобную ошибку в интерактивном режиме, проблема легко решается: удалите нежелательную привязку командой `del`, чтобы снова получить доступ к переопределенному встроенному имени, или заново импортируйте свой модуль для восстановления доступа:

```
>>> del list
>>> list("Peyto Lake")
['P', 'e', 'y', 't', 'o', ' ', 'L', 'a', 'k', 'e']
>>> import mymath
>>> mymath.pi
3.14159
```

Функции `locals` и `globals` могут стать простыми средствами отладки. Функция `dir` не возвращает текущую конфигурацию, но при вызове без параметров она возвращает отсортированный список идентификаторов в локальном пространстве имен. Этот прием поможет вам в поиске опечаток в именах переменных, которые обычно автоматически обнаруживаются компилятором в языках с обязательными объявлениями:

```
>>> x1 = 6
>>> xl = x1 - 2
>>> x1
6
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
     '__package__', '__spec__', 'x1', 'xl']
```

В отладчике, включенном в IDLE, предусмотрена функция просмотра локальных и глобальных переменных при пошаговом выполнении кода; она выводит результаты функций `locals` и `globals`.

БЫСТРАЯ ПРОВЕРКА: ПРОСТРАНСТВА ИМЕН И ОБЛАСТИ ВИДИМОСТИ

Имеется переменная `width`, определенная в модуле make_window.py. В каком из следующих контекстов `width` находится в области видимости?

(А) В самом модуле.

(Б) Внутри функции `resize()` из этого модуля.

(В) Внутри сценария, импортировавшего модуль make_window.py.

ПРАКТИЧЕСКАЯ РАБОТА 10: СОЗДАНИЕ МОДУЛЯ

Упакуйте функции, созданные в конце главы 9, в автономный модуль. Хотя код выполнения модуля может быть запущен как основная программа, ваша цель — добиться того, чтобы функции могли использоваться из других сценариев.

Итоги

- Модули Python позволяют разместить взаимосвязанный код и объекты в файле.
- Модули также способствуют предотвращению конфликтов имен, потому что импортированные объекты обычно используются с указанием имен их модулей.

11 Программы Python

Эта глава охватывает следующие темы:

- ✓ Создание простейшей программы
- ✓ Создание программы, выполняемой непосредственно в Linux/UNIX
- ✓ Написание программ для macOS
- ✓ Выбор параметров выполнения в Windows
- ✓ Объединение программ и модулей
- ✓ Распространение приложений на Python

До настоящего момента вы использовали интерпретатор Python исключительно в интерактивном режиме. Для реального использования придется создавать программы Python или сценарии. Некоторые разделы этой главы посвящены программам командной строки. Если у вас имеется опыт работы на платформах Linux/UNIX, возможно, вам знакомы сценарии, которые запускаются из командной строки и получают аргументы, которые могут использоваться для передачи информации, а также для возможного перенаправления ввода и вывода. Если же вы работали на платформах Windows или Mac, эта концепция может показаться новой, а ее ценность — неочевидной.

Действительно, в графических средах сценарии командной строки менее удобны. Тем не менее на Mac поддерживается терминал UNIX, Windows также предоставляет расширенный инструментарий командной строки. В какой-то момент вам будет полезно прочитать основной материал этой главы. Возможно, вы попадете в ситуацию, в которой эти приемы будут полезны, или вам нужно будет разобраться в коде, в котором они используются. В частности, средства командной строки пригодятся при обработке большого количества файлов.

11.1. Создание простейшей программы

Любая группа команд Python, последовательно размещенных в файле, может использоваться в качестве программы, или *сценария* (script). Тем не менее более стандартный и практичный подход основан на введении дополнительной структуры. В своей простейшей форме эта задача сводится к простому созданию управляющей функции в файле и вызову этой функции.

Листинг 11.1. Файл script1.py

```
def main():  <-- Управляющая функция main()
    print("this is our first test script file")
main()  <-- Вызывает main
```

В этом сценарии `main` является управляющей — и единственной — функцией. Сначала эта функция определяется, а затем она вызывается. Хотя в маленькой программе это ни на что не влияет, такая структура предоставляет больше возможностей и большую свободу действий при создании крупных приложений, поэтому лучше сразу выработать в себе эту привычку.

11.1.1. Запуск сценария из командной строки

Если вы используете Linux/UNIX, убедитесь в том, что каталог Python включен в путь поиска, а текущим является каталог с вашим сценарием. Введите в командной строке следующую команду для запуска сценария:

```
python script1.py
```

Если вы используете Macintosh с OS X, процедура запуска будет такой же, как в других системах UNIX. Запустите программу терминала (она находится в папке Utilities папки Applications). Также в OS X предусмотрено несколько других вариантов запуска сценариев, которые будут описаны выше.

Если вы работаете в Windows, откройте приглашение командной строки (команда может находиться в разных меню в зависимости от версии Windows, в Windows 10 она находится в меню Windows System) или PowerShell. В обоих случаях приглашение открывается в домашней папке, а при необходимости вы можете воспользоваться командой `cd` для перехода в подкаталог. Если сценарий script1.py был сохранен на рабочем столе, процесс запуска выглядит примерно так:

```
C:\Users\naomi> cd Desktop  <-- Переходит в папку Desktop

C:\Users\naomi\Desktop> python script1.py  <-- Запускает script1.py
this is our first test script file

C:\Users\naomi\Desktop>  <-- Вывод script1.py
```

Другие способы запуска сценариев будут рассмотрены позднее в этой главе, а пока ограничимся этим вариантом.

11.1.2. Аргументы командной строки

У сценариев существует простой механизм передачи аргументов командной строки:

Листинг 11.2. Файл script2.py

```
import sys
def main():
    print("this is our second test script file")
    print(sys.argv)
main()
```

При запуске командой

```
python script2.py arg1 arg2 3
```

вы получите следующий результат:

```
this is our second test script file
['script2.py', 'arg1', 'arg2', '3']
```

Как видите, аргументы командной строки хранятся в `sys.argv` в виде списка строк.

11.1.3. Перенаправление ввода и вывода сценария

Ввод и/или вывод сценария можно перенаправить из командной строки. Для демонстрации перенаправления мы воспользуемся следующим коротким сценарием.

Листинг 11.3. Файл replace.py

```
import sys
def main():
   contents = sys.stdin.read()
   sys.stdout.write(contents.replace(sys.argv[1], sys.argv[2]))
main()
```

Читает данные из stdin в contents

Заменяет первый аргумент вторым

Этот сценарий читает свой стандартный ввод и записывает прочитанные данные в стандартный вывод, заменяя все вхождения первого аргумента вторым аргументом. При следующем вызове сценарий помещает в `outfile` копию `infile`, в которой все вхождения `zero` заменяются `0`:

```
python replace.py zero 0 < infile > outfile
```

Обратите внимание: этот сценарий работает в UNIX, но в Windows перенаправление ввода и/или вывода работает только при запуске сценария из окна командной строки.

В общем случае строка

```
python script.py arg1 arg2 arg3 arg4 < infile > outfile
```

означает, что весь ввод операций `sys.stdin` будет получаться из `infile`, а весь вывод операций `sys.stdout` направляется в `outfile`. Эффект будет таким, как если бы вы

связали sys.stdin с infile в режиме 'r' (чтение), а sys.stdout — с outfile в режиме 'w' (запись):

```
python replace.py a A < infile >> outfile
```

С этой командой вывод будет присоединяться к outfile (вместо перезаписи, как в предыдущем примере).

Также вывод одной команды можно *передать* на ввод другой команды:

```
python replace.py 0 zero < infile | python replace.py 1 one > outfile
```

В результате выполнения этого кода в outfile будет сохранено содержимое infile, в которой все вхождения 0 заменяются строкой zero, а все вхождения 1 заменяются строкой one.

11.1.4. Модуль argparse

Сценарий можно настроить так, чтобы он получал не только аргументы, но и ключи командной строки. Модуль argparse обеспечивает поддержку разных типов аргументов и даже может генерировать содержательные сообщения.

Чтобы использовать модуль argparse, создайте экземпляр ArgumentParser, заполните его аргументами, а затем прочитайте как необязательные, так и позиционные аргументы. Листинг 11.4 демонстрирует использование модуля.

Листинг 11.4. Файл opts.py

```
from argparse import ArgumentParser

def main():
    parser = ArgumentParser()
    parser.add_argument("indent", type=int, help="indent for report")
    parser.add_argument("input_file", help="read data from this file") ❶
    parser.add_argument("-f", "--file", dest="filename", ❷
                  help="write report to FILE", metavar="FILE")
    parser.add_argument("-x", "--xray",
                  help="specify xray strength factor")
    parser.add_argument("-q", "--quiet",
                  action="store_false", dest="verbose", default=True, ❸
                  help="don't print status messages to stdout")

    args = parser.parse_args()

    print("arguments:", args)
main()
```

Код создает экземпляр ArgumentParser и добавляет два позиционных аргумента, indent и input_file, которые будут введены после разбора всех необязательных аргументов. *Позиционные аргументы* не имеют префикса (обычно "-") и являются обязательными, и в этом случае аргумент indent также должен преобразоваться в int ❶.

Следующая строка добавляет необязательный аргумент с именем файла с префиксом `'-f'` или `'--file'` ❷. Последний ключ командной строки `"quiet"` также добавляет возможность отключить режим подробного вывода, который по умолчанию включен (`action="store_false"`). Тот факт, что эти параметры начинаются с префиксного символа `"-"`, сообщает парсеру, что они являются необязательными.

Последний аргумент `"-q"` также имеет значение по умолчанию (`True` в данном случае), которое будет использоваться в том случае, если этот ключ не задан. Параметр `action="store_false"` указывает, что если аргумент задан, будет использовано значение `False` ❸.

Модуль `argparse` возвращает объект `Namespace`, содержащий аргументы как атрибуты. Для получения значений аргументов используется точечная запись. Если аргумент не задан, значение равно `None`. Таким образом, при запуске предыдущего сценария командой

```
python opts.py -x100 -q -f outfile 2 arg2  ◄—  Параметры перечисляются после имени сценария
```

будет получен следующий вывод:

```
arguments: Namespace(filename='outfile', indent=2, input_file='arg2',
     verbose=False, xray='100')
```

Если будет обнаружен недействительный аргумент или если обязательный аргумент не задан, `parse_args` выдает ошибку:

```
python opts.py -x100 -r
```

Эта команда выводит следующий результат:

```
usage: opts.py [-h] [-f FILE] [-x XRAY] [-q] indent input_file
opts.py: error: the following arguments are required: indent, input_file
```

11.1.5. Использование модуля fileinput

Модуль `fileinput` также бывает полезным в сценариях. Он обеспечивает поддержку обработки входных данных, содержащихся в одном или нескольких файлах. Он автоматически читает аргументы командной строки (из `sys.argv`) и интерпретирует их как список входных файлов. Затем модуль позволяет последовательно перебрать эти строки. Простой пример сценария в листинге 11.5 (который отсекает все строки, начинающиеся с `##`) демонстрирует простейшее использование модуля.

Листинг 11.5. Файл script4.py

```
import fileinput
def main():
    for line in fileinput.input():
        if not line.startswith('##'):
            print(line, end="")
main()
```

Предположим, также имеются файлы данных, показанные в листингах 11.6 и 11.7.

Листинг 11.6. Файл sole1.tst

```
## sole1.tst: тестовые данные для функции sole
0 0 0
0 100 0
##
0 100 100
```

Листинг 11.7. Файл sole2.tst

```
## sole2.tst: другие тестовые данные для функции sole
12 15 0
##
100 100 0
```

Также предположим, что сценарий запускается следующей командой:

```
python script4.py sole1.tst sole2.tst
```

Вы получите следующий результат (строки комментариев удалены, а данные двух файлов объединены):

```
0 0 0
0 100 0
0 100 100
12 15 0
100 100 0
```

Если аргументы командной строки отсутствуют, то данные читаются только из стандартного ввода. Если одним из аргументов является дефис (`-`), то в этой точке читаются данные из стандартного ввода.

Модуль предоставляет ряд других функций. Эти функции позволяют в любой момент определить общее количество прочитанных строк (`lineno`), количество строк, прочитанных из текущего файла (`filelineno`), имя текущего файла (`filename`), признаки нахождения в первой строке файла (`isfirstline`) и/или чтения стандартного ввода в настоящий момент (`isstdin`). Вы можете в любой момент перейти к следующему файлу (`nextfile`) или закрыть весь поток (`close`). Короткий сценарий в листинге 11.8 (он объединяет строки входных файлов и добавляет разделители начала файла) демонстрирует использование этих функций.

Листинг 11.8. Файл script5.py

```
import fileinput
def main():
    for line in fileinput.input():
        if fileinput.isfirstline():
            print("<start of file {0}>".format(fileinput.filename()))
        print(line, end="")
main()
```

Команда

```
python script5.py file1 file2
```

выводит следующий результат (пунктирные линии обозначают строки исходных файлов):

```
<start of file file1>
.....................
.....................
<start of file file2>
.....................
.....................
```

Наконец, если вызвать `fileinput.input` с одним аргументом с именем файла или списком имен файлов, они будут использованы в качестве входных файлов вместо аргументов из `sys.argv`. `fileinput.input` также поддерживает ключ `inplace`, который оставляет свой вывод в том файле, из которого были прочитаны входные данные (также существует возможность сохранения оригинала в резервном файле). За описанием обращайтесь к документации.

БЫСТРАЯ ПРОВЕРКА: СЦЕНАРИИ И АРГУМЕНТЫ

Соедините варианты взаимодействия с командной строкой с правильными ситуациями для их использования.

Несколько аргументов и ключей	`sys.argv`
Без аргументов или только один аргумент	Использование модуля `file_input`
Обработка нескольких файлов	Перенаправление стандартного ввода и вывода
Использование сценария в качестве фильтра	Использование модуля `argparse`

11.2. Прямое исполнение сценариев в UNIX

Если вы работаете на платформе UNIX, сценарий можно легко сделать напрямую исполняемым. Включите следующую строку в начало файла и измените его разрешения (команда `chmod +x replace.py`):

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

Учтите, что если Python 3.*x* не является вашей версией Python по умолчанию, возможно, вам придется заменить `python` в этом примере на `python3`, `python3.6` или на что-нибудь в этом роде, чтобы приказать использовать Python 3.*x* вместо более ранней версии по умолчанию.

Если теперь разместить сценарий где-то в пути поиска (например, в каталог `bin`), его можно будет выполнить независимо от того, какой каталог является текущим; для этого введите имя файла и нужные аргументы:

```
replace.py zero 0 < infile > outfile
```

На платформе UNIX вы сможете использовать перенаправление ввода и вывода, а если вы используете еще и современный командный интерпретатор — историю команд и автозавершение.

Если вы пишете административные сценарии в UNIX, некоторые библиотечные модули будут вам особенно полезны. К числу таких модулей относится модуль `grp` для обращения к базе данных групп, `pwd` для обращения к базе данных паролей, `resource` для обращения к информации об использовании ресурсов, `syslog` для работы с инструментарием `syslog` и `stat` для работы с информацией о файле или каталоге, полученной в результате вызова `os.stat`. Информацию об этих модулях можно найти в справочнике «*Python Library Reference*».

11.3. Сценарии в macOS

Во многих отношениях сценарии Python в macOS ведут себя так же, как и в Linux/UNIX. Вы можете запускать сценарии Python из окна терминала точно так же, как на любой машине с UNIX. При этом на Mac вы можете запускать программы Python из Finder — либо перетащив файл сценария на приложение Python Launcher, либо настроив Python Launcher как приложение по умолчанию для открытия сценария (а возможно, всех файлов с расширением .py).

Существует несколько вариантов использования Python на Mac. Описание всех возможностей выходит за рамки книги, но вы можете получить полную информацию, посетив сайт www.python.org и ознакомившись с подразделом Mac раздела «*Using Python*» документации вашей версии Python. Также в разделе 11.6 документации «*Distributing Python applications*» приведена более подробная информация о распространении приложений и библиотек Python для платформы Mac.

Если вас интересует тема написания административных сценариев для macOS, обратите внимание на пакеты, заполняющие пробел между Apple OSA (Open Scripting Architectures) и Python. В частности, к этой категории относятся пакеты `appscript` и `PyOSA`.

11.4. Возможности выполнения сценариев в Windows

Если вы работаете в Windows, у вас есть несколько вариантов запуска сценариев, различающихся как по функциональности, так и по простоте использования. К сожалению, конкретный состав этих возможностей и их настройка существенно зависят от версий Windows. В этой книге мы сосредоточимся на запуске Python на Windows из командной строки или PowerShell. За информацией о других возможностях запуска Python в вашей системе обращайтесь к сетевой документации Python для вашей версии (найдите раздел «*Using Python on Windows*»).

11.4.1. Запуск сценария из окна командной строки или PowerShell

Чтобы запустить сценарий из окна командной строки или окна PowerShell, откройте окно командной строки или окно PowerShell. Если вы открыли окно командной строки и перешли в папку, в которой хранятся сценарии, вы сможете использовать Python для запуска сценариев по аналогии с системами UNIX/Linux/MacOS:

```
> python replace.py zero 0 < infile > outfile
```

PYTHON НЕ ЗАПУСКАЕТСЯ?

Если Python не запускается при вводе команды `python` в окне командной строки Windows, скорее всего, это связано с тем, что исполняемый файл Python не включен в путь поиска, поэтому либо добавьте исполняемый файл Python в переменную среду PATH вашей системы вручную, либо снова запустите программу установки, чтобы она выполнила работу за вас. За дополнительной информацией о настройке Python в Windows обращайтесь к разделу «Python Setup and Usage» сетевой документации Python. Здесь вы найдете раздел об использовании Python в Windows, содержащий инструкции по установке Python.

Это самый гибкий из способов запуска сценариев в Windows, потому что он позволяет использовать перенаправление ввода и вывода.

11.4.2. Другие средства Windows

Также стоит обратить внимание на другие способы. Если вы умеете писать пакетные файлы, вы можете включить команды в них. В инструментарий Cygwin включена портированная версия оболочки GNU BASH, о которой можно найти информацию по адресу www.cygwin.com; она предоставляет функциональность оболочки в стиле UNIX для Windows.

В системе Windows можно отредактировать переменные среды (см. предыдущий раздел) и добавить .py в список расширений, чтобы сценарии могли запускаться автоматически:

```
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.JS;.PY
```

ПОПРОБУЙТЕ САМИ: РАЗРЕШЕНИЕ ИСПОЛНЕНИЯ СЦЕНАРИЯ

Поэкспериментируйте с запуском сценариев на вашей платформе. Также попробуйте перенаправить ввод и вывод для ваших сценариев.

11.5. Программы и модули

Для небольших сценариев, содержащих всего несколько строк кода, достаточно и одной функции. Но если сценарий вырастает за пределы этого размера, желательно отделить управляющую функцию от остального кода. Оставшаяся часть этого раздела демонстрирует этот прием и некоторые его преимущества. Начнем с примера, в котором используется простая управляющая функция. Сценарий в листинге 11.9 возвращает англоязычное название для заданного числа в диапазоне от 0 до 99.

Листинг 11.9. Файл script6.py

```
#! /usr/bin/env python3
import sys
# Соответствия для преобразований
_1to9dict = {'0': '', '1': 'one', '2': 'two', '3': 'three', '4': 'four',
             '5': 'five', '6': 'six', '7': 'seven', '8': 'eight',
```

```
             '9': 'nine'}
_10to19dict = {'0': 'ten', '1': 'eleven', '2': 'twelve',
               '3': 'thirteen', '4': 'fourteen', '5': 'fifteen',
               '6': 'sixteen', '7': 'seventeen', '8': 'eighteen',
               '9': 'nineteen'}
_20to90dict = {'2': 'twenty', '3': 'thirty', '4': 'forty', '5': 'fifty',
               '6': 'sixty', '7': 'seventy', '8': 'eighty', '9': 'ninety'}
def num2words(num_string):
    if num_string == '0':
        return'zero'
    if len(num_string) > 2:
        return "Sorry can only handle 1 or 2 digit numbers"
    num_string = '0' + num_string ◄— Дополняется слева на случай, если число состоит из одной цифры
    tens, ones = num_string[-2], num_string[-1]
    if tens == '0':
        return _1to9dict[ones]
    elif tens == '1':
        return _10to19dict[ones]
    else:
        return _20to90dict[tens] + ' ' + _1to9dict[ones]
def main():
    print(num2words(sys.argv[1])) ❶
main()
```

Если запустить программу командой

```
python script6.py 59
```

вы получите следующий результат:

```
fifty nine
```

Управляющая функция вызывает функцию `num2words` с соответствующим аргументом и выводит результат ❶. Вызов принято размещать в нижней части, но иногда определение управляющей функции встречается в начале файла. Я предпочитаю определять эту функцию внизу, непосредственно над вызовом, чтобы мне не приходилось прокручивать код и искать ее после того, как я узнаю ее имя в нижней части листинга. Кроме того, такая практика четко отделяет служебный код сценария от остальной части файла, что может быть полезно при объединении сценариев и модулей.

Люди объединяют сценарии с модулями, когда они хотят предоставить доступ к функциям, созданным ими в сценарии, другим модулям или сценариям. Кроме того, модуль может быть организован так, чтобы он мог запускаться как сценарий — либо для того, чтобы предоставить пользователям простой интерфейс, либо для создания точек входа для автоматического тестирования.

Объединение сценария с модулем сводится к простому включению проверки условия для управляющей функции:

```
if __name__ == '__main__':
    main()
else:
    # Специфический код инициализации для модуля (если нужно)
```

При запуске в качестве сценария будет использовано имя `__main__` и будет вызвана управляющая функция `main`. Если проверка была импортирована в интерактивный сеанс или другой модуль, то имя будет соответствовать имени файла.

При создании сценария я часто с самого начала также оформляю его в виде модуля. Эта практика позволяет импортировать его в сеанс и в интерактивном режиме тестировать и отлаживать мои функции в процессе их создания. Внешняя отладка потребуется только для управляющей функции. По мере роста сценария или когда я начинаю писать функции, которые можно было бы использовать в других местах, я могу выделить эти функции в отдельный модуль, который будет импортироваться другими модулями.

Сценарий в листинге 11.10 расширяет предыдущий сценарий, но в него также внесены изменения, чтобы он мог использоваться в качестве модуля. Функциональность была расширена, чтобы модуль позволял вводить числа в диапазоне от 0 до 999999999999999 (вместо диапазона от 0 до 99). Управляющая функция (`main`) проверяет аргумент и удаляет из него все запятые, чтобы при вводе можно было использовать более понятную для читателя форму записи 1,234,567.

Листинг 11.10. Файл n2w.py

```
#! /usr/bin/env python3
"""n2w: модуль для преобразования чисел в словесное описание; содержит
   функцию num2words. Также может выполняться как сценарий.
Использование в качестве сценария: n2w num  <-- Справка по использованию; включает пример
           (Преобразует число в его описание на английском языке)
           num: целое число в диапазоне от 0 до 999,999,999,999
           (запятые не обязательны).
example: n2w 10,003,103
           for 10,003,103 say: ten million three thousand one hundred three
"""
import sys, string, argparse
_1to9dict = {'0': '', '1': 'one', '2': 'two', '3': 'three', '4': 'four',
             '5': 'five', '6': 'six', '7': 'seven', '8': 'eight',
             '9': 'nine'}                                          <-- Соответствия для преобразований
_10to19dict = {'0': 'ten', '1': 'eleven', '2': 'twelve',
               '3': 'thirteen', '4': 'fourteen', '5': 'fifteen',
               '6': 'sixteen', '7': 'seventeen', '8': 'eighteen',
               '9': 'nineteen'}
_20to90dict = {'2': 'twenty', '3': 'thirty', '4': 'forty', '5': 'fifty',
               '6': 'sixty', '7': 'seventy', '8': 'eighty', '9': 'ninety'}
_magnitude_list = [(0, ''), (3, ' thousand '), (6, ' million '),
                   (9, ' billion '), (12, ' trillion '),(15, '')]
def num2words(num_string):
    """num2words(num_string): convert number to English words"""
    if num_string == '0':  <-- Обрабатывает специальные случаи (число равно 0 или слишком велико)
        return 'zero'
    num_string = num_string.replace(",", "")  <-- Удаляет запятые из чисел
    num_length = len(num_string)
    max_digits = _magnitude_list[-1][0]
    if num_length > max_digits:
```

```
        return "Sorry, can't handle numbers with more than  " \
               "{0} digits".format(max_digits)
    num_string = '00' + num_string  <-- Дополняет число пробелами слева
    word_string = ''  <-- Инициализирует строку для описания
    for mag, name in _magnitude_list:
        if mag >= num_length:
            return word_string
        else:
            hundreds, tens, ones = num_string[-mag-3], \
                 num_string[-mag-2], num_string[-mag-1]
            if not (hundreds == tens == ones == '0'):
                word_string = _handle1to999(hundreds, tens, ones) + \
                                            name + word_string
def _handle1to999(hundreds, tens, ones):
    if hundreds == '0':
        return _handle1to99(tens, ones)
    else:
        return _1to9dict[hundreds] + ' hundred ' + _handle1to99(tens, ones)
def _handle1to99(tens, ones):
    if tens == '0':
        return _1to9dict[ones]
    elif tens == '1':
        return _10to19dict[ones]
    else:
        return _20to90dict[tens] + ' ' + _1to9dict[ones]
def test():  <-- Функция для режима тестирования модуля
    values = sys.stdin.read().split()
    for val in values:
        print("{0} = {1}".format(val, num2words(val)))
def main():
    parser = argparse.ArgumentParser(usage=__doc__)
    parser.add_argument("num", nargs='*')  <-- Собирает все значения в этом аргументе в список
    parser.add_argument("-t", "--test", dest="test",
                        action='store_true', default=False,
                        help="Test mode: reads from stdin")
    args = parser.parse_args()
    if args.test:  <-- Запускается в тестовом режиме, если установлена переменная test
        test()
    else:
        try:
            result = num2words(args.num[0])
        except KeyError:
            parser.error('argument contains non-digits')
        else:
            print("For {0}, say: {1}".format(args.num[0], result))
if __name__ == '__main__':
    main() ❶
else:
    print("n2w  loaded as a module")
```

Создает строку с числом

Перехватывает ошибки KeyError, появляющиеся из-за того, что аргументы содержат нецифровые данные

Если код запускается как сценарий, будет использовано имя `__main__`. Если он импортируется как модуль, ему будет присвоено имя `n2w` ❶.

Функция `main` демонстрирует предназначение управляющей функции в сценарии командной строки, которая, по сути, создает простой пользовательский интерфейс. Она может решать следующие задачи:

- Проверить правильность количества аргументов командной строки и правильность их типов; в случае нарушения уведомить пользователя и вывести информацию об использовании. В данном случае функция проверяет, что передан один аргумент, но не проверяет, что этот аргумент состоит только из цифр.
- Обрабатывать специальные режимы. В данном случае аргумент `'--test'` включает тестовый режим.
- Связывать аргументы командной строки с аргументами, необходимыми функциям, и вызывать их соответствующим образом. В нашем случае удаляются запятые и вызывается функция `num2words`.
- Перехватить и вывести более понятное сообщение для исключений, которые могут ожидаться. В данном случае перехватывается исключение `KeyErrors`, которое происходит при обнаружении в аргументах нецифровых символов[1].
- При необходимости привести вывод в более понятную для пользователя форму, что в данном примере происходит в команде `print`. Если бы это был сценарий для Windows, вероятно, лучше было бы разрешить пользователю открыть его двойным щелчком, то есть использовать `input` для запроса параметра, вместо того чтобы передавать его в командной строке, и оставить выходные данные на экране, завершив сценарий командой

```
input("Press the Enter key to exit")
```

- Но вы все равно можете оставить тестовый режим в качестве варианта командной строки.

Тестовый режим в следующем листинге обеспечивает поддержку регрессионного тестирования для модуля и его функции `num2words`. В данном случае он основан на размещении последовательности чисел в файле.

Листинг 11.11. Файл n2w.tst

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 98 99 100
101 102 900 901 999
999,999,999,999,999
1,000,000,000,000,000
```

Затем введите команду

```
python n2w.py --test < n2w.tst > n2w.txt
```

[1] Правильнее было бы явно проверить наличие нецифровых символов в аргументе с использованием модуля регулярных выражений, о котором будет рассказано ниже. Такое решение гарантирует, что приложение не скроет ошибки `KeyError`, возникающие по другим причинам.

Выходной файл легко проверяется на правильность. Этот пример был запущен несколько раз в процессе создания, и вы можете снова запустить его после любого изменения функции `num2words` или любой из вызываемых ею функций. Да, я отлично понимаю, что о полном исчерпывающем тестировании здесь речи не идет — более 999 триллионов корректных вариантов входных данных для этой программы остались без проверки!

Нередко поддержка тестового режима для модуля является единственной функцией сценария. Я знаю по крайней мере одну компанию, в которой политика разработки требует всегда создавать его для каждого разрабатываемого модуля Python. Встроенные типы объектов данных и методы Python обычно упрощают этот процесс, и разработчики, практикующие этот прием на практике, единогласно сходятся на том, что время и усилия не пропадут даром. За дополнительной информацией о тестировании кода Python обращайтесь к главе 21.

Другой способ заключается в создании отдельного файла, содержащего только часть функции `main`, которая обрабатывает аргумент, и импортировании `n2w` в этот файл. Только тогда тестовый режим останется в функции `main` файла `n2w.py`.

БЫСТРАЯ ПРОВЕРКА: ПРОГРАММЫ И МОДУЛИ

Какую проблему должно предотвратить использование конструкции `if __name__ == "__main__":` и как это делается? Можете ли вы предложить другой способ предотвращения этой проблемы?

11.6. Распространение приложений Python

Сценарии и приложения Python могут распространяться несколькими способами. Конечно, можно передавать пользователю исходные файлы (скорее всего, упакованные в файл .zip или .tar). Если приложения были написаны в портируемом виде, также можно распространять только байт-код в файла .pyc. Тем не менее оба этих способа оставляют желать лучшего.

11.6.1. Wheel-пакеты

В настоящее время стандартным способом упаковки и распространения модулей и приложений Python считается использование пакетов, называемых wheel-пакетами. Wheel-пакеты спроектированы с таким расчетом, чтобы сделать установку кода Python более надежной и упростить управление зависимостями. Подробное описание создания wheel-пакетов выходит за рамки этой главы; полную информацию о требованиях и описание процесса создания можно найти в руководстве «Python Packaging User Guide» по адресу https://packaging.python.org.

11.6.2. zipapp и pex

Если у вас имеется приложение, разбитое на несколько модулей, вы можете распространять его в формате исполняемого zip-файла. Использование этого формата зависит от двух фактов, относящихся к Python.

Во-первых, если zip-файл содержит файл с именем __main__.py, Python может использовать этот файл как точку входа в архив и выполнить файл __main__.py напрямую. Кроме того, содержимое zip-файла добавляется в `sys.path`, чтобы оно было доступно для импортирования и выполнения файлом __main__.py.

Во-вторых, zip-файлы позволяют добавить произвольный контент в начало архива. Если добавить строку `#!` с путем к интерпретатору Python (например, `#!/usr/bin/env python3`) и предоставить файлу необходимые разрешения, он превращается в автономный исполняемый файл.

На самом деле создать исполняемое zip-приложение вручную не так уж сложно. Создайте zip-архив, содержащий файл __main__.py, добавьте строку `#!` в начало и установите разрешения.

Начиная с Python 3.5, в стандартную библиотеку включается модуль zipapp; он создает zip-приложения либо из командной строки, либо с использованием API библиотеки.

Более мощный инструмент — `pex` — не входит в стандартную библиотеку, но доступен в каталоге пакетов pip. pex решает ту же базовую задачу, но предоставляет куда больше возможностей, при необходимости он доступен для Python 2.7. В любом случае zip-приложения удобны для упаковки и распространения многофайловых приложений Python, готовых к выполнению.

11.6.3. py2exe и py2app

Хотя в этой книге я стараюсь воздерживаться от подробного описания платформенно-зависимых инструментов, стоит упомянуть, что программа `py2exe` создает автономные Windows-программы, а `py2app` делает то же самое на платформе macOS. Под *автономностью* я имею в виду то, что приложение представляет собой один исполняемый файл, выполняемый на машинах, на которых Python не установлен. Автономные исполняемые файлы не идеальны во многих отношениях, потому что они обычно имеют больший размер и уступают по гибкости «родным» приложениям Python. Тем не менее в некоторых ситуациях они оказываются лучшим, а иногда и единственным решением.

11.6.4. Создание исполняемых программ с freeze

Также для создания программ Python, работающих на машинах без установки Python, можно воспользоваться программой `freeze`. Инструкции приведены в файле Readme в каталоге freeze из подкаталога Tools исходного каталога Python. Если

вы собираетесь использовать `freeze`, вероятно, для этого вам придется загрузить исходный дистрибутив Python.

В процессе создания программы создаются файлы на языке C, которые затем компилируются и компонуются компилятором C. Для этого компилятор должен быть установлен в вашей системе. Автономное приложение будет работать только на платформе, для которой у используемого компилятора C существуют исполняемые файлы.

Также есть еще несколько программ, пытающихся тем или иным способом преобразовать и упаковать приложение вместе с интерпретатором / исполнительной средой Python в одно автономное приложение. Однако в общем случае этот путь остается сложным и запутанным; лучше избегать его, если только у вас нет веских причин, времени и необходимых ресурсов для того, чтобы заставить его работать.

ПРАКТИЧЕСКАЯ РАБОТА 11: СОЗДАНИЕ ПРОГРАММЫ

В главе 8 вы создали версию утилиты UNIX `wc` для подсчета строк, слов и символов в файле. Теперь, когда в вашем распоряжении появилось больше инструментов, переработайте эту программу и добейтесь того, чтобы она стала ближе к оригиналу. В частности, программа должна поддерживать ключи для вывода количества только строк (`-l`), только слов (`-w`) и только символов (`-c`). Если ни один из этих ключей не задан, выводятся все три счетчика. Но если присутствует хотя бы один из ключей, то выводятся только заданные счетчики.

Чтобы немного усложнить задачу, просмотрите `man`-страницу `wc` для системы Linux/UNOX и добавьте ключ `-L`, чтобы выводить наибольшую длину строки. Попробуйте полностью реализовать поведение, описанное в man-странице, и сравните его с поведением утилиты `wc` в вашей системе.

Итоги

- Сценарии и модули Python в базовом виде представляют собой последовательности команд Python, сохраненные в файле.
- Модули могут быть организованы так, чтобы они могли запускаться как сценарии, а сценарии могут быть настроены так, чтобы их можно было импортировать как модули.
- Сценарии могут исполняться в UNIX, macOS или командной строке Windows. Их можно настроить для поддержки перенаправления ввода и вывода, а модуль `argparse` позволяет разбирать сложные комбинации аргументов командной строки.

- В macOS для запуска программ Python можно использовать Python Launcher — либо для отдельных файлов, либо в качестве приложения по умолчанию для открытия файлов Python.
- В системе Windows сценарии можно запускать несколькими способами: открывая их двойным щелчком, из окна Запуск программы или из окна командной строки.
- Сценарии Python могут распространяться в форме сценариев, в форме байт-кода или в виде специальных wheel-пакетов.
- Программы `py2exe`, `py2app` и `freeze` создают исполняемые программы Python, которые могут работать на машинах без интерпретатора Python.
- Итак, теперь вы представляете способы создания сценариев и приложений. В следующей главе будут рассмотрены средства Python для работы с файловыми системами.

12 Работа с файловой системой

Эта глава охватывает следующие темы:

- ✓ Управление путями и именами путей
- ✓ Получение информации о файлах
- ✓ Выполнение операций с файловой системой
- ✓ Обработка всех файлов в поддереве каталога

Работа с файлами состоит из двух аспектов: базовый ввод/вывод (он рассматривается в главе 13 «Чтение и запись файлов») и работа с файловой системой (например, переименование, создание, перемещение или обращение к файлами). И это может создать проблемы, потому что в разных операционных системах используются разные правила работы с файловой системой.

Вы можете освоить базовые операции ввода/вывода с файлами без изучения всех возможностей, предоставляемых Python для упрощения кроссплатформенного взаимодействия с файловыми системами, однако я так поступать не рекомендую. Ппрочитайте хотя бы первую часть главы, в которой описаны все средства, необходимые для обращения к файлам способом, не зависящим от вашей конкретной операционной системы. Таким образом вы можете открывать нужные файлы при использовании базовых операций ввода/вывода.

12.1. os и os.path против pathlib

Традиционный механизм операций с путями и файловыми системами в Python основан на использовании функций, включенных в модули `os` и `os.path`. Эти функции работали достаточно хорошо, но код часто получался более объемным, чем реально необходимо. В Python 3.5 была добавлена новая библиотека `pathlib`; она предоставляет более объектно-ориентированный и унифицированный механизм выполнения тех же операций. Так как в значительной части существующего кода

все еще используется старый стиль, я сохранила эти примеры и их объяснения. С другой стороны, у `pathlib` хорошие перспективы и с большой вероятностью она станет новым стандартом, поэтому после каждого примера старого метода я привожу пример (и краткое объяснение, где это необходимо) того, как то же самое делается с `pathlib`.

12.2. Пути и имена

Во всех операционных системах для обращения к файлам и каталогам используются строки с именами файлов или каталогов. Строки, используемые таким образом, обычно называются *путями* (paths) или *полными именами файлов*. Я буду использовать этот термин. Тот факт, что полные имена представляют собой строки, создает немало потенциальных трудностей при работе с ними. Python стремится предоставить разработчику функции, которые помогают избежать этих трудностей, но для эффективного использования этих функций необходимо понимать суть возникающих проблем. Все эти нюансы рассматриваются в этом разделе.

Семантика полных имен в разных операционных системах похожа, потому что файловые системы практически во всех операционных системах строятся на базе иерархической модели: диск является корнем дерева, а каталоги, подкаталоги и т. д. — ветвями. Таким образом, в большинстве операционных систем обращение к конкретному файлу происходит по одному принципу: по полному имени, которое определяет путь от корня дерева файловой системы (диска) до файла. (Сопоставление корня с жестким диском — чрезмерное упрощение, но оно достаточно близко к истине для целей этой главы.) Полное имя состоит из последовательности каталогов, которые необходимо пройти для того, чтобы добраться до нужного файла.

В разных операционных системах действуют разные соглашения относительно точного синтаксиса полных имен. Например, в путях Linux/UNIX для разделения имен файлов и каталогов используется символ `/`, тогда как в Windows для этой цели используется символ `\`. Кроме того, файловая система UNIX имеет единственный корень (который обозначается включением символа `/` в первую позицию полного имени), а в файловой системе Windows для каждого диска используется свой корень `A:\`, `B:\`, `C:\` и т. д. (`C:` обычно является основным диском). Из-за этих различий полное имя файла по-разному выглядит в разных операционных системах. Файл с именем `C:\data\myfile` в MS Windows может называться `/data/myfile` в UNIX или Mac OS. Python предоставляет функции и константы для выполнения стандартных операций с полными именами, с которыми можно не беспокоиться о таких синтаксических подробностях. При небольших усилиях вы сможете писать свои программы Python так, чтобы они правильно работали независимо от текущей файловой системы.

12.2.1. Абсолютные и относительные полные имена

В этих операционных системах возможны полные имена двух типов:

- *Абсолютные* полные имена задают точное местонахождение файла в файловой системе без каких-либо неоднозначностей, для этого они указывают полный путь к файлу, начиная от корня файловой системы.
- *Относительные* полные имена задают позицию файла относительно другой точки файловой системы, которая не указывается в самом относительном имени; вместо этого абсолютная начальная точка для таких имен определяется контекстом их использования.

В качестве примера приведу два абсолютных полных имени Windows:

```
C:\Program Files\Doom
D:\backup\June
```

и два абсолютных полных имени Linux с абсолютным именем Mac:

```
/bin/Doom
/floppy/backup/June
/Applications/Utilities
```

Здесь вы видите два относительных полных имени Windows:

```
mydata\project1\readme.txt
games\tetris
```

и относительные имена Linux/UNIX/Mac:

```
mydata/project1/readme.txt
games/tetris
Utilities/Java
```

Относительным именам необходим контекст для привязки к начальной точке. Этот контекст обычно предоставляется одним из двух способов.

Проще всего присоединить относительный путь к существующему абсолютному пути, получая таким образом новый абсолютный путь. Допустим, у вас имеется относительный путь Windows `Start Menu\Programs\Startup` и абсолютный путь `C:\Users\Administrator`. Объединяя эти два пути, вы получаете новый абсолютный путь `C:\Users\Administrator\Start Menu\Programs\Startup`, определяющий конкретную точку файловой системы. Присоединив тот же относительный путь к другому абсолютному пути (скажем, `C:\Users\myuser`), вы получите путь для папки `Startup` в каталоге `Profiles` другого пользователя (myuser).

Второй способ получения контекста в относительных путях основан на неявной ссылке на *текущий рабочий каталог* — тот каталог, в котором программа Python находится в любой точке своего выполнения. Команды Python могут неявно использовать текущий рабочий каталог при получении относительного пути в аргументе.

Например, при использовании команды `os.listdir(path)` с аргументом, который представляет собой относительный путь, точкой привязки относительного пути является текущий рабочий каталог, а результатом выполнения команды будет список имен файлов в каталоге, путь к которому образуется присоединением к текущему рабочему каталогу аргумента с относительным путем.

12.2.2. Текущий рабочий каталог

Когда вы редактируете документ на своем компьютере, вы знаете, в какой точке файловой структуры компьютера вы находитесь — в одном каталоге (папке) с файлом, с которым вы работаете. Аналогичным образом во время своей работы Python знает свое текущее положение в структуре каталогов в любой момент времени. Это важный момент, потому что программа, например, может запросить список файлов, находящихся в текущем каталоге. Каталог, с которым работает программа Python, называется *текущим рабочим каталогом* для этой программы. Этот каталог может отличаться от того каталога, в котором находится сама программа.

Запустите Python и используйте команду `os.getcwd` (получение текущего рабочего каталога) для нахождения исходного текущего рабочего каталога Python:

```
>>> import os
>>> os.getcwd()
```

Обратите внимание: `os.getcwd` используется как вызов функции без аргументов; тем самым подчеркивается тот факт, что возвращаемое значение не является константой, но изменяется при вводе команд, изменяющих текущий рабочий каталог. (Вероятно, вы получите каталог, в котором находится сама программа Python, или же каталог, в котором вы находились в момент запуска Python. На машине Linux я получаю результат /home/myuser — мой домашний каталог.) На машине с системой Windows в путь вставляются лишние символы \, потому что в Windows символ \ используется в качестве разделителя компонентов пути, а в строках Python (раздел 6.3.1) \ имеет специальный смысл, если он не будет экранирован другим таким же символом.

Теперь введите команду:

```
>>> os.listdir(os.curdir)
```

Константа `os.curdir` возвращает строку, которая используется вашей системой для обозначения текущего каталога. В UNIX и Windows текущий каталог представляется одной точкой, но для того, чтобы код были портируемым, всегда используйте `os.curdir` вместо точки. Эта строка содержит относительный путь, а следовательно, `os.listdir` присоединит ее к пути текущего рабочего каталога и вы получите тот же путь. Следующая команда возвращает список всех файлов или папок в текущем рабочем каталоге. Выберите имя папки и введите:

```
>>> os.chdir(имя_папки)  <-- Функция смены каталога
>>> os.getcwd()
```

Как видите, Python переходит в папку, заданную аргументом функции `os.chdir`. Другой вызов `os.listdir(os.curdir)` вернет список файлов в заданной папке, потому что `os.curdir` будет выполняться относительно нового текущего рабочего каталога. Многие операции с файловыми системами в Python используют текущий рабочий каталог подобным образом.

12.2.3. Обращение к каталогам с использованием pathlib

Чтобы получить текущий каталог с помощью `pathlib`, можно поступить так:

```
>>> import pathlib
>>> cur_path = pathlib.Path()
>>> cur_path.cwd()
PosixPath('/home/naomi')
```

`pathlib` не может изменить текущий каталог так, как это делает `os.chdir()` (см. предыдущий раздел), но для работы с новым каталогом можно создать новый объект пути, как показано в разделе 12.2.5 «Работа с полными именами в pathlib».

12.2.4. Операции с полными именами

Теперь, когда у вас имеется необходимая информация для понимания полных имен файлов и каталогов, рассмотрим инструменты для работы с этими полными именами в Python. К их числу относятся функции и константы субмодуля `os.path`, которые могут использоваться для манипуляции с путями без явного использования синтаксиса, присущего конкретной операционной системе. Пути по-прежнему представляются в виде строк, но вам никогда не следует рассматривать или работать с ними на этом уровне.

Для начала создадим несколько полных имен для разных операционных систем с использованием функции `os.path.join`. Обратите внимание: при импортировании `os` также подключается субмодуль `os.path`; добавлять команду явного импортирования `os.path` не нужно.

Сначала запустите Python в Windows:

```
>>> import os
>>> print(os.path.join('bin', 'utils', 'disktools'))
bin\utils\disktools
```

Функция `os.path.join` интерпретирует аргументы как серию имен каталогов или файлов, которые объединяются для формирования строки, воспринимаемой текущей операционной системой как относительный путь. В системе Windows это означает, что имена компонентов пути должны объединяться символами \; отсюда и результат.

Теперь попробуем сделать то же самое в UNIX:

```
>>> import os
>>> print(os.path.join('bin', 'utils', 'disktools'))
bin/utils/disktools
```

В результате будет получен тот же путь, но с использованием разделителей /, принятых в Linux/UNIX (в отличие от разделителей \ системы Windows). Иначе говоря, `os.path.join` позволяет формировать пути из последовательности каталогов или файлов, не беспокоясь о соглашениях текущей операционной системы. Функция `os.path.join` является основным способом построения путей без привязки к конкретной операционной системе, в которой будут выполняться ваши программы.

Аргументы `os.path.join` не обязаны представлять один каталог или имя файла; это также могут быть целые ветви, которые объединяются для построения более длинного имени. Следующий пример демонстрирует данную возможность в среде Windows; обратите внимание на удвоение символа обратного слеша в строке. Также при вводе полного имени можно было указать символы /, потому что Python автоматически преобразует их перед вызовом служебной функции Windows:

```
>>> import os
>>> print(os.path.join('mydir\\bin', 'utils\\disktools\\chkdisk'))
mydir\bin\utils\disktools\chkdisk
```

Конечно, если вы всегда используете `os.path.join` для построения путей, вам не придется беспокоиться об этой ситуации. Чтобы записать этот пример в портируемой форме, введите следующие команды:

```
>>> path1 = os.path.join('mydir', 'bin');
>>> path2 = os.path.join('utils', 'disktools', 'chkdisk')
>>> print(os.path.join(path1, path2))
mydir\bin\utils\disktools\chkdisk
```

В команду `os.path.join` также заложены некоторые сведения об абсолютных и относительных путях. В Linux/UNIX абсолютный путь всегда начинается с / (потому что один символ / обозначает каталог верхнего уровня всей системы, который содержит все остальное, включая различные накопители — CD-дисковод, флоппи-дисковод и т. д.). Относительный путь в UNIX представляет собой любой действительный путь, который *не* начинается с символа /. В любой из операционных систем семейства Windows ситуация усложняется, потому что в Windows правила относительных и абсолютных путей не столь очевидны. Я не стану погружаться в технические подробности, а просто скажу, что в такой ситуации проще всего воспользоваться следующими правилами.

- Полное имя, начинающееся с буквы диска, двоеточия и символа \, за которыми следует путь, представляет собой абсолютный путь: C:\Program Files\Doom. (Учтите, что символы `C:` без завершающего символа \ не могут считаться надежным обозначением каталога верхнего уровня на диске `C:` — для этой цели следует использовать запись `C:\`. Это требование обусловлено соглашениями DOS, а не архитектурой Python.)
- Полный путь, не начинающийся с буквы диска или с символа \, является относительным: mydirectory\letters\business.

- Полный путь, начинающийся с символов \\, за которыми следует имя сервера, является путем к сетевому ресурсу.
- Все остальные комбинации считаются недействительными полными именами[1].

Независимо от используемой операционной системы, команда `os.path.join` не проверяет создаваемые имена. Вы можете построить полное имя с символами, запрещенными в полных именах по правилам вашей ОС. Если такая проверка необходима, вероятно, лучше всего будет написать маленькую проверочную функцию самостоятельно.

Команда `os.path.split` возвращает кортеж из двух элементов, отделяющий базовое имя (имя файла или каталога в конце пути) от остальной части. В системе Windows это может выглядеть так:

```
>>> import os
>>> print(os.path.split(os.path.join('some', 'directory', 'path')))
('some\\directory', 'path')
```

Функция `os.path.basename` возвращает только базовое имя из пути, а функция `os.path.dirname` возвращает часть пути до последнего имени, не включая его, как в следующем примере:

```
>>> import os
>>> os.path.basename(os.path.join('some', 'directory', 'path.jpg'))
'path.jpg'
>>> os.path.dirname(os.path.join('some', 'directory', 'path.jpg'))
'some\\directory'
```

Для работы с расширениями через точку, используемыми в большинстве файловых систем для обозначения типа файлов (заметным исключением является Macintosh), Python предоставляет функцию `os.path.splitext`:

```
>>> os.path.splitext(os.path.join('some', 'directory', 'path.jpg'))
('some/directory/path', '.jpg')
```

Последний элемент возвращаемого кортежа содержит расширение указанного файла через точку (если оно было). Первый элемент возвращаемого кортежа содержит все символы исходного аргумента, за исключением расширения.

Также для работы с полными именами можно использовать специализированные функции. `os.path.commonprefix(путь1, путь2, ...)` находит общий префикс для набора путей (если он есть). Эта функция особенно полезна, если вы хотите найти каталог самого нижнего уровня, содержащий все файлы из заданного набора. `os.path.expanduser` расширяет сокращенные обозначения пользователя в путях (например, в UNIX). Функция `os.path.expandvars` делает то же самое с переменными среды. Пример для системы Windows 10:

[1] На самом деле Microsoft Windows допускает и другие конструкции, но лучше придерживаться этих определений.

```
>>> import os
>>> os.path.expandvars('$HOME\\temp')
'C:\\Users\\administrator\\personal\\temp'
```

12.2.5. Работа с полными именами pathlib

Как и в предыдущем разделе, начнем с построения нескольких полных имен для разных операционных систем с использованием методов объекта `path`.

Сначала запустите Python в Windows:

```
>>> from pathlib import Path
>>> cur_path = Path()
>>> print(cur_path.joinpath('bin', 'utils', 'disktools'))
bin\utils\disktools
```

Того же результата можно добиться при помощи оператора `/`:

```
>>> cur_path / 'bin' / 'utils' / 'disktools'
WindowsPath('bin/utils/disktools')
```

Обратите внимание: в представлении объекта `path` всегда используются символы `/`, а в объектах путей Windows символы `/` автоматически преобразуются в `\`, как того требует ОС. Посмотрим, удастся ли сделать то же самое в UNIX:

```
>>> cur_path = Path()
>>> print(cur_path.joinpath('bin', 'utils', 'disktools'))
bin/utils/disktools
```

Свойство `parts` возвращает кортеж со всеми компонентами пути. Следующий пример работает в системе Windows:

```
>>> a_path = WindowsPath('bin/utils/disktools')
>>> print(a_path.parts)
('bin', 'utils', 'disktools')
```

Свойство `name` возвращает только базовое имя, свойство `parent` возвращает путь до последнего имени (не включая его), а свойство `suffix` возвращает расширение, используемое в большинстве файловых систем для обозначения типа файла (заметным исключением является Macintosh). Пример:

```
>>> a_path = Path('some', 'directory', 'path.jpg')
>>> a_path.name
'path.jpg'
>>> print(a_path.parent)
some\directory
>>> a_path.suffix
'.jpg'
```

Другие методы, связанные с объектами `Path`, позволяют гибко работать как с именами, так и с самими файлами; обращайтесь к документации модуля `pathlib`.

Вполне возможно, что модуль `pathlib` упростит вашу жизнь и сделает код работы с файлами более компактным.

12.2.6. Полезные константы и функции

В вашем распоряжении несколько констант и функций для работы с путями, с которыми ваш код Python станет более системно-независимым, чем он мог бы быть. Простейшие из этих констант — `os.curdir` и `os.pardir` — определяют соответственно символы, используемые операционной системой для обозначения текущего и родительского каталогов. В Windows, а также в Linux/UNIX и macOS это индикаторы `.` и `..` соответственно; они могут использоваться в качестве обычных элементов путей. Следующий пример

```
os.path.isabs(os.path.join(os.pardir, path))
```

проверяет, является ли каталог родителем для заданного пути. Константа `os.curdir` особенно полезна для выполнения команд с текущим рабочим каталогом. Например, вызов

```
os.listdir(os.curdir)
```

возвращает список имен файлов в текущем рабочем каталоге (потому что `os.curdir` является относительным путем, а `os.listdir` всегда считает, что относительные пути задаются относительно текущего рабочего каталога).

Константа `os.name` возвращает имя модуля Python, импортированного для обработки специфических подробностей операционной системы. Пример в моей системе Windows XP:

```
>>> import os
>>> os.name
'nt'
```

Обратите внимание: `os.name` возвращает `'nt'` даже в Windows 10. Большинство версий Windows, кроме Windows CE, идентифицируются как `'nt'`.

На компьютере Mac с OS X и Linux/UNIX выдается ответ `posix`. Вы можете использовать этот ответ для выполнения специальных операций в зависимости от платформы, на которой вы работаете:

```
import os
if os.name == 'posix':
    root_dir = "/"
elif os.name == 'nt':
    root_dir = "C:\\"
else:
    print("Don't understand this operating system!")
```

В программах также иногда используется константа `sys.platform`, которая выдает более точную информацию. В Windows 10 `sys.platform` присваивается значение `win32`, даже если на машине работает 64-разрядная версия операционной системы.

В Linux иногда встречается значение `linux2`, тогда как в Solaris может быть присвоено значение `sunos5` в зависимости от используемой версии.

Все переменные среды и связанные с ними значения доступны в словаре с именем `os.environ`. Во многих операционных системах этот словарь содержит переменные, относящиеся к путям, — как правило, это пути поиска для двоичных файлов и т. д. Если для вашей работы необходима эта информация, вы знаете, где ее можно найти.

К настоящему моменту вы познакомились с основными аспектами работы с полными именами в Python. Если сейчас вас интересует, как открыть файл для чтения или записи, переходите сразу к следующей главе. А в оставшейся части этой главы приводится дополнительная информация о полных именах, тестировании, полезных константах и т. д.

БЫСТРАЯ ПРОВЕРКА: ОПЕРАЦИИ С ПУТЯМИ

Как бы вы использовали функции модуля `os`, чтобы получить путь к файлу с именем test.log, создать новый путь в том же каталоге для файла с именем test.log.old? А как бы вы сделали то же самое с модулем `pathlib`?

Какой путь вы получите при создании объекта pathlib `Path` на базе `os.pardir`? Попробуйте и узнайте.

12.3. Получение информации о файлах

Пути к файлам должны задавать местонахождение файлов и каталогов на вашем жестком диске. Скорее всего, вам понадобится передавать пути при вызовах, потому что вы хотите получить некую информацию об объекте, которому он соответствует. Для этой цели существуют различные функции Python.

Наиболее часто используемые функции Python для получения информации о путях — `os.path.exists`, `os.path.isfile` и `os.path.isdir`. Все эти функции получают аргумент с путем. Функция `os.path.exists` возвращает `True`, если ее аргумент является путем, представляющим путь к существующему объекту файловой системы. `os.path.isfile` возвращает `True` в том (и только в том) случае, если полученный путь обозначает нормальный файл данных (исполняемые файлы относятся к этой категории). В противном случае возвращается `False` — это означает, что аргумент path может не представлять реально существующий объект в файловой системе. `os.path.isdir` возвращает `True` в том и только в том случае, если его аргумент path обозначает каталог; в противном случае возвращается `False`. Следующие примеры нормально работают в моей системе. Возможно, в вашей системе вам придется использовать другие пути, чтобы изучить поведение этих функций:

```
>>> import os
>>> os.path.exists('C:\\Users\\myuser\\My Documents')
True
>>> os.path.exists('C:\\Users\\myuser\\My Documents\\Letter.doc')
```

```
True
>>> os.path.exists('C:\\Users\\myuser\\\My Documents\\ljsljkflkjs')
False
>>> os.path.isdir('C:\\Users\\myuser\\My Documents')
True
>>> os.path.isfile('C:\\Users\\ myuser\\My Documents')
False
>>> os.path.isdir('C:\\Users\\ myuser\\My Documents
\\Letter.doc')
False
>>> os.path.isfile('C:\\Users\\ myuser\\My Documents\\Letter.doc')
True
```

Некоторые похожие функции предоставляют средства для получения более специализированной информации. `os.path.islink` и `os.path.ismount` полезны в контексте Linux и других операционных систем UNIX, предоставляющих файловые ссылки и точки монтирования; они возвращают `True`, если путь представляет файл, являющий ссылкой или точкой монтирования соответственно. `os.path.islink` *не* возвращает `True` для файлов ярлыков Windows (файлы с расширением .lnk) по одной простой причине: эти файлы не являются полноценными ссылками. Однако `os.path.islink` возвращает `True` в системах Windows для настоящих символических ссылок, созданных командой `mklink()`. ОС не назначает им специальный статус, а программы не могут прозрачно использовать их так, как если бы они были настоящими файлами. `os.path.samefile(path1, path2)` возвращает `True` в том и только в том случае, если два аргумента указывают на один и тот же файл. `os.path.isabs(path)` возвращает `True` для абсолютных путей; в противном случае возвращается `False`. `os.path.getsize(path)`, `os.path.getmtime(path)` и `os.path.getatime(path)` возвращают размер, время последнего изменения и время последнего обращения соответственно.

12.3.1. Получение информации о файлах функцией scandir

Кроме перечисленных функций `os.path`, для получения более полной информации о файлах в каталоге также можно воспользоваться функцией `os.scandir`, которая возвращает итератор для объектов `os.DirEntry`. Объекты `os.DirEntry` открывают доступ к атрибутам элемента каталога, так что использование `os.scandir` может быть быстрее и эффективнее объединения `os.listdir` (см. следующий раздел) с операциями `os.path`. Если, например, вам понадобится узнать, соответствует ли элемент каталога файлу или каталогу, возможность `os.scandir` получения более подробной информации, нежели простое имя, может оказаться полезной. У объектов `os.DirEntry` имеются методы, соответствующие функциям `os.path`, упомянутым в предыдущем разделе, включая `exists`, `is_dir`, `is_file`, `is_socket` и `is_symlink`.

`os.scandir` также поддерживает менеджеры контекста с `with`; рекомендуется использовать их для того, чтобы обеспечить правильность освобождения ресурсов. Следующий фрагмент кода перебирает все элементы каталога и выводит имя каждого элемента и признак того, является ли он файлом:

```
>>> with os.scandir(".") as my_dir:
...     for entry in my_dir:
...         print(entry.name, entry.is_file())
...
pip-selfcheck.json True
pyvenv.cfg True
include False
test.py True
lib False
lib64 False
bin False
```

12.4. Другие операции с файловой системой

Кроме получения информации о файлах, Python позволяет выполнять некоторые операции с файловой системой напрямую через набор базовых, но в высшей степени полезных команд модуля `os`.

В этом разделе описаны только полноценные кроссплатформенные операции. Во многих операционных системах доступны более сложные функции файловой системы; за подробностями обращайтесь к документации библиотеки Python.

Как вы уже видели, для получения списка файлов в каталоге используется функция `os.listdir`:

```
>>> os.chdir(os.path.join('C:', 'my documents', 'tmp'))
>>> os.listdir(os.curdir)
['book1.doc.tmp', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp']
```

Обратите внимание: в отличие от команды вывода содержимого каталога во многих других языках и оболочках, Python *не* включает индикаторы `os.curdir` и `os.pardir` в списке, возвращаемом `os.listdir`.

Функция `glob` из модуля `glob` (названная по имени старой функции UNIX, выполнявшей поиск по шаблону) расширяет метасимволы в стиле оболочки Linux/UNIX и последовательности символов в именах, возвращая подходящие файлы в текущем рабочем каталоге. `*` представляет любую последовательность символов, `?` представляет один отдельный символ. Последовательность символов (`[h,H]` или `[0-9]`) представляет один отдельный символ в этой последовательности:

```
>>> import glob
>>> glob.glob("*")
['book1.doc.tmp', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp']
>>> glob.glob("*bkp")
['registry.bkp']
>>> glob.glob("?.tmp")
['a.tmp', '1.tmp', '7.tmp', '9.tmp']
>>> glob.glob("[0-9].tmp")
['1.tmp', '7.tmp', '9.tmp']
```

Для переименования (перемещения) файла или каталога используется функция `os.rename`:

```
>>> os.rename('registry.bkp', 'registry.bkp.old')
>>> os.listdir(os.curdir)
['book1.doc.tmp', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
```

Команда может использоваться для перемещения файлов как между каталогами, так и внутри каталогов.

Удаление файлов данных осуществляется функцией `os.remove`:

```
>>> os.remove('book1.doc.tmp')
>>> os.listdir(os.curdir)
['a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
```

Функция `os.remove` не может использоваться для удаления каталогов. Это ограничение введено по соображениям безопасности, чтобы предотвратить случайное удаление целых подструктур каталогов.

Файлы могут создаваться при записи, как обсуждалось в главе 11. Для создания каталогов используется функция `os.makedirs` или `os.mkdir`. Различия между ними заключаются в том, что `os.makedirs` создает промежуточные каталоги, а `os.mkdir` этого не делает:

```
>>> os.makedirs('mydir')
>>> os.listdir(os.curdir)
['mydir', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
>>> os.path.isdir('mydir')
True
```

Для удаления каталогов используется функция `os.rmdir`. Эта функция удаляет только пустые каталоги. При попытке вызвать ее для непустого каталога происходит исключение:

```
>>> os.rmdir('mydir')
>>> os.listdir(os.curdir)
['a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
```

Для удаления непустых каталогов используется функция `shutil.rmtree`, которая рекурсивно удаляет все файлы в дереве каталогов. За подробностями обращайтесь к документации стандартной библиотеки Python.

12.4.1. Другие операции с файловой системой с использованием pathlib

Объекты путей поддерживают большинство методов, упоминавшихся ранее. Тем не менее существуют и различия. Метод `iterdir` аналогичен функции `os.path.listdir`, за исключением того, что он возвращает итератор вместо списка строк:

```
>>> new_path = cur_path.joinpath('C:', 'my documents', 'tmp'))
>>> list(new_path.iterdir())
[WindowsPath('book1.doc.tmp'), WindowsPath('a.tmp'), WindowsPath('1.tmp'),
     WindowsPath('7.tmp'), WindowsPath('9.tmp'), WindowsPath('registry.bkp')]
```

Учтите, что в среде Windows возвращаются объекты `WindowsPath`, а в Mac OS и Linux вместо них используются объекты `PosixPath`.

У объектов путей `pathlib` имеется встроенный метод `glob`, который возвращает не список строк, а итератор для объектов путей. В остальном эта функция ведет себя так же, как функция `glob.glob`, продемонстрированная выше:

```
>>> list(cur_path.glob("*"))
[WindowsPath('book1.doc.tmp'), WindowsPath('a.tmp'), WindowsPath('1.tmp'),
     WindowsPath('7.tmp'), WindowsPath('9.tmp'), WindowsPath('registry.bkp')]
>>> list(cur_path.glob("*bkp"))
[WindowsPath('registry.bkp')]
>>> list(cur_path.glob("?.tmp"))
[WindowsPath('a.tmp'), WindowsPath('1.tmp'), WindowsPath('7.tmp'),
     WindowsPath('9.tmp')]
>>> list(cur_path.glob("[0-9].tmp"))
[WindowsPath('1.tmp'), WindowsPath('7.tmp'), WindowsPath('9.tmp')]
```

Для переименования (перемещения) файла или каталога используется метод `rename` объекта пути:

```
>>> old_path = Path('registry.bkp')
>>> new_path = Path('registry.bkp.old')
>>> old_path.rename(new_path)
>>> list(cur_path.iterdir())
[WindowsPath('book1.doc.tmp'), WindowsPath('a.tmp'), WindowsPath('1.tmp'),
     WindowsPath('7.tmp'), WindowsPath('9.tmp'),
     WindowsPath('registry.bkp.old')]
```

Команда может использоваться для перемещения файлов как между каталогами, так и внутри каталогов.

Файлы данных удаляются методом `unlink`:

```
>>> new_path = Path('book1.doc.tmp')
>>> new_path.unlink()
>>> list(cur_path.iterdir())
[WindowsPath('a.tmp'), WindowsPath('1.tmp'), WindowsPath('7.tmp'),
     WindowsPath('9.tmp'), WindowsPath('registry.bkp.old')]
```

Как и в случае с `os.remove`, метод `unlink` не может использоваться для удаления каталогов. Это ограничение введено по соображениям безопасности, чтобы предотвратить случайное удаление целых подструктур каталогов.

Чтобы создать каталог с использованием объекта пути, используйте метод `mkdir` объекта пути. Если передать методу `mkdir` параметр `parents=True`, он создаст все необходимые промежуточные каталоги; в противном случае при отсутствии промежуточного каталога инициируется ошибка `FileNotFoundError`:

```
>>> new_path = Path ('mydir')
>>> new_path.mkdir(parents=True)
>>> list(cur_path.iterdir())
[WindowsPath('mydir'), WindowsPath('a.tmp'), WindowsPath('1.tmp'),
```

```
     WindowsPath('7.tmp'), WindowsPath('9.tmp'),
     WindowsPath('registry.bkp.old')]]
>>> new_path.is_dir('mydir')
True
```

Для удаления каталогов используется метод `rmdir`. Этот метод удаляет только пустые каталоги. При попытке вызвать его для непустого каталога происходит исключение:

```
>>> new_path = Path('mydir')
>>> new_path.rmdir()
>>> list(cur_path.iterdir())
[WindowsPath('a.tmp'), WindowsPath('1.tmp'), WindowsPath('7.tmp'),
     WindowsPath('9.tmp'), WindowsPath('registry.bkp.old']
```

ПРАКТИЧЕСКАЯ РАБОТА 12: ДРУГИЕ ОПЕРАЦИИ С ФАЙЛАМИ

Как бы вы вычислили общий размер всех файлов с расширением .txt, которые не являются символическими ссылками, в каталоге? Если в вашем первом ответе использовался модуль `os.path`, попробуйте сделать то же с `pathlib`, и наоборот.

Напишите код, который расширяет ваше предыдущее решение и перемещает те же файлы .txt в новый подкаталог с именем backup.

12.5. Обработка всех файлов в поддереве каталогов

Наконец, для рекурсивного перебора структур каталогов существует в высшей степени полезная функция `os.walk`. С ее помощью можно обойти все дерево каталогов, причем для каждого пройденного каталога возвращаются три вещи: корень (путь), список подкаталогов и список файлов.

При вызове `os.walk` передается путь к стартовому каталогу, а также три необязательных аргумента: `os.walk(directory, topdown=True, onerror=None, followlinks=False)`. Аргумент `directory` задает стартовый путь; если аргумент `topdown` равен `True` или не задан, файлы каждого каталога обрабатываются *до* его подкаталогов, в результате чего будет получен список, который начинается сверху и идет вниз. С другой стороны, если аргумент `topdown` равен `False`, *сначала* обрабатываются подкаталоги каждого подкаталога, и обход дерева осуществляется снизу вверх. Параметру `onerror` можно присвоить функцию обработки ошибок, возникающих при вызовах `os.listdir` (по умолчанию ошибки игнорируются). Функция `os.walk` по умолчанию не переходит в папки, которые являются символическими ссылками, если только не передать ей параметр `followlinks=True`.

При вызове `os.walk` создает итератор, который рекурсивно применяет себя ко всем каталогам, содержащимся в параметре `top`. Иначе говоря, для каждого подкаталога `subdir` функция `os.walk` рекурсивно вызывает сама себя в форме `os.walk(subdir, ...)`. Учтите, что если аргумент `topdown` равен `True` или не задан, то список каталогов может быть изменен (любыми операторами или методами, изменяющими список)

перед тем, как его элементы будут использованы на следующем уровне рекурсии; вы можете использовать его для управления тем, в какие подкаталоги будет заходить `os.walk` (и будет ли заходить вообще).

Чтобы получить представление о работе `os.walk`, я рекомендую обойти дерево каталогов и вывести значения, возвращаемые для каждого каталога. Для примера выведем список содержимого текущего рабочего каталога вместе с количеством элементов в каждом каталоге, исключая каталоги .git:

```
import os
for root, dirs, files in os.walk(os.curdir):
    print("{0} has {1} files".format(root, len(files)))
    if ".git" in dirs:  <-- Проверяет каталоги с именем .git
        dirs.remove(".git")  <-- Удаляет .git (только каталог) из списка каталогов
```

Пример достаточно сложен, и если вы хотите использовать возможности `os.walk` в полной мере, вам стоит поэкспериментировать с этой функцией и понять, что же происходит.

Функция `copytree` модуля `shutil` осуществляет рекурсивное копирование всех файлов в каталоге и во всех его подкаталогах, сохраняя разрешения доступа и статистику (то есть время обращения/изменения). Модуль `shutil` также содержит уже упоминавшуюся функцию `rmtree` для удаления каталога и всех его подкаталогов, а также несколько функций для копирования отдельных файлов. За подробностями обращайтесь к документации стандартной библиотеки.

Итоги

- Python предоставляет группу функций и констант для работы со ссылками файловой системы (полными именами) и операций файловой системы способом, не зависящим от текущей операционной системы.
- За информацией о более сложных и специализированных операциях файловой системы, обычно ассоциированных с конкретной операционной системой или системами, обращайтесь к документации Python по модулям `os`, `pathlib` и `posix`.
- В табл. 12.1 и 12.2 приведена сводка функций, описанных в этой главе.

Таблица 12.1. Сводка значений и функций файловой системы

Функция	Значение или операция
os.getcwd(), Path.cwd()	Получает текущий каталог
os.name	Предоставляет общую информацию о платформе
sys.platform	Предоставляет расширенную информацию о платформе
os.environ	Выдает информацию о переменных среды

Функция	Значение или операция
os.listdir(path)	Получает список файлов в каталог
os.scandir(path)	Получает итератор для объектов os.DirEntry в каталоге
os.chdir(path)	Изменяет каталог
os.path.join(elements), Path.joinpath(elements)	Объединяет элементы в путь
os.path.split(path)	Разбивает путь на базу и завершитель (последний элемент пути)
Path.parts	Кортеж элементов пути
os.path.splitext(path)	Разбивает путь на базу и расширение файла
Path.suffix	Расширение файла для пути
os.path.basename(path)	Получает базовое имя для пути
Path.name	Базовое имя для пути
os.path.commonprefix(list_of_paths)	Получает общий префикс для всех путей в списке
os.path.expanduser(path)	Расширяет ~ или ~user в полное имя
os.path.expandvars(path)	Расширяет переменные среды
os.path.exists(path)	Проверяет, существует ли путь
os.path.isdir(path), Path.is_dir()	Проверяет, является ли путь каталогом
os.path.isfile(path), Path.is_file()	Проверяет, является ли путь файлом
os.path.islink(path), Path.is_link()	Проверяет, является ли путь символической ссылкой (не ярлыком Windows!)
os.path.ismount(path)	Проверяет, является ли путь точкой монтирования
os.path.isabs(path), Path.is_absolute()	Проверяет, является ли путь абсолютным
os.path.samefile(path_1, path_2)	Проверяет, относятся ли два пути к одному файлу
os.path.getsize(path)	Получает размер файла
os.path.getmtime(path)	Получает время изменения
os.path.getatime(path)	Получает время обращения
os.rename(old_path, new_path)	Переименовывает файл
os.mkdir(path)	Создает каталог
os.makedirs(path)	Создает каталог и необходимые родительские каталоги
os.rmdir(path)	Удаляет каталог
glob.glob(pattern)	Получает совпадения для заданного шаблона
`os.walk(path)`	Получает все имена файлов в дереве каталогов

Таблица 12.2. Неполный список свойств и функций pathlib

Метод или свойство	Значение или операция
`Path.cwd()`	Получает текущий каталог
`Path.joinpath(elements) или Path / element / element`	Объединяет элементы в новый путь
`Path.parts`	Кортеж элементов пути
`Path.suffix`	Расширение файла для пути
`Path.name`	Базовое имя для пути
`Path.exists()`	Проверяет, существует ли путь
`Path.is_dir()`	Проверяет, является ли путь каталогом
`Path.is_file()`	Проверяет, является ли путь файлом
`Path.is_symlink()`	Проверяет, является ли путь символической ссылкой (не ярлыком Windows!)
`Path.is_absolute()`	Проверяет, является ли путь абсолютным
`Path.samefile(Path2)`	Проверяет, относятся ли два пути к одному файлу
`Path1.rename(Path2)`	Переименовывает файл
`Path.mkdir([parents=True])`	Создает каталог; если аргумент parents равен True, также создает необходимые родительские каталоги
`Path.rmdir()`	Удаляет каталог
`Path.glob(pattern)`	Получает совпадения для шаблона

13 Чтение и запись файлов

Эта глава охватывает следующие темы:

- ✓ Открытие файлов и объектов `file`
- ✓ Закрытие файлов
- ✓ Открытие файлов в разных режимах
- ✓ Чтение и запись текстовых или двоичных данных
- ✓ Перенаправление ввода/вывода экрана
- ✓ Использование модуля `struct`
- ✓ Травление объектов в файлы
- ✓ Стеллажные объекты

13.1. Открытие файлов и объектов file

Пожалуй, самая распространенная операция с файлами — открытие и чтение данных.

В Python для открытия и чтения файлов используется встроенная функция `open` и различные встроенные операции чтения. Следующая короткая программа Python читает одну строку из текстового файла с именем myfile:

```
with open('myfile', 'r') as file_object:
    line = file_object.readline()
```

Функция `open` ничего не читает из файла; вместо этого она возвращает объект, называемый объектом `file` (или файловым объектом), который используется для обращения к открытому файлу. Объект `file` хранит информацию о том, какая часть файла была прочитана или записана. Весь файловый ввод/вывод в Python осуществляется через объекты `file`, а не по именам файлов.

Первый вызов `readline` возвращает первую строку объекта `file` вплоть до первого символа новой строки (включительно) или весь файл, если в нем нет ни одного

символа новой строки; следующий вызов `readline` возвращает вторую строку, если она существует, и т. д.

Первый аргумент функции `open` содержит полный путь. В предыдущем примере мы открывали то, что должно было быть существующим файлом в текущем рабочем каталоге. Следующий фрагмент открывает файл в конкретном месте: c:\My Documents\test\myfile:

```
import os
file_name = os.path.join("c:", "My Documents", "test", "myfile")
file_object = open(file_name, 'r')
```

Также обратите внимание на использование ключевого слова `with`; это означает, что файл будет открыт с менеджером контекста (глава 14). Пока достаточно знать, что такой способ открытия файлов лучше управляет потенциальными ошибками ввода/вывода и в общем случае считается предпочтительным.

13.2. Закрытие файлов

После того как из объекта `file` будут прочитаны (или записаны) все данные, этот объект следует закрыть. Закрытие объекта `file` освобождает системные ресурсы, позволяет выполнять чтение или запись в файл из другого кода и в целом повышает надежность программы. Для небольших сценариев незакрытый объект `file` обычно не приводит к особым последствиям; объекты `file` автоматически закрываются при завершении сценария или программы. Для больших программ наличие большого количества открытых объектов `file` может привести к исчерпанию системных ресурсов и аварийному завершению программы.

Если объект `file` стал ненужным, следует закрыть его методом `close`. Приведенная выше короткая программа приходит к следующему виду:

```
file_object = open("myfile", 'r')
line = file_object.readline()
# . . . дальнейшее чтение из file_object . . .
file_object.close()
```

Использование менеджера контекста и ключевого слова `with` также позволяет автоматически закрывать файлы после завершения работы с ними:

```
with open("myfile", 'r') as file_object:
    line = file_object.readline()
    # . . . дальнейшее чтение из file_object . . .
```

13.3. Открытие файлов для записи или в других режимах

Второй аргумент команды `open` содержит строку, которая определяет режим открытия файла. Режим `'r'` означает «открыть файл для чтения», `'w'` — «открыть файл для записи» (все существующие данные стираются), `'a'` — «открыть файл

для присоединения» (новые данные будут присоединяться после данных, уже хранящихся в файле). Если вы хотите открыть файл для чтения, второй аргумент можно опустить; режим `'r'` используется по умолчанию.

Следующая короткая программа записывает в файл сообщение «Hello, World»:

```
file_object = open("myfile", 'w')
file_object.write("Hello, World\n")
file_object.close()
```

В зависимости от операционной системы функции `open` также могут быть доступны другие режимы. В большинстве случаев они не являются строго необходимыми. Когда вы начнете писать более сложные программы Python, обращайтесь к справочной документации Python.

Функция `open` может получать необязательный третий аргумент, который определяет способ буферизации операции чтения или записи для этого файла. *Буферизацией* называется процесс хранения данных в памяти до тех пор, пока объем запрашиваемых или записываемых данных не оправдает затраты времени на обращение к диску. Другие параметры `open` управляют кодировкой текстовых файлов и обработкой символов новой строки в текстовых файлах. И снова поначалу они вряд ли будут представлять интерес для вас, но со временем вам стоит узнать о них больше.

13.4. Функции чтения и записи текстовых и двоичных данных

Я уже представила основную функцию чтения текстовых файлов `readline`. Эта функция читает и возвращает одну строку из объекта файла, включая конечный символ новой строки. Если в файле не осталось данных, которые можно было бы прочитать, `readline` возвращает пустую строку, что позволяет (например) легко подсчитать количество строк в файле:

```
file_object = open("myfile", 'r')
count = 0
while file_object.readline() != "":
    count = count + 1
print(count)
file_object.close()
```

В этой конкретной задаче для подсчета всех строк проще воспользоваться встроенным методом `readlines`, который читает *все* строки в файле и возвращает их в виде списка (при этом завершающие символы новой строки остаются на своих местах):

```
file_object = open("myfile", 'r')
print(len(file_object.readlines()))
file_object.close()
```

Конечно, если вы подсчитываете все строки в очень большом файле, такое решение может привести к исчерпанию свободной памяти на компьютере, потому что весь файл читается в память одновременно. Метод `readline` также может вызвать переполнение памяти, если данные читаются из огромного файла, в котором нет ни одного символа новой строки, — впрочем, такая ситуация крайне маловероятна. Для таких ситуаций оба метода, `readline` и `readlines`, могут получать необязательный аргумент с объемом памяти, который может быть прочитан за один раз. За подробностями обращайтесь к справочной документации Python.

Другой способ перебора всех строк в файле основан на использовании объекта `file` в качестве итератора в цикле `for`:

```
file_object = open("myfile", 'r')
count = 0
for line in file_object:
    count = count + 1
print(count)
file_object.close()
```

К преимуществам этого способа можно отнести то, что строки читаются в память по мере надобности, так что даже с очень большими файлами переполнение памяти не угрожает. Другое преимущество — простота и удобство чтения.

Возможная проблема с методом `read` может возникнуть из-за преобразований текстового режима, выполняемых на платформах Windows и Macintosh при выполнении `open` в текстовом режиме, то есть без добавления `b` в режим. В текстовом режиме на Macintosh все последовательности `\r` преобразуются в `"\n"`, тогда как в Windows пары `"\r\n"` преобразуются в `"\n"`. Режим обработки символов новой строки можно задать при открытии файла: укажите `newline="\n"`, `"\r"` или `"\r\n"`, в результате чего только эта последовательность будет использоваться как признак новой строки:

```
input_file = open("myfile", newline="\n")
```

В этом примере признаком новой строки будет считаться только `"\n"`. Если файл открывается в двоичном режиме, параметр newline не нужен, потому что все байты возвращаются точно в том виде, в котором они хранятся в файле.

Методам чтения `readline` и `readlines` соответствуют методы записи `write` и `writelines` (обратите внимание: функция `writeline` не существует). Функция `write` записывает одну строку, которая может содержать внутренние символы новой строки. Пример:

```
myfile.write("Hello")
```

Функция `write` не записывает символ новой строки после записи своего аргумента; если вы хотите, чтобы вывод включал символ новой строки, включите его туда вручную. Если открыть файл в текстовом режиме (`w`), все символы `\n` будут отображаться на завершители данной платформы (то есть `'\r\n'` в Windows или

'\r' на платформах Macintosh). И снова открытие файла с заданным аргументом newline предотвращает эту проблему.

Функция writelines получает в аргументе список строк и записывает их одну за другой в заданный объект файла, не добавляя символы новой строки. Если строки в списке не содержат завершающих символов новой строки, фактически будет происходить их конкатенация в файле. При этом функция writelines является обратной по отношению к readlines: при использовании со списком, возвращенным readlines, она запишет файл, полностью идентичный тому, из которого читала данные функция readlines.

Если файл myfile.txt существует и является текстовым файлом, следующий фрагмент создает точную копию myfile.txt с именем myfile2.txt:

```
input_file = open("myfile.txt", 'r')
lines = input_file.readlines()
input_file.close()
output = open("myfile2.txt", 'w')
output.writelines(lines)
output.close()
```

13.4.1. Двоичный режим

В некоторых случаях требуется прочитать все данные из файла в один объект байтовой строки bytes, особенно если данные не были текстовыми и вы хотите разместить их все в памяти, чтобы работать с ними как с последовательностью байтов. А может быть, вы хотите читать данные из файла как объекты bytes фиксированного размера — скажем, данные читаются без символов новой строки, а каждый блок считается последовательностью символов фиксированного размера. Для этой цели используется метод read. Без аргументов этот метод читает все содержимое файла от текущей позиции и возвращает прочитанные данные в виде объекта bytes. С одним целочисленным аргументом читается указанное количество байтов (или меньше, если в файле не хватит данных для запроса) и возвращает объект bytes заданного размера:

```
input_file = open("myfile", 'rb')
header = input_file.read(4)
data = input_file.read()
input_file.close()
```

Первая строка открывает файл для чтения в двоичном режиме, вторая строка читает первые четыре байта как заголовок, а третья строка читает остаток файла как один блок данных.

Помните, что при открытии файла в двоичном режиме вы работаете только с байтами, но не со строками. Чтобы использовать данные как строки, необходимо декодировать объекты bytes в объекты string. Этот момент играет важную роль с сетевыми протоколами, где потоки данных часто обладают поведением файлов, но должны интерпретироваться как байты, а не как строки.

БЫСТРАЯ ПРОВЕРКА

Зачем добавлять `"b"` в строку режима открытия файла — например, в `open("file", "wb")`?

Допустим, вы хотите открыть файл с именем myfile.txt и записать дополнительные данные в конец файла. Какую команду вы используете для открытия myfile.txt? Какая команда будет использоваться, чтобы повторно открыть файл для чтения данных от начала?

13.5. Чтение и запись с использованием pathlib

Кроме средств работы с путями, рассмотренными в главе 12, объект `Path` также может использоваться для чтения и записи текстовых и двоичных файлов. Данная возможность может быть удобна тем, что она не требует открытия и закрытия файла, а для текстовых и двоичных операций используются разные методы. С другой стороны, есть и ограничение: методы `Path` не могут использоваться для присоединения данных, потому что при записи замещается весь существующий контент:

```
>>> from pathlib import Path
>>> p_text = Path('my_text_file')
>>> p_text.write_text('Text file contents')
18
>>> p_text.read_text()
'Text file contents'
>>> p_binary = Path('my_binary_file')
>>> p_binary.write_bytes(b'Binary file contents')
20
>>> p_binary.read_bytes()
b'Binary file contents'
```

13.6. Экранный ввод/вывод и перенаправление

Встроенный метод `input` выводит запрос и получает входную строку:

```
>>> x = input("enter file name to use: ")
enter file name to use: myfile
>>> x
'myfile'
```

Строка запроса не является обязательной, а символ новой строки в конце введенной строки отсекается. Чтобы прочитать числовые данные с использованием `input`, необходимо явно преобразовать строку, возвращенную `input`, к правильному числовому типу. В следующем примере используется `int`:

```
>>> x = int(input("enter your number: "))
enter your number: 39
>>> x
39
```

Функция `input` выводит запрос в *стандартный вывод* и читает данные из *стандартного ввода*. Для получения низкоуровневого доступа к этим потокам и к *стандартному потоку ошибок* используется модуль `sys`, имеющий атрибуты `sys.stdin`, `sys.stdout` и `sys.stderr`. Эти атрибуты могут рассматриваться как специализированные объекты файлов.

Для `sys.stdin` доступны методы `read`, `readline` и `readlines`. Для `sys.stdout` и `sys.stderr` можно использовать как стандартную функцию `print`, так и методы `write` и `writelines`, которые работают так же, как и с другими объектами файлов:

```
>>> import sys
>>> print("Write to the standard output.")
Write to the standard output.
>>> sys.stdout.write("Write to the standard output.\n")
Write to the standard output.
30  <-- sys.stdout.write возвращает количество записанных символов
>>> s = sys.stdin.readline()
An input line
>>> s
'An input line\n'
```

Перенаправление стандартного ввода позволяет читать данные из файла. Аналогичным образом стандартный вывод или стандартный поток ошибок можно настроить так, чтобы данные записывались в файлы, а затем на программном уровне восстанавливались исходные значения; для этого используются значения `sys.__stdin__`, `sys.__stdout__` и `sys.__stderr__`:

```
>>> import sys
>>> f = open("outfile.txt", 'w')
>>> sys.stdout = f
>>> sys.stdout.writelines(["A first line.\n", "A second line.\n"])
>>> print("A line from the print function")
>>> 3 + 4
>>> sys.stdout = sys.__stdout__
>>> f.close()
>>> 3 + 4
7
```

После выполнения этой команды outfile.txt содержит две строки:
A first line
A second line

outfile.txt содержит три строки:
A first line
A second line
A line from the print function

Функция `print` также может быть перенаправлена на любой файл без изменения стандартного вывода:

```
>>> import sys
>>> f = open("outfile.txt", 'w')
>>> print("A first line.\n", "A second line.\n", file=f)
>>> 3 + 4
7
>>> f.close()
>>> 3 + 4
7
```

outfile.txt содержит:
A first line
A second line

Пока стандартный вывод перенаправлен, вы будете получать запросы и трассировку из стандартного потока ошибок, но не другой вывод. Если вы используете IDLE,

примеры с `sys.__stdout__` не будут работать так, как здесь показано; вам придется использовать интерактивный режим интерпретатора.

Обычно этот прием применяется при выполнении из сценария или программы. Но если вы используете интерактивный режим Windows, временное перенаправление стандартного вывода может использоваться для сохранения информации, которая может уйти с экрана в результате прокрутки. Приведенный ниже короткий модуль реализует набор функций, предоставляющих эту возможность.

Листинг 13.1. Файл mio.py

```
"""mio: модуль (содержит функции capture_output, restore_output,
     print_file и clear_file )"""
import sys
_file_object = None
def capture_output(file="capture_file.txt"):
    """capture_output(file='capture_file.txt'): перенаправление
    стандартного вывода в 'file'."""
    global _file_object
    print("output will be sent to file: {0}".format(file))
    print("restore to normal by calling 'mio.restore_output()'")
    _file_object = open(file, 'w')
    sys.stdout = _file_object

def restore_output():
    """restore_output(): восстановление стандартного вывода
             по умолчанию (также закрывает файл сохранения)"""
    global _file_object
    sys.stdout = sys.__stdout__
    _file_object.close()
    print("standard output has been restored back to normal")

def print_file(file="capture_file.txt"):
    """print_file(file="capture_file.txt"): передача заданного файла
         в стандартный вывод"""
    f = open(file, 'r')
    print(f.read())
    f.close()

def clear_file(file="capture_file.txt"):
    """clear_file(file="capture_file.txt"): очистка содержимого
         заданного файла"""
    f = open(file, 'w')
    f.close()
```

Здесь функция `capture_output()` перенаправляет стандартный вывод в файл (по умолчанию «capture_file.txt»). Функция `restore_output()` восстанавливает стандартный вывод по умолчанию. Если предположить, что функция `capture_output` не вызывалась, `print_file()` направляет файл в стандартный вывод, а `clear_file()` стирает его текущее содержимое.

ПОПРОБУЙТЕ САМИ: ПЕРЕНАПРАВЛЕНИЕ ВВОДА И ВЫВОДА

Напишите код, использующий модуль mio.py из листинга 13.1, который сохраняет весь вывод сценария в файле с именем myfile.txt, восстанавливает стандартный вывод и выводит этот файл на экран.

13.7. Чтение структурированных двоичных данных с использованием модуля struct

Вообще говоря, при работе с собственными файлами в программах Python вам вряд ли потребуется читать или записывать двоичные данные. Для простейшего хранения данных обычно лучше использовать текстовый или байтовый ввод/вывод. В более сложных приложениях Python предоставляет возможность простого чтения или записи произвольных объектов Python (раздел 13.8). Такое решение намного надежнее прямой записи и чтения ваших двоичных данных, поэтому я настоятельно рекомендую использовать именно его.

Однако существует по крайней мере одна ситуация, в которой будет полезно уметь читать или записывать двоичные данные: при работе с файлами, сгенерированными или используемыми другими программами. В этом разделе рассказано, как сделать это с помощью модуля `struct`. За подробностями обращайтесь к справочной документации Python.

Как вы уже видели, Python поддерживает явный двоичный ввод и вывод с использованием байтов (вместо строк) при открытии файла в двоичном режиме. Но поскольку многие двоичные файлы зависят от конкретной структуры для разбора значений, написание собственного кода для чтения и правильного разбиения таких файлов на переменные часто оказывается слишком трудоемким. Вместо этого вы можете воспользоваться стандартным модулем `struct`, который позволяет интерпретировать эти блоки как отформатированные последовательности байтов с некоторым конкретным смыслом.

Допустим, вы хотите прочитать двоичный файл с именем data, который содержит серию записей, сгенерированных программой на языке C. Каждая запись состоит из значения C типа `short int`, значения C типа `double` и последовательности из четырех символов, которая должна интерпретироваться как строка из четырех символов. Требуется прочитать эти данные в список кортежей Python, в котором каждый кортеж содержит целое число, число с плавающей точкой и строку.

Прежде всего необходимо определить *форматную строку*, понятную модулю `struct`. Она сообщает модулю, как упакованы данные в записях. Форматная строка содержит символы, которые используются `struct` для обозначения того, какой тип данных ожидается в той или иной позиции записи. Например, символ `'h'` обозначает одно короткое целое значение C, а символ `'d'` — значение C с плавающей

точкой двойной точности. Как нетрудно догадаться, `'s'` обозначает строку. Любому символу может предшествовать целое число, обозначающее количество значений; в данном случае `'4s'` обозначает строку из четырех символов. Для наших записей правильная форматная строка имеет вид `'hd4s'`. Модуль `struct` поддерживает широкий диапазон числовых, символьных и строковых форматов. За подробностями обращайтесь к документации «*Python Library Reference*».

Прежде чем переходить к чтению записей из файлов, необходимо знать, сколько байтов будет читаться за один раз. К счастью, `struct` включает функцию `calcsize`, которая получает форматную строку в аргументе и возвращает количество байтов, необходимых для хранения данных в этом формате.

Для чтения записей используется метод read, описанный ранее в этой главе. Затем удобная функция `struct.unpack` возвращает кортеж значений, полученный в результате разбора записи в соответствии с форматной строкой. Программа для файла с двоичными данными получается на удивление простой:

```
import struct
record_format = 'hd4s'
record_size = struct.calcsize(record_format)
result_list = []
input = open("data", 'rb')
while 1:
    record = input.read(record_size)  <-- Читает одну запись
    if record == '': ❶
        input.close()
        break
    result_list.append(struct.unpack(record_format, record))  <-- Распаковывает запись в кортеж
```

Если запись пуста, значит, достигнут конец файла, поэтому цикл завершается ❶. Обратите внимание: проверка на корректность файла отсутствует; если последняя запись имеет некорректный размер, функция `struct.unpack` инициирует ошибку.

Как вы, вероятно, уже догадались, модуль `struct` также предоставляет возможность взять значения Python и преобразовать их в упакованные последовательности байтов. Преобразование выполняется функцией `struct.pack`, которая почти (хотя и не полностью) противоположна `struct.unpack`. *Почти* связано с тем, что `struct.unpack` возвращает кортеж значений Python, а `struct.pack` не получает кортеж значений Python, вместо этого в первом аргументе передается форматная строка, а затем дополнительные аргументы в количестве, достаточном для форматной строки. Например, двоичная запись из предыдущего примера может создаваться следующим образом:

```
>>> import struct
>>> record_format = 'hd4s'
>>> struct.pack(record_format, 7, 3.14, b'gbye')
b'\x07\x00\x00\x00\x00\x00\x00\x00\x1f\x85\xebQ\xb8\x1e\t@gbye'
```

Возможности модуля `struct` этим не ограничиваются; в форматную строку можно вставлять другие специальные символы, означающие, что данные должны читаться

с использованием обратного, прямого или машинного порядка байтов (по умолчанию используется машинный порядок), а также указать, следует ли использовать для таких значений, как короткое целое языка C, размер для текущей машины (по умолчанию) или стандартный размер C. Если вам понадобятся эти возможности, вы по крайней мере знаете, что они существуют. За подробностями обращайтесь к справочнику «*Python Library Reference*».

БЫСТРАЯ ПРОВЕРКА: STRUCT

Предложите несколько ситуаций, в которых модуль struct было бы удобно использовать для чтения или записи двоичных данных.

13.8. Сериализация и модуль pickle

Python позволяет записать в файл любую структуру данных, прочитать эту структуру данных из файла, а затем воссоздать ее всего несколькими командами. Данная возможность, называемая *сериализацией*, несколько необычна, но она может быть полезной, поскольку избавит вас от написания многих страниц кода, которые, по сути, просто сохраняют состояние программы в файле (а также аналогичных объемов кода, который не делает ничего, кроме чтения этого состояния).

Python предоставляет эту функциональность в модуле `pickle`. Сериализация — механизм мощный, но достаточно простой в использовании. Предположим, все состояние программы хранится в трех переменных: `a`, `b` и `c`. Это состояние можно сохранить в файле state следующим образом:

```
import pickle
.
.
.
file = open("state", 'wb')
pickle.dump(a, file)
pickle.dump(b, file)
pickle.dump(c, file)
file.close()
```

Неважно, что именно хранилось в переменных `a`, `b` и `c`. Их содержимое может быть любым — от обычных чисел до списков словарей, содержащих экземпляры классов, определяемых пользователем. `pickle.dump` сохранит все.

Чтобы прочитать эти данные при последующем запуске программы, достаточно выполнить следующий код:

```
import pickle
file = open("state", 'rb')
a = pickle.load(file)
b = pickle.load(file)
c = pickle.load(file)
file.close()
```

Вызов `pickle.load` восстанавливает любые данные, которые ранее хранились в переменных `a`, `b` и `c`.

Модуль `pickle` позволяет сохранить, таким образом, почти любые данные. Он работает со списками, кортежами, числами, строками, словарями и практически любыми структурами, созданными из этих типов, в том числе и со всеми экземплярами классов. Он также корректно работает с совместно используемыми объектами, циклическими ссылками и другими сложными структурами памяти, сохраняя совместно используемые объекты только в одном экземпляре и восстанавливая их как совместно используемые объекты, а не как идентичные копии. С другой стороны, объекты кода (в которых Python хранит откомпилированный байт-код) и системные ресурсы (например, файлы или сокеты) сериализоваться не могут.

Как правило, сохранять все состояние программы при помощи метода `pickle` не нужно. Например, многие приложения позволяют одновременно открыть несколько документов. Если сохранить все состояние программы, вы фактически сохраните все открытые документы в одном файле. Простой и эффективный способ сохранения только тех данных, которые действительно необходимы, заключается в том, чтобы написать функцию сохранения, которая заносит все нужные данные в словарь, а затем использовать `pickle` для сохранения словаря. Затем парная функция восстановления загружает словарь (снова с помощью `pickle`), а значения из словаря присваиваются соответствующим переменным программы. Одно из преимуществ этого приема заключается в том, что он исключает возможность чтения значений в неправильном порядке — то есть не в том порядке, в котором эти значения были сохранены. Используя этот прием в предыдущем примере, вы получите код, который выглядит примерно так:

```
import pickle
.
.
.
def save_data():
    global a, b, c
    file = open("state", 'wb')
    data = {'a': a, 'b': b, 'c': c}
    pickle.dump(data, file)
    file.close()

def restore_data():
    global a, b, c
    file = open("state", 'rb')
    data = pickle.load(file)
    file.close()
    a = data['a']
    b = data['b']
    c = data['c']
    .
    .
```

Пример получился немного искусственным. Скорее всего, вам нечасто потребуется сохранять состояние переменных верхнего уровня в интерактивном режиме.

Следующее приложение представляет собой расширенную версию примера с кэшем из главы 7. В этой главе рассматривалась функция, которая проводит интенсивные вычисления на основании трех своих аргументов. Во время выполнения программы функция многократно вызывалась с одинаковыми наборами аргументов. Кэширование результатов в словаре с ключом, составленным из аргументов, позволило добиться значительного повышения быстродействия. Однако при этом сеансы программы запускались по несколько раз в течение дней, недель и месяцев. А следовательно, сериализация кэша позволит избежать того, чтобы начинать кэширование заново с каждым сеансом. Ниже приведена упрощенная версия модуля, который может использоваться для этой цели.

Листинг 13.2. Файл sole.py

```
"""модуль sole: содержит функции sole, save, show"""
import pickle
_sole_mem_cache_d = {}
_sole_disk_file_s = "solecache"
file = open(_sole_disk_file_s, 'rb')  <-- Код инициализации выполняется при загрузке модуля
_sole_mem_cache_d = pickle.load(file)
file.close()

def sole(m, n, t): /// Открытые функции
    """sole(m, n, t): выполнение вычислений с использованием кэша."""
    global _sole_mem_cache_d
    if _sole_mem_cache_d.has_key((m, n, t)):
        return _sole_mem_cache_d[(m, n, t)]
    else:
        # . . . Очень долгие вычисления . . .
        _sole_mem_cache_d[(m, n, t)] = result
        return result

def save():
    """save(): сохранение обновленного кэша на диске."""
    global _sole_mem_cache_d, _sole_disk_file_s
    file = open(_sole_disk_file_s, 'wb')
    pickle.dump(_sole_mem_cache_d, file)
    file.close()

def show():
    """show(): вывод кэша."""
    global _sole_mem_cache_d
    print(_sole_mem_cache_d)
```

Предполагается, что файл кэша уже существует. Если вы захотите поэкспериментировать с ним, добавьте следующий фрагмент для инициализации файла кэша:

```
>>> import pickle
>>> file = open("solecache",'wb')
>>> pickle.dump({}, file)
>>> file.close()
```

И конечно, комментарий `# . . . Очень долгие вычисления` следует заменить настоящими вычислениями. В реальном приложении в такой ситуации, вероятно, для файла кэша использовался бы абсолютный путь. Кроме того, здесь не учитывается возможность параллелизма. Если два пользователя запустят перекрывающиеся сеансы, в итоге останутся только последние сохраненные изменения. Если такая ситуация может создать проблемы, окно перекрытия можно существенно ограничить за счет использования метода обновления словаря в функции `save`.

13.8.1. Когда лучше обойтись без сериализации

Хотя идея использования сериализованных объектов выглядит разумно, также стоит помнить о некоторых недостатках сериализации с использованием pickle:

- Сериализация с использованием pickle не отличается быстротой и эффективностью по объему дискового пространства. Даже сохранение объектов в формате JSON выполняется быстрее и занимает меньше места на диске.
- Сериализация с использованием pickle небезопасна, а загрузка сериализованных данных с вредоносным содержимым может привести к выполнению произвольного кода на вашей машине. Если существует хоть малейший риск того, что сериализованный файл будет доступен кому-то, кто сможет его изменить, лучше обойтись без сериализации.

БЫСТРАЯ ПРОВЕРКА: СЕРИАЛИЗАЦИЯ PICKLE

Подумайте, будет ли сериализация с использованием pickle хорошим решением в следующих ситуациях:

- Сохранение переменных состояния между запусками.
- Хранение рекордных счетов в игре.
- Хранение имен пользователей и паролей.
- Хранение большого словаря со словами английского языка.

13.9. Модуль shelve

Эта тема нетривиальная, но особо сложной ее не назовешь. Объект `shelve` можно представить себе как словарь, хранящий данные в файле на диске вместо оперативной памяти; таким образом, вы пользуетесь всеми удобствами обращения по ключу, но без ограничений объема доступной памяти.

Пожалуй, этот раздел представляет наибольший интерес для людей, работа которых связана с хранением или загрузкой данных из больших файлов, потому что модуль Python `shelve` решает именно эту задачу: он позволяет читать и записывать блоки данных в больших файлах без чтения или записи всего файла. Для приложений, часто обращающихся к большим файлам (например, приложениям баз данных),

экономия времени может оказаться довольно значительной. Как и модуль `pickle` (используемый им во внутренней реализации), модуль `shelve` несложен.

В этом разделе мы исследуем этот модуль на примере адресной книги. Обычно объем адресных книг невелик, что позволяет прочитать весь адресный файл при запуске приложения и записать его при завершении работы. Но если у вас очень много друзей, а адресная книга слишком велика для такого решения, будет лучше использовать `shelve` и не беспокоиться об этом.

Предположим, каждый элемент адресной книги представляет собой кортеж из трех элементов с именем, номером телефона и адресом. Каждый элемент индексируется по фамилии человека, к которому относится запись. Такая структура настолько проста, что приложение может быть реализовано в интерактивном сеансе Python.

Сначала импортируйте модуль `shelve` и откройте адресную книгу. Метод `shelve.open` создает файл адресной книги, если его не существует:

```
>>> import shelve
>>> book = shelve.open("addresses")
```

Теперь добавьте пару записей. Обратите внимание: мы работаем с объектом, возвращаемым `shelve.open`, как со словарем (хотя это словарь, ключами которого могут быть только строки):

```
>>> book['flintstone'] = ('fred', '555-1234', '1233 Bedrock Place')
>>> book['rubble'] = ('barney', '555-4321', '1235 Bedrock Place')
```

Наконец, закройте файл и завершите сеанс:

```
>>> book.close()
```

Что теперь? Снова запустите Python из того же каталога и откройте ту же адресную книгу:

```
>>> import shelve
>>> book = shelve.open("addresses")
```

Но теперь вместо того, чтобы вводить данные, убедитесь в том, что введенные ранее данные никуда не исчезли:

```
>>> book['flintstone']
('fred', '555-1234', '1233 Bedrock Place')
```

Файл адресной книги, созданный вызовом `shelve.open` в первом интерактивном сеансе, работает как словарь, хранящийся на диске. Введенные данные были сохранены на диске, хотя операции записи не выполнялись явно. Именно это делает модуль `shelve`.

В более широком смысле `shelve.open` возвращает объект `shelf`, который поддерживает базовые операции словарей, присваивание или поиск по ключу, `del`, `in` и метод `keys`. Но в отличие от обычных словарей, объекты `shelf` хранят объекты

на диске, а не в памяти. К сожалению, у объектов `shelf` есть одно серьезное ограничение по сравнению со словарями: в качестве ключей могут использоваться только строки (в отличие от широкого спектра типов ключей, разрешенных для словарей).

Важно понимать преимущества объектов `shelf` перед словарями при работе с большими наборами данных. Метод `shelve.open` открывает доступ к файлу; он не загружает весь объект `shelf` в память. Обращения к файлу происходят только по мере надобности (обычно при поиске элемента), а файл имеет такую структуру, что операции поиска выполняются очень быстро. Даже если ваш файл данных очень велик, для нахождения нужного объекта достаточно пары обращений к диску; это может повысить эффективность вашей программы в некоторых отношениях. Программа может запускаться быстрее, потому что ей не нужно считывать в память файл, который может быть очень большим. Кроме того, программа может работать быстрее, потому что для нее доступен больший объем памяти (а следовательно, сокращается необходимость в выгрузке кода из виртуальной памяти). Вы можете работать с наборами данных, которые иначе не поместились бы в памяти.

Использование модуля `shelve` сопряжено с некоторыми ограничениями. Как упоминалось ранее, ключи объектов `shelf` могут быть только строками, но любой сериализуемый объект Python может быть сохранен с ключом в объекте `shelf`. Кроме того, объекты `shelf` не подходят для многопользовательских баз данных, потому что они не позволяют управлять параллельным доступом. Проследите за тем, чтобы объект `shelf` был закрыт после завершения; закрытие иногда необходимо для того, чтобы внесенные изменения (добавления или удаления элементов) были записаны обратно на диск.

В том виде, как он написан, пример в листинге 13.1 прекрасно подходит для `shelve`. Например, не придется следить за тем, чтобы пользователь явно сохранял свою работу на диске. Единственная потенциальная проблема — невозможность полного низкоуровневого контроля при записи обратно в файл.

БЫСТРАЯ ПРОВЕРКА: SHELVE

Работа с объектом `shelf` очень напоминает работу со словарем. Чем отличаются объекты `shelf`? Каких недостатков следует ожидать при использовании объекта `shelf`?

ПРАКТИЧЕСКАЯ РАБОТА 13: ПОСЛЕДНИЕ ИСПРАВЛЕНИЯ В WC

Обратившись к `man`-странице утилиты `wc`, вы увидите, что два ключа командной строки решают очень похожие задачи. С ключом `-c` утилита подсчитывает байты в файле, а с ключом `-m` она подсчитывает символы (некоторые символы Юникода могут состоять из двух и более байтов). Кроме того, если файл задан, утилита должна прочитать и обработать файл, но при его отсутствии для чтения и обработки данных должен использоваться `stdin` (стандартный ввод).

Перепишите свою версию утилиты `wc`, чтобы реализовать как раздельный подсчет байтов и символов, так и возможность чтения из файлов и стандартного ввода.

Итоги

- Механизм файлового ввода/вывода в Python использует различные встроенные функции для открытия, чтения, записи и закрытия файлов.
- Кроме чтения и записи текста, модуль `struct` предоставляет возможность чтения и записи упакованных двоичных данных.
- Модули `pickle` и `shelve` предоставляют простые, безопасные и мощные средства сохранения и загрузки структур данных Python произвольной сложности.

14 Исключения

Эта глава охватывает следующие темы:

- ✓ Объяснение исключений
- ✓ Обработка исключений в Python
- ✓ Использование ключевого слова `with`

В этой главе рассматриваются исключения — языковой механизм, предназначенный специально для обработки аномальных ситуаций во время выполнения программы. Исключения чаще всего используются для обработки ошибок, возникающих в ходе выполнения программ, но они также находят эффективное применение для многих других целей. Python предоставляет набор исключений для многих стандартных ситуаций, а пользователи могут определять собственные исключения для своих целей.

Концепция исключений как механизма обработки ошибок существует достаточно давно. C и Perl, самые популярные языки системного и сценарного программирования, не предоставляют средств обработки исключений, и даже программисты, работающие на таких языках с поддержкой исключений, как C++, нередко не знакомы с ними. Для понимания этой главы читателю не обязательно знать, как работают исключения; здесь приводятся подробные объяснения.

14.1. Знакомство с исключениями

В этом разделе вы узнаете об исключениях и их использовании. Если вы уже знакомы с исключениями, переходите сразу к разделу «Исключения в Python» (раздел 14.2).

14.1.1. Философия ошибок и обработки исключений

Любая программа может столкнуться с ошибками в ходе выполнения. Чтобы продемонстрировать работу с исключениями, мы рассмотрим систему форматирования

текста, которая записывает файлы на диск, а следовательно, может столкнуться с нехваткой места на диске до того, как будут записаны все данные. Есть несколько разных подходов к решению этой проблемы.

Решение 1: оставить проблему без решения

Простейшее решение проблемы дискового пространства — считать, что для любых операций записи в файлы всегда хватает дискового пространства, поэтому беспокоиться не о чем. К сожалению, этот вариант чаще всего встречается на практике. Для маленьких программ, работающих с небольшими объемами данных, он обычно приемлем, но для критических программ он совершенно неудовлетворителен.

Решение 2: все функции возвращают код успеха/неудачи

На следующем уровне обработки ошибок разработчик признает, что ошибки возможны, и определяет методологию их обнаружения и обработки с использованием стандартных языковых механизмов. Это можно делать многими разными способами, но в типичном варианте каждая функция или процедура возвращает код состояния, который показывает, успешно ли был выполнен вызов функции или процедуры. Нормальные результаты могут возвращаться в параметрах, передаваемых по ссылке.

Посмотрим, как это решение может работать в гипотетической программе форматирования текста. Допустим, программа вызывает одну высокоуровневую функцию `save_to_file` для сохранения текущего документа в файле. Эта функция вызывает подфункции для сохранения в файле разных частей документа: `save_text_to_file` для сохранения текста, `save_prefs_to_file` для сохранения пользовательских настроек этого документа, `save_formats_to_file` для сохранения форматов, определяемых пользователем, и т. д. Все эти подфункции могут, в свою очередь, вызывать другие подфункции, сохраняющие меньшие части файла. На самом нижнем уровне располагаются встроенные системные функции, которые выводят в файл примитивные данные и сообщают об успехе или неудаче операций записи в файл.

Код обработки ошибок можно включить в каждую функцию, в которой может произойти ошибка дискового пространства, но вряд ли это имеет смысл. Единственное, что может сделать обработчик ошибки, — вывести диалоговое окно, которое уведомляет пользователя о нехватке места, предлагает удалить какие-нибудь файлы и повторить попытку. Нет смысла дублировать этот код повсюду, где осуществляется запись на диск. Вместо этого следует поместить один блок кода обработки ошибок в основную функцию записи на диск: `save_to_file`.

К сожалению, чтобы функция `save_to_file` могла определить, когда вызывать этот код обработки ошибки, каждая вызываемая ею функция, записывающая данные на диск, сама должна проверять свободное место на диске и возвращать код состояния, обозначающий успех или неудачу операции. Кроме того, функция `save_to_file` должна явно проверять каждый вызов функции записи, хотя ее не интересует,

в какой именно функции произошел сбой. В синтаксисе, сходном с C, код выглядит примерно так:

```
const ERROR = 1;
const OK = 0;
int save_to_file(filename) {
    int status;
    status = save_prefs_to_file(filename);
    if (status == ERROR) {
        ...обработка ошибки...
    }
    status = save_text_to_file(filename);
    if (status == ERROR) {
        ...обработка ошибки...
    }
    status = save_formats_to_file(filename);
    if (status == ERROR) {
        ...обработка ошибки...
    }
    .
    .
    .
}
int save_text_to_file(filename) {
    int status;
    status = ...низкоуровневый вызов для записи размера текста...
    if (status == ERROR) {
        return(ERROR);
    }
    status = ...низкоуровневый вызов для записи текстовых данных...
    if (status == ERROR) {
        return(ERROR);
    }
    .
    .
    .
}
```

То же относится к `save_prefs_to_file`, `save_formats_to_file` и всем остальным функциям, которые либо записывают данные в `filename` напрямую, либо (тем или иным способом) вызывают функции, которые записывают в `filename`.

При такой методологии код обнаружения и обработки ошибок может занять существенную часть программы, потому что каждая функция и процедура, содержащая вызовы, которые могут привести к ошибке, должна содержать код проверки ошибок. Часто у программиста не хватает времени или сил, чтобы реализовать полную обработку ошибок; такие программы получаются ненадежными и в них часто происходят сбои.

Решение 3: механизм исключений

Очевидно, что большая часть кода проверки ошибок в программах предыдущего типа в основном повторяется: код проверяет ошибки при каждой попытке записи

в файл и при обнаружении ошибки передает сообщение вызывающей процедуре. Ошибки дискового пространства обрабатываются только в одном месте: в высокоуровневой функции `save_to_file`. Другими словами, большая часть обработки ошибок составляет служебный код, который связывает место возникновения ошибки с местом ее обработки. На самом деле хотелось бы избавиться от всего лишнего и написать код следующего вида:

```
def save_to_file(filename)
    попытаться выполнить следующий блок
        save_text_to_file(filename)
        save_formats_to_file(filename)
        save_prefs_to_file(filename)
        .
        .
        .
    если на диске кончится свободное пространство во время
        выполнения предыдущего блока
        ...обработать ошибку...

def save_text_to_file(filename)
    ...низкоуровневый вызов для записи размера текста...
    ...низкоуровневый вызов для записи текстовых данных...
    .
    .
    .
```

Код обработки ошибок полностью отделен от низкоуровневых функций; ошибка (если она происходит) генерируется встроенными функциями записи файлов и распространяется прямо в функцию `save_to_file`, где (как предполагается) ею займется код обработки ошибок. Хотя такой код невозможно написать на C, языки с поддержкой исключений реализуют именно такое поведение, и конечно, Python принадлежит к их числу. Исключения позволяют писать более элегантный код и лучше обрабатывать аномальные ситуации.

14.1.2. Более формальное определение исключений

Генерирование исключения называется *выдачей,* или *инициированием*, исключения. В предыдущем примере все исключения инициируются функциями записи на диск, однако инициировать исключения могут любые другие функции, и даже ваш собственный код. В нашем примере исключение инициируется низкоуровневыми функциями записи на диск (не показанными в листинге), если на диске закончится свободное пространство.

Реакция на исключение называется *перехватом* исключения, а код, обрабатывающий исключение, называется *кодом обработки исключения*, или просто *обработчиком исключения*. В приведенном выше примере строка `если на диске...` перехватывает исключение записи на диск, а код, заменяющий строку `...обработать ошибку...`, станет обработчиком исключений записи на диск (нехватки дискового пространства). Также в программе могут быть обработчики для других типов

исключений, и даже другие обработчики для исключений того же типа (но в другом месте кода).

14.1.3. Обработка разных типов исключений

В зависимости от того, какое событие породило исключение, программа может выполнять разные действия. Исключение, инициируемое при нехватке свободного места на диске, должно обрабатываться совсем не так, как исключение, инициируемое при нехватке памяти, а оба этих исключения не имеют ничего общего с исключением, возникающим по ошибке деления на ноль. В одном из вариантов обработки разных типов исключений выполняется глобальная регистрация сообщения об ошибке, обозначающего причину исключения, а все обработчики исключений анализируют это сообщение об ошибке и предпринимают соответствующие действия. На практике другой метод оказался намного более гибким.

Вместо того чтобы определять один тип исключения, Python, как и многие современные языки с поддержкой исключений, определяет разные типы исключений для разных возникающих проблем. В зависимости от произошедшего события могут инициироваться разные типы исключений. Вдобавок коду, перехватывающему исключения, можно приказать перехватывать только исключения определенных типов. Кстати, эта возможность используется в псевдокоде из решения 3 `...если на диске закончится свободное пространство...`. Такой псевдокод указывает, что этот конкретный код обработки ошибок интересуется только исключениями дискового пространства. Другие типы исключений не будут перехватываться данным кодом обработки исключений. Например, они могут перехватываться обработчиком, который обрабатывает числовые исключения, или (если такой обработчик не существует) произойдет аварийное завершение программы с ошибкой.

14.2. Исключения в Python

В оставшейся части этой главы речь пойдет о механизмах обработки исключений, встроенных в Python. Весь механизм исключений Python строится на основе объектно-ориентированной парадигмы, которой он обязан своей гибкостью и расширяемостью. Впрочем, если вы не знакомы с объектно-ориентированным программированием (ООП), вы все равно сможете пользоваться исключениями.

Исключение представляет собой объект, автоматически генерируемый в функциях Python командой `raise`. После того как объект будет сгенерирован, команда `raise` изменяет нормальную последовательность выполнения программы Python. Вместо того чтобы продолжать выполнение со следующей команды после `raise` (или другой команды, породившей исключение), в текущей цепочке вызовов ищется обработчик, способный обработать сгенерированное исключение. Если такой обработчик будет найден, он вызывается и может обратиться к объекту исключения за дополнительной информацией. Если подходящий обработчик не будет найден, то программа аварийно завершается с сообщением об ошибке.

ПРОЩЕ ПРОСИТЬ ПРОЩЕНИЯ, ЧЕМ РАЗРЕШЕНИЯ

Подход к обработке ошибок в Python в целом отличается от подхода в таких языках, как, скажем, Java. Эти языки по возможности стараются выявить как можно больше возможных ошибок заранее, поскольку обработка исключений после их возникновения может обойтись достаточно дорого. Такой стиль описан в первой части этой главы; иногда он обозначается сокращением LBYL (Look Before You Leap, то есть «Смотри, прежде чем прыгать»).

С другой стороны, Python скорее полагается на то, что исключения будут обработаны после их возникновения. И хотя такой подход может показаться рискованным, при разумном использовании исключений код получается менее громоздким и лучше читается, а ошибки обрабатываются только в случае их возникновения. Подход Python к обработке ошибок часто описывается сокращением EAFP (Easier to Ask Forgiveness than Permission, то есть «Проще просить прощения, чем разрешения»).

14.2.1. Типы исключений Python

Программа может генерировать разные типы исключений в зависимости от непосредственной причины ошибки или возникшей аномальной ситуации. В Python 3.6 поддерживаются следующие типы исключений:

```
BaseException
    SystemExit
    KeyboardInterrupt
    GeneratorExit
    Exception
        StopIteration
        ArithmeticError
            FloatingPointError
            OverflowError
           ZeroDivisionError
        AssertionError
        AttributeError
        BufferError
        EOFError
        ImportError
            ModuleNotFoundError
        LookupError
            IndexError
            KeyError
        MemoryError
        NameError
            UnboundLocalError
        OSError
            BlockingIOError
            ChildProcessError
            ConnectionError
                BrokenPipeError
                ConnectionAbortedError
                ConnectionRefusedError
                ConnectionResetError
            FileExistsError
            FileNotFoundError
```

```
        InterruptedError
        IsADirectoryError
        NotADirectoryError
        PermissionError
        ProcessLookupError
        TimeoutError
    ReferenceError
    RuntimeError
        NotImplementedError
        RecursionError
    SyntaxError
        IndentationError
            TabError
    SystemError
    TypeError
    ValueError
        UnicodeError
            UnicodeDecodeError
            UnicodeEncodeError
            UnicodeTranslateError
    Warning
        DeprecationWarning
        PendingDeprecationWarning
        RuntimeWarning
        SyntaxWarning
        UserWarning
        FutureWarning
        ImportWarning
        UnicodeWarning
        BytesWarningException
        ResourceWarning
```

Набор исключений Python имеет иерархическую структуру, на что указывают отступы в списке исключений. Как упоминалось в предыдущей главе, алфавитный список можно получить из модуля `__builtins__`.

Каждый тип исключения представляет собой класс Python, наследующий от родительского типа исключения. Но если вы еще не знакомы с ООП, не беспокойтесь. Скажем, исключение `IndexError` также является `LookupError` (за счет наследования), `Exception` и `BaseException`.

Такая иерархия была создана намеренно: большинство исключений наследует от `Exception`, и все исключения, определяемые пользователем, должны субклассировать `Exception`, а не `BaseException`. Дело в том, что код вида

```
try:
    # Что-то сделать
except Exception:
    # Обработать исключения
```

позволит прервать код в блоке `try` клавишами Ctrl+C без активации кода обработки ошибок, потому что исключение `KeyboardInterrupt` *не* является субклассом `Exception`.

Описания смысла любого типа исключений можно найти в документации, но вы довольно быстро привыкнете к самым распространенным типам исключений в процессе программирования.

14.2.2. Инициирование исключений

Исключения инициируются многими встроенными функциями Python:

```
>>> alist = [1, 2, 3]
>>> element = alist[7]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Код проверки ошибок, встроенный в Python, обнаруживает, что вторая строка запрашивает несуществующий индекс списка, и инициирует исключение `IndexError`. Это исключение распространяется до самого верхнего уровня (интерактивного интерпретатора Python), который обрабатывает его, выдавая сообщение о возникновении исключения.

Исключения также могут инициироваться явно командой `raise`. Простейшая форма этой команды выглядит так:

```
raise exception(args)
```

Часть `exception(args)` создает исключение. Аргументами нового исключения обычно являются значения, которые помогают определить, что же произошло, но об этом чуть позже. После того как исключение будет создано, `raise` запускает его по стеку функций Python, вызванных для перехода к строке, содержащей команду `raise`. Новое исключение передается ближайшему (в стеке) обработчику исключения, предназначенному для данного типа исключений. Если такой обработчик не будет обнаружен до самого верхнего уровня программы, программа завершается с ошибкой или (в интерактивном сеансе) на консоль выводится сообщение об ошибке.

Попробуйте выполнить следующую команду:

```
>>> raise IndexError("Just kidding")
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: Just kidding
```

В результате использования `raise` здесь генерируется сообщение, которое на первый взгляд похоже на все сообщения об ошибках индексирования в Python, встречавшиеся вам до настоящего момента. Если присмотреться внимательнее, становится ясно, что это не так. Ошибка, упоминаемая в сообщении, не так серьезна, как предыдущие.

Передача строкового аргумента при создании исключения — вполне распространенное явление. Многие встроенные исключения Python при передаче первого

аргумента считают, что этот аргумент содержит сообщение, которое должно выводиться для объяснения сути произошедшего. Впрочем, это не всегда так, потому что каждый тип исключения связан с собственным классом, а интерпретация аргументов, ожидаемых при создании нового исключения этого класса, зависит только от определения класса. Кроме того, пользовательские исключения (определяемые вами или другими программистами) часто используются по причинам, не связанным с обработкой ошибок, а следовательно, они могут и не получать текстового сообщения.

14.2.3. Перехват и обработка исключений

У исключений есть одна важная особенность: они не заставляют программу аварийно завершиться с сообщением об ошибке. Определяя соответствующие обработчики исключений, можно сделать так, чтобы часто возникающие аномальные обстоятельства не приводили к аварийному завершению программы; программа выведет сообщение об ошибке, сделает что-то еще, возможно, даже устранит проблему, но аварийного завершения не будет.

Базовый синтаксис перехвата и обработки исключений в Python использует ключевые слова `try`, `except`, а иногда и `else`:

```
try:
    тело
except тип_исключения1 as var1:
    код_исключения1
except тип_исключения2 as var2:
    код исключения2
    .
    .
    .
except:
    обработка_исключения_по_умолчанию
else:
    тело_else
finally:
    тело_finally
```

Сначала выполняется секция `try`, то есть часть `тело` в команде. Если выполнение проходит успешно (то есть в коде не возникают исключения, которые должны перехватываться частью `except`), выполняется `тело_else`, и команда `try` на этом завершается. Так как в команде присутствует секция `finally`, выполняется код `тело_finally`. Если в процессе выполнения `try` произошло исключение, последовательно перебираются секции `except`, и среди них ищется секция с подходящим типом исключения. Если такая секция будет найдена, то инициированное исключение присваивается соответствующей переменной, и выполняется код, связанный с данным типом исключения. Если секция `except тип_исключения1 as var1:` соответствует некоторому исключению `exc`, то будет создана переменная `var`, а `exc` присваивается как значение `var` перед выполнением кода обработки исключения. Вообще говоря,

задействовать переменную `var` не обязательно; можно использовать конструкцию вида `except тип_исключения:`, которая перехватывает исключения указанного типа, но не присваивает их переменным.

Если подходящая секция `except` не будет обнаружена, то инициированное исключение не может быть обработано командой `try` и передается далее по цепочке вызовов в надежде, что оно будет обработано некоторой внешней командой `try`.

В последней секции `except` команды `try` тип исключения может быть не указан; в этом случае она будет перехватывать все типы исключений. Этот прием удобен для отладки и быстрого построения прототипа, но в общем случае так поступать не рекомендуется: секция `except` будет скрывать все ошибки, что приведет к загадочному и непонятному поведению вашей программы.

Секция `else` команды `try` не является обязательной и редко используется разработчиками. Она выполняется в том и только в том случае, если тело `try` выполняется без ошибок.

Секция `finally` команды `try` также не обязательна; она выполняется после выполнения секций `try`, `except` и `else`. Если исключение выдается в блоке `try` и не обрабатывается ни одним из блоков `except`, оно инициируется заново после выполнения блока `finally`. Так как блок `finally` выполняется всегда, в нем обычно содержится код зачистки после обработки исключений: закрытие файлов, сброс переменных и т. д.

ПОПРОБУЙТЕ САМИ: ПЕРЕХВАТ ИСКЛЮЧЕНИЙ

Напишите код, который получает два числа от пользователя и делит первое число на второе. Проверьте и перехватите исключение, возникающее в том случае, если второе число равно 0 (`ZeroDivisionError`).

14.2.4. Определение новых исключений

Вы можете легко определять собственные исключения. В простейшем варианте это делается так:

```
class MyError(Exception):
    pass
```

Код создает класс, который наследует всю функциональность от базового класса `Exception`. Впрочем, вы пока можете не беспокоиться об этом.

Это исключение можно инициировать, перехватывать и обрабатывать точно так же, как и любое другое. Если передать ему один аргумент (и исключение не будет перехвачено и обработано), оно выводится в конце трассировки:

```
>>> raise MyError("Some information about what went wrong")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyError: Some information about what went wrong
```

Конечно, этот аргумент доступен обработчику, который вы тоже напишете:

```
try:
    raise MyError("Some information about what went wrong")
except MyError as error:
    print("Situation:", error)
```

Результат:

```
Situation: Some information about what went wrong
```

Если исключение создается с несколькими аргументами, эти аргументы передаются обработчику в форме кортежа, к которому можно обратиться через переменную `args` объекта ошибки:

```
try:
    raise MyError("Some information", "my_filename", 3)
except MyError as error:
    print("Situation: {0} with file {1}\n error code: {2}".format(
        error.args[0],
 error.args[1], error.args[2]))
```

Результат:

```
Situation: Some information with file my_filename
error code: 3
```

Так как тип исключения является обычным классом Python, который наследует от корневого класса `Exception`, все сводится к созданию отдельной иерархии типов исключений, предназначенной для вашего кода. Если вы читаете эту книгу впервые, пока не обращайте внимания на этот процесс. Вы всегда можете вернуться к нему после того, как прочитаете главу 15. Вопрос о том, как именно создавать собственные исключения, зависит от конкретных потребностей. Если вы пишете маленькую программу, которая может генерировать несколько уникальных ошибок или исключений, субклассируйте класс `Exception`, как это было сделано здесь. Если вы пишете большую многофайловую библиотеку для конкретной области (скажем, для метеорологического прогнозирования), возможно, вам стоит определить уникальный класс `WeatherLibraryException`, а затем определять все уникальные исключения библиотеки как субклассы `WeatherLibraryException`.

БЫСТРАЯ ПРОВЕРКА: ИСКЛЮЧЕНИЯ КАК КЛАССЫ

Если `MyError` наследует от `Exception`, чем отличаются конструкции `except Exception as e` и `except MyError as e`?

14.2.5. Команда assert при отладке программ

Команда `assert` является специализированной формой `raise`:

```
assert выражение, аргумент
```

Если выражение дает результат `False`, а системная переменная `__debug__` равна `True`, инициируется исключение `AssertionError` с необязательным аргументом. Переменная `__debug__` по умолчанию содержит `True`, а для ее отключения следует запустить интерпретатор Python с ключом `-O` или `-OO` или присвоить системной переменной `PYTHONOPTIMIZE` значение `True`. Необязательный аргумент может использоваться для передачи описания.

Если переменная `__debug__` равна `False`, для команд `assert` код не генерируется. Таким образом, при помощи команд `assert` можно оснастить код командами отладочного вывода на стадии разработки и оставить их в коде на будущее без каких-либо затрат ресурсов при нормальном использовании:

```
>>> x = (1, 2, 3)
>>> assert len(x) > 5, "len(x) not > 5"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: len(x) not > 5
```

ПОПРОБУЙТЕ САМИ: КОМАНДА ASSERT

Напишите простую программу, которая запрашивает у пользователя число, а затем при помощи команды `assert` выдает исключение, если число равно 0. Протестируйте программу и убедитесь в том, что команда assert работает; затем отключите ее одним из способов, упомянутых в этом разделе.

14.2.6. Иерархия наследования исключений

В этом разделе я дополнительно разъясню то, что исключения Python имеют иерархическую структуру и что означает эта структура для перехвата исключений секциями `except`.

Следующий код:

```
try:
    тело
except LookupError as error:
    код исключения
except IndexError as error:
    код исключения
```

перехватывает исключения двух типов: `IndexError` и `LookupError`. Так уж случилось, что `IndexError` является субклассом `LookupError`. Если `тело` выдает исключение `IndexError`, эта ошибка сначала проверяется строкой `except LookupError as error:`, а поскольку `IndexError` является частным случаем `LookupError` из-за наследования, первая же проверка `except` срабатывает. Вторая секция `except` никогда не используется, потому что она полностью замещается первой секцией `except`.

С другой стороны, изменение порядка двух секций `except` потенциально может принести пользу; затем первая секция обрабатывает исключения `IndexError`,

а вторая секция обрабатывает все исключения `LookupError`, которые не являются ошибками `IndexError`.

14.2.7. Пример: программа записи на диск в программе Python

В этом разделе мы вернемся к примеру программы форматирования текста, которая должна проверять условия нехватки места на диске при записи документа:

```
def save_to_file(filename) :
    try:
        save_text_to_file(filename)
        save_formats_to_file(filename)
        save_prefs_to_file(filename)
        .
        .
        .
    except IOError:
        ...обработка ошибки...
def save_text_to_file(filename):
    ...низкоуровневый вызов для записи размера текста...
    ...низкоуровневый вызов для записи текстовых данных...
    .
    .
    .
```

Обратите внимание, каким ненавязчивым стал код обработки ошибок. Он упакован в основную последовательность вызовов записи на диск в функции `save_to_file`. Ни одной из дочерних функций записи на диск не нужен код обработки ошибок. Разработчику будет удобно сначала разработать программу и добавить код обработки ошибок позднее. Программисты часто действуют именно так, хотя эта практика не оптимальна.

Еще одно замечание: этот код не реагирует конкретно на ошибки переполнения диска, скорее он реагирует на исключение `IOError`, которое инициируется автоматически встроенными функциями Python каждый раз, когда он не может завершить запрос ввода/вывода по любой причине. Вероятно, для ваших целей этого будет достаточно, но если вам нужно выявить условие переполнения диска, это можно сделать двумя способами. Тело `except` может проверить, сколько места доступно на диске. Если на диске кончилось место, очевидно, проблема связана с переполнением диска и должна быть обработана в теле `except`; в противном случае код в теле `except` может направить `IOError` далее по цепочке вызовов для обработки другой секцией `except`. Если этого решения недостаточно, можно действовать более радикально — например, заглянуть в исходный код функций Python, записывающих данные на диск на языке C, и выдавать собственные исключения `DiskFull` в случае необходимости. Я так поступать не рекомендую, но все же полезно знать об этой возможности на случай, если вам когда-нибудь придется ее использовать.

14.2.8. Пример: исключения в нормальных ситуациях

Исключения чаще всего используются при обработке ошибок, но они также могут быть весьма полезными во многих ситуациях, которые мы бы отнесли к нормальному ходу выполнения. Рассмотрим проблемы в реализации программы, выполняющей функции электронной таблицы. Как и большинство электронных таблиц, она должна выполнять арифметические операции с ячейками, а также поддерживать хранение в ячейках других типов, кроме числовых. В таких приложениях можно считать, что пустые ячейки, задействованные в числовых вычислениях, содержат значение 0, а ячейки с любыми ненулевыми строками считаются недействительными и содержащими значение Python `None`. Любые вычисления, включающие недействительное значение, должны возвращать недействительное значение.

Первым шагом должно стать написание функции, которая преобразует содержимое ячейки электронной таблицы в строку и возвращает соответствующее значение:

```
def cell_value(string):
    try:
        return float(string)
    except ValueError:
        if string == "":
            return 0
        else:
            return None
```

Благодаря средствам обработки исключений Python написать эту функцию совсем несложно. Код пытается преобразовать строку из ячейки в число и вернуть результат в блоке `try` с использованием встроенной функции `float`. Функция `float` выдает исключение `ValueError`, если она не может преобразовать свой строковый аргумент в число, поэтому код перехватывает это исключение и возвращает либо `0`, либо `None` в зависимости от того, является аргумент пустой или непустой строкой.

На следующем шаге нужно как-то решить проблему с тем, что некоторым арифметическим операциям придется иметь дело со значением `None`. В языках без поддержки исключений для этого обычно пришлось бы определять специальный набор арифметических функций, которые проверяют свои аргументы на `None`, а затем использовать эти функции вместо встроенных арифметических функций для выполнения всех операций. Однако такое решение занимает много времени, получается ненадежным и замедляет выполнение программы, потому что вы фактически встраиваете интерпретатор в электронную таблицу. В этом проекте применяется другой подход: все формулы электронной таблицы могут быть функциями Python, которые получают в аргументах координаты ячейки и саму электронную таблицу и вычисляют результат для заданной ячейки при помощи стандартных арифметических операторов Python, используя `cell_value` для получения необходимых значений из электронной таблицы. Вы можете определить функцию `safe_apply`, которая применяет одну из этих формул к соответствующим аргументам в блоке `try` и возвращает либо результат формулы, либо `None` в зависимости от того, была ли формула вычислена успешно:

```
def safe_apply(function, x, y, spreadsheet):
    try:
        return function(x, y, spreadsheet)
    except TypeError:
        return None
```

Этих двух изменений достаточно для того, чтобы встроить концепцию пустых (`None`) значений в семантику электронной таблицы. Попытка реализовать эту возможность без исключений будет довольно поучительной.

14.2.9. Где используются исключения

Исключения — естественные кандидаты для обработки практически любых условий ошибок. К сожалению, обработка ошибок часто добавляется уже тогда, когда основной код программы в целом написан, но исключения прекрасно подходят для управления таким кодом обработки ошибок, добавляемым «задним числом» (или более оптимистично — если добавляется новый код обработки ошибок).

Исключения также в высшей степени полезны в обстоятельствах, в которых большой объем вычислений может быть потерян после того, как вычислительная ветвь вашей программы вышла из-под контроля. Один из таких примеров — электронные таблицы, другие — алгоритмы разбора и алгоритмы оптимального выбора методом ветвей и границ.

БЫСТРАЯ ПРОВЕРКА: ИСКЛЮЧЕНИЯ

Приводят ли исключения Python к вынужденному прерыванию выполнения программы? Предположим, вы хотите, чтобы обращение к словарю `x` всегда возвращало `None`, если ключ не существует в словаре (то есть при выдаче исключения `KeyError`). Какой код вы бы использовали для этого?

ПОПРОБУЙТЕ САМИ: ИСКЛЮЧЕНИЯ

Какой код вы бы использовали для создания нестандартного исключения `ValueTooLarge` и выдачи этого исключения, если значение переменной `x` превышает 1000?

14.3. Менеджеры контекста и ключевое слово with

Некоторые ситуации, например чтение файлов, следуют по прогнозируемой схеме с четко определенным началом и концом. В случае чтения из файла файл часто требуется открыть всего один раз: при чтении данных. Затем файл должен быть закрыт. При использовании исключений такой код работы с файлом может выглядеть так:

```
try:
    infile = open(filename)
    data = infile.read()
finally:
    infile.close()
```

Python 3 предоставляет более универсальный механизм для подобных ситуаций: менеджеры контекста. *Менеджеры контекста* содержат блок и управляют требованиями *входа* и *выхода* из блока; для их пометки используется ключевое слово `with`. Объекты файлов являются менеджерами контекста, что позволяет использовать их для чтения файлов:

```
with open(filename) as infile:
    data = infile.read()
```

Эти две строки кода эквивалентны пяти предыдущим. В обоих случаях вы знаете, что файл будет закрыт сразу же после последнего чтения независимо от того, успешно была выполнена операция или нет. Во втором случае закрытие файла гарантируется тем, что оно является частью управления контекстом объекта файла, так что вам не придется писать код. Иначе говоря, благодаря использованию `with` в сочетании с управлением контекстом (в данном случае объекта файла) вам не придется писать рутинный завершающий код.

Как нетрудно предположить, вы можете создавать собственные менеджеры контекста в случае необходимости. О том, как создавать менеджеры контекста, а также о различных возможностях работы с ними можно узнать в документации модуля `contextlib` стандартной библиотеки.

Менеджеры контекста превосходно подходят для таких задач, как установление и снятие блокировки ресурсов, закрытие файлов, закрепление транзакций в базах данных и т. д. С момента их появления менеджеры контекстов стали фактически лучшей практикой для подобных ситуаций.

БЫСТРАЯ ПРОВЕРКА: МЕНЕДЖЕРЫ КОНТЕКСТА

Допустим, менеджер контекста используется в сценарии, который выполняет чтение и/или запись нескольких файлов. Как вы думаете, какое из следующих решений будет лучшим?

- Заключить весь сценарий в блок, управляемый командой `with`.
- Использовать одну команду `with` для всех операций чтения файлов, а другую — для всех операций записи файлов.
- Использовать команду `with` каждый раз, когда выполняется чтение или запись файла (для каждой строки, например).
- Использовать команду `with` для каждого файла, с которым выполняется чтение или запись.

ПРАКТИЧЕСКАЯ РАБОТА 14: ПОЛЬЗОВАТЕЛЬСКИЕ ИСКЛЮЧЕНИЯ

Вспомните модуль для подсчета вхождений слов, написанный в главе 9. Какие ошибки могут возникнуть в этих функциях? Проведите рефакторинг функций, чтобы корректно обработать эти аномальные ситуации.

Итоги

- Механизм обработки исключений и классы исключений в Python образуют мощную систему для обработки ошибок времени выполнения в вашем коде.
- Блоки `try`, `except`, `else` и `finally`, а также выбор или даже создание новых типов исключений позволяют чрезвычайно точно управлять обработкой или игнорированием исключений.
- Согласно философии Python, ошибки не должны оставаться скрытыми, если только они не будут скрыты специально.
- Типы исключений Python образуют иерархию, потому что исключения, как и все объекты Python, основаны на классах.

ЧАСТЬ 3

Расширенные возможности языка

В предыдущих главах рассматривались базовые возможности Python — те, которые часто используются большинством программистов. В этой части будут описаны некоторые нетривиальные вещи. Возможно, вы не будете использовать их в своей повседневной работе (впрочем, это зависит от специфики работы), но если такая необходимость возникнет, они сыграют очень важную роль.

15 Классы и объектно-ориентированное программирование

Эта глава охватывает следующие темы:

- ✓ Определение классов
- ✓ Использование переменных экземпляра и `@property`
- ✓ Определение методов
- ✓ Определение переменных и методов класса
- ✓ Наследование других классов
- ✓ Создание переменных и приватных методов
- ✓ Наследование нескольких классов

В этой главе рассматриваются классы Python, предназначенные для хранения данных вместе с программным кодом. Вероятно, большинство программистов знакомы с классами и объектами в других языках — я не буду делать предположений относительно знаний конкретных языков или парадигм. Кроме того, в этой главе описаны только объектно-ориентированные конструкции, доступные в Python. Глава не является вводным курсом в объектно-ориентированное программирование (ООП).

15.1. Определение классов

Класс в Python фактически является типом данных. Все типы данных, встроенные в Python, представляют собой классы, и Python предоставляет мощные средства для управления всеми аспектами поведения класса. Класс определяется командой `class`:

```
class MyClass:
    тело
```

`Тело` представляет собой последовательность команд Python — обычно присваиваний переменных и определений функций. Ни присваивания, ни определения функций не обязательны — `тело` может состоять из единственной команды `pass`.

По соглашениям Python идентификаторы классов записываются в «верблюжьем регистре», то есть первая буква каждого внутреннего слова записывается в верхнем регистре, чтобы слова лучше выделялись. После определения класса вы можете создать новый объект типа класса (экземпляр класса), для этого следует вызвать имя класса как функцию:

```
instance = MyClass()
```

15.1.1. Использование экземпляра класса как структуры или записи

Экземпляры классов могут использоваться как структуры или записи. В отличие от структур C или классов Java, поля данных экземпляра необязательно объявлять заранее, они могут создаваться «на ходу». В следующем коротком примере определяется класс с именем `Circle`, создается экземпляр `Circle`, полю `radius` экземпляра присваивается значение, после чего это поле используется для вычисления длины окружности:

```
>>> class Circle:
...     pass
...
>>> my_circle = Circle()
>>> my_circle.radius = 5
>>> print(2 * 3.14 * my_circle.radius)
31.4
```

Как и в Java (а также во многих других языках), для обращения к полям экземпляра/структуры и присваивания им значений используется точечная запись.

Чтобы поля класса инициализировались автоматически, включите в тело класса метод инициализации `__init__`. Эта функция выполняется при каждом создании экземпляра класса; при этом новый экземпляр передается в первом аргументе `self`. Метод `__init__` похож на конструктор в языке Java, но он ничего *не конструирует*, а только *инициализирует* поля класса. Кроме того, в отличие от классов Java и C++, классы Python могут иметь только один метод `__init__`. В следующем примере создается объект с полем radius, которое по умолчанию равно `1`:

```
class Circle:
    def __init__(self): ❶
        self.radius = 1
my_circle = Circle() ❷
print(2 * 3.14 * my_circle.radius) ❸
6.28
my_circle.radius = 5 ❹
print(2 * 3.14 * my_circle.radius) ❺
31.400000000000002
```

По правилам первый аргумент `__init__` всегда называется `self`. Ему присваивается вновь созданный экземпляр при выполнении `__init__` ❶. Затем код использует определение класса. Сначала создается экземпляр класса `Circle` ❷. В следующей

строке используется тот факт, что поле `radius` уже инициализировано ❸. Полю `radius` также можно присвоить другое значение ❹. В итоге результат в последней строке отличается от результата предыдущей команды `print` ❺.

В Python также существует более близкий аналог конструктора — метод `__new__`, который вызывается при создании объекта и возвращает неинициализированный объект. Если только вы не субклассируете неизменяемый тип (например, `str` или `int`) или не используете метакласс для изменения процесса создания объекта, вряд ли вам потребуется переопределять существующий метод `__new__`.

Полноценное использование ООП открывает безграничные возможности, и если вы еще не знакомы с этой темой, я рекомендую обратиться к литературе. А темой оставшейся части этой главы станут ООП-конструкции языка Python.

15.2. Переменные экземпляров

Переменные экземпляров принадлежат к числу базовых возможностей ООП. Еще раз взгляните на класс `Circle`:

```
class Circle:
    def __init__(self):
        self.radius = 1
```

Здесь `radius` — *переменная экземпляра* `Circle`. Другими словами, каждый экземпляр класса `Circle` содержит собственную копию `radius`, а значение, хранящееся в этой копии, может отличаться от значений, хранящихся в переменной `radius` других экземпляров. Python позволяет создавать переменные экземпляров по мере необходимости, присваивая значение полю экземпляра класса:

```
instance.variable = value
```

Если переменная еще не существует, она автоматически создается. Именно так в `__init__` была создана переменная `radius`.

При любом использовании переменных экземпляров (как для присваивания, так и для обращения) требуется *явно указать* экземпляр, то есть используется синтаксис `экземпляр.переменная`. Ссылка на переменную без указания экземпляра обозначает не переменную экземпляра, а локальную переменную в выполняемом методе. В этом отношении Python отличается от таких языков, как C++ и Java, где обращения к переменным экземпляров выглядят так же, как и обращения к локальным переменным методов. Мне нравится требование Python о явном указании экземпляра, потому что оно более наглядно отличает переменные экземпляров от локальных переменных.

ПОПРОБУЙТЕ САМИ: ПЕРЕМЕННЫЕ ЭКЗЕМПЛЯРОВ

Какой код вы бы использовали для создания класса `Rectangle`, представляющего прямоугольник?

15.3. Методы

Метод представляет собой функцию, связанную с конкретным классом. Вы уже видели специальный метод `__init__`, который вызывается для нового экземпляра при его создании. В следующем примере определяется другой метод `area` для класса `Circle`; этот метод вычисляет и возвращает площадь круга для экземпляра `Circle`. Как и большинство методов, определяемых пользователем, метод `area` вызывается в *синтаксисе вызова методов*, напоминающем обращения к переменным экземпляров:

```
>>> class Circle:
...     def __init__(self):
...         self.radius = 1
...     def area(self):
...         return self.radius * self.radius * 3.14159
...
>>> c = Circle()
>>> c.radius = 3
>>> print(c.area())
28.27431
```

Синтаксис вызова метода состоит из экземпляра, за которым следует точка и имя метода, вызываемого для экземпляра. Подобный способ вызова называется *связанным* вызовом метода. С другой стороны, метод также может вызываться как *несвязанный* — для обращения к нему используется его содержащий класс. Этот способ менее удобен, и он почти никогда не применяется на практике, потому что при таком вызове метода в его первом аргументе должен передаваться экземпляр класса, в котором он определен, и код получается менее понятным:

```
>>> print(Circle.area(c))
28.27431
```

По аналогии с `__init__`, метод `area` определяется как функция в теле определения класса. В первом аргументе любого метода передается экземпляр, для которого он вызывается; по действующим правилам он называется `self`. Во многих языках экземпляр, часто называемый `this`, передается неявно, но философия Python требует, чтобы все намерения выражались по возможности конкретно.

Методы могут вызываться с аргументами, если эти аргументы поддерживаются определениями метода. Следующая версия `Circle` добавляет аргумент в метод `__init__`, чтобы вы могли создавать `Circle` с нужным значением `radius` без его присваивания после создания экземпляра:

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return self.radius * self.radius * 3.14159
```

Обратите внимание на два применения `radius`. `self.radius` — переменная экземпляра с именем `radius`. Переменная `radius` без указателя экземпляра — локальный

параметр функции с именем `radius`. Это не одно и то же! На практике локальному параметру функции можно присвоить имя `r` или `rad`, чтобы исключить возможную путаницу.

С таким определением `Circle` можно создавать экземпляры с произвольным значением `radius` одним вызовом класса `Circle`. Следующая команда создает экземпляр `Circle` с полем radius, равным 5:

```
c = Circle(5)
```

В методах можно использовать все стандартные возможности функций Python: значения аргументов по умолчанию, дополнительные аргументы, передачу аргументов по ключевым словам и т. д. Первая строка `__init__` может выглядеть так:

```
def __init__(self, radius=1):
```

Последующие вызовы `circle` будут работать с дополнительным аргументом или без него; `Circle()` вернет экземпляр с radius = 1, а `Circle(3)` вернет экземпляр с radius = 3.

В вызове методов в Python нет ничего мистического; можно считать его сокращенной записью для обычного вызова функций. Встретив вызов метода `instance.method(arg1, arg2, . . .)`, Python преобразует его в обычный вызов функции по следующим правилам:

- Провести поиск имени метода в пространстве имен экземпляра. Если метод был изменен или добавлен в данный экземпляр, ему отдается предпочтение перед методами класса или суперкласса. Поиск проводится аналогично описанной в разделе 15.4.1 этой главы схеме.
- Если метод не обнаружен в пространстве имен экземпляра, поиск продолжается в типе `класса экземпляра`. В предыдущих примерах это будет тип `Circle` — тип экземпляра `c`.
- Если метод и здесь не будет найден, поиск метода продолжается в суперклассах.
- После того как метод будет найден, он напрямую вызывается как обычная функция Python; при этом `экземпляр` передается в первом аргументе функции, а все остальные аргументы вызова метода сдвигаются на одну позицию вправо. Таким образом, запись `instance.method(arg1, arg2, …)` превращается в `class.method(instance, arg1, arg2, …)`.

ПОПРОБУЙТЕ САМИ: ПЕРЕМЕННЫЕ ЭКЗЕМПЛЯРА И МЕТОДЫ

Обновите код класса `Rectangle`, чтобы размеры можно было задавать как при создании экземпляра, так и при создании класса `Circle`. Также добавьте метод `area()`.

15.4. Переменные класса

Переменная класса (class variable) представляет собой переменную, связанную с классом, а не с его конкретным экземпляром и доступную для *всех* экземпляров класса. Переменная класса может использоваться для отслеживания информации на уровне класса — например, количества экземпляров класса, созданных в любой момент времени. Python поддерживает переменные классов, хотя их использование требует несколько больших усилий от разработчика, чем во многих других языках. Кроме того, вам придется следить за возможными конфликтами имен между переменными классов и переменными экземпляров.

Переменная класса создается присваиванием в теле *класса*, а не в функции `__init__`. После того как она будет создана, переменная становится видимой для всех экземпляров класса. Например, с помощью переменной класса можно предоставить доступ к значению `pi` всем экземплярам класса `Circle`:

```
class Circle:
    pi = 3.14159
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return self.radius * self.radius * Circle.pi
```

После того как вы ввели это определение, введите следующие команды:

```
>>> Circle.pi
3.14159
>>> Circle.pi = 4
>>> Circle.pi
4
>>> Circle.pi = 3.14159
>>> Circle.pi
3.14159
```

Этот пример показывает, как должна работать переменная класса; она связана с классом, определившим ее, и содержится в нем. Обратите внимание: в этом примере мы обращаемся к `Circle.pi` до того, как будет создан хотя бы один экземпляр класса. Очевидно, `Circle.pi` существует независимо от любых конкретных экземпляров класса `Circle`.

К переменной класса также можно обратиться из метода класса по имени класса. Мы делаем это в определении `Circle.area`, где функция `area` обращается к `Circle.pi`. Это дает желаемый эффект: правильное значение `pi` читается из класса и используется в вычислениях:

```
>>> c = Circle(3)
>>> c.area()
28.27431
```

Возможно, вам не понравится, что имя класса жестко фиксируется в методах этого класса. Этого можно избежать при помощи специального атрибута `__class__`,

доступного для всех экземпляров класса Python. Этот атрибут возвращает класс, к которому принадлежит экземпляр, например:

```
>>> Circle
<class '__main__.Circle'>
>>> c.__class__
<class '__main__.Circle'>
```

Класс с именем `Circle` во внутренней реализации представляется абстрактной структурой данных; именно эта структура данных будет получена при обращении к атрибуту `__class__` экземпляра `c` (который является экземпляром класса `Circle`). Пример позволяет получить значение `Circle.pi` из `c` без явного указания имени класса `Circle`:

```
>>> c.__class__.pi
3.14159
```

Этот код можно было использовать в методе `area` для того, чтобы избавиться от внутреннего упоминания класса `Circle`; ссылка `Circle.pi` заменяется на `self.__class__.pi`.

15.4.1. Странности переменных классов

У переменных классов есть некоторые странности, о которые вы можете споткнуться, если не подозреваете о них. Если при поиске переменной экземпляра Python не может найти переменную с заданным именем, он пытается найти и вернуть значение из переменной класса с тем же именем. Только если подходящая переменная класса не будет найдена, Python сообщит об ошибке. Переменные классов позволяют эффективно реализовать значения по умолчанию для переменных экземпляров: просто создайте переменную класса с тем же именем и значением по умолчанию, и вы избежите затрат времени и памяти на инициализацию переменной экземпляра при каждом создании экземпляра класса. С другой стороны, это может привести к тому, что программа случайно обратится к переменной экземпляра вместо переменной класса, не выдавая ошибки. В этом разделе я покажу, как переменные классов работают в сочетании с предыдущим примером.

Во-первых, вы можете обратиться к переменной `c.pi` даже в том случае, если `c` не содержит связанной переменной экземпляра с именем `pi`. Python сначала пытается найти такую переменную экземпляра; не обнаружив ее, он продолжает поиск и находит переменную класса `pi` в `Circle`:

```
>>> c = Circle(3)
>>> c.pi
3.14159
```

Возможно, это именно то, что вам нужно... а может, и нет. Этот прием удобен, но он повышает риск ошибок, так что будьте осторожны.

Что произойдет, если вы попытаетесь использовать `c.pi` как полноценную переменную класса и измените ее из одного экземпляра, чтобы изменения были видны всем экземплярам? И снова мы используем предыдущее определение `Circle`:

```
>>> c1 = Circle(1)
>>> c2 = Circle(2)
>>> c1.pi = 3.14
>>> c1.pi
3.14
>>> c2.pi
3.14159
>>> Circle.pi
3.14159
```

Этот пример работает не так, как должен работать для полноценной переменной класса; `c1` теперь содержит собственную копию `pi`, отличную от копии `Circle.pi`, к которой обращается `c2`. Это происходит потому, что присваивание `c1.pi` *создает* переменную экземпляра в `c1`; на переменную класса `Circle.pi` это никак не влияет. Последующие обращения к `c1.pi` возвращают значение из этой переменной экземпляра, тогда как при последующих обращениях к `c2.pi` Python ищет переменную экземпляра `pi` в `c2`, не находит ее и переходит к поиску значения переменной класса `Circle.pi`. Если вы хотите изменить значение переменной класса, обращайтесь к ней по имени класса, а не через переменную экземпляра `self`.

15.5. Статические методы и методы классов

Классы Python также могут содержать методы — аналоги статических методов в таких языках, как Java. Кроме того, в Python поддерживаются методы классов, которые устроены чуть сложнее.

15.5.1. Статические методы

Как и в Java, статические методы можно вызывать даже в том случае, если ни один экземпляр класса не был создан, хотя их также можно вызывать с использованием экземпляра класса. Чтобы создать статический метод, используйте декоратор `@staticmethod`, как показано в листинге 15.1.

Листинг 15.1. Файл circle.py

```
"""Модуль circle: содержит класс Circle."""
class Circle:
    """Класс Circle """
    all_circles = []  <-- Переменная класса содержит список всех созданных экземпляров Circle
    pi = 3.14159
    def __init__(self, r=1):
        """Создать экземпляр Circle с заданным значением radius"""
        self.radius = r
        self.__class__.all_circles.append(self)
    def area(self):
        """Вычислить площадь круга для экземпляра Circle"""
        return self.__class__.pi * self.radius * self.radius

    @staticmethod
    def total_area():
```

При инициализации экземпляра он добавляет себя в список all_circles

```
        """Статический метод для вычисления площади всех Circle """
        total = 0
        for c in Circle.all_circles:
            total = total + c.area()
        return total
```

Теперь введите в интерактивном сеансе следующие команды:

```
>>> import circle
>>> c1 = circle.Circle(1)
>>> c2 = circle.Circle(2)
>>> circle.Circle.total_area()
15.70795
>>> c2.radius = 3
>>> circle.Circle.total_area()
31.415899999999997
```

Также обратите внимание на строки документации. В реальном модуле строки следовало бы сделать более содержательными, перечислить в строке документации класса доступные методы и включить информацию об использовании в строки документации методов:

```
>>> circle.__doc__
'Модуль circle: содержит класс Circle.'
>>> circle.Circle.__doc__
'класс Circle'
>>> circle.Circle.area.__doc__
'Вычислить площадь круга для экземпляра Circle'
```

15.5.2. Методы класса

Методы классов похожи на статические методы в том отношении, что они могут вызываться до того, как будет создан объект класса, и они могут использоваться с указанием экземпляра класса. Однако методы класса неявно получают класс, к которому они принадлежат, в первом параметре, поэтому их можно запрограммировать в более простом виде.

Листинг 15.2. Файл circle_cm.py

```
"""Модуль circle_cm: содержит класс Circle."""
class Circle:
    """Класс Circle"""
    all_circles = []  <-- Переменная содержит список всех созданных экземпляров Circle
    pi = 3.14159
    def __init__(self, r=1):
        """Создает экземпляр Circle с заданным значением radius"""
        self.radius = r
        self.__class__.all_circles.append(self)
    def area(self):
        """Вычислить площадь круга для экземпляра Circle"""
        return self.__class__.pi * self.radius * self.radius
```

```
    @classmethod ❶
    def total_area(cls): ❷
        total = 0
        for c in cls.all_circles: ❸
            total = total + c.area()
        return total
>>> import circle_cm
>>> c1 = circle_cm.Circle(1)
>>> c2 = circle_cm.Circle(2)
>>> circle_cm.Circle.total_area()
15.70795
>>> c2.radius = 3
>>> circle_cm.Circle.total_area()
31.415899999999997
```

Декоратор `@classmethod` используется до определения метода ❶. Параметру класса традиционно присваивается имя `cls` ❷. Вы можете использовать `cls` вместо `self.__class__` ❸.

При использовании метода класса вместо статического метода вам не придется жестко программировать имя класса в `total_area`. В результате все субклассы `Circle` смогут вызвать `total_area` и обращаться к своим полям и методам (а не к полям и методам `Circle`).

ПОПРОБУЙТЕ САМИ: МЕТОДЫ КЛАССА

Напишите метод класса, аналогичный `total_area()`, который возвращает суммарную длину окружности для всех экземпляров `Circle`.

15.6. Наследование

Механизм наследования в Python проще и гибче наследования в компилируемых языках (таких, как Java и C++), потому что динамическая природа Python не накладывает столько ограничений на язык.

Чтобы понять, как используется наследование в Python, начнем с класса `Circle`, рассмотренного ранее в этой главе, и попробуем обобщить идею. Например, можно определить дополнительный класс для квадратов:

```
class Square:
    def __init__(self, side=1):
        self.side = side  ◄—— Длина стороны квадрата
```

Если теперь вы захотите использовать эти классы в графическом редакторе, они должны хранить информацию о том, в каком месте поверхности изображения располагается каждый экземпляр. Для этого можно определить координаты `x` и `y` в каждом экземпляре:

```
class Square:
    def __init__(self, side=1, x=0, y=0):
        self.side = side
        self.x = x
        self.y = y
class Circle:
    def __init__(self, radius=1, x=0, y=0):
        self.radius = radius
        self.x = x
        self.y = y
```

Такое решение работает, но оно приводит к появлению большого количества дублирующегося кода при расширении набора классов геометрических фигур, потому что для каждой фигуры потребуется хранить информацию о текущей позиции. Наверняка вы поняли, к чему я клоню: перед вами стандартная ситуация для применения наследования в объектно-ориентированном языке. Вместо того чтобы определять переменные `x` и `y` в каждом классе геометрической фигуры, можно абстрагировать их в обобщенный класс `Shape`, а каждый класс, определяющий конкретную фигуру, будет наследовать от общего класса. На языке Python это выглядит примерно так:

```
class Shape:
    def __init__(self, x, y):
        self.x = x
        self.y = y
class Square(Shape):  ◄── Указывает, что Square наследует от Shape
    def __init__(self, side=1, x=0, y=0):
        super().__init__(x, y)  ◄── Должен вызвать метод __init__ класса Shape
        self.side = side
class Circle(Shape):  ◄── Указывает, что Circle наследует от Shape
    def __init__(self, r=1, x=0, y=0):
        super().__init__(x, y)  ◄── Должен вызвать метод __init__ класса Shape
        self.radius = r
```

Чтобы использовать наследование в Python, необходимо выполнить (обычно) два требования; оба они продемонстрированы в коде классов `Circle` и `Square`, выделенном жирным шрифтом. Первое требование — определение иерархии наследования; для этого классы, от которых наследует текущий класс, перечисляются в круглых скобках непосредственно за именем определяемого класса. В приведенном коде оба класса `Circle` и `Square` наследуют от `Shape`. Второе, менее очевидное требование — необходимость явного вызова метода `__init__` классов, используемых при наследовании. Python не сделает этого автоматически за вас, но вы можете воспользоваться функцией `super`, чтобы Python автоматически определил нужный класс. Эта задача решается в коде примера вызовом `super().__init__(x,y)`. Код вызывает функцию инициализации `Shape` с инициализируемым экземпляром и правильными аргументами. Без этого в данном примере у экземпляров `Circle` и `Square` не будут инициализированы переменные экземпляров `x` и `y`.

Вместо использования `super` также можно вызвать метод `__init__` класса `Shape` с явным указанием суперкласса в форме `Shape.__init__(self, x, y)`; это также

обеспечит вызов функции инициализации Shape с инициализируемым экземпляром. Однако этот способ будет менее гибким в долгосрочной перспективе, потому что он жестко фиксирует имя суперкласса, а это может создать проблемы в будущем при изменении структуры и иерархии наследования. С другой стороны, использование super может создать проблемы в более сложных случаях. Так как эти два способа плохо сочетаются друг с другом, четко документируйте, какой из них вы используете в своем коде.

Наследование также вступает в силу тогда, когда вы пытаетесь использовать метод, определенный не в самом субклассе или производном классе, а в его суперклассе. Чтобы посмотреть, как это происходит, определите в классе Shape еще один метод с именем move, который сдвигает фигуру с заданным смещением. Этот метод изменяет координаты x и y фигуры на величину, определяемую аргументами метода. Определение Shape принимает следующий вид:

```
class Shape:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self, delta_x, delta_y):
        self.x = self.x + delta_x
        self.y = self.y + delta_y
```

Если вы введете это определение Shape и предыдущие определения Circle и Square, вы сможете провести следующий интерактивный сеанс:

```
>>> c = Circle(1)
>>> c.move(3, 4)
>>> c.x
3
>>> c.y
4
```

Если вы протестируете этот код в интерактивном сеансе, не забудьте заново ввести класс Circle после переопределения класса Shape.

Класс Circle в этом примере не содержал собственного определения метода move, но поскольку он наследует от класса, реализующего move, все экземпляры Circle могли использовать move. В более традиционной терминологии ООП можно сказать, что все методы Python являются виртуальными, то есть если метод не существует в текущем классе, то Python ищет его по списку суперклассов и использует первый найденный метод.

ПОПРОБУЙТЕ САМИ: НАСЛЕДОВАНИЕ

Перепишите код класса Rectangle так, чтобы он наследовал от Shape. У квадратов много общего с прямоугольниками; стоит ли наследовать один класс от другого? И если стоит, то какой класс должен стать базовым, а какой должен наследовать?

Как бы вы написали код добавления метода `area()` для класса `Square`? Следует ли переместить метод area в базовый класс `Shape`, чтобы он наследовался классами Circle, Square и Rectangle? И если переместить метод, к каким проблемам это приведет?

15.7. Наследование и переменные экземпляров и классов

Наследование позволяет экземпляру наследовать атрибуты класса. Переменные экземпляров связываются с экземплярами объектов, и в заданном экземпляре существует только одна переменная экземпляра с заданным именем.

Рассмотрим пример. Используя следующие определения классов:

```
class P:
    z = "Hello"
    def set_p(self):
        self.x = "Class P"
    def print_p(self):
         print(self.x)
class C(P):
    def set_c(self):
        self.x = "Class C"
    def print_c(self):
        print(self.x)
```

выполните следующий код:

```
>>> c = C()
>>> c.set_p()
>>> c.print_p()
Class P
>>> c.print_c()
Class P
>>> c.set_c()
>>> c.print_c()
Class C
>>> c.print_p()
Class C
```

Объект `c` в этом примере является экземпляром класса `C`. Класс `C` наследует от `P`, но `c` не наследует от некоего невидимого экземпляра класса `P`. Он наследует методы и переменные класса непосредственно от `P`. Так как существует только один экземпляр (`c`), любая ссылка на переменную экземпляра `x` в вызове метода для `c` должна относиться к `c.x`. Это утверждение истинно независимо от того, какой класс определяет метод, вызываемый для `c`. Как видите, при вызове для `c` оба метода — `set_p` и `print_p`, определенные в классе `P`, обращаются к той же переменной, что и методы `set_c` и `print_c` при вызове для `c`.

В общем случае для переменных экземпляров требуется именно такое поведение — вполне логично, что обращения к переменным экземпляров с одинаковыми именами будут относиться к одной переменной. Однако иногда требуется другое поведение, которое может быть реализовано при помощи приватных переменных (раздел 15.9).

Переменные классов наследуются, но вы должны действовать осторожно, чтобы избежать конфликтов имен, и помнить об обобщенном поведении, описанном в подразделе, посвященном переменным классов. В следующем примере переменная класса `z` определяется для суперкласса `P`, а обратиться к ней можно тремя способами: через экземпляр `c`, через производный класс `C` или напрямую через суперкласс `P`:

```
>>> c.z; C.z; P.z
'Hello'
'Hello'
'Hello'
```

Но если вы попробуете задать переменную класса `z` через класс `C`, для класса `C` создается новая переменная класса. Этот результат не влияет на саму переменную класса `P` (с обращением через `P`). Но при последующих обращениях через класс `C` или его экземпляр `c` будет видна эта новая переменная, а не оригинал:

```
>>> C.z = "Bonjour"
>>> c.z; C.z; P.z
'Bonjour'
'Bonjour'
'Hello'
```

Аналогичным образом при попытке задать `z` через экземпляр `c` создается новая переменная экземпляра, и у вас появляются три разные переменные:

```
>>> c.z = "Ciao"
>>> c.z; C.z; P.z
'Ciao'
'Bonjour'
'Hello'
```

15.8. Основные возможности классов Python

Все, о чем говорилось выше, относится к основам использования классов и объектов в Python. Прежде чем двигаться дальше, я объединю эти основы в одном примере. В этом разделе мы создадим пару классов, обладающих базовыми возможностями, а потом посмотрим, как работают эти возможности.

Начнем с создания базового класса:

```
class Shape:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self, delta_x, delta_y):
        self.x = self.x + delta_x
        self.y = self.y + delta_y
```

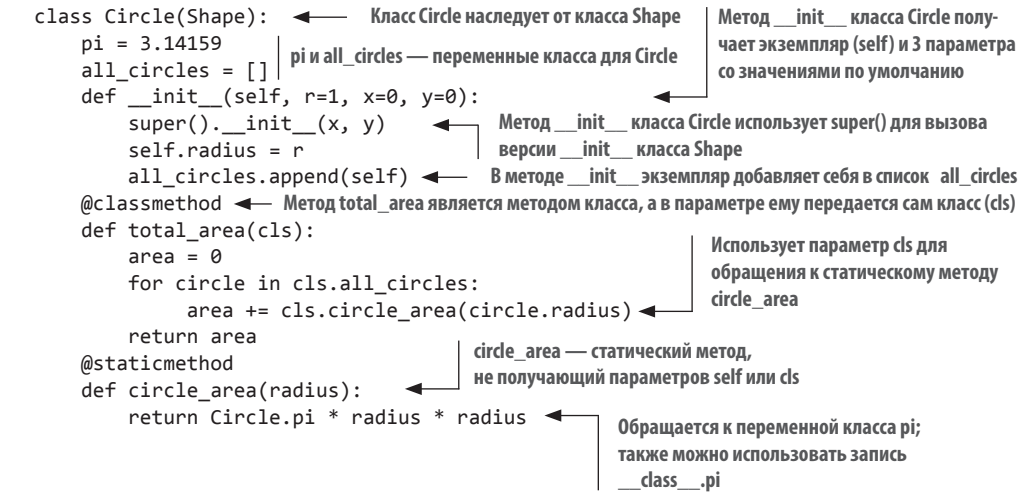
Метод `__init__` получает экземпляр (self) и два параметра

Обращение к переменным экземпляров осуществляется через self

Метод move получает экземпляр (self) и два параметра

Значения переменных экземпляров задаются в методе move

Затем создайте субкласс, наследующий от базового класса `Shape`:



Теперь можно создать несколько экземпляров класса `Circle` и опробовать их на практике. Так как метод `__init__` класса `Circle` имеет параметры по умолчанию, мы можем создать экземпляр `Circle` без передачи каких-либо параметров:

```
>>> c1 = Circle()
>>> c1.radius, c1.x, c1.y
(1, 0, 0)
```

Если же передать параметры, они используются для инициализации полей экземпляра:

```
>>> c2 = Circle(2, 1, 1)
>>> c2.radius, c2.x, c2.y
(2, 1, 1)
```

Если вызвать метод `move()`, Python не находит `move()` в классе `Circle`, поэтому он перемещается по иерархии наследования и использует метод `move()` из `Shape`:

```
>>> c2.move(2, 2)
>>> c2.radius, c2.x, c2.y
(2, 3, 3)
```

Кроме того, поскольку в процессе своей работы метод `__init__` добавляет каждый экземпляр в список, являющийся переменной класса, вы сможете получить экземпляры `Circle`:

```
>>> Circle.all_circles
[<__main__.Circle object at 0x7fa88835e9e8>, <__main__.Circle object at
     0x7fa88835eb00>]
>>> [c1, c2]
[<__main__.Circle object at 0x7fa88835e9e8>, <__main__.Circle object at
     0x7fa88835eb00>]
```

Вы также можете вызвать метод класса `total_area()` класса `Circle` — либо через сам класс, либо через экземпляр:

```
>>> Circle.total_area()
15.70795
>>> c2.total_area()
15.70795
```

Наконец, вы можете вызвать статический метод `circle_area()` — либо через сам класс, либо через экземпляр. Как статический метод, `circle_area` не получает экземпляр класса и ведет себя скорее как независимая функция в пространстве имен класса. На практике статические методы часто используются для включения служебных функций в класс:

```
>>> Circle.circle_area(c1.radius)
3.14159
>>> c1.circle_area(c1.radius)
3.14159
```

Эти примеры демонстрируют базовое поведение классов в Python. Теперь, когда вы ознакомились с основами работы классов, можно переходить к более сложным темам.

15.9. Приватные переменные и приватные методы

Приватная переменная или *приватный метод* не видны за пределами методов класса, в котором они определяются. Приватные переменные и методы полезны по двум причинам: они повышают уровень безопасности и надежности за счет избирательного ограничения доступа к важным или критичным частям реализации объекта, а также предотвращают конфликты имен, которые могут возникнуть из-за применения наследования. Класс может определить приватную переменную и наследовать от класса, определяющего приватную переменную с тем же именем, но это не создает проблем, так как приватность переменных гарантирует хранение их раздельных копий. Приватные переменные упрощают чтение кода, поскольку они явно указывают, что должно использоваться только внутри класса. Все остальное относится к интерфейсу класса.

Многие языки, определяющие приватные переменные, используют для этого ключевое слово «private» или что-нибудь в этом роде. Синтаксис Python проще, к тому же с ним сразу видно, какие переменные или методы являются приватными, а какие нет. Любой метод или переменная экземпляра, имя которой начинается (именно начинается, а не заканчивается!) с *двойного символа подчеркивания* (`__`), являются приватными; все остальное приватным не является.

Например, рассмотрим следующее определение класса:

```
class Mine:
    def __init__(self):
        self.x = 2
        self.__y = 3  <-- Двойное подчеркивание определяет __y как приватную переменную
    def print_y(self):
        print(self.__y)
```

Создайте экземпляр класса с использованием этого определения:

```
>>> m = Mine()
```

Переменная `x` приватной не является, поэтому к ней можно обратиться напрямую:

```
>>> print(m.x)
2
```

Переменная `__y` является приватной. При попытке обратиться к ней напрямую инициируется ошибка:

```
>>> print(m.__y)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: 'Mine' object has no attribute '__y'
```

Метод `print_y` не является приватным, а поскольку он принадлежит классу `Mine`, он может обратиться к переменной `__y` и вывести ее значение:

```
>>> m.print_y()
3
```

Наконец, следует помнить, что механизм реализации приватности *преобразует* имена приватных переменных и приватных методов при компиляции в байт-код. Конкретно к имени переменной присоединяется префикс `_classname`:

```
>>> dir(m)
['_Mine__y', 'x', ...]
```

Это делается для предотвращения любых случайных обращений. При желании разработчик может намеренно имитировать преобразование имен и обратиться к значению. С другой стороны, преобразование в столь легко читаемой форме сильно упрощает отладку.

ПОПРОБУЙТЕ САМИ: ПРИВАТНЫЕ ПЕРЕМЕННЫЕ ЭКЗЕМПЛЯРОВ

Измените код класса `Rectangle`, чтобы переменные размеров сторон были приватными. Какие ограничения на использование класса накладывает это изменение?

15.10. Использование @property для создания более гибких переменных экземпляров

Python позволяет вам как программисту обращаться к переменным экземпляров напрямую — без промежуточных get- и set-методов, часто используемых в Java и других объектно-ориентированных языках. При отсутствии get- и set-методов код классов Python становятся более лаконичным и наглядным, но в некоторых ситуациях get- и set-методы оказываются удобными. Представьте, что значение

нужно проверить перед сохранением в переменной экземпляра или же значение атрибута должно вычисляться «на ходу». В обоих случаях get- и set-методы решат задачу, но за счет потери удобного доступа к переменным Python.

В таких ситуациях следует использовать свойство (property). *Свойства* объединяют возможность передачи обращений к переменной экземпляра через аналоги get- и set-методов и прямолинейные обращения к переменным экземпляров в точечной записи.

Чтобы создать свойство, используйте декоратор свойства с методом, имя которого соответствует имени свойства:

```
class Temperature:
    def __init__(self):
        self._temp_fahr = 0
    @property
    def temp(self):
        return (self._temp_fahr - 32) * 5 / 9
```

Без set-метода такое свойство доступно только для чтения. Чтобы изменить свойство, необходимо добавить set-метод:

```
    @temp.setter
    def temp(self, new_temp):
        self._temp_fahr = new_temp * 9 / 5 + 32
```

Теперь стандартный точечный синтаксис может использоваться как для чтения, так и для записи свойства `temp`. Обратите внимание: имя метода остается неизменным, но декоратор из имени свойства (`temp` в данном случае) и суффикса `.setter` показывает, что определяется set-метод для свойства `temp`:

```
>>> t = Temperature()
>>> t._temp_fahr
0
>>> t.temp
-17.77777777777778

>>> t.temp = 34 ❶
>>> t._temp_fahr
93.2

>>> t.temp ❷
34.0
```

Значение `0` в `_temp_fahr` преобразуется в шкалу Цельсия перед возвращением ❶. Значение `34` преобразуется обратно в шкалу Фаренгейта set-методом ❷.

Одно из больших преимуществ возможности создания свойств в Python заключается в том, что в ходе разработки могут использоваться обычные переменные экземпляров, которые затем будут легко преобразованы в свойства там, где это понадобится, без изменения клиентского кода. Обращения остаются неизменными — в них также используется точечный синтаксис.

ПОПРОБУЙТЕ САМИ: СВОЙСТВА

Измените поля размеров в классе `Rectangle` и преобразуйте их в свойства с get- и set-методами, не допускающими использования отрицательных размеров.

15.11. Правила области видимости и пространств имен для экземпляров классов

Теперь вы знаете все необходимое для того, чтобы понять правила области видимости и пространств имен для экземпляров классов.

В методе класса напрямую доступно *локальное пространство имен* (параметры и переменные, объявленные в методе), *глобальное пространство имен* (функции и переменные, объявленные на уровне модуля) и *встроенное пространство имен* (встроенные функции и встроенные исключения). Поиск по этим трем пространствам имен производится в следующем порядке: локальное, глобальное и встроенное (рис. 15.1).

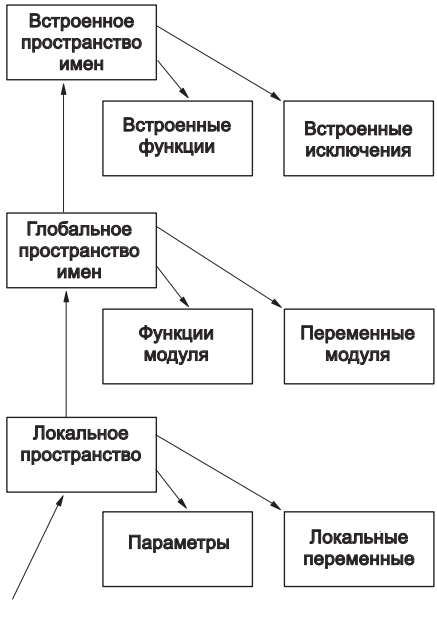


Рис. 15.1. Непосредственно доступные пространства имен

Также через переменную `self` можно обратиться к *пространству имен экземпляра* (переменные экземпляра, приватные переменные экземпляра и переменные экземпляра суперкласса), *пространству имен класса* (методы, переменные класса, приватные методы и приватные переменные класса) и *пространству имен* его

суперкласса (методы суперкласса и переменные класса из суперкласса). Поиск по этим трем пространствам имен производится в следующем порядке: экземпляр, класс и затем суперкласс (рис. 15.2).

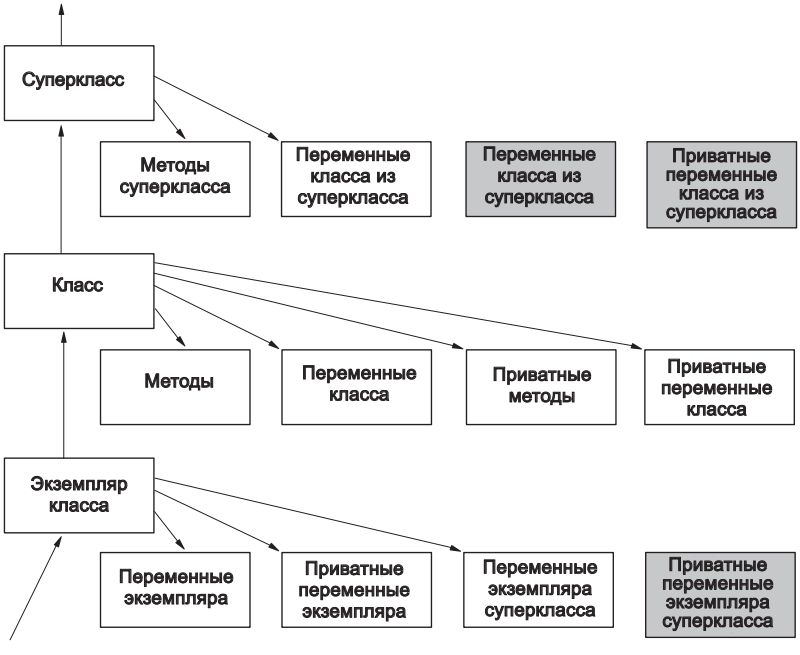


Рис. 15.2. Пространства имен переменной self

К приватным переменным экземпляров суперкласса, приватным методам суперкласса и приватным переменным класса из суперкласса невозможно обратиться через `self`. Класс скрывает эти имена от своих потомков.

Модуль в листинге 15.3 объединяет эти два примера для наглядной демонстрации того, к чему можно обратиться из метода.

Листинг 15.3. Файл cs.py

```
"""Модуль cs: демонстрация области видимости класса."""
mv ="module variable: mv"
def mf():
    return "module function (can be used like a class method in " \
           "other languages): mf()"
class SC:
    scv = "superclass class variable: self.scv"
    __pscv = "private superclass class variable: no access"
    def __init__(self):
        self.siv = "superclass instance variable: self.siv " \
                   "(but use SC.siv for assignment)"
        self.__psiv = "private superclass instance variable: " \
```

```
                      "no access"
    def sm(self):
        return "superclass method: self.sm()"
    def __spm(self):
        return "superclass private method: no access"
class C(SC):
    cv = "class variable: self.cv (but use C.cv for assignment)"
    __pcv = "class private variable: self.__pcv (but use C.__pcv " \
            "for assignment)"
    def __init__(self):
        SC.__init__(self)
        self.__piv = "private instance variable: self.__piv"
    def m2(self):
        return "method: self.m2()"
    def __pm(self):
        return "private method: self.__pm()"
    def m(self, p="parameter: p"):
        lv = "local variable: lv"
        self.iv = "instance variable: self.xi"
        print("Access local, global and built-in " \
              "namespaces directly")
        print("local namespace:", list(locals().keys()))
        print(p)  <-- Параметр

        print(lv)  <-- Локальная переменная
        print("global namespace:", list(globals().keys()))

        print(mv)  <-- Переменная модуля

        print(mf())  <-- Функция модуля
        print("Access instance, class, and superclass namespaces " \

              "through 'self'")
        print("Instance namespace:",dir(self))

        print(self.iv)  <-- Переменная экземпляра

        print(self.__piv)  <-- Приватная переменная экземпляра

        print(self.siv)  <-- Переменная экземпляра суперкласса
        print("Class namespace:",dir(C))
        print(self.cv)  <-- Переменная класса

        print(self.m2())  <-- Метод

        print(self.__pcv)  <-- Приватная переменная экземпляра

        print(self.__pm())  <-- Приватный метод
        print("Superclass namespace:",dir(SC))
        print(self.sm())  <-- Метод суперкласса

        print(self.scv)  <-- Обращение к переменной класса из суперкласса через экземпляр
```

Вывод получается довольно длинным, поэтому мы рассмотрим его по частям.

В первой части локальное пространство имен метода `m` класса `C` содержит параметры `self` (переменная экземпляра) и `p`, а также локальную переменную `lv` (ко всем значениям можно обращаться напрямую):

```
>>> import cs
>>> c = cs.C()
>>> c.m()
Access local, global and built-in namespaces directly
local namespace: ['lv', 'p', 'self']
parameter: p
local variable: lv
```

Затем глобальное пространство имен метода `m` содержит переменную модуля `mv` и функцию модуля `mf` (которая, как описано в предыдущем разделе, может использоваться для предоставления функциональности метода класса). Также есть классы, определенные в модуле (класс `C` и суперкласс `SC`). Ко всем этим классам можно обращаться напрямую:

```
global namespace: ['C', 'mf', '__builtins__', '__file__', '__package__',
     'mv', 'SC', '__name__', '__doc__']
module variable: mv
module function (can be used like a class method in other languages): mf()
```

Пространство имен экземпляра `C` содержит переменные экземпляра `iv` и переменную экземпляра суперкласса `siv` (которая, как описано в предыдущем разделе, не отличается от обычной переменной класса). Оно также содержит преобразованное имя приватной переменной экземпляра `__piv` (к которой можно обратиться через `self`) и преобразованное имя приватной переменной экземпляра суперкласса `__psiv` (к которой обратиться невозможно):

```
Access instance, class, and superclass namespaces through 'self'
Instance namespace: ['_C__pcv', '_C__piv', '_C__pm', '_SC__pscv',
     '_SC__psiv', '_SC__spm', '__class__', '__delattr__', '__dict__',
     '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
     '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__',
     '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
     '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
     '__weakref__', 'cv', 'iv', 'm', 'm2', 'scv', 'siv', 'sm']
instance variable: self.xi
private instance variable: self.__piv
superclass instance variable: self.siv (but use SC.siv for assignment)
```

Пространство имен класса `C` содержит переменную класса `cv` и преобразованное имя приватной переменной класса `__pcv`. К обеим переменным можно обратиться через `self`, но для присваивания необходимо использовать класс `C`. Класс `C` содержит два метода — `m` и `m2`, а также преобразованное имя приватного метода `__pm` (к которому можно обратиться через `self`):

```
Class namespace: ['_C__pcv', '_C__pm', '_SC__pscv', '_SC__spm', '__class__',
     '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__ge__',
     '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
```

```
     '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
     '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
     '__subclasshook__', '__weakref__', 'cv', 'm', 'm2', 'scv', 'sm']
class variable: self.cv (but use C.cv for assignment)
method: self.m2()
class private variable: self.__pcv (but use C.__pcv for assignment)
private method: self.__pm()
```

Наконец, пространство имен суперкласса `SC` содержит переменную класса из суперкласса `scv` (к которой можно обратиться через `self`, но для присваивания необходимо использовать суперкласс `SC`) и метод суперкласса `sm`. Оно также содержит преобразованные имена приватного метода суперкласса `__spm` и приватной переменной суперкласса `__pscv`; и к тому и к другому через `self` обратиться невозможно:

```
Superclass namespace: ['_SC__pscv', '_SC__spm', '__class__', '__delattr__',
     '__dict__', '__doc__', '__eq__', '__format__', '__ge__',
     '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
     '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
     '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
     '__subclasshook__', '__weakref__', 'scv', 'sm']
superclass method: self.sm()
superclass class variable: self.scv
```

Чтобы полностью разобраться в этом примере, потребуется немало времени. Вы можете использовать его для справки или сделать отправной точкой для собственных исследований. Как и со многими конструкциями Python, вы можете составить прочное понимание происходящего, поэкспериментировав с несколькими упрощенными примерами.

15.12. Деструкторы и управление памятью

Ранее вам уже встречались инициализаторы классов (методы `__init__`). Для класса также можно определить деструктор. Тем не менее, в отличие от C++, вам не обязательно создавать и вызывать дескриптор, чтобы гарантировать освобождение памяти, занимаемой экземпляром. Python предоставляет автоматическое управление памятью на базе механизма подсчета ссылок. Другими словами, Python отслеживает количество ссылок на экземпляр; когда это число достигнет нуля, память, используемая экземпляром, освобождается, а у всех объектов Python, ссылки на которые хранятся в вашем экземпляре, счетчики ссылок уменьшаются на единицу. *Вам почти никогда не придется определять деструкторы.*

Иногда может возникнуть ситуация, в которой необходимо явно освободить внешний ресурс при уничтожении объекта. В таких случаях лучше всего воспользоваться менеджером контекста (глава 14). Как упоминалось ранее, модуль `contextlib` из стандартной библиотеки позволяет создать нестандартный менеджер контекста для вашей конкретной ситуации.

15.13. Множественное наследование

В компилируемых языках устанавливаются жесткие ограничения на применение *множественного наследования*, то есть способности объектов наследовать данные и поведение от более чем одного родительского класса. Например, правила множественного наследования в C++ настолько сложны, что многие разработчики предпочитают обходиться без него. В Java множественное наследование запрещено, хотя там существует механизм интерфейсов.

Python не устанавливает подобные ограничения для множественного наследования. Класс может наследовать от любого количества родительских классов точно так же, как он может наследовать от одного родительского класса. В простейшем случае задействованные классы (включая косвенно унаследованные через родительский класс) не содержат переменных экземпляров или методов с одинаковыми именами. В таких случаях наследующий класс ведет себя как результат синтеза своих собственных определений и определений всех его предков. Допустим, класс `A` наследует от классов `B`, `C` и `D`, класс `B` наследует от классов `E` и `F`, а класс `D` наследует от класса `G` (рис. 15.3). Также предположим, что в этих классах нет одноименных методов. В таком случае экземпляр класса `A` может использоваться так, как если бы он был экземпляром любого из классов `B`–`G`, а также как экземпляр `A`; экземпляр класса `B` может использоваться как экземпляр классов `E` или `F`, а также как экземпляр `B`; наконец, экземпляр класса `D` может использоваться как экземпляр класса `G` и как экземпляр `D`. В коде определения классов выглядят так:

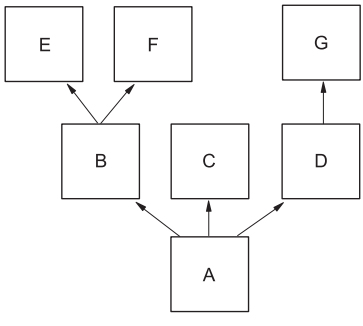


Рис. 15.3. Иерархия наследования

```
class E:
        . . .
class F:
        . . .
class G:
        . . .
class D(G):
        . . .
class C:
        . . .
class B(E, F):
        . . .
class A(B, C, D):
        . . .
```

Ситуация усложняется, когда некоторые классы содержат одноименные методы, потому что Python приходится решать, какое из совпадающих имен должно считаться правильным. Предположим, вы хотите разрешить вызов метода `a.f()` для экземпляра класса `A`, причем метод `f` не определяется в `A`, но определяется во всех классах `F`, `C` и `G`. Какой метод будет вызван?

Ответ определяется тем порядком, в котором Python просматривает базовые классы при поиске метода, не определенного в исходном классе, определяющем метод. В простейших случаях Python перебирает базовые классы исходного класса слева направо, но он всегда просматривает всех предков одного базового класса перед тем, как переходить к следующему базовому классу. При попытке выполнить `a.f()` поиск проходит примерно так:

1. Python сначала ищет метод в классе объекта, для которого вызывается метод, то есть в классе `A`.
2. Так как `A` не определяет метод `f`, Python переходит к поиску в базовых классах `A`. Первым базовым классом `A` является `B`, поэтому Python продолжает поиск в `B`.
3. Так как `B` не определяет метод `f`, Python продолжает поиск в базовых классах `B`. Поиск начинается с первого базового класса `B`, то есть класса `E`.
4. Класс `E` не определяет метод `f` и не имеет базовых классов, поиск в `E` на этом прерывается. Python переходит к классу `B` и обращается к следующему базовому классу `B`, то есть классу `F`.

Класс `F` содержит метод `f`, и поскольку это первый найденный метод с заданным именем, именно этот метод будет использован. Методы с именем `f` в классах `C` и `G` игнорируются.

Конечно, подобная внутренняя логика не способствует созданию самого надежного или простого в сопровождении кода. С более сложными иерархиями в игру вступают другие факторы, которые гарантируют, что ни в одном классе поиск не будет происходить дважды, а также обеспечивают объединенные вызовы `super`.

Вероятно, эта иерархия сложнее тех, которые будут вам встречаться на практике. Если придерживаться более стандартных вариантов применения множественного наследования, как при создании классов примесей (mixin) или добавок (addin), вы сможете легко сохранить удобочитаемость кода и предотвратить конфликты имен.

Некоторые разработчики твердо убеждены, что множественное наследование — это плохо. Конечно, злоупотребления возможны, и ничто в Python не заставляет вас применять его. Одна из величайших опасностей наследования — создание слишком глубоких иерархий, и множественное наследование иногда способствует предотвращению этой проблемы. Эта тема выходит за рамки настоящей книги. Приведенный пример только показывает, как множественное наследование работает в Python, и не пытается продемонстрировать ситуации, в которых его уместно применять (например, классы-примеси или добавки).

ПРАКТИЧЕСКАЯ РАБОТА 15: КЛАССЫ HTML

В этой практической работе вы создадите классы для представления документов HTML. Чтобы упростить задачу, будем считать, что каждый элемент может содержать только текст и один подэлемент. Таким образом, элемент `<html>` содержит только элемент `<body>`, а элемент `<body>` содержит (необязательный) текст и элемент `<p>`, содержащий только текст.

Главное, что вам предстоит реализовать, — это метод `__str__()`, который, в свою очередь, вызывает методы `__str__()` своих подэлементов, так что при вызове функции `str()` для элемента `<html>` возвращается весь документ. Предполагается, что весь текст предшествует подэлементу.

Пример вывода с использованием класса:

```
para = p(text="this is some body text")
doc_body = body(text="This is the body", subelement=para)
doc = html(subelement=doc_body)
print(doc)

<html>
<body>
This is the body
<p>
this is some body text
</p>
</body>
</html>
```

Итоги

- Определение класса фактически означает создание нового типа данных.
- Метод `__init__` используется для инициализации данных при создании нового экземпляра класса, но это не конструктор.
- Параметр `self` ссылается на текущий экземпляр класса и передается в первом параметре методов класса.
- Статические методы могут вызываться без создания экземпляра класса, поэтому им параметр `self` не передается.
- Методам классов вместо `self` передается параметр `cls`, который является ссылкой на класс.
- Все методы Python являются виртуальными. Иначе говоря, если метод не переопределяется в субклассе и не является приватным для суперкласса, он доступен для всех субклассов.
- Переменные классов наследуются от суперклассов, если только они не начинаются с двух символов подчеркивания (`__`) — в этом случае они считаются приватными и не видны субклассам. Методы могут быть назначены приватными аналогичным образом.
- Свойства позволяют реализовать атрибуты с определением get- и set-методов, но при этом они ведут себя как обычные атрибуты экземпляров.
- В Python поддерживается множественное наследование, которое часто используется с примесями.

16 Регулярные выражения

Эта глава охватывает следующие темы:

- ✓ Определение регулярных выражений
- ✓ Создание регулярных выражений со специальными символами
- ✓ Использование необработанных строк в регулярных выражениях
- ✓ Извлечение согласованного текста из строк
- ✓ Замена текста регулярными выражениями

У кого-то может возникнуть вопрос, почему в этой книге вообще рассматриваются регулярные выражения. Регулярные выражения реализуются в одном модуле Python. Эта тема достаточно сложная, поэтому регулярные выражения даже не включаются в стандартную библиотеку в таких языках, как C или Java. Но если вы работаете на Python, скорее всего, вам придется заниматься разбором текста. В этом случае регулярные выражения слишком полезны, чтобы ими пренебрегать. Если вы используете Perl, Tcl или Linux/UNIX, возможно, вы уже знакомы с регулярными выражениями; если нет — эта глава познакомит вас с ними.

16.1. Что такое регулярное выражение?

Регулярное выражение представляет собой шаблон для распознавания, а часто и для извлечения данных из текста. Если регулярное выражение распознает фрагмент текста или строки, говорят, что оно *совпадает* с этим текстом или строкой. Регулярное выражение определяется строкой, в которой определенные символы (называемые *метасимволами*) могут иметь специальный смысл, что позволяет одному регулярному выражению совпадать со многими разными строками.

Регулярные выражения проще понять на примере, чем из объяснений. Следующая программа с регулярным выражением подсчитывает, сколько строк в текстовом

файле содержит слово *hello*. Строка, в которой слово *hello* встречается более одного раза, учитывается только один раз:

```
import re
regexp = re.compile("hello")
count = 0
file = open("textfile", 'r')
for line in file.readlines():
    if regexp.search(line):
        count = count + 1
file.close()
print(count)
```

Программа начинается с импортирования модуля регулярных выражений Python с именем `re`. Затем она получает строку `"hello"` как *текстовое регулярное выражение* и преобразует его в *откомпилированное регулярное выражение* функцией `re.compile`. Компиляция не является строго необходимой, но откомпилированные регулярные выражения могут заметно ускорить выполнение программы, поэтому они почти всегда используются в программах, обрабатывающих большие объемы текста.

Для чего может использоваться регулярное выражение, откомпилированное из строки `"hello"`? Для распознавания других экземпляров слова `"hello"` в других строках; иначе говоря, с его помощью можно определить, содержит ли другая строка подстроку `"hello"`. Эта задача решается методом `search`, который возвращает `None`, если совпадение регулярного выражения не найдено в строке-аргументе; Python интерпретирует `None` как `False` в логическом контексте. Если совпадение регулярного выражения будет найдено в строке, Python возвращает специальный объект, при помощи которого можно получить различную информацию о совпадении (например, в какой позиции строки оно было обнаружено). Эта тема будет рассмотрена позднее.

16.2. Регулярные выражения со специальными символами

У предыдущего примера есть один недостаток: он правильно подсчитывает, сколько строк содержат `"hello"`, но игнорирует строки с `"Hello"`, потому что при поиске совпадения учитывается регистр символов.

Одно из возможных решений заключается в том, чтобы взять два регулярных выражения — одно для `"hello"`, другое для `"Hello"` — и проверить оба для каждой строки. Тем не менее лучше воспользоваться более сложными возможностями регулярных выражений. Приведите вторую строку к следующему виду:

```
regexp = re.compile("hello|Hello")
```

В этом регулярном выражении используется специальный символ | (вертикальная черта). *Специальные символы* в регулярных выражениях не интерпретируются буквально, а имеют особый смысл. | означает «*или*», поэтому регулярное выражение совпадает с `"hello"` *или* `"Hello"`.

В другом возможном решении задачи используется регулярное выражение

```
regexp = re.compile("(h|H)ello")
```

Кроме использования |, данное регулярное выражение использует *круглые скобки* для группировки, это в данном случае означает, что | выбирает между строчной и прописной буквой *H*. Полученное регулярное выражение совпадает с буквой *h* или *H*, за которой следует *ello*.

Другой способ выполнения поиска выглядит так:

```
regexp = re.compile("[hH]ello")
```

Набор символов, заключенный между специальными символами [и], совпадает с любым отдельным символом в этом наборе. Существует специальная сокращенная запись для обозначения диапазонов символов в [и]. Так, диапазон `[a-z]` совпадает с одним символом от *a* до *z*, а `[0-9A-Z]` — с любой цифрой или символом верхнего регистра, и т. д. Иногда в диапазон `[]` требуется включить дефис; в этом случае его следует поставить на первое место, чтобы избежать определения диапазона: `[-012]` совпадает с дефисом, *0*, *1* или *2* и ни с каким другим символом.

В регулярных выражениях Python поддерживаются и другие специальные символы. Описание всех нюансов их использования в регулярных выражениях выходит за рамки книги. Полный список специальных символов, доступных в регулярных выражениях Python, а также их описания приведены в электронной документации модуля регулярных выражений `re` стандартной библиотеки. В оставшейся части этой главы я буду описывать специальные символы, которые будут использоваться в примерах.

БЫСТРАЯ ПРОВЕРКА: СПЕЦИАЛЬНЫЕ СИМВОЛЫ В РЕГУЛЯРНЫХ ВЫРАЖЕНИЯХ

Какое регулярное выражение вы бы использовали для нахождения строк, представляющих числа от –5 до 5?

Какое регулярное выражение вы бы использовали для совпадения с шестнадцатеричной цифрой? Предполагается, что шестнадцатеричные цифры образуют множество 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, a, B, b, C, c, D, d, E, e, F и f.

16.3. Регулярные выражения и необработанные строки

Функции, которые компилируют регулярные выражения или ищут совпадения, понимают, что некоторые последовательности символов в строках имеют специальный смысл в контексте регулярных выражений. Например, функции регулярных выражений понимают, что `\n` представляет символ новой строки. Но если

в качестве регулярных выражений используются обычные строки Python, функции регулярных выражений обычно не распознают такие специальные последовательности, потому что многие из этих последовательностей также обладают специальным смыслом в обычных строках. `\n`, например, также означает новую строку в контексте обычных строк Python, и Python автоматически заменяет строковую последовательность `\n` символом новой строки до того, как функция увидит эту последовательность. В результате функция компилирует строки со встроенными символами новой строки — не со встроенными последовательностями `\n`.

В случае `\n` эта ситуация ни на что не влияет, потому что функции регулярных выражений правильно интерпретируют символ новой строки и делают именно то, что ожидалось: они ищут для символа совпадение с другим символом новой строки в тексте.

Теперь рассмотрим другую специальную последовательность `\\`, которая представляет *один* символ `\` в регулярных выражениях. Допустим, вы хотите проверить текст на вхождение строки `"\ten"`. Поскольку вы уже знаете, что для представления символа обратной косой черты используется последовательность `\\`, можно попытаться поступить так:

```
regexp = re.compile("\\ten")
```

Пример компилируется нормально, но работает неправильно. Проблема в том, что последовательность `\\` также обозначает один символ `\` в строках Python. Перед вызовом `re.compile` Python интерпретирует введенную строку как `\ten`. Эта последовательность символов и будет передана `re.compile`. В контексте регулярных выражений `\t` означает символ *табуляции*, так что откомпилированное регулярное выражение ищет символ табуляции, за которым следуют два символа *en*.

Чтобы исправить эту проблему при использовании обычных строк Python, потребуется последовательность из четырех символо в `\\\\`. Python интерпретирует первые два символа как специальную последовательность, представляющую один символ `\`. То же самое происходит со второй парой `\\`, в результате чего в строке Python оказываются два символа `\\`. Эта строка передается функции `re.compile`, которая интерпретирует два символа `\\` как специальную последовательность регулярного выражения, представляющую один символ `\`. Код выглядит так:

```
regexp = re.compile("\\\\ten")
```

Все это выглядит странно, поэтому в Python предусмотрен способ определения строк, не применяющий нормальные правила Python к специальным символам. Строки, определяемые подобным образом, называются *необработанными* (raw).

16.3.1. Необработанные строки решают проблему

Необработанные строки во всем похожи на обычные строки, если не считать того, что начальной кавычке строки предшествует символ *r*. Несколько примеров необработанных строк:

```
r"Hello"
r"""\tTo be\n\tor not to be"""
r'Goodbye'
r'''12345'''
```

Как видите, вы можете использовать необработанные строки в одиночных и двойных кавычках, в обычном формате или с утроением кавычек. Также при желании вместо *r* можно использовать *R*. Какой бы вариант вы ни выбрали, синтаксис необработанных строк может рассматриваться как инструкция для Python, означающая: «Не обрабатывать специальные последовательности в этой строке». В предыдущих примерах все необработанные строки эквивалентны своим обычным строковым аналогам, кроме второго примера, в котором последовательности `\t` и `\n` не интерпретируются как табуляция и новая строка, а остаются двухсимвольными последовательностями, начинающимися с символа `\`.

Необработанные строки не являются особым типом строк — просто это другой способ *определения* строк. Чтобы понять, что происходит, выполните несколько примеров в интерактивном режиме:

```
>>> r"Hello" == "Hello"
True
>>> r"\the" == "\\the"
True
>>> r"\the" == "\the"
False
>>> print(r"\the")
\the
>>> print("\the")
        he
```

Использование необработанных строк с регулярными выражениями означает, что вам не придется беспокоиться о странных взаимодействиях между специальными последовательностями строк и специальными последовательностями регулярных выражений. Используются только специальные последовательности регулярных выражений. В этом случае предыдущий пример принимает вид

```
regexp = re.compile(r"\\ten")
```

который работает так, как ожидалось. Откомпилированное регулярное выражение ищет один символ `\`, за которым следуют буквы *ten*.

Вам стоит привыкнуть к использованию необработанных строк везде, где определяются регулярные выражения. Именно так мы будем поступать в оставшейся части этой главы.

16.4. Извлечение совпавшего текста из строк

Одно из самых распространенных применений регулярных выражений — простой разбор текста по шаблону. Это одна из тех задач, которые вы должны знать, кроме того, это позволит вам больше узнать о специальных символах регулярных выражений.

Допустим, у вас имеется текстовый файл со списком имен и телефонных номеров. Каждая строка файла выглядит так:

```
фамилия, имя отчество: телефон
```

За фамилией следует запятая и пробел, далее идет имя, пробел, отчество, двоеточие, пробел и телефон.

Однако ситуация усложняется: отчество может отсутствовать, а у телефонного номера может не быть кода города (800-123-4567 или 123-4567). Конечно, можно написать специальный код для разбора данных в строке, но такая работа будет рутинной, а результат ненадежным. Регулярные выражения предоставляют более простое решение.

Начните с составления регулярного выражения, совпадающего со строками в заданном формате. В ближайших абзацах вы встретите несколько новых специальных символов. Не беспокойтесь, если не все они сразу будут понятны; если вы понимаете суть происходящего, это нормально.

Для простоты будем считать, что имена, фамилии и отчества состоят из букв и, возможно, дефисов. Вы можете использовать специальные символы `[]`, упомянутые в предыдущем разделе, для определения шаблона, определяющего только символы имен:

```
[-a-zA-z]
```

Этот шаблон определяет один дефис, один символ нижнего регистра или один символ верхнего регистра.

Чтобы получить совпадение для полного имени (например, McDonald), шаблон необходимо повторить. Метасимвол `+` повторяет то, что находится перед ним, один или более раз для поиска совпадения с обрабатываемой строкой. Таким образом, шаблон

```
[-a-zA-Z]+
```

совпадает с полным именем — Kenneth, McDonald или Perkin-Elmer. Он также совпадает с некоторыми строками, которые не являются именами, такими как – или `-a-b-c-`, но для нашего примера это несущественно.

Что делать с телефонным номером? Специальная последовательность `\d` совпадает с любой цифрой, а дефис вне `[]` обозначает обычный дефис. Хороший шаблон для телефонного номера выглядит так:

```
\d\d\d-\d\d\d-\d\d\d\d
```

Три цифры, за которыми следует дефис, затем три цифры, еще один дефис и четыре цифры. Этот шаблон совпадает только с телефонными номерами, содержащими код города, а в вашем списке могут быть номера, у которых этого кода нет. Лучше всего заключить часть шаблона с кодом города в круглые скобки; создать группу и поставить после группы специальный символ `?`, который означает, что часть, стоящая непосредственно перед `?`, является необязательной:

```
(\d\d\d-)?\d\d\d-\d\d\d\d
```

Этот шаблон совпадает с телефонным номером, который может содержать (а может и не содержать) код города. Аналогичный прием может использоваться для отражения того факта, что у одних людей из списка есть отчество (или инициал), а у других его нет. (Чтобы сделать отчество необязательным, используйте группировку и специальный символ `?`.)

Фигурные скобки `{}` можно использовать для определения количества повторений шаблона, так что в примерах для телефонного номера можно использовать запись:

```
(\d{3}-)?\d{3}-\d{4}
```

Этот шаблон также обозначает необязательную группу из трех цифр, за которой следует дефис, три цифры, дефис и еще четыре цифры.

Запятые, двоеточия и пробелы не имеют специального смысла в регулярных выражениях; они обозначают сами себя.

Объединяя все сказанное, мы получаем шаблон следующего вида:

```
[-a-zA-Z]+, [-a-zA-Z]+( [-a-zA-Z]+)?: (\d{3}-)?\d{3}-\d{4}
```

Вероятно, реальный шаблон будет более сложным: нельзя предполагать, что после запятой следует ровно один пробел, после имени и отчества — ровно один пробел и ровно один пробел после двоеточия. Впрочем, это расширение легко добавить позднее.

Проблема в том, что этот шаблон позволяет проверить, имеет ли строка ожидаемый формат, но мы еще не можем извлечь из него никакие данные. Все, что можно сделать, — написать программу, которая выглядит примерно так:

```
import re
regexp = re.compile(r"[-a-zA-Z]+,"  <-- Фамилия и запятая
                    r" [-a-zA-Z]+"  <-- Имя
                    r"( [-a-zA-Z]+)?"  <-- Необязательное отчество
                    r": (\d{3}-)?\d{3}-\d{4}"  <-- Двоеточие и телефон
                   )
file = open("textfile", 'r')
for line in file.readlines():
    if regexp.search(line):
        print("Yeah, I found a line with a name and number. So what?")
file.close()
```

Обратите внимание: регулярное выражение разбито на части благодаря тому факту, что Python неявно проводит конкатенацию всех строк, разделенных пропусками. С ростом шаблона этот прием сильно упростит сопровождение и понимание шаблона. Он также решает проблему с длиной строки, которая может выйти за правый край экрана.

К счастью, регулярные выражения могут использоваться для извлечения данных из шаблонов, а также проверки существования совпадений. Первым шагом должна стать группировка каждого подшаблона, соответствующего извлекаемому фрагменту данных, при помощи специальных символов `()`. Затем каждому подшаблону

присваивается уникальное имя при помощи специальной последовательности `?P<имя>`:

```
(?P<last>[-a-zA-Z]+), (?P<first>[-a-zA-Z]+)( (?P<middle>([-a-zA-Z]+)))?:
(?P<phone>(\d{3}-)?\d{3}-\d{4}
```

(Обратите внимание: это выражение должно вводиться без разбивки на строки. Из-за нехватки места на печатной странице мы не можем привести его в таком виде.)

И здесь возникает явная путаница: вопросительные знаки в `?P<...>` и специальные символы `?`, указывающие, что отчество и код города не являются обязательными, не имеют ничего общего. То, что они представляются одним символом, — всего лишь неудачное совпадение.

Теперь, когда элементам шаблона присвоены имена, вы можете получить совпадения для этих элементов методом `group`. Дело в том, что при успешном совпадении функция `search` возвращает не только значение `True`, но и структуру данных с информацией о совпадении. Вы можете написать простую программу для извлечения имен и телефонов, преобразования списка и их вывода:

```
import re
regexp = re.compile(r"(?P<last>[-a-zA-Z]+),"  <-- Фамилия и запятая
                    r" (?P<first>[-a-zA-Z]+)"  <-- Имя
                    r"( (?P<middle>([-a-zA-Z]+)))?"  <-- Необязательное отчество
                    r": (?P<phone>(\(\d{3}-)?\d{3}-\d{4})"  <-- Двоеточие и телефон
                   )
file = open("textfile", 'r')
for line in file.readlines():
    result = regexp.search(line)
    if result == None:
        print("Oops, I don't think this is a record")
    else:
        lastname = result.group('last')
        firstname = result.group('first')
        middlename = result.group('middle')
        if middlename == None:
            middlename = ""
        phonenumber = result.group('phone')
    print('Name:', firstname, middlename, lastname,' Number:', phonenumber)
file.close()
```

Здесь стоит обратить внимание на несколько интересных моментов:

- Чтобы узнать, было ли найдено совпадение, можно проверить значение, возвращаемое `search`. Если оно равно `None`, то поиск завершился неудачей; в противном случае поиск прошел успешно и вы можете извлечь информацию из объекта, возвращаемого `search`.
- Метод `group` используется для извлечения данных, совпавших с именованными подшаблонами. При вызове передается имя интересующего вас подшаблона.
- Так как подшаблон отчества не является обязательным, нельзя быть уверенными в том, что ему будет соответствовать значение, даже если поиск в целом был

успешным. Если общее совпадение найдено, но для отчества его не существует, при использовании `group` для обращения к данным соответствующего подшаблона возвращается `None`.

- Одна часть телефонного номера не является обязательной, другая обязательна. Если совпадение будет найдено, с подшаблоном `phone` будет связан текст, поэтому вам не придется беспокоиться о том, что он имеет значение `None`.

ПОПРОБУЙТЕ САМИ: ИЗВЛЕЧЕНИЕ СОВПАВШЕГО ТЕКСТА

При международных звонках обычно указывается символ `+` и код страны. Если предположить, что код страны состоит из двух цифр, как бы вы изменили приведенный выше код? (Код страны также присутствует не во всех номерах.)

Как бы вы реализовали обработку кодов стран, содержащих от одной до трех цифр?

16.5. Замена текста с использованием регулярных выражений

Помимо извлечения подстрок из текста, модуль регулярных выражений Python также может использоваться для поиска строк в тексте и замены их другими строками. Эта задача решается методом замены `sub`. В следующем примере все вхождения `"the the"` (вероятно, опечатка) заменяются одним экземпляром `"the"`:

```
>>> import re
>>> string = "If the the problem is textual, use the the re module"
>>> pattern = r"the the"
>>> regexp = re.compile(pattern)
>>> regexp.sub("the", string)
'If the problem is textual, use the re module'
```

Метод `sub` использует регулярное выражение, для которого делается вызов (`regexp` в данном случае), для сканирования своего второго аргумента (`string`) и строит новую строку, заменяя все совпадающие подстроки значением первого аргумента (`"the"`).

Но что если вы захотите заменить совпавшие подстроки новыми, зависящими от текста совпадениями? Здесь-то и проявляется элегантность Python. Первый аргумент `sub` — заменяющая строка, `"the"` в данном примере, — может вообще не быть строкой. Это может быть и функция. Если это функция, Python вызывает ее для текущего объекта совпадения и дает возможность функции вычислить и вернуть строку замены.

Чтобы понять, как работает этот механизм, мы построим пример, который получает строку с целочисленными значениями (без точки и дробной части) и возвращает строку с теми же числовыми значениями в формате с плавающей точкой (с точкой и завершающим нулем):

```
>>> import re
>>> int_string = "1 2 3 4 5"
>>> def int_match_to_float(match_obj):
...     return(match_obj.group('num') + ".0")
...
>>> pattern = r"(?P<num>[0-9]+)"
>>> regexp = re.compile(pattern)
>>> regexp.sub(int_match_to_float, int_string)
'1.0 2.0 3.0 4.0 5.0'
```

Шаблон ищет число, состоящее из одной или нескольких цифр (часть `[0-9]+`). Но ему также назначается имя (часть `?P<num>...`), так что функция замены может получить любую совпавшую подстроку по имени. Затем метод `sub` сканирует строку аргумента `"1 2 3 4 5"` и ищет совпадения для `[0-9]+`. При обнаружении совпадающей подстроки `sub` создает объект, точно определяющий подстроку совпадения, и вызывает функцию `int_match_to_float` с этим объектом в качестве единственного аргумента. `int_match_to_float` использует `group` для извлечения совпавшей подстроки из объекта совпадения (по имени группы `num`) и строит новую строку, выполняя конкатенацию совпавшей подстроки с `".0"`. `sub` возвращает новую строку и вставляет ее в общий результат. Наконец, `sub` продолжает сканирование сразу же от позиции, в которой было обнаружено последнее совпадение, и продолжает работать до того момента, когда не сможет найти новых совпадений.

ПОПРОБУЙТЕ САМИ: ЗАМЕНА ТЕКСТА

В упражнении из раздела 16.4 вы доработали регулярное выражение для телефонного номера, чтобы оно также распознавало код страны. Как бы вы использовали функцию, чтобы любые номера, не содержащие кода страны, теперь имели код +1 (код страны для США и Канады)?

ПРАКТИЧЕСКАЯ РАБОТА 16: НОРМАЛИЗАЦИЯ ТЕЛЕФОННЫХ НОМЕРОВ

В США и Канаде телефонные номера состоят из десяти цифр, обычно разбитых на код города из трех цифр, код шлюза из трех цифр и код станции из четырех цифр. Как упоминалось в разделе 16.4, таким номерам может предшествовать код страны +1. Однако на практике существует много способов форматирования телефонных номеров — (NNN)NNN-NNNN, NNN-NNN-NNNN, NNN NNN-NNNN, NNN.NNN.NNNN, NNN NNN NNNN и т. д. Кроме того, код страны может отсутствовать, может не содержать + и обычно (но не всегда) отделяется от номера дефисом или пробелом.

В этом упражнении вам предлагается создать нормализатор телефонных номеров, который получает номер в любом формате и возвращает нормализованный номер вида 1-NNN-NNN-NNNN. Все номера в следующей таблице являются допустимыми:

+1 223-456-7890	1-223-456-7890	+1 223 456-7890
(223) 456-7890	1 223 456 7890	223.456.7890

Дополнительное задание: первой цифрой кода города и кода шлюза могут быть только цифры 2–9, а второй цифрой кода города не может быть 9. Используйте эту информацию для проверки ввода, чтобы при недействительном номере выдавалось исключение `ValueError`.

Итоги

- За полным списком и объяснением специальных символов регулярных выражений обращайтесь к документации Python.
- Кроме методов `search` и `sub`, существуют другие методы для разбивки строк, получения расширенной информации из объектов совпадения, определения позиций подстрок в основной строке и точного управления процессом поиска совпадения в строке-аргументе.
- Кроме специальной последовательности `\d`, которая может использоваться для обозначения цифрового символа, в документации перечислены многие другие специальные последовательности.
- Также существуют флаги регулярных выражений, с помощью которых можно управлять более экзотическими аспектами особенно сложного поиска совпадений.

17 Типы данных как объекты

Эта глава охватывает следующие темы:

- ✓ Обработка типов данных как объектов
- ✓ Использование типов данных
- ✓ Создание пользовательских классов
- ✓ Объяснение утиной типизации
- ✓ Использование специальных атрибутов метода
- ✓ Субклассификация встроенных типов

К настоящему моменту вы уже освоили базовые типы Python и научились создавать собственные типы данных с использованием классов. Во многих языках возможности в отношении типов данных этим и исчерпываются. Однако Python является языком с динамической типизацией; это означает, что типы определяются во время выполнения, а не во время компиляции. И этот факт — одна из причин, по которым Python так прост в использовании. Кроме того, он позволяет (а иногда и заставляет) проводить вычисления с типами объектов, а не с самими объектами.

17.1. Типы тоже являются объектами

Запустите сеанс Python и попробуйте выполнить следующие команды:

```
>>> type(5)
<class 'int'>
>>> type(['hello', 'goodbye'])
<class 'list'>
```

В этом примере вы впервые встречаетесь со встроенной функцией `type` в Python. Эта функция может применяться к любому объекту Python; она возвращает тип объекта. В данном случае функция сообщает, что `5` является целым числом (`int`), а `['hello', 'goodbye']` является списком, — хотя вы, наверное, и так догадывались.

Интереснее другое: в ответ на вызовы `type` Python возвращает объекты; `<class 'int'>` и `<class 'list'>` являются экранными представлениями возвращенных объектов. Какой объект вернет вызов `type(5)`? Это нетрудно узнать. Просто вызовите `type` для этого результата:

```
>>> type_result = type(5)
>>> type(type_result)
<class 'type'>
```

Вызов `type` возвращает объект, типом которого является `<class 'type'>`; его можно назвать *объектом типа*. Объект типа — просто очередная разновидность объекта Python, единственной выдающейся особенностью которой становится путаница, возникающая из-за названия. Если сказать, что объект типа относится к типу `<class 'type'>`, это прозвучит настолько же понятно, как если бы фраза была произнесена на китайском.

17.2. Использование типов

Итак, вы знаете, что типы данных могут представляться объектами типов Python; что с ними можно сделать? Их можно сравнивать, потому что любые два объекта Python можно сравнить:

```
>>> type("Hello") == type("Goodbye")
True
>>> type("Hello") == type(5)
False
```

Типы `"Hello"` и `"Goodbye"` совпадают (оба являются строками), но типы `"Hello"` и `5` различаются. Среди прочего, этот прием может использоваться для реализации проверки типов в функциях и определениях методов.

17.3. Типы и пользовательские классы

Самая распространенная причина, по которой в программе может проверяться тип объекта (особенно для экземпляров пользовательских классов), — проверка того, является ли конкретный объект экземпляром некоторого класса. Определив, относится ли объект к заданному типу, код может обработать его соответствующим образом. Следующий пример существенно прояснит ситуацию. Для начала определите пару пустых классов для создания простой иерархии наследования:

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
```

Теперь создайте экземпляр класса `B`:

```
>>> b = B()
```

Как и следовало ожидать, при вызове функции `type` для `b` вы узнаете, что `b` является экземпляром класса `B`, определенного в текущем пространстве имен `__main__`:

```
>>> type(b)
<class '__main__.B'>
```

Ту же информацию можно получить обращением к специальному атрибуту `__class__` экземпляра:

```
>>> b.__class__
<class '__main__.B'>
```

Мы будем работать с этим классом для извлечения дополнительной информации, поэтому сохраните его в переменной:

```
>>> b_class = b.__class__
```

Теперь, чтобы показать, что в Python нет ничего, кроме объектов, проверьте, что класс, полученный от `b`, является классом, который вы определили с именем `B`:

```
>>> b_class == B
True
```

В этом примере сохранять класс `b` не обязательно, но я хотела наглядно показать, что класс всего лишь является очередным объектом Python и его можно сохранить или передать при вызове, как любой другой объект Python.

Для заданного класса `b` его имя можно узнать из атрибута `__name__`:

```
>>> b_class.__name__
'B'
```

Чтобы узнать, от каких классов наследует данный класс, обратитесь к атрибуту `__bases__`, который содержит кортеж всех его базовых классов:

```
>>> b_class.__bases__
(<class '__main__.A'>,)
```

Атрибуты `__class__`, `__bases__` и `__name__` позволяют полностью проанализировать структуру наследования классов, связанную с любым экземпляром.

Впрочем, две встроенные функции — `isinstance` и `issubclass` — позволяют получить большую часть обычно используемой информации в более удобном виде. Например, при помощи функции `isinstance` можно узнать, относится ли класс, переданный функции или методу, к ожидаемому типу:

```
>>> class C:
...     pass
...
>>> class D:
...     pass
...
>>> class E(D):
...     pass
```

```
...
>>> x = 12
>>> c = C()
>>> d = D()
>>> e = E()
>>> isinstance(x, E)
False
>>> isinstance(c, E) ❶
False
>>> isinstance(e, E)
True
>>> isinstance(e, D) ❷
True
>>> isinstance(d, E) ❸
False
>>> y = 12
>>> isinstance(y, type(5)) ❹
True
```

Функция `issubclass` работает только с типами-классами.

```
>>> issubclass(C, D)
False
>>> issubclass(E, D)
True
>>> issubclass(D, D) ❺
True
>>> issubclass(e.__class__, D)
True
```

Для экземпляров классов проверка осуществляется по классу ❶. `e` является экземпляром класса `D`, потому что `E` наследует от `D` ❷. Однако `d` не является экземпляром `E` ❸. Для других типов можно использовать тип-пример ❹. Класс считается субклассом самого себя ❺.

БЫСТРАЯ ПРОВЕРКА: ТИПЫ

Предположим, вы хотите убедиться в том, что объект `x` является списком, прежде чем пытаться присоединять к нему элемент. Какой код вы используете? Чем различается использование `type()` и `isinstance()`? К какому стилю программирования относится такая проверка — LBYL («Смотри, прежде чем прыгать») или EAFP («Проще просить прощения, чем разрешения»)? Какие еще возможны варианты, кроме явной проверки типа?

17.4. Утиная типизация

С помощью функций `type`, `isinstance` и `issubclass` можно относительно легко определить иерархию наследования объекта или класса. Хотя это несложно, в Python также существует механизм, с которым использовать объекты становится еще проще: *утиная типизация*. Этим термином (который происходит от поговорки «Если

оно ходит как утка и крякает как утка, то это, вероятно, и *есть* утка») обозначается используемый в Python способ определения того, относится ли объект к необходимому типу для операции; при этом основное внимание уделяется интерфейсу объекта, а не его типу. Например, если для операции нужен итератор, сам объект не обязан быть субклассом какого-то конкретного итератора, да и вообще какого-либо итератора. Все, что необходимо, — чтобы объект, используемый в качестве итератора, мог выдавать серию объектов ожидаемым образом.

С другой стороны, в таких языках, как Java, устанавливаются более жесткие правила наследования. В двух словах утиная типизация означает, что в Python вам не обязательно (да, наверное, и не следует) беспокоиться о функции проверки типа, об аргументах методов и т. д. Вместо этого следует полагаться на удобочитаемый и документированный код в сочетании с тщательным тестированием, чтобы убедиться в том, что объект «крякает как утка» там, где потребуется.

Утиная типизация может повысить гибкость хорошо написанного кода, а в сочетании с более мощными объектно-ориентированными средствами предоставляет возможность создания классов и объектов практически для любой ситуации.

17.5. Что такое специальный метод-атрибут?

Специальный метод-атрибут представляет собой атрибут класса Python, имеющий специальное значение для Python. Он определяется как метод, но не предназначен для прямого использования в качестве метода. Специальные методы не вызываются напрямую; вместо этого они вызываются автоматически Python по требованию объекта этого класса.

Возможно, простейшим примером служит специальный метод-атрибут `__str__`. Если он определен в классе, то каждый раз, когда экземпляр этого класса используется там, где Python требуется строковое представление этого экземпляра в форме, удобной для пользователя, вызывается метод-атрибут `__str__`, а возвращаемое им значение используется как требуемая строка. Чтобы видеть этот атрибут в действии, определите класс, представляющий красный, зеленый и синий цвета (RGB) в виде тройки чисел, по одному для красной, зеленой и синей составляющей. Кроме определения стандартного метода `__init__` для инициализации экземпляров класса, определите метод `__str__`, который возвращает строки, представляющие экземпляры в удобном для человека формате. Ваше определение может выглядеть примерно так:

Листинг 17.1. Файл color_module.py

```
class Color:
    def __init__(self, red, green, blue):
        self._red = red
        self._green = green
        self._blue = blue
    def __str__(self):
        return "Color: R={0:d}, G={1:d}, B={2:d}".format (self._red,
                                                self._green, self._blue)
```

Если разместить это определение в файле с именем color_module.py, вы сможете загрузить его и использовать обычным способом:

```
>>> from color_module import Color
>>> c = Color(15, 35, 3)
```

Присутствие специального метода-атрибута `__str__` проявляется тогда, когда вы попытаетесь вывести `c` функцией `print`:

```
>>> print(c)
Color: R=15, G=35, B=3
```

И хотя специальный метод-атрибут `__str__` нигде не вызывается явно в вашем коде, его использует Python, который знает, что атрибут `__str__` (если он присутствует) определяет метод для преобразования объектов в строки, понятные для пользователя. Это одна из определяющих характеристик специальных методов-атрибутов; они позволяют определять функциональность, которая подключается к Python специальным образом. Среди прочего, специальные методы-атрибуты могут использоваться для определения классов, объекты которых на синтаксическом и семантическом уровне могут работать как эквиваленты списков и словарей. Например, таким образом можно определить объекты, которые используются точно так же, как списки Python, но хранят данные в сбалансированных деревьях вместо массивов. С точки зрения программиста, такие объекты выглядят как списки, но отличаются более быстрой вставкой, медленным перебором и другими различиями в быстродействии, которые могут быть полезны для конкретной задачи.

В оставшейся части этой главы рассматриваются более длинные примеры с использованием специальных методов-атрибутов. В ней не обсуждаются все доступные специальные методы-атрибуты, но концепция представлена достаточно подробно для того, чтобы вы смогли легко использовать другие специальные методы-атрибуты, которые определяются в документации стандартной библиотеки для встроенных типов.

17.6. Поведение объекта как списка

В этой задаче используется большой текстовый файл с информацией о людях; каждая запись представляет собой одну строку с именем, возрастом и местом проживания, поля разделяются двойными двоеточиями (`::`). Несколько строк из такого файла могли бы выглядеть примерно так:

```
.
.
.
John Smith::37::Springfield, Massachusetts
Ellen Nelle::25::Springfield, Connecticut
Dale McGladdery::29::Springfield, Hawaii
.
.
.
```

Допустим, вы хотите собрать информацию о распределении возрастов людей в файле. Существует много способов обработки подобных файлов. Один из них выглядит так:

```
fileobject = open(filename, 'r')
lines = fileobject.readlines()
fileobject.close()
for line in lines:
    ... любые действия ...
```

Этот способ, теоретически, работает, но он читает в память сразу весь файл. Если бы файл был слишком большим и не поместился в памяти (а такие файлы бывают *настолько* большими), программа работать бы не стала.

Другое возможное решение:

```
fileobject = open(filename, 'r')
for line in fileobject:
    ... любые действия ...
fileobject.close()
```

Этот код решает проблему нехватки памяти, читая файл по одной строке. Он работает, но представьте, что вам хотелось бы упростить открытие файла, а из каждой строки файла вам нужны только первые два поля (имя и возраст). Нужен какой-то механизм, который мог бы (по крайней мере для цикла `for`) работать с текстовым файлом как со списком строк, но без чтения всего текстового файла сразу.

17.7. Специальный метод-атрибут __getitem__

В такой ситуации следует использовать специальный метод-атрибут `__getitem__`, который может определяться в любом пользовательском классе, чтобы экземпляры этого класса могли реагировать на синтаксис и семантику списковых операций. Если `AClass` — класс Python, определяющий `__getitem__`, а `obj` — экземпляр этого класса, то конструкции вида `x = obj[n]` и `for x in obj:` имеют смысл; `obj` может использоваться практически так же, как список.

Код выглядит так (объяснения приводятся ниже):

```
class LineReader:
    def __init__(self, filename):
        self.fileobject = open(filename, 'r')  <-- Открывает файл для чтения
    def __getitem__(self, index):
        line = self.fileobject.readline()  <-- Пытается читать файл
        if line == "":  <-- Если данных больше нет...
            self.fileobject.close()  <-- ... объект файла закрывается...
            raise IndexError  <-- и инициируется исключение IndexError

        else:
            return line.split("::")[:2]  <-- В противном случае производится разбиение строки и возвращаются первые два поля

for name, age in LineReader("filename"):
    ... любые действия ...
```

На первый взгляд этот пример выглядит хуже предыдущего решения, потому что он занимает больше места и более сложен для понимания. Но большая часть этого кода содержится в классе, который можно выделить в отдельный модуль — например, `myutils`. Тогда программа принимает вид:

```
import myutils
for name, age in myutils.LineReader("filename"):
    ... любые действия ...
```

Класс `LineReader` берет на себя все хлопоты по открытию файла, последовательному чтению строк и закрытию файла. Время исходной разработки немного увеличивается, но зато работа с большими текстовыми файлами, содержащими по одной записи на строку, упрощается и становится более надежной. В Python уже есть несколько мощных механизмов чтения файлов, но у этого примера свое преимущество: он достаточно прост для понимания. Стоит один раз понять суть происходящего, и вы сможете применять этот принцип во многих ситуациях.

17.7.1. Как это работает

`LineReader` является классом, а метод `__init__` открывает файл с заданным именем для чтения и сохраняет открытый объект файла для последующего доступа. Чтобы понять, как работает метод `__getitem__`, необходимо знать три обстоятельства:

- Любой объект, определяющий `__getitem__` как метод экземпляра, может возвращать элементы так, словно он является списком: все обращения в форме `object[i]` преобразуются Python в вызов метода вида `object.__getitem__(i)`, который реализуется как обычный вызов метода. В конечном итоге он выполняется в форме `__getitem__(object, i)`, с использованием версии `__getitem__`, определенной в классе. В первом аргументе каждого вызова `__getitem__` передается объект, из которого извлекаются данные, а во втором — индекс этих данных.
- Поскольку циклы `for` последовательно перебирают все элементы данных в списке, цикл в форме `for arg in sequence:` вызывает `__getitem__` снова и снова, последовательно увеличивая индексы. Цикл `for` сначала присваивает `arg` значение `sequence.__getitem__(0)`, затем `sequence.__getitem__(1)` и т. д.
- Цикл `for` перехватывает исключения `IndexError` и обрабатывает их, прерывая цикл. Именно так прерываются циклы `for` при использовании с обычными списками или последовательностями.

Класс `LineReader` предназначен только для использования в циклах `for`, а цикл `for` всегда генерирует вызовы с постоянным приращением индекса: `__getitem__(self, 0)`, `__getitem__(self, 1)`, `__getitem__(self, 2)` и т. д. Предыдущий код использует это обстоятельство и возвращает строки одну за другой, игнорируя аргумент `index`.

Зная все это, вы легко поймете, как объект `LineReader` эмулирует последовательность для цикла `for`. При каждой итерации цикла для объекта вызывается специальный метод-атрибут `__getitem__`; в результате объект читает следующую строку

из хранимого объекта файла и анализирует эту строку. Если строка непустая, то она возвращается. Пустая строка означает, что был достигнут конец файла; объект закрывает объект файла и инициирует исключение `IndexError`. Исключение `IndexError` перехватывается внешним циклом `for`, который при этом завершается.

Помните, что этот пример приведен только для пояснения. Обычно перебора строк файла в цикле типа `for line in fileobject:` достаточно, но этот пример показывает, как легко в Python создать объекты, которые ведут себя как списки или другие типы.

БЫСТРАЯ ПРОВЕРКА: __GETITEM__

Возможности приведенного примера использования `__getitem__` чрезвычайно ограниченны, и во многих ситуациях он работает некорректно. В каких случаях в приведенной реализации произойдет сбой или она будет работать некорректно?

17.7.2. Реализация полной функциональности списка

В приведенном примере объект класса `LineReader` ведет себя как объект списка только в той степени, что он правильно реагирует на последовательные обращения к строкам файла, из которого читаются данные. Но как расширить эту функциональность, чтобы поведение объекта `LineReader` (или других объектов) в большей степени напоминало поведение списков?

Метод `__getitem__` должен каким-то образом обрабатывать свой аргумент `index`. Так как класс `LineReader` создается прежде всего для того, чтобы предотвратить чтение большого файла в память, будет бессмысленно хранить весь файл в памяти в возвращать соответствующую строку. Пожалуй, самое умное, что можно сделать, — проверять, что каждый индекс при вызове `__getitem__` на единицу больше индекса из предыдущего вызова `__getitem__` (или равен нулю при первом вызове `__getitem__` для экземпляра `LineReader`), и выдавать ошибку в том случае, если это условие не выполняется. Такая практика гарантирует, что экземпляры `LineReader` используются только в циклах `for`, как и предполагалось.

В более широком смысле Python предоставляет несколько специальных методов-атрибутов, относящихся к поведению списков. Метод `__setitem__` определяет, что должно быть сделано при использовании объекта в синтаксическом контексте спискового присваивания, `obj[n] = val`. Другие специальные методы-атрибуты предоставляют менее очевидную функциональность списков; например, атрибут `__add__` позволяет объектам реагировать на оператор `+`, а следовательно, выполнять свою версию конкатенации списков. Для полноценной эмуляции списков необходимо определить несколько других специальных методов. Тем не менее вы можете обеспечить полную эмуляцию списков, определяя соответствующие специальные методы-атрибуты Python. В следующем разделе приведен пример, который в значительной мере приближается к полноценной эмуляции списков.

17.8. Полноценная эмуляция списков объектами

Метод `__getitem__` — один из многих специальных атрибутов Python, которые могут определяться в классе для того, чтобы наделить экземпляры этого класса специальным поведением. Чтобы увидеть, на что способны специальные методы-атрибуты, как они фактически интегрируют новую функциональность в Python, мы рассмотрим другой, более серьезный пример.

При использовании списков каждый конкретный список чаще содержит элементы только одного типа, как, например, список строк или список чисел. Некоторые языки (к числу которых относится C++) позволяют жестко обеспечивать выполнение этого ограничения. В больших программах возможность объявления списка, содержащего элементы определенного типа, помогает находить ошибки. Попытка добавить в типизованный список элемент неправильного типа приводит к выдаче сообщения об ошибке, а ошибка выявляется на более ранней стадии разработки.

Python не содержит встроенной функциональности типизованных списков, и многие программисты Python об этом не жалеют. Но если вы непременно желаете контролировать однородность списков, специальные атрибуты-методы позволяют легко создать класс с поведением типизованного списка. Ниже приведено начало такого класса (в котором широко используются встроенные типы Python и функции `isinstance` для проверки типов объектов):

```
class TypedList:
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element) ❶
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
                if not isinstance(element, self.type):
                    raise TypeError("Attempted to add an element of "
                                    "incorrect type to a typed list.")
        self.elements = initial_list[:]
```

Аргумент `example_element` определяет тип, который может храниться в списке. Для этого он определяет пример типа элемента ❶.

Класс `TypedList` в том виде, в каком он определен здесь, дает возможность использовать вызовы в форме

```
x = TypedList ('Hello', ["List", "of", "strings"])
```

Первый аргумент `'Hello'` вообще не встраивается в итоговую структуру данных. Он всего лишь является примером типа элементов, которые должны содержаться в списке (строки в данном случае). Второй аргумент содержит необязательный список, который может использоваться для инициализации значений. Функция `__init__` для класса `TypedList` проверяет, что все элементы списка, переданного при создании экземпляра `TypedList`, относятся к тому же типу, что и значение, переданное в качестве примера. При любых несоответствиях типов инициируется исключение.

Эта версия `TypedList` не может использоваться как список, потому что она не реагирует на стандартные способы присваивания или чтения элементов списка. Чтобы решить эту проблему, необходимо определить специальные методы-атрибуты `__setitem__` и `__getitem__`. Метод `__setitem__` автоматически вызывается Python каждый раз, когда выполняется команда вида `TypedListInstance[i] = `*значение*, а метод `__getitem__` вызывается каждый раз, когда вычисляется выражение `TypedListInstance[i]` для получения значения `i`-го элемента `TypedListInstance`. Ниже приведена следующая версия класса `TypedList`. Так как мы выполняем проверку типа многих новых элементов, эта функция абстрагируется в новый приватный метод `__check`:

```
class TypedList:
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)
        self.elements = initial_list[:]

    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")

    def __setitem__(self, i, element):
        self.__check(element)
        self.elements[i] = element
    def __getitem__(self, i):
        return self.elements[i]
```

Теперь экземпляры класса `TypedList` больше похожи на списки. Например, следующий код является действительным:

```
>>> x = TypedList("", 5 * [""])
>>> x[2] = "Hello"
>>> x[3] = "There"
>>> print(x[2] + ' ' + x[3])
Hello There
>>> a, b, c, d, e = x
>>> a, b, c, d
('', '', 'Hello', 'There')
```

Обращения к элементам `x` в команде `print` обрабатываются методом `__getitem__`, который передает их экземпляру списка, хранящемуся в объекте `TypedList`. Присваивания `x[2]` и `x[3]` обрабатываются методом `__setitem__`, который проверяет, что присваиваемый элемент относится к правильному типу, после чего выполняет присваивание в списке, содержащемся в `self.elements`. Последняя строка использует `__getitem__` для распаковки первых пяти элементов в `x` и их последующей упаковки

в переменные a, b, c, d и e соответственно. Вызовы `__getitem__` и `__setitem__` Python выполняет автоматически.

Для завершения класса `TypedList`, чтобы объекты `TypedList` вели себя как объекты списков во всех отношениях, потребуется еще больше кода. Необходимо определить специальные методы-атрибуты `__setitem__` и `__getitem__`, чтобы экземпляры `TypedList` поддерживали синтаксис сегментов, а также обращения к отдельным элементам. Необходимо определить метод `__add__` для выполнения сложения списков (конкатенации), а также метод `__mul__` для умножения списков. Необходимо определить метод `__len__`, чтобы результат вызова `len(TypedListInstance)` вычислялся правильно. Необходимо определить метод `__delitem__` , чтобы класс `TypedList` правильно обрабатывал команды `del`. Кроме того, необходимо определить метод `append`, чтобы элементы могли присоединяться к экземплярам `TypedList` посредством стандартных списковых методов `append`, а также `insert` и `extend`.

ПОПРОБУЙТЕ САМИ: РЕАЛИЗАЦИЯ СПЕЦИАЛЬНЫХ МЕТОДОВ СПИСКОВ

Попробуйте реализовать специальные методы `__len__` и `__delitem__`, а также метод `append`.

17.9. Субклассирование встроенных типов

Предыдущий пример помогает понять, как реализовать класс с поведением списка с нуля, но он также требует больших усилий. На практике, если вы планируете реализовать собственную структуру с поведением списка в соответствии с приведенными рекомендациями, рассмотрите возможность субклассирования типа списка или типа `UserList`.

17.9.1. Субклассирование списка

Вместо того чтобы создавать класс типизованного списка с нуля, как это делалось в предыдущих примерах, вы можете субклассировать тип списка и переопределить все методы, которые должны отслеживать разрешенный тип. Большое преимущество такого подхода заключается в том, что ваш класс сразу будет содержать стандартные версии всех списковых операций, потому что он уже является списком. Главное, о чем следует помнить, — что каждый тип в Python является классом, а если вам потребуется изменить поведение встроенного типа, рассмотрите возможность субклассирования этого типа:

```
class TypedListList(list):
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
```

```
                            "be a list.")
        for element in initial_list:
            self.__check(element)
        super().__init__(initial_list)

    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")
    def __setitem__(self, i, element):
        self.__check(element)
        super().__setitem__(i, element)

>>> x = TypedListList("", 5 * [""])
>>> x[2] = "Hello"
>>> x[3] = "There"
>>> print(x[2] + ' ' + x[3])
Hello There
>>> a, b, c, d, e = x
>>> a, b, c, d
('', '', 'Hello', 'There')
>>> x[:]
['', '', 'Hello', 'There', '']
>>> del x[2]
>>> x[:]
['', '', 'There', '']
>>> x.sort()
>>> x[:]
['', '', '', 'There']
```

Теперь остается совсем немного: реализовать метод для проверки типа добавляемых элементов, а затем изменить `__setitem__` для того, чтобы эта проверка производилась перед вызовом обычного метода `__setitem__` списка. Другие методы, такие как `sort` и `del`, работают без дополнительного кодирования. Переопределение встроенного типа сэкономит вам немало времени, если вы хотите внести несколько второстепенных поправок в поведение, а основная часть класса остается неизменной.

17.9.2. Субклассирование UserList

Если вам нужна модифицированная версия списка (как в предыдущих примерах), есть третий вариант: вы можете субклассировать класс `UserList`, класс-обертку списка из модуля `collections`. Класс `UserList` создавался для предыдущих версий Python, где субклассирование типа списка было невозможно, однако он остается полезным, особенно в текущей ситуации, потому что используемый список доступен в атрибуте `data`:

```
from collections import UserList
class TypedUserList(UserList):
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
```

```
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)
        super().__init__(initial_list)

    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")
    def __setitem__(self, i, element):
        self.__check(element)
        self.data[i] = element
    def __getitem__(self, i):
        return self.data[i]

>>> x = TypedUserList("", 5 * [""])
>>> x[2] = "Hello"
>>> x[3] = "There"
>>> print(x[2] + ' ' + x[3])
Hello There
>>> a, b, c, d, e = x
>>> a, b, c, d
('', '', 'Hello', 'There')
>>> x[:]
['', '', 'Hello', 'There', '']
>>> del x[2]
>>> x[:]
['', '', 'There', '']
>>> x.sort()
>>> x[:]
['', '', '', 'There']
```

Пример выглядит почти так же, как при субклассировании списка, если не считать того, что в реализации класса список элементов доступен в поле `data`. В некоторых ситуациях прямой доступ к нижележащей структуре данных может быть полезен. Кроме того, наряду с `UserList` имеются классы-обертки `UserDict` и `UserString`.

17.10. Когда используются специальные методы-атрибуты

Как правило, к использованию специальных методов-атрибутов следует подходить осторожно. Другим программистам, которым придется работать с вашим кодом, может быть непонятно, почему один объект последовательности правильно реагирует на стандартный синтаксис индексирования, а другой нет.

Я считаю, что специальные методы-атрибуты следует применять в двух ситуациях:

- Если в вашем коде имеется часто используемый класс, который в каких-то отношениях ведет себя как встроенный тип Python, такие специальные методы-

атрибуты могут быть полезны. Такая ситуация чаще всего встречается с объектами, которые ведут себя как последовательности в том или ином отношении.

- Если имеется класс, поведение которого идентично (или почти идентично) поведению встроенного класса, можно выбрать между определением всех необходимых специальных функций-атрибутов или субклассированием встроенного класса Python. Пример такого решения — списки, реализованные в форме сбалансированных деревьев; такие списки уступают стандартным по времени доступа, но превосходят их по скорости вставки.

Не стоит считать эти правила непреложными. Например, часто бывает полезно определить для класса специальный метод-атрибут `__str__`, чтобы вы могли включить в отладочный код команду `print(экземпляр)` и получить на экране содержательное, удобное представление вашего объекта в текстовой форме.

БЫСТРАЯ ПРОВЕРКА: СПЕЦИАЛЬНЫЕ МЕТОДЫ-АТРИБУТЫ И СУБКЛАССИРОВАНИЕ СУЩЕСТВУЮЩИХ ТИПОВ

Допустим, вам нужен тип, похожий на словарь, который разрешает использовать в качестве ключей только строки (возможно, для использования по аналогии с объектом `shelf` — глава 13). Какие существуют варианты создания такого класса? Какими достоинствами и недостатками обладает каждый вариант?

Итоги

- Python предоставляет средства для проверки типов в коде на случай необходимости, но благодаря утиной типизации вы можете писать более гибкий код, в котором проверка типов уже не столь важна.
- Специальные методы-атрибуты и субклассирование встроенных классов могут использоваться для добавления поведения, характерного для списков, в классы, создаваемые пользователем.
- Использование утиной типизации в Python, специальные методы-атрибуты и субклассирование позволяют строить разные классы и объединять их различными способами.

18 Пакеты

Эта глава охватывает соедующие темы:

- ✓ Определение пакета
- ✓ Создание простого пакета
- ✓ Изучение конкретного примера
- ✓ Использование атрибута `__all__`
- ✓ Правильное использование пакетов

Модули упрощают повторное использование мелких фрагментов кода. Проблема возникает тогда, когда проект разрастается, а загружаемый код выходит за границы (физические или логические) того, что может поместиться в один файл. Если один гигантский файл модуля вам не подходит, то множество мелких разрозненных модулей не намного лучше. Проблема решается объединением логически связанных модулей в пакет.

18.1. Что такое пакет?

Модуль представляет собой файл с программным кодом. Модуль определяет группу взаимосвязанных функций Python или других объектов. Имя модуля определяется именем файла.

Если вы понимаете, как работают модули, разобраться с пакетами должно быть несложно, потому что пакет представляет собой каталог с программным кодом и, возможно, несколькими подкаталогами. Пакет содержит группу (обычно) логически связанных файлов с кодом (модулей). Имя пакета определяется именем главного каталога пакета.

Пакеты являются естественным расширением концепции модуля; они предназначены для очень больших проектов. По аналогии с тем, как модули группируют взаимосвязанные функции, классы и переменные, пакеты группируют взаимосвязанные модули.

18.2. Первый пример

Чтобы понять, как пакеты применяются на практике, рассмотрим структуру типов в проекте, который по своей природе очень велик: универсальный математический пакет, построенный по образцу Mathematica, Maple или MATLAB. Maple, например, состоит из тысяч файлов, и иерархическая структура жизненно необходима для упорядочения проекта. Назовем этот проект `mathproj`.

Существуют разные варианты организации таких проектов, но в одной из разумных структур проект разбивается на две части: `ui` (элементы пользовательского интерфейса) и `comp` (вычислительные элементы). В `comp` вычислительные аспекты стоит дополнительно разбить на *символические* (вещественные и комплексные символические вычисления, как в школьном курсе алгебры) и *числовые* (вещественные и комплексные числовые расчеты — например, численное интегрирование). Как в *символической*, так и в *числовой* части должен присутствовать свой файл constants.py.

В файле constants.py из числовой части проекта значение `pi` определяется следующим образом:

```
pi = 3.141592
```

тогда как в файле constants.py в числовой части проекта определение `pi` выглядит так:

```
class PiClass:
    def __str__(self):
        return "PI"
pi = PiClass()
```

Это означает, что такое имя, как `pi`, может использоваться в двух разных файлах (и импортироваться из них) с именами constants.py, как показано на рис. 18.1.

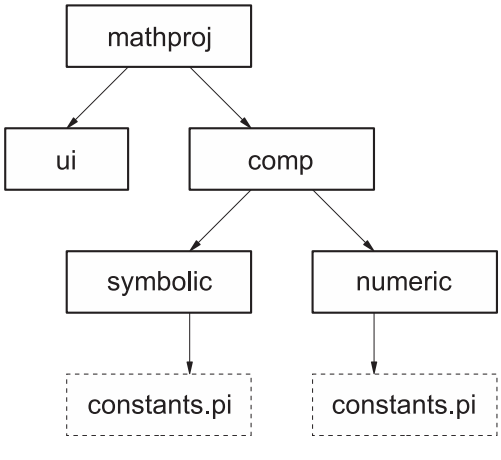


Рис. 18.1. Структура математического пакета

Символический файл constants.py определяет `pi` как абстрактный объект Python, единственный экземпляр класса `PiClass`. По мере развития системы в этом классе могут реализовываться различные операции, которые возвращают символические результаты вместо числовых.

Существует естественное соответствие между структурой проекта и структурой каталогов. Каталог верхнего уровня проекта с именем mathproj содержит подкаталоги ui и comp; в свою очередь, comp содержит подкаталоги symbolic и numeric; наконец, каждый из подкаталогов symbolic и numeric содержит собственный файл constants.pi.

С такой структурой каталогов и в предположении, что корневой каталог mathproj установлен где-то в пути поиска Python, код Python в проекте `mathproj` и вне его может обращаться к двум версиям `pi` по именам mathproj.symbolic.constants.pi и mathproj.numeric.constants.pi. Иначе говоря, имя Python для элемента пакета отражает путь к файлу, содержащему этот элемент.

Для этого и нужны пакеты. Они предоставляют средства для организации очень больших объемов кода Python в единое целое. Таким образом, код распределяется между разными файлами и каталогами, а в проекте устанавливается схема назначения имен модулей/субмодулей на основании структуры каталогов файлов пакетов. К сожалению, на практике с пакетами не все так просто; из-за разных нюансов пользоваться ими сложнее, чем в теории. Практическим аспектам использования пакетов посвящена оставшаяся часть этой главы.

18.3. Конкретный пример

В оставшейся части этой главы для демонстрации внутренних правил работы механизма пакетов будет использоваться пример (рис. 18.2). Файлы, которые будут использоваться в примере, приведены в листингах 18.1–18.6.

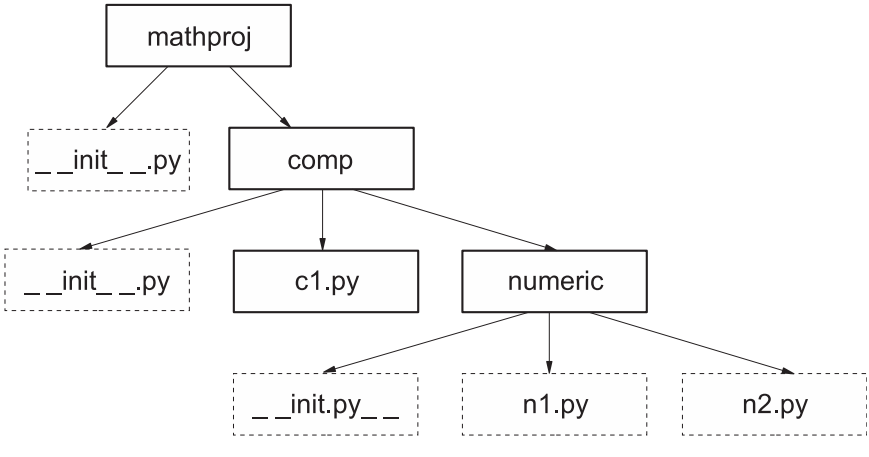


Рис. 18.2. Пример пакета

Листинг 18.1. Файл mathproj/__init__.py

```
print("Hello from mathproj init")
__all__ = ['comp']
version = 1.03
```

Листинг 18.2. Файл mathproj/comp/__init__.py

```
__all__ = ['c1']
print("Hello from mathproj.comp init")
```

Листинг 18.3. Файл mathproj/comp/c1.py

```
x = 1.00
```

Листинг 18.4. Файл mathproj/comp/numeric/__init__.py

```
print("Hello from numeric init")
```

Листинг 18.5. Файл mathproj/comp/numeric/n1.py

```
from mathproj import version
from mathproj.comp import c1
from mathproj.comp.numeric.n2 import h
def g():
    print("version is", version)
    print(h())
```

Листинг 18.6. Файл mathproj/comp/numeric/n2.py

```
def h():
    return "Called function h in module n2"
```

В рамках примеров этой главы будем считать, что эти файлы были созданы в каталоге mathproj, входящем в путь поиска Python. (Достаточно проследить за тем, чтобы при выполнении этих примеров текущим рабочим каталогом Python был каталог, содержащий проект `mathproj`.)

ПРИМЕЧАНИЕ

В большинстве примеров этой книги необязательно запускать новую оболочку Python для каждого примера. Обычно можно выполнить примеры в оболочке Python, использованной в предыдущих примерах, и все равно получить показанные результаты. Тем не менее *это не относится к примерам данной главы*, потому что для правильной работы примеров пространство имен Python должно быть чистым (не измененным предыдущими командами `import`). Если вы будете запускать следующие примеры, *проследите за тем, чтобы каждый пример запускался в отдельном сеансе*. В IDLE для этого приходится выходить из программы и перезапускать ее; простого закрытия и повторного открытия окна Shell недостаточно.

18.3.1. Файлы __init__.py в пакетах

Возможно, вы заметили, что во всех каталогах вашего пакета — mathproj, mathproj/comp и mathproj/numeric — присутствует файл с именем __init__.py. Файл __init__.py решает две задачи:

- Python требует, чтобы каталог содержал файл __init__.py, прежде чем он может быть распознан пакетом. Это требование предотвращает случайное импортирование каталогов, содержащих посторонний код Python, в качестве пакетов.
- Файл __init__.py автоматически выполняется Python при первой загрузке пакета или субпакета. При этом может быть выполнена любая инициализация пакета, которую вы сочтете необходимой.

Первый пункт обычно важнее. Для многих пакетов в файл __init__.py не нужно включать никакой код; просто проследите за тем, чтобы в каталоге присутствовал пустой файл __init__.py.

18.3.2. Простейшее использование пакета mathproj

Прежде чем переходить к подробностям использования пакетов, посмотрим, как обратиться к элементам из пакета `mathproj`. Запустите новую оболочку Python и выполните следующие команды:

```
>>> import mathproj
Hello from mathproj init
```

Если все пройдет хорошо, появится новое приглашение без сообщений об ошибках. Кроме того, код из файла mathproj/__init__.py должен вывести сообщение `"Hello from mathproj init"`. Вскоре мы поговорим о файлах __init__.py подробнее, а пока достаточно знать, что эти файлы автоматически выполняются при первой загрузке пакета.

Файл mathproj/__init__.py присваивает переменной `version` значение 1.03. Переменная `version` находится в области видимости пространства имен пакета `mathproj`, а после создания вы можете обратиться к ней через `mathproj` даже за пределами файла mathproj/__init__.py:

```
>>> mathproj.version
1.03
```

При использовании пакеты очень похожи на модули; они могут предоставить доступ к определенным в них объектам через атрибуты. И это вполне естественно, потому что пакеты являются обобщенной формой модулей.

18.3.3. Загрузка субпакетов и субмодулей

А теперь посмотрим, как различные файлы, определенные в пакете `mathproj`, взаимодействуют друг с другом. Для этого попробуем вызвать функцию g, определенную в файле mathproj/comp/numeric/n1.py. Первый очевидный вопрос: был ли загружен этот модуль? Вы уже загрузили `mathproj`, но как насчет его субпакета? Чтобы понять, знает ли о нем Python, введите команду

```
>>> mathproj.comp.numeric.n1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'mathproj' has no attribute 'comp'
```

Иначе говоря, загрузки модуля верхнего уровня пакета недостаточно для того, чтобы загрузить все его субмодули, и это соответствует философии Python, согласно которой ничего не должно происходить у вас за спиной. Ясность важнее лаконичности.

Это ограничение достаточно легко преодолевается. Импортируйте нужный модуль и выполните функцию g из этого модуля:

```
>>> import mathproj.comp.numeric.n1
Hello from mathproj.comp init
Hello from numeric init
>>> mathproj.comp.numeric.n1.g()
version is 1.03
Called function h in module n2
```

Однако обратите внимание на побочный эффект загрузки mathproj.comp.numeric.n1: вывод строк, начинающихся с Hello. Эти две строки выводятся командами print в файлах __init__.py из каталогов mathproj/comp и mathproj/comp/numeric. Другими словами, прежде чем Python сможет импортировать mathproj.comp.numeric.n1, он должен импортировать mathproj.comp, а затем mathproj.comp.numeric. Каждый раз, когда пакет импортируется впервые, выполняется связанный с ним файл __init__.py; отсюда и появление строк Hello. Чтобы убедиться в том, что в процессе импортирования mathproj.comp.numeric.n1 импортируются mathproj.comp и mathproj.comp.numeric, можно проверить, что сеанс теперь знает о mathproj.comp и mathproj.comp.numeric:

```
>>> mathproj.comp
<module 'mathproj.comp' from 'mathproj/comp/__init__.py'>
>>> mathproj.comp.numeric
<module 'mathproj.comp.numeric'  from 'mathproj/comp/numeric/__init__.py'>
```

18.3.4. Команды import с пакетами

Файлы в пакете не получают автоматического доступа к объектам, определенным в других файлах того же пакета. Как и во внешних модулях, для явного обращения к объектам из других файлов пакетов необходимо использовать команды import. Чтобы увидеть, как работает импортирование на практике, вернемся к субпакету n1. Файл n1.py содержит следующий код:

```
from mathproj import version
from mathproj.comp import c1
from mathproj.comp.numeric.n2 import h
def g():
    print("version is", version)
    print(h())
```

В функции g используется как переменная version из пакета mathproj верхнего уровня, так и функция h из модуля n2, а следовательно, модуль, содержащий g, должен импортировать version и h, чтобы получить доступ к ним. Переменная version импортируется так же, как при импортировании за пределами mathproj, — командой

`from mathproj import version`. В этом примере функция `h` явно импортируется командой `from mathproj.comp.numeric.n2 import h`, причем этот прием работает в любом файле; явное импортирование файлов пакетов разрешено всегда. Но поскольку n2.py находится в одном каталоге с n1.py, также можно воспользоваться относительным импортированием, поставив точку перед именем субмодуля. Другими словами, в третью строку n1.py можно включить команду

```
from .n2 import h
```

и все будет работать нормально.

Вы можете добавить больше точек, чтобы подняться на несколько уровней в иерархии пакетов, и добавить имена модулей. Вместо

```
from mathproj import version
from mathproj.comp import c1
from mathproj.comp.numeric.n2 import h
```

команды импортирования n1.py можно было записать в следующем виде:

```
from ... import version
from ..  import c1
from . n2 import h
```

Команды относительного импортирования удобны и быстро вводятся, но помните, что они задаются *относительно* свойства `__name__` модуля. А значит, любой модуль, выполняемый в качестве основного (у которого `__name__` содержит `__main__`), не может использовать относительное импортирование.

18.4. Атрибут __all__

Вернувшись к разным файлам __init__.py, определенным в `mathproj`, вы заметите, что некоторые из них определяют атрибут с именем `__all__`. Этот атрибут имеет отношение к выполнению команд в форме `from ... import *` и заслуживает более подробного объяснения.

Казалось бы, если внешний код выполняет команду `from mathproj import *`, он должен импортировать все неприватные имена из `mathproj`. На практике все не так просто. Главная проблема заключается в том, что в некоторых операционных системах существует неоднозначное определение регистра символов в именах файлов. Так как объекты в пакетах могут определяться по файлам или каталогам, появляется неоднозначность в отношении точного имени для импортирования субпакета. Если выполняется команда `from mathproj import *`, в каком виде будет импортироваться `comp` — `comp`, `Comp` или `COMP`? Если полагаться только на имя, полученное от операционной системы, результаты могут быть непредсказуемыми.

У этой проблемы нет хорошего решения, поскольку она обусловлена неудачной архитектурой ОС. Впрочем, атрибут `__all__` дает решение хотя бы приемлемое. Если атрибут `__all__` присутствует в файле __init__.py, он возвращает список строк,

определяющий те имена, которые должны импортироваться при выполнении команды `from ... import *` для этого конкретного пакета. Если атрибут `__all__` отсутствует, команда `from ... import *` для заданного пакета не делает ничего. Так как регистр символов в текстовом файле всегда учитывается, имена, под которыми импортируются объекты, становятся однозначными, и если операционная система полагает, что `comp` и `COMP` — одно и то же, это ее проблема.

Снова запустите Python и выполните следующую команду:

```
>>> from mathproj import *
Hello from mathproj init
Hello from mathproj.comp init
```

Атрибут `__all__` в файле mathproj/__init__.py содержит одну запись `comp`, поэтому команда `import` импортирует только `comp`. Достаточно легко проверить, знает ли теперь сеанс Python о существовании `comp`:

```
>>> comp
<module 'mathproj.comp' from 'mathproj/comp/__init__.py'>
```

Учтите, что рекурсивное импортирование имен командой `from ... import *` не поддерживается. Атрибут `__all__` для пакета `comp` содержит `c1`, но имя `c1` не будет автоматически загружено командой `from mathproj import *`:

```
>>> c1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c1' is not defined
```

Чтобы вставить имена из `mathproj.comp`, придется снова выполнить явное импортирование:

```
>>> from mathproj.comp import c1
>>> c1
<module 'mathproj.comp.c1' from 'mathproj/comp/c1.py'>
```

18.5. Правильное использование пакетов

Обычно ваши пакеты не будут иметь такую сложную структуру, как в наших примерах. Механизм пакетов обеспечивает значительное разнообразие сложности и уровня вложенности при проектировании пакетов. Конечно, вы *можете* строить очень сложные пакеты, но не так очевидно, *стоит ли* это сделать.

Приведу пару рекомендаций, уместных в большинстве случаев:

- Пакеты не должны использовать структуры каталогов с большой глубиной вложенности. Если не считать действительно огромных библиотек кода, делать это не нужно. Для большинства пакетов достаточно одного каталога верхнего уровня. Двухуровневая иерархия справится почти со всеми оставшимися случаями, за очень редкими исключениями. Как пишет Тим Петерс (Tim Peters), «плоское лучше, чем вложенное» (приложение А).

- Хотя атрибут `__all__` может использоваться для сокрытия имен из `from ... import *`, вероятно, это *не* лучшее решение из-за его непоследовательности. Если вы хотите скрыть имена, объявите их приватными при помощи префикса `_`.

БЫСТРАЯ ПРОВЕРКА: ПАКЕТЫ

Допустим, вы пишете пакет, который получает URL-адрес, загружает все изображения со страницы, представленной URL-адресом, масштабирует их до стандартного размера и сохраняет их. Оставляя в стороне подробности реализации каждой из этих функций, как бы вы организовали эти функциональные возможности в пакет?

ПРАКТИЧЕСКАЯ РАБОТА 18: СОЗДАНИЕ ПАКЕТА

В главе 14 вы добавили обработку ошибок в модуль для чистки текста и подсчета слов, созданный в главе 11. Проведите рефакторинг кода и создайте пакет, который содержит один модуль для функций очистки, другой — для функций обработки и третий — для нестандартных исключений. Затем напишите простую функцию, которая использует все три модуля.

Итоги

- Пакеты позволяют создавать библиотеки кода, содержащие много файлов и каталогов.
- Пакеты позволяют структурировать большие объемы кода лучше, чем это можно сделать с помощью одиночных модулей.
- Остерегайтесь вложения каталогов в пакеты более чем на один-два уровня, если только вы не создаете очень большую и сложную библиотеку.

19 Использование библиотек Python

Эта глава охватывает следующие темы:

- ✓ Управление различными типами данных — строками, числами и т. д.
- ✓ Управление файлами и хранилищем
- ✓ Доступ к сервисам операционной системы
- ✓ Использование интернет-протоколов и форматов
- ✓ Разработка и отладка инструментов
- ✓ Доступ к PyPI (CheeseShop)
- ✓ Установка библиотек Python и виртуальных сред с помощью `pip` и `venv`

Создатели Python давно провозгласили, что одним из ключевых преимуществ языка является его философия «батарейки в комплекте». Это означает, что в базовую установку Python входит полнофункциональная стандартная библиотека, которая позволяет справиться с широким спектром ситуаций без необходимости установки дополнительных библиотек. В этой главе представлен высокоуровневый обзор части контента стандартной библиотеки, а также некоторые рекомендации по поиску и установке внешних модулей.

19.1. «Батарейки в комплекте»: стандартная библиотека

То, что считается *библиотекой* в Python, состоит из нескольких компонентов, включающих встроенные типы данных и константы, которые могут использоваться без команды `import`, как, например, числа и списки, некоторые встроенные функции и исключения. Самой большой частью библиотеки является обширная подборка модулей. Если вы установили Python на своем компьютере, у вас также имеются библиотеки для работы с разными типами данных и файлов, для взаимодействия

с операционной системой, для написания серверов и клиентов для многих протоколов интернета, разработки и отладки вашего кода.

Далее приводится краткий обзор наиболее важных моментов. Хотя в тексте упоминаются многие важные модули, чтобы получить самую полную и актуальную информацию, я рекомендую заняться самостоятельным исследованием справочного описания библиотеки, которое является частью документации Python. В частности, прежде чем браться за поиски внешних библиотек, обязательно ознакомьтесь с теми возможностями, которые уже предлагает Python. Вы не представляете, сколько всего здесь можно найти.

19.1.1. Управление различными типами данных

Естественно, стандартная библиотека содержит поддержку встроенных типов Python, которые будут рассмотрены в этом разделе. Кроме того, в стандартную библиотеку входят три категории, предназначенные для работы с различными типами данных: работа со строками, типы данных и числовые модули.

В табл. 19.1 перечислены модули из категории работы со строками — как с последовательностями байтов, так и традиционными строками. Эти модули предназначены для работы со строками и текстом, последовательностями байтов и выполнения операций Юникода.

Таблица 19.1. Модули для работы со строками

Модуль	Описание и возможное использование
`string`	Сравнение со строковыми константами (например, `digits` или `whitespace`); форматные строки
`re`	Поиск и замена текста с использованием регулярных выражений (глава 16)
`struct`	Интерпретация байтов как упакованных двоичных данных, чтение и запись структурированных данных в файлы
`difflib`	Использование вспомогательных средств для вычисления расхождений, поиск различий между строками или последовательностями, создание файлов различий и исправлений
`textwrap`	Перенос и заполнение текста, форматирование текста посредством разбиения строк или добавления пробелов

Категория типов данных содержит разнообразную подборку типов данных, включая типы для времени, даты и коллекций (табл. 19.2).

Как подсказывает название, числовые и математические модули предназначены для работы с числами и выполнения математических операций; самые популярные из этих модулей перечислены в табл. 19.3. Эти модули содержат все необходимое для создания собственных числовых типов и реализации широкого спектра математических операций.

Таблица 19.2. Модули типов данных

Модуль	Описание и возможное использование
`datetime, calendar`	Работа с датой, временем и календарем
`collections`	Контейнерные типы данных
`enum`	Создание классов перечислений, связывающих символические имена с постоянными значениями
`array`	Эффективная реализация массивов с числовыми элементами
`sched`	Планирование событий
`queue`	Класс синхронизированной очереди
`copy`	Операции глубокого и поверхностного копирования данных
`pprint`	Структурированная печать данных
`typing`	Поддержка пометки кода рекомендациями относительно типа объектов, особенно параметров функций и возвращаемых значений

Таблица 19.3. Числовые и математические модули

Модуль	Описание и возможное использование
`numbers`	Числовые абстрактные базовые классы
`math, cmath`	Математические функции для работы с вещественными и комплексными числами
`decimal`	Десятичные операции с фиксированной и вещественной точкой
`statistics`	Функции для вычисления математической статистики
`fractions`	Рациональные числа
`random`	Генерирование псевдослучайных чисел и вариантов, случайные перестановки последовательностей
`itertools`	Функции, возвращающие итераторы для эффективного перебора
`functools`	Функции более высокого порядка и операции с вызываемыми объектами
`operator`	Стандартные операторы как функции

19.1.2. Работа с файлами и хранение данных

Другую широкую категорию в стандартной библиотеке составляют модули для работы с файлами и хранения данных (табл. 19.4). В эту категорию входят модули для разных задач, от работы с файлами до долгосрочного хранения данных, сжатия и работы со специальными форматами файлов.

Таблица 19.4. Модули для работы с файлами и хранения данных

Модуль	Описание и возможное использование
`os.path`	Стандартные манипуляции с полными именами файлов
`pathlib`	Объектно-ориентированная работа с полными именами файлов
`fileinput`	Перебор строк из разных входных потоков
`filecmp`	Сравнение файлов и каталогов
`tempfile`	Генерирование временных файлов и каталогов
`glob, fnmatch`	Работа с путями и именами файлов с использованием шаблонов в стиле UNIX
`linecache`	Произвольный доступ к строкам текста
`shutil`	Высокоуровневые операции с файлами
`pickle, shelve`	Сериализация и хранение объектов Python
`sqlite3`	Работа с интерфейсом DB-API 2.0 к базам данных SQL
`zlib, gzip, bz2, zipfile, tarfile`	Работа с архивными файлами и сжатием
`csv`	Чтение и запись файлов в формате CSV
`configparser`	Разбор конфигурационных файлов; чтение/запись конфигурационных файлов .ini в стиле Windows

19.1.3. Сервисные функции операционной системы

Еще одна широкая категория содержит модули для работы с операционной системой. Как показано в табл. 19.5, в нее включены инструменты для работы с параметрами командной строки, перенаправления ввода/вывода, записи в файлы журналов, запуска нескольких потоков или процессов и загрузки библиотек других языков (обычно языка C) для использования в Python.

Таблица 19.5. Модули операционной системы

Модуль	Описание
`os`	Различные интерфейсы операционной системы
`io`	Основные средства для работы с потоками
`time`	Работа со временем и преобразования
`optparse`	Мощный парсер параметров командной строки
`logging`	Средства ведения журнала
`getpass`	Портируемые средства ввода пароля
`curses`	Поддержка терминала для символьного вывода
`platform`	Работа с идентификационными данными платформы
`ctypes`	Библиотека для работы с внешними функциями в Python
`select`	Ожидание завершения ввода/вывода

Модуль	Описание
`threading`	Высокоуровневый интерфейс программных потоков
`multiprocessing`	Интерфейс программных потоков уровня процессов
`subprocess`	Управление подпроцессами

19.1.4. Интернет-протоколы и форматы

Категория инернет-протоколов и форматов относится к кодированию и декодированию многих стандартных форматов, применяемых для передачи данных в интернете, от MIME и других кодировок до JSON и XML. В категорию также входят модули для написания серверов и клиентов для стандартных сервисов (прежде всего HTTP) и обобщенного сервера сокетов, используемого для написания серверов нестандартных служб. Наиболее часто используемые модули перечислены в табл. 19.6.

Таблица 19.6. Модули поддержки интернет-протоколов и форматов

Модуль	Описание
`socket, ssl`	Низкоуровневые сетевые интерфейсы и обертка SSL для объектов сокетов
`email`	Пакет для работы с электронной почтой и MIME
`json`	Кодирование и декодирование JSON
`mailbox`	Работа с почтовыми ящиками в разных форматах
`mimetypes`	Отображение имен файлов на типы MIME
`base64, binhex, binascii, quopri, uu`	Кодирование/декодирование файлов или потоков с различными кодировками
`html.parser, html.entities`	Разбор HTML и XHTML
`xml.parsers.expat, xml.dom, xml.sax, xml.etree.ElementTree`	Разные парсеры и инструменты для XML
`cgi, cgitb`	Поддержка CGI (Common Gateway Interface)
`wsgiref`	Служебные средства WSGI и эталонная реализация
`urllib.request, urllib.parse`	Открытие и разбор URL-адресов
`ftplib, poplib, imaplib, nntplib, smtplib, telnetlib`	Клиенты для различных интернет-протоколов
`socketserver`	Инфраструктура для сетевых серверов
`http.server`	Серверы HTTP
`xmlrpc.client, xmlrpc.server`	Клиент и сервер XML-RPC

19.1.5. Средства разработки и отладки, сервисные функции времени выполнения

Python содержит ряд модулей, упрощающих отладку, тестирование, изменение и прочие взаимодействия с кодом Python во время выполнения. Как показано в табл. 19.7, эта категория включает два инструмента тестирования, профилировщики, модули для работы с трассировкой ошибок, средства уборки мусора и т. д., а также модули, позволяющие настроить процесс импортирования других модулей.

Таблица 19.7. Разработка, отладка и модули времени выполнения

Модуль	Описание
`pydoc`	Генератор документации и электронной справки
`doctest`	Тестирование интерактивных примеров Python
`unittest`	Инфраструктура модульного тестирования
`test.support`	Служебные функции для тестов
`pdb`	Отладчик Python
`profile, cProfile`	Профилировщики Python
`timeit`	Изменение времени выполнения небольших фрагментов кода
`trace`	Трассировка выполнения команд Python
`sys`	Системные параметры и функции
`atexit`	Обработчики выхода
`__future__`	Определения будущих команд — возможностей, которые будут добавлены в Python
`gc`	Интерфейс к уборщику мусора
`inspect`	Инспектирование существующих объектов
`imp`	Обращение к внутренней реализации импортирования
`zipimport`	Импортирование модулей из zip-архивов
`modulefinder`	Поиск модулей, используемых сценарием

19.2. За пределами стандартной библиотеки

Хотя принятая в Python философия «батарейки в комплекте» и богатая стандартная библиотека означают, что с исходной установкой Python можно сделать очень много, но со временем неизбежно возникнет ситуация, в которой вам понадобится функциональность, отсутствующая в Python. В этом разделе рассматриваются возможные варианты действий в тех случаях, когда вам нужно сделать нечто такое, что отсутствует в стандартной библиотеке.

19.3. Установка дополнительных библиотек Python

Чтобы найти нужный пакет или модуль Python, иногда бывает достаточно ввести описание искомой функциональности (скажем, mp3 tags и Python) в поисковую систему, а потом отсортировать результаты. Если повезет, вам может попасться нужный модуль упакованной для вашей ОС в виде исполняемого файла Windows установочной программы macOS или пакета для вашего дистрибутива Linux.

Это один из простейших способов добавления библиотек в установку Python, потому что установочная программа или менеджер пакетов берут на себя все подробности корректного добавления модуля в систему. Данный способ также может решить проблему установки более сложных библиотек — например, научных библиотек со сложными требованиями к сборке и множеством зависимостей.

В общем случае, если не считать научных библиотек, заранее построенные пакеты нетипичны для программных продуктов Python. Такие пакеты часто оказываются немного устаревшими, они обладают меньшей гибкостью в отношении того, где и как они должны устанавливаться.

19.4. Установка библиотек Python с использованием pip и venv

Если вам понадобится сторонний модуль, который не был заранее упакован для вашей платформы, придется обращаться к исходному коду. При этом возникает пара проблем:

- Чтобы установить модуль, вы должны найти и загрузить его.
- Правильная установка даже одного модуля Python потребует изрядных хлопот по настройке путей Python и разрешений вашей системы, поэтому стандартная система установки может быть полезной.

Для решения обеих проблем в Python существует менеджер пакетов `pip`. Он пытается найти модуль в каталоге Python Package Index (об этом позднее), загружает его со всеми зависимостями и берет на себя установку. Базовый синтаксис `pip` очень прост. Например, чтобы установить популярную библиотеку запросов из командной строки, достаточно выполнить команду:

```
$ python3.6 -m pip install requests
```

Для обновления библиотеки до последней версии достаточно добавить ключ `--upgrade`:

```
$ python3.6 -m pip install --upgrade requests
```

Наконец, если вам понадобится установить конкретную версию пакета, укажите ее после имени:

```
$ python3.6 -m pip install requests==2.11.1
$ python3.6 -m pip install requests>=2.9
```

19.4.1. Установка с флагом –user

Достаточно часто вы не можете или не хотите установить пакет Python в основном системном экземпляре Python. Возможно, вам нужна самая последняя версия библиотеки, а другое приложение (или сама система) продолжает использовать старую версию. А может быть, вы не обладаете привилегиями доступа для модификации основной установки Python в системе. В таких случаях можно установить библиотеку с флагом `--user`. С этим флагом библиотека устанавливается в домашнем каталоге пользователя, недоступном для других пользователей. Например, следующая команда устанавливает `requests` только для локального пользователя:

```
$ python3.6 -m pip install --user requests
```

Как упоминалось ранее, этот способ особенно полезен при работе в системе, в которой вы не обладаете достаточными административными правами для установки программ, или если вы хотите установить другую версию модуля. Если ваши потребности выходят за рамки базовых методов установки, описанных здесь, начните с раздела *Installing Python Modules* в документации Python.

19.4.2. Виртуальные среды

Если вы хотите избежать установки библиотек в системную копию Python, есть другой, более эффективный вариант — создание *виртуальной среды*. Виртуальная среда представляет собой автономную структуру каталогов, которая содержит как установку Python, так и ее дополнительные пакеты. Так как все окружение Python содержится в виртуальной среде, установленные в ней библиотеки и модули не будут конфликтовать с библиотеками и модулями основной системы или других виртуальных сред, что позволяет разным приложениям использовать разные версии Python и его пакетов.

Создание и использование виртуальной среды состоит из двух этапов. Сначала создается сама среда:

```
$ python3.6 -m venv test-env
```

Эта команда создает среду, в которой Python и `pip` установлены в каталоге с именем test-env. Затем, когда среда будет создана, ее нужно активировать. В Windows это делается так:

```
> test-env\Scripts\activate.bat
```

В системах UNIX или MacOS для этого выполняется сценарий активации:

```
$ source test-env/bin/activate
```

При активации среды вы можете использовать `pip` для управления пакетами, как и прежде, но в виртуальной среде `pip` является автономной командой:

```
$ pip install requests
```

Кроме того, та версия Python, которую вы используете для создания среды, станет версией Python по умолчанию для этой среды, поэтому вы можете использовать команду `python` вместо `python3` или `python3.6`.

Виртуальные среды чрезвычайно полезны для управления проектами и их зависимостями. Их применение фактически превратилось в стандартную практику, особенно для разработчиков, трудящихся над несколькими проектами. За дополнительной информацией обращайтесь к разделу *Virtual Environments and Packages* учебника Python в электронной документации Python.

19.5. PyPI (CheeseShop)

Хотя пакеты `distutils` справляются со своей работой, есть одна загвоздка: вы должны найти правильный пакет, что может быть непросто. А после того как пакет будет найден, было бы неплохо иметь достаточно надежный источник для загрузки пакета.

Для этого за прошедшие годы появилось много разных репозиториев пакетов Python. В настоящее время официальным (но ни в коей мере не единственным) репозиторием Python является каталог Python Package Index, или PyPI, на сайте Python. Вы можете обратиться к нему по ссылке на главной странице или напрямую по адресу https://pypi.python.org. PyPI содержит более 6000 пакетов для разных версий Python, с указанием дат добавления и имен, с возможностью поиска и разбивкой на категории.

На момент написания книги стало известно о появлении новой версии PyPI. Сейчас она называется *The Warehouse*. Эта версия все еще находится на стадии тестирования, но она обещает предоставить намного более удобные и комфортные средства поиска.

PyPI станет логичным следующим шагом, если вам не удается найти нужную функциональность в стандартной библиотеке.

Итоги

- Python содержит мощную стандартную библиотеку, которая подходит для большего количества стандартных ситуаций, чем во многих других языках программирования. Внимательно проверьте доступную функциональность стандартной библиотеки перед тем, как переходить к поиску внешних модулей.
- Если вам понадобится внешний модуль, проще всего воспользоваться заранее построенными пакетами для операционной системы, но иногда они устаревают и их бывает труднее найти.
- Стандартный способ установки из исходного кода основан на использовании `pip`, но для предотвращения конфликтов между несколькими проектами лучше всего создать виртуальную среду с помощью модуля `venv`.
- Обычно следующим логичным шагом в поиске внешних модулей становится каталог Python Package Index (PyPI).

ЧАСТЬ 4

Работа с данными

В этой части вы немного потренируетесь в практическом использовании Python — в частности, для работы с данными. Обработка данных является одной из самых сильных сторон Python. Мы начнем с простейшей обработки файлов; потом перейдем к чтению и записи в плоские файлы, работе со структурированными форматами (такими как JSON и Excel), работе с базами данных и использованию Python для анализа данных.

Эти главы будут более объектно-ориентированными, чем оставшаяся часть книги. Они дадут вам возможность попрактиковаться в применении Python для обработки данных. Главы и проекты этой части можно изучать в том порядке и в том сочетании, которые лучше соответствуют вашим целям.

20 Обработка данных в файлах

Эта глава охватывает следующие темы:

- ✓ Перемещение и переименование файлов
- ✓ Сжатие и шифрование файлов
- ✓ Выборочное удаление файлов

В этой главе рассматриваются основные операции, которые могут использоваться при управлении постоянно растущей группой файлов. Это могут быть файлы журналов или файлы из регулярной поставки данных, но каким бы ни был их источник, вы не можете просто забыть об их существовании. Как сохранить их, как управлять ими и в конечном итоге избавиться от них по плану, но без ручного вмешательства?

20.1. Проблема: бесконечный поток файлов данных

Многие системы генерируют непрерывную серию файлов данных. Такими файлами могут быть журналы сайта электронной торговли или обычного процесса, ночные поставки информации о продуктах от сервера, автоматизированные поставки данных для рекламы в интернете, исторические данные биржевых котировок, словом, они могут приходить из тысяч разных источников. Часто это неструктурированные текстовые файлы без сжатия, содержащие низкоуровневые данные, которые либо являются входными, либо становятся побочным продуктом деятельности других процессов. Несмотря на свое скромное происхождение, содержащиеся в них данные обладают потенциальной ценностью. Однако файлы невозможно отбросить в конце дня, а это означает, что каждый день их количество будет расти. Со временем файлы будут накапливаться, но в какой-то момент работать с ними вручную станет невозможно, а объем занимаемого ими пространства станет неприемлемо высоким.

20.2. Сценарий: адовая поставка продуктов

Типичная ситуация, с которой я сталкивалась, — ежедневная поставка данных по продуктам. Данные могут поступать от поставщиков или отправляться для сетевого маркетинга, но базовые аспекты остаются неизменными.

Допустим, вы получаете от поставщика данные о продуктах. Поставка данных приходит один раз в день, каждая строка соответствует одному товару. В каждой строке присутствуют поля для кода товара (SKU, Stock-Keepint Unit); краткого описания товара; цены, ширины, высоты и длины; статуса товара (допустим, товар имеется в наличии/временно отсутствует на складе); и возможно, нескольких других характеристик в зависимости от области деятельности.

Кроме основного файла с информацией, вы также можете получать несколько других файлов — возможно, с информацией о взаимосвязанных продуктах, расширенных атрибутах и т. д. В этом случае у вас появятся файлы с одинаковыми именами, которые поступают каждый день и оказываются в одном каталоге для обработки.

Теперь предположим, что вы каждый день получаете три взаимосвязанных файла: item_info.txt, item_attributes.txt, related_items.txt. Эти три файла поступают каждый день и обрабатываются. Если бы обработка была единственным требованием, беспокоиться было бы не о чем; достаточно ежедневно заменять файлы. Но что, если старые данные нельзя просто выкинуть? Представьте, что необработанные данные нужно хранить на случай, если точность процесса окажется под сомнением и вам нужно будет обратиться к старым файлам. А может, вы хотите отслеживать изменения в данных со временем. Независимо от причины, необходимость в хранении исходных данных означает, что с файлами нужно что-то сделать.

Самое простое, что можно сделать, — пометить файлы датами получения и переместить их в архивную папку. В этом случае каждый новый набор файлов можно принять, обработать, переименовать и переместить, а процесс повторяется без потери данных.

После нескольких повторений структура каталогов может выглядеть примерно так, как показано ниже.

```
working/  ◄——  Главный рабочий каталог с текущими файлами для обработки
    item_info.txt
    item_attributes.txt
    related_items.txt
    archive/  ◄——  Подкаталог для архивации обработанных файлов
        item_info_2017-09-15.txt
        item_attributes_2017-09-15.txt
        related_items_2017-09-15.txt
        item_info_2016-07-16.txt
        item_attributes_2017-09-16.txt
        related_items_2017-09-16.txt
        item_info_2017-09-17.txt
        item_attributes_2017-09-17.txt
        related_items_2017-09-17.txt
        ...
```

Подумайте, что необходимо сделать для организации этого процесса? Сначала нужно переименовать файлы и добавить к имени файла текущую дату. Для этого необходимо получить имена файлов, которые нужно переименовать, затем следует получить основу имен файлов без расширений. К основе добавляется строка, построенная на основе текущей даты, к ней добавляется расширение, после чего файл переименовывается и перемещается в архивный каталог.

БЫСТРАЯ ПРОВЕРКА: РАССМОТРЕНИЕ ВАРИАНТОВ

Какие варианты существуют для решения перечисленных задач? Какие модули стандартной библиотеки вы бы предложили для этого использовать? Если хотите, прервите чтение и напишите код для решения этой задачи. Затем сравните решение с тем, которое будет разработано позднее.

Имена файлов можно получить несколькими способами. Если вы твердо уверены в том, что имена файлов остаются неизменными, а файлов не так много, вы *можете* жестко зафиксировать их в своем сценарии. Однако более надежный путь основан на использовании модуля `pathlib` и метода `glob` объекта пути.

```
>>> import pathlib
>>> cur_path = pathlib.Path(".")
>>> FILE_PATTERN = "*.txt"
>>> path_list = cur_path.glob(FILE_PATTERN)
>>> print(list(path_list))
[PosixPath('item_attributes.txt'), PosixPath('related_items.txt'),
     PosixPath('item_info.txt')]
```

Теперь можно перебрать пути, соответствующие шаблону `FILE_PATTERN`, и применить необходимые изменения. Помните, что к имени каждого файла нужно добавить дату, а переименованные файлы перемещаются в архивный каталог. При использовании `pathlib` вся операция может выглядеть так, как показано ниже.

Листинг 20.1. Файл files_01.py

```
import datetime
import pathlib

FILE_PATTERN = "*.txt"
ARCHIVE = "archive"

if __name__ == '__main__':

    date_string = datetime.date.today().strftime("%Y-%m-%d")

    cur_path = pathlib.Path(".")
    paths = cur_path.glob(FILE_PATTERN)

    for path in paths:
```

Задает шаблоны для файлов и архивного каталога

Для выполнения этого кода должен существовать каталог с именем «archive»

Использует объект date из библиотеки datetime для создания строки, представляющей текущую дату

```
new_filename = "{}_{}{}".format(path.stem, date_string, path.suffix)
new_path = cur_path.joinpath(ARCHIVE, new_filename)
path.rename(new_path)
```

Создает новый путь из текущего пути, архивного каталога и нового имени файла

Переименовывает (и перемещает) файл за один шаг

Стоит заметить, что объекты `Path` упрощают эту операцию, потому что отделение основы от суффикса не требует никакого специального разбора. Эта операция также выполняется проще, чем можно было ожидать, потому что метод `rename` фактически перемещает файл, если ей будет передан путь с новым каталогом.

Сценарий получается очень простым, и он эффективно решает свою задачу в нескольких строках кода. В следующих ближайших разделах вы узнаете, как справиться с более сложными требованиями.

БЫСТРАЯ ПРОВЕРКА: ПОТЕНЦИАЛЬНЫЕ ПРОБЛЕМЫ

Так как предыдущее решение устроено очень просто, скорее всего, оно не сможет справиться со многими ситуациями. Какие потенциальные проблемы могут возникнуть со сценарием из примера? Как решить эти проблемы?

Возьмем схему формирования имен файлов, которая использует год, месяц и имя файла в указанном порядке. Какими преимуществами обладает эта схема? А какие у нее есть недостатки? Сможете ли вы привести доводы в пользу размещения строки даты в другом месте имени файла (например, в начале или в конце)?

20.3. Дальнейшая организация

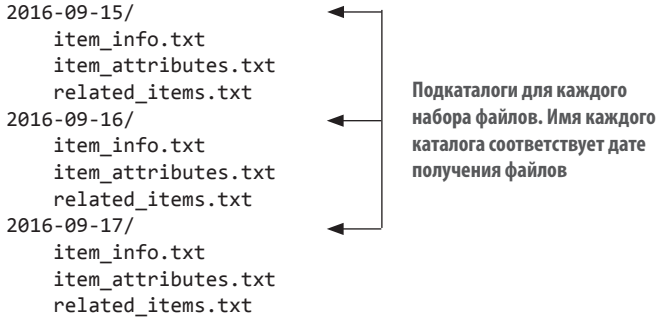
Способ хранения файлов, описанный в предыдущем разделе, работает, но у него есть некоторые недостатки. Во-первых, с накоплением файлов управление станет более хлопотным, потому что за год в одном каталоге появится 365 групп взаимосвязанных файлов и отыскать взаимосвязанные файлы можно только по именам. Конечно, если файлы будут поступать чаще или в наборе будет больше файлов, проблем станет только больше.

Ситуацию можно исправить изменением способа архивации файлов. Вместо того чтобы изменять имена файлов и включать даты их получения, можно создать отдельный подкаталог для каждого набора файлов и присвоить имя подкаталога в соответствии с датой получения. Структура каталогов может выглядеть так:

```
working/
    item_info.txt
    item_attributes.txt
    related_items.txt
    archive/
```

Главный рабочий каталог с текущими файлами для обработки

Главный подкаталог для архивации обработанных файлов



К преимуществам этой схемы следует отнести то, что файлы каждого набора хранятся вместе. Сколько бы наборов файлов у вас ни было и сколько бы файлов ни входило в набор, вы легко найдете все файлы конкретного набора.

ПОПРОБУЙТЕ САМИ: РЕАЛИЗАЦИЯ С НЕСКОЛЬКИМИ КАТАЛОГАМИ

Как бы вы изменили разработанный ранее код для архивации каждого набора файлов в подкаталоге с именем, соответствующим дате его получения? Не торопитесь, реализуйте и протестируйте каждый код.

Как выясняется, архивация файлов по подкаталогам не требует значительно большей работы по сравнению с первым решением. Достаточно сделать всего один дополнительный шаг — создание подкаталога перед переименованием файла. Этот сценарий демонстрирует один из способов выполнения этого шага.

Листинг 20.2. Файл files_02.py

```
import datetime
import pathlib

FILE_PATTERN = "*.txt"
ARCHIVE = "archive"

if __name__ == '__main__':

    date_string = datetime.date.today().strftime("%Y-%m-%d")

    cur_path = pathlib.Path(".")

    new_path = cur_path.joinpath(ARCHIVE, date_string)
    new_path.mkdir()

    paths = cur_path.glob(FILE_PATTERN)

    for path in paths:
        path.rename(new_path.joinpath(path.name))
```

Обратите внимание: каталог создается только один раз перед тем, как файлы будут перемещены

Это решение группирует взаимосвязанные файлы, несколько упрощая работу с ними в составе наборов.

БЫСТРАЯ ПРОВЕРКА: АЛЬТЕРНАТИВНЫЕ РЕШЕНИЯ

Как бы вы написали сценарий, делающий то же самое без использования `pathlib`? Какие библиотеки и функции вы бы использовали?

20.4. Экономия места: сжатие и удаление

До настоящего момента нас интересовало прежде всего управление группами полученных файлов. Однако со временем файлы данных накапливаются, и в какой-то момент возникает проблема с объемом занимаемого ими дискового пространства. В такой ситуации есть несколько вариантов. Конечно, можно увеличить размер диска. Эта стратегия будет особенно простой и экономичной в облачных платформах. Однако следует помнить, что расширение дискового пространства не решает проблему, а только откладывает ее решение на будущее.

20.4.1. Сжатие файлов

Если у вас возникнут проблемы с пространством, занимаемым файлами, стоит подумать об их сжатии. Существует много разных способов сжатия файлов или групп файлов, но в целом они похожи. В этом разделе рассматривается архивация файлов данных, относящихся к разным дням, в один zip-файл. Если файлы в основном содержат текст и имеют относительно большие размеры, экономия от сжатия может быть довольно впечатляющей.

Для этого сценария в качестве имени каждого zip-файла используется та же строка даты, к которой присоединяется расширение .zip. В листинге 20.2 мы создаем новый каталог в архивном каталоге и перемещаем в него файлы. Полученная структура каталогов выглядит так:

```
working/                    <-- Главный рабочий каталог с текущими файлами;
    archive/                    после обработки эти файлы архивируются и удаляются
        2016-09-15.zip      |
        2016-09-16.zip      |   Zip-архивы, каждый из которых содержит файлы item_info.txt,
        2016-09-17.zip      |   attribute_info.text и related_items.txt за этот день
```

Разумеется, для использования zip-файлов необходимо изменить некоторые этапы описанного процесса.

ПОПРОБУЙТЕ САМИ: ПСЕВДОКОД АРХИВАЦИИ В ZIP-ФАЙЛЫ

Напишите псевдокод для решения, сохраняющего файлы данных в zip-файлах. Какие модули и функции вы предлагаете использовать? Попробуйте запрограммировать свое решение и убедитесь в том, что оно работает.

Одним из ключевых дополнений в новом сценарии становится импортирование библиотеки zipfile с кодом для создания нового объекта zip-файла в архивном каталоге. После этого объект zip-файла может использоваться для записи файлов данных в новый zip-файл. Наконец, так как перемещение файлов уже не происходит, необходимо удалить исходные файлы из рабочего каталога. Одно из возможных решений может выглядеть так:

Листинг 20.3. Файл files_03.py

```
import datetime
import pathlib
import zipfile  <-- Импортирует библиотеку zipfile

FILE_PATTERN = "*.txt"
ARCHIVE = "archive"

if __name__ == '__main__':

    date_string = datetime.date.today().strftime("%Y-%m-%d")

    cur_path = pathlib.Path(".")
    paths = cur_path.glob(FILE_PATTERN)

    zip_file_path = cur_path.joinpath(ARCHIVE, date_string + ".zip")
    zip_file = zipfile.ZipFile(str(zip_file_path), "w")
    for path in paths:
        zip_file.write(str(path))  <-- Записывает текущий файл в zip-файл
        path.unlink()  <-- Удаляет текущий файл из рабочего каталога
```

Создает путь к zip-файлу в архивном каталоге

Открывает новый объект zip-файла для записи; вызов str() необходим для преобразования Path в строку

20.4.2. Удаление старых файлов

Сжатие файлов данных в архивы в формате zip позволит достичь значительной экономии места; возможно, большего вам не понадобится. Тем не менее если файлов очень много или файлы плохо сжимаются (как, например, графические файлы в формате JPEG), может оказаться, что вскоре со свободным пространством снова возникнут проблемы. Также может оказаться, что изменения в данных не столь значительны, и хранить архивную копию каждого набора данных в долгосрочной перспективе не обязательно. Таким образом, хотя хранение ежедневных данных в течение недели или месяца может быть полезным, для хранения всех наборов данных в течение более длительного времени нет оснований. Если возраст данных превышает несколько месяцев, может быть достаточно хранить один набор файлов за неделю или даже, возможно, за месяц.

Предположим, через несколько месяцев ежедневного получения файлов данных и их архивации в zip-файлах вам говорят, что для файлов более месячной давности достаточно хранить только один файл за неделю.

Простейший сценарий удаления старых файлов стирает все файлы, которые вам больше не нужны, — в данном случае все файлы, кроме одного за неделю, для файлов, возраст которых превышает один месяц. При проектировании такого сценария полезно знать ответы на два вопроса:

- Если вы намерены хранить один файл за неделю, как выбрать день недели для сохранения?
- Как часто проводить удаление старых файлов? Каждый день, каждую неделю или даже раз в месяц? Если вы решите, что удаление должно происходить ежедневно, будет разумно объединить удаление со сценарием архивации. С другой стороны, если удаление должно происходить только раз в неделю или даже в месяц, эти две операции должны выполняться в разных сценариях.

В нашем примере для большей ясности будет написан раздельный сценарий удаления лишних файлов, который может запускаться с произвольным интервалом. Допустим, вы решили, что сохраняться должны только файлы, полученные во вторник, если с момента их получения прошло более одного месяца. Ниже приведен пример сценария.

Листинг 20.4. Файл files_04.py

```
from datetime import datetime, timedelta
import pathlib
import zipfile

FILE_PATTERN = "*.zip"
ARCHIVE = "archive"
ARCHIVE_WEEKDAY = 1
if __name__ == '__main__':
    cur_path = pathlib.Path(".")
    zip_file_path = cur_path.joinpath(ARCHIVE)

    paths = zip_file_path.glob(FILE_PATTERN)
    current_date = datetime.today()

    for path in paths:
        name = path.stem
        path_date = datetime.strptime(name, "%Y-%m-%d")
        path_timedelta = current_date - path_date
        if path_timedelta > timedelta(days=30) and path_date.weekday() !=
    ARCHIVE_WEEKDAY:
            path.unlink()
```

Получает объект даты/времени для текущего дня

path.stem возвращает имя файла без расширения

При вычитании одной даты из другой будет получен объект timedelta

strptime разбирает строку и преобразует ее в объект даты/времени на основании форматной строки

timedelta(days=30) создает объект timedelta для 30 дней; метод weekday() возвращает целое число, представляющее день недели (понедельник=0)

Этот код показывает, как при помощи библиотек Python datetime и pathlib реализовать удаление файлов по дате всего в нескольких строках кода. Так как имена

архивных файлов определяются датами их получения, для получения путей к этим файлам можно воспользоваться методом `glob`, отделить основу и использовать `strptime` для преобразования ее в объект `datetime`. Далее объекты `timedelta` модуля `datetime` и метод `weekday()` используются для нахождения возраста файла и дня недели, с последующим удалением ненужных файлов.

БЫСТРАЯ ПРОВЕРКА: РАЗЛИЧНЫЕ ПАРАМЕТРЫ

Рассмотрите другие варианты удаления. Как изменить код в листинге 20.4, чтобы сохранялся только один файл за месяц? Как изменить его, чтобы удалялись все файлы за предыдущий месяц и старше, кроме одного за неделю (подсказка: это не то же самое, что файлы возрастом более 30 дней!).

Итоги

- Модуль `pathlib` может серьезно упростить такие операции с файлами, как получение корня и расширения, перемещение и переименование, а также поиск по шаблону.
- С ростом количества и сложности файлов автоматизированная процедура архивации становится жизненно необходимой. Python предоставляет несколько простых способов ее создания.
- Сжатие и удаление старых файлов данных позволяет радикально сократить затраты дискового пространства.

21 Обработка файлов данных

Эта глава охватывает следующие темы:

- ✓ Использование ETL (extract-transform-load)
- ✓ Чтение текстовых файлов данных (обычный текст и CSV)
- ✓ Чтение файлов электронных таблиц
- ✓ Нормализация, очистка и сортировка данных
- ✓ Запись файлов данных

Большая часть данных распространяется в текстовых файлах. Это может быть как неструктурированный текст (например, подборка сообщений или собрание литературных текстов), так и более структурированные данные, в которых каждая строка представляет собой запись, а поля разделяются специальным символом-разделителем — запятой, символом табуляции или вертикальной чертой (|). Текстовые файлы могут быть огромными; набор данных может занимать десятки и даже сотни файлов, а содержащиеся в нем данные могут быть неполными или искаженными. При таком разнообразии вы почти неизбежно столкнетесь с задачей чтения и использования данных из текстовых файлов. В этой главе представлены основные стратегии для решения этой задачи в Python.

21.1. Знакомство с ETL

Необходимость извлекать данные из файлов, разбирать их, преобразовывать в удобный формат, а затем что-то делать появилась практически одновременно с файлами данных. Более того, для этого процесса даже существует стандартный термин: ETL (Extract-Transform-Load, то есть «извлечение–преобразование–загрузка»). Под извлечением подразумевается процесс чтения источника данных и его разбора при

необходимости. Преобразование может подразумевать чистку и нормализацию данных, а также объединение, разбиение и реорганизацию содержащихся в них записей. Наконец, под загрузкой понимается сохранение преобразованных данных в новом месте (в другом файле или базе данных). В этой главе рассматриваются основы реализации ETL на языке Python, начиная с текстовых файлов данных и заканчивая сохранением преобразованных данных в других файлах. Более структурированные файлы данных рассматриваются в главе 22, а хранение информации в базе данных — в главе 23.

21.2. Чтение текстовых файлов

Первая составляющая ETL — извлечение — подразумевает открытие файла и чтение его содержимого. На первый взгляд звучит просто, но даже здесь могут возникнуть проблемы — например, размер файла. Если файл слишком велик для размещения в памяти, код необходимо структурировать так, чтобы он работал с меньшими сегментами файла (возможно, по одной строке).

21.2.1. Кодировка текста: ASCII, Юникод и другие

Другая возможная проблема связана с кодировкой. Эта глава посвящена работе с текстовыми файлами, и на самом деле большая доля данных, передаваемых в реальном мире, хранится в текстовых файлах. Тем не менее точная природа *текста* может изменяться в зависимости от приложения, от пользователя и, конечно, от страны.

Иногда *текст* несет информацию в кодировке ASCII, включающей 128 символов, только 95 из которых относятся к категории печатаемых. К счастью, кодировка ASCII является «наименьшим общим кратным» большинства ситуаций с передачей данных. С другой стороны, она никак не справится со сложностями многочисленных алфавитов и систем письменности, существующих в мире. Чтение файлов в кодировке ASCII почти наверняка приведет к тому, что при чтении неподдерживаемых символов, будь то немецкое ü, португальское ç или практически любой символ из языка, кроме английского, начнутся проблемы и появятся ошибки.

Эти ошибки возникают из-за того, что в ASCII используются 7-битовые значения, тогда как байты в типичном файле состоят из 8 бит, что позволяет представить 256 возможных значений вместо 128 для 7-разрядных значений. Эти дополнительные коды обычно используются для хранения дополнительных значений — от расширенных знаков препинания (таких, как среднее и короткое тире) до различных знаков (товарный знак, знак авторского права и знак градуса) и версий алфавитных символов с диакритическими знаками. Всегда существовала одна проблема: при чтении текстового файла вы могли столкнуться с символом, выходящим за пределы ASCII-диапазона из 128 символов, и не могли быть уверены в том, какой именно символ закодирован. Допустим, вам встречается символ с кодом 214. Что это? Знак деления, буква Ö или что-нибудь еще? Без исходного кода, создавшего этот файл, узнать это невозможно.

Юникод и UTF-8

Для устранения этой неоднозначности можно воспользоваться Юникодом. Кодировка Юникода, называемая UTF-8, поддерживает базовые ASCII-символы без каких-либо изменений, но при этом также допускает практически неограниченный набор других символов и знаков из стандарта Юникод. Благодаря своей гибкости UTF-8 используется более чем в 85 % веб-страниц, существовавших на момент написания этой главы. Это означает, что при чтении текстовых файлов лучше всего ориентироваться на UTF-8. Если файлы содержат только ASCII-символы, они будут прочитаны правильно, но вы также получаете страховку на случай, если другие символы закодированы в UTF-8. К счастью, строковый тип данных Python 3 по умолчанию рассчитан на поддержку Юникода.

Даже с Юникодом возможны ситуации, когда в тексте встречаются значения, которые невозможно успешно декодировать. Функция `open` в Python получает дополнительный параметр `errors`, который определяет, как следует поступать с ошибками кодирования при чтении или записи файлов. По умолчанию используется значение `'strict'`, с которым при каждом обнаружении ошибки кодирования инициируется ошибка. Другие полезные значения — `'ignore'` (пропустить символ, вызвавший ошибку); `'replace'` (символ заменяется специальным маркером — обычно `?`); `'backslashreplace'` (символ заменяется экранирующей последовательностью с \) и `'surrogateescape'` (символ-нарушитель преобразуется в приватный кодовый пункт Юникода при чтении и обратно в исходную последовательность байтов при записи). Выбор способа обработки или разрешения ошибок кодирования зависит от конкретной ситуации.

Рассмотрим короткий пример файла, содержащего недопустимый символ UTF-8, и посмотрим, как этот символ обрабатывается в разных режимах. Сначала запишите файл с использованием `bytes` и двоичного режима:

```
>>> open('test.txt', 'wb').write(bytes([65, 66, 67, 255, 192,193]))
```

В результате выполнения команды создается файл из символов `"ABC"`, за которыми следуют три символа, не входящие в ASCII, которые могут по-разному отображаться в зависимости от используемого способа кодирования. Если воспользоваться vim для просмотра файла, результат будет выглядеть так:

```
ABCÿÀÁ
~
```

Когда файл будет создан, попробуйте прочитать его в используемом по умолчанию режиме обработки ошибок `'strict'`:

```
>>> x = open('test.txt').read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.6/codecs.py", line 321, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 3:
     invalid start byte
```

Четвертый байт со значением 255 не является действительным символом UTF-8 в этой позиции, поэтому в режиме `'strict'` происходит исключение. А теперь посмотрим, как другие режимы обработки ошибок справляются с тем же файлом, не забывая о том, что последние три символа инициируют ошибку:

```
>>> open('test.txt', errors='ignore').read()
'ABC'
>>> open('test.txt', errors='replace').read()
'ABC���'
>>> open('test.txt', errors='surrogateescape').read()
'ABC\udcff\udcc0\udcc1'
>>> open('test.txt', errors='backslashreplace').read()
'ABC\\xff\\xc0\\xc1'
>>>
```

Если вы хотите, чтобы проблемные символы просто исчезли, используйте режим `'ignore'`. Режим `'replace'` только помечает позиции недействительных символов, а другие режимы по-разному пытаются сохранить недействительные символы без интерпретации.

21.2.2. Неструктурированный текст

Неструктурированные текстовые файлы читаются проще всего, но они же создают больше всего проблем с извлечением информации. Обработка неструктурированного текста может меняться в широчайших пределах в зависимости как от природы текста, так и от того, что вы собираетесь с ним делать, так что сколько-нибудь подробное обсуждение обработки текста выходит за рамки книги. Однако короткий пример поможет продемонстрировать некоторые базовые проблемы и заложит основу для обсуждения файлов со структурированными текстовыми данными.

Одна из самых простых проблем — выбор базовой логической единицы в файле. Если вы используете подборку из тысяч сообщений «Твиттера», текст «Моби Дика» или коллекцию новостей, их нужно как-то разбить на блоки. В случае твитов каждый блок может помещаться в одной строке, а чтение и обработка каждой строки файла организуется достаточно просто.

В случае с «Моби Диком» и даже отдельной новостью проблема усложняется. Конечно, текст романа и даже текст новости обычно нежелательно рассматривать как единый блок. В таком случае нужно решить, какие блоки вам нужны, а затем выработать стратегию разделения файла на блоки. Возможно, вы предпочтете обрабатывать текст по абзацам. В таком случае следует определить, как организована разбивка текста на абзацы в файле, и написать код соответствующим образом. Если абзацы совпадают со строками текстового файла, сделать это будет несложно. Однако нередко один абзац текстового файла может состоять из нескольких строк в текстовом файле, и вам придется потрудиться.

А теперь рассмотрим пару примеров.

```
Call me Ishmael. Some years ago--never mind how long precisely--
having little or no money in my purse, and nothing particular
to interest me on shore, I thought I would sail about a little
and see the watery part of the world.  It is a way I have
of driving off the spleen and regulating the circulation.
Whenever I find myself growing grim about the mouth;
whenever it is a damp, drizzly November in my soul; whenever I
find myself involuntarily pausing before coffin warehouses,
and bringing up the rear of every funeral I meet;
and especially whenever my hypos get such an upper hand of me,
that it requires a strong moral principle to prevent me from
deliberately stepping into the street, and methodically knocking
people's hats off--then, I account it high time to get to sea
as soon as I can.  This is my substitute for pistol and ball.
With a philosophical flourish Cato throws himself upon his sword;
I quietly take to the ship.  There is nothing surprising in this.
If they but knew it, almost all men in their degree, some time
or other, cherish very nearly the same feelings towards
the ocean with me.

There now is your insular city of the Manhattoes, belted round by wharves
as Indian isles by coral reefs--commerce surrounds it with her surf.
Right and left, the streets take you waterward.  Its extreme downtown
is the battery, where that noble mole is washed by waves, and cooled
by breezes, which a few hours previous were out of sight of land.
Look at the crowds of water-gazers there.
```

В этом примере (с началом текста «Моби Дика») строки разбиваются более-менее так, как они бы разбивались на страницы, а абзацы обозначаются одиночной пустой строкой. Если вы хотите обрабатывать каждый абзац как единое целое, необходимо разбить текст по пустым строкам. К счастью, эта задача легко решается методом строк `split()`. Каждый символ новой строки в тексте представляется комбинацией `"\n"`. Естественно, последняя строка текста каждого абзаца завершается символом новой строки, и если следующая строка текста пуста, то за ней немедленно следует второй символ новой строки:

```
>>> moby_text = open("moby_01.txt").read()
>>> moby_paragraphs = moby_text.split("\n\n")
>>> print(moby_paragraphs[1])
There now is your insular city of the Manhattoes, belted round by wharves
as Indian isles by coral reefs--commerce surrounds it with her surf.
Right and left, the streets take you waterward.  Its extreme downtown
is the battery, where that noble mole is washed by waves, and cooled
by breezes, which a few hours previous were out of sight of land.
Look at the crowds of water-gazers there.
```

Читает все содержимое файла в одну строку

Разбивает по двум смежным символам новой строки

Разбиение текста на абзацы — очень простой шаг процесса обработки неструктурированного текста. Возможно, также потребуется провести дополнительную нормализацию текста перед дальнейшей обработкой. Допустим, вы хотите подсчитать

частоту вхождения каждого слова в текстовом файле. Если просто разбить файл по пропускам, вы получите список слов в файле, однако точно подсчитать вхождения будет не так просто, потому что *This*, *this*, *this.* и *this,* не будут считаться одним и тем же словом. Чтобы этот код правильно работал, необходимо нормализовать текст, удалив знаки препинания и преобразовав весь текст к одному регистру перед обработкой. В приведенном выше примере текста код построения нормализованного списка слов может выглядеть так:

```
>>> moby_text = open("moby_01.txt").read()  <-- Читает все содержимое файла в одну строку
>>> moby_paragraphs = moby_text.split("\n\n")
>>> moby = moby_paragraphs[1].lower()  <-- Преобразует все символы к нижнему регистру
>>> moby = moby.replace(".", "")  <-- Удаляет точки
>>> moby = moby.replace(",", "")  <-- Удаляет запятые
>>> moby_words = moby.split()
>>> print(moby_words)
['there', 'now', 'is', 'your', 'insular', 'city', 'of', 'the', 'manhattoes,',
     'belted', 'round', 'by', 'wharves', 'as', 'indian', 'isles', 'by',
     'coral', 'reefs--commerce', 'surrounds', 'it', 'with', 'her', 'surf',
     'right', 'and', 'left,', 'the', 'streets', 'take', 'you', 'waterward',
     'its', 'extreme', 'downtown', 'is', 'the', 'battery,', 'where', 'that',
     'noble', 'mole', 'is', 'washed', 'by', 'waves,', 'and', 'cooled', 'by',
     'breezes,', 'which', 'a', 'few', 'hours', 'previous', 'were', 'out',
     'of', 'sight', 'of', 'land', 'look', 'at', 'the', 'crowds', 'of',
     'water-gazers', 'there']
```

БЫСТРАЯ ПРОВЕРКА: НОРМАЛИЗАЦИЯ

Внимательно присмотритесь к сгенерированному списку слов. Вы видите какие-нибудь проблемы с нормализацией? Какие еще трудности могут возникнуть с более обширным блоком текста? Как бы вы подошли к решению этих проблем?

21.2.3. Неструктурированные файлы с разделителями

Неструктурированные файлы читаются очень просто, однако отсутствие структуры также является их недостатком. Часто бывает удобнее определить для файла некоторую структуру, чтобы упростить выборку отдельных значений. В простейшем варианте файл разбивается на строки, и в каждой строке хранится один информационный элемент. Например, это может быть список имен файлов для обработки, список имен людей или серия показаний температуры от удаленного датчика. В таких случаях разбор данных организуется очень просто: вы читаете строку и при необходимости преобразуете ее к нужному типу. Это все, что нужно, чтобы файл был готов к использованию.

Впрочем, ситуация не настолько проста. Чаще требуется сгруппировать несколько взаимосвязанных элементов данных, а ваш код должен прочитать их вместе. Обычно для этого взаимосвязанные данные размещаются в одной строке и разделяются специальным символом. В этом случае при чтении каждой строки файла

специальные символы используются для разбиения данных на поля и сохранения значений полей в переменных для последующей обработки.

В следующем файле содержатся данные температуры в формате с разделителями:

```
State|Month Day, Year Code|Avg Daily Max Air Temperature (F)|Record Count for
     Daily Max Air Temp (F)
Illinois|1979/01/01|17.48|994
Illinois|1979/01/02|4.64|994
Illinois|1979/01/03|11.05|994
Illinois|1979/01/04|9.51|994
Illinois|1979/05/15|68.42|994
Illinois|1979/05/16|70.29|994
Illinois|1979/05/17|75.34|994
Illinois|1979/05/18|79.13|994
Illinois|1979/05/19|74.94|994
```

Данные в файле разделяются символом вертикальной черты (|). В данном примере они состоят из четырех полей: штата, даты наблюдений, средней максимальной температуры и количества станций, поставляющих данные. Другими стандартными разделителями являются символ табуляции и запятая. Пожалуй, запятая используется чаще всего, но разделителем может быть любой символ, который не будет встречаться в значениях (об этом чуть позже). Разделение данных запятыми настолько распространено, что этот формат часто называется CSV (Comma-Separated Values, то есть данные, разделенные запятыми), и файлы такого типа снабжаются расширением .csv как признаком формата.

Какой бы символ ни использовался в качестве разделителя, если вы знаете, что это за символ, вы можете написать собственный код на языке Python для разбиения строки на поля и возвращения их в виде списка. В предыдущем случае можно воспользоваться методом `split()` для преобразования строки в список значений:

```
>>> line = "Illinois|1979/01/01|17.48|994"
>>> print(line.split("|"))
['Illinois', '1979/01/01', '17.48', '994']
```

Этот прием очень легко реализуется, но все значения сохраняются в строковом виде, а это может быть неудобно для последующей обработки.

ПОПРОБУЙТЕ САМИ: ЧТЕНИЕ ФАЙЛА

Напишите код для чтения текстового файла (с именем temp_data_pipes_00a.txt, как показано в примере), преобразования каждой строки файла в список значений и добавления этого списка в общий список записей.

С какими проблемами вы столкнулись при реализации этого кода? Как бы вы преобразовали последние три поля к правильным типам (дата, вещественное значение и целое число?).

21.2.4. Модуль csv

Если вам часто приходится обрабатывать файлы данных с разделителями, стоит поближе познакомиться с модулем `csv` и его возможностями. Когда меня просили назвать любимый модуль из стандартной библиотеки Python, я не раз называла модуль `csv` — не от того, что он эффектно выглядит (это не так), а из-за того, что он, пожалуй, экономил мне больше времени и спасал меня от моих же потенциальных ошибок чаще, чем любой другой модуль.

Модуль `csv` — идеальный пример философии Python «батарейки в комплекте». Хотя вы прекрасно можете написать собственный код чтения файлов с разделителями (более того, это не так уж сложно), намного проще и надежнее использовать модуль Python. Модуль `csv` был протестирован и оптимизирован, и он предоставляет ряд возможностей, которые вы вряд ли бы стали реализовывать самостоятельно, но которые тем не менее достаточно удобны и экономят время.

Взгляните на предыдущие данные и решите, как бы вы прочитали их с модулем `csv`. Код разбора данных должен прочитать каждую строку и удалить завершающий символ новой строки, а затем разбить строку по символам `|` и присоединить список значений к общему списку строк. Решение могло бы выглядеть примерно так:

```
>>> results = []
>>> for line in open("temp_data_pipes_00a.txt"):
...     fields = line.strip().split("|")
...     results.append(fields)
...
>>> results
[['State', 'Month Day, Year Code', 'Avg Daily Max Air Temperature (F)',
     'Record Count for Daily Max Air Temp (F)'], ['Illinois', '1979/01/01',
     '17.48', '994'], ['Illinois', '1979/01/02', '4.64', '994'], ['Illinois',
     '1979/01/03', '11.05', '994'], ['Illinois', '1979/01/04', '9.51',
     '994'], ['Illinois', '1979/05/15', '68.42', '994'], ['Illinois', '1979/
     05/16', '70.29', '994'], ['Illinois', '1979/05/17', '75.34', '994'],
     ['Illinois', '1979/05/18', '79.13', '994'], ['Illinois', '1979/05/19',
     '74.94', '994']]
```

Если вы захотите сделать то же самое с модулем `csv`, код может выглядеть примерно так:

```
>>> import csv
>>> results = [fields for fields in
     csv.reader(open("temp_data_pipes_00a.txt", newline=''), delimiter="|")]
>>> results
[['State', 'Month Day, Year Code', 'Avg Daily Max Air Temperature (F)',
     'Record Count for Daily Max Air Temp (F)'], ['Illinois', '1979/01/01',
     '17.48', '994'], ['Illinois', '1979/01/02', '4.64', '994'], ['Illinois',
     '1979/01/03', '11.05', '994'], ['Illinois', '1979/01/04', '9.51',
     '994'], ['Illinois', '1979/05/15', '68.42', '994'], ['Illinois', '1979/
     05/16', '70.29', '994'], ['Illinois', '1979/05/17', '75.34', '994'],
     ['Illinois', '1979/05/18', '79.13', '994'], ['Illinois', '1979/05/19',
     '74.94', '994']]
```

В этом простом случае выигрыш по сравнению с самостоятельной реализацией решения не так уж велик. Тем не менее код получился на две строки короче и немного понятнее, и вам не нужно беспокоиться об отсечении символов новой строки. Настоящее преимущество проявляется, когда вы сталкиваетесь с более сложными случаями.

Данные в этом примере реальны, но в действительности они были упрощены и очищены. Реальные данные от источника будут более сложными. Реальные данные содержат больше полей, одни поля будут заключены в кавычки, а другие нет, а первое поле может быть пустым. Оригинал разделяется табуляциями, но в целях демонстрации я привожу их разделенными запятыми:

```
"Notes","State","State Code","Month Day, Year","Month Day, Year Code",Avg
     Daily Max Air Temperature (F),Record Count for Daily Max Air Temp
     (F),Min Temp for Daily Max Air Temp (F),Max Temp for Daily Max Air Temp
     (F),Avg Daily Max Heat Index (F),Record Count for Daily Max Heat Index
     (F),Min for Daily Max Heat Index (F),Max for Daily Max Heat Index
     (F),Daily Max Heat Index (F) % Coverage

,"Illinois","17","Jan 01, 1979","1979/01/
     01",17.48,994,6.00,30.50,Missing,0,Missing,Missing,0.00%
,"Illinois","17","Jan 02, 1979","1979/01/02",4.64,994,-
     6.40,15.80,Missing,0,Missing,Missing,0.00%
,"Illinois","17","Jan 03, 1979","1979/01/03",11.05,994,-
     0.70,24.70,Missing,0,Missing,Missing,0.00%
,"Illinois","17","Jan 04, 1979","1979/01/
     04",9.51,994,0.20,27.60,Missing,0,Missing,Missing,0.00%
,"Illinois","17","May 15, 1979","1979/05/
     15",68.42,994,61.00,75.10,Missing,0,Missing,Missing,0.00%
,"Illinois","17","May 16, 1979","1979/05/
     16",70.29,994,63.40,73.50,Missing,0,Missing,Missing,0.00%
,"Illinois","17","May 17, 1979","1979/05/
     17",75.34,994,64.00,80.50,82.60,2,82.40,82.80,0.20%
,"Illinois","17","May 18, 1979","1979/05/
     18",79.13,994,75.50,82.10,81.42,349,80.20,83.40,35.11%
,"Illinois","17","May 19, 1979","1979/05/
     19",74.94,994,66.90,83.10,82.87,78,81.60,85.20,7.85%
```

Обратите внимание: некоторые поля включают запятые. По правилам в таких случаях поле заключается в кавычки, чтобы показать, что его содержимое не предназначено для разбора и поиска разделителей. На практике (как и в данном случае) в кавычки часто заключается лишь часть полей, прежде всего те, значения которых могут содержать разделитель. Впрочем (как опять-таки в данном примере), некоторые поля заключаются в кавычки даже тогда, когда они вряд ли будут содержать разделитель.

В таких случаях доморощенные решения становятся слишком громоздкими. Теперь простое разбиение строки по символу-разделителю уже не работает; нужно проследить за тем, чтобы при поиске использовались только те разделители, которые не находятся внутри строк. Кроме того, необходимо удалить кавычки, которые

могут находиться в произвольной позиции или не находиться нигде. С модулем `csv` вам вообще не придется менять свой код. Более того, поскольку запятая считается разделителем по умолчанию, его даже не нужно указывать:

```
>>> results2 = [fields for fields in csv.reader(open("temp_data_01.csv",
     newline=''))]
>>> results2
[['Notes', 'State', 'State Code', 'Month Day, Year', 'Month Day, Year Code',
     'Avg Daily Max Air Temperature (F)', 'Record Count for Daily Max Air
     Temp (F)', 'Min Temp for Daily Max Air Temp (F)', 'Max Temp for Daily
     Max Air Temp (F)', 'Avg Daily Min Air Temperature (F)', 'Record Count
     for Daily Min Air Temp (F)', 'Min Temp for Daily Min Air Temp (F)', 'Max
     Temp for Daily Min Air Temp (F)', 'Avg Daily Max Heat Index (F)',
     'Record Count for Daily Max Heat Index (F)', 'Min for Daily Max Heat
     Index (F)', 'Max for Daily Max Heat Index (F)', 'Daily Max Heat Index
     (F) % Coverage'], ['', 'Illinois', '17', 'Jan 01, 1979', '1979/01/01',
     '17.48', '994', '6.00', '30.50', '2.89', '994', '-13.60', '15.80',
     'Missing', '0', 'Missing', 'Missing', '0.00%'], ['', 'Illinois', '17',
     'Jan 02, 1979', '1979/01/02', '4.64', '994', '-6.40', '15.80', '-9.03',
     '994', '-23.60', '6.60', 'Missing', '0', 'Missing', 'Missing', '0.00%'],
     ['', 'Illinois', '17', 'Jan 03, 1979', '1979/01/03', '11.05', '994', '-
     0.70', '24.70', '-2.17', '994', '-18.30', '12.90', 'Missing', '0',
     'Missing', 'Missing', '0.00%'], ['', 'Illinois', '17', 'Jan 04, 1979',
     '1979/01/04', '9.51', '994', '0.20', '27.60', '-0.43', '994', '-16.30',
     '16.30', 'Missing', '0', 'Missing', 'Missing', '0.00%'], ['',
     'Illinois', '17', 'May 15, 1979', '1979/05/15', '68.42', '994', '61.00',
     '75.10', '51.30', '994', '43.30', '57.00', 'Missing', '0', 'Missing',
     'Missing', '0.00%'], ['', 'Illinois', '17', 'May 16, 1979', '1979/05/
     16', '70.29', '994', '63.40', '73.50', '48.09', '994', '41.10', '53.00',
     'Missing', '0', 'Missing', 'Missing', '0.00%'], ['', 'Illinois', '17',
     'May 17, 1979', '1979/05/17', '75.34', '994', '64.00', '80.50', '50.84',
     '994', '44.30', '55.70', '82.60', '2', '82.40', '82.80', '0.20%'], ['',
     'Illinois', '17', 'May 18, 1979', '1979/05/18', '79.13', '994', '75.50',
     '82.10', '55.68', '994', '50.00', '61.10', '81.42', '349', '80.20',
     '83.40', '35.11%'], ['', 'Illinois', '17', 'May 19, 1979', '1979/05/19',
     '74.94', '994', '66.90', '83.10', '58.59', '994', '50.90', '63.20',
     '82.87', '78', '81.60', '85.20', '7.85%']]
```

Обратите внимание на то, что лишние кавычки были удалены, а все значения полей с запятыми остались без изменений — и все это без включения новых символов в команду.

БЫСТРАЯ ПРОВЕРКА: КАВЫЧКИ

Подумайте, как бы вы подошли к проблемам обработки полей в кавычках и внутренних символов-разделителей без библиотеки `csv`. Какая задача была бы проще: кавычки или внутренние разделители?

21.2.5. Чтение csv-файла как списка словарей

В предыдущих примерах строка данных возвращается в виде списка полей. Во многих случаях такой результат работает хорошо, но иногда бывает удобнее получать данные в виде словарей, у которых имя поля является ключом. Для таких случаев в библиотеке `csv` имеется класс `DictReader`, который получает список полей в параметре или читает их из первой строки данных. Если вы хотите открыть данные с использованием `DictReader`, код принимает следующий вид:

```
>>> results = [fields for fields in csv.DictReader(open("temp_data_01.csv",
     newline=''))]
>>> results[0]
OrderedDict([('Notes', ''), ('State', 'Illinois'), ('State Code', '17'),
     ('Month Day, Year', 'Jan 01, 1979'), ('Month Day, Year Code', '1979/01/
     01'), ('Avg Daily Max Air Temperature (F)', '17.48'), ('Record Count for
     Daily Max Air Temp (F)', '994'), ('Min Temp for Daily Max Air Temp (F)',
     '6.00'), ('Max Temp for Daily Max Air Temp (F)', '30.50'), ('Avg Daily
     Min Air Temperature (F)', '2.89'), ('Record Count for Daily Min Air Temp
     (F)', '994'), ('Min Temp for Daily Min Air Temp (F)', '-13.60'), ('Max
     Temp for Daily Min Air Temp (F)', '15.80'), ('Avg Daily Max Heat Index
     (F)', 'Missing'), ('Record Count for Daily Max Heat Index (F)', '0'),
     ('Min for Daily Max Heat Index (F)', 'Missing'), ('Max for Daily Max
     Heat Index (F)', 'Missing'), ('Daily Max Heat Index (F) % Coverage',
     '0.00%')])
```

Следует заметить, что `csv.DictReader` возвращает `OrderedDict`, поэтому поля сохраняют исходный порядок. Несмотря на некоторые отличия в представлении, поля по-прежнему ведут себя как словари:

```
>>> results[0]['State']
'Illinois'
```

Если данные особенно сложны и вам требуется обрабатывать конкретные поля, с `DictReader` вам будет намного проще получить правильное поле; кроме того, этот класс делает ваш код чуть более понятным. И наоборот, при достаточно большом объеме данных необходимо помнить, что у `DictReader` чтение того же объема данных может занимать в два раза больше времени.

21.3. Файлы Excel

Еще один стандартный формат, который рассматривается в этой главе, — файлы Excel — используется Microsoft Excel для хранения электронных таблиц. Я привожу здесь описание файлов Excel, потому что программы работают с ними практически так же, как с файлами с разделителями. Собственно, поскольку Excel умеет читать и записывать файлы CSV, самый быстрый и простой способ извлечения данных из файла электронной таблицы Excel часто заключается в том, чтобы открыть файл

в Excel, а затем сохранить его в формате CSV. Тем не менее эта процедура не всегда имеет смысл, особенно если вы работаете с множеством файлов. Хотя теоретически процесс открытия и сохранения каждого файла в формате CSV можно автоматизировать, вероятно, быстрее будет работать с файлами Excel напрямую.

Подробное обсуждение файлов электронных таблиц с их возможностями по сохранению нескольких таблиц в одном файле, макросов и различных средств форматирования выходит за рамки книги. Вместо этого в этом разделе мы рассмотрим пример чтения простого файла с одной таблицей — просто для извлечения данных.

Как выясняется, в стандартной библиотеке Python нет модуля для чтения или записи файлов Excel. Чтобы прочитать файлы в этом формате, необходимо установить внешний модуль. К счастью, есть несколько модулей для решения этой задачи. В нашем примере будет использован модуль OpenPyXL, доступный в репозитории пакетов Python. Его можно установить следующей командой:

```
$pip install openpyxl
```

А вот как выглядят приводившиеся выше данные в формате электронной таблицы:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	C
1	Notes	State	State Code	Month Day, Year	Month Day, Year Code	Avg Daily M	Record Co	Min Temp	Max Temp	Avg Daily M	Record Co	Min for Dai	Max for Da	Daily Max Heat In	
2		Illinois	17	Jan 01, 1979	1979/01/01	17.48	994	6	30.5	Missing	0	Missing	Missing	0.00%	
3		Illinois	17	Jan 02, 1979	1979/01/02	4.64	994	-6.4	15.8	Missing	0	Missing	Missing	0.00%	
4		Illinois	17	Jan 03, 1979	1979/01/03	11.05	994	-0.7	24.7	Missing	0	Missing	Missing	0.00%	
5		Illinois	17	Jan 04, 1979	1979/01/04	9.51	994	0.2	27.6	Missing	0	Missing	Missing	0.00%	
6		Illinois	17	May 15, 1979	1979/05/15	68.42	994	61	75.1	Missing	0	Missing	Missing	0.00%	
7		Illinois	17	May 16, 1979	1979/05/16	70.29	994	63.4	73.5	Missing	0	Missing	Missing	0.00%	
8		Illinois	17	May 17, 1979	1979/05/17	75.34	994	64	80.5	82.6	2	82.4	82.8	0.20%	
9		Illinois	17	May 18, 1979	1979/05/18	79.13	994	75.5	82.1	81.42	349	80.2	83.4	35.11%	
10		Illinois	17	May 19, 1979	1979/05/19	74.94	994	66.9	83.1	82.87	78	81.6	85.2	7.85%	
11															
12															
13															

Прочитать этот файл несложно, но это все равно потребует большей работы по сравнению с файлами CSV. Сначала нужно загрузить книгу, затем получить конкретный лист, затем перебрать строки — и только после этого можно извлечь значения ячеек. Примерный код чтения электронной таблицы может выглядеть так:

```
>>> from openpyxl import load_workbook
>>> wb = load_workbook('temp_data_01.xlsx')
>>> results = []
>>> ws = wb.worksheets[0]
>>> for row in ws.iter_rows():
...     results.append([cell.value for cell in row])
...
>>> print(results)
[['Notes', 'State', 'State Code', 'Month Day, Year', 'Month Day, Year Code',
     'Avg Daily Max Air Temperature (F)', 'Record Count for Daily Max Air
     Temp (F)', 'Min Temp for Daily Max Air Temp (F)', 'Max Temp for Daily
     Max Air Temp (F)', 'Avg Daily Max Heat Index (F)', 'Record Count for
     Daily Max Heat Index (F)', 'Min for Daily Max Heat Index (F)', 'Max for
     Daily Max Heat Index (F)', 'Daily Max Heat Index (F) % Coverage'],
     [None, 'Illinois', 17, 'Jan 01, 1979', '1979/01/01', 17.48, 994, 6,
     30.5, 'Missing', 0, 'Missing', 'Missing', '0.00%'], [None, 'Illinois',
     17, 'Jan 02, 1979', '1979/01/02', 4.64, 994, -6.4, 15.8, 'Missing', 0,
     'Missing', 'Missing', '0.00%'], [None, 'Illinois', 17, 'Jan 03, 1979',
```

```
'1979/01/03', 11.05, 994, -0.7, 24.7, 'Missing', 0, 'Missing',
'Missing', '0.00%'], [None, 'Illinois', 17, 'Jan 04, 1979', '1979/01/
04', 9.51, 994, 0.2, 27.6, 'Missing', 0, 'Missing', 'Missing', '0.00%'],
[None, 'Illinois', 17, 'May 15, 1979', '1979/05/15', 68.42, 994, 61,
75.1, 'Missing', 0, 'Missing', 'Missing', '0.00%'], [None, 'Illinois',
17, 'May 16, 1979', '1979/05/16', 70.29, 994, 63.4, 73.5, 'Missing', 0,
'Missing', 'Missing', '0.00%'], [None, 'Illinois', 17, 'May 17, 1979',
'1979/05/17', 75.34, 994, 64, 80.5, 82.6, 2, 82.4, 82.8, '0.20%'],
[None, 'Illinois', 17, 'May 18, 1979', '1979/05/18', 79.13, 994, 75.5,
82.1, 81.42, 349, 80.2, 83.4, '35.11%'], [None, 'Illinois', 17, 'May 19,
1979', '1979/05/19', 74.94, 994, 66.9, 83.1, 82.87, 78, 81.6, 85.2,
'7.85%']]
```

Этот код возвращает те же результаты, как и намного более простой код для csv-файла. Неудивительно, что код чтения электронной таблицы получается более сложным, потому что электронные таблицы сами по себе являются намного более сложными объектами. Также всегда необходимо понимать, как данные хранятся в электронной таблице. Если таблица содержит форматирование, важное для вас, если метки должны игнорироваться или обрабатываться иначе, если программа должна обрабатывать формулы и ссылки — вам придется более серьезно разбираться в том, как обрабатываются эти элементы, и писать более сложный код.

С электронными таблицами также часто возникают другие проблемы. На момент написания книги электронные таблицы обычно ограничивались миллионом записей. Хотя это значение кажется очень большим, все чаще приходится иметь дело с наборами данных большего размера. Кроме того, электронные таблицы иногда автоматически применяют неудобное форматирование. Одна компания, в которой я работала, использовала номера деталей из цифры и как минимум одной буквы, за которой следовала комбинация цифр и букв. Таким образом, номер детали мог иметь вид 1E20. Многие электронные таблицы автоматически интерпретируют 1E20 как научную запись числа и сохраняют его как 1.00E+20 (10 в 20-й степени), а 1F20 оставляют в виде строки. По какой-то причине предотвратить подобные проблемы оказывается сложно, и особенно с большими наборами данных проблема обнаруживается намного дальше в конвейере обработки (если вообще обнаруживается). Из-за этого я рекомендую по возможности использовать CSV и файлы с разделителями. Пользователи обычно могут сохранить электронную таблицу в формате CSV, поэтому обычно не нужно создавать себе лишние сложности и проблемы с форматированием, присущие электронным таблицам.

21.4. Очистка данных

Одна из типичных проблем, встречающихся при обработке текстовых файлов данных, — грязные данные. Под *грязными* я имею в виду, что данные могут преподносить всевозможные сюрпризы: значения null, значения, не подходящие для вашей кодировки, или дополнительные пропуски. Данные также могут храниться без сортировки или в порядке, затрудняющем обработку. В таких случаях приходится применять процесс, который называется *очисткой данных*.

21.4.1. Очистка данных

В очень простом примере очистки данных требуется обработать файл, экспортированный из электронной таблицы или другой финансовой программы. Столбцы, связанные с обработкой финансовых данных, могут содержать знаки процента и денежные знаки (`%`, `$`, `£` и `?`), а также дополнительные группировки с использованием точек и запятых. Данные из других источников могут содержать другие сюрпризы, которые усложнят обработку, если не выявить их заранее. Еще раз взгляните на температурные данные, которые уже встречались вам раньше. Первая строка данных выглядит так:

```
[None, 'Illinois', 17, 'Jan 01, 1979', '1979/01/01', 17.48, 994, 6, 30.5,
    2.89, 994, -13.6, 15.8, 'Missing', 0, 'Missing', 'Missing', '0.00%']
```

Некоторые столбцы, такие как `'State'` (поле 2) и `'Notes'` (поле 1), очевидно содержат текст, и с ними вряд ли можно что-то сделать. Также имеются два поля данных в разных форматах; вполне возможно, что вы захотите провести вычисления с датами — например, для изменения порядка данных и группировки строк по месяцам или дням или для вычисления промежутка времени, разделяющего две строки.

Остальные поля содержат разные типы чисел; температуры представлены дробными числами, а количество записей — целыми. Однако обратите внимание на то, что с тепловыми индексами есть некоторая неоднозначность: если значение поля `'Max Temp for Daily Max Air Temp (F)'` меньше 80, вместо значений полей теплового индекса указывается строка `'Missing'`, а количество записей равно 0. Также обратите внимание на то, что поле `'Daily Max Heat Index (F) % Coverage'` выражено в процентах от количества температурных записей, которые также пригодны для наличия теплового индекса. Оба обстоятельства создадут проблемы, если вы хотите проводить вычисления со значениями этих полей, потому что как `'Missing'`, так и любое число, завершающееся знаком `%`, будет интерпретироваться как строка, а не как число.

Такая очистка данных может производиться на разных этапах процесса. Довольно часто я предпочитаю проводить очистку данных при чтении из файла, чтобы `'Missing'` можно было заменить значением None или пустой строкой в процессе обработки. Также можно оставить строки `'Missing'` на месте и написать код, который блокирует выполнение математических операций со значением, если оно равно `'Missing'`.

ПОПРОБУЙТЕ САМИ: ОЧИСТКА ДАННЫХ

Как бы вы поступили с полями с возможными значениями `'Missing'` в математических вычислениях? Сможете ли вы написать фрагмент кода, вычисляющий среднее значение по одному из таких столбцов?

Что бы вы сделали в конце с колонкой, вычисляющей среднее значение, чтобы можно было также сообщить средний охват? Будет ли решение этой проблемы, по-вашему мнению, вообще связано с обработкой значения `'Missing'`?

21.4.2. Сортировка

Как упоминалось ранее, данные в текстовом файле часто бывает полезно отсортировать перед обработкой. Сортировка данных упрощает поиск и обработку дубликатов, и она также позволит сгруппировать взаимосвязанные записи для ускорения или упрощения обработки. Однажды я получила файл с 20 миллионами атрибутов и значений, и заранее не известное их количество нужно было проверить по главному списку кодов товаров. Сортировка записей по идентификатору товара существенно упростила выборку атрибутов любого товара. Конкретный способ сортировки зависит от размера файла данных относительно доступной памяти и сложности сортировки. Если все строки файла легко помещаются в памяти, проще всего будет прочитать все строки в список и воспользоваться методом `sort` списка:

```
>>> lines = open("datafile").readlines()
>>> lines.sort()
```

Также можно воспользоваться функцией `sorted()` — например, `sorted_lines = sorted(lines)`. Функция сохраняет порядок строк из исходного списка, что обычно излишне. Недостаток функции `sorted()` заключается в том, что она создает новую копию списка. Процесс занимает чуть больше времени и занимает вдвое больше памяти, что может быть более серьезной проблемой.

Если размер набора данных превышает объем памяти, а сортировка очень проста (например, по легко извлекаемому полю), возможно, будет проще выполнить предварительную обработку данных — например, командой UNIX `sort`:

```
$ sort data > data.srt
```

В любом случае сортировка может осуществляться в обратном порядке, а в качестве ключа могут использоваться произвольные значения (не обязательно начало строки). В таких случаях необходимо изучить документацию программы сортировки, которую вы будете использовать. В простом примере на языке Python сортировка строк будет производиться без учета регистра символов. Для этого методу `sort` передается ключевая функция, которая преобразует элемент к нижнему регистру перед сравнением:

```
>>> lines.sort(key=str.lower)
```

В этом примере используется лямбда-функция для игнорирования первых пяти символов каждой строки:

```
>>> lines.sort(key=lambda x: x[5:])
```

Использование ключевых функций для определения поведения сортировки в Python чрезвычайно удобно, но вы должны помнить, что ключевая функция часто вызывается в процессе сортировки, так что слишком сложная ключевая функция может обернуться серьезной потерей производительности, особенно при большом наборе данных.

21.4.3. Проблемы и ловушки очистки данных

Похоже, «грязных» данных не меньше, чем источников и вариантов использования этих данных. В ваших данных всегда имеются дефекты с самыми разнообразными последствиями — от снижения точности обработки до невозможности даже загрузки данных. В результате я не смогу привести исчерпывающий список проблем, с которыми вы можете столкнуться, и их возможных решений. Возможно только дать несколько общих рекомендаций.

- *Остерегайтесь пропусков и null-символов.* Символы-пропуски не видны, но это не значит, что они не могут создать проблем. Лишние пропуски в начале и в конце строк данных, лишние пропуски рядом с отдельными полями, табуляции вместо пробелов (и наоборот) — все это может сделать загрузку данных и обработку более хлопотной, причем проблемы не всегда очевидны с первого взгляда. Аналогичным образом текстовые файлы с null-символами (ASCII 0) могут выглядеть нормально при просмотре, но вызывать сбои при загрузке и обработке.
- *Остерегайтесь знаков препинания.* Знаки препинания всегда могут быть проблемой. Лишние запятые и точки могут нарушить структуру формата CSV и процесс обработки числовых полей; неэкранированные или непарные кавычки тоже могут создать путаницу.
- *Разбивайте процесс на этапы и отлаживайте их.* Проблему будет проще отлаживать, если каждый этап изолирован от других; это означает, что каждую операцию следует размещать в отдельной строке, по возможности приводить более подробные объяснения и использовать больше переменных. Тем не менее результат того стоит. Во-первых, все возникающие исключения будет легче понять, к тому же процесс отладки станет проще, какое бы средство вы ни использовали (команды `print`, журнал или отладчик Python). Также может быть полезно сохранять данные после каждого этапа и сократить файл до нескольких строк, в которых происходит ошибка.

21.5. Запись файлов данных

В последней части процесса ETL преобразованные данные могут сохраняться в базе данных (глава 22), но часто данные записываются в файлы. Эти файлы затем используются в качестве входных данных других приложений, а также для анализа другими людьми или приложениями. Обычно у вас имеется конкретная спецификация файла с указанием того, какие поля данных должны быть включены в файл, какие имена им присваиваются, какой формат и ограничения устанавливаются для каждого поля и т. д.

21.5.1. CSV и другие файлы с разделителями

Пожалуй, самой простой задачей является запись данных в файлы CSV. Так как вы уже загрузили, разобрали, почистили и преобразовали данные, вряд ли вы

столкнетесь с какими-либо нерешенными проблемами в самих данных. И снова модуль `csv` из стандартной библиотеки Python существенно упростит вашу работу.

Запись файлов с разделителями при использовании модуля `csv` практически противоположна процессу чтения. И снова необходимо указать используемый разделитель, а модуль `csv` позаботится о ситуациях с присутствием символа-разделителя в значениях полей:

```
>>> temperature_data = [['State', 'Month Day, Year Code', 'Avg Daily Max Air
     Temperature (F)', 'Record Count for Daily Max Air Temp (F)'],
     ['Illinois', '1979/01/01', '17.48', '994'], ['Illinois', '1979/01/02',
     '4.64', '994'], ['Illinois', '1979/01/03', '11.05', '994'], ['Illinois',
     '1979/01/04', '9.51', '994'], ['Illinois', '1979/05/15', '68.42',
     '994'], ['Illinois', '1979/05/16', '70.29', '994'], ['Illinois', '1979/
     05/17', '75.34', '994'], ['Illinois', '1979/05/18', '79.13', '994'],
     ['Illinois', '1979/05/19', '74.94', '994']]
>>> csv.writer(open("temp_data_03.csv", "w",
     newline='')).writerows(temperature_data)
```

Этот код создает следующий файл:

```
 State,"Month Day, Year Code",Avg Daily Max Air Temperature (F),Record Count
     for Daily Max Air Temp (F)
Illinois,1979/01/01,17.48,994
Illinois,1979/01/02,4.64,994
Illinois,1979/01/03,11.05,994
Illinois,1979/01/04,9.51,994
Illinois,1979/05/15,68.42,994
Illinois,1979/05/16,70.29,994
Illinois,1979/05/17,75.34,994
Illinois,1979/05/18,79.13,994
Illinois,1979/05/19,74.94,994
```

Как и при чтении из файла в формате CSV, вы можете записывать словари вместо списков при использовании `DictWriter`. Если вы собираетесь использовать `DictWriter`, следует учитывать пару моментов. Имена полей должны быть заданы в списке при создании объекта, и вы можете использовать метод `writeheader` объекта `DictWriter` для записи заголовка в начало файла. Допустим, вы используете те же данные, что и прежде, но в формате словаря:

```
{'State': 'Illinois', 'Month Day, Year Code': '1979/01/01', 'Avg Daily Max
     Air Temperature (F)': '17.48', 'Record Count for Daily Max Air Temp
     (F)': '994'}
```

При помощи объекта `DictWriter` из модуля `csv` можно записать каждую строку (словарь) в правильные поля файла CSV:

```
>>> fields = ['State', 'Month Day, Year Code', 'Avg Daily Max Air Temperature
    (F)', 'Record Count for Daily Max Air Temp (F)']
>>> dict_writer = csv.DictWriter(open("temp_data_04.csv", "w"),
    fieldnames=fields)
>>> dict_writer.writeheader()
```

```
>>> dict_writer.writerows(data)
>>> del dict_writer
```

21.5.2. Запись файлов Excel

Как и следовало ожидать, запись файлов электронных таблиц имеет много общего с их чтением. Необходимо создать книгу или файл электронной таблицы, затем создать лист (или несколько листов) и, наконец, записать данные в соответствующие ячейки. Создание новой электронной таблицы из файла данных CSV может выглядеть так:

```
>>> from openpyxl import Workbook
>>> data_rows = [fields for fields in csv.reader(open("temp_data_01.csv"))]
>>> wb = Workbook()
>>> ws = wb.active
>>> ws.title = "temperature data"
>>> for row in data_rows:
...     ws.append(row)
...
>>> wb.save("temp_data_02.xlsx")
```

Также возможно добавить форматирование в ячейки при записи их в файл. За дополнительной информацией о добавлении форматирования обращайтесь к документации `xlswriter`.

21.5.3. Упаковка файлов данных

Если вы используете несколько взаимосвязанных файлов данных или ваши файлы особенно велики, возможно, стоит упаковать их в сжатый архив. В настоящее время используются разные форматы архивов, формат zip сохраняет популярность и практически общедоступен для пользователей на любых платформах. О том, как создавать zip-файлы из файлов данных, рассказано в главе 20.

ПРАКТИЧЕСКАЯ РАБОТА 21: МЕТЕОРОЛОГИЧЕСКИЕ НАБЛЮДЕНИЯ

Файл погодных данных, приведенный в этой главе, упорядочен по месяцам и затем по округам для штата Иллинойс с 1979 по 2011 год. Напишите код, который обрабатывает этот файл, для извлечения данных Чикаго (Chicago, Cook County) в один файл CSV или файл электронной таблицы. При этом строки `'Missing'` должны заменяться пустыми строками, а проценты должны преобразовываться в дробные величины. Также подумайте о том, какие поля содержат повторяющуюся информацию (а следовательно, могут быть опущены или сохранены в другом месте). Чтобы убедиться в том, что все сделано правильно, загрузите файл в приложении электронной таблицы. Решение можно загрузить в архиве исходного кода книги.

Итоги

- ETL — процесс получения данных в одном формате, проверки их целостности и перевода в формат, который вы можете использовать. ETL является одним из основных этапов большинства задач обработки данных.
- При работе с текстовыми файлами могут возникнуть проблемы с кодировкой, но Python помогает решить некоторые проблемы кодирования при загрузке файлов.
- Файлы с разделителями или файлы CSV пользуются наибольшим распространением и работать с ними лучше всего при помощи модуля `csv`.
- Файлы электронных таблиц могут быть сложнее файлов CSV, но работа с ними строится по тем же принципам.
- Символы денежних единиц, знаки препинания и null-символы — основные источники проблем при очистке данных; будьте внимательны с ними.
- Предварительная сортировка файлов данных может ускорить другие этапы обработки данных.

22 Передача данных по сети

Эта глава охватывает следующие темы:

- ✓ Получение файлов через FTP/SFTP, SSH/SCP и HTTPS
- ✓ Получение данных через API
- ✓ Форматы файлов структурированных данных: JSON и XML
- ✓ Извлечение данных

В главе 21 мы рассмотрели работу с текстовыми файлами данных. В этой главе мы используем Python для передачи файлов данных по Сети. В некоторых случаях это могут быть текстовые файлы или файлы электронных таблиц, как упоминалось в главе 21, но в других случаях они могут храниться в более структурированных форматах и предоставляться программными интерфейсами (API) REST или SOAP. Иногда для получения данных приходится извлекать их с сайта. В этой главе обсуждаются все эти ситуации, а также демонстрируются некоторые стандартные сценарии использования.

22.1. Получение файлов

Прежде чем что-то сделать с файлом данных, его необходимо как-то получить. Иногда это делается очень просто: например, один zip-архив загружается вручную или файлы отправляются на вашу машину из другого места. Достаточно часто процесс оказывается более сложным: например, с удаленного сервера нужно загрузить множество файлов, файлы должны передаваться с регулярным интервалом или процесс загрузки достаточно сложен, чтобы его хотелось проводить вручную. Во всех перечисленных случаях вы, вероятно, предпочтете автоматизировать получение данных в программах Python.

Прежде всего я хочу четко указать, что использование сценария Python не единственный (и не всегда лучший) способ загрузки файлов. В следующей врезке я более подробно объясняю факторы, которые учитываю, принимая решение о том, стоит

ли использовать сценарии Python для загрузки файлов. Если предположить, что использование Python имеет смысл в вашей конкретной ситуации, в этом разделе продемонстрированы некоторые стандартные паттерны, которые вы можете применять в своей работе.

СТОИТ ЛИ ИСПОЛЬЗОВАТЬ PYTHON ДЛЯ ЗАГРУЗКИ ФАЙЛОВ?

Хотя Python может очень хорошо подойти для загрузки файлов, такой способ не всегда является лучшим. Принимая решение, следует учитывать два обстоятельства:

- *Нет ли более простых вариантов?* В зависимости от операционной системы и вашего опыта может оказаться, что простые сценарии и инструменты командной строки проще и удобнее. Если эти инструменты вам недоступны или вы недостаточно уверенно владеете ими (или ими слабо владеют те люди, которые будут заниматься сопровождением сценариев), можно рассмотреть возможность применения сценария Python.
- *Процесс загрузки особенно сложен или тесно связан с обработкой?* Такие ситуации нежелательны, но они встречаются. Сейчас я руководствуюсь правилом, что если сценарий командной строки занимает больше нескольких строк или мне приходится напряженно думать о том, как что-то сделать в сценарии командной строки, вероятно, пора переключаться на Python.

22.1.1. Использование Python для загрузки файлов с сервера FTP

Протокол FTP (File Transfer Protocol) существует уже давно, но он до сих пор остается простым и удобным механизмом передачи файлов в тех случаях, когда безопасность не критична. Чтобы подключиться к серверу FTP на Python, можно воспользоваться модулем `ftplib` из стандартной библиотеки. Процедура проста и прямолинейна: создайте объект FTP, подключитесь к серверу, а затем введите свои учетные данные — имя пользователя и пароль (чаще имя пользователя anonymous и пустой пароль).

Чтобы продолжить работу с метеорологическими данными, можно подключиться к FTP-серверу Национального управления по исследованию океанов и атмосферы (NOAA):

```
>>> import ftplib
>>> ftp = ftplib.FTP('tgftp.nws.noaa.gov')
>>> ftp.login()
'230 Login successful.'
```

После подключения объект `ftp` используется для получения списка и смены каталогов:

```
>>> ftp.cwd('data')
'250 Directory successfully changed.'
>>> ftp.nlst()
['climate', 'fnmoc', 'forecasts', 'hurricane_products', 'ls_SS_services',
     'marine', 'nsd_bbsss.txt', 'nsd_cccc.txt', 'observations', 'products',
     'public_statement', 'raw', 'records', 'summaries', 'tampa',
     'watches_warnings', 'zonecatalog.curr', 'zonecatalog.curr.tar']
```

Скажем, после этого можно получить последний отчет METAR для международного аэропорта О'Хара в Чикаго:

```
>>> x = ftp.retrbinary('RETR observations/metar/decoded/KORD.TXT',
     open('KORD.TXT', 'wb').write)
'226 Transfer complete.'
```

Методу `ftp.retrbinary` передается путь к файлу на удаленном сервере и метод обработки данных этого файла на вашей стороне — в данном случае метод `write` файла, открытого для двоичной записи. Просмотрев файл KORD.TXT, вы увидите, что он содержит загруженные данные:

```
CHICAGO O'HARE INTERNATIONAL, IL, United States (KORD) 41-59N 087-55W 200M
Jan 01, 2017 - 09:51 PM EST / 2017.01.02 0251 UTC
Wind: from the E (090 degrees) at 6 MPH (5 KT):0
Visibility: 10 mile(s):0
Sky conditions: mostly cloudy
Temperature: 33.1 F (0.6 C)
Windchill: 28 F (-2 C):1
Dew Point: 21.9 F (-5.6 C)
Relative Humidity: 63%
Pressure (altimeter): 30.14 in. Hg (1020 hPa)
Pressure tendency: 0.01 inches (0.2 hPa) lower than three hours ago
ob: KORD 020251Z 09005KT 10SM SCT150 BKN250 01/M06 A3014 RMK AO2 SLP214
     T00061056 58002
cycle: 3
```

Также модуль `ftplib` может использоваться для подключения к серверам с использованием шифрования TLS; для этого укажите FTP_TLS вместо FTP:

```
ftp = ftplib.FTPTLS('tgftp.nws.noaa.gov')
```

22.1.2. Загрузка файлов с использованием SFTP

Если данные требуют более высокого уровня безопасности (например, в корпоративной среде, где по сети передаются бизнес-данные), достаточно часто применяется SFTP — полнофункциональный протокол с поддержкой обращения к файлам, передачи и управления файлами по подключению SSH (Secure Shell). И хотя SFTP означает *SSH File Transfer Protocol* (протокол передачи файлов по SSH), а FTP означает *File Transfer Protocol* (протокол передачи файлов), эти два протокола между собой не связаны. SFTP не является повторной реализацией FTP на базе SSH, это новый протокол, разработанный специально для SSH.

Передача файлов на базе SSH привлекает тем, что протокол SSH уже является фактическим стандартом для обращения к удаленным серверам, а включить поддержку SFTP на сервере несложно (и это часто делается по умолчанию).

В стандартной библиотеке Python нет клиентского модуля SFTP/SCP, но разработанная в сообществе библиотека `paramiko` управляет операциями SFTP и подключениями SSH. Чтобы использовать `paramiko`, проще всего установить библиотеку с помощью `pip`. Если бы упоминавшийся ранее сайт NOAA использовал SFTP

(а он не использует, поэтому следующий код работать не будет!), то эквивалент приведенного выше кода для SFTP выглядел бы так:

```
>>> import paramiko
>>> t = paramiko.Transport((hostname, port))
>>> t.connect(username, password)
>>> sftp = paramiko.SFTPClient.from_transport(t)
```

Также стоит заметить, что хотя модуль `paramiko` поддерживает выполнение команд на удаленном сервере и получение их выходных данных, как и в прямых сеансах `ssh`, он не включает функцию `scp`. Вряд ли вы будете часто огорчаться по поводу отсутствия этой функции; если все, что вам нужно, — это передать один-два файла по подключению `ssh`, проблема обычно быстрее и проще решается утилитой командной строки `scp`.

22.1.3. Передача файлов через HTTP/HTTPS

Последний способ загрузки файлов данных, который будет рассмотрен в этой главе, — получение файлов по подключению HTTP или HTTPS. Вероятно, этот вариант проще всех остальных; фактически данные загружаются с веб-сервера, а поддержка обращения к веб-серверам присутствует практически везде. Как и прежде, в этом случае не обязательно использовать Python. Различные средства командной строки загружают файлы по подключениям HTTP/HTTPS и обладают большинством необходимых возможностей. Два самых популярных инструмента такого рода — `wget` и `curl`. Впрочем, если у вас есть причины выполнять загрузку в коде Python, процесс будет не намного сложнее. Пожалуй, библиотека `requests` предоставляет самые простые и надежные средства для работы с серверами HTTP/HTTPS из кода Python. Кроме того, библиотека `requests` проще всего устанавливается командой `pip install requests`.

После установки `requests` получить файл несложно: импортируйте `requests` и используйте правильную команду HTTP (обычно GET) для подключения к серверу и получения данных.

В следующем примере выполняется загрузка ежемесячных температурных данных для аэропорта Хитроу с 1948 года — этот текстовый файл предоставляется веб-сервером. При желании вы можете ввести URL в своем браузере, загрузить страницу и сохранить ее. Впрочем, если страница велика или загружать приходится слишком много страниц, проще использовать код следующего вида:

```
>>> import requests
>>> response = requests.get("http://www.metoffice.gov.uk/pub/data/weather/uk/
     climate/stationdata/heathrowdata.txt")
```

Ответ содержит разнообразную информацию, включая заголовок, возвращенный веб-сервером; он может пригодиться для отладки, если что-то не работает. Впрочем, из всех частей объекта ответа вас чаще всего будут интересовать возвращаемые данные. Для получения данных необходимо обратиться к свойству `text` ответа,

содержащему тело ответа в формате строки, или к свойству `content`, содержащему тело ответа в формате bytes:

```
>>> print(response.text)
Heathrow (London Airport)
Location 507800E 176700N, Lat 51.479 Lon -0.449, 25m amsl
Estimated data is marked with a * after the value.
Missing data (more than 2 days missing in month) is marked by  ---.
Sunshine data taken from an automatic Kipp & Zonen sensor marked with a #,
     otherwise sunshine data taken from a Campbell Stokes recorder.
   yyyy  mm   tmax    tmin      af    rain     sun
              degC    degC    days      mm   hours
   1948   1    8.9     3.3    ---     85.0    ---
   1948   2    7.9     2.2    ---     26.0    ---
   1948   3   14.2     3.8    ---     14.0    ---
   1948   4   15.4     5.1    ---     35.0    ---
   1948   5   18.1     6.9    ---     57.0    ---
```

Обычно текст ответа записывается в файл для последующей обработки, но в зависимости от ваших целей иногда проводится очистка или даже непосредственная обработка данных.

ПОПРОБУЙТЕ САМИ: ЗАГРУЗКА ФАЙЛА

Допустим, вы работаете с файлом данных из нашего примера и хотите разбить каждую строку на поля; как бы вы это сделали? Какая еще обработка может потребоваться? Попробуйте написать код для загрузки этого файла и вычислить средний ежегодный уровень осадков (rain) или среднюю максимальную и минимальную температуру за каждый код (это более сложная задача).

22.2. Получение данных через API

Механизм получения данных через API весьма распространен; он следует популярной тенденции разделения приложений на службы, взаимодействующие через программные интерфейсы (API). Существуют разные способы взаимодействия API, но, как правило, они работают с обычными протоколами HTTP/HTTPS с использованием стандартных команд HTTP GET, POST, PUT и DELETE. Получение данных таким способом имеет много общего с загрузкой файлов, описанной в разделе 22.1.3, но данные не хранятся в статическом файле. Вместо того чтобы предоставлять статические файлы с данными, приложение обращается с запросом к другому источнику данных, а затем собирает и динамически поставляет данные по запросу.

Хотя существует много вариантов настройки конфигурации API, одним из самых распространенных является REST-совместимый (REpresentational State Transfer) интерфейс, использующий те же протоколы HTTP/HTTPS, что и Всемирная паутина. Количество вариаций в работе API бесконечно, но обычно данные поставляются по запросу GET, который используется браузером для запроса веб-страниц. Когда

вы запрашиваете данные командой GET, нужные параметры выборки данных часто присоединяются к URL-адресу в строке запроса.

Если вы хотите получить данные о текущей погоде на Марсе у марсохода «Кьюриосити», используйте URL-адрес http://marsweather.ingenology.com/v1/latest/?format=json[1]. Параметр строки запроса ?format=json указывает, что информация возвращается в формате JSON (он будет рассмотрен в разделе 22.3.1). Если вас интересует погода на Марсе за конкретный марсианский день — скажем, 155-й, используйте URL-адрес http://marsweather.ingenology.com/v1/archive/?sol=155&format=json. Если вы хотите получить данные о погоде на Марсе в определенном диапазоне земных дат (скажем, за октябрь 2012 года), используйте адрес http://marsweather.ingenology.com/v1/archive/?terrestrial_date_start=2012-10-01&terrestrial_date_end=2012-10-31. Обратите внимание на то, что элементы строки запроса разделены амперсандами (&).

Зная URL-адрес, можно использовать библиотеку requests для получения данных от API и либо обработать их на месте, либо сохранить в файле для последующей обработки. В простейшем варианте это делается точно так же, как при загрузке файла:

```
>>> import requests
>>> response = requests.get("http://marsweather.ingenology.com/v1/latest/
     ?format=json")
>>> response.text
'{"report": {"terrestrial_date": "2017-01-08", "sol": 1573, "ls": 295.0,
    "min_temp": -74.0, "min_temp_fahrenheit": -101.2, "max_temp": -2.0,
    "max_temp_fahrenheit": 28.4, "pressure": 872.0, "pressure_string":
    "Higher", "abs_humidity": null, "wind_speed": null, "wind_direction": "-
    -", "atmo_opacity": "Sunny", "season": "Month 10", "sunrise": "2017-01-
    08T12:29:00Z", "sunset": "2017-01-09T00:45:00Z"}}'
>>> response = requests.get("http://marsweather.ingenology.com/v1/archive/
    ?sol=155&format=json")
>>> response.text
'{"count": 1, "next": null, "previous": null, "results":
    [{"terrestrial_date": "2013-01-18", "sol": 155, "ls": 243.7, "min_temp":
    -64.45, "min_temp_fahrenheit": -84.01, "max_temp": 2.15,
    "max_temp_fahrenheit": 35.87, "pressure": 9.175, "pressure_string":
    "Higher", "abs_humidity": null, "wind_speed": 2.0, "wind_direction":
    null, "atmo_opacity": null, "season": "Month 9", "sunrise": null,
    "sunset": null}]}'
```

Помните о необходимости экранировать пробелы и большинство знаков препинания в параметрах запросов, потому что эти элементы не разрешены в URL-адресах (впрочем, многие браузеры автоматически экранируют символы в URL).

И последний пример: допустим, вы хотите получить данные о преступлениях, совершенных в Чикаго от полудня до часа дня 10 января 2017 года. По правилам этого API диапазон дат в параметрах строки запроса указывается в формате $where date=between <начальная_дата> и <конечная_дата>, где начальная и конечная дата заключаются в кавычки в формате ISO. Таким образом, URL-адрес для получения

[1] Этот сайт (ingenology.com) надежно работал в прошлом, но на момент написания книги он вышел из строя, а его будущее неизвестно.

данных за этот час будет иметь вид https://data.cityofchicago.org/resource/6zsd-86xi.json?$where=date between '2015-01-10T12:00:00' and '2015-01-10T13:00:00'.

В этом примере некоторые символы недопустимы в URL — например, кавычки и пробелы. Это еще одна ситуация, в которой библиотека requests старается упростить задачу для пользователя; перед отправкой URL-адреса библиотека заменяет недопустимые символы. Фактически передаваемый URL-адрес имеет вид https://data.cityofchicago.org/resource/6zsd-86xi.json?$where=date%20between%20%222015-01-10T12:00:00%22%20and%20%222015-01-10T14:00:00%22'.

Обратите внимание: все одинарные кавычки автоматически заменяются кодом `%22`, а все пробелы — кодом `%20`, и вам даже не пришлось об этом задумываться.

ПОПРОБУЙТЕ САМИ: ИСПОЛЬЗОВАНИЕ API

Напишите код для получения данных с городского сайта Чикаго. Просмотрите поля, упоминаемые в результатах, и попробуйте выполнить выборку записей по другому полю в сочетании с диапазоном дат.

22.3. Структурированные форматы данных

Хотя API иногда предоставляют данные в формате простого текста, чаще данные поставляются в структурированных форматах. Два самых популярных формата — JSON и XML. Оба формата состоят из простого текста, но их содержимое структурируется таким образом, чтобы оно было более гибким и позволяло хранить более сложную информацию.

22.3.1. Данные JSON

Формат JSON (аббревиатура JavaScript Object Notation) появился в 1999 году. Он состоит всего из двух структур: пары «ключ–значение», называемые *структурами*, очень похожи на словари Python, а упорядоченные списки значений, называемые *массивами*, очень похожи на списки Python.

Ключами могут быть только строки в двойных кавычках, а значения могут быть строками в двойных кавычках, числами, True, False, null, массивами или объектами. При наличии этих элементов JSON предоставляет упрощенный формат представления данных, легко передаваемый по Сети и легко читаемый человеком. Формат JSON настолько распространен, что во многих языках предусмотрены средства преобразования JSON во внутренние типы данных и обратно. В Python эти средства реализованы в модуле `json`, который стал частью стандартной библиотеки версии 2.6. Все еще доступна исходная версия модуля `simplejson`, находящаяся на внешнем сопровождении. Впрочем, в Python 3 гораздо чаще используется версия из стандартной библиотеки.

Данные, полученные от API марсохода и города Чикаго в разделе 22.2, хранятся в формате JSON. Для отправки данных JSON по сети объект JSON должен быть

сериализован, то есть преобразован в последовательность байтов. Таким образом, хотя пакет данных, полученный от API марсохода и города Чикаго, внешне напоминает JSON, на самом деле это всего лишь представление объекта JSON в виде последовательности байтов. Чтобы преобразовать байтовую строку в реальный объект JSON и перевести в словарь Python, необходимо воспользоваться функцией JSON `loads()`. Например, если вы захотите получить отчет о погоде на Марсе, это можно сделать точно так же, как прежде, но на этот раз данные преобразуются в словарь Python:

```
>>> import json
>>> import requests
>>> response = requests.get("http://marsweather.ingenology.com/v1/latest/
     ?format=json")
>>> weather = json.loads(response.text)
>>> weather
{'report': {'terrestrial_date': '2017-01-10', 'sol': 1575, 'ls': 296.0,
     'min_temp': -58.0, 'min_temp_fahrenheit': -72.4, 'max_temp': 0.0,
     'max_temp_fahrenheit': None, 'pressure': 860.0, 'pressure_string':
     'Higher', 'abs_humidity': None, 'wind_speed': None, 'wind_direction': '-
     -', 'atmo_opacity': 'Sunny', 'season': 'Month 10', 'sunrise': '2017-01-
     10T12:30:00Z', 'sunset': '2017-01-11T00:46:00Z'}}
>>> weather['report']['sol']
1575
```

Обратите внимание на то, что вызов `json.loads()` получает строковое представление объекта JSON и преобразует (или загружает) его в словарь Python. Кроме того, функция `json.load()` читает из любого объекта, сходного с файлом и поддерживающего метод `read`.

Если вы взглянете на представление словаря, приведенное выше, вам будет очень сложно разобраться в нем. *Улучшенное форматирование* делает структуры данных более понятными. В этом вам поможет модуль Python `prettyprint`:

```
>>> from pprint import pprint as pp
>>> pp(weather)
{'report': {'abs_humidity': None,
            'atmo_opacity': 'Sunny',
            'ls': 296.0,
            'max_temp': 0.0,
            'max_temp_fahrenheit': None,
            'min_temp': -58.0,
            'min_temp_fahrenheit': -72.4,
            'pressure': 860.0,
            'pressure_string': 'Higher',
            'season': 'Month 10',
            'sol': 1575,
            'sunrise': '2017-01-10T12:30:00Z',
            'sunset': '2017-01-11T00:46:00Z',
            'terrestrial_date': '2017-01-10',
            'wind_direction': '--',
            'wind_speed': None}
```

Обе функции можно настроить для управления процессом разбора и декодирования исходного формата JSON в объекты Python, но преобразование по умолчанию представлено в табл. 22.1.

Таблица 22.1. Стандартные правила декодирования JSON в Python

JSON	Python
`object`	`dict`
`array`	`list`
`string`	`str`
`number (int)`	`int`
`number (real)`	`float`
`true`	`True`
`false`	`False`
`null`	`None`

ЗАГРУЗКА JSON С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ REQUESTS

В этом разделе мы использовали библиотеку requests для получения данных в формате JSON и последующего преобразования их в объект Python методом `json.loads()`. Этот прием хорошо работает, но поскольку библиотека requests так часто используется именно для этой цели, в библиотеке предусмотрена сокращенная запись: у объекта ответа имеется метод `json()`, который выполняет преобразование за вас. Таким образом, в данном примере вместо

```
>>> weather = json.loads(response.text)
```

можно было бы использовать запись

```
>>> weather = response.json()
```

Результат получится тем же, но этот вариант проще, лучше читается и в большей степени соответствует стилю Python.

Если вы хотите записать формат JSON в файл или сериализовать его в строку, используйте функции `dump()` и `dumps()`, обратные по отношению к `load()` и `loads()`. `json.dump()` получает в параметре объект файла с методом `write()`, а `json.dumps()` возвращает строку. В обоих случаях процесс кодирования в отформатированную строку JSON может настраиваться, но процесс по умолчанию все равно базируется на табл. 22.1. Итак, если вы хотите записать собственный отчет погоды на Марсе в файл JSON, это делается так:

```
>>> outfile = open("mars_data_01.json", "w")
>>> json.dump(weather, outfile)
>>> outfile.close()
>>> json.dumps(weather)
```

```
'{"report": {"terrestrial_date": "2017-01-11", "sol": 1576, "ls": 296.0,
     "min_temp": -72.0, "min_temp_fahrenheit": -97.6, "max_temp": -1.0,
     "max_temp_fahrenheit": 30.2, "pressure": 869.0, "pressure_string":
     "Higher", "abs_humidity": null, "wind_speed": null, "wind_direction": "-
     -", "atmo_opacity": "Sunny", "season": "Month 10", "sunrise": "2017-01-
     11T12:31:00Z", "sunset": "2017-01-12T00:46:00Z"}}'
```

Как видите, весь объект был закодирован в одну строку. И снова будет удобно отформатировать строку в более наглядном виде, как это было сделано раньше с помощью модуля `pprint`. Для этого используйте параметр `indent` функции `dump` или `dumps`:

```
>>> print(json.dumps(weather, indent=2))
{
  "report": {
    "terrestrial_date": "2017-01-10",
    "sol": 1575,
    "ls": 296.0,
    "min_temp": -58.0,
    "min_temp_fahrenheit": -72.4,
    "max_temp": 0.0,
    "max_temp_fahrenheit": null,
    "pressure": 860.0,
    "pressure_string": "Higher",
    "abs_humidity": null,
    "wind_speed": null,
    "wind_direction": "--",
    "atmo_opacity": "Sunny",
    "season": "Month 10",
    "sunrise": "2017-01-10T12:30:00Z",
    "sunset": "2017-01-11T00:46:00Z"
  }
}
```

Однако следует учитывать, что при использовании повторных вызовов `json.dump()` для записи серии объектов в файл результатом будет *серия* действительных объектов в формате JSON, но содержимое файла в целом *не является* действительным объектом в формате JSON, и попытка чтения и разбора всего файла одним вызовом `json.load()` завершится неудачей. Если имеется несколько объектов, которые вам хотелось бы закодировать в один объект JSON, необходимо поместить все эти объекты в список (или еще лучше в объект), а затем закодировать этот элемент в файл.

Если вы хотите сохранить в JSON данные о погоде на Марсе за два и более дня, у вас есть выбор. Можно использовать `json.dump()` по одному разу для каждого объекта, в результате чего будет создан файл, содержащий объекты в формате JSON. Если допустить, что `weather_list` содержит список объектов метеорологических отчетов, код может выглядеть так:

```
>>> outfile = open("mars_data.json", "w")
>>> for report in weather_list:
...     json.dump(weather, outfile)
>>> outfile.close()
```

В этом случае каждая строка должна загружаться как отдельный объект в формате JSON:

```
>>> for line in open("mars_data.json"):
...     weather_list.append(json.loads(line))
```

Другой возможный вариант — размещение списка в одном объекте JSON. Так как у высокоуровневых массивов в JSON существует потенциальная уязвимость безопасности, рекомендуется помещать массивы в словарь:

```
>>> outfile = open("mars_data.json", "w")
>>> weather_obj = {"reports": weather_list, "count": 2}
>>> json.dump(weather, outfile)
>>> outfile.close()
```

При таком подходе для загрузки объекта в формате JSON из файла достаточно одной операции:

```
>>> with open("mars_data.json") as infile:
>>> weather_obj = json.load(infile)
```

Второй способ хорошо подходит для файлов JSON относительно небольшого размера, но для очень больших файлов он далеко не идеален из-за усложнения обработки ошибок и возможной нехватки памяти.

ПОПРОБУЙТЕ САМИ: СОХРАНЕНИЕ ДАННЫХ О ПРЕСТУПЛЕНИЯХ В ФОРМАТЕ JSON

Измените код, написанный в разделе 22.2, для загрузки данных о преступлениях в Чикаго. Преобразуйте загруженные данные из строки в формате JSON в объект Python. Затем посмотрите, удастся ли вам сохранить события преступлений в виде серии разных объектов JSON в одном файле и как один объект JSON в другом файле. Определите, какой код потребуется для загрузки каждого из файлов.

22.3.2. Данные XML

Язык разметки XML (eXtensible Markup Language) появился в конце XX века. В XML, как и в HTML, используется синтаксис тегов в угловых скобках, а элементы вкладываются в другие элементы, образуя древовидную структуру. Предполагалось, что XML будет хорошо читаться как машинами, так и людьми, но часто разметка XML оказывается настолько объемной и сложной, что разобраться в ней человеку оказывается слишком трудно. Тем не менее, поскольку XML может считаться установленным стандартом, данные в формате XML встречаются очень часто. И поскольку XML предназначен для чтения машинами, вполне возможно, что вы захотите преобразовать его в формат, более удобный для работы.

Рассмотрим пример данных XML — в нашем случае это XML-версия погодных данных для Чикаго:

```
<dwml xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://
     www.w3.org/2001/XMLSchema-instance" version="1.0"
     xsi:noNamespaceSchemaLocation="http://www.nws.noaa.gov/forecasts/xml/
     DWMLgen/schema/DWML.xsd">
  <head>
    <product srsName="WGS 1984" concise-name="glance" operational-
mode="official">
      <title>
NOAA's National Weather Service Forecast at a Glance
      </title>
      <field>meteorological</field>
      <category>forecast</category>
      <creation-date refresh-frequency="PT1H">2017-01-08T02:52:41Z</creation-
date>
    </product>
    <source>
      <more-information>http://www.nws.noaa.gov/forecasts/xml/</more-
information>
      <production-center>
Meteorological Development Laboratory
<sub-center>Product Generation Branch</sub-center>
      </production-center>
      <disclaimer>http://www.nws.noaa.gov/disclaimer.html</disclaimer>
      <credit>http://www.weather.gov/</credit>
      <credit-logo>http://www.weather.gov/images/xml_logo.gif</credit-logo>
      <feedback>http://www.weather.gov/feedback.php</feedback>
    </source>
  </head>
  <data>
    <location>
      <location-key>point1</location-key>
      <point latitude="41.78" longitude="-88.65"/>
    </location>
    …
  </data>
</dwml>
```

Приведен только первый раздел документа, большая часть данных опущена. Но даже этот фрагмент наглядно демонстрирует некоторые проблемы, присущие данным XML. В частности, вы видите, насколько «многословен» этот протокол: теги в некоторых случаях занимают больше места, чем содержащиеся в них значения. Этот пример также демонстрирует древовидную структуру данных, типичную для XML, а также типичное использование объемного заголовка с метаданными до начала фактических данных. По шкале сложности файлов данных можно считать, что CSV или файлы с разделителями находятся на простом конце шкалы, а XML — на сложном.

Наконец, файл демонстрирует еще одну особенность XML, которая несколько усложняет извлечение данных. В XML атрибуты могут использоваться для хранения как данных, так и текстовых значений в тегах. Присмотревшись к элементу `point` в конце примера, вы увидите, что элемент `point` не имеет текстового значения.

Он содержит только значения latitude и longitude, включенные непосредственно в тег `<point>`:

```
<point latitude="41.78" longitude="-88.65"/>
```

Безусловно, этот код является действительной разметкой XML, и он может использоваться для хранения данных, но те же данные могут быть (и скорее всего, будут) сохранены в виде:

```
<point>
   <latitude>41.78</ latitude >
   <longitude>-88.65</longitude>
</point>
```

Таким образом, вы не знаете, как должен обрабатываться каждый конкретный блок данных, без тщательного анализа данных или документа спецификации.

Подобные сложности затрудняют простое извлечение данных из XML. В Python предусмотрено несколько способов работы с XML. Стандартная библиотека Python включает модули для разбора и обработки данных XML, но их нельзя назвать удобными для простого извлечения данных.

Самое удобное средство для извлечения простых данных я нашла в библиотеке `xmltodict`, которая разбирает данные XML и возвращает словарь, отражающий структуру дерева. Собственно, «за кулисами» она использует XML-парсер expat стандартной библиотеки, разбирает документ XML в дерево и использует это дерево для создания словаря. В результате `xmltodict` может обработать любую разметку, с которой справляется парсер, а также может взять словарь и при необходимости «вернуть» его в XML, что делает его очень удобным инструментом. За несколько лет использования это решение обеспечивало практически все мои потребности в работе с XML. Чтобы установить `xmltodict`, выполните команду `pip install xmltodict`.

Чтобы преобразовать XML в словарь, импортируйте `xmltodict` и используйте метод `parse` для строки в формате XML:

```
>>> import xmltodict
>>> data = xmltodict.parse(open("observations_01.xml").read())
```

В данном случае для компактности содержимое файла напрямую передается методу `parse`. После разбора этот объект данных содержит упорядоченный словарь с такими же значениями, как если бы он был загружен из следующей разметки JSON:

```
{
    "dwml": {
        "@xmlns:xsd": "http://www.w3.org/2001/XMLSchema",
        "@xmlns:xsi": "http://www.w3.org/2001/XMLSchema-instance",
        "@version": "1.0",
        "@xsi:noNamespaceSchemaLocation": "http://www.nws.noaa.gov/forecasts/
 xml/DWMLgen/schema/DWML.xsd",
        "head": {
            "product": {
                "@srsName": "WGS 1984",
                "@concise-name": "glance",
```

```
                "@operational-mode": "official",
                "title": "NOAA's National Weather Service Forecast at a Glance",
                "field": "meteorological",
                "category": "forecast",
                "creation-date": {
                    "@refresh-frequency": "PT1H",
                    "#text": "2017-01-08T02:52:41Z"
                }
            },
            "source": {
                "more-information": "http://www.nws.noaa.gov/forecasts/xml/",
                "production-center": {
                    "sub-center": "Product Generation Branch",
                    "#text": "Meteorological Development Laboratory"
                },
                "disclaimer": "http://www.nws.noaa.gov/disclaimer.html",
                "credit": "http://www.weather.gov/",
                "credit-logo": "http://www.weather.gov/images/xml_logo.gif",
                "feedback": "http://www.weather.gov/feedback.php"
            }
        },
        "data": {
            "location": {
                "location-key": "point1",
                "point": {
                    "@latitude": "41.78",
                    "@longitude": "-88.65"
                }
            }
        }
    }
}
```

Обратите внимание: атрибуты были извлечены из тегов, но к ним присоединяется префикс `@`, который указывает, что изначально они были атрибутами своего родительского тега. Если узел XML содержит как текстовое значение, так и вложенный элемент, то в качестве ключа текстового значения используется строка `#text`, как в элементе `"sub-center"` из `"production-center"`.

Как упоминалось ранее, результатом разбора является *упорядоченный словарь* (официально `OrderedDict`), поэтому при выводе результат будет выглядеть так:

```
OrderedDict([('dwml', OrderedDict([('@xmlns:xsd', 'http://www.w3.org/2001/
     XMLSchema'), ('@xmlns:xsi', 'http://www.w3.org/2001/XMLSchema-
     instance'), ('@version', '1.0'), ('@xsi:noNamespaceSchemaLocation',
     'http://www.nws.noaa.gov/forecasts/xml/DWMLgen/schema/DWML.xsd'),
     ('head', OrderedDict([('product', OrderedDict([('@srsName', 'WGS 1984'),
     ('@concise-name', 'glance'), ('@operational-mode', 'official'),
     ('title', "NOAA's National Weather Service Forecast at a Glance"),
     ('field', 'meteorological'), ('category', 'forecast'), ('creation-date',
     OrderedDict([('@refresh-frequency', 'PT1H'), ('#text', '2017-01-
     08T02:52:41Z')]))])])), ('source', OrderedDict([('more-information',
     'http://www.nws.noaa.gov/forecasts/xml/'), ('production-center',
     OrderedDict([('sub-center', 'Product Generation Branch'), ('#text',
```

```
    'Meteorological Development Laboratory')])), ('disclaimer', 'http://
    www.nws.noaa.gov/disclaimer.html'), ('credit', 'http://www.weather.gov/
    '), ('credit-logo', 'http://www.weather.gov/images/xml_logo.gif'),
    ('feedback', 'http://www.weather.gov/feedback.php')]))])), ('data',
    OrderedDict([('location', OrderedDict([('location-key', 'point1'),
    ('point', OrderedDict([('@latitude', '41.78'), ('@longitude', '-
    88.65')]))])), ('#text', '…')]))]))])
```

И хотя представление `OrderedDict` с его списком кортежей выглядит довольно странно, оно ведет себя точно так же, как обычный словарь, если не считать полезной в данном случае гарантии сохранения порядка элементов.

Повторяющиеся элементы превращаются в список. В одном из последующих разделов полной версии файла, приведенного выше, встречается следующий элемент (некоторые элементы опущены):

```
<time-layout >
    <start-valid-time period-name="Monday">2017-01-09T07:00:00-06:00</start-
    valid-time>
    <end-valid-time>2017-01-09T19:00:00-06:00</end-valid-time>
    <start-valid-time period-name="Tuesday">2017-01-10T07:00:00-06:00</start-
     valid-time>
    <end-valid-time>2017-01-10T19:00:00-06:00</end-valid-time>
    <start-valid-time period-name="Wednesday">2017-01-11T07:00:00-06:00</
     start-valid-time>
    <end-valid-time>2017-01-11T19:00:00-06:00</end-valid-time>
</time-layout>
```

Два элемента — `"start-valid-time"` и `"end-valid-time"` — повторяются с чередованием. Каждый из этих двух повторяющихся элементов преобразуется в список в словаре, при этом набор элементов сохраняет правильный порядок:

```
"time-layout":
    {
        "start-valid-time": [
            {
                "@period-name": "Monday",
                "#text": "2017-01-09T07:00:00-06:00"
            },
            {
                "@period-name": "Tuesday",
                "#text": "2017-01-10T07:00:00-06:00"
            },
            {
                "@period-name": "Wednesday",
                "#text": "2017-01-11T07:00:00-06:00"
            }
        ],
        "end-valid-time": [
            "2017-01-09T19:00:00-06:00",
            "2017-01-10T19:00:00-06:00",
            "2017-01-11T19:00:00-06:00"
        ]
    },
```

Так как в Python легко работать со словарями и списками и даже вложенными словарями и списками, модуль `xmltodict` является эффективным инструментом для работы с XML. Я использовала его в последние годы для построения различных документов XML, и у меня ни разу не было никаких проблем.

ПОПРОБУЙТЕ САМИ: ЗАГРУЗКА И РАЗБОР XML

Напишите код для извлечения прогноза погоды в Чикаго в формате XML по адресу http://mng.bz/103V. Затем используйте `xmltodict` для разбора XML в словарь Python и извлечения прогноза максимальной температуры на завтрашний день. Подсказка: чтобы сопоставить периоды времени и значения, сравните значение layout-key первой секции time-layout и атрибут time-layout элемента temperature в элементе parameters.

22.4. Извлечение веб-данных

В некоторых ситуациях данные на сайте по какой-то причине недоступны в других местах. Тогда есть смысл собрать данные с самих веб-страниц с использованием процесса, называемого *извлечением данных* или *автоматическим сбором данных* (scraping, crawling).

Прежде чем что-либо говорить об извлечении данных, я должна однозначно заявить: извлечение данных на сайтах, которые вам не принадлежат или не находятся под вашим управлением, — практика в лучшем случае неоднозначная с юридической точки зрения. Существует великое множество неполных и противоречивых соображений, относящихся к таким аспектам, как правила использования сайта, способ обращения к сайту и применение извлеченных данных. Если только сайт не находится под вашим контролем, на вопрос: «А у меня есть право извлекать данные с этого сайта?» обычно приходится отвечать: «Это как посмотреть».

Если вы решите извлекать данные с сайта, находящегося в реальной эксплуатации, вам также нужно учитывать нагрузку, которую вы создаете для сайта. Возможно, проверенный временем сайт с высоким трафиком справится с любой нагрузкой, которую вы для него создадите, но серия многократных запросов может парализовать менее активный сайт. Как минимум нужно проследить за тем, чтобы извлечение данных не превратилось в непреднамеренную атаку типа отказа в обслуживании (DoS).

И наоборот, я оказывалась в ситуациях, когда бывало проще извлечь нужные данные с сайта, чем запрашивать их по корпоративным каналам. Хотя извлечение данных находит свое применение, эта тема слишком сложна, чтобы полностью изложить ее здесь. В этом разделе я приведу очень простой пример, который даст общее представление об основных принципах, и дам рекомендации для более сложных случаев.

Извлечение данных с сайта состоит из двух частей: загрузка веб-страницы и получение данных. Загрузка страницы может осуществляться посредством запросов, этот шаг относительно прост.

Рассмотрим код очень простой веб-страницы с минимумом контента, без CSS и JavaScript.

Листинг 22.1. Файл test.html

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>Title</title>
</head>

<body>
<h1>Heading 1</h1>

This is plan text, and is boring
<span class="special">this is special</span>

Here is a <a href="http://bitbucket.dev.null">link</a>

<hr>
<address>Ann Address, Somewhere, AState 00000
</address>
</body> </html>
```

Предположим, на этой странице вас интересуют данные только двух видов: все содержимое элементов с именем класса `"special"` и все ссылки. Файл можно обработать, проведя поиск по строкам `'class="special"'` и `"<a href"`, а затем написать код выборки данных от найденных позиций, но даже с регулярными выражениями этот процесс будет утомительным и ненадежным, а код будет трудным в сопровождении. Гораздо проще воспользоваться библиотекой, которая умеет разбирать HTML, например Beautiful Soup. Если вы хотите опробовать следующий код и поэкспериментировать с разбором HTML-страниц, используйте `pip install bs4`.

Если вы уже установили Beautiful Soup, разобрать HTML-страницу будет просто. В данном примере предполагается, что веб-страница уже загружена (возможно, при помощи библиотеки `requests`), поэтому мы просто разберем разметку HTML.

Обработка начинается с загрузки текста и создания парсера Beautiful Soup:

```
>>> import bs4
>>> html = open("test.html").read()
>>> bs = bs4.BeautifulSoup(html, "html.parser")
```

И это все, что нужно для разбора HTML в объект парсера `bs`. Объект парсера Beautiful Soup обладает множеством интересных возможностей, и если вы работаете с HTML, вам определенно стоит поэкспериментировать и понять, что он может сделать за вас. В данном примере нас интересуют две возможности: извлечение контента по тегу HTML и получение данных по классу CSS.

Начнем с поиска ссылки. Тег HTML для ссылок имеет вид `<a>` (Beautiful Soup по умолчанию преобразует все теги к нижнему регистру), поэтому чтобы найти все теги ссылок, можно вызвать сам объект `bs`, передав `"a"` в параметре:

```
>>> a_list = bs("a")
>>> print(a_list)
[<a href="http://bitbucket.dev.null">link</a>]
```

Теперь у вас имеется список всех тегов ссылок HTML (в данном случае ссылка всего одна). Если этот список — все, что вам нужно, уже неплохо, но на самом деле элементы, возвращаемые списком, также являются объектами парсера и могут выполнить остальную работу по получению ссылок и текста:

```
>>> a_item = a_list[0]
>>> a_item.text
'link'
>>> a_item["href"]
'http://bitbucket.dev.null'
```

Другая нужная возможность — поиск всех элементов с классом CSS `"special"`. Их можно получить при помощи метода `select` объекта парсера:

```
>>> special_list = bs.select(".special")
>>> print(special_list)
[<span class="special">this is special</span>]
>>> special_item = special_list[0]
>>> special_item.text
'this is special'
>>> special_item["class"]
['special']
```

Так как элементы, возвращаемые тегом или методом `select`, сами по себе являются объектами парсера, они могут использоваться с вложением, что позволяет извлечь практически любую информацию из HTML или даже XML.

ПОПРОБУЙТЕ САМИ: РАЗБОР HTML

Для заданного файла forecast.html (находится в коде на веб-сайте книги) напишите сценарий с использованием Beautiful Soup, который извлекает данные и сохраняет их в файле CSV.

Листинг 22.2. Файл forecast.html

```
<html>
  <body>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Tonight</b></div>
        <div class="grid col-75 forecast-text">A slight chance of showers and
     thunderstorms before 10pm. Mostly cloudy, with a low around 66. West
     southwest wind around 9 mph. Chance of precipitation is 20%. New
     rainfall amounts between a tenth and quarter of an inch possible.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Friday</b></div>
        <div class="grid col-75 forecast-text">Partly sunny. High near 77,
```

```
     with temperatures falling to around 75 in the afternoon. Northwest wind
     7 to 12 mph, with gusts as high as 18 mph.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Friday Night</b></div>
        <div class="grid col-75 forecast-text">Mostly cloudy, with a low
     around 63. North wind 7 to 10 mph.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Saturday</b></div>
        <div class="grid col-75 forecast-text">Mostly sunny, with a high near
     73. North wind around 10 mph.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Saturday Night</b></div>
        <div class="grid col-75 forecast-text">Partly cloudy, with a low
     around 63. North wind 5 to 10 mph.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Sunday</b></div>
        <div class="grid col-75 forecast-text">Mostly sunny, with a high near
     73.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Sunday Night</b></div>
        <div class="grid col-75 forecast-text">Mostly cloudy, with a low
     around 64.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Monday</b></div>
        <div class="grid col-75 forecast-text">Mostly sunny, with a high near
     74.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Monday Night</b></div>
        <div class="grid col-75 forecast-text">Mostly clear, with a low
around 65.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Tuesday</b></div>
        <div class="grid col-75 forecast-text">Sunny, with a high near 75.</
div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Tuesday Night</b></div>
        <div class="grid col-75 forecast-text">Mostly clear, with a low
     around 65.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Wednesday</b></div>
        <div class="grid col-75 forecast-text">Sunny, with a high near 77.</
div>
    </div>
```

```
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Wednesday Night</b></div>
        <div class="grid col-75 forecast-text">Mostly clear, with a low
     around 67.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Thursday</b></div>
        <div class="grid col-75 forecast-text">A chance of rain showers after
     1pm. Mostly sunny, with a high near 81. Chance of precipitation is
     30%.</div>
    </div>
  </body>
</html>
```

ПРАКТИЧЕСКАЯ РАБОТА 22: СБОР ПОГОДНЫХ ДАННЫХ ОТ МАРСОХОДА

Используйте API, описанный в разделе 22.2, для сбора истории метеорологических данных во время пребывания марсохода «Кьюриосити» на Марсе в течение месяца. Подсказка: чтобы задать марсианские сутки, добавьте ?sol=число в конец запроса к архиву, например

http://marsweather.ingenology.com/v1/archive/?sol=155

Преобразуйте данные, чтобы их можно было загрузить в электронной таблице, и создайте их графическое представление. Одна из версий проекта приведена в исходном коде книги.

Итоги

- Сценарий Python может быть не лучшим вариантом загрузки файлов. Обязательно рассмотрите другие возможности.
- Модуль `requests` — лучший кандидат для загрузки файлов с использованием HTTP/HTTPS и Python.
- Процесс получения файлов из API очень похож на процесс получения статических файлов.
- Параметры запросов API часто заключаются в кавычки и добавляются в строку запроса.
- Данные от API обычно предоставляются в формате JSON; также часто встречается формат XML.
- Извлечение данных с сайтов, которые не находятся под вашим контролем, может быть нелегальным или неэтичным. Также проследите за тем, чтобы ваши действия не создавали излишней нагрузки на сервер.

23 Хранение данных

Эта глава охватывает следующие темы:

- ✓ Хранение данных в реляционных базах данных
- ✓ Использование DB-API Python
- ✓ Доступ к базам данных через объект Relational Mapper (ORM)
- ✓ Описание баз данных NoSQL и их отличие от реляционных баз данных

Если у вас имеются данные, которые уже прошли очистку, вполне вероятно, что вы захотите сохранить их, и не только сохранить, но и получить к ним доступ в будущем с минимальными хлопотами. Для хранения и выборки значительных объемов данных обычно применяются те или иные разновидности *баз данных*. Реляционные базы данных, такие как PostgreSQL, MySQL и SQL Server, десятилетиями считались лидерами в области хранения данных, и сейчас они остаются отличным решением во многих случаях. В последние годы стали популярными базы данных NoSQL, включая MongoDB и Redis. Подробное описание баз данных заняло бы не одну книгу, поэтому в этой главе мы рассмотрим несколько примеров, демонстрирующих работу с базами данных SQL и NoSQL из Python.

23.1. Реляционные базы данных

Реляционные базы данных в течение долгого времени считались стандартом хранения и обработки данных. Эта технология проверена временем и получила повсеместное распространение. Python может подключаться к множеству разных реляционных баз данных, но у меня нет ни времени, ни желания подробно описывать особенности каждой базы данных в книге. Вместо этого, поскольку Python в основном работает с базами данных по единой схеме, я представлю основы на примере одной из них — sqlite3, а затем мы обсудим некоторые различия и факторы выбора и использования реляционных баз данных для хранения информации.

23.1.1. Python Database API

Как упоминалось ранее, в Python обращения к базам данных SQL осуществляются почти одинаково в разных реализациях баз данных благодаря документу PEP-249 (www.python.org/dev/peps/pep-0249/), в котором указаны некоторые стандартные правила подключения к базам данных SQL. Этот стандарт, часто обозначаемый термином Database API или DB-API, был создан для содействия «созданию кода, который в общем случае обладает лучшей портируемостью между базами данных и более широким спектром средств подключения к базам данных». Благодаря DB-API примеры SQLite, которые встречаются в этой главе, очень похожи на те, которые встречаются при использовании PostgreSQL, MySQL или других баз данных.

23.2. SQLite: использование базы данных sqlite3

Хотя в Python существуют модули для многих баз данных, в следующих примерах будет рассматриваться база данных sqlite3. И хотя sqlite3 не подходит для больших приложений с высоким трафиком, эта база данных обладает двумя преимуществами:

- Так как sqlite3 является частью стандартной библиотеки, вы сможете использовать эту базу данных везде, где вам потребуется поддержка базы данных, не беспокоясь о добавлении зависимостей.
- sqlite3 сохраняет все свои записи в локальном файле, поэтому ей не потребуется клиентская и серверная часть — в отличие от PostgreSQL, MySQL и других больших баз данных.

Благодаря этим особенностям sqlite3 хорошо подходит для небольших приложений и прототипов.

Чтобы использовать базу данных sqlite3, вам прежде всего понадобится объект подключения `Connection`. Для получения объекта `Connection` достаточно вызвать функцию `connect` с именем файла, который будет использоваться для хранения данных:

```
>>> import sqlite3
>>> conn = sqlite3.connect("datafile.db")
```

Также возможно хранить данные в памяти, указав вместо имени файла строку `":memory:"`. Для хранения целых чисел, чисел с плавающей точкой и строк ничего больше не потребуется. Если вы хотите, чтобы результаты запросов для некоторых столбцов автоматически преобразовывались sqlite3 в другие типы, присвойте параметру `detect_types` значение `sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_COLNAMES`. Оно приказывает объекту `Connection` разобрать имена и типы столбцов в запросах и попытаться сопоставить их с уже определенными преобразователями (converters).

Вторым шагом становится создание объекта `Cursor` для подключения:

```
>>> cursor = conn.cursor()
>>> cursor
<sqlite3.Cursor object at 0xb7a12980>
```

В этот момент вы уже можете выдавать запросы к базе данных. В текущей ситуации, поскольку база данных еще не содержит ни таблиц, ни записей, сначала необходимо создать таблицу и вставить пару записей:

```
>>> cursor.execute("create table people (id integer primary key, name text,
count integer)")
>>> cursor.execute("insert into people (name, count) values ('Bob', 1)")
>>> cursor.execute("insert into people (name, count) values (?, ?)",
...                ("Jill", 15))
>>> conn.commit()
```

Последний запрос `insert` демонстрирует предпочтительный способ создания запросов с переменными. Вместо того чтобы строить строку запроса, безопаснее поставить `?` на место каждой переменной, а затем передать переменные в параметре-кортеже методу `execute`. Этот способ хорош тем, что вам не нужно беспокоиться о неправильном экранировании значений; sqlite3 сделает все за вас.

Также в запросе можно использовать имена переменных с префиксом `:` и передать соответствующий словарь со вставляемыми значениями:

```
>>> cursor.execute("insert into people (name, count) values (:username, \
                   :usercount)", {"username": "Joe", "usercount": 10})
```

После того как таблица будет заполнена, вы сможете запросить данные командами SQL — снова с использованием либо `?` для привязки имен переменных, либо имен и словарей:

```
>>> result = cursor.execute("select * from people")
>>> print(result.fetchall())
[('Bob', 1), ('Jill', 15), ('Joe', 10)]
>>> result = cursor.execute("select * from people where name like :name",
...                         {"name": "bob"})
>>> print(result.fetchall())
[('Bob', 1)]
>>> cursor.execute("update people set count=? where name=?", (20, "Jill"))
>>> result = cursor.execute("select * from people")
>>> print(result.fetchall())
[('Bob', 1), ('Jill', 20), ('Joe', 10)]
```

Кроме метода `fetchall`, метод `fetchone` получает одну строку результата, а `fetchmany` возвращает произвольное количество строк. Для удобства также возможно перебрать строки объекта курсора по аналогии с перебором по файлу:

```
>>> result = cursor.execute("select * from people")
>>> for row in result:
...     print(row)
...
('Bob', 1)
('Jill', 20)
('Joe', 10)
```

Наконец, по умолчанию sqlite3 не выполняет немедленного закрепления транзакций. Этот факт означает, что при неудаче у вас будет возможность откатить

транзакцию, но это также означает, что вы должны вызвать объект `commit` метода `Connection`, чтобы обеспечить сохранение всех внесенных изменений. Особенно желательно делать это до закрытия подключения к базе данных, потому что метод `close` не осуществляет автоматического закрепления всех активных транзакций:

```
>>> cursor.execute("update people set count=? where name=?", (20, "Jill"))
>>> conn.commit()
>>> conn.close()
```

В табл. 23.1 приведен обзор основных операций с базой данных sqlite3.

Таблица 23.1. Основные операции с базой данных sqlite3

Операция	Команда sqlite3
Создание подключения к базе данных	`conn = sqlite3.connect(filename)`
Создание курсора для подключения	`Cursor = conn.cursor()`
Выполнение запроса с курсором	`cursor.execute(query)`

Возвращение результатов запроса	`cursor.fetchall(),cursor.fetchmany(num_rows),`
	`cursor.fetchone()`
	`for row in cursor:`
	`....`

Закрепление транзакции к базе данных	`conn.commit()`
Закрытие подключения	`conn.close()`

Обычно этих операций достаточно для работы с базой данных sqlite3. Конечно, существуют специальные параметры, позволяющие управлять их поведением; за дополнительной информацией обращайтесь к документации Python.

ПОПРОБУЙТЕ САМИ: СОЗДАНИЕ И МОДИФИКАЦИЯ ТАБЛИЦ

Используя sqlite3, напишите код, который создает таблицу базы данных для метеорологических данных штата Иллинойс, загруженных из файла в разделе 21.2. Предположим, у вас имеются аналогичные данные по другим штатам и вы хотите сохранить дополнительную информацию о штате. Как изменить базу данных, чтобы для хранения информации о штате использовалась связанная таблица?

23.3. MySQL, PostgreSQL и другие реляционные базы данных

Как упоминалось ранее в этой главе, у ряда других баз данных SQL имеются клиентские библиотеки, работающие через DB-API. В результате обращения к этим базам данных в Python выглядят очень похоже, но есть и различия, о которых стоит упомянуть.

- В отличие от SQLite, этим базам данных необходим сервер базы данных, к которому подключается клинет. Этот сервер может находиться на другой машине (хотя это необязательно), поэтому для подключения приходится указывать больше параметров — обычно хост, имя учетной записи и пароль.
- Механизм включения параметров в запросы, например, `"select * from test where name like :name"` может использовать другой формат, например, `?, %s 5(name)s`.

Изменения не столь значительны, но они могут препятствовать полной портируемости между разными базами данных.

23.4. Простая работа с базами данных с ORM

У клиентских библиотек баз данных DB-API, упоминавшихся ранее в этой главе, и их требований по написанию низкоуровневого кода SQL есть свои проблемы.

- В разных базах данных SQL используются несколько различающиеся реализации SQL, поэтому одни и те же команды SQL не всегда будут работать при переходе на другую базу данных, как бы вам ни хотелось обратного, если вы, скажем, ведете локальную разработку на sqlite3, а затем переходите на MySQL или PostgreSQL в итоговой версии. Кроме того, как упоминалось ранее, в разных реализациях по-разному решаются некоторые задачи, как, например, передача параметров запросам.
- Вторая проблема — необходимость использования низкоуровневых команд SQL. Включение команд SQL в код может усложнить сопровождение кода, особенно если таких команд будет много. В таком случае некоторые команды будут шаблонными и стандартными, другие — сложными и хитроумными, и все эти команды придется протестировать, что может быть довольно трудоемким делом.
- Необходимость написания кода SQL означает, что вы должны мыслить понятиями как минимум двух языков: Python и конкретного диалекта SQL. Во многих случаях использование низкоуровневых команд SQL стоит затраченных усилий, во многих других случаях — нет.

Из-за этих проблем разработчикам требовался механизм работы с базами данных на языке Python, которым было бы проще управлять и который бы не требовал ничего, кроме написания обычного кода Python. Для этого была создана технология объектно-реляционного отображения, или *ORM* (Object Relational Mapper), которая преобразует структуры и типы баз данных в объекты Python. Две самые распространенные системы ORM в мире Python — Django ORM и SQLAlchemy, хотя, конечно, есть и много других. Система Django ORM плотно интегрируется с веб-фреймворком Django и обычно не используется за ее пределами. Так как в этой книге я не стану подробно рассматривать Django, в обсуждении Django ORM я просто скажу, что эта технология используется по умолчанию для приложений Django — и это хороший вариант с полностью разработанными инструментами и бескорыстной поддержкой сообщества.

23.4.1. SQLAlchemy

SQLAlchemy — другая известная система ORM в мире Python. Целью SQLAlchemy является автоматизация лишних задач баз данных и создание объектно-базированных интерфейсов Python для данных, позволяющих разработчику управлять базой данных и предоставляющих разработчику доступ к низкоуровневым командам SQL. В этом разделе мы рассмотрим простые примеры сохранения данных в реляционных базах данных и их выборки с использованием SQLAlchemy.

Чтобы установить SQLAlchemy в вашей среде, воспользуйтесь `pip`:

```
> pip install sqlalchemy
```

ПРИМЕЧАНИЕ

С этого момента при работе с SQLAlchemy и сопутствующими инструментами будет удобнее держать открытыми два окна оболочки в одной виртуальной среде: для Python и для командной строки вашей системы.

SQLAlchemy предоставляет несколько способов взаимодействия с базой данных и ее таблицами. Хотя ORM позволяет писать команды SQL при необходимости (или если вам этого захочется), главное достоинство ORM следует из самого названия: ORM отображает таблицы и столбцы реляционной базы данных на объекты Python.

Используем SQLAlchemy для повторения того, что было сделано в разделе 23.2: создайте таблицу, добавьте три строки, обратитесь с запросом к таблице и обновите одну строку. Для использования ORM придется проделать чуть более серьезную подготовку, но в больших проектах эти усилия с лихвой окупятся.

Сначала необходимо импортировать компоненты, необходимые для подключения к базе данных, и отобразить таблицу на объекты Python. Из базового пакета `sqlalchemy` вам понадобятся методы `create_engine` и `select`, а также классы `MetaData` и `Table`. Но поскольку при создании объекта таблицы `table` необходимо задать информацию схемы, вы также должны импортировать класс `Column` и классы типов данных для всех столбцов, в данном случае это `Integer` и `String`. Из субпакета `sqlalchemy.orm` вам также понадобится функция `sessionmaker`:

```
>>> from sqlalchemy import create_engine, select, MetaData, Table, Column,
Integer, String
>>> from sqlalchemy.orm import sessionmaker
```

А теперь можно заняться подключением к базе данных:

```
>>> dbPath = 'datafile2.db'
>>> engine = create_engine('sqlite:///%s' % dbPath)
>>> metadata = MetaData(engine)
>>> people  = Table('people', metadata,
...                 Column('id', Integer, primary_key=True),
...                 Column('name', String),
...                 Column('count', Integer),
...                )
```

```
>>> Session = sessionmaker(bind=engine)
>>> session = Session()
>>> metadata.create_all(engine)
```

Для создания и подключения необходимо создать объект ядра, соответствующий вашей базе данных; также понадобится объект `MetaData`, выполняющий функции контейнера для управления таблицами и их схемами. Создайте объект `Table` с именем `data`. При вызове передайте имя таблицы в базе данных, только что созданный объект `MetaData` и объекты создаваемых столбцов с их типами данных. Наконец, при помощи функции `sessionmaker` создайте класс `Session` для ядра и используйте этот класс для создания экземпляра объекта сеанса. В этот момент вы подключены к базе данных, остается вызвать метод `create_all` для создания таблицы.

При создании таблицы на следующем этапе в нее вставляются записи. И снова SQL Alchemy предоставляет для этого много разных возможностей, но в данном случае мы будем действовать предельно однозначно. Создайте объект `insert` и выполните его вызовом `execute`:

```
>>> people_ins = people.insert().values(name='Bob', count=1)
>>> str(people_ins)
'INSERT INTO people (name, count) VALUES (?, ?)'
>>> session.execute(people_ins)
<sqlalchemy.engine.result.ResultProxy object at 0x7f126c6dd438>
>>> session.commit()
```

Метод `insert()` используется для создания объекта `insert`, при этом указываются поля и значения, которые в них вставляются. `people_ins` — объект `insert`, вызов функции `str()` показывает, что «за кулисами» была создана правильная команда SQL. Затем метод `execute` объекта сеанса выполняет вставку, а метод `commit` закрепляет изменения в базе данных:

```
>>> session.execute(people_ins, [
...     {'name': 'Jill', 'count':15},
...     {'name': 'Joe', 'count':10}
... ])
<sqlalchemy.engine.result.ResultProxy object at 0x7f126c6dd908>
>>> session.commit()
>>> result = session.execute(select([people]))
>>> for row in result:
...     print(row)
...
(1, 'Bob', 1)
(2, 'Jill', 15)
(3, 'Joe', 10)
```

Запись можно немного упростить и выполнить несколько вставок, передавая список словарей с именами и значениями полей для каждой вставки:

```
>>> result = session.execute(select([people]).where(people.c.name == 'Jill'))
>>> for row in result:
...     print(row)
...
(2, 'Jill', 15)
```

Также можно использовать метод `select()` с методом `where()` для поиска конкретной записи. В нашем примере мы ищем все записи, у которых `имя` столбца равно `'Jill'`. Обратите внимание: в выражении `where` используется запись `people.c.name`, где `c` означает, что `name` является столбцом таблицы `people`:

```
>>> result = session.execute(people.update().values(count=20).where
(people.c.name == 'Jill'))
>>> session.commit()
>>> result = session.execute(select([people]).where(people.c.name == 'Jill'))
>>> for row in result:
...     print(row)
...
(2, 'Jill', 20)
>>>
```

Наконец, метод `update()` можно объединить с методом `where()`, чтобы обновить только одну строку.

Отображение объектов таблиц на классы

До настоящего момента вы использовали объекты таблиц напрямую, но SQLAlchemy также может использоваться для отображения таблицы непосредственно на класс. Преимущество такого решения заключается в том, что столбцы напрямую отображаются на атрибуты класса. Для демонстрации создайте класс `People`:

```
>>> from sqlalchemy.ext.declarative import declarative_base
>>> Base = declarative_base()
>>> class People(Base):
...     __tablename__ = "people"
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     count = Column(Integer)
...
>>> results = session.query(People).filter_by(name='Jill')
>>> for person in results:
...     print(person.id, person.name, person.count)
...
2 Jill 20
```

Вставка может выполняться простым созданием экземпляра отображенного класса и добавлением его в сеанс:

```
>>> new_person = People(name='Jane', count=5)
>>> session.add(new_person)
>>> session.commit()
>>>
>>> results = session.query(People).all()
>>> for person in results:
...     print(person.id, person.name, person.count)
...
1 Bob 1
2 Jill 20
3 Joe 10
4 Jane 5
```

Обновление тоже выполняется достаточно элементарно. Вы получаете запись, которую нужно обновить, изменяете значения в отображенном экземпляре, а затем добавляете обновленную запись в сеанс для записи в базу данных:

```
>>> jill = session.query(People).filter_by(name='Jill').first()
>>> jill.name
'Jill'
>>> jill.count = 22
>>> session.add(jill)
>>> session.commit()
>>> results = session.query(People).all()
>>> for person in results:
...     print(person.id, person.name, person.count)
...
1 Bob 1
2 Jill 22
3 Joe 10
4 Jane 5
```

Удаление происходит аналогично обновлению: вы получаете удаляемую запись, а затем вызываете метод `delete()` сеанса для ее удаления:

```
>>> jane = session.query(People).filter_by(name='Jane').first()
>>> session.delete(jane)
>>> session.commit()
>>> jane = session.query(People).filter_by(name='Jane').first()
>>> print(jane)
None
```

Использование SQLAlchemy требует чуть больших усилий, чем низкоуровневые команды SQL, но у этого варианта есть бесспорные преимущества. Во-первых, использование ORM означает, что вам не придется беспокоиться о тонких различиях в версиях SQL, поддерживаемых разными базами данных. Пример с равным успехом работает в sqlite3, MySQL и PostgreSQL без внесения каких-либо изменений в код, кроме передачи строки при создании ядра и проверке наличия драйвера правильной базы данных.

Другое преимущество заключается в том, что взаимодействие с данными может осуществляться через объекты Python, которые более понятны программистам с недостаточным опытом SQL. Вместо построения команд SQL они могут использовать объекты Python и их методы.

ПОПРОБУЙТЕ САМИ: ИСПОЛЬЗОВАНИЕ ORM

Для базы данных из предыдущего примера напишите класс SQLAlchemy, отображаемый на таблицу с данными. Используйте его для чтения записей из таблицы.

23.4.2. Использование Alembic для изменения схемы базы данных

В процессе разработки кода, использующего реляционную базу данных, нередко (и даже часто) возникает необходимость в изменении структуры (схемы) базы данных уже после того, как вы стали работать с ней. В базу данных добавляются новые поля, изменяются типы существующих полей и т. д. Конечно, можно вручную внести изменения как в таблицы базы данных, так и в код ORM, обращающийся к ним, но у такого подхода есть свои недостатки. Такие изменения труднее отменить в случае необходимости; кроме того, труднее отслеживать конфигурацию базы данных, которая соответствует конкретной версии вашего кода.

Проблема решается средствами миграции баз данных, которые упрощают внесение изменений и их последующее отслеживание. Миграции записываются в виде программного кода. Они должны включать код как для применения необходимых изменений, так и для их отмены. Далее изменения могут отслеживаться, применяться и отменяться в правильной последовательности. В результате база данных может быть безопасно возвращена к любому из состояний, в которых она находилась в ходе разработки.

Для примера в этом разделе кратко рассматривается Alembic — популярный и несложный инструмент миграции для SQLAlchemy. Для начала переключитесь в системное окно командной строки в каталоге проекта, установите Alembic и создайте обобщенную среду командой `alembic init`:

```
> pip install alembic
> alembic init alembic
```

Этот код создает файловую структуру, необходимую Alembic для миграции данных. В ней имеется файл alembic.ini, который необходимо будет отредактировать как минимум в одном месте. Строку `squalchemy.url` следует привести в соответствие с текущей ситуацией:

```
sqlalchemy.url = driver://user:pass@localhost/dbname
```

Замените эту строку следующей:

```
sqlalchemy.url = sqlite:///datafile.db
```

Так как вы используете локальный файл sqlite, имя пользователя или пароль не нужны.

На следующем шаге создается *ревизия* (revision) командой `alembic revision`:

```
> alembic revision -m "create an address table"
Generating /home/naomi/qpb_testing/alembic/versions/
     384ead9efdfd_create_a_test_address_table.py ... done
```

Код генерирует сценарий ревизии 384ead9efdfd_create_a_test_address_table.py в каталоге alembic/versions. Файл выглядит так:

```
"""create an address table

Revision ID: 384ead9efdfd
Revises:
Create Date: 2017-07-26 21:03:29.042762

"""
from alembic import op
import sqlalchemy as sa

# revision identifiers, used by Alembic.
revision = '384ead9efdfd'
down_revision = None
branch_labels = None
depends_on = None

def upgrade():
    pass


def downgrade():
    pass
```

Как видите, заголовок файла содержит идентификатор ревизии и дату. Также он содержит переменную `down_revision`, управляющую отменой каждой версии. Если вы создадите вторую ревизию, ее переменная `down_revision` должна содержать идентификатор этой ревизии.

Чтобы выполнить ревизию, измените сценарий ревизии и включите в него как код, описывающий выполнение ревизии в методе `upgrade()`, так и код ее отмены в методе `downgrade()`:

```
def upgrade():
    op.create_table(
        'address',
        sa.Column('id', sa.Integer, primary_key=True),
        sa.Column('address', sa.String(50), nullable=False),
        sa.Column('city', sa.String(50), nullable=False),
        sa.Column('state', sa.String(20), nullable=False),
    )

def downgrade():
    op.drop_table('address')
```

Когда этот код будет создан, можно применить обновление базы данных. Но сначала переключитесь обратно в окно оболочки Python и посмотрите, какие таблицы присутствуют в базе данных:

```
>>> print(engine.table_names())
['people']
```

Как и следовало ожидать, база данных содержит всего одну таблицу, которая была создана ранее. Теперь можно выполнить команду `alembic upgrade`, чтобы применить обновление и добавить новую таблицу. Переключитесь в системное окно командной строки и выполните команду:

```
> alembic upgrade head
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> 384ead9efdfd, create an
address table
```

Если теперь переключиться обратно на Python и проверить, вы увидите, что в базе данных появились две дополнительные таблицы:

```
>>> engine.table_names()
['alembic_version', 'people', 'address'
```

Первая таблица, `'alembic version'`, была создана Alembic для отслеживания текущей версии базы данных (для будущих повышений и понижений версии). Вторая новая таблица `'address'` добавлена кодом обновления и готова к использованию.

Чтобы вернуть базу данных к прежнему состоянию, достаточно выполнить команду `alembic downgrade` в системном окне. Команде `downgrade` передается значение `-1` — тем самым вы сообщаете Alembic, что хотите вернуться на одну версию назад.

```
> alembic downgrade -1
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running downgrade 384ead9efdfd -> , create
an address table
```

Проверка в сеансе Python показывает, что все вернулось к исходному состоянию, если не считать того, что таблица отслеживания версий осталась:

```
>>> engine.table_names()
['alembic_version', 'people']
```

Конечно, при желании вы можете снова провести обновление и вернуть таблицу в прежнее состояние, добавить новые ревизии, определить другие обновления и т. д.

ПОПРОБУЙТЕ САМИ: ИЗМЕНЕНИЕ БАЗЫ ДАННЫХ С ALEMBIC

Поэкспериментируйте с созданием обновления Alembic, которое добавляет в базу данных таблицу штатов со столбцами для идентификатора, названия и сокращенного обозначения штата. Проведите обновление и возврат. Какие еще изменения потребуются, если бы вы собирались использовать таблицу штатов с существующей таблицей данных?

23.5. Базы данных NoSQL

Несмотря на свою давнюю популярность, реляционные базы данных не единственный подход к хранению данных. Хотя реляционные базы данных предназначены для нормализации данных в связанных таблицах, другие технологии рассматривают данные иначе. Обычно базы данных этого типа называются базами данных

NoSQL, поскольку они не придерживаются структуры строка/столбец/таблица, для описания которой создавался язык SQL.

Вместо того чтобы обрабатывать данные как совокупность строк, столбцов и таблиц, базы данных NoSQL рассматривают хранимые данные как пары «ключ–значение», как индексированные документы и даже как графы. Существует много баз данных NoSQL, использующих разные подходы к обработке данных. В общем случае базы данных NoSQL обычно обладают менее жесткой нормализацией, что может ускорить и упростить выборку информации. В этом разделе мы рассмотрим примеры использования Python для обращения к двум стандартным базам данных NoSQL: Redis и MongoDB. Дальнейшее описание в лучшем случае познакомит вас с тем, что можно сделать с базами данных NoSQL и Python, но по крайней мере вы получите представление об основных возможностях. Читатели, уже имеющие опыт работы с Redis или MongoDB, увидят, как работают клиентские библиотеки Python, а читатели, которые только начинают осваивать базы данных NoSQL, поймут, как работают базы данных этого типа.

23.6. Хранение пар «ключ–значение» в Redis

Redis представляет собой сетевое хранилище пар «ключ–значение» в памяти. Так как данные хранятся в памяти, выборка осуществляется относительно быстро, а благодаря поддержке доступа к данным по сети Redis может использоваться в разнообразных ситуациях. Redis часто применяется для кэширования, как брокер сообщений и для быстрой выборки информации. Собственно, само название (сокращение от «Remote Dictionary Server», то есть «удаленный сервер словарей») отлично объясняет, как можно рассматривать функциональность Redis. Эта база данных ведет себя как словарь Python, преобразованный в сетевую службу.

Следующий пример дает представление об использовании Redis в Python. Если вы знакомы с интерфейсом командной строки Redis или использовали клиент Redis в другом языке программирования, после этих коротких примеров вы начнете представлять, как использовать Redis в Python. Если же у вас нет никакого опыта работы с Redis, примеры дадут вам представление о том, как работает эта технология; за дополнительной информацией обращайтесь по адресу https://redis.io.

Существует несколько разных клиентов Python для Redis; на момент написания книги (по информации с веб-сайта Redis) рекомендуется использовать `redis-py`. Эта библиотека устанавливается командой `pip install redis`.

ЗАПУСК СЕРВЕРА REDIS

Для экспериментов вам понадобится работающий сервер Redis. Хотя вы можете использовать облачные сервисы Redis, для экспериментов лучше всего использовать экземпляр Docker или установить сервер на машине.

Если у вас установлена поддержка Docker, пожалуй, использование Docker-экземпляра Redis будет самым быстрым и простым способом создания работающего сервера. Экземпляр Redis также можно запустить из командной строки командой вида `> docker run -p 6379:6379 redis`.

В системах Linux Redis легко устанавливается при помощи системного менеджера пакетов, а в системах Mac должна работать команда `brew install`. Если вы используете систему Windows, обращайтесь за информацией на сайт https://redis.io или поищите в интернете доступные варианты запуска Redis в Windows. Возможно, после установки Redis вам придется поискать в интернете инструкции о том, как обеспечить работоспособность сервера Redis.

Когда сервер заработает, вы сможете повторить приведенные примеры простых взаимодействий с Redis из Python. Сначала необходимо импортировать библиотеку Redis и создать объект подключения Redis:

```
>>> import redis
>>> r = redis.Redis(host='localhost', port=6379)
```

При создании подключения Redis можно указать параметры подключения, включая хост, порт и пароль или сертификат SSH. Если сервер работает на локальном хосте с портом по умолчанию 6379, то никакие дополнительные параметры не нужны. После создания подключения вы можете использовать его для обращения к хранилищу пар «ключ–значение».

Начните с вызова метода `keys()` для получения списка ключей в базе данных. Метод возвращает текущий список существующих ключей (если они есть). Затем вы можете задать ключи со значениями разных типов и опробовать разные способы получения связанных с ними значений:

```
>>> r.keys()
[]
>>> r.set('a_key', 'my value')
True
>>> r.keys()
[b'a_key']
>>> v = r.get('a_key')
>>> v
b'my value'
>>> r.incr('counter')
1
>>> r.get('counter')
b'1'
>>> r.incr('counter')
2
>>> r.get('counter')
b'2'
```

Эти примеры показывают, как получить список ключей в базе данных Redis, как создать пару «ключ–значение», как создать ключ с именем `counter` и последовательно увеличивать его значение.

В следующих примерах рассматривается хранение массивов или списков:

```
>>> r.rpush("words", "one")
1
>>> r.rpush("words", "two")
2
```

```
>>> r.lrange("words", 0, -1)
[b'one', b'two']
>>> r.rpush("words", "three")
3
>>> r.lrange("words", 0, -1)
[b'one', b'two', b'three']
>>> r.llen("words")
3
>>> r.lpush("words", "zero")
4
>>> r.lrange("words", 0, -1)
[b'zero', b'one', b'two', b'three']
>>> r.lrange("words", 2, 2)
[b'two']
>>> r.lindex("words", 1)
b'one'
>>> r.lindex("words", 2)
b'two'
```

При создании ключа `"words"` нет в базе данных, но операция добавления или занесения значения (от конца, то есть справа, — отсюда буква «r» в `rpush`) создает ключ, создает пустой список как связанное с ним значение и присоединяет к списку значение `'one'`. Повторное использование `rpush` добавляет в конец списка еще одно слово. Для получения значений из списка используется функция `lrange()`, которой при вызове передается ключ, начальный индекс и конечный индекс (`-1` — обозначение конца списка).

Обратите внимание на то, что элементы также могут добавляться от начала списка (слева) вызовом `lpush()`. Функция `lindex()` используется для получения отдельного значения по аналогии с `lrange()`, если не считать того, что ей передается индекс нужного значения.

Срок жизни значений

Одна из особенностей Redis, которая особенно полезна при кэшировании, — возможность ограничения срока жизни пар «ключ–значение». По истечении заданного времени ключ и значение удаляются из хранилища. Срок жизни в секундах может быть установлен при задании значения для ключа:

```
>>> r.setex("timed", "10 seconds", 10)
True
>>> r.pttl("timed")
7165
>>> r.pttl("timed")
5208
>>> r.pttl("timed")
1542
>>> r.pttl("timed")
>>>
```

В данном случае срок жизни `"timed"` ограничивается десятью секундами. Затем вызов метода `pttl()` возвращает время, оставшееся до истечения срока жизни, в миллисекундах. По истечении этого значения и ключ и значение автоматически удаляются из базы данных. Ограничение срока жизни и средства точного управления им в Redis чрезвычайно полезны. Возможно, для простого кэширования вам практически не придется писать дополнительный код для решения вашей проблемы.

Стоит заметить, что Redis хранит данные в памяти, поэтому при сбое сервера информация может быть потеряна. Для предотвращения потери данных в Redis предусмотрены средства долгосрочного хранения — от записи всех изменений на диск до регулярного создания «снимков» с заранее определенными интервалами или полного отказа от сохранения на диск. Кроме того, методы клиента Python `save()` и `bgsave()` позволяют принудительно сохранять снимки — либо с блокировкой до завершения сохранения методом `save()`, либо в фоновом режиме в случае `bgsave()`.

В этой главе была затронута лишь небольшая часть технологии Redis, ее типов данных и возможностей работы с ними. Если вы захотите узнать больше, в интернете можно найти много источников документации, включая https://redislabs.com и https://redis-py.readthedocs.io.

БЫСТРАЯ ПРОВЕРКА: ИСПОЛЬЗОВАНИЕ ХРАНИЛИЩ «КЛЮЧ–ЗНАЧЕНИЕ»

Для каких типов данных и приложений лучше всего подойдет хранилище «ключ–значение», такое как Redis?

23.7. Документы в MongoDB

Другая популярная база данных NoSQL — MongoDB — иногда называется *документной базой данных*, потому что она не делится на строки и столбцы, но используется для хранения документов. База данных MongoDB рассчитана на масштабирование по многим узлам в нескольких кластерах; потенциально она может обрабатывать миллиарды документов. В случае MongoDB документ хранится в формате BSON (Binary JSON), поэтому документ состоит из пар «ключ–значение» и выглядит как объект JSON или словарь Python. Следующий пример дает представление об использовании Python для взаимодействия с коллекциями MongoDB и документами, однако я должна предупредить: в ситуациях, требующих масштабирования и распределенного хранения данных, высокого темпа вставки, сложных и нестабильных схем и т. д., MongoDB будет превосходным вариантом. Однако во многих ситуациях MongoDB оказывается не лучшим выбором, поэтому вам стоит проанализировать свои потребности и доступные варианты перед тем, как принимать решение.

ЗАПУСК СЕРВЕРА MONGODB

Как и в случае с Redis, для экспериментов с MongoDB необходимо иметь доступ к серверу MongoDB. Сейчас доступны многочисленные облачные сервисы Mongo, но опять-таки, пока вы экспериментируете, разумнее ограничиться экземпляром Docker или установить MongoDB на принадлежащем вам сервере.

Как и в случае с Redis, проще всего запустить экземпляр Docker. Если программное обеспечение Docker уже установлено, для этого достаточно ввести команду `> docker run -p 27017:27017 mongo` в командной строке. В системе Linux эта задача решается менеджером пакетов, а на Mac — командой `brew install mongodb`. В системах семейства Windows версия MongoDB и инструкции по ее установке можно найти на сайте www.mongodb.com. Как и в случае с Redis, инструкции по настройке и запуску сервера можно найти в интернете.

Как и в случае с Redis, существует несколько клиентских библиотек для подключения к базе данных MongoDB. Чтобы получить представление о том, как они работают, взгляните на библиотеку `pymongo`. Прежде всего, библиотеку нужно установить; это можно сделать следующей командой:

```
> pip install pymongo
```

После того как вы установили pymongo, вы можете подключиться к серверу MongoDB. Создайте экземпляр MongoClient и укажите стандартную информацию для подключения:

```
>>> from pymongo import MongoClient
>>> mongo = MongoClient(host='localhost', port=27017)
```

host='localhost' и port=27017 — настройки по умолчанию, задавать их не обязательно

База данных MongoDB содержит коллекции, каждая из которых может содержать документы. Тем не менее базы данных и коллекции не обязательно создавать перед тем, как обращаться к ним. Если их не существует, то они будут созданы при вставке, иначе при попытке получения записей результаты не вернутся.

Чтобы протестировать клиент, создайте тестовый документ, который может представлять собой словарь Python:

```
>>> import datetime
>>> a_document = {'name': 'Jane',
...               'age': 34,
...               'interests': ['Python', 'databases', 'statistics'],
...               'date_added': datetime.datetime.now()
... }
>>> db = mongo.my_data
>>> collection = db.docs
>>> collection.find_one()
>>> db.collection_names()
[]
```

Выбирает базу данных (которая еще не была создана)

Выбирает коллекцию в базе данных (также еще не создана)

Ищет первый элемент; исключение не выдается, хотя ни коллекция, ни база данных еще не существуют

Здесь вы подключаетесь к базе данных и коллекции документов. В данном случае они не существуют, но будут созданы при обращении. Обратите внимание: хотя база данных и коллекция не существуют, никакие исключения не выдаются. Однако

когда вы запрашиваете список коллекций, вы получаете пустой список, потому что в коллекции ничего не было сохранено. Чтобы сохранить документ, используйте метод `insert()` коллекции, который возвращает уникальный идентификатор документа `ObjectId` в случае успешного выполнения операции:

```
>>> collection.insert(a_document)
ObjectId('59701cc4f5ef0516e1da0dec')  <-- Уникальное значение ObjectId
>>> db.collection_names()
['docs']
```

Теперь, когда документ был сохранен в коллекции `docs`, он отображается при запросе имен коллекций в базе данных. После того как документ будет сохранен в коллекции, вы можете использовать его для получения информации, для операций обновления, замены и удаления:

```
>>> collection.find_one()  <-- Получает первую запись
{'_id': ObjectId('59701cc4f5ef0516e1da0dec'), 'name': 'Jane', 'age': 34,
     'interests': ['Python', 'databases', 'statistics'], 'date_added':
     datetime.datetime(2017, 7, 19, 21, 59, 32, 752000)}
>>> from bson.objectid import ObjectId
>>> collection.find_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')})
{'_id': ObjectId('59701cc4f5ef0516e1da0dec'), 'name': 'Jane',
     'age': 34, 'interests': ['Python', 'databases',
     'statistics'], 'date_added': datetime.datetime(2017,
     7, 19, 21, 59, 32, 752000)}
>>> collection.update_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')},
     {"$set": {"name":"Ann"}})  <-- Обновляет запись в соответствии с содержимым объекта $set
<pymongo.results.UpdateResult object at 0x7f4ebd601d38>
>>> collection.find_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')})
{'_id': ObjectId('59701cc4f5ef0516e1da0dec'), 'name': 'Ann', 'age': 34,
     'interests': ['Python', 'databases', 'statistics'], 'date_added':
     datetime.datetime(2017, 7, 19, 21, 59, 32, 752000)}
>>> collection.replace_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')},
     {"name":"Ann"})  <-- Заменяет запись новым объектом
<pymongo.results.UpdateResult object at 0x7f4ebd601750>
>>> collection.find_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')})
{'_id': ObjectId('59701cc4f5ef0516e1da0dec'), 'name': 'Ann'}
>>> collection.delete_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')})
<pymongo.results.DeleteResult object at 0x7f4ebd601d80>
>>> collection.find_one()
```

Получает запись, соответствующую спецификации — в данном случае ObjectId

Удаляет запись по спецификации

Во-первых, обратите внимание, что MongoDB совпадает со словарями полей и их значениями. Словари также используются для обозначения операторов, таких как `$lt` (меньше) и `$gt` (больше), а для обновления — такие команды, как `$set`. Другая важная вещь заключается в том, что даже если запись была удалена и коллекция теперь пуста, она все еще существует, если только не была удалена:

```
>>> db.collection_names()
['docs']
>>> collection.drop()
>>> db.collection_names()
[]
```

Конечно, этим возможности MongoDB не ограничиваются. Кроме работы с отдельной записью, также существуют разновидности тех же команд для работы с несколькими записями, такие как `find_many` и `update_many`. MongoDB также поддерживает индексирование для повышения производительности и содержит методы для группировки, подсчета и агрегирования данных, а также встроенный метод отображения-свертки.

БЫСТРАЯ ПРОВЕРКА: ИСПОЛЬЗОВАНИЕ MONGODB

Вспомните различные примеры данных, встречавшиеся ранее, и другие типы данных, с которыми вы имели дело. Как вы думаете, какие из этих данных хорошо подошли бы для хранения в такой базе данных, как MongoDB? Будут ли другие данные явно неподходящими, и если да, то почему?

ПРАКТИЧЕСКАЯ РАБОТА 23: СОЗДАНИЕ БАЗЫ ДАННЫХ

Выберите один из наборов данных, рассматривавшихся в последних главах. Решите, какой тип базы данных лучше подойдет для хранения этих данных. Создайте эту базу данных и напишите код для загрузки данных. Затем выберите два самых распространенных и/или вероятных типа критериев поиска и напишите код для получения как одиночных, так и нескольких совпадающих записей.

Итоги

- В Python существует Database API (DB-API) — универсальный интерфейс для клиентов разных реляционных баз данных.
- Объектно-реляционное отображение, или ORM, позволяет еще больше стандартизировать код для разных баз данных.
- Использование ORM также позволяет работать с реляционными базами данных с использованием кода Python и объектов (вместо запросов SQL).
- Такие инструменты, как Alembic, работают в сочетании с ORM и позволяют использовать программный код для внесения обратимых изменений в схемах реляционных баз данных.
- Хранилища пар «ключ–значение» (например, Redis) обеспечивают быстрый доступ к данным в памяти.
- MongoDB обеспечивает превосходную масштабируемость без жесткой структуры реляционных баз данных.

24 Анализ данных

Эта глава охватывает следующие темы:

- ✓ Преимущества Python для обработки данных
- ✓ Jupyter Notebook
- ✓ pandas
- ✓ Агрегирование данных
- ✓ Участки с matplotlib

В нескольких последних главах рассматривались различные аспекты использования Python для получения и очистки данных. Пришло время рассмотреть некоторые средства, предоставляемые Python в области обработки и анализа данных.

24.1. Средства Python для анализа данных

В этой главе будут рассмотрены некоторые популярные средства анализа данных в языке Python: Jupyter Notebook, pandas и matplotlib. Я лишь кратко опишу некоторые возможности этих средств; моя цель — дать вам представление о том, что с ними можно делать, а также начальный набор инструментов для анализа данных в Python.

24.1.1. Преимущества анализа данных на языке Python

Python стал одним из ведущих языков теории обработки данных, а его популярность в этой области продолжает расти. Но как упоминалось ранее, Python не всегда является самым быстрым языком в отношении банального быстродействия. Некоторые библиотеки обработки данных (такие, как NumPy) в основном написаны на C и оптимизированы до такой степени, что скорость уже не является проблемой. Кроме того, такие факторы, как доступность и удобочитаемость кода, часто оказываются важнее простой скорости работы; сокращение затрат времени разработчика часто

выходит на первый план. Код Python обладает удобочитаемостью и доступностью, а собственные возможности языка в сочетании с инструментами, разработанными в сообществе Python, делают его невероятно мощным инструментом обработки и анализа данных.

24.1.2. Python лучше электронных таблиц

Электронные таблицы десятилетиями оставались фаворитами в области анализа данных. Специалисты, хорошо владеющие электронными таблицами, способны на действительно впечатляющие трюки: могут заставить объединять электронные таблицы в разные, но логически связанные с ними наборы данных, строить сводные таблицы, использовать подстановочные таблицы для связывания наборов данных и многое другое. Но хотя люди повсюду ежедневно проделывают массу полезной работы в электронных таблицах, у них есть свои ограничения, а Python поможет вам выйти за пределы этих ограничений.

Одно из таких ограничений, о котором я уже упоминала, связано с тем фактом, что у многих программ для работы с электронными таблицами существует лимит строк — в настоящее время около 1 миллиона строк, а для многих наборов данных этого уже недостаточно. Другое ограничение связано с центральной метафорой самой электронной таблицы. Электронная таблица представляет собой двумерную матрицу, состоящую из строк и столбцов, или в лучшем случае «стопку» таких матриц, что ограничивает возможности интерпретации и обработки сложных данных.

В языке Python разработчик может обойти ограничения электронных таблиц на программном уровне и манипулировать с данными так, как считает нужным. Вы можете объединять структуры данных Python — списки, кортежи, множества и словари — бесконечно гибкими способами или же создавать собственные классы, в которых будут упакованы точно те данные и поведение, которые вам нужны.

24.2. Jupyter Notebook

Один из самых полезных инструментов для анализа данных в Python не изменяет то, что делает сам язык, а изменяет то, как вы используете язык для взаимодействия с данными. Jupyter Notebook — веб-приложение, позволяющее создавать и распространять документы с работающим кодом, формулами, визуализациями и пояснительным текстом. Хотя в настоящее время поддерживается ряд других языков программирования, проект изначально появился в связи с IPython — альтернативной оболочкой для Python, разработанной научным сообществом.

Jupyter становится таким удобным и мощным инструментом благодаря тому , что вы взаимодействуете с ним в браузере. Он позволяет объединять текст с кодом, а также изменять и выполнять код в интерактивном режиме. Вы можете не только запускать и изменять отдельные фрагменты кода, но и сохранять полученные блокноты, а также передавать их другим.

Чтобы понять, что можно сделать с Jupyter Notebook, лучше всего начать экспериментировать с ним. Вы можете без труда запустить локальный процесс Jupyter на своей машине или же обратиться к сетевой версии. Некоторые возможности запуска Jupyter описаны во врезке.

СПОСОБЫ РАБОТЫ С JUPYTER

Сетевой запуск: обращение к сетевым экземплярам Jupyter всегда было одним из самых простых вариантов. В настоящее время Project Jupyter — сообщество, стоящее за Jupyter, — размещает бесплатные экземпляры по адресу https://jupyter.org/try. Здесь также можно найти демонстрационные блокноты и ядра для других языков. На момент написания книги бесплатные версии также были доступны на платформе Microsoft Azure по адресу https://notebooks.azure.com. Также доступно много других вариантов.

Локальный запуск: использовать сетевой экземпляр удобно, но запустить собственный экземпляр Jupyter на локальной машине не так уж сложно. Обычно для локальных версий в браузере открывается адрес localhost:8888.

Если вы используете Docker, вы можете выбрать один из нескольких контейнеров. Чтобы запустить контейнер с блокнотом анализа данных, используйте команду следующего вида:

```
docker run -it --rm -p 8888:8888 jupyter/datascience-notebook
```

Если же вы предпочитаете запустить экземпляр непосредственно в вашей системе, Jupyter легко устанавливается и запускается в виртуальной среде.

Системы macOS и Linux: откройте окно командной строки и введите следующие команды:

```
> python3 -m venv jupyter
> cd jupyter
> source bin/activate
> pip install jupyter
> jupyter-notebook
```

Системы Windows:

```
> python3 -m venv jupyter
> cd jupyter
> Scripts/bin/activate
> pip install jupyter
> Scripts/jupyter-notebook
```

Последняя команда должна запустить веб-приложение Jupyter Notebook и открыть окно браузера с этим приложением.

24.2.1. Запуск ядра

После того как приложение Jupyter будет установлено, запущено и открыто в браузере, необходимо запустить ядро Python. Одно из достоинств Jupyter заключается в том, что оно позволяет запускать несколько ядер одновременно. Вы можете запускать ядра разных версий Python и других языков, таких как R, Julia и даже Ruby.

Запустить ядро несложно. Щелкните на кнопке New и выберите в списке Python 3 (рис. 24.1).

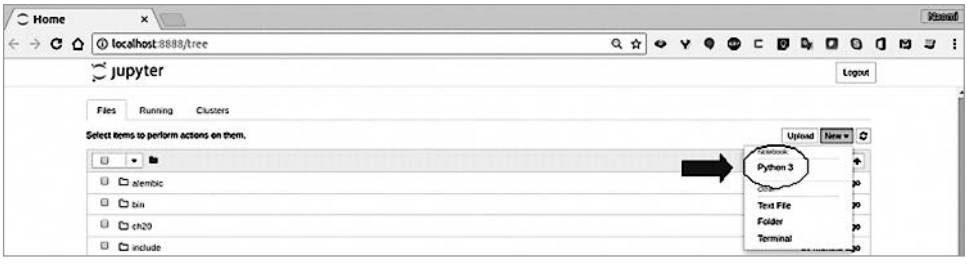


Рис. 24.1. Запуск ядра Python

24.2.2. Выполнение кода в ячейке

Когда ядро заработает, можно переходить к вводу и запуску кода Python. Вы сразу же заметите несколько отличий от обычной командной оболочки Python: отсутствует приглашение >>> из стандартной оболочки Python, а нажатие Enter только добавляет новые строки в ячейку. Чтобы выполнить код в ячейке, как показано на рис. 24.2, выберите команду Cell ▸ Run Cells и щелкните на кнопке Run слева от кнопки ↓ на панели кнопок или нажмите клавиши Alt+Enter. После того как вы поработаете в Jupyter Notebook в течение некоторого времени, комбинация Alt+Enter станет для вас вполне естественной.

Чтобы протестировать, как все это работает, введите код или выражение в первую ячейку нового блокнота и нажмите Alt+Enter.

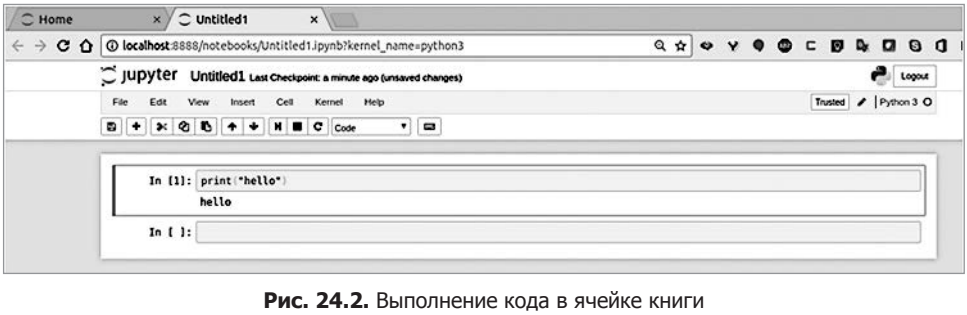


Рис. 24.2. Выполнение кода в ячейке книги

Как видите, весь вывод отображается непосредственно под ячейкой, а в приложении создается новая ячейка, готовая к получению ввода. Также следует заметить, что ячейки нумеруются в порядке выполнения.

ПОПРОБУЙТЕ САМИ: JUPYTER NOTEBOOK

Введите в блокноте код и поэкспериментируйте с его выполнением. Откройте меню Edit, Cell и Kernel и просмотрите содержащиеся в них команды. Когда код будет выполняться успешно, используйте меню Kernel для перезапуска ядра, повторите свои действия и используйте меню Cell для повторного запуска кода во всех ячейках.

24.3. Python и pandas

В процессе анализа и обработки данных выполняются некоторые типичные операции — загрузка данных в список или словарь, очистка и фильтрация данных. Многие из этих операций часто повторяются, выполняются по стандартным схемам, просты и часто однообразны. Если вы полагаете, что этот факт становится веским доводом для автоматизации этих задач, — вы в этом не одиноки. Один из инструментов обработки данных в Python, ставший фактически стандартным, — pandas — был создан именно для автоматизации рутинных задач обработки наборов данных.

24.3.1. Для чего используется pandas

Библиотека pandas создавалась для упрощения обработки и анализа табличных или реляционных данных. Она предоставляет стандартную инфраструктуру для хранения данных и удобные средства для часто выполняемых операций. В результате она представляет собой скорее расширение Python, нежели библиотеку, и изменяет подход к взаимодействию с данными. Когда вы вникнете в логику работы pandas, вы сможете проделывать всякие эффектные трюки и сэкономите себе немало времени. Тем не менее, чтобы научиться использовать pandas с максимальной эффективностью, потребуется время. Как это бывает со многими инструментами, если вы используете библиотеку pandas для тех целей, для которых она проектировалась, — она работает блестяще. Простые примеры, которые я приведу ниже, дают некоторое представление о том, насколько хорошо pandas подойдет для вашей конкретной ситуации.

24.3.2. Установка pandas

Pandas легко устанавливается с помощью `pip`. Часто pandas используется вместе с matplotlib для построения графиков, поэтому оба инструмента можно установить из командной строки виртуальной среды Jupyter с помощью кода:

```
> pip install pandas matplotlib
```

В ячейке Jupyter Notebook можно ввести команду:

```
In [ ]: !pip install pandas matplotlib
```

Если вы используете pandas, следующие три строки немного упростят вашу работу:

```
%matplotlib inline
import pandas as pd
import numpy as np
```

Первая строка — «волшебная» функция Jupyter, позволяющая matplotlib строить график данных в ячейке, в которой находится код (что очень полезно). Вторая строка импортирует pandas под псевдонимом pd, более компактным и распространенным среди пользователей pandas, последняя строка импортирует numpy. Хотя pandas отчасти зависит от numpy, этот пакет не будет явно использоваться

в следующих примерах, однако вам лучше привыкнуть к тому, чтобы импортировать его в любом случае.

24.3.3. Кадры данных

Одна из основных структур, используемых в pandas, — *кадр данных* (data frame). Кадр данных представляет собой матрицу — двумерную табличную структуру, похожую на таблицу реляционных баз данных, но находящуюся в памяти. Создать кадр данных несложно; от вас потребуются лишь некоторые данные. Чтобы первый пример был как можно проще, мы создадим для него матрицу 3×3. В Python такая таблица реализуется в виде списка списков:

```
grid = [[1,2,3], [4,5,6], [7,8,9]]
print(grid)
```

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

К сожалению, в Python такая матрица не будет похожа на таблицу, если не принять дополнительных мер. Посмотрим, что можно сделать с той же таблицей как с кадром данных pandas:

```
import pandas as pd
df = pd.DataFrame(grid)
print(df)
```

```
   0  1  2
0  1  2  3
1  4  5  6
2  7  8  9
```

Код достаточно элементарный. Все, что от вас требовалось, — преобразовать матрицу в кадр данных. Вывод уже больше напоминает традиционную таблицу, а у строк и столбцов есть номера. Конечно, запоминать, какие данные хранятся в том или ином столбце, утомительно, поэтому столбцам лучше присвоить имена:

```
df = pd.DataFrame(grid, columns=["one", "two", "three"] )
print(df)
```

```
   one  two  three
0    1    2      3
1    4    5      6
2    7    8      9
```

Польза от таких имен столбцов не сразу очевидна, но имена столбцов также позволяют выполнять еще один трюк pandas: выбирать столбцы по именам. Например, если вас интересует содержимое только столбца `"two"`, вы можете очень легко получить его:

```
print(df["two"])
```

```
0    2
1    5
2    8
Name: two, dtype: int64
```

Здесь уже экономится время по сравнению с Python. Чтобы получить только второй столбец матрицы, вам пришлось бы использовать генератор списка, да еще помнить о том, что индексы начинаются с нуля (а результат все равно выглядит не очень хорошо):

```
print([x[1] for x in grid])
[2, 5, 8]
```

Перебор значений столбцов кадра данных выполняется так же, как и перебор списка — при помощи генератора:

```
for x in df["two"]:
    print(x)
2
5
8
```

Неплохо для начала, но список столбцов в двойных квадратных скобках позволяет сделать больше — получить подмножество кадра данных, которое тоже является кадром данных. Вместо среднего столбца вы получаете первый и последний столбцы кадра данных, которые образуют другой кадр данных:

```
edges = df[["one", "three"]]
print(edges)
   one  three
0    1      3
1    4      6
2    7      9
```

Кадр данных также содержит несколько методов, которые применяют одну операцию и аргумент к каждому элементу кадра. Например, чтобы увеличить каждый элемент крайних столбцов кадра данных на 2, используйте метод `add()`:

```
print(edges.add(2))
   one  three
0    3      5
1    6      8
2    9     11
```

И этого результата можно добиться при помощи генераторов списков и/или вложенных циклов, но решение получается громоздким и неудобным. Понятно, что такая функциональность упростит вашу работу, особенно если вас больше интересует информация, содержащаяся в данных, а не процесс их обработки.

24.4. Очистка данных

В предыдущих главах рассматривались некоторые способы использования Python для очистки данных. Теперь, когда в общую картину добавилась библиотека pandas, я покажу, как применить ее функциональность для очистки данных. При описании следующих операций я также буду демонстрировать возможность выполнения тех

же операций в «чистом» коде Python, как чтобы показать, что дает использование pandas, так и чтобы объяснить, почему pandas не будет идеальным решением для каждой ситуации (или каждого пользователя).

24.4.1. Загрузка и сохранение данных в pandas

pandas содержит впечатляющую подборку методов для загрузки данных из разных источников. Библиотека поддерживает несколько форматов файлов (включая текстовые файлы с фиксированной шириной и разделителями, электронные таблицы, JSON, XML и HTML), но также возможно чтение из баз данных SQL, Google BiqQuery, HDF и даже из буфера обмена. Следует помнить, что многие из этих операций не являются частью pandas; работа pandas зависит от установки других библиотек для выполнения этих операций (например, SQLAlchemy для чтения из баз данных SQL). Это важно только в том случае, если что-то пойдет не так; часто проблема, которую приходится решать, лежит за пределами pandas и вам приходится разбираться с задействованной библиотекой.

Файлы JSON читаются методом `read_json()`:

```
mars = pd.read_json("mars_data_01.json")
```

Этот код возвращает кадр данных следующего вида:

```
                                report
abs_humidity                      None
atmo_opacity                     Sunny
ls                                 296
max_temp                            -1
max_temp_fahrenheit               30.2
min_temp                           -72
min_temp_fahrenheit              -97.6
pressure                           869
pressure_string                 Higher
season                        Month 10
sol                               1576
sunrise             2017-01-11T12:31:00Z
sunset              2017-01-12T00:46:00Z
terrestrial_date              2017-01-11
wind_direction                        --
wind_speed                        None
```

Чтобы ознакомиться с другим примером, демонстрирующим простоту чтения данных в pandas, загрузите данные из файла CSV с температурами из главы 21 и из файла JSON с метеорологическими данными Марса, использованными в главе 22. В первом случае используется метод `read_csv()`:

Обратите внимание: символ \ в конце строки заголовка указывает на то, что таблица не помещается в одной строке, а ниже выводятся дополнительные столбцы

```
temp = pd.read_csv("temp_data_01.csv")

         4      5    6     7     8      9    10    11    12     13    14 \
```

```
0  1979/01/01  17.48  994   6.0  30.5   2.89  994 -13.6  15.8    NaN    0
1  1979/01/02   4.64  994  -6.4  15.8  -9.03  994 -23.6   6.6    NaN    0
2  1979/01/03  11.05  994  -0.7  24.7  -2.17  994 -18.3  12.9    NaN    0
3  1979/01/04   9.51  994   0.2  27.6  -0.43  994 -16.3  16.3    NaN    0
4  1979/05/15  68.42  994  61.0  75.1  51.30  994  43.3  57.0    NaN    0
5  1979/05/16  70.29  994  63.4  73.5  48.09  994  41.1  53.0    NaN    0
6  1979/05/17  75.34  994  64.0  80.5  50.84  994  44.3  55.7  82.60    2
7  1979/05/18  79.13  994  75.5  82.1  55.68  994  50.0  61.1  81.42  349
8  1979/05/19  74.94  994  66.9  83.1  58.59  994  50.9  63.2  82.87   78

     15    16      17
0   NaN   NaN  0.0000
1   NaN   NaN  0.0000
2   NaN   NaN  0.0000
3   NaN   NaN  0.0000
4   NaN   NaN  0.0000
5   NaN   NaN  0.0000
6  82.4  82.8  0.0020
7  80.2  83.4  0.3511
8  81.6  85.2  0.0785
```

Очевидно, загрузка файла за один шаг — это удобно, и вы видите, что у pandas нет проблем с загрузкой файлов. Также видно, что пустой первый столбец был преобразован в `NaN` (нечисло). Для некоторых значений возникает уже знакомая проблема с `'Missing'`, поэтому будет логично преобразовать эти значения `'Missing'` в `NaN`:

```
temp = pd.read_csv("temp_data_01.csv", na_values=['Missing'])
```

Параметр `na_values` управляет тем, какие значения будут преобразовываться в `NaN` при загрузке. В данном случае добавляется строка `'Missing'`, чтобы строка кадра данных была преобразована из

```
NaN  Illinois  17  Jan 01, 1979  1979/01/01  17.48  994  6.0  30.5  2.89994
     -13.6  15.8  Missing  0  Missing  Missing  0.00%
```

в

```
NaN  Illinois  17  Jan 01, 1979  1979/01/01  17.48  994  6.0  30.5  2.89994
     -13.6  15.8  NaN0  NaN  NaN  0.00%
```

Такое преобразование может быть особенно полезным, если вам достался один из тех файлов данных, в которых по какой-то неведомой причине «отсутствие данных» обозначается разными способами: `NA`, `N/A`, `?`, `-` и т. д. В таком случае вам стоит просмотреть данные, узнать, какие обозначения используются, а затем перезагрузить файл, задав значение параметра `na_values` для стандартизации всех этих вариантов в `NaN`.

Сохранение данных

Если вам потребуется сохранить содержимое кадра данных, объекты кадров данных в pandas поддерживают столь же широкую подборку методов. Простой матричный кадр данных из нашего примера можно записать несколькими способами. Команда

```
df.to_csv("df_out.csv", index=False)
```

Присваивание index значения False означает, что индексы строк записываться не должны

записывает файл, который выглядит примерно так:

```
one,two,three
1,2,3
4,5,6
7,8,9
```

Аналогичным образом матрицу данных можно преобразовать в объект JSON или записать ее в файл:

Если передать путь к файлу в аргументе, разметка JSON будет записана в указанный файл (вместо того, чтобы возвращаться при вызове)

```
df.to_json()
'{"one":{"0":1,"1":4,"2":7},"two":{"0":2,"1":5,"2":8},"three":{"0":3,"1":6,"2
":9}}'
```

24.4.2. Очистка данных

Преобразование конкретного набора значений в `NaN` при загрузке — очень простое средство очистки данных, которое тривиально выполняется благодаря pandas. Кроме этого, кадры данных поддерживают ряд операций, которые сокращают уровень рутины при очистке данных. Чтобы понять, как работают эти операции, снова откройте файл CSV с температурными данными, но на этот раз вместо определения имен столбцов по заголовкам используйте функцию `range()` с параметром `names` и назначьте им номера (чтобы к ним было удобнее обращаться). Возможно, вы также помните из предыдущего примера, что первое поле каждой строки — поле `"Notes"` — не содержит данных и загружается значениями `NaN`. Хотя этот столбец можно игнорировать, было бы проще, если бы его вообще не было. Вы можете снова воспользоваться функцией `range()`, но на этот раз начать с 1, чтобы приказать pandas загрузить все столбцы, кроме первого. Но если вы точно знаете, что все значения относятся к штату Иллинойс и вас не интересует поле даты в длинном формате, можно начать с 4 для удобства работы:

```
temp = pd.read_csv("temp_data_01.csv", na_values=['Missing'], header=0,
names=range(18), usecols=range(4,18))
print(temp)
```

Присваивание header=0 отключает чтение меток столбцов из заголовка

```
           4      5    6     7     8      9    10    11    12     13   14 \
0  1979/01/01  17.48  994   6.0  30.5   2.89  994 -13.6  15.8    NaN    0
1  1979/01/02   4.64  994  -6.4  15.8  -9.03  994 -23.6   6.6    NaN    0
2  1979/01/03  11.05  994  -0.7  24.7  -2.17  994 -18.3  12.9    NaN    0
3  1979/01/04   9.51  994   0.2  27.6  -0.43  994 -16.3  16.3    NaN    0
4  1979/05/15  68.42  994  61.0  75.1  51.30  994  43.3  57.0    NaN    0
5  1979/05/16  70.29  994  63.4  73.5  48.09  994  41.1  53.0    NaN    0
6  1979/05/17  75.34  994  64.0  80.5  50.84  994  44.3  55.7  82.60    2
7  1979/05/18  79.13  994  75.5  82.1  55.68  994  50.0  61.1  81.42  349
```

```
8  1979/05/19  74.94  994  66.9  83.1  58.59  994  50.9  63.2  82.87   78

     15    16      17
0   NaN   NaN   0.00%
1   NaN   NaN   0.00%
2   NaN   NaN   0.00%
3   NaN   NaN   0.00%
4   NaN   NaN   0.00%
5   NaN   NaN   0.00%
6  82.4  82.8   0.20%
7  80.2  83.4  35.11%
8  81.6  85.2   7.85%
```

Теперь кадр данных состоит только из тех столбцов, с которыми вы собираетесь работать. Но проблемы еще остаются: последний столбец, содержащий процент покрытия для теплового индекса, остается строкой, завершающейся знаком `%` вместо фактического значения. Проблема наглядно видна, если вы взглянете на значение столбца 17 первой строки:

```
temp[17][0]
'0.00%'
```

Чтобы решить эту проблему, необходимо сделать две вещи: удалить `%` в конце строки и преобразовать значение из строки в число. Кроме того, если значение нужно представить в виде дробной величины, его следует разделить на 100. Первый шаг делается просто — pandas позволяет использовать одну команду для повторения операции со столбцом:

```
temp[17] = temp[17].str.strip("%")
temp[17][0]
'0.00'
```

Этот код получает столбец и вызывает для него строковую операцию `strip()`, чтобы убрать завершающий символ `%`. Если теперь взглянуть на первое (или любое другое) значение в столбце, вы увидите, что нежелательный знак процента пропал. Также стоит заметить, что того же результата можно добиться при помощи других операций, например `replace("%", "")`.

Вторая операция преобразует строку в числовое значение. И снова pandas позволяет выполнить эту операцию всего одной командой:

```
temp[17] = pd.to_numeric(temp[17])
temp[17][0]
0.0
```

Теперь значения столбца 17 стали числовыми, и при желании можно воспользоваться методом `div()` для завершения работы по преобразованию этих значений в дробные величины:

```
temp[17] = temp[17].div(100)
temp[17]

0    0.0000
```

```
1    0.0000
2    0.0000
3    0.0000
4    0.0000
5    0.0000
6    0.0020
7    0.3511
8    0.0785
Name: 17, dtype: float64
```

Собственно, того же результата можно добиться в одной строке, объединив все три операции:

```
temp[17] = pd.to_numeric(temp[17].str.strip("%")).div(100)
```

Это очень простой пример, но он дает представление о том, насколько pandas может упростить очистку данных. pandas поддерживает различные операции для преобразования данных, а также возможность применения пользовательских функций. Трудно представить себе сценарий, в котором библиотека pandas не сделала бы очистку данных более удобной.

Круг возможностей pandas чрезвычайно широк, но в интернете существуют разнообразные учебники и видеокурсы, а на сайте http://pandas.pydata.org имеется превосходная документация.

ПОПРОБУЙТЕ САМИ: ОЧИСТКА ДАННЫХ С PANDAS И БЕЗ

Поэкспериментируйте с операциями, упомянутыми выше. После того как последний столбец был преобразован в дробное значение, как бы вы преобразовали его обратно в строку с завершающим знаком `%`?

Для сравнения загрузите те же данные в простой список Python при помощи модуля `csv` и примените те же изменения в «чистом» коде Python.

24.5. Агрегирование и преобразования данных

Вероятно, предыдущий пример дал вам некоторое представление о том, как pandas позволяет выполнять относительно сложные операции с данными всего несколькими командами. Как и следовало ожидать, этот уровень функциональности также доступен для агрегирования данных. В этом разделе будут рассмотрены простые примеры агрегирования данных для демонстрации некоторых возможностей. И хотя вариантов очень много, я сосредоточусь на слиянии кадров данных, выполнении простого агрегирования, группировке и фильтрации данных.

24.5.1. Слияние кадров данных

В процессе обработки данных довольно часто возникает необходимость в связывании двух наборов. Допустим, один файл содержит количество звонков,

совершенных участниками группы продаж за месяц, а в другом файле хранятся данные по объемам продаж для их территорий:

```
calls = pd.read_csv("sales_calls.csv")
print(calls)
```

```
   Team member  Territory  Month  Calls
0        Jorge          3      1    107
1        Jorge          3      2     88
2        Jorge          3      3     84
3        Jorge          3      4    113
4          Ana          1      1     91
5          Ana          1      2    129
6          Ana          1      3     96
7          Ana          1      4    128
8          Ali          2      1    120
9          Ali          2      2     85
10         Ali          2      3     87
11         Ali          2      4     87
```

```
revenue = pd.read_csv("sales_revenue.csv")
print(revenue)
```

```
    Territory  Month  Amount
0           1      1   54228
1           1      2   61640
2           1      3   43491
3           1      4   52173
4           2      1   36061
5           2      2   44957
6           2      3   35058
7           2      4   33855
8           3      1   50876
9           3      2   57682
10          3      3   53689
11          3      4   49173
```

Очевидно, связать доходы с активностью участников команды будет очень полезно. Файлы очень простые, однако их слияние в простом коде Python — не совсем тривиальная задача. В pandas имеется функция для слияния двух кадров данных:

```
calls_revenue = pd.merge(calls, revenue, on=['Territory', 'Month'])
```

Функция `merge` создает новый кадр данных, объединяя два кадра по столбцам, заданным в поле `column`. Она работает примерно так же, как и операция JOIN в реляционных базах данных: вы получаете таблицу, объединяющую столбцы из двух файлов:

```
print(calls_revenue)
```

```
   Team member  Territory  Month  Calls  Amount
0        Jorge          3      1    107   50876
1        Jorge          3      2     88   57682
2        Jorge          3      3     84   53689
```

```
3         Jorge           3      4    113   49173
4           Ana           1      1     91   54228
5           Ana           1      2    129   61640
6           Ana           1      3     96   43491
7           Ana           1      4    128   52173
8           Ali           2      1    120   36061
9           Ali           2      2     85   44957
10          Ali           2      3     87   35058
11          Ali           2      4     87   33855
```

В данном примере между строками двух таблиц имеется однозначное соответствие, но функция `merge` также может выполнять слияния типа «один ко многим» и «многие ко многим», а также правосторонние и левосторонние объединения.

БЫСТРАЯ ПРОВЕРКА: СЛИЯНИЕ НАБОРОВ ДАННЫХ

Как бы вы реализовали слияние таких наборов данных на Python?

24.5.2. Выбор данных

Также может быть полезно отбирать (или фильтровать) строки кадра данных по некоторому условию. Допустим, в примере с данными продаж вы хотите получить данные только для территории 3. Это тоже делается легко:

```
print(calls_revenue[calls_revenue.Territory==3])
```

```
  Team member  Territory  Month  Calls  Amount
0       Jorge          3      1    107   50876
1       Jorge          3      2     88   57682
2       Jorge          3      3     84   53689
3       Jorge          3      4    113   49173
```

В этом примере выбираются только те строки, в которых код территории равен 3, но для этого указанное выражение `revenue.Territory==3` используется в качестве индекса кадра данных. С точки зрения «традиционного» кода Python такое использование бессмысленно и незаконно, но для кадров данных pandas оно работает и обеспечивает намного более компактную запись.

Конечно, возможны и более сложные выражения. Если вы захотите выбрать только те строки, в которых объем продаж на один звонок превышает 500, используйте следующее выражение:

```
print(calls_revenue[calls_revenue.Amount/calls_revenue.Calls>500])
```

```
  Team member  Territory  Month  Calls  Amount
1       Jorge          3      2     88   57682
2       Jorge          3      3     84   53689
4         Ana          1      1     91   54228
9         Ali          2      2     85   44957
```

Что еще лучше, вы можете вычислить этот столбец и добавить его в кадр данных аналогичной операцией:

```
calls_revenue['Call_Amount'] = calls_revenue.Amount/calls_revenue.Calls
print(calls_revenue)
```

```
   Team member  Territory  Month  Calls  Amount  Call_Amount
0        Jorge          3      1    107   50876   475.476636
1        Jorge          3      2     88   57682   655.477273
2        Jorge          3      3     84   53689   639.154762
3        Jorge          3      4    113   49173   435.159292
4          Ana          1      1     91   54228   595.912088
5          Ana          1      2    129   61640   477.829457
6          Ana          1      3     96   43491   453.031250
7          Ana          1      4    128   52173   407.601562
8          Ali          2      1    120   36061   300.508333
9          Ali          2      2     85   44957   528.905882
10         Ali          2      3     87   35058   402.965517
11         Ali          2      4     87   33855   389.137931
```

И снова встроенная логика pandas заменяет более громоздкую структуру «обычного» Python.

БЫСТРАЯ ПРОВЕРКА: ВЫБОР В PYTHON

Какую программную структуру Python вы бы использовали для выбора только тех строк, которые удовлетворяют заданным условиям?

24.5.3. Группировка и агрегирование

Как и следовало ожидать, pandas также содержит разнообразные инструменты для обобщения и агрегирования данных. В частности, для вычисления суммы, среднего значения, медианы, минимума и максимума по столбцу используются методы с четкими, понятными именами:

```
print(calls_revenue.Calls.sum())
print(calls_revenue.Calls.mean())
print(calls_revenue.Calls.median())
print(calls_revenue.Calls.max())
print(calls_revenue.Calls.min())
```

```
1215
101.25
93.5
129
84
```

Если, например, вы хотите получить все строки, для которых величина продаж на клик превышает медиану, объедините этот прием с операцией выбора:

```
print(calls_revenue.Call_Amount.median())
print(calls_revenue[calls_revenue.Call_Amount >=
```

```
calls_revenue.Call_Amount.median()])

464.2539427570093
  Team member  Territory  Month  Calls  Amount  Call_Amount
0       Jorge          3      1    107   50876   475.476636
1       Jorge          3      2     88   57682   655.477273
2       Jorge          3      3     84   53689   639.154762
4         Ana          1      1     91   54228   595.912088
5         Ana          1      2    129   61640   477.829457
9         Ali          2      2     85   44957   528.905882
```

Кроме возможности вычисления сводных значений, часто бывает полезно группировать данные по содержимому других столбцов. В этом простом примере для группировки данных можно воспользоваться методом `groupby`. Допустим, вы хотите узнать суммарное количество звонков и объемы продаж по месяцам или по территориям. Тогда эти поля используются с методом `groupby` кадра данных:

```
print(calls_revenue[['Month', 'Calls', 'Amount']].groupby(['Month']).sum())

       Calls  Amount
Month
1        318  141165
2        302  164279
3        267  132238
4        328  135201

print(calls_revenue[['Territory', 'Calls',
      'Amount']].groupby(['Territory']).sum())

           Calls  Amount
Territory
1            444  211532
2            379  149931
3            392  211420
```

В обоих случаях вы выбираете столбцы, по которым производится агрегирование, группируете их по значениям одного из этих столбцов и (в данном примере) суммируете значения по каждой группе. Также можно использовать любые другие методы, упоминавшиеся ранее в этой главе.

Несмотря на свою простоту, эти примеры демонстрируют некоторые возможности обработки и выбора данных в pandas. Если эти концепции соответствуют вашим потребностям, вы сможете узнать больше в документации pandas по адресу http://pandas.pydata.org.

ПОПРОБУЙТЕ САМИ: ГРУППИРОВКА И АГРЕГИРОВАНИЕ

Поэкспериментируйте с pandas и данными из предыдущих примеров. Сможете ли вы получить информацию о звонках и объемах продаж, сгруппированную как по участникам команды, так и по месяцам?

24.6. Графическое представление данных

У pandas есть еще одна чрезвычайно интересная возможность: очень простые средства построения графиков и диаграмм для кадра данных. Хотя в Python и Jupyter Notebook предусмотрены разнообразные возможности построения графиков, pandas может использовать matplotlib прямо из кадра данных. Вспомните: в начале сеанса Jupyter одной из первых введенных команд была «волшебная» команда Jupyter, включавшая использование matplotlib для построения графиков «на месте»:

```
%matplotlib inline
```

Рассмотрим примеры графического представления данных (рис. 24.3). Продолжая пример с продажами, если вы хотите построить диаграмму средних продаж за квартал по территориям, вы можете включить ее прямо в блокнот; для этого достаточно добавить вызов `.plot.bar()`:

```
calls_revenue[['Territory', 'Calls']].groupby(['Territory']).sum().plot.bar()
```

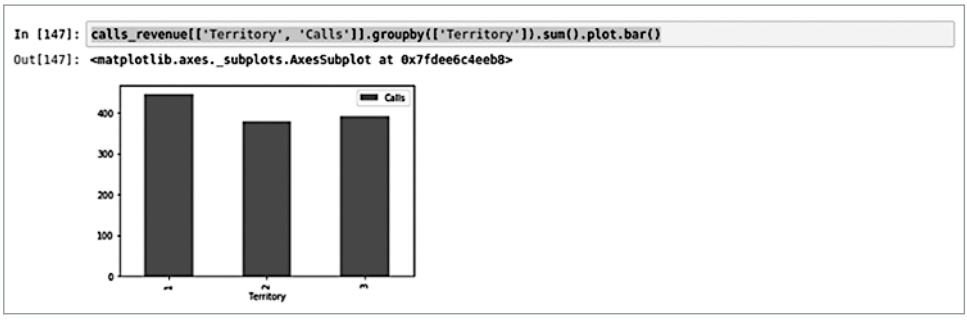


Рис. 24.3. Столбиковая диаграмма кадра данных pandas в Jupyter Notebook

Возможны и другие варианты. Вызов `plot()` сам по себе или в виде `.plot.line()` строит график, вызов `.plot.pie()` создает круговую диаграмму и т. д.

Благодаря сочетанию pandas и matplotlib построить диаграммы и графики в Jupyter Notebook совсем несложно. Однако стоит отметить, что при всей простоте многие задачи графического представления данных решаются не очень хорошо.

ПОПРОБУЙТЕ САМИ: ПОСТРОЕНИЕ ГРАФИКА

Постройте линейчатый график среднего ежемесячного объема продаж на один звонок.

24.7. Когда pandas использовать не рекомендуется

Приведенные примеры демонстрируют лишь малую часть тех возможностей, которые pandas может предоставить в области очистки, анализа и обработки данных. Как упоминалось в начале этой главы, pandas — превосходный инструментарий, отлично справляющийся с теми задачами, для которых он проектировался. Однако это не означает, что pandas идеально подходит для всех пользователей во всех ситуациях.

Существуют причины, по которым вы можете предпочесть традиционный код Python (или другой инструмент). Во-первых, как я уже говорила, полноценное освоение pandas можно сравнить с изучением нового языка, для чего у вас может не быть времени или желания. Кроме того, pandas может быть не лучшим решением для некоторых реальных условий, особенно для очень больших наборов данных или данных, которые трудно преобразовать в форматы, наиболее подходящие для pandas. Например, обработка огромных наборов с информацией о товарах вряд ли сильно выиграет от применения pandas; то же относится и к базовой обработке потока транзакций.

Мораль: вы должны разумно выбирать свои инструменты в зависимости от решаемой проблемы. Во многих случаях pandas действительно упростит вашу работу с данными, но в других ситуациях может лучше подойти обычный код Python.

Итоги

- Python предоставляет много удобных средств для обработки данных, включая возможность обработки очень больших наборов данных и гибкий выбор средств их обработки для конкретных потребностей.
- Jupyter Notebook — удобный инструмент работы с Python из браузера, который также упрощает графическое представление данных.
- Библиотека pandas значительно упрощает многие стандартные операции обработки данных, включая очистку, объединение и обобщение данных.
- pandas также упрощает построение несложных диаграмм и графиков.

Практический пример

В этом конкретном примере мы последовательно рассмотрим использование Python для получения данных, их очистки и представления в графическом виде. Проект получился небольшим, но он объединяет возможности языка, о которых говорилось выше, и позволяет проследить за ходом работы над проектом от начала до конца. Почти на каждом шаге кратко упоминаются альтернативы и возможные усовершенствования.

Глобальные изменения температуры — тема неоднозначная, но ее обсуждения основаны на данных глобального масштаба. Предположим, вы хотите узнать, что происходит с температурой рядом с местом вашего проживания. Одно из возможных решений — получить исторические данные для вашего местоположения, обработать их и нанести на график, чтобы увидеть, что же происходит.

КОД ПРАКТИЧЕСКОГО ПРИМЕРА

Проект этой главы был реализован с использованием Jupyter Notebook (см. главу 24). Если вы используете Jupyter, вы найдете мой блокнот (с текстом и кодом) в архиве исходного кода под именем Case Study.ipynb. Также можно выполнить код в стандартной оболочке Python; версия, поддерживающая эту оболочку, хранится в исходном коде под именем Case Study.py.

К счастью, в интернете имеются бесплатные источники исторических метеорологических данных. Мы воспользуемся данными из глобальной климатической базы (GHCN). Вы также можете найти другие источники с данными в других форматах, но основные этапы и процессы, которые мы здесь рассмотрим, применимы к любому набору данных.

Загрузка данных

Прежде всего необходимо загрузить данные. Архив ежедневных исторических погодных данных доступен по адресу https://www1.ncdc.noaa.gov/pub/data/ghcn/daily/ в форме огромного массива данных. Прежде всего, нужно разобраться, какие файлы нам нужны и где они хранятся; затем можно переходить к загрузке. Получив данные, можно переходить к обработке и, в конечном итоге, к отображению результатов.

Чтобы загрузить файлы, доступ к которым осуществляется через HTTPS, понадобится библиотека `requests`. Она устанавливается командой `pip install requests` в командной строке. После установки `requests` прежде всего загрузите файл readme.txt, в котором могут быть описаны форматы и местонахождение нужных вам данных:

```
# import requests

import requests
# get readme.txt file

r = requests.get('https://www1.ncdc.noaa.gov/pub/data/ghcn/daily/readme.txt')
readme = r.text.
```

Заглянув в файл readme, вы увидите код следующего вида:

```
print(readme)
README FILE FOR DAILY GLOBAL HISTORICAL CLIMATOLOGY NETWORK (GHCN-DAILY)
Version 3.22

-------------------------------------------------------------------------------
How to cite:

Note that the GHCN-Daily dataset itself now has a DOI (Digital Object Identifier)
so it may be relevant to cite both the methods/overview journal article as well
as the specific version of the dataset used.

The journal article describing GHCN-Daily is:
Menne, M.J., I. Durre, R.S. Vose, B.E. Gleason, and T.G. Houston, 2012:  An
     overview
of the Global Historical Climatology Network-Daily Database.  Journal of
     Atmospheric
and Oceanic Technology, 29, 897-910, doi:10.1175/JTECH-D-11-00103.1.

To acknowledge the specific version of the dataset used, please cite:
Menne, M.J., I. Durre, B. Korzeniewski, S. McNeal, K. Thomas, X. Yin, S.
     Anthony, R. Ray,
R.S. Vose, B.E.Gleason, and T.G. Houston, 2012: Global Historical Climatology
     Network -
Daily (GHCN-Daily), Version 3. [indicate subset used following decimal,
e.g. Version 3.12].
NOAA National Climatic Data Center. http://doi.org/10.7289/V5D21VHZ [access
     date].
```

Нас интересует раздел II, в котором перечислено содержимое:

```
II. CONTENTS OF ftp://ftp.ncdc.noaa.gov/pub/data/ghcn/daily

all:                Directory with ".dly" files for all of GHCN-Daily
gsn:                Directory with ".dly" files for the GCOS Surface Network
                    (GSN)
hcn:                  Directory with ".dly" files for U.S. HCN
by_year:              Directory with GHCN Daily files parsed into yearly
                      subsets with observation times where available. See the
              /by_year/readme.txt and
              /by_year/ghcn-daily-by_year-format.rtf
              files for further information
grid:                 Directory with the GHCN-Daily gridded dataset known
                      as HadGHCND
papers:           Directory with pdf versions of journal articles relevant
```

```
                    to the GHCN-Daily dataset
figures:          Directory containing figures that summarize the inventory
                    of GHCN-Daily station records

ghcnd-all.tar.gz:  TAR file of the GZIP-compressed files in the "all"
     directory
ghcnd-gsn.tar.gz:  TAR file of the GZIP-compressed "gsn" directory
ghcnd-hcn.tar.gz:  TAR file of the GZIP-compressed "hcn" directory

ghcnd-countries.txt:  List of country codes (FIPS) and names
ghcnd-inventory.txt:  File listing the periods of record for each station and

                    element
ghcnd-stations.txt:   List of stations and their metadata (e.g., coordinates)
ghcnd-states.txt:     List of U.S. state and Canadian Province codes
                    used in ghcnd-stations.txt
ghcnd-version.txt:    File that specifies the current version of GHCN Daily

readme.txt:           This file
status.txt:           Notes on the current status of GHCN-Daily
```

Просматривая список доступных файлов, мы видим, что в файле ghcnd-inventory.txt приведен список периодов наблюдений для каждой станции, он поможет вам найти хороший набор данных; в файле ghcnd-stations.txt перечислены станции, среди которых следует найти станцию, расположенную ближе всего к вам. С этих двух файлов следует начать:

```
II. CONTENTS OF ftp://ftp.ncdc.noaa.gov/pub/data/ghcn/daily

all:                  Directory with ".dly" files for all of GHCN-Daily
gsn:                  Directory with ".dly" files for the GCOS Surface Network
                      (GSN)
hcn:                  Directory with ".dly" files for U.S. HCN
by_year:              Directory with GHCN Daily files parsed into yearly
                      subsets with observation times where available. See the
              /by_year/readme.txt and
              /by_year/ghcn-daily-by_year-format.rtf
              files for further information
grid:                 Directory with the GHCN-Daily gridded dataset known
                      as HadGHCND
papers:             Directory with pdf versions of journal articles relevant
                      to the GHCN-Daily dataset
figures:            Directory containing figures that summarize the inventory
                      of GHCN-Daily station records
ghcnd-all.tar.gz:   TAR file of the GZIP-compressed files in the "all"
                    directory
ghcnd-gsn.tar.gz:   TAR file of the GZIP-compressed "gsn" directory
ghcnd-hcn.tar.gz:   TAR file of the GZIP-compressed "hcn" directory

ghcnd-countries.txt:  List of country codes (FIPS) and names
ghcnd-inventory.txt:  File listing the periods of record for each station and
                      element
ghcnd-stations.txt:   List of stations and their metadata (e.g., coordinates)
```

```
ghcnd-states.txt:     List of U.S. state and Canadian Province codes
                      used in ghcnd-stations.txt
ghcnd-version.txt:    File that specifies the current version of GHCN Daily

readme.txt:           This file
status.txt:           Notes on the current status of GHCN-Daily

# Получение файлов периодов наблюдений и станций

r = requests.get('https://www1.ncdc.noaa.gov/pub/data/ghcn/daily/ghcnd-
     inventory.txt')
inventory_txt = r.text
r = requests.get('https://www1.ncdc.noaa.gov/pub/data/ghcn/daily/ghcnd-
     stations.txt')
stations_txt = r.text
```

Полученные файлы можно сохранить на локальном диске, чтобы их не пришлось загружать заново, если вам потребуется вернуться к исходным данным:

```
# сохранение файлов на диске на случай, если они еще понадобятся снова

with open("inventory.txt", "w") as inventory_file:
    inventory_file.write(inventory_txt)

with open("stations.txt", "w") as stations_file:
    stations_file.write(stations_txt)
```

Начнем с файла периодов наблюдений. Вот как выглядят первые 137 символов:

```
print(inventory_txt[:137])
ACW00011604  17.1167  -61.7833 TMAX 1949 1949
ACW00011604  17.1167  -61.7833 TMIN 1949 1949
ACW00011604  17.1167  -61.7833 PRCP 1949 1949
Из раздела VII файла readme.txt следует, что файл периодов наблюдений имеет
следующий формат:

VII. FORMAT OF "ghcnd-inventory.txt"

------------------------------
Variable   Columns   Type
------------------------------
ID            1-11   Character
LATITUDE     13-20   Real
LONGITUDE    22-30   Real
ELEMENT      32-35   Character
FIRSTYEAR    37-40   Integer
LASTYEAR     42-45   Integer
------------------------------
```

Переменные имеют следующие определения:

```
ID         идентификационный код станции. За полным списком станций
           и их метаданными обращайтесь к файлу ghcnd-stations.txt.
LATITUDE   широта станции (в градусах).
```

```
LONGITUDE  долгота станции (в градусах).
ELEMENT    тип элемента. За определениями элементов обращайтесь к разделу III.
FIRSTYEAR  первый год непомеченных данных для элемента.
LASTYEAR   последний год непомеченных данных для элемента.
```

Можно сказать, что список периодов наблюдений содержит большую часть информации, необходимой для поиска интересующей вас станции. Ближайшие станции можно найти по широте и долготе, а затем воспользоваться полями FIRSTYEAR и LASTYEAR для нахождения станции с записями, покрывающими длинный период времени.

Остается единственный вопрос: какая информация содержится в поле ELEMENT? Файл рекомендует обратиться к разделу III. В разделе III (который будет более подробно рассмотрен позднее) содержится следующее описание основных элементов:

```
ELEMENT - тип элемента. Всего определены пять основных элементов, а также
          ряд дополнительных элементов.

       Пять основных элементов:

           PRCP = осадки (в десятых долях миллиметра)
           SNOW = количество снеговых осадков (мм)
           SNWD = толщина снежного покрова (мм)
           TMAX = максимальная температура (в десятых долях градусов C)
           TMIN = минимальная температура (в десятых долях градусов C)
```

Для целей нашего примера нужны элементы TMAX и TMIN, содержащие максимальную и минимальную температуру в десятых долях градусов Цельсия.

Разбор данных периодов наблюдения

Файл readme.txt сообщает, какая информация хранится в файле периодов наблюдений, чтобы вы могли разобрать данные в более удобный формат. Конечно, разобранные данные можно просто сохранить в формате списка списков или списка кортежей, но при совсем небольших дополнительных усилиях можно воспользоваться классом `namedtuple` из библиотеки коллекций для создания класса с именованными атрибутами:

```
# Разбор в именованные кортежи

# namedtuple используется для создания пользовательского класса Inventory
from collections import namedtuple
Inventory = namedtuple("Inventory", ['station', 'latitude', 'longitude',
     'element', 'start', 'end'])
```

Использовать созданный класс `Inventory` несложно; вы просто создаете каждый экземпляр с соответствующими значениями, которые в данном случае образуют разобранную строку данных периодов наблюдения.

Разбор выполняется в два этапа. Сначала необходимо получить сегменты строки в соответствии с заданными размерами полей. Обратившись к описанию полей

в файле readme, мы видим, что между полями имеется дополнительное пространство, которое необходимо учитывать при создании любой процедуры разбора. В данном случае, поскольку вы задаете каждый сегмент, дополнительные пробелы игнорируются. Кроме того, поскольку размеры полей STATION и ELEMENT точно соответствуют хранящимся в них значениям, вам не нужно беспокоиться об удалении из них лишних пробелов.

Второе, что было бы неплохо сделать, — преобразовать значения широты и долготы к типу с плавающей точкой, а начальный и конечный коды — к целочисленному типу. Это можно сделать на более поздней стадии очистки данных, если целостность данных нарушена, а в строках присутствуют значения, которые не могут быть преобразованы, лучше подождать. Но в этом случае данные позволяют выполнить преобразования на шаге разбора, поэтому мы сделаем это прямо сейчас:

```
# Разбор данных периодов наблюдений и преобразование значений в числа

inventory = [Inventory(x[0:11], float(x[12:20]), float(x[21:30]), x[31:35],
     int(x[36:40]), int(x[41:45]))
             for x in inventory_txt.split("\n") if x.startswith("US")]

for line in inventory[:5]:
    print(line)
Inventory(station='US009052008', latitude=43.7333, longitude=-96.6333,
     element='TMAX', start=2008, end=2016)
Inventory(station='US009052008', latitude=43.7333, longitude=-96.6333,
     element='TMIN', start=2008, end=2016)
Inventory(station='US009052008', latitude=43.7333, longitude=-96.6333,
     element='PRCP', start=2008, end=2016)
Inventory(station='US009052008', latitude=43.7333, longitude=-96.6333,
     element='SNWD', start=2009, end=2016)
Inventory(station='US10RMHS145', latitude=40.5268, longitude=-105.1113,
     element='PRCP', start=2004, end=2004)
```

Выбор станции по широте и долготе

Теперь данные загружены, и вы можете использовать широту и долготу для нахождения станций, ближайших к вашему местоположению, а затем выбрать самую длинную серию температур на основании начального и конечного года. Даже в первой строке данных встречаются два обстоятельства, заслуживающих внимания:

- Существуют различные типы элементов, но нас интересуют только элементы TMIN и TMAX для минимальной и максимальной температуры.
- Ни одна из первых записей периодов наблюдений не распространяется более чем на несколько лет. Если вы хотите взглянуть на данные в исторической перспективе, найдите намного более длинную серию температурных данных.

Чтобы быстро получить нужную информацию, мы можем при помощи генератора списка создать подсписок, содержащий только элементы периодов наблюдений

с типом TMIN или TMAX. Еще один важный момент — получение станций с длинной серией данных, поэтому при создании подсписка также следует убедиться в том, что серия начинается до 1920 года, а заканчивается после 2015 года. Таким образом, мы отбираем станции, имеющие не менее 95 лет наблюдений:

```
inventory_temps = [x for x in inventory if x.element in ['TMIN', 'TMAX']
                   and x.end >= 2015 and x.start < 1920]
inventory_temps[:5]

[Inventory(station='USC00010252', latitude=31.3072, longitude=-86.5225,
     element='TMAX', start=1912, end=2017),
 Inventory(station='USC00010252', latitude=31.3072, longitude=-86.5225,
     element='TMIN', start=1912, end=2017),
 Inventory(station='USC00010583', latitude=30.8839, longitude=-87.7853,
     element='TMAX', start=1915, end=2017),
 Inventory(station='USC00010583', latitude=30.8839, longitude=-87.7853,
     element='TMIN', start=1915, end=2017),
 Inventory(station='USC00012758', latitude=31.445, longitude=-86.9533,
     element='TMAX', start=1890, end=2017)]
```

При взгляде на первые пять записей нового списка мы видим, что ситуация улучшилась. Теперь список содержит только температурные элементы, а начальный и конечный год показывают, что список содержит длинные серии данных.

Остается проблема выбора ближайшей станции. Для этого следует сравнить широту и долготу данных станций с широтой и долготой вашего местоположения. Существуют разные способы получения широты и долготы произвольной точки, но проще всего использовать картографическое приложение или поиск в Сети. (Например, я так получила широту 41,882 и долготу –87,629.)

Так как вас интересуют станции, ближайшие к вашему местоположению, задача подразумевает сортировку списка на основании близости широты и долготы станций к широте и долготе вашего местоположения. Отсортировать список несложно, в том числе и по широте/долготе. Но как отсортировать по расстоянию от вашей текущей широты и долготы?

Для этого нужно определить ключевую функцию сортировки, которая вычисляет разность между вашей широтой и широтой станции и разность между вашей долготой и долготой станции и объединяет их в одно число. Остается упомянуть о том, что перед сложением необходимо вычислить абсолютные значения разностей, чтобы суммирование отрицательной разности с положительной не сбило с толку процедуру сортировки:

```
# Пригород Chicago по данным картографического приложения
latitude, longitude = 41.882, -87.629

inventory_temps.sort(key=lambda x:  abs(latitude-x.latitude) + abs(longitude-
x.longitude))

inventory_temps[:20]
Out[24]:
```

```
[Inventory(station='USC00110338', latitude=41.7806, longitude=-88.3092,
     element='TMAX', start=1893, end=2017),
 Inventory(station='USC00110338', latitude=41.7806, longitude=-88.3092,
     element='TMIN', start=1893, end=2017),
 Inventory(station='USC00112736', latitude=42.0628, longitude=-88.2861,
     element='TMAX', start=1897, end=2017),
 Inventory(station='USC00112736', latitude=42.0628, longitude=-88.2861,
     element='TMIN', start=1897, end=2017),
 Inventory(station='USC00476922', latitude=42.7022, longitude=-87.7861,
     element='TMAX', start=1896, end=2017),
 Inventory(station='USC00476922', latitude=42.7022, longitude=-87.7861,
     element='TMIN', start=1896, end=2017),
 Inventory(station='USC00124837', latitude=41.6117, longitude=-86.7297,
     element='TMAX', start=1897, end=2017),
 Inventory(station='USC00124837', latitude=41.6117, longitude=-86.7297,
     element='TMIN', start=1897, end=2017),
 Inventory(station='USC00119021', latitude=40.7928, longitude=-87.7556,
     element='TMAX', start=1893, end=2017),
 Inventory(station='USC00119021', latitude=40.7928, longitude=-87.7556,
     element='TMIN', start=1894, end=2017),
 Inventory(station='USC00115825', latitude=41.3708, longitude=-88.4336,
     element='TMAX', start=1912, end=2017),
 Inventory(station='USC00115825', latitude=41.3708, longitude=-88.4336,
     element='TMIN', start=1912, end=2017),
 Inventory(station='USC00115326', latitude=42.2636, longitude=-88.6078,
     element='TMAX', start=1893, end=2017),
 Inventory(station='USC00115326', latitude=42.2636, longitude=-88.6078,
     element='TMIN', start=1893, end=2017),
 Inventory(station='USC00200710', latitude=42.1244, longitude=-86.4267,
     element='TMAX', start=1893, end=2017),
 Inventory(station='USC00200710', latitude=42.1244, longitude=-86.4267,
     element='TMIN', start=1893, end=2017),
 Inventory(station='USC00114198', latitude=40.4664, longitude=-87.685,
     element='TMAX', start=1902, end=2017),
 Inventory(station='USC00114198', latitude=40.4664, longitude=-87.685,
     element='TMIN', start=1902, end=2017),
 Inventory(station='USW00014848', latitude=41.7072, longitude=-86.3164,
     element='TMAX', start=1893, end=2017),
 Inventory(station='USW00014848', latitude=41.7072, longitude=-86.3164,
     element='TMIN', start=1893, end=2017)]
```

Выбор станции и получение метаданных

При просмотре первых 20 элементов только что отсортированного списка мы видим, что первая станция USC00110338 неплохо подходит. У нее заданы оба элемента, TMIN и TMAX, а период наблюдений начинается в 1893 году и продолжается до 2017-го — данные более чем за 120 лет. Сохраним эту станцию в переменной `station` и быстро разберем уже полученные данные, чтобы получить чуть более подробную информацию о станции.

В файле readme можно найти следующую информацию о данных станций:

```
IV. FORMAT OF "ghcnd-stations.txt"

------------------------------
Variable   Columns   Type
------------------------------
ID            1-11   Character
LATITUDE     13-20   Real
LONGITUDE    22-30   Real
ELEVATION    32-37   Real
STATE        39-40   Character
NAME         42-71   Character
GSN FLAG     73-75   Character
HCN/CRN FLAG 77-79   Character
WMO ID       81-85   Character
------------------------------
Определения этих переменных:
ID          идентификационный код станции. Первые два символа содержат код страны
            по классификации FIPS, третий символ содержит код сети, определяющий
            используемую систему нумерации станций, а остальные восемь символов
            содержат собственно идентификатор станции.

            За полным списком кодов стран обращайтесь к файлу "ghcnd-countries.txt".
            За списком кодов штатов/провинций/территорий обращайтесь
            к файлу "ghcnd-states.txt".

            Код сети состоит из следующих пяти значений:

            0 = не задано (станция определяется восемью алфавитно-цифровыми
            символами).
        1 = идентификатор CoCoRaHS (Community Collaborative Rain, Hail, and Snow).
            Для согласования со стандартом GHCN Daily все числа в исходных
            идентификаторах CoCoRaHS были дополнены слева до четырех цифр. Кроме
            того, были удалены символы "-" и "_", чтобы длина идентификатора не
            превышала 11 символов с префиксом "US1". Например, идентификатор
            CoCoRaHS "AZ-MR-156" в in GHCN-Daily превращается в "US1AZMR0156".
        C = идентификатор U.S. Cooperative Network (последние шесть символов
            идентификатора GHCN
            Daily ID)/
        E = идентификатор, используемый в ECA&D.
        M = идентификатор Всемирной метеорологической организации (последние
            пять символов идентификатора GHCN-Daily ID).
        N = идентификатор, используемый в данных Национального метеорологического
            или гидрологического центра.
        R = идентификатор Межведомственных дистанционно управляемых автоматических
            метеорологических станций США (RAWS).
        S = идентификатор Службы сбережения природных ресурсов SNOTEL
            (SNOwpack TELemetry).
```

```
            W = идентификатор WBAN (последние пять символов идентификатора
            GHCN-Daily).

LATITUDE    широта станции (в градусах).
LONGITUDE   долгота станции (в градусах).
ELEVATION   возвышение станции (в метрах, если данные отсутствуют = -999.9).
STATE       почтовый код штата (только для станций в США).
NAME        название станции.
GSN FLAG    флаг, указывающий, является ли станция частью сети GSN (GCOS Surface
            Network). Значение флага задается проверкой числа из поля WMOID
            по официальному списку станций GSN. Определены два значения:
            пустое = станция не входит в GSN или код станции WMO недоступен.
            GSN    = станция GSN.

HCN/CRN     флаг, указывающий, является ли станция частью сети HCN (Historical
FLAG        Climatology Network) США. Определены три значения:
            пустое = станция не входит в сеть HCN или CRN.
            HCN    = станция входит в сеть HCN.
            CRN    = станция входит в сеть CRN (Climate Reference Network).

WMO ID      идентификатор станции во Всемирной метеорологической организации. Если
            станция не имеет кода WMO (или код еще не был назначен этой станции),
            поле остается пустым.
```

Хотя в более серьезном проекте поля метаданных представляли бы больший интерес, непосредственно сейчас мы хотим просто сравнить начальный и конечный код из данных периодов наблюдений с остальными метаданными станций в файле станций.

Перебрать содержимое файла станций и найти станцию с нужным идентификатором можно несколькими способами. Можно создать цикл `for`, который перебирает все строки и прерывается при нахождении нужной; можно разбить данные на строки, а затем отсортировать их и воспользоваться бинарным поиском, и т. д. В зависимости от природы и объема данных могут быть уместны разные подходы. В данном случае, поскольку данные уже загружены и не слишком велики, мы воспользуемся генератором списка для получения списка, единственным элементом которого является искомая станция:

```
station_id = 'USC00110338'

# Разбор данных станций
Station = namedtuple("Station", ['station_id', 'latitude', 'longitude',
'elevation', 'state', 'name', 'start', 'end'])

stations = [(x[0:11], float(x[12:20]), float(x[21:30]), float(x[31:37]),
x[38:40].strip(), x[41:71].strip())
            for x in stations_txt.split("\n") if x.startswith(station_id)]

station = Station(*stations[0] + (inventory_temps[0].start,
     inventory_temps[0].end))
print(station)
Station(station_id='USC00110338', latitude=41.7806, longitude=-88.3092,
     elevation=201.2, state='IL', name='AURORA', start=1893, end=2017)
```

На данный момент мы определили, что нас интересуют данные от метеорологической стации в Ороре (штат Иллинойс) — ближайшей станции к пригородам Чикаго, располагающей температурными данными более чем за 100 лет.

Получение и разбор погодных данных

После того как вы определитесь с выбором станции, на следующем шаге необходимо получить погодные данные для этой станции и разобрать их. Процесс очень похож на тот, который использовался в предыдущем разделе.

Получение данных

Сначала загрузите файл данных и сохраните его на случай, если он еще понадобится в будущем:

```
# Получение ежедневных данных для выбранной станции

r = requests.get('https://www1.ncdc.noaa.gov/pub/data/ghcn/daily/all/
     {}.dly'.format(station.station_id))
weather = r.text

# Сохранение данных в текстовом файле, чтобы их не пришлось загружать повторно

with open('weather_{}.txt'.format(station), "w") as weather_file:
    weather_file.write(weather)

# Чтение из сохраненного файла (используется только в том случае,
# если вы хотите начать процесс заново без загрузки файла).

with open('weather_{}.txt'.format(station)) as weather_file:
    weather = weather_file.read()

print(weather[:540])
USC00110338189301TMAX  -11  6  -44  6 -139  6  -83  6 -100  6  -83  6  -72  6
      -83  6  -33  6 -178  6 -150  6 -128  6 -172  6 -200  6 -189  6 -150  6 -
      106  6  -61  6  -94  6  -33  6  -33  6  -33  6  -33  6    6  6  -33  6
      -78  6  -33  6   44  6  -89 I6  -22  6    6  6
USC00110338189301TMIN  -50  6 -139  6 -250  6 -144  6 -178  6 -228  6 -144  6
     -222  6 -178  6 -250  6 -200  6 -206  6 -267  6 -272  6 -294  6 -294  6
     -311  6 -200  6 -233  6 -178  6 -156  6  -89  6 -200  6 -194  6 -194  6
     -178  6 -200  6  -33 I6 -156  6 -139  6 -167  6
```

Разбор метеорологических данных

Итак, данные успешно загружены, и вы видите, что они несколько сложнее данных станций и наблюдений. Очевидно, пришло время вернуться к файлу readme.txt и разделу III, в котором приведено описание файла данных с погодой. Отфильтруйте данные. Оставьте только те, которые представляют для вас интерес, и исключите остальные типы элементов, а также всю систему флагов, описывающих источник, качество и тип значений:

```
III. ФОРМАТ ФАЙЛОВ ДАННЫХ (ФАЙЛЫ ".dly")

Каждый файл ".dly" содержит данные одной станции. Имя файла соответствует
идентификационному коду станции. Например, файл "USC00026481.dly"
содержит данные для станции с идентификационным кодом USC00026481).

Каждая запись в файле содержит данные за один месяц. Переменные в каждой строке:
------------------------------
Variable   Columns   Type
------------------------------
ID            1-11   Character
YEAR         12-15   Integer
MONTH        16-17   Integer
ELEMENT      18-21   Character
VALUE1       22-26   Integer
MFLAG1       27-27   Character
QFLAG1       28-28   Character
SFLAG1       29-29   Character
VALUE2       30-34   Integer
MFLAG2       35-35   Character
QFLAG2       36-36   Character
SFLAG2       37-37   Character
  .           .          .
  .           .          .
  .           .          .
VALUE31    262-266   Integer
MFLAG31    267-267   Character
QFLAG31    268-268   Character
SFLAG31    269-269   Character
------------------------------
Переменные имеют следующие определения:

ID          идентификационный код станции. За полным списком станций и их
            метаданными обращайтесь к файлу ghcnd-stations.txt.
YEAR        год, к которому относится запись.

MONTH       месяц, к которому относится запись.

ELEMENT     тип элемента. Всего определены пять основных элементов, а также ряд
            дополнительных элементов.

      Пять основных элементов:

            PRCP  = осадки (в десятых долях миллиметра)
            SNOW  = количество снеговых осадков (мм)
            SNWD  = толщина снежного покрова (мм)
            TMAX  = максимальная температура (в десятых долях градусов C)
            TMIN  = минимальная температура (в десятых долях градусов C)
...
VALUE1 — значение для первого дня месяца (данные отсутствуют = -9999).

MFLAG1 — флаг измерения для первого дня месяца.
```

```
QFLAG1 — флаг качества для первого дня месяца.
SFLAG1 — флаг источника для первого дня месяца.

VALUE2 — значение для второго дня месяца.

MFLAG2 — флаг измерения для второго дня месяца.

QFLAG2 — флаг качества для второго дня месяца.

SFLAG2 — флаг источника для второго дня месяца.

... и т. д. до 31-го числа месяца. Обратите внимание: если месяц содержит менее
31 дня, то лишним переменным присваивается признак отсутствия данных (например,
для апреля VALUE31 = –9999, MFLAG31 = пусто, QFLAG31 = пусто, SFLAG31 = пусто).
```

Сейчас нас интересует прежде всего то, что идентификатор станции занимает 11 символов строки, год — следующие 4 символа, месяц — следующие 2 символа, а элемент — следующие 4 символа после него. Далее следует 31 позиция для ежедневных данных; каждая позиция состоит из 5 символов для температуры, выраженной в десятых долях градусов Цельсия, и 3 символов флагов. Как упоминалось ранее, на флаги в этом упражнении можно не обращать внимания. Также видно, что отсутствующие значения температуры обозначаются –9999, если этого дня нет в месяце, поэтому, например, для типичного февраля 29-е, 30-е и 31-е значения будут равны –9999.

При обработке данных в этом упражнении мы стремимся определить общие тенденции, поэтому беспокоиться о каждом отдельном дне необязательно. Вместо этого стоит вычислить средние значения за месяц. Можно сохранить максимальное, минимальное и среднее значения за целый месяц и использовать эти значения.

Это означает, что обработка каждой строки погодных данных должна происходить следующим образом:

- Разбить строку на отдельные поля, отбрасывая (или игнорируя) флаги для всех ежедневных значений.
- Удалить данные со значением –9999, преобразовать год и месяц в целые числа, а значения температуры — в числа с плавающей точкой. Помните, что значения температуры выражены в десятых долях градусов Цельсия.
- Вычислить среднее значение, отобрать наибольшее и наименьшее значение.

Здесь можно выбрать один из двух путей. Во-первых, вы можете сделать несколько проходов по данным, разбить их на поля, отбросить значения-заменители, преобразовать строки в числа и, наконец, вычислить сводные значения. Во-вторых, можно написать функцию, которая выполняет все эти операции подряд, и сделать все за один проход. Оба решения будут действительными. В данном случае мы выберем второй подход и создадим функцию `parse_line` для выполнения всех преобразований данных:

```
def parse_line(line):
    """ Разбирает строку метеорологических данных
        Удаляет значения -9999 (данные отсутствуют)
    """

    # Вернуть None, если строка пуста
    if not line:
        return None

    # Выделить первые 4 поля и строку со значениями температур
    record, temperature_string = (line[:11], int(line[11:15]),
     int(line[15:17]), line[17:21]), line[21:]

    # Выдать исключение, если строка температур слишком коротка
    if len(temperature_string) < 248:
        raise ValueError("String not long enough - {}
     {}".format(temperature_string, str(line)))

    # Использовать генератор списка для извлечения и преобразования
    values = [float(temperature_string[i:i + 5])/10 for i in range(0, 248, 8)
              if not temperature_string[i:i + 5].startswith("-9999")]

    # Получить количество значений, минимум и максимум, вычислить среднее
    count = len(values)
    tmax = round(max(values), 1)
    tmin = round(min(values), 1)
    mean = round(sum(values)/count, 1)

    # Добавить сводные значения температур к полям,
    # извлеченным ранее, и вернуть
    return record + (tmax, tmin, mean, count)
```

Если протестировать функцию с первой строкой необработанных метеорологических данных, вы получите следующий результат:

```
parse_line(weather[:270])
Out[115]:
('USC00110338', 1893, 1, 'TMAX', 4.4, -20.0, -7.8, 31)
```

Похоже, эта функция успешно справляется с разбором данных. А раз функция работает, вы можете разобрать данные и либо сохранить их, либо продолжить обработку:

```
# Обработка всех метеорологических данных

# Генератор списка, пустые строки не разбираются
weather_data = [parse_line(x) for x in weather.split("\n") if x]

len(weather_data)

weather_data[:10]

[('USC00110338', 1893, 1, 'TMAX', 4.4, -20.0, -7.8, 31),
 ('USC00110338', 1893, 1, 'TMIN', -3.3, -31.1, -19.2, 31),
```

```
 ('USC00110338', 1893, 1, 'PRCP', 8.9, 0.0, 1.1, 31),
 ('USC00110338', 1893, 1, 'SNOW', 10.2, 0.0, 1.0, 31),
 ('USC00110338', 1893, 1, 'WT16', 0.1, 0.1, 0.1, 2),
 ('USC00110338', 1893, 1, 'WT18', 0.1, 0.1, 0.1, 11),
 ('USC00110338', 1893, 2, 'TMAX', 5.6, -17.2, -0.9, 27),
 ('USC00110338', 1893, 2, 'TMIN', 0.6, -26.1, -11.7, 27),
 ('USC00110338', 1893, 2, 'PRCP', 15.0, 0.0, 2.0, 28),
 ('USC00110338', 1893, 2, 'SNOW', 12.7, 0.0, 0.6, 28)]
```

Итак, теперь все метеорологические данные (не только данные температуры) разобраны и занесены в список.

Сохранение информации в базе данных (необязательно)

На данный момент все метеорологические записи (а при желании также записи станций и периодов наблюдения) можно сохранить в базе данных. Это позволит вам вернуться к ним позднее и использовать те же данные без хлопот, связанных с повторной загрузкой и разбором.

Например, следующий код показывает, как сохранить погодные данные в базе данных sqlite3:

```
import sqlite3

conn = sqlite3.connect("weather_data.db")
cursor = conn.cursor()

# Создание таблицы weather

create_weather = """CREATE TABLE "weather" (
    "id" text NOT NULL,
    "year" integer NOT NULL,
    "month" integer NOT NULL,
    "element" text NOT NULL,
    "max" real,
    "min" real,
    "mean" real,
    "count" integer)"""
cursor.execute(create_weather)
conn.commit()

# Сохранение разобранных данных в базе

for record in weather_data:
    cursor.execute("""insert into weather (id, year, month, element, max,
min, mean, count) values (?,?,?,?,?,?,?,?) """,
                       record)

conn.commit()
```

Когда данные будут сохранены, их можно загрузить из базы данных. Следующий код извлекает только записи TMAX:

```
cursor.execute("""select * from weather where element='TMAX' order by year,
     month""")
tmax_data = cursor.fetchall()
tmax_data[:5]

[('USC00110338', 1893, 1, 'TMAX', 4.4, -20.0, -7.8, 31),
 ('USC00110338', 1893, 2, 'TMAX', 5.6, -17.2, -0.9, 27),
 ('USC00110338', 1893, 3, 'TMAX', 20.6, -7.2, 5.6, 30),
 ('USC00110338', 1893, 4, 'TMAX', 28.9, 3.3, 13.5, 30),
 ('USC00110338', 1893, 5, 'TMAX', 30.6, 7.2, 19.2, 31)]
```

Выбор и графическое представление данных

Так как в постановке задачи речь идет только о температуре, необходимо выбрать из массива данных только температурные записи. Это достаточно легко делается при помощи пары генераторов списков: один получает список TMAX, а другой получает список TMIN. Также можно воспользоваться функциональностью pandas, которая будет использоваться для графического представления данных и исключения ненужных записей. Так как нас сейчас интересует использование «чистого» кода Python, а не pandas, выберем первый вариант.

```
tmax_data = [x for x in weather_data if x[3] == 'TMAX']
tmin_data = [x for x in weather_data if x[3] == 'TMIN']
tmin_data[:5]

[('USC00110338', 1893, 1, 'TMIN', -3.3, -31.1, -19.2, 31),
 ('USC00110338', 1893, 2, 'TMIN', 0.6, -26.1, -11.7, 27),
 ('USC00110338', 1893, 3, 'TMIN', 3.3, -13.3, -4.6, 31),
 ('USC00110338', 1893, 4, 'TMIN', 12.2, -5.6, 2.2, 30),
 ('USC00110338', 1893, 5, 'TMIN', 14.4, -0.6, 5.7, 31)]
```

Использование pandas для графического представления данных

Итак, данные очищены и готовы к нанесению на график. Чтобы упростить графическое представление данных, мы воспользуемся библиотеками pandas и matplotlib (см. главу 24). Для этого необходим работающий сервер Jupyter и установленные библиотеки pandas и matplotlib. Чтобы проверить, что они установлены, прямо из Jupyter Notebook введите следующую команду:

```
# Установка pandas и matplotlib с использованием pip
! pip3.6 install pandas matplotlib

import pandas as pd
%matplotlib inline
```

После установки pandas и matplotlib можно загрузить pandas и создать кадры данных для записей TMAX и TMIN:

```
tmax_df = pd.DataFrame(tmax_data, columns=['Station', 'Year', 'Month',
     'Element', 'Max', 'Min', 'Mean', 'Days'])
```

```
tmin_df = pd.DataFrame(tmin_data, columns=['Station', 'Year', 'Month',
     'Element', 'Max', 'Min', 'Mean', 'Days'])
```

Конечно, на графике можно изобразить все ежемесячные значения, но 123 года из 12 месяцев дают почти 1500 точек данных, а сезонные колебания температур также усложняют выявление закономерностей.

Пожалуй, вместо этого будет разумнее усреднить максимальное, минимальное и среднее ежемесячное значение, вычислить годовые значения и нанести их на график. Конечно, это можно сделать в Python, но поскольку данные уже загружены в кадре данных pandas, мы можем сгруппировать данные по годам и вычислить средние значения:

```
# Выбрать столбцы Year, Min, Max, Mean с группировкой по году,
# вычислить среднее и построить график

tmin_df[['Year','Min', 'Mean', 'Max']].groupby('Year').mean().plot(
kind='line', figsize=(16, 4))
```

Несмотря на относительно высокие колебания, результат показывает, что минимальная температура за последние 20 лет растет.

Если вы хотите построить тот же график без использования Jupyter Notebook и matplotlib, вы также можете использовать pandas, но записать данные в файл CSV или Microsoft Excel методом `to_csv` или `to_excel` кадра данных. Затем полученный файл загружается в электронную таблицу, которая используется для построения графика.

Приложение А
Документация Python

Самый лучший и актуальный источник справочной информации о Python — документация, входящая в поставку Python. С учетом этого обстоятельства будет полезно исследовать возможности работы с документацией, выходящие за рамки простой распечатки документации.

Стандартный пакет документации состоит из нескольких разделов, включая инструкции по документированию, распространению, установке и расширению Python на разных платформах. С него и следует начинать поиски ответов на любые вопросы по поводу Python. Скорее всего, двумя самыми полезными областями документации Python будут *Справочное руководство по библиотеке* (Library Reference) и *Справочное руководство по языку программирования* (Language Reference). *Справочное руководство по библиотеке* абсолютно необходимо, потому что в нем описаны встроенные типы данных и все модули, включенные в Python. *Справочное руководство по языку программирования* объясняет, как работает ядро Python. В нем представлена официальная точка зрения на ядро языка, описания типов данных, команд и т. д. Также заслуживает внимания раздел «What's New» — особенно при выпуске новой версии Python, потому что в нем приводится сводка изменений в новой версии.

A.1. Обращение к документации Python в интернете

Многие люди считают, что самый удобный способ работы с документацией Python — посещение сайта www.python.org и просмотр документации на сайте. Хотя для этого требуется подключение к интернету, у него есть несомненное преимущество: информация всегда самая актуальная. Учитывая, что для многих проектов часто бывает полезно провести поиск документации и информации в интернете, постоянно открытая в браузере вкладка с документацией Python позволит вам постоянно держать под рукой всю справочную информацию Python.

A.1.1. Просмотр документации Python на компьютере

Во многие комплекты поставки Python по умолчанию включается полная документация. Во многих дистрибутивах Linux документация находится в отдельном

пакете, который приходится устанавливать отдельно. Тем не менее во многих случаях полная документация уже находится на компьютере и легкодоступна.

Обращение к справке в интерактивной оболочке или из командной строки

В главе 2 было показано, как использовать команду `help` в интерактивном интерпретаторе для обращения к электронной справке по любому модулю или объекту Python:

```
>>> help(int)
Справка по объекту int:

class int(object)
 |  int(x[, base]) -> integer
 |
 |  Преобразует строку или число в целое число, если это возможно. Аргумент
 |  с плавающей точкой округляется в меньшую сторону (это не относится
 |  к строковому представлению числа с плавающей точкой!) При преобразовании
 |  строки может использоваться необязательный аргумент base. Передача
 |  аргумента base при преобразовании не строкового значения
 |  является ошибкой.
 |
 |  Методы, определенные в объекте:
... (список методов для int)
```

Здесь интерпретатор вызывает модуль `pydoc` для генерирования документации. Также можно воспользоваться модулем `pydoc` для поиска документации Python из командной строки. В системе Linux или macOS вы получите тот же вывод, что и в окне терминала; для этого достаточно ввести `pydoc int` в приглашении (для выхода введите `q`). В командной строке Windows, если только вы не настроите путь поиска так, чтобы он включал каталог Python Lib, вам придется вводить полный путь вида C:\Users\<пользователь>\AppData\Local\Programs\Python\Python36\Lib\pydoc.py int, где <пользователь> — ваше имя пользователя Windows.

Генерирование справки в формате HTML с использованием pydoc

Если вы предпочитаете более элегантную документацию, генерируемую `pydoc` для объекта или модуля Python, вывод можно записать в файл в формате HTML, который может быть просмотрен в любом браузере. Для этого добавьте в команду `pydoc` ключ `-w`, таким образом, в Windows команда принимает вид `C:\Users\<пользователь>\AppData\Local\Programs\Python\Python36\Lib\pydoc.py -w int`. В данном случае при поиске документации по объекту `int` `pydoc` создает файл с именем int.html в текущем каталоге, вы можете открыть и просмотреть его в браузере. На рис. А.1 показано, как файл int.html выглядит в браузере.

Если по какой-то причине требуется сгенерировать небольшой объем документации, этот способ отлично работает. Но в большинстве случаев, пожалуй, лучше использовать `pydoc` для генерирования более полной документации, как показано в следующем разделе.

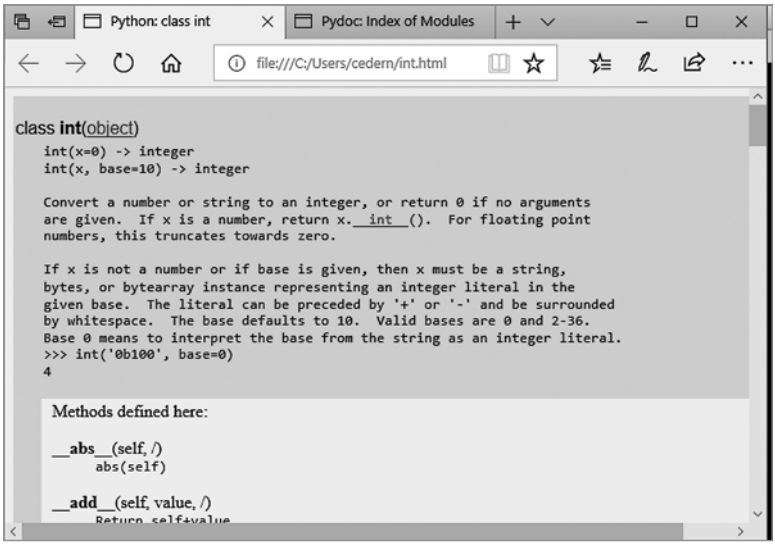


Рис. А.1. Файл int.html, сгенерированный pydoc

Использование pydoc в качестве сервера документации

Кроме возможности генерирования документации в текстовом формате и в формате HTML по любому объекту Python, модуль `pydoc` может использоваться как сервер для веб-документации. Если запустить `pydoc` с ключом `-p` и номером порта, команда откроет сервер на указанном порте. Далее команда `"b"` открывает браузер и обращается к документации по всем доступным модулям (рис. А.2).

Одно из преимуществ использования `pydoc` для предоставления документации заключается в том, что `pydoc` также сканирует текущий каталог и извлекает информацию из строк документации всех найденных модулей, даже если они не являются частью стандартной библиотеки. Однако есть один недостаток: чтобы извлечь документацию из модуля, модуль `pydoc` должен импортировать его; это означает, что он выполнит любой код на верхнем уровне модуля. Таким образом, будут выполнены сценарии, не написанные с расчетом на импортирование без побочных эффектов (см. главу 11), поэтому данную возможность следует применять с осторожностью.

Использование файла справки в Windows

В системах Windows стандартный пакет Python 3 включает полную документацию Python в виде справочных файлов Windows. Эти файлы находятся в папке `Doc` внутри папки, в которой был установлен Python. При открытии главного файла откроется окно, показанное на рис. А.3.

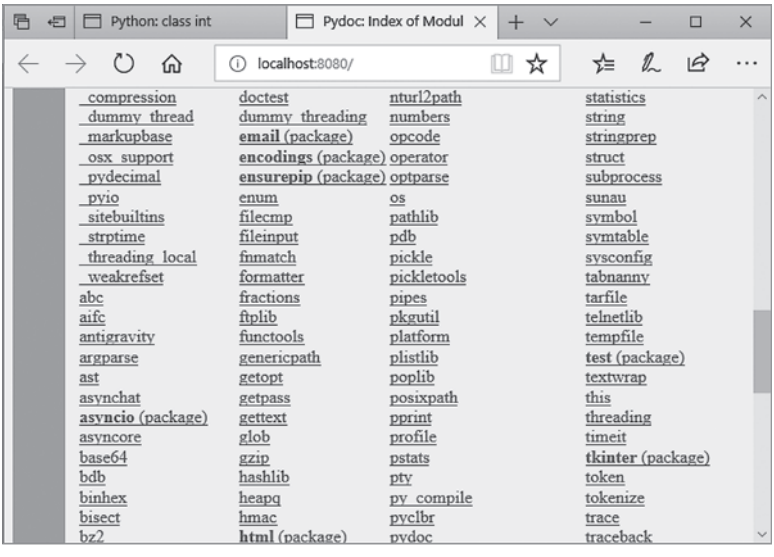


Рис. А.2. Часть документации по модулям, предоставляемой pydoc

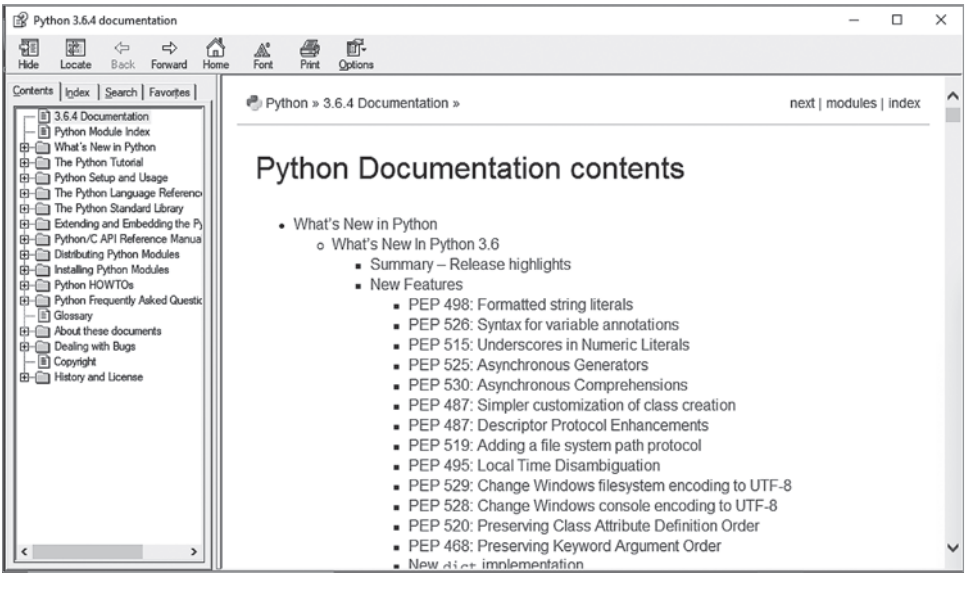


Рис. А.3. Если вы привыкли работать со справочными файлами Windows, возможно, никакая другая документация вам не понадобится

А.1.2. Загрузка документации

Если вы хотите иметь документацию Python на своем компьютере без обязательного запуска Python, загрузите полную документацию на сайте www.python.org в формате PDF, HTML или в текстовом формате, который может быть удобен для работы с документацией на ридере или другом аналогичном устройстве.

А.2. Как стать питонистом

Вокруг каждого языка программирования развиваются свои традиции и культура. Python является ярким тому примером. Многие опытные программисты Python (*питонисты*, как они себя называют) стараются писать код так, чтобы он соответствовал стилю и негласным правилам Python. Такой тип кода обычно называется *питоническим* и высоко ценится — в отличие от кода Python, похожего на код Java, C или JavaScript.

Вопрос, с которым сталкиваются все новые разработчики Python: как научиться писать питонический код? И хотя чтобы по-настоящему прочувствовать язык и его стиль, потребуется некоторое время и усилия, в оставшейся части приложения приводятся рекомендации относительно того, с чего стоит начать.

А.2.1. Десять советов для желающих стать питонистами

Советы в этом разделе нередко встречаются на учебных курсах Python среднего уровня. Я считаю, что они помогают поднять квалификацию Python на должный уровень. Не стану утверждать, что со мной все полностью согласны, но по собственному многолетнему опыту работы убеждена, что они выведут вас на правильный путь к тому, чтобы стать настоящим питонистом.

- *Осмыслите Дзен Python*. Дзен Python, или PEP 20, обобщает философию проектирования Python как языка программирования и часто упоминается в обсуждениях того, что делает сценарии более питоническими. В частности, вы должны руководствоваться принципами «Красивое лучше, чем уродливое» и «Простое лучше, чем сложное» в своей работе. Текст *Дзен Python* приводится в конце этого приложения; чтобы просмотреть его, вы всегда можете ввести команду `import this` в приглашении оболочки Python.
- *Следуйте рекомендациям PEP 8*. PEP 8 — официальное руководство по стилю Python, которое также приводится позднее в этом приложении. PEP 8 содержит разнообразные полезные советы: от форматирования кода и выбора имен переменных до применения языка. Если вы хотите писать питонический код — хорошенько изучите PEP 8.
- *Освойте работу с документацией*. В Python реализована богатая, хорошо сопровождаемая система документации, и вы должны почаще обращаться к ней.

Пожалуй, самой полезной является документация стандартной библиотеки, но учебники и файлы how-to тоже содержат много полезнейшей информации об эффективном использовании языка.

- *Пишите как можно меньше кода.* Хотя этот совет применим ко многим языкам, для Python он особенно уместен. Я имею в виду, что вы должны стараться делать свои программы настолько простыми и короткими, насколько это возможно (но не проще и короче), и этот стиль программирования следует применять как можно чаще.
- *Читайте как можно больше кода.* С самого начала сообщество Python понимало, что читать код важнее, чем писать его. Читайте как можно больше кода и по возможности обсуждайте прочитанный код с другими.
- *В первую очередь используйте встроенные структуры данных.* Прежде чем писать собственные классы для хранения данных, обратитесь к встроенным структурам Python. Различные типы данных Python можно комбинировать друг с другом практически с безграничной гибкостью, а ведь за ними стоят годы отладки и оптимизации. Пользуйтесь ими.
- *Разберитесь в генераторах.* Начинающие программисты Python почти всегда упускают из виду, какую роль в питоническом программировании занимают генераторы списков и словарей, а также выражения-генераторы. Найдите примеры их использования в коде Python, который вы читаете, и потренируйтесь в работе с ними. Вы не станете питонистом, пока не сможете написать генератор списков практически не задумываясь.
- *Используйте стандартную библиотеку.* Если вы не нашли того, что нужно, среди встроенных типов, обратитесь к стандартной библиотеке. Элементы стандартной библиотеки — это и есть знаменитые «батарейки в комплекте» языка Python. Они прошли проверку временем и были оптимизированы и документированы лучше, чем практически любой другой код Python. Старайтесь использовать их, если это возможно.
- *Пишите как можно меньше классов.* Пишите собственные классы только в том случае, если без этого не обойтись. Опытные питонисты обычно очень расчетливо подходят к написанию классов. Они знают, что проектирование хорошего класса — нетривиальная задача, а все созданные ими классы придется тестировать и отлаживать.
- *Будьте осмотрительны с фреймворками.* Фреймворки могут выглядеть привлекательно (особенно для программистов, только осваивающих язык), потому что они предоставляют много эффективных «обходных путей». Конечно, вы должны использовать фреймворки, когда это полезно, но не забывайте об их оборотных сторонах: может оказаться, что на изучение странностей «непитонического» фреймворка у вас уйдет больше времени, чем на изучение самого языка Python, или же вам придется приспосабливать свою работу к фреймворку, хотя должно быть наоборот.

А.3. PEP 8 — руководство по стилю программирования Python

В этом разделе приведена слегка отредактированная выдержка из PEP (Python Enhancement Proposal) 8. Документ PEP 8, написанный Гвидо ван Россумом и Барри Уорсо (Barry Warsaw), — ближайший аналог руководства по стилю, существующий в Python. Некоторые более конкретные разделы были опущены, но все основные моменты остались. Стремитесь к тому, чтобы ваш код соответствовал PEP 8, насколько это возможно. Ваш стиль программирования Python от этого только выиграет.

Чтобы получить доступ к полному тексту PEP 8 и ко всем остальным PEP в истории Python, перейдите в раздел документации www.python.org и найдите список PEP. Документы PEP — превосходный источник исторической и культурной информации о Python. Кроме того, вы найдете в них объяснение существующих проблем и планы на будущее.

А.3.1. Введение

В этом документе приведены соглашения по оформлению кода Python, входящего в стандартную библиотеку из основного дистрибутива Python. За описанием руководства по стилю для кода C в реализации Python на C обращайтесь к прилагаемому информационному документу PEP[1]. Этот документ был создан по материалам исходного очерка Гвидо «Руководство по стилю Python» с некоторыми добавлениями из руководства по стилю, написанному Барри[2]. В случае возникновения конфликтов для целей этого PEP предпочтение отдается стилю Гвидо. Этот документ PEP может быть незавершенным (на самом деле он никогда не будет завершен).

Бездумная последовательность — удел слабых умов

Одно из ключевых наблюдений Гвидо заключается в том, что мы читаем код намного чаще, чем пишем его. Рекомендации, представленные здесь, призваны улучшить удобочитаемость кода и сделать его последовательным во всем спектре кода Python. Как сказано в PEP 20[3], «читаемость имеет значение».

Вся суть руководства по стилю — последовательное оформление кода. Последовательность в руководстве по стилю очень важна. Последовательность в проектах еще важнее. Последовательность в пределах одного модуля или функции важнее всего.

Но самое важное — знать, когда можно быть непоследовательным. Есть ситуации, в которых руководство по стилю попросту неприменимо. Если вы сомневаетесь, руководствуйтесь здравым смыслом. Рассмотрите другие примеры и решите, что выглядит лучше. И не стесняйтесь спрашивать!

---

[1] PEP 7, Style Guide for C Code, van Rossum, https://www.python.org/dev/peps/pep-0007/.

[2] Руководство по стилю GNU Mailman, http://barry.warsaw.us/software/STYLEGUIDE.txt. В настоящее время URL-адрес пуст, хотя он и представлен в руководстве по стилю PEP 8.

[3] PEP 20, The Zen of Python, www.python.org/dev/peps/pep-0020/.

Для нарушения конкретных правил есть две веские причины:

- Если применение правила затруднит восприятие кода даже для тех, кто привык читать код, соблюдающий правила.
- Ради соблюдения целостности с окружающим кодом, в котором эти правила тоже нарушаются (возможно, по историческим причинам). Впрочем, в такой ситуации стоит воспользоваться возможностью и убрать чужой мусор (в стиле экстремального программирования).

А.3.2. Структура кода

Отступы

Используйте четыре пробела на уровень отступа.

В очень старом коде, который вам не хотелось бы испортить, можно продолжать использовать табуляции из восьми пробелов.

Табуляции или пробелы?

Никогда не смешивайте табуляции с пробелами.

Самый популярный способ создания отступов в Python — использование только пробелов. Второй по популярности способ — использование только табуляций. Код, в котором отступы создаются смесью табуляций и пробелов, должен быть преобразован, чтобы в нем использовались только пробелы. При запуске интерпретатора командной строки Python с ключом `-t` интерпретатор выдает предупреждения о коде, в котором табуляции смешиваются с пробелами. С ключом `-tt` предупреждения превращаются в ошибки. Настоятельно рекомендуется использовать эти ключи!

В новых проектах пробелам следует отдавать предпочтение перед табуляциями. В большинстве редакторов имеются средства, с которыми эта задача решается просто.

Максимальная длина строки

Максимальная длина строки не должна превышать 79 символов.

Все еще существует много устройств, ограниченных 80-символьными строками. Кроме того, ограничение окон 80 символами позволяет расположить несколько окон рядом друг с другом. Стандартный режим переноса на таких устройствах искажает визуальную структуру кода и усложняет его понимание. По этой причине все строки следует ограничивать 79 символами. Для длинных блоков текста с переносами (строк документации или комментариев) рекомендуется устанавливать ограничение в 72 символа.

Для переноса длинных строк рекомендуется использовать механизм неявного продолжения строк в круглых, квадратных и фигурных скобках. При необходимости можно заключить выражение в лишнюю пару круглых скобок, но иногда перенос

с обратной косой чертой выглядит лучше. Проследите за тем, чтобы продолжение строки было снабжено соответствующим отступом. Бинарные операторы следует разбивать *после* оператора, а не до него. Несколько примеров:

```
class Rectangle(Blob):
    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if width == 0 and height == 0 and \
           color == 'red' and emphasis == 'strong' or \
           highlight > 100:
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Пустые строки

Высокоуровневые определения функций и определения классов отделяются двумя пустыми строками. Определения методов внутри класса отделяются одной пустой строкой.

Дополнительные пустые строки можно использовать (осмотрительно) для разбиения групп взаимосвязанных функций. Пустые строки между несколькими взаимосвязанными однострочными конструкциями (например, набором фиктивных реализаций) можно опустить.

Используйте пустые строки в функциях (умеренно) для обозначения логических разделов.

Python воспринимает символ подачи страницы Control+L (^L) как пустую область (whitespace). Многие программы воспринимают эти символы как разделители страниц, поэтому вы можете использовать их для разделения страниц в разделах вашего файла.

Импортирование

Команды импортирования обычно размещаются в отдельных строках, например:

```
import os
import sys
```

Не объединяйте их в одну строку:

```
import sys, os
```

Тем не менее следующая конструкция допустима:

```
from subprocess import Popen, PIPE
```

Команды импортирования всегда размещаются в начале файла, после комментариев модулей и строк документации, но перед константами и глобальными переменными модулей.

Команды импортирования группируются в следующем порядке:

1. Импортирование из стандартной библиотеки.
2. Импортирование из сопутствующих сторонних модулей.
3. Локальное импортирование, относящееся к приложению/библиотеке.

Группы команд импортирования разделяются пустыми строками.

После команд импортирования размещаются все необходимые спецификации `__all__`.

Использовать относительное импортирование для внутрипакетного импортирования в высшей степени нежелательно. При импортировании всегда используйте абсолютные пути к пакетам. Даже сейчас, после полной реализации PEP 328[1] в Python 2.5, использовать этот стиль явного относительного импортирования настоятельно не рекомендуется. Абсолютные пути лучше портируются и обычно лучше читаются.

При импортировании класса из модуля, содержащего класс, обычно можно использовать запись

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

Если при этом возникают локальные конфликты имен, запишите команды импортирования в виде

```
import myclass
import foo.bar.yourclass
и используйте запись myclass.MyClass и foo.bar.yourclass.YourClass.
```

Пропуски в выражениях и командах

Больная тема — избегайте лишних пропусков в следующих ситуациях:

- Непосредственно внутри круглых, квадратных или фигурных скобок:

  Да:

  ```
  spam(ham[1], {eggs: 2})
  ```

  Нет:

  ```
  spam( ham[ 1 ], { eggs: 2 } )
  ```

- Непосредственно перед запятой, точкой с запятой или двоеточием:

  Да:

  ```
  if x == 4: print x, y; x, y = y, x
  ```

[1] PEP 328, Imports: Multi-Line and Absolute/Relative, www.python.org/dev/peps/pep-0328/.

Нет:

```
if x == 4 : print x , y ; x , y = y , x
```

- Непосредственно перед круглой скобкой, открывающей список аргументов при вызове функции:

  Да:

  ```
  spam(1)
  ```

  Нет:

  ```
  spam (1)
  ```

- Непосредственно перед открывающей квадратной скобкой при индексировании или создании сегмента:

  Да:

  ```
  dict['key'] = list[index]
  ```

  Нет:

  ```
  dict ['key'] = list [index]
  ```

- Более одного пробела вокруг оператора присваивания (или другого оператора) для выравнивания его с другим оператором:

  Да:

  ```
  x = 1
  y = 2
  long_variable = 3
  ```

  Нет:

  ```
  x             = 1
  y             = 2
  long_variable = 3
  ```

Другие рекомендации

Всегда дополняйте следующие бинарные операторы одним пробелом с каждой стороны: присваивание (`=`), расширенное присваивание (`+=`, `-=` и т. д.), сравнения (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`) и логические операторы (`and`, `or`, `not`).

Добавляйте пробелы вокруг арифметических операторов.

Да:

```
i = i + 1
submitted += 1
```

```
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Нет:

```
i=i+1
submitted +=1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

Не ставьте пробелы вокруг знака = при обозначении аргумента с ключевым словом или значения параметра по умолчанию.

Да:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

Нет:

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

Составные команды (несколько команд в одной строке) использовать обычно не рекомендуется.

Да:

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Лучше не надо:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

Хотя иногда можно разместить команду `if/for/while` с маленьким телом в той же строке, никогда не делайте этого для команд с несколькими секциями. И не пытайтесь сворачивать такие длинные строки!

Лучше не надо:

```
if foo == 'blah': do_blah_thing()
for x in lst: total += x
   while t < 10: t = delay()
```

Определенно нет:

```
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()
try: something()
finally: cleanup()
do_one(); do_two(); do_three(long, argument,
                             list, like, this)
if foo == 'blah': one(); two(); three()
```

А.4. Комментарии

Комментарии, противоречащие коду, — хуже, чем отсутствие комментариев. В первую очередь всегда обновляйте комментарии при изменении кода!

Комментарии должны быть законченными предложениями. Если комментарий представляет собой предложение, его первое слово должно быть записано с прописной буквы, если только оно не является идентификатором, начинающимся со строчной буквы (никогда не изменяйте регистр идентификаторов!).

Если комментарий короткий, точку в конце можно опустить. Блочные комментарии обычно состоят из одного или нескольких абзацев, построенных из законченных предложений, при этом каждое предложение должно завершаться точкой.

Используйте два пробела после точки, завершающей предложение.

В английском тексте действуют правила Странка и Уайта[1].

Программистам Python из неанглоязычных стран: пожалуйста, пишите свои комментарии на английском, если только вы не уверены на 120 % в том, что код никогда не будет прочитан людьми, не говорящими на вашем языке.

Блочные комментарии

Блочные комментарии обычно применяются к части кода (или всему коду), следующему за ними, и снабжаются отступом того же уровня, что и код. Каждая строка блочного комментария начинается с # и одного пробела (за исключением текста с отступом внутри комментария).

Абзацы внутри блочного комментария разделяются строкой, содержащей один символ #.

Встроенные комментарии

Будьте осмотрительны при использовании встроенных комментариев.

Встроенный комментарий записывается в одной строке с командой. Встроенные комментарии должны отделяться от команды как минимум двумя пробелами. Они должны начинаться с символа # и одного пробела.

Обычно без встроенных комментариев можно обойтись. Если встроенный комментарий утверждает очевидное, он только отвлекает. Не поступайте так:

```
x = x + 1                 # Увеличить x
```

Впрочем, иногда они бывают полезны:

```
x = x + 1                 # Компенсация границы
```

---

[1] Имеется в виду книга «Элементы стиля» проф. Уильяма Странка-мл., вышедшая в 1920 году в США. В книге приводятся распространенные ошибки и правила, как их избегать. https://en.wikipedia.org/wiki/The_Elements_of_Style. — *Примеч. пер.*

Строки документации

Правила написания хороших строк документации увековечены в PEP 257[1].

Пишите строки документации для всех общедоступных модулей, функций, классов и методов. Строки документации не являются необходимыми для методов, не предназначенных для общего доступа, но в такой метод следует включить комментарий с описанием того, что делает метод. Этот комментарий должен следовать после строки `def`.

В PEP 257 описаны правила написания хороших строк документации. Самое важное из них гласит, что последовательность `"""`, завершающая многострочный текст, должна располагаться в отдельной строке; при этом желательно, чтобы ей предшествовала пустая строка, например:

```
"""Return a foobang
Optional plotz says to frobnicate the bizbaz first.

"""
```

Для однострочных строк документации закрывающая последовательность `"""` может располагаться в той же строке.

Контроль версий

Если вам приходится включать в исходный файл служебный код Subversion, CVS или RCS, это следует делать так:

```
__version__ = "$Revision: 68852 $"     # $Source$
```

Эти строки включаются после строки документации модуля, перед любым другим кодом, и отделяются пустыми строками сверху и снизу.

А.4.1. Выбор имен

В схемах назначения имен в библиотеке Python существует некоторый беспорядок, поэтому добиться идеальной целостности не удастся. Тем не менее ниже приведены стандарты имен, которые рекомендуется применять на настоящий момент. Новые модули и пакеты (включая сторонние фреймворки) должны писаться с учетом этих стандартов, но если в существующей библиотеке применяется другой стиль, предпочтение следует отдавать внутреннему соответствию.

Стили выбора имен

Существует много разных стилей выбора имен. Их полезно знать независимо от того, для чего они используются.

---

[1] PEP 257, Docstring Conventions, Goodger, van Rossum, www.python.org/dev/peps/pep-0257/.

Обычно различаются следующие стили выбора имен:

- b (одна буква нижнего регистра).
- B (одна буква верхнего регистра).
- lowercase (нижний регистр).
- lower_case_with_underscores (нижний регистр с подчеркиваниями).
- UPPERCASE (верхний регистр).
- UPPER_CASE_WITH_UNDERSCORES (верхний регистр с подчеркиваниями).
- CapitalizedWords (слова, начинающиеся с буквы верхнего регистра) — также называется «верблюжьим регистром» из-за выступающих «горбов»[1]. Примечание: при использовании сокращений в таких именах все буквы сокращения записываются в верхнем регистре: HTTPServerError лучше, чем HttpServerError.
- mixedCase (смешанный регистр) — отличается от предыдущего варианта первой буквой нижнего регистра.
- Capitalized_Words_With_Underscores (слова, начинающиеся с буквы верхнего регистра, с подчеркиваниями — уродство!).

Также существует стиль, использующий короткий уникальный префикс для группировки взаимосвязанных имен. В Python он встречается редко, но я упоминаю его, поскольку он существует. Например, функция `os.stat()` возвращает кортеж, элементам которого традиционно присваиваются имена вида `st_mode`, `st_size`, `st_mtime` и т. д. (Это делается, чтобы подчеркнуть соответствие с полями структуры системной функции POSIX и помочь программистам быстрее привыкнуть к ним.)

Библиотека X11 использует префикс X для всех своих общедоступных функций. В Python этот стиль обычно считается избыточным, потому что имена атрибутов и методов снабжаются префиксом объекта, а имена функций снабжаются префиксом имени модуля.

Кроме того, различаются следующие специальные формы с использованием начальных и конечных подчеркиваний (обычно могут объединяться с любыми схемами регистра символов):

- _single_leading_underscore (один символ подчеркивания в начале) — слабый индикатор «для внутреннего использования». Например, команда `from M import *` не импортирует объекты, имена которых начинаются с символа подчеркивания.
- single_trailing_underscore_ (один символ подчеркивания в конце) — используется для предотвращения конфликтов с ключевыми словами Python, например:
- `tkinter.Toplevel(master, class_='ClassName').`
- __double_leading_underscore (два символа подчеркивания в начале) — для атрибутов классов активизирует преобразование имени (внутри класса `FooBar __boo` превращается в `_FooBar__boo`; см. ниже).

[1] За дополнительной информацией обращайтесь по адресу https://ru.wikipedia.org/wiki/CamelCase

- __double_leading_and_trailing_underscore__ (два символа подчеркивания в начале и в конце) — специальные («волшебные») объекты или атрибуты в пространствах имен под контролем пользователя, например `__init__`, `__import__` или `__file__`. Никогда не пытайтесь подбирать такие имена; используйте только документированные возможности.

Рекомендации по выбору имен

- Нежелательные имена.

  Никогда не используйте символы *l* (буква нижнего регистра), *O* (буква верхнего регистра) или *I* (буква верхнего регистра) в качестве односимвольных имен переменных.

  В некоторых шрифтах эти символы неотличимы от цифр 1 (один) и 0 (ноль). Если вам нужно использовать *l*, используйте *L*.

- Имена пакетов и модулей.

  Модули должны иметь короткие имена, записанные только символами нижнего регистра. В именах модулей могут использоваться подчеркивания, если это упрощает их чтение. Пакеты Python должны иметь короткие имена, записанные в нижнем регистре, хотя включать в них символы подчеркивания не рекомендуется.

  Так как имена модулей соответствуют именам файлов, а некоторые файловые системы игнорируют регистр символов и усекают длинные имена, важно, чтобы имена модулей были относительно короткими — в UNIX с ними проблем не будет, но проблемы могут возникнуть при переносе кода на старые версии Mac, Windows или DOS.

  Если модуль расширения, написанный на C или C++, имеет сопутствующий модуль Python, предоставляющий высокоуровневый (например, более объектно-ориентированный) интерфейс, модуль C/C++ снабжается префиксом с символом подчеркивания (например, `_socket`).

- Имена классов.

  Почти без исключений в именах классов применяется схема верблюжьего регистра. Классы, предназначенные для внутреннего использования, дополнительно помечаются начальным символом подчеркивания.

- Имена исключений.

  Поскольку исключения должны быть классами, на них распространяются правила выбора имен классов. При этом в именах исключений должен использоваться суффикс Error (если исключение является признаком ошибки).

- Имена глобальных переменных.

  (Будем надеяться, что эти переменные предназначены исключительно для использования внутри одного модуля.) Правила практически те же, что и для функций.

Модули, спроектированные для использования командой `from M import *`, должны использовать механизм `__all__`, чтобы предотвратить экспортирование глобальных имен, или старое соглашение с включением в такие имена префикса из символа подчеркивания (этот префикс показывает, что такие глобальные имена не являются общедоступными).

- Имена функций.

  Имена функций должны записываться в нижнем регистре, с разделением слов символами подчеркивания для удобства чтения.

  Смешанный регистр разрешается только в контекстах, в которых этот стиль уже превалирует (например, в threading.py) для сохранения обратной совместимости.

- Аргументы функций и методов.

  Всегда используйте `self` в первом аргументе методов экземпляров.

  Всегда используйте `cls` в первом аргументе методов классов.

  Если имя аргумента функции конфликтует с зарезервированным ключевым словом, обычно лучше присоединить один завершающий символ подчеркивания, чем использовать сокращение или искажать написание. Таким образом, `print_` лучше `prnt`. (Возможно, подобные конфликты следует предотвращать за счет использования синонима.)

- Имена методов и переменных экземпляров.

  Используйте правила выбора имен функций: нижний регистр символов, слова разделяются подчеркиваниями для удобства чтения.

  Используйте один начальный символ подчеркивания только для методов и переменных экземпляров, которые не должны быть общедоступными.

  Чтобы предотвратить конфликты имен с подклассами, используйте два начальных подчеркивания для активизации правил преобразования имен в Python.

  Python дополняет такие имена именем класса: если у класса `Foo` имеется атрибут с именем `__a`, к нему нельзя будет обратиться по имени `Foo.__a`. (Впрочем, настойчивый пользователь может получить к нему доступ по имени `Foo._Foo__a`.) В общем случае двойные символы подчеркивания в начале имени должны использоваться только для предотвращения конфликтов имен с атрибутами в классах, спроектированных в расчете на субклассирование.

  Обратите внимание: по поводу использования `__`имен мнения расходятся (см. ниже).

- Константы.

  Константы обычно объявляются на уровне модуля и записываются в верхнем регистре с разделением слов подчеркиваниями. Примеры: MAX_OVERFLOW, TOTAL.

- Проектирование с учетом наследования.

  Всегда решайте, должны ли методы класса и переменные экземпляра (называемые атрибутами) быть открытыми или приватными. Если есть сомнения, выбирайте приватные; проще сделать их открытыми позже, чем сделать открытый атрибут приватным.

  Открытые атрибуты классов предназначены для использования клиентами вашего класса, не знающими его внутренней структуры; вы обязуетесь избегать изменений, нарушающих обратную совместимость. Неоткрытые атрибуты не предназначены для стороннего использования; нет никаких гарантий, что такие атрибуты не изменятся или даже не будут удалены.

  Здесь мы не используем термин «приватный», потому что приватных атрибутов в Python на самом деле не существует (без большого объема избыточной работы).

  Другую категорию составляют атрибуты, являющиеся частью API субклассов (часто называемые *защищенными* в других языках). Некоторые классы проектируются для дальнейшего наследования с целью расширения или изменения отдельных аспектов поведения класса. При проектировании таких классов постарайтесь явно принять решения относительно того, какие атрибуты являются открытыми, какие являются частью API субклассов, а какие предназначены для использования только вашим базовым классом.

С учетом сказанного питонические рекомендации выглядят так:

- Открытые атрибуты не должны иметь начальных символов подчеркивания.
- Если имя открытого атрибута конфликтует с зарезервированным ключевым словом, присоедините один завершающий символ подчеркивания к имени атрибута. Этот вариант предпочтительнее сокращения или искаженного написания. (Впрочем, невзирая на это правило, `cls` является предпочтительным вариантом написания для любой переменной или аргумента, заведомо являющимися классом, особенно первого аргумента метода класса.)

  Примечание 1: см. приведенные выше рекомендации для методов классов.
- Для простых открытых атрибутов данных лучше предоставить доступ только к имени атрибута без сложных методов чтения/записи. Помните, что Python предоставляет простой механизм будущих расширений, если выяснится, что функциональное поведение простого атрибута данных необходимо расширить. В таком случае свойства используются для скрытия функциональной реализации за простым синтаксисом доступа к атрибутам данных.

  Примечание 1: свойства работают только в классах нового стиля.

  Примечание 2: старайтесь, чтобы функциональное поведение было свободно от побочных эффектов (хотя такие побочные эффекты, как кэширование, обычно допустимы).

  Примечание 3: избегайте использования свойств для операций, затратных в вычислительном отношении; синтаксис атрибутов создает

у вызывающей стороны впечатление, что затраты на обращение (относительно) невелики.

- Если класс предназначен для субклассирования и у вас имеются атрибуты, которые не должны использоваться субклассами, рассмотрите возможность присваивания им имен с двойным подчеркиванием в начале и без подчеркивания в конце. При этом активизируется алгоритм преобразования имен Python, который включает имя класса в имя атрибута. Тем самым предотвращаются конфликты имен в том случае, если субклассы по случайности будут содержать атрибуты с тем же именем.

  Примечание 1: в преобразованном имени используется только простое имя класса, поэтому если субклассы выберут как одинаковые имена классов, так и одинаковые имена атрибутов, конфликт имен все равно произойдет.

  Примечание 2: преобразование имен несколько усложняет некоторые задачи, такие как отладка и __getattr__(). Впрочем, алгоритм преобразования имен хорошо документирован и легко выполняется вручную.

  Примечание 3: преобразование имен нравится не всем. Постарайтесь выдержать баланс между предотвращением возможных конфликтов имен и потенциальным удобством нетривиального использования.

A.4.2. Рекомендации по программированию

Пишите код так, чтобы не создавать проблем для других реализаций Python (PyPy, Jython, IronPython, Pyrex, Psyco и т. д.).

Например, не полагайтесь на эффективную реализациию в CPython конкатенации строк «на месте» для команд в форме `a+=b` или `a=a+b`. В Jython эти команды выполняются медленнее. В частях библиотеки, критичных по быстродействию, следует использовать форму `''.join()`. Это гарантирует, что конкатенация выполняется за линейное время в разных реализациях.

Сравнения с синглетными значениями вроде `None` всегда должны осуществляться с использованием `is` или `is not` и никогда с использованием операторов проверки равенства.

Кроме того, остерегайтесь записи `if x`, когда в действительности имеется в виду `if x is not None`: например, при проверке того, было ли присвоено другое значение переменной или аргументу, по умолчанию равным `None`. Другое значение может иметь тип (например, контейнер), который может быть ложным в логическом контексте!

Используйте исключения на базе классов.

Строковые исключения запрещены в новом коде, поскольку данная возможность языка была удалена из Python 2.6.

Модули или пакеты должны определять собственный базовый класс исключений для конкретной предметной области, который должен субклассировать встроенный класс `Exception`. Всегда включайте строку документации класса, например:

```
class MessageError(Exception):
    """Базовый класс для ошибок в пакете электронной почты."""
```

При этом действуют правила выбора имен классов, хотя к классам исключений должен присоединяться суффикс `Error`, если исключение является ошибкой. Исключениям, ошибками не являющимся, специальный суффикс не нужен.

При инициировании исключения используйте форму `raise ValueError('message')` вместо старой формы `raise ValueError, 'message'`.

Форма с круглыми скобками предпочтительна, потому что если аргументы исключения образуют длинный список или включают строковое форматирование, вам не нужно использовать символы продолжения строки благодаря круглым скобкам. Старая форма была исключена в Python 3.

При перехвате исключений вместо минимальной секции `except:` везде, где это возможно, упоминайте конкретные исключения. Например, используйте

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

Минимальная секция `except:` будет перехватывать исключения `SystemExit` и `KeyboardInterrupt`, усложняя прерывание программ комбинацией Ctrl+C, и может скрывать другие проблемы. Если вы хотите перехватывать все исключения, сигнализирующие об ошибках программы, используйте `except Exception:`.

Хорошее практическое правило ограничивает использование минимальных секций `except` двумя случаями:

- если обработчик исключения будет выводить трассировку или сохранять ее в журнале — по крайней мере пользователь будет знать о возникновении ошибки;
- если код должен выполнить какую-то завершающую работу, а потом разрешает дальнейшее прохождение исключения `raise`, — но для таких случаев лучше подходит `try...finally`.

Кроме того, для всех конструкций `try/except` секция `try` должна ограничиваться минимумом абсолютно необходимого кода. И снова это помогает предотвратить сокрытие ошибок.

Да:

```
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
```

Нет:

```
try:           # Слишком широко!
    return handle_value(collection[key])
```

```
except KeyError:  ◄—— Также перехватывает KeyError, инициированное handle_value()
    return key_not_found(key)
```

Используйте строковые методы вместо модуля `string`.

Строковые методы всегда работают быстрее и используют общий API со строками Юникода. Это правило нарушается только в том случае, если необходима обратная совместимость с версиями Python старше 2.0.

Используйте `''.startswith()` и `''.endswith()` вместо сегментов строк для проверки префиксов и суффиксов.

`startswith()` и `endswith()` компактнее и снижают риск ошибок.

Да:

```
if foo.startswith('bar'):
```

Нет:

```
if foo[:3] == 'bar':
```

Исключение из правила встречается тогда, когда ваш код должен работать с Python 1.5.2 (будем надеяться, что это не так!).

Для сравнений типов объектов всегда следует использовать `isinstance()` вместо прямого сравнения типов.

Да:

```
if isinstance(obj, int):
```

Нет:

```
if type(obj) is type(1):
```

Проверяя, является ли объект строкой, помните, что он также может быть строкой Юникода! В Python 2.3 `str` и `unicode` имеют общий базовый класс `basestring`, поэтому вы можете поступить так:

```
if isinstance(obj, basestring):
```

В Python 2.2 модуль `types` содержит тип `StringTypes`, определенный для этой цели, например:

```
from types import StringTypes
if isinstance(obj, StringTypes):
```

В Python 2.0 и 2.1 можно действовать так:

```
from types import StringType, UnicodeType
if isinstance(obj, StringType) or \
   isinstance(obj, UnicodeType) :
```

Для последовательностей (строк, списков, кортежей) можно воспользоваться тем фактом, что пустые последовательности интерпретируются как False:

Да:

```
if not seq:        if seq:
```

Нет:

```
if len(seq)       if not len(seq)
```

Не пишите строковые литералы, зависящие от завершающих пропусков. Такие завершающие пропуски обычно визуально неразличимы, и некоторые редакторы (а с недавнего времени и reindent.py) усекают их.

Не сравнивайте логические значения с `True` и `False` с использованием `==`.

Да:

```
if greeting:
```

Нет:

```
if greeting == True:
```

Еще хуже:

```
if greeting is True:
```

Сведения об авторских правах: этот документ находится в открытом доступе.

А.4.3. Другие руководства по стилю Python

Хотя PEP 8 остается самым авторитетным руководством по стилю для Python, есть и другие варианты. В общем случае они не противоречат PEP 8, но предоставляют более подробные примеры и более подробные обоснования того, как сделать ваш код питоническим. Один из хороших вариантов — руководство *The Elements of Python Style* — бесплатно доступно по адресу https://github.com/amontalenti/elements-of-python-style/blob/master/README.md. Еще одно полезное руководство *The Hitchhiker's Guide to Python* Кеннета Рейтца (Kenneth Reitz) и Тани Шлюссер (Tanya Schlusser) также бесплатно доступно по адресу http://docs.python-guide.org/en/latest/.

По мере развития языка и квалификации программистов наверняка появятся новые руководства. Я рекомендую пользоваться ими, но только после того, как вы сначала ознакомитесь с PEP 8.

А.5. Дзен Python

Следующий документ PEP 20, также называемый Дзен Python, представляет собой несколько ироничное изложение философии Python. Он не только входит в документацию Python, но и включен в виде пасхалки в интерпретатор Python. Чтобы просмотреть его, введите команду `import this` в приглашении.

Питонист с большим стажем Тим Петерс (Tim Peters) свел основополагающие принципы проектирования Python от BDFL (Benevolent Dictator for Life, то есть

«великодушный пожизненный диктатор»[1]) в 20 афоризмов, только 19 из которых были записаны.

Дзен Python

Красивое лучше, чем уродливое.

Явное лучше, чем неявное.

Простое лучше, чем сложное.

Сложное лучше, чем запутанное.

Плоское лучше, чем вложенное.

Разреженное лучше, чем плотное.

Читаемость имеет значение.

Особые случаи недостаточно особые, чтобы нарушать правила.

При этом практичность важнее безупречности.

Ошибки никогда не должны замалчиваться.

Если не замалчиваются явно.

Встретив двусмысленность, отбрось искушение угадать.

Должен существовать один — и желательно только один — очевидный способ сделать это.

Хотя он поначалу может быть и не очевиден, если вы не голландец.

Сейчас лучше, чем никогда.

Хотя никогда зачастую лучше, чем прямо сейчас.

Если реализацию сложно объяснить — идея плоха.

Если реализацию легко объяснить — идея, возможно, хороша.

Пространства имен — отличная вещь! Давайте будем делать их больше!

Сведения об авторских правах: этот документ находится в открытом доступе.

[1] Имеется в виду Гвидо ван Россум. — *Примеч. пер.*

Приложение Б
Ответы на упражнения

Б.1. Глава 4

ПОПРОБУЙТЕ САМИ: ПЕРЕМЕННЫЕ И ВЫРАЖЕНИЯ

Создайте в оболочке Python несколько переменных. Что произойдет, если вы попробуете включить пробелы, дефисы или другие неалфавитные символы в имя переменной? Поэкспериментируйте с более сложными выражениями — например, $x = 2 + + 4 * 5 - 6 / 3$. Используйте круглые скобки для группировки чисел и посмотрите, как изменяется результат по сравнению с исходным выражением без группировки.

```
>>> x = 3
>>> y = 3.14
>>> y
3.14
>>> x
3
>>> big var = 12
  File "<stdin>", line 1
    big var = 12
          ^

SyntaxError: invalid syntax
>>> big-var
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'big' is not defined
>>> big&var
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'big' is not defined
>>> x = 2 + 4 * 5 - 6 /3
>>> x
20.0
>>> x = (2 + 4) * 5 - 6 /3
>>> x
28.0
>>> x = (2 + 4) * (5 - 6) /3
>>> x
-2.0
```

ПОПРОБУЙТЕ САМИ: РАБОТА СО СТРОКАМИ И ЧИСЛАМИ

В оболочке Python создайте несколько строковых и числовых переменных (целые числа, числа с плавающей точкой *и* комплексные числа). Поэкспериментируйте с различными операциями, в том числе и между типами. Можно ли, например, умножить строку на число? А умножить ее на число с плавающей точкой или комплексное число? Загрузите модуль `math` и опробуйте некоторые из его функций; затем загрузите модуль `cmath` и сделайте то же самое. Что произойдет, если вы попытаетесь вызвать одну из этих функций для целого числа или числа с плавающей точкой после загрузки модуля `cmath`? Как снова получить доступ к функциям модуля `math`?

```
>>> i = 3
>>> f = 3.14
>>> c = 3j2
  File "<stdin>", line 1
    c = 3j2
          ^

SyntaxError: invalid syntax
>>> c = 3J2
  File "<stdin>", line 1
    c = 3J2
          ^

SyntaxError: invalid syntax
>>> c = 3 + 2j
>>> c
(3+2j)
>>> s = 'hello'
>>> s * f
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'float'
>>> s * i
'hellohellohello'
>>> s * c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'complex'
>>> c * i
(9+6j)
>>> c * f
(9.42+6.28j)
>>> from math import sqrt
>>> sqrt(16)
4.0
>>> from cmath import sqrt
>>> sqrt(16)
(4+0j)
```

Чтобы вернуть первую версию `sqrt` в текущее пространство имен, его следует импортировать заново. Обратите внимание: следующий код не перезагружает файл:

```
>>> from math import sqrt
>>> sqrt(4)
2.0
```

ПОПРОБУЙТЕ САМИ: ПОЛУЧЕНИЕ ВХОДНЫХ ДАННЫХ

Поэкспериментируйте с функцией `input()` для получения строковых и целочисленных данных. Если вы используете код вроде приведенного выше, что получится, если не применять `int()` к вызову `input()` для ввода целого числа? Сможете ли вы изменить этот код, чтобы программа получала число с плавающей запятой — скажем, 28,5? Что произойдет, если намеренно ввести значение неправильного типа — например, число с плавающей точкой вместо целого, строку вместо числа или наоборот?

```
>>> x = input("int?")
int?3
>>> x
'3'
>>> y = float(input("float?"))
float?3.5
>>> y
3.5
>>> z = int(input("int?"))
int?3.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3.5'
```

БЫСТРАЯ ПРОВЕРКА: СТИЛЬ ПРОГРАММИРОВАНИЯ PYTHON

Какие из следующих имен переменных и функций *не* относятся к хорошему стилю программирования Python? Почему? `bar(, varName, VERYLONGVARNAME, foobar, longvarname, foo_bar(), really_very_long_var_name`

`bar(`: плохо и недопустимо, включает запрещенный знак.

`varName`: плохо, смешанный регистр.

`VERYLONGVARNAME`: плохо — имя слишком длинное, верхний регистр, плохо читается.

`foobar`: хорошо

`longvarname`: хорошо, хотя лучше бы разделить слова подчеркиваниями.

`foo_bar()`: хорошо.

`really_very_long_var_name`: имя длинное, но хорошее, если все слова в нем необходимы — например, чтобы различать похожие переменные.

Б.2. Глава 5

БЫСТРАЯ ПРОВЕРКА: LEN()

Что вернет функция `len()` для каждого из следующих списков: `[0]; []; [[1, 3, [4, 5], 6], 7]`?

`len([0])` – 1

`len([])`– 0

`len([[1, 3, [4, 5], 6], 7 ])` - 2

`[1, 3, [4, 5], 6]` — внутренний список является одним элементом списка перед вторым элементом 7.

ПОПРОБУЙТЕ САМИ: СЕГМЕНТЫ И ИНДЕКСЫ

Используя то, что вы знаете о функции `len()` и сегментах списков, как бы вы получили вторую половину списка неизвестного размера? Поэкспериментируйте в сеансе Python и убедитесь в том, что ваше решение работает.

```
>>> my_list = [1, 2, 3, 4, 5, 6]
>>> last_half = my_list[len(my_list)//2:]
>>> last_half
[4, 5, 6]
```

`len(my_list) // 2` — середина; сегмент от нее до конца.

ПОПРОБУЙТЕ САМИ: МОДИФИКАЦИЯ СПИСКОВ

Допустим, список состоит из 10 элементов. Как переместить три последних элемента из конца в начало списка без нарушения их исходного порядка?

```
>>> my_list = my_list[-3:] + my_list[:-3]
>>> my_list
[4, 5, 6, 1, 2, 3]
```

ПОПРОБУЙТЕ САМИ: СОРТИРОВКА СПИСКОВ

Имеется список, каждый элемент которого также является списком: `[[1, 2, 3], [2, 1, 3], [4, 0, 1]]`. Допустим, вы хотите отсортировать этот список по второму элементу каждого списка, чтобы получить результат `[[4, 0, 1], [2, 1, 3], [1, 2, 3]]`. Какую функцию вы бы написали для передачи в параметре `key` метода `sort()`?

```
>>> the_list =  [[1, 2, 3], [2, 1, 3], [4, 0, 1]]
>>> the_list.sort(key=lambda x: x[1])
>>> the_list
[[4, 0, 1], [2, 1, 3], [1, 2, 3]]
```

или

```
>>> the_list =  [[1, 2, 3], [2, 1, 3], [4, 0, 1]]
>>> the_list.sort(key=lambda x: x[1])
>>> the_list
[[4, 0, 1], [2, 1, 3], [1, 2, 3]]
```

БЫСТРАЯ ПРОВЕРКА: ОПЕРАЦИИ СО СПИСКАМИ

Какой результат вернет вызов `len([[1,2]] * 3)`?

3

Опишите два различия между оператором `in` и методом `index()` списков:

- index возвращает позицию; in возвращает логическое значение.
- index выдает ошибку, если элемент отсутствует в списке.

Какой из следующих вызовов приведет к выдаче исключения: `min(["a", "b", "c"]); max([1, 2, "three"]); [1, 2, 3].count("one")`?

`max([1, 2, "three"])`: строки и целые числа сравниваться не могут, поэтому получить максимальное значение не удастся.

ПОПРОБУЙТЕ САМИ: ОПЕРАЦИИ СО СПИСКАМИ

Имеется список `x`. Напишите код безопасного удаления элемента в том и только в том случае, если значение присутствует в списке.

```
if element in x:
    x.remove(element)
```

Измените код так, чтобы элемент удалялся только в том случае, если элемент присутствует в списке более чем в одном экземпляре.

```
if x.count(element) > 1:
    x.remove(element)
```

Примечание: этот код удаляет только первое вхождение `element`.

ПОПРОБУЙТЕ САМИ: КОПИРОВАНИЕ СПИСКОВ

Имеется следующий список: `x = [[1, 2,3], [4, 5, 6], [7, 8, 9]]`. Какой код вы бы использовали для создания копии этого списка, в которой элементы можно было бы изменять без побочного эффекта с изменением содержимого `x`?

```
import copy
copy_x = copy.deepcopy(x)
```

БЫСТРАЯ ПРОВЕРКА: КОРТЕЖИ

Объясните, почему следующие операции недопустимы для кортежа `x = (1, 2, 3, 4)`:

```
x.append(1)
x[1] = "hello"
del x[2]
```

Все эти операции изменяют объект «на месте», а кортежи изменяться не могут.

Если у вас имеется кортеж `x = (3, 1, 4, 2)`, как можно отсортировать элементы `x`?

```
x = sorted(x)
```

БЫСТРАЯ ПРОВЕРКА: МНОЖЕСТВА

Если бы вам потребовалось построить множество на базе следующего списка, то сколько элементов будет содержать это множество? `[1, 2, 5, 1, 0, 2, 3, 1, 1, (1, 2, 3)]`

Шесть уникальных элементов: 1, 2, 5, 0, 3 и кортеж (1, 2, 3)

ПРАКТИЧЕСКАЯ РАБОТА 5: АНАЛИЗ СПИСКА

В этой лабораторной работе вам поручено прочитать из файла множество температурных данных (ежемесячные температурные максимумы аэропорта Хитроу с 1948 по 2016 год), а затем вычислить некоторые статистические характеристики: максимальной и минимальной температуры, средней температуры и медианной температуры (то есть температуры, которая будет занимать центральную позицию при сортировке температур).

Температурные данные находятся в файле lab_05.txt в каталоге исходного кода этой главы. Так как чтение файлов еще не рассматривалось, приведу готовый код чтения файла в список:

```
with open('lab_05.txt') as infile:
     for row in infile:
        temperatures.append(int(row.strip())
```

Определите самую высокую и самую низкую температуру, среднюю и медианную температуру. Вероятно, вам для этого понадобятся функции/методы `min()`, `max()`, `sum()`, `len()` и `sort()`.

```
max_temp = max(temperatures)
min_temp = min(temperatures)
mean_temp = sum(temperatures)/len(temperatures)
# we'll need to sort to get the median temp
temperatures.sort()
median_temp = temperatures[len(temperatures)//2]
print("max = {}".format(max_temp))
print("min = {}".format(min_temp))
print("mean = {}".format(mean_temp))
print("median = {}".format(median_temp))
```

```
max = 28.2
min = 0.8
mean = 14.848309178743966
median = 14.7
```

Дополнительное задание: определите, сколько уникальных температур содержит список:

```
unique_temps = len(set(temperatures))
```

```
print("number of temps - {}".format(len(temperatures)))
print("number of temps - {}".format(unique_temps))
number of temps - 828
number of unique temps – 217
```

Б.3. Глава 6

БЫСТРАЯ ПРОВЕРКА: SPLIT И JOIN

Как использовать методы `split` и `join` для замены всех пропусков в строке `x` дефисами — например, преобразовать `"this is a test"` в `"this-is-a-test"`?

```
>>> x = "this is a test"
>>> "-".join(x.split())
'this-is-a-test'
```

БЫСТРАЯ ПРОВЕРКА: ПРЕОБРАЗОВАНИЕ СТРОК В ЧИСЛА

Какая из следующих строк не будет преобразована в число и почему?

1. `int('a1')`
2. `int('12G', 16)`
3. `float("12345678901234567890")`
4. `int("12*2")`

Преобразуется только третья строка `float("12345678901234567890")`; во всех остальных строках присутствует символ, запрещенный для преобразования в целое число.

БЫСТРАЯ ПРОВЕРКА: STRIP

Если строка x равна `"(name, date),\n"`, какой из следующих вызовов вернет строку `"name, date"`?

1. `x.rstrip("),")`
2. `x.strip("),\n")`
3. `x.strip("\n)(,")` удаляет символ новой строки, запятую и круглые скобки.

БЫСТРАЯ ПРОВЕРКА: ПОИСК В СТРОКАХ

Допустим, вы хотите проверить, завершается ли строка подстрокой `"rejected"`. Какой строковый метод вы для этого используете? Можно ли добиться того же результата другими способами?

```
endswith('rejected')
```

Также можно использовать запись `line[:-8] == rejected`, но такое решение будет менее понятным и питоническим.

БЫСТРАЯ ПРОВЕРКА: ИЗМЕНЕНИЕ СТРОК

Как быстро заменить все знаки препинания в строке пробелами?

```
>>> punct = str.maketrans("!.,:;-?", "       ")
>>> x = "This is text, with: punctuation! Right?"
>>> x.translate(punct)
'This is text  with  punctuation  Right '
```

ПОПРОБУЙТЕ САМИ: ОПЕРАЦИИ СО СТРОКАМИ

Допустим, имеется список строк, где некоторые (но не обязательно все) строки начинаются и завершаются символом двойной кавычки:

```
x = ['"abc"', 'def', '"ghi"', '"klm"', 'nop']
```

Какой код вы бы использовали для удаления только двойных кавычек из каждого элемента?

```
>>> for item in x:
...     print(item.strip('"'))
...
abc
def
ghi
klm
nop
```

Какой код вы бы использовали для нахождения позиции *последней буквы p* в слове *Mississippi*? А после того, как эта позиция будет найдена, какой код вы бы использовали для удаления только этой буквы?

```
>>> state = "Mississippi"
>>> pos = state.rfind("p")

>>> state = state[:pos] + state[pos+1:]
>>> print(state)

Mississipi
```

БЫСТРАЯ ПРОВЕРКА: МЕТОД FORMAT()

Что будет содержать переменная x при выполнении следующих фрагментов кода?

```
x = "{1:{0}}".format(3, 4)
'  4'

x = "{0:$>5}".format(3)
'$$$$3'

x = "{a:{b}}".format(a=1, b=5)
'    1'

x = "{a:{b}}:{0:$>5}".format(3, 4, a=1, b=5, c=10)
'    1:$$$$3'
```

БЫСТРАЯ ПРОВЕРКА: ФОРМАТИРОВАНИЕ СТРОК С СИМВОЛОМ %

Что будет содержать переменная x при выполнении следующих фрагментов кода?

```
x = "%.2f" % 1.1111
x будет содержать '1.11'
x = "%(a).2f" % {'a':1.1111}
x будет содержать '1.11'
x = "%(a).08f" % {'a':1.1111}
x будет содержать '1.11110000'
```

БЫСТРАЯ ПРОВЕРКА: БАЙТОВЫЕ СТРОКИ

Для каких из следующих разновидностей данных вы бы использовали обычные строки? В каких случаях можно использовать байтовые строки?

(1) Файл данных с двоичными данными.

Байтовые. Содержимое двоичных данных обычно вас больше интересует в числовой, а не в текстовой форме. А значит, разумнее будет использовать байты.

(2) Текст на языке, содержащем символы с диакритическими знаками.

Обычные. Строки Python 3 используют Юникод, поэтому они могут нормально обрабатывать символы с диакритическими знаками.

(3) Текст, состоящий только из букв латинского алфавита в верхнем и нижнем регистре.

Обычные. В Python 3 строки должны использоваться для всего текста.

(4) Серия целых чисел, не превышающих 255.

Байтовые. Байт позволяет представить целое число со значением не более 255, так что байтовые строки идеально подойдут для хранения таких целых чисел.

ПРАКТИЧЕСКАЯ РАБОТА 6: ПРЕДВАРИТЕЛЬНАЯ ОБРАБОТКА ТЕКСТА

При обработке текста часто требуется почистить и нормализовать текст перед тем, как делать с ним что-то еще. Например, если вы хотите подсчитать количество вхождений слов в тексте, для упрощения задачи перед началом подсчета можно позаботиться о том, чтобы весь текст был записан в нижнем регистре (или в верхнем, если предпочитаете) и из него были удалены все знаки препинания. Также для упрощения задачи можно разбить текст на серии слов.

В этой практической работе вы должны прочитать первую часть первой главы «Моби Дика» (присутствует в исходном коде книги), позаботиться о том, чтобы все символы относились к одному регистру, удалить все знаки препинания и записать слова по одному на строку во второй файл. Так как операции чтения и записи файлов в книге еще не рассматривались, я приведу код этих операций.

Ваша задача — написать код для замены закомментированных строк в следующем примере:

```
with open("moby_01.txt") as infile, open("moby_01_clean.txt", "w") as
  outfile:
    for line in infile:
        # Привести к одному регистру
        # Удалить знаки препинания
        # Разбить на слова
        # Записать все слова по одному на строку файла
        outfile.write(cleaned_words)
punct = str.maketrans("",  "", "!.,:;-?")

with open("moby_01.txt") as infile, open("moby_01_clean.txt", "w") as
outfile:
```

```
    for line in infile:
        # Привести к одному регистру
        cleaned_line = line.lower()

        # Удалить знаки препинания
        cleaned_line = cleaned_line.translate(punct)

        # Разбить на слова
        words = cleaned_line.split()
        cleaned_words = "\n".join(words)
        # Записать все слова по одному на строку файла
        outfile.write(cleaned_words)
```

Б.4. Глава 7

ПОПРОБУЙТЕ САМИ: СОЗДАНИЕ СЛОВАРЯ

Напишите код, который запрашивает у пользователя три имени и три возраста. После ввода имен и возрастов запросите у пользователя одно из имен и выведите соответствующий возраст.

```
>>> name_age = {}
>>> for i in range(3):
...     name = input("Name? ")
...     age = int(input("Age? "))
...     name_age[name] = age

>>> name_choice = input("Name to find? ")
>>> print(name_age[name_choice])

Name? Tom
Age? 33
Name? Talita
Age? 28
Name? Rania
Age? 35
Name to find? Talita
28
```

БЫСТРАЯ ПРОВЕРКА: ОПЕРАЦИИ СО СЛОВАРЯМИ

Допустим, имеются словари `x = {'a':1, 'b':2, 'c':3, 'd':4}` и `y = {'a':6, 'e':5, 'f':6}`. Что будет содержать переменная `x` при выполнении следующих фрагментов кода?

```
del x['d']
z = x.setdefault('g', 7)
x.update(y)

>>> x = {'a':1, 'b':2, 'c':3, 'd':4}
>>> y = {'a':6, 'e':5, 'f':6}
>>> del x['d']
```

```
>>> print(x)
{'a': 1, 'b': 2, 'c': 3}
>>> z = x.setdefault('g', 7)
>>> print(x)
{'a': 1, 'b': 2, 'c': 3, 'g': 7}
>>> x.update(y)
>>> print(x)
{'a': 6, 'b': 2, 'c': 3, 'g': 7, 'e': 5, 'f': 6}
```

БЫСТРАЯ ПРОВЕРКА: ЧТО МОЖЕТ ИСПОЛЬЗОВАТЬСЯ В КАЧЕСТВЕ КЛЮЧА?

Решите, какие из следующих выражений могут быть ключами словаря: `1`; `'bob'`; `('tom', [1, 2, 3])`; `["file-name"]`; `"filename"`; `("filename", "extension")`

`1`: Да.

`'bob'`: Да.

`('tom', [1, 2, 3])`: Нет; содержит список, который не является хешируемым.

`["filename"]`: Нет, это список, который не является хешируемым.

`"filename"`: Да.

`("filename", "extension")`: Да, это кортеж.

ПОПРОБУЙТЕ САМИ: РАБОТА СО СЛОВАРЯМИ

Предположим, вы пишете программу, которая должна выполнять функции электронной таблицы. Как использовать словарь для хранения содержимого таблицы? Напишите код для хранения значения и чтения значения конкретной ячейки. Какими недостатками может обладать такое решение?

Для хранения значений в словаре можно использовать в качестве ключа кортежи из строки и столбца. Один из недостатков такого решения — отсутствие сортировки ключей, поэтому вам придется самостоятельно управлять ситуацией при получении ключей/значений для отображения в электронной таблице.

```
>>> sheet = {}
>>> sheet[('A', 1)] = 100
>>> sheet[('B', 1)] = 1000

>>> print(sheet[('A', 1)])
100
```

ПРАКТИЧЕСКАЯ РАБОТА 7: ПОДСЧЕТ СЛОВ

В предыдущей практической работе вы взяли текст первой главы «Моби Дика», нормализовали регистр, удалили знаки препинания и записали разделенные слова в файл. В этой практической работе прочитайте этот файл, используйте словарь для подсчета вхождений каждого слова, а затем выведите самые частые и самые редкие слова.

Следующий код читает слова из файла в список `moby_words`:

```
moby_words = []
    for word in infile:
        if word.strip():
            moby_words.append(word.strip())

moby_words = []
with open('moby_01_clean.txt') as infile:
    for word in infile:
        if word.strip():
            moby_words.append(word.strip())

word_count = {}
for word in moby_words:
    count = word_count.setdefault(word, 0)
    count += 1
    word_count[word] += 1

word_list = list(word_count.items())
word_list.sort(key=lambda x: x[1])
print("Most common words:")
for word in reversed(word_list[-5:]):
    print(word)
print("\nLeast common words:")
for word in word_list[:5]:
    print(word)

Most common words:
('the', 14)
('and', 9)
('i', 9)
('of', 8)
('is', 7)

Least common words:
('see', 1)
('growing', 1)
('soul', 1)
('having', 1)
('regulating', 1)
```

Б.5. Глава 8

ПОПРОБУЙТЕ САМИ: ЦИКЛЫ И КОМАНДЫ IF

Допустим, имеется список `x = [1, 3, 5, 0, -1, 3, -2]`, из которого нужно удалить все отрицательные числа. Напишите код для решения этой задачи.

```
x = [1, 3, 5, 0, -1, 3, -2]
for i in x:
    if i < 0:
        x.remove(i)
```

```
print(x)
```

```
[1, 3, 5, 0, 3]
```

Как бы вы подсчитали общее количество отрицательных чисел в списке `y = [[1, -1, 0], [2, 5, -9], [-2, -3, 0]]`?

```
count = 0
y = [[1, -1, 0], [2, 5, -9], [-2, -3, 0]]
for row in y:
    for col in row:
        if col < 0:
            count += 1
print(count)
```

```
4
```

Какой код вы бы использовали для вывода описания: `"very low"`, если значение `x` меньше –5; `"low"`, если оно лежит в диапазоне от –5 до 0; `"neutral"`, если оно равно 0; `"high"`, если оно лежит в диапазоне от 0 до 5; и `"very high"`, если оно больше 5?

```
if x < -5:
    print("very low")
elif x <= 0:
    print("low")
elif x <= 5:
    print("high")
else:
    print("very high")
```

ПОПРОБУЙТЕ САМИ: ГЕНЕРАТОРЫ

Какой генератор списков вы бы использовали для обработки списка `x` с удалением всех отрицательных выражений?

```
x = [1, 3, 5, 0, -1, 3, -2]
new_x = [i for i in x if i >= 0]
print(new_x)
[1, 3, 5, 0, 3]
```

Создайте генератор, возвращающий только нечетные числа от 1 до 100. (Подсказка: нечетные числа можно отличить по наличию остатка от деления на 2; чтобы узнать остаток от деления на 2, используйте операцию `%2`.)

```
odd_100 = (x for x in range(100) if x % 2)
for i in odd_100:
    print(i))
```

Напишите код создания словаря, содержащего числа от 11 до 15 и их кубы.

```
cubes = {x: x**3 for x in range(11, 16)}
print(cubes)
{11: 1331, 12: 1728, 13: 2197, 14: 2744, 15: 3375}
```

БЫСТРАЯ ПРОВЕРКА: ЛОГИЧЕСКИЕ ЗНАЧЕНИЯ И ИСТИННОСТЬ

Решите, будет ли каждое из следующих выражений интерпретировано как истинное или ложное: `1`, `0`, `-1`, `[0]`, `1 and 0`, `1 > 0 or []`.

`1` –>: True.

`0` –>: False.

`-1`: True.

`[0]`: True; список из одного элемента.

`1 and 0`: False.

`1 > 0 or []`: True.

ПРАКТИЧЕСКАЯ РАБОТА 8: РЕФАКТОРИНГ WORD_COUNT

Перепишите программу word-count из раздела 8.7, чтобы сделать ее короче. Возможно, вам стоит вспомнить рассмотренные выше операции со строками и списками, а также продумать различные способы организации кода.

Также попробуйте сделать программу более умной, чтобы словами считались только алфавитные строки (но не знаки препинания или специальные знаки).

Листинг Б.1. Файл word_count_refactored.py

```
# File: word_count_refactored.py
""" Reads a file and returns the number of lines, words,
    and characters - similar to the UNIX wc utility
"""

# Инициализация счетчиков
line_count = 0
word_count = 0
char_count = 0

# Открытие файла
with  open('word_count.tst') as infile:
    for line in infile:
        line_count += 1
        char_count += len(line)
        words = line.split()
        word_count += len(words)

# Вывод ответов методом format()
print("File has {0} lines, {1} words, {2} characters".format(line_count,
                                            word_count, char_count))
```

Б.6. Глава 9

БЫСТРАЯ ПРОВЕРКА: ФУНКЦИИ И ПАРАМЕТРЫ

Как вы напишете функцию, которая получает любое количество неименованных аргументов, а затем выводит их значения в обратном порядке?

```
def my_funct(*params):
    for i in reversed(params):
        print(i)

my_funct(1,2,3,4)
```

Что нужно сделать, чтобы создать процедуру, то есть функцию без возвращаемого значения?

Либо не возвращайте значение (используйте минимальную команду `return`), либо вообще не включайте команду `return`.

Что произойдет, если сохранить возвращаемое значение функции в переменной?

Только одно: вы сможете использовать это значение, каким бы оно ни было.

БЫСТРАЯ ПРОВЕРКА: ИЗМЕНЯЕМЫЕ ПАРАМЕТРЫ ФУНКЦИЙ

Что произойдет при изменении списка или словаря, который был передан функции как значение параметра? Какие операции с большой вероятностью породят изменения, которые будут видны за пределами функции? Что можно сделать, чтобы свести риск к минимуму?

Изменения сохранятся для будущих использований параметра по умолчанию. Особенно вероятны проблемы от таких операций, как добавление и удаление элементов, а также изменение значения элемента. Чтобы свести риск к минимуму, лучше не использовать изменяемые типы как параметры по умолчанию.

ПОПРОБУЙТЕ САМИ: ГЛОБАЛЬНЫЕ И ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

Если предположить, что `x = 5`, чему будет равно значение `x` после выполнения приведенной ниже функции `funct_1()`? А после выполнения `funct_2()`?

```
def funct_1():
    x = 3
def funct_2():
    global x
    x = 2
```

После вызова `funct_1()` переменная `x` не изменится; после `funct_2()` значение глобальной версии `x` будет равно 2.

БЫСТРАЯ ПРОВЕРКА: ФУНКЦИИ-ГЕНЕРАТОРЫ

Что бы вы изменили в предыдущем коде функции `four()`, чтобы она работала для любого числа? Что нужно изменить в коде, чтобы начальное число последовательности тоже могло задаваться при вызове?

```
>>> def four(limit):
...     x = 0
...     while x < limit:
...         print("in generator, x =", x)
...         yield x
```

```
...         x += 1
...
>>> for i in four(4):
...     print(i)
```

Чтобы задать начальное значение:

```
>>> def four(start, limit):
...     x = start
...     while x < limit:
...         print("in generator, x =", x)
...         yield x
...         x += 1
...
>>> for i in four(1, 4):
...     print(i)
```

ПОПРОБУЙТЕ САМИ: ДЕКОРАТОРЫ

Как бы вы изменили код функции-декоратора, чтобы она не выдавала лишние сообщения и заключала возвращаемое значение упакованной функции в теги `"<html>"` и `"</html>"`, чтобы вызов `myfunction("hello")` возвращал `"<html>hello<html>"`?

Это сложное упражнение; ведь чтобы определить функцию, изменяющую возвращаемое значение, необходимо добавить внутреннюю функцию-обертку, которая вызывает исходную функцию и дополняет возвращаемое значение.

```
def decorate(func):
    def wrapper_func(*args):
        def inner_wrapper(*args):
                return_value = func(*args)
                return "<html>{}<html>".format(return_value)

        return inner_wrapper(*args)
    return wrapper_func

@decorate
def myfunction(parameter):
    return parameter

print(myfunction("Test"))
```

```
<html>Test<html>
```

ПРАКТИЧЕСКАЯ РАБОТА 9: ПОЛЕЗНЫЕ ФУНКЦИИ

Вернитесь к практическим работам глав 6 и 7 и проведите рефакторинг, выделив код очистки и обработки данных в отдельные функции. В результате большая часть логики должна размещаться в функциях. Выбирайте функции и типы параметров на свое усмотрение, но помните, что функции должны решать только одну задачу без побочных эффектов, выходящих за границы функции.

```
punct = str.maketrans("",  "", "!.,:;-?")

def clean_line(line):
    """Изменяет регистр и удаляет знаки препинания"""
```

```
    # Привести к одному регистру
    cleaned_line = line.lower()

    # Удалить знаки препинания
    cleaned_line = cleaned_line.translate(punct)
    return cleaned_line


def get_words(line):
    """Разбивает строку на слова и соединяет их с символами новой строки"""
    words = line.split()
    return "\n".join(words) + "\n"

with open("moby_01.txt") as infile, open("moby_01_clean.txt", "w")
  as outfile:
    for line in infile:
        cleaned_line = clean_line(line)

        cleaned_words = get_words(cleaned_line)

        # Записать все слова
        outfile.write(cleaned_words)

def count_words(words):
    """Получает очищенный список слов, возвращает словарь счетчиков """
    word_count = {}
    for word in moby_words:
        count = word_count.setdefault(word, 0)
        word_count[word] += 1
    return word_count


def word_stats(word_count):
    """Получает словарь счетчиков и возвращает верхние и нижние 5 элементов"""
    word_list = list(word_count.items())
    word_list.sort(key=lambda x: x[1])
    least_common = word_list[:5]
    most_common = word_list[-1:-6:-1]
    return most_common, least_common

moby_words = []
with open('moby_01_clean.txt') as infile:
    for word in infile:
        if word.strip():
            moby_words.append(word.strip())

word_count = count_words(moby_words)

most, least = word_stats(word_count)
print("Most common words:")
for word in most:
    print(word)
print("\nLeast common words:")
for word in least:
    print(word)
```

Б.7. Глава 10

БЫСТРАЯ ПРОВЕРКА: МОДУЛИ

Предположим, имеется модуль с именем `new_math`, содержащий функцию с именем `new_divide`. Какими способами можно импортировать и использовать эту функцию? Какими достоинствами и недостатками обладает каждый способ?

```
import new_math
new_math.new_divide(...)
```

Это решение часто считается предпочтительным, так как оно гарантирует отсутствие конфликтов между идентификаторами в `new_module` и импортирующими пространствами имен. С другой стороны, такое решение труднее вводить.

```
from new_math import new_divide
new_divide(...)
```

Эта версия удобнее в использовании, но она увеличивает вероятность конфликтов имен между идентификаторами модуля и импортирующего пространства имен.

Предположим, модуль `new_math` содержит вызов функции `_helper_math()`. Как начальный символ подчеркивания влияет на импортирование функции `_helper_math()`?

Она не будет импортироваться при использовании конструкции `from new_math import *`.

БЫСТРАЯ ПРОВЕРКА: ПРОСТРАНСТВА ИМЕН И ОБЛАСТИ ВИДИМОСТИ

Имеется переменная `width`, определенная в модуле `make_window.py`. В каком из следующих контекстов `width` находится в области видимости?

(A) В самом модуле.

(B) Внутри функции `resize()` из этого модуля.

(C) Внутри сценария, импортировавшего модуль `make_window.py`.

В A и B, но не в C.

ПРАКТИЧЕСКАЯ РАБОТА 10: СОЗДАНИЕ МОДУЛЯ

Упакуйте функции, созданные в конце главы 9, в автономный модуль. Хотя код выполнения модуля может быть запущен как основная программа, ваша цель — добиться того, чтобы функции могли использоваться из других сценариев.

(без ответа)

Б.8. Глава 11

ПОПРОБУЙТЕ САМИ: РАЗРЕШЕНИЕ ИСПОЛНЕНИЯ СЦЕНАРИЯ

Поэкспериментируйте с запуском сценариев на вашей платформе. Также попробуйте перенаправить ввод и вывод для ваших сценариев.

(без ответа)

БЫСТРАЯ ПРОВЕРКА: ПРОГРАММЫ И МОДУЛИ

Какую проблему должно предотвратить использование конструкции `if __name__ == "__main__":` и как это делается? Можете ли вы предложить другой способ предотвращения этой проблемы?

При загрузке модуля Python выполняет весь его код. Этот способ позволяет запустить некоторый код только в том случае, если он выполняется как основной файл сценария.

ПРАКТИЧЕСКАЯ РАБОТА 11: СОЗДАНИЕ ПРОГРАММЫ

В главе 8 вы создали версию утилиты UNIX `wc` для подсчета строк, слов и символов в файле. Теперь, когда в вашем распоряжении появилось больше инструментов, переработайте эту программу и добейтесь того, чтобы она стала ближе к оригиналу. В частности, программа должна поддерживать ключи для вывода количества только строк (`-l`), только слов (`-w`) и только символов (`-c`). Если ни один из этих ключей не задан, выводятся все три счетчика. Но если присутствует хотя бы один из ключей, то выводятся только заданные счетчики.

Чтобы немного усложнить задачу, просмотрите man-страницу `wc` для системы Linux/UNIX и добавьте ключ `-L`, чтобы выводить наибольшую длину строки. Попробуйте полностью реализовать поведение, описанное в man-странице, и сравните его с поведением утилиты `wc` в вашей системе.

```
# Файл: word_count_program.py
""" Читает файл и возвращает количество строк, слов
    и символов - по аналогии с утилитой UNIX wc
"""
import sys
def main():
    # Инициализация счетчиков
    line_count = 0
    word_count = 0
    char_count = 0

    option = None
    params = sys.argv[1:]
    if len(params) > 1:
        # Если параметров несколько, извлечь первый
        option = params.pop(0).lower().strip()
    filename = params[0]    # Открыть файл
    with  open(filename) as infile:
```

```
        for line in infile:
            line_count += 1
            char_count += len(line)
            words = line.split()
            word_count += len(words)

    if option == "-c":
        print("File has {} characters".format(char_count))
    elif option == "-w":
        print("File has {} words".format(word_count))
    elif option == "-l":
        print("File has {} lines".format(line_count))
    else:
        # Вывести ответы при помощи метода format()
        print("File has {0} lines, {1} words, {2}
characters".format(line_count,
           word_count, char_count))

if __name__ == '__main__':
    main()
```

Б.9. Глава 12

БЫСТРАЯ ПРОВЕРКА: ОПЕРАЦИИ С ПУТЯМИ

Как бы вы использовали функции модуля os, чтобы получить путь к файлу с именем test.log, создать новый путь в том же каталоге для файла с именем test.log.old? А как бы вы сделали то же самое с модулем `pathlib`?

```
import os.path
old_path = os.path.abspath('test.log')
print(old_path)
new_path = '{}.{}'.format(old_path, "old")
print(new_path)

import pathlib
path = pathlib.Path('test.log')
abs_path = path.resolve()
print(abs_path)
new_path = str(abs_path) + ".old"
print(new_path)
```

Какой путь вы получите при создании объекта pathlib `Path` на базе `os.pardir`? Попробуйте и узнайте.

```
test_path = pathlib.Path(os.pardir)
print(test_path)
test_path.resolve()

..
PosixPath('/home/naomi/Documents/QPB3E/qpbe3e')
```

ПРАКТИЧЕСКАЯ РАБОТА 12: ДРУГИЕ ОПЕРАЦИИ С ФАЙЛАМИ

Как бы вы вычислили общий размер всех файлов с расширением .txt, которые не являются символическими ссылками, в каталоге? Если в вашем первом ответе использовался модуль `os.path`, попробуйте сделать то же с `pathlib`, и наоборот.

```
import pathlib
cur_path = pathlib.Path(".")

size = 0
for text_path in cur_path.glob("*.txt"):
    if not text_path.is_symlink():
        size += text_path.stat().st_size

print(size)
```

Напишите код, который расширяет ваше предыдущее решение и перемещает те же файлы .txt в новый подкаталог с именем backup.

```
import pathlib
cur_path = pathlib.Path(".")
new_path = cur_path.joinpath("backup")

size = 0
for text_path in cur_path.glob("*.txt"):
    if not text_path.is_symlink():
        size += text_path.stat().st_size
        text_path.rename(new_path.joinpath(text_path.name))

print(size)
```

Б.10. Глава 13

БЫСТРАЯ ПРОВЕРКА

Зачем добавлять `"b"` в строку режима открытия файла — например, в `open("file", "wb")`?

Чтобы файл был открыт в двоичном режиме для чтения и записи байтов, а не символов.

Допустим, вы хотите открыть файл с именем myfile.txt и записать дополнительные данные в конец файла. Какую команду вы используете для открытия myfile.txt? Какая команда будет использоваться, чтобы повторно открыть файл для чтения данных от начала?

```
open("myfile.txt", "a")
open("myfile.txt")
```

ПОПРОБУЙТЕ САМИ: ПЕРЕНАПРАВЛЕНИЕ ВВОДА И ВЫВОДА

Напишите код, использующий модуль `mio.py`, который сохраняет весь вывод сценария в файле с именем myfile.txt, восстанавливает стандартный вывод и выводит этот файл на экран.

```
# mio_test.py

import mio

def main():
    mio.capture_output("myfile.txt")
    print("hello")
    print(1 + 3)
    mio.restore_output()

    mio.print_file("myfile.txt")


if __name__ == '__main__':
    main()

output will be sent to file: myfile.txt
restore to normal by calling 'mio.restore_output()'
standard output has been restored back to normal
hello
4
```

БЫСТРАЯ ПРОВЕРКА: STRUCT

Предложите несколько ситуаций, в которых модуль `struct` было бы удобно использовать для чтения или записи двоичных данных.

- Вы пытаетесь читать/записывать данные из файла приложения в двоичном формате или в файл с графическим изображением.
- Вы читаете данные из внешнего интерфейса (например, от термометра или акселерометра) и хотите сохранить необработанные данные точно в том виде, в каком они были переданы.

БЫСТРАЯ ПРОВЕРКА: СЕРИАЛИЗАЦИЯ PICKLE

Подумайте, будет ли сериализация с использованием `pickle` хорошим решением в следующих ситуациях:

(A) Сохранение переменных состояния между запусками.

(B) Хранение рекордных счетов в игре.

(C) Хранение имен пользователей и паролей.

(D) Хранение большого словаря со словами английского языка.

(E) Варианты A и B разумны, хотя решения с `pickle` небезопасны.

Варианты C и D нежелательны; недостаточный уровень безопасности создаст большие проблемы для C, а для D весь словарь придется загрузить в память.

БЫСТРАЯ ПРОВЕРКА: SHELVE

Работа с объектом `shelf` сильно напоминает работу со словарем. Чем отличаются объекты `shelf`? Каких недостатков следует ожидать при использовании объекта `shelf`?

Главное различие заключается в том, что объекты хранятся на диске, а не в памяти. С очень большими объемами данных, особенно с множеством операций вставки и/или удаления, обращения к диску могут замедлить работу программы.

ПРАКТИЧЕСКАЯ РАБОТА 13: ПОСЛЕДНИЕ ИСПРАВЛЕНИЯ В WC

Обратившись к man-странице утилиты `wc`, вы увидите, что два ключа командной строки решают очень похожие задачи. С ключом `-c` утилита подсчитывает байты в файле, а с ключом `-m` она подсчитывает символы (некоторые символы Юникода могут состоять из двух и более байтов). Кроме того, если файл задан, утилита должна прочитать файл и обработать его, но при отсутствии файла данных для чтения и обработки используется стандартный ввод.

Перепишите свою версию утилиты `wc`, чтобы реализовать как раздельный подсчет байтов и символов, так и возможность чтения из файлов и стандартного ввода.

```
# Файл: word_count_program_stdin.py
""" Читает файл и возвращает количество строк, слов
    и символов - по аналогии с утилитой UNIX wc
"""
import sys


def main():
    # Инициализация счетчиков
    line_count = 0
    word_count = 0
    char_count = 0
    filename = None

    option = None
    if len(sys.argv) > 1:
        params = sys.argv[1:]
        if params[0].startswith("-"):
        # Если параметров несколько, извлечь первый
            option = params.pop(0).lower().strip()
        if params:
            filename = params[0]    # Открыть файл
    file_mode = "r"
    if option == "-c":
        file_mode = "rb"
    if filename:
        infile =  open(filename, file_mode)
```

```
    else:
        infile = sys.stdin
    with infile:
        for line in infile:
            line_count += 1
            char_count += len(line)
            words = line.split()
            word_count += len(words)

    if option in ("-c", "-m"):
        print("File has {} characters".format(char_count))
    elif option == "-w":
        print("File has {} words".format(word_count))
    elif option == "-l":
        print("File has {} lines".format(line_count))
    else:
        # Вывести ответы при помощи метода format()
        print("File has {0} lines, {1} words, {2}
characters".format(line_count, word_count, char_count))

if __name__ == '__main__':
    main()
```

Б.11. Глава 14

ПОПРОБУЙТЕ САМИ: ПЕРЕХВАТ ИСКЛЮЧЕНИЙ

Напишите код, который получает два числа от пользователя и делит первое число на второе. Проверьте и перехватите исключение, возникающее в том случае, если второе число равно 0 (`ZeroDivisionError`).

```
# код вашей программы должен сделать следующее
x = int(input("Please enter an integer: "))
y = int(input("Please enter another integer: "))

try:
    z = x / y
except ZeroDivisionError as e:
    print("Can't divide by zero.")
```

```
Please enter an integer: 1
Please enter another integer: 0
Can't divide by zero.
```

БЫСТРАЯ ПРОВЕРКА: ИСКЛЮЧЕНИЯ КАК КЛАССЫ

Если `MyError` наследует от `Exception`, чем отличаются конструкции `except Exception as e` и `except MyError as e`?

Первая перехватывает все исключения, наследующие от `Exception` (то есть большинство исключений), тогда как вторая перехватывает только исключения `MyError`.

ПОПРОБУЙТЕ САМИ: КОМАНДА ASSERT

Напишите простую программу, которая запрашивает у пользователя число, а затем при помощи команды `assert` выдает исключение, если число равно 0. Протестируйте программу и убедитесь в том, что команда `assert` работает; затем отключите ее одним из способов, упомянутых в этом разделе.

```
x = int(input("Please enter a non-zero integer: "))

assert x != 0, "Integer can not be zero."
```

```
Please enter a non-zero integer: 0
---------------------------------------------------------------------
AssertionError                          Traceback (most recent call last)
<ipython-input-222-9f7a09820a1c> in <module>()
      2 x = int(input("Please enter a non-zero integer: "))
      3
----> 4 assert x != 0, "Integer can not be zero."

AssertionError: Integer can not be zero.
```

БЫСТРАЯ ПРОВЕРКА: ИСКЛЮЧЕНИЯ

Приводят ли исключения Python к вынужденному прерыванию выполнения программы?

Нет. Если исключение будет перехвачено и корректно обработано, прерывать выполнение программы не обязательно.

Предположим, вы хотите, чтобы обращение к словарю `x` всегда возвращало `None`, если ключ не существует в словаре (то есть при выдаче исключения `KeyError`). Какой код вы бы использовали для этого?

```
try:
    x = my_dict[some_key]
except KeyError as e:
    x = None
```

ПОПРОБУЙТЕ САМИ: ИСКЛЮЧЕНИЯ

Какой код вы бы использовали для создания нестандартного исключения `ValueTooLarge` и выдачи этого исключения, если значение переменной `x` превышает 1000?

```
class ValueTooLarge(Exception):
    pass

x = 1001
if x > 1000:
    raise ValueTooLarge()
```

БЫСТРАЯ ПРОВЕРКА: МЕНЕДЖЕРЫ КОНТЕКСТА

Допустим, менеджер контекста используется в сценарии, который выполняет чтение и/или запись нескольких файлов. Как вы думаете, какое из следующих решений будет лучшим?

(A) Заключить весь сценарий в блок, управляемый командой `with`.

(B) Использовать одну команду `with` для всех операций чтения файлов, а другую — для всех операций записи файлов.

(C) Использовать команду `with` каждый раз, когда выполняется чтение или запись файла (для каждой строки, например).

(D) Использовать команду `with` для каждого файла, с которым выполняется чтение или запись.

ПРАКТИЧЕСКАЯ РАБОТА 14: ПОЛЬЗОВАТЕЛЬСКИЕ ИСКЛЮЧЕНИЯ

Вспомните модуль для подсчета вхождений слов, написанный в главе 9. Какие ошибки могут возникнуть в этих функциях? Проведите рефакторинг функций, чтобы корректно обработать эти аномальные ситуации.

```
class EmptyStringError(Exception):
    pass
def clean_line(line):
    """Изменяет регистр и удаляет знаки препинания"""

    # Выдать исключение, если строка пустая
    if not line.strip():
        raise EmptyStringError()
    # Привести к одному регистру
    cleaned_line = line.lower()

    # Удалить знаки препинания
    cleaned_line = cleaned_line.translate(punct)
    return cleaned_line

def count_words(words):
    """Получает очищенный список слов, возвращает словарь счетчиков """
    word_count = {}
    for word in words:
        try:
            count = word_count.setdefault(word, 0)
        except TypeError:
            #Если 'word' не хешируется, перейти к следующему слову.
            pass
        word_count[word] += 1
    return word_count

def word_stats(word_count):
    """Получает словарь счетчиков и возвращает верхние и нижние пять элементов"""
    word_list = list(word_count.items())
    word_list.sort(key=lambda x: x[1])
    try:
```

```
        least_common = word_list[:5]
        most_common = word_list[-1:-6:-1]
    except IndexError as e:
        # Если список пустой или слишком короткий, просто вернуть список
        least_common = word_list
        most_common = list(reversed(word_list))

    return most_common, least_common
```

Б.12. Глава 15

ПОПРОБУЙТЕ САМИ: ПЕРЕМЕННЫЕ ЭКЗЕМПЛЯРОВ

Какой код вы бы использовали для создания класса `Rectangle`, представляющего прямоугольник?

```
class Rectangle:
    def __init__(self):
        self.height = 1
        self.width = 2
```

ПОПРОБУЙТЕ САМИ: ПЕРЕМЕННЫЕ ЭКЗЕМПЛЯРА И МЕТОДЫ

Обновите код класса `Rectangle`, чтобы размеры можно было задавать при создании экземпляра, как и при создании класса `Circle`. Также добавьте метод `area()`.

```
class Rectangle:
    def __init__(self, width, height):
        self.height = height
        self.width = width

    def area(self):
        return self.height * self.width
```

ПОПРОБУЙТЕ САМИ: МЕТОДЫ КЛАССА

Напишите метод класса, аналогичный `total_area()`, который возвращает суммарную длину окружности для всех экземпляров Circle.

```
class Circle:
    pi = 3.14159
    all_circles = []
    def __init__(self, radius):
        self.radius = radius
        self.__class__.all_circles.append(self)

    def area(self):
        return self.radius * self.radius * Circle.pi

    def circumference(self):
        return 2 * self.radius * Circle.pi

    @classmethod
    def total_circumference(cls):
```

```
        """Метод класса для вычисления суммарной длины окружности для всех
           экземпляров Circle"""
        total = 0
        for c in cls.all_circles:
            total = total + c.circumference()
        return total
```

ПОПРОБУЙТЕ САМИ: НАСЛЕДОВАНИЕ

Перепишите код класса `Rectangle` так, чтобы он наследовал от `Shape`. У квадратов много общего с прямоугольниками; стоит ли наследовать один класс от другого? И если стоит, то какой класс должен стать базовым, а какой должен наследовать?

```
class Shape:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Rectangle(Shape):
    def __init__(self, x, y):
        super().__init__(x, y)
```

Вероятно, наследование здесь имеет смысл. Так как квадрат является особой разновидностью прямоугольника, классу `Square` стоило бы наследовать от класса `Rectangle`.

Если бы класс `Square` содержал только один размер `x`, то метод `area()` выглядел бы так:

```
def area(self):
    return self.x * self.x
```

Как бы вы написали код добавления метода `area()` для класса `Square`? Следует ли переместить метод `area` в базовый класс `Shape`, чтобы он наследовался классами `Circle`, `Square` и `Rectangle`? И если переместить метод, к каким проблемам это приведет?

Было бы логично включить метод `area()` в класс `Rectangle`, от которого наследует `Square`, но размещение его в `Shape` особого смысла не имеет, потому что разные типы фигур используют разные правила вычисления площади. Каждая фигура все равно будет переопределять базовый метод `area()`.

ПОПРОБУЙТЕ САМИ: ПРИВАТНЫЕ ПЕРЕМЕННЫЕ ЭКЗЕМПЛЯРОВ

Измените код класса `Rectangle`, чтобы переменные размеров сторон были приватными. Какие ограничения на использование класса накладывает это изменение?

Переменные размеров будут недоступны за пределами класса в записи `.x` и `.y`.

```
class Rectangle():
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
```

ПОПРОБУЙТЕ САМИ: СВОЙСТВА

Измените поля размеров в классе `Rectangle` и преобразуйте их в свойства с get- и set-методами, не допускающими использования отрицательных размеров.

```
class Rectangle():
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, new_x):
        if new_x >= 0:
            self.__x = new_x

    @property
    def y(self):
        return self.__y

    @y.setter
    def y(self, new_y):
        if new_y >= 0:
            self.__y = new_y

my_rect = Rectangle(1,2)
print(my_rect.x, my_rect.y)
my_rect.x = 4
my_rect.y = 5
print(my_rect.x, my_rect.y)

1 2
4 5
```

ПРАКТИЧЕСКАЯ РАБОТА 15: КЛАССЫ HTML

В этой практической работе вы создадите классы для представления документов HTML. Чтобы упростить задачу, будем считать, что каждый элемент может содержать только текст и один подэлемент. Таким образом, элемент `<html>` содержит только элемент `<body>`, а элемент `<body>` содержит (необязательный) текст и элемент `<p>`, содержащий только текст.

Главное, что вам предстоит реализовать, — это метод `__str__()`, который, в свою очередь, вызывает методы `__str__()` своих подэлементов, так что при вызове функции `str()` для элемента `<html>` возвращается весь документ. Предполагается, что весь текст предшествует подэлементу.

Пример вывода с использованием классов:

```
para = p(text="this is some body text")
doc_body = body(text="This is the body", subelement=para)
```

```
doc = html(subelement=doc_body)
print(doc)

<html>
<body>
This is the body
<p>
this is some body text
</p>
</body>
</html>
```

Ответ:

```
class element:
    def __init__(self, text=None, subelement=None):
        self.subelement = subelement
        self.text = text

    def __str__(self):
        value = "<{}>\n".format(self.__class__.__name__)
        if self.text:
            value += "{}\n".format(self.text)
        if self.subelement:
            value += str(self.subelement)
        value += "</{}>\n".format(self.__class__.__name__)
        return value

class html(element):
    def __init__ (self, text=None, subelement=None):
        super().__init__(text, subelement)
    def __str__(self):
        return super().__str__()

class body(element):
    def __init__ (self, text=None, subelement=None):
        return super().__init__(text, subelement)
    def __str__(self):
        return super().__str__()

class p(element):
    def __init__(self, text=None, subelement=None):
        super().__init__(text, subelement)
    def __str__(self):
        return super().__str__()


para = p(text="this is some body text")
doc_body = body(text="This is the body", subelement=para)
doc = html(subelement=doc_body)
print(doc)
```

Б.13. Глава 16

БЫСТРАЯ ПРОВЕРКА: СПЕЦИАЛЬНЫЕ СИМВОЛЫ В РЕГУЛЯРНЫХ ВЫРАЖЕНИЯХ

Какое регулярное выражение вы бы использовали для нахождения строк, представляющих числа от –5 до 5?

`` `r"-{0,1}[0-5]"` `` совпадает со строками, представляющими числа от –5 до 5.

Какое регулярное выражение вы бы использовали для совпадения с шестнадцатеричной цифрой? Предполагается, что шестнадцатеричные цифры образуют множество 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, a, B, b, C, c, D, d, E, e, F, f.

```
`r"[0-9A-Fa-f]"`
```

ПОПРОБУЙТЕ САМИ: ИЗВЛЕЧЕНИЕ СОВПАВШЕГО ТЕКСТА

При международных звонках обычно указывается символ + и код страны. Если предположить, что код страны состоит из двух цифр, как бы вы изменили приведенный выше код? (Код страны также присутствует не во всех номерах.) Как бы вы реализовали обработку кодов стран, содержащих от одной до трех цифр?

```
re.match(r": (?P<phone>(\+\d{2}-)?(\d\d\d-)?\d\d\d-\d\d\d\d)", ":
    +01-111-222-3333")
```

или

```
re.match(r": (?P<phone>(\+\d{2}-)?(\d{3}-)?\d{3}-\d{4})", ":
    +01-111-222-3333")
```

Для кодов стран, содержащих от одной до трех цифр:

```
re.match(r": (?P<phone>(\+\d{1,3}-)?(\d{3}-)?\d{3}-\d{4})", ":
    +011-111-222-3333")
```

ПОПРОБУЙТЕ САМИ: ЗАМЕНА ТЕКСТА

В предыдущем упражнении вы доработали регулярное выражение для телефонного номера, чтобы оно также распознавало код страны. Как бы вы использовали функцию, чтобы любые номера, не содержащие кода страны, теперь имели код +1 (код страны для США и Канады)?

```
def add_code(match_obj):
    return("+1 "+match_obj.group('phone'))

re.sub(r"(?P<phone>(\d{3}-)?\d{3}-\d{4})", add_code, "111-222-3333")
```

ПРАКТИЧЕСКАЯ РАБОТА 16: НОРМАЛИЗАЦИЯ ТЕЛЕФОННЫХ НОМЕРОВ

В США и Канаде телефонные номера состоят из десяти цифр, обычно разбитых на код города из трех цифр, код шлюза из трех цифр и код станции из четырех цифр. Как упоминалось в разделе 16.4, таким номерам может предшествовать код

страны +1. Однако на практике существует много способов форматирования телефонных номеров — (NNN)NNN-NNNN, NNN-NNN-NNNN, NNN NNN-NNNN, NNN.NNN.NNNN, NNN NNN NNNN и т. д. Кроме того, код страны может отсутствовать, может не содержать + и обычно (но не всегда) отделяется от номера дефисом или пробелом.

В этом упражнении вам предлагается создать нормализатор телефонных номеров, который получает номер в любом формате и возвращает нормализованный номер вида 1-NNN-NNN-NNNN. Все номера в следующей таблице являются допустимыми:

+1 223-456-7890	1-223-456-7890	+1 223 456-7890
(223) 456-7890	1 223 456 7890	223.456.7890

Дополнительное задание: первой цифрой кода города и кода шлюза могут быть только цифры 2–9, а второй цифрой кода города не может быть 9. Используйте эту информацию для проверки ввода, чтобы при недействительном номере выдавалось исключение `ValueError`.

```
test_numbers = ["+1 223-456-7890",
                "1-223-456-7890",
                "+1 223 456-7890",
                "(223) 456-7890",
                "1 223 456 7890",
                "223.456.7890",
                "1-989-111-2222"]

def return_number(match_obj):

    # Проверить номер, выдать ValueError в случае ошибки
    if not re.match(r"[2-9][0-8]\d", match_obj.group("area") ):
        raise ValueError("invalid phone number area code
{}".format(match_obj.group("area")))
    if not re.match(r"[2-9]\d\d", match_obj.group("exch") ):
        raise ValueError("invalid phone number exchange
{}".format(match_obj.group("exch")))

    return("{}-{}-{}-{}".format(country, match_obj.group('area'),
                                match_obj.group('exch'),
match_obj.group('number')))

    country = match_obj.group("country")
    if not country:
        country = "1"

regexp = re.compile(r"\+?(?P<country>\d{1,3})?[- .]?\(?(?P<area>\
  d{3})\)?[- .]?(?P<exch>(\d{3}))[- .](?P<number>\d{4})")
for number in test_numbers:
    print(regexp.sub(return_number, number))
```

Б.14. Глава 17

БЫСТРАЯ ПРОВЕРКА: ТИПЫ

Предположим, вы хотите убедиться в том, что объект x является списком, прежде чем пытаться присоединять к нему элемент. Какой код вы используете? Чем различается использование `type()` и `isinstance()`? К какому стилю программирования относится такая проверка — LBYL («Смотри, прежде чем прыгать») или EAFP («Проще просить прощения, чем разрешения»). Какие еще возможны варианты, кроме явной проверки типа?

```
x = []
if isinstance(x, list):
    print("is list")
```

При использовании типа проверяются только списки, а не классы, субклассирующие списки. В любом случае это программирование LBYL.

Также можно заключить присоединение в блок `try… except` и перехватить исключения `TypeError`, что будет больше в стиле EAFP.

БЫСТРАЯ ПРОВЕРКА: __GETITEM__

Возможности приведенного примера использования `__getitem__` чрезвычайно ограниченны, и во многих ситуациях он работает некорректно. В каких случаях в приведенной реализации произойдет сбой или она будет работать некорректно?

Эта реализация не будет работать, если вы попытаетесь обратиться к элементу напрямую по индексу; также невозможно перемещение в обратном направлении.

ПОПРОБУЙТЕ САМИ: РЕАЛИЗАЦИЯ СПЕЦИАЛЬНЫХ МЕТОДОВ СПИСКОВ

Попробуйте реализовать специальные методы `__len__` и `__delitem__`, а также метод `append`. Реализация в коде выделена жирным шрифтом.

```
class TypedList:
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)
        self.elements = initial_list[:]
    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")
    def __setitem__(self, i, element):
        self.__check(element)
        self.elements[i] = element
    def __getitem__(self, i):
```

```
        return self.elements[i]

# Добавленные методы
def __delitem__(self, i):
    del self.elements[i]
def __len__(self):
    return len(self.elements)
def append(self, element):
    self.__check(element)
    self.elements.append(element)

x = TypedList(1, [1,2,3])
print(len(x))
x.append(1)
del x[2]
```

БЫСТРАЯ ПРОВЕРКА: СПЕЦИАЛЬНЫЕ МЕТОДЫ-АТРИБУТЫ И СУБКЛАССИРОВАНИЕ СУЩЕСТВУЮЩИХ ТИПОВ

Допустим, вам нужен тип, похожий на словарь, который разрешает использовать в качестве ключей только строки (возможно, для использования по аналогии с объектом `shelf` (см. главу 13)). Какие существуют варианты создания такого класса? Какими достоинствами и недостатками обладает каждый вариант?

Можно использовать тот же подход, который использовался для `TypedList`, и наследовать от класса `UserDict`. Также можно наследовать напрямую от `dict` или же реализовать всю функциональность `dict` самостоятельно.

Полная самостоятельная реализация предоставляет наивысшую степень контроля за происходящим, но она также требует наибольшей работы и создает наивысший риск ошибок. Если необходимые изменения малы (в данном случае проверка типа перед добавлением ключа), прямое наследование от `dict` будет наиболее логичным. С другой стороны, наследование от `UserDict`, пожалуй, является самым надежным вариантом, потому что внутренний объект `dict` так и останется обычным объектом `dict` — высокооптимизированным и с реализацией, прошедшей проверку временем.

Б.15. Глава 18

БЫСТРАЯ ПРОВЕРКА: ПАКЕТЫ

Допустим, вы пишете пакет, который получает URL-адрес, загружает все изображения со страницы, представленной URL-адресом, масштабирует их до стандартного размера и сохраняет их. Оставляя в стороне подробности реализации каждой из этих функций, как бы вы организовали эти функциональные возможности в пакет?

Пакет будет выполнять три типа действий: загрузку страницы и разбор HTML для URL-адресов изображений, загрузку изображений и изменение размеров графики. По этой причине можно рассмотреть возможность создания трех модулей для разделения этих действий:

```
picture_fetch/
    __init__.py
    find.py
    fetch.py
    resize.py
```

ПРАКТИЧЕСКАЯ РАБОТА 18: СОЗДАНИЕ ПАКЕТА

В главе 14 вы добавили обработку ошибок в модуль для чистки текста и подсчета слов, созданный в главе 11. Проведите рефакторинг кода и создайте пакет, который содержит один модуль для функций чистки, другой — для функций обработки и третий — для нестандартных исключений. Затем напишите простую функцию, которая использует все три модуля.

```
word_count
    __init__.py
    exceptions.py
    cleaning.py
    counter.py
```

Б.16. Глава 20

БЫСТРАЯ ПРОВЕРКА: РАССМОТРЕНИЕ ВАРИАНТОВ

Какие варианты существуют для решения перечисленных задач? Какие модули стандартной библиотеки вы бы предложили для этого использовать? Если хотите, прервите чтение и напишите код для решения этой задачи. Затем сравните решение с тем, которое будет разработано позднее.

Используйте модуль `datetime` из стандартной библиотеки для управления датой/временем файлов и модуль `os.path`, `os` или `pathlib` для переименования и архивации этих файлов.

БЫСТРАЯ ПРОВЕРКА: ПОТЕНЦИАЛЬНЫЕ ПРОБЛЕМЫ

Так как предыдущее решение устроено очень просто, скорее всего, оно не сможет справиться со многими ситуациями. Какие потенциальные проблемы могут возникнуть со сценарием из примера? Как решить эти проблемы?

Прежде всего появление нескольких файлов за один день создаст проблемы. Чем больше будет файлов, тем сложнее перемещаться по каталогу архива.

Возьмем схему формирования имен файлов, которая использует год, месяц и имя файла в указанном порядке. Какими преимуществами обладает эта схема? А какие у нее есть недостатки? Сможете ли вы привести доводы в пользу размещения строки даты в другом месте имени файла (например, в начале или в конце)?

При использовании форматов даты «год–месяц–день» текстовая сортировка файлов также обеспечивает их хронологическую сортировку. Размещение даты в конце имени файла, но перед расширением усложняет визуальный разбор даты.

ПОПРОБУЙТЕ САМИ: РЕАЛИЗАЦИЯ С НЕСКОЛЬКИМИ КАТАЛОГАМИ

Как бы вы изменили разработанный ранее код для архивации каждого набора файлов в подкаталоге с именем, соответствующим дате его получения? Не торопитесь, реализуйте и протестируйте каждый код.

```
import datetime
import pathlib

FILE_PATTERN = "*.txt"
ARCHIVE = "archive"

if __name__ == '__main__':

    date_string = datetime.date.today().strftime("%Y-%m-%d")

    cur_path = pathlib.Path(".")

    new_path = cur_path.joinpath(ARCHIVE, date_string)
    new_path.mkdir()

    paths = cur_path.glob(FILE_PATTERN)

    for path in paths:
        path.rename(new_path.joinpath(path.name))
```

БЫСТРАЯ ПРОВЕРКА: АЛЬТЕРНАТИВНЫЕ РЕШЕНИЯ

Как бы вы написали сценарий, делающий то же самое без использования `pathlib`? Какие библиотеки и функции вы бы использовали?

Библиотеки `os.path` и `os`, а еще конкретнее — `os.path.join()`, `os.mkdir()` и `os.rename()`.

ПОПРОБУЙТЕ САМИ: ПСЕВДОКОД АРХИВАЦИИ В ZIP-ФАЙЛЫ

Напишите псевдокод для решения, сохраняющего файлы данных в zip-файлах. Какие модули и функции вы предлагаете использовать? Попробуйте запрограммировать свое решение и убедитесь в том, что оно работает.

Псевдокод:

```
создать путь для zip-файла
создать пустой zip-файл
для каждого файла
    записать в zip-файл
    удалить исходный файл
```

(Пример кода будет приведен в следующем разделе.)

БЫСТРАЯ ПРОВЕРКА: РАЗЛИЧНЫЕ ПАРАМЕТРЫ

Рассмотрите другие варианты удаления. Как изменить код в листинге 20.4, чтобы сохранялся только один файл за месяц? Как изменить его, чтобы удалялись все

файлы за предыдущий месяц и старше, кроме одного за неделю? (Подсказка: это *не* то же самое, что файлы возрастом более 30 дней.)

Используйте код вроде приведенного выше, но также сравните месяц файла с текущим месяцем.

Б.17. Глава 21

БЫСТРАЯ ПРОВЕРКА: НОРМАЛИЗАЦИЯ

Внимательно присмотритесь к сгенерированному списку слов. Вы видите какие-нибудь проблемы с нормализацией? Какие еще трудности могут возникнуть с более обширным блоком текста? Как бы вы подошли к решению этих проблем?

Двойные дефисы вместо длинных тире, переносы при разрывах строк, любые другие знаки препинания — все это может стать источником потенциальных проблем.

Доработка модуля очистки слов, созданного в главе 18, позволила бы решить большинство таких проблем.

ПОПРОБУЙТЕ САМИ: ЧТЕНИЕ ФАЙЛА

Напишите код для чтения текстового файла (с именем temp_data_pipes_00a.txt, как показано в примере), разбиения каждой строки файла в список значений и добавления этого списка в общий список записей.

(без ответа)

С какими проблемами вы столкнулись при реализации этого кода? Как бы вы преобразовали последние три поля к правильным типам (дата, вещественное значение и целое число)?

Можно использовать генератор списка для явного преобразования этих полей.

БЫСТРАЯ ПРОВЕРКА: КАВЫЧКИ

Подумайте, как бы вы подошли к проблемам обработки полей в кавычках и внутренних символов-разделителей без библиотеки `csv`. Какая задача была бы проще: кавычки или внутренние разделители?

Без модуля `csv` вам придется проверить, что поле начинается и завершается кавычками, и затем удалить их вызовом `strip()`.

Чтобы обработать внутренние разделители без библиотеки `csv`, вам пришлось бы выделить поля в кавычках и обрабатывать их особым образом; тогда остальные поля можно будет разбить по разделителю.

ПОПРОБУЙТЕ САМИ: ОЧИСТКА ДАННЫХ

Как бы вы поступили с полями с возможными значениями `'Missing'` в математических вычислениях? Сможете ли вы написать фрагмент кода, вычисляющий среднее значение по одному из таких столбцов?

```
clean_field = [float(x[13]) for x in data_rows if x[13] != 'Missing']
average = sum(clean_field)/len(clean_field)
```

Что бы вы сделали с колонкой средних значений, чтобы сообщить средний охват? Будет ли, на ваш взгляд, решение этой проблемы вообще связано с обработкой значений `'Missing'`?

```
coverage_values = [float (x [-1] .strip ("%")) / 100]
```

Это может быть сделано не одновременно с обработкой значений `'Missing'`.

ПРАКТИЧЕСКАЯ РАБОТА 21: МЕТЕОРОЛОГИЧЕСКИЕ НАБЛЮДЕНИЯ

Файл погодных данных, приведенный в этой главе, упорядочен по месяцам и затем по округам для штата Иллинойс с 1979 по 2011 год. Напишите код, который обрабатывает этот файл для извлечения данных Чикаго (Chicago, Cook County), в один файл CSV или файл электронной таблицы. При этом строки `'Missing'` должны заменяться пустыми строками, а проценты должны преобразовываться в дробные величины. Также подумайте о том, какие поля содержат повторяющуюся информацию (а следовательно, могут быть опущены или сохранены в другом месте). Чтобы убедиться в том, что все сделано правильно, загрузите файл в приложении электронной таблицы. Решение можно загрузить в архиве исходного кода книги.

Б.18. Глава 22

ПОПРОБУЙТЕ САМИ: ЗАГРУЗКА ФАЙЛА

Допустим, вы работаете с файлом данных из нашего примера и хотите разбить каждую строку на поля; как бы вы это сделали? Какая еще обработка может потребоваться? Попробуйте написать код для загрузки этого файла и вычислить средний ежегодный уровень осадков (rain) или среднюю максимальную и минимальную температуру за каждый код (это более сложная задача).

```
import requests
response = requests.get("http://www.metoffice.gov.uk/pub/data/weather
/uk/climate/stationdata/heathrowdata.txt")

data = response.text
data_rows = []
rainfall = []
for row in data.split("\r\n")[7:]:
    fields = [x for x in row.split(" ") if x]
    data_rows.append(fields)
    rainfall.append(float(fields[5]))

print("Average rainfall = {} mm".format(sum(rainfall)/len(rainfall)))

Average rainfall = 50.43794749403351 mm
```

ПОПРОБУЙТЕ САМИ: ИСПОЛЬЗОВАНИЕ API

Напишите код для получения данных с городского сайта Чикаго. Просмотрите поля, упоминаемые в результатах, и попробуйте выполнить выборку записей по другому полю в сочетании с диапазоном дат.

```
import requests
response = requests.get("https://data.cityofchicago.org/resource/
  6zsd-86xi.json?$where=date between '2015-01-10T12:00:00' and
  '2015-01-10T13:00:00'&arrest=true")

print(response.text)
```

ПОПРОБУЙТЕ САМИ: СОХРАНЕНИЕ ДАННЫХ О ПРЕСТУПЛЕНИЯХ В ФОРМАТЕ JSON

Измените код, написанный в разделе 22.2, для загрузки данных о преступлениях в Чикаго. Преобразуйте загруженные данные из строки в формате JSON в объект Python. Затем посмотрите, удастся ли вам сохранить события преступлений в виде серии разных объектов JSON в одном файле и как один объект JSON в другом файле. Определите, какой код потребуется для загрузки каждого из файлов.

```
import json
import requests

response = requests.get("https://data.cityofchicago.org/resource/
  6zsd-86xi.json?$where=date between '2015-01-10T12:00:00' and
  '2015-01-10T13:00:00'&arrest=true")

crime_data = json.loads(response.text)

with open("crime_all.json", "w") as outfile:
    json.dump(crime_data, outfile)

with open("crime_series.json", "w") as outfile:
    for record in crime_data:
        json.dump(record, outfile)
        outfile.write("\n")

with open("crime_all.json") as infile:
    crime_data_2 = json.load(infile)

crime_data_3 = []
with open("crime_series.json") as infile:
    for line in infile:
        crime_data_3 = json.loads(line)
```

ПОПРОБУЙТЕ САМИ: ЗАГРУЗКА И РАЗБОР XML

Напишите код для извлечения прогноза погоды в Чикаго в формате XML по адресу https://graphical.weather.gov/xml/SOAP_server/ndfdXMLclient.php?whichClient=NDFDgen&lat=41.87&lon=+-87.65&product=glance. Затем используйте `xmltodict` для разбора XML

в словарь Python и извлечения прогноза максимальной температуры на завтрашний день. Подсказка: чтобы сопоставить периоды времени и значения, сравните значение `layout-key` первой секции `time-layout` и атрибут `time-layout` элемента `temperature` в элементе `parameters`.

```
import requests
import xmltodict

response = requests.get("https://graphical.weather.gov/xml/SOAP_server/
  ndfdXMLclient.php?whichClient=NDFDgen&lat=41.87&lon=+-87.65&
  product=glance")

parsed_dict = xmltodict.parse(response.text)
layout_key = parsed_dict['dwml']['data']['time-layout'][0]['layout-key']
forecast_temp =
  parsed_dict['dwml']['data']['parameters']['temperature'][0]['value'][0]
print(layout_key)
print(forecast_temp)
```

ПОПРОБУЙТЕ САМИ: РАЗБОР HTML

Для заданного файла forecast.html (находится в коде на веб-сайте книги) напишите сценарий с использованием Beautiful Soup, который извлекает данные и сохраняет их в файле CSV.

```
import csv
import bs4

def read_html(filename):
    with open(filename) as html_file:
        html = html_file.read()
        return html


def parse_html(html):
    bs = bs4.BeautifulSoup(html, "html.parser")
    labels = [x.text for x in bs.select(".forecast-label")]
    forecasts = [x.text for x in bs.select(".forecast-text")]

    return list(zip(labels, forecasts))

def write_to_csv(data, outfilename):
    csv.writer(open(outfilename, "w")).writerows(data)

if __name__ == '__main__':
    html = read_html("forecast.html")
    values = parse_html(html)
    write_to_csv(values, "forecast.csv")
    print(values)
```

ПРАКТИЧЕСКАЯ РАБОТА 22: СБОР ПОГОДНЫХ ДАННЫХ ОТ МАРСОХОДА

Используйте API, описанный в разделе 22.2, для сбора истории метеорологических данных во время пребывания марсохода «Кьюриосити» на Марсе в течение месяца.

Подсказка: чтобы задать марсианские сутки, добавьте `?sol=`*`число`* в конец запроса к архиву, например:

http://marsweather.ingenology.com/v1/archive/?sol=155

Преобразуйте данные, чтобы их можно было загрузить в электронной таблице, и создайте их графическое представление. Одна из версий проекта приведена в исходном коде книги.

```
import json
import csv
import requests

for sol in range(1830, 1863):
    response = requests.get("http://marsweather.ingenology.com/v1/
      archive/?sol={}&format=json".format(sol))
    result = json.loads(response.text)
    if not result['count']:
        continue
    weather = result['results'][0]
    print(weather)
    csv.DictWriter(open("mars_weather.csv", "a"),
  list(weather.keys())).writerow(weather)
```

Б.19. Глава 23

ПОПРОБУЙТЕ САМИ: СОЗДАНИЕ И МОДИФИКАЦИЯ ТАБЛИЦ

Используя sqlite3, напишите код, который создает таблицу базы данных для метеорологических данных штата Иллинойс, загруженных из файла в разделе 21.2. Предположим, у вас имеются аналогичные данные по другим штатам и вы хотите сохранить дополнительную информацию о штате. Как изменить базу данных, чтобы для хранения информации о штате использовалась связанная таблица?

```
import sqlite3
conn = sqlite3.connect("datafile.db")

cursor = conn.cursor()

cursor.execute("""create table weather (id integer primary key,
  state text, state_code text,
              year_text text, year_code text, avg_max_temp real,
  max_temp_count integer,
              max_temp_low real, max_temp_high real,
              avg_min_temp real, min_temp_count integer,
              min_temp_low real, min_temp_high real,
              heat_index real, heat_index_count integer,
              heat_index_low real, heat_index_high real,
              heat_index_coverage text)
              """)
conn.commit()
```

Вы можете добавить таблицу штатов и хранить в метеорологической базе данных только идентификатор каждого штата.

ПОПРОБУЙТЕ САМИ: ИСПОЛЬЗОВАНИЕ ORM

Для базы данных из предыдущего примера напишите класс `SQLAlchemy`, отображаемый на таблицу с данными. Используйте его для чтения записей из таблицы.

```
from sqlalchemy import create_engine, select, MetaData, Table, Column,
  Integer, String, Float
from sqlalchemy.orm import sessionmaker
dbPath = 'datafile.db'
engine = create_engine('sqlite:///%s' % dbPath)
metadata = MetaData(engine)
weather  = Table('weather', metadata,
                 Column('id', Integer, primary_key=True),
                 Column("state", String),
                 Column("state_code", String),
                 Column("year_text", String ),
                 Column("year_code", String),
                 Column("avg_max_temp", Float),
                 Column("max_temp_count", Integer),
                 Column("max_temp_low", Float),
                 Column("max_temp_high", Float),
                 Column("avg_min_temp", Float),
                 Column("min_temp_count", Integer),
                 Column("min_temp_low", Float),
                 Column("min_temp_high", Float),
                 Column("heat_index", Float),
                 Column("heat_index_count", Integer),
                 Column("heat_index_low", Float),
                 Column("heat_index_high", Float),
                 Column("heat_index_coverage", String)
                 )
Session = sessionmaker(bind=engine)
session = Session()
result = session.execute(select([weather]))
for row in result:
    print(row)
```

ПОПРОБУЙТЕ САМИ: ИЗМЕНЕНИЕ БАЗЫ ДАННЫХ С ALEMBIC

Поэкспериментируйте с созданием обновления Alembic, которое добавляет в базу данных таблицу штатов со столбцами для идентификатора, названия и сокращенного обозначения штата. Проведите обновление и возврат. Какие еще изменения потребовались бы, если бы вы собирались использовать таблицу штатов с существующей таблицей данных?

(без ответа)

БЫСТРАЯ ПРОВЕРКА: ИСПОЛЬЗОВАНИЕ ХРАНИЛИЩ «КЛЮЧ–ЗНАЧЕНИЕ»

Для каких типов данных и приложений лучше всего подойдет хранилище «ключ–значение», такое как Redis?

- Быстрый поиск данных.
- Кэширование.

БЫСТРАЯ ПРОВЕРКА: ИСПОЛЬЗОВАНИЕ MONGODB

Вспомните различные примеры данных, встречавшиеся ранее, и другие типы данных, с которыми вы имели дело. Как вы думаете, какие из этих данных хорошо подошли бы для хранения в такой базе данных, как MongoDB? Будут ли другие данные явно неподходящими, и если да, то почему?

Данные, поступающие в виде больших и/или менее формально структурированных блоков (например, содержимое веб-страниц или документов), хорошо подходят для MongoDB.

Данные с четко определенной структурой лучше подходят для реляционного хранения. Хороший пример такого рода — приводившиеся ранее метеорологические данные.

ПРАКТИЧЕСКАЯ РАБОТА 23: СОЗДАНИЕ БАЗЫ ДАННЫХ

Выберите один из наборов данных, рассматривавшихся в последних главах. Решите, какой тип базы данных лучше подойдет для хранения этих данных. Создайте эту базу данных и напишите код для загрузки данных. Затем выберите два самых распространенных и/или вероятных типа критериев поиска и напишите код для получения как одиночных, так и нескольких совпадающих записей.

(без ответа)

Б.20. Глава 24

ПОПРОБУЙТЕ САМИ: JUPYTER NOTEBOOK

Введите в блокноте код и поэкспериментируйте с его выполнением. Откройте меню Edit, Cell и Kernel и просмотрите содержащиеся в них команды. Когда код будет выполняться успешно, используйте меню Kernel для перезапуска ядра, повторите свои действия и используйте меню Cell для повторного запуска кода во всех ячейках.

(без ответа)

ПОПРОБУЙТЕ САМИ: ОЧИСТКА ДАННЫХ С PANDAS И БЕЗ

Поэкспериментируйте с операциями, упомянутыми выше. После того как последний столбец был преобразован в дробное значение, как бы вы преобразовали его обратно в строку с завершающим знаком %?

Для сравнения загрузите те же данные в простой список Python при помощи модуля `csv` и примените те же изменения в «чистом» коде Python.

БЫСТРАЯ ПРОВЕРКА: СЛИЯНИЕ НАБОРОВ ДАННЫХ

Как бы вы реализовали слияние таких наборов данных на Python?

Если вы уверены, что количество элементов в каждом наборе в точности совпадает, а элементы следуют в правильном порядке, используйте функцию `zip()`.

В противном случае можно создать словарь с ключами, общими между двумя наборами данных, а затем присоединить данные по ключу из обоих наборов.

БЫСТРАЯ ПРОВЕРКА: ВЫБОР В PYTHON

Какую программную структуру Python вы бы использовали для выбора только тех строк, которые удовлетворяют заданным условиям?

Вероятно, стоит воспользоваться генератором списка:

```
selected = [x for x in old_list if <x удовлетворяет критерию выбора>]
```

ПОПРОБУЙТЕ САМИ: ГРУППИРОВКА И АГРЕГИРОВАНИЕ

Поэкспериментируйте с pandas и данными из предыдущих примеров. Сможете ли вы получить информацию о звонках и объемах продаж, сгруппированную как по участникам команды, так и по месяцам?

```
calls_revenue[['Team member','Month', 'Calls', 'Amount']]
   .groupby(['Team member','Month']).sum())
```

ПОПРОБУЙТЕ САМИ: ПОСТРОЕНИЕ ГРАФИКА

Постройте линейчатый график среднего ежемесячного объема продаж на один звонок.

```
%matplotlib inline
import pandas as pd
import numpy as np

# См. описание в тексте
calls = pd.read_csv("sales_calls.csv")
revenue = pd.read_csv("sales_revenue.csv")
calls_revenue = pd.merge(calls, revenue, on=['Territory', 'Month'])
calls_revenue['Call_Amount'] = calls_revenue.Amount/calls_revenue.Calls

# Вывод графика
calls_revenue[['Month', 'Call_Amount']].groupby(['Month']).mean().plot()
```