Учимся программировать
с помощью мини-игр и загадок

Python
для начинающих

Кен Юэнс-Кларк

MANNING

Серия «Библиотека программиста»

Python для начинающих

Учимся программировать с помощью мини-игр и загадок

Кен Юэнс-Кларк

Перевод с английского М. Райтмана

Ростов-на-Дону
ФЕНИКС
2025

УДК 004.43
ББК 32.973
КТК 211
Ю99

Юэнс-Кларк, Кен.

Ю99 Python для начинающих : учимся программировать с помощью мини-игр и загадок / Кен Юэнс-Кларк ; пер. с англ. М. Райтмана. — Ростов н/Д : Феникс, 2025. — 544 с. — (Библиотека программиста).

ISBN 978-5-222-37959-2

В книге «Python для начинающих» читатели найдут задания, позволяющие изучить этот язык программирования через решение увлекательных головоломок и создание игр. Каждая глава предлагает новую программу: от создания паролей до генерации шекспировских оскорблений. Издание подойдет для читателей с нулевым или начальным опытом в программировании, а также для тех, кто уже имеет опыт, но хочет изучить новый язык.

УДК 004.43
ББК 32.973

ISBN 978-5-222-37959-2

Tiny Python Projects
text by Ken Youens-Clark

Оглавление

Предисловие

Почему именно Python?

Python — отличный универсальный язык программирования. Вы можете написать на нем программу для отправки секретных сообщений своим друзьям или игру в шахматы. Существуют модули Python, помогающие обрабатывать сложные научные данные, конструировать алгоритмы машинного обучения и создавать изображения отменного качества. Многие учебные заведения с курсами по программированию для начинающих предпочли Python таким языкам, как C и Java, из-за того, что он прост в изучении. Можно использовать Python для освоения фундаментальных и мощных идей компьютерных наук. Показав вам такие вещи, как регулярные выражения и функции высшего порядка, я надеюсь вдохновить вас на дальнейшее обучение.

Зачем я написал эту книгу?

Долгие годы я по мере возможностей помогаю людям изучать программирование и считаю его очень полезным навыком. Структура данной книги основана на моем собственном опыте работы преподавателем, опирающемся на идею, что формальные спецификации и тесты помогают создать сложную программу, разделив ее на отдельные мелкие задачи, решив которые вы придете к целому.

На мой взгляд, самое серьезное препятствие в изучении нового языка связано с тем, что его концепции обычно представляются вне полезного контекста. Большинство руководств по языкам программирования

начинаются с вывода на экран фразы "HELLO, WORLD!". И данная книга не исключение. Но после этой простой задачи я всеми силами стараюсь написать полноценную программу, которая будет принимать некие аргументы и делать что-то *полезное*.

В книге я привожу множество примеров полезных программ в надежде, что в дальнейшем вы сможете модифицировать их и использовать для создания собственных программ.

Самое важное, на мой взгляд, — много практиковаться. Как в бородатом анекдоте про музыканта, вопрошающего: «Как попасть в Карнегихолл?» Ответ звучит так: «Репетировать, уважаемый, только репетировать». Приведенные в книге задачи по программированию достаточно короткие, и вы, думаю, справитесь с каждой из них за несколько часов, ну или дней. Здесь собрано больше материала, чем я даю за семестр в вузе, поэтому на работу с ним может уйти несколько месяцев. Я надеюсь, что вы решите задачи, проанализируете их, а позже снова к ним вернетесь, чтобы попробовать решить их иначе. Например, использовать более продвинутый способ или ускорить работу программ.

Благодарности

Это моя первая книга, и для меня очень важно, что многие люди помогали мне в ее создании. Все началось со звонка Майка Стефенса, рецензента издательства Manning, у которого возникла идея написать книгу о том, как с помощью простеньких игр и головоломок научиться разрабатывать серьезное, рабочее ПО. Далее я созвонился с Марьян Бейс, издателем, и она с энтузиазмом восприняла концепцию разработки через тестирование, мотивирующую читателей к активному участию в написании программ.

Один из редакторов-консультантов по аудитории, Сюзанна Клайн, помогла мне превратить первые несколько глав книги в нечто читабельное. Второй редактор, Элеша Хайд, поддерживала меня долгие месяцы, пока я писал книгу, терпеливо и вдумчиво редактируя и рецензируя рукопись. Я благодарю технических редакторов Скотта Чаусси, Эла Шерера и Матийса Аффуртита за тщательную проверку всех листингов и текста на отсутствие ошибок. Я высоко ценю усилия сотрудников программы раннего доступа издательства Manning, особенно Мехмеда Пашича, за создание PDF-файлов и инструктаж по использованию AsciiDoc. Я также хотел бы поблагодарить менеджера проектов Дейрдру Хиам, копирайтера Энди Кэрролла, корректора Кэти Теннант и рецензента Александра Драгосавлевича. Кроме того, спасибо всей аудитории liveBook и обозревателям за замечательные отзывы: Аманде Деблер, Конору Редмонду, Дрю Леону, Хоакину Белтрану, Хосе Апаблаза, Кимберли Уинстон-Джексон, Мацею Юрковски, Мафинару Хан, Мануэлю Рикардо Гонсалес Кове, Марселю ван ден Бринку, Марчину Секу, Матийсу Аффуртиту, Полу Р. Хендрику, Шейну Корнуэллу, Виктору М. Пересу.

Отдельно я хочу поблагодарить бесчисленное количество людей, создающих ПО с открытым исходным кодом, на котором работает весь наш мир. От родоначальников языка Python, модулей и документации к нему до бесчисленных гениев, отвечающих на вопросы на форумах в интернете. Всем вам спасибо за то, что вы делаете.

Конечно, книга не увидела бы свет без поддержки моей семьи, в частности моей жены Лори Киндлер, неиссякаемого источника любви свыше 27 лет. (Я до сих пор очень и очень сожалею о том, что свалился с горного велосипеда и целый год приходил в себя!) Наши трое детей приносят столько испытаний и радости, и я надеюсь, что они будут гордиться мной. Им постоянно приходится имитировать интерес к знакомым и мало волнующим их темам, и они проявляли потрясающее терпение на протяжении всего времени, пока я писал эту книгу.

О книге

Кому предназначена книга

Я надеюсь, что, прочитав эту книгу и воспользовавшись практическими примерами, вы станете фанатом создания документируемых, тестируемых и работающих программ.

В моем понимании идеальный читатель — это тот, кто пытается научиться хорошо программировать, но не знает, как прокачать свой уровень. Возможно, вы экспериментировали с Python или другим языком с похожим синтаксисом вроде Java(Script) или Perl. Или вы освоили что-то совершенно иное, например Haskell или Scheme, и решили узнать, как воплотить те же идеи на Python. А может, вы какое-то время программировали на Python и ищете интересные задачи, чтобы понимать, что двигаетесь в правильном направлении.

Данная книга научит вас писать идеально структурированный, документированный и тестируемый код на Python. Она познакомит с лучшими программистскими методами, такими как *разработка через тестирование*, когда *тесты* для программы появляются раньше ее самой! Я покажу вам, как читать документацию и предложения по улучшению Python (PEP, Python Enhancement Proposals), а также как писать идиоматический код, с ходу понятный другим программистам на Python.

Вероятно, книга не совсем подойдет начинающему программисту. Я не описываю сами основы языка Python, потому что предполагаю, что вы с ними уже знакомы (ну или с основами другого какого-то языка). Если вы никогда не писали программы *ни на каком* языке, то прежде, чем читать данную книгу, вам стоит освоить такие концепции, как переменные, циклы и функции.

Структура книги

Каждая глава данной книги основана на концепциях предыдущих глав, поэтому я настоятельно рекомендую вам читать с самого начала и последовательно работать с материалом.

- Все программы содержат аргументы командной строки, поэтому мы начнем с изучения модуля `argparse`. Кроме того, все программы тестируются, поэтому вам следует научиться устанавливать и использовать фреймворк `pytest`. Введение и первая глава помогут вам в этом.
- В главах 2–4 обсуждаются основные структуры Python, такие как строки, списки и словари.
- Главы 5 и 6 посвящены приемам работы с файлами в качестве входных и выходных данных и объясняют, как файлы связаны со «стандартными потоками» `STDIN` и `STDOUT`.
- В главах 7 и 8 мы начинаем комбинировать концепции, чтобы вы могли писать более сложные программы.
- В главах 9 и 10 рассказывается о модуле `random` и о том, как отслеживать и тестировать случайные события.
- В главах 11–13 вы узнаете о разделении кода на функции и о том, как писать и запускать для них тесты.
- В главах 14–18 мы начнем разбираться в более сложных темах, таких как функции высшего порядка, а также освоим регулярные выражения, чтобы использовать поисковые шаблоны.
- В главах 19–22 мы будем писать более сложные, «реальные» программы, которые объединят все полученные навыки, углубляя ваши знания языка Python и тестирования.

О коде

Программы и тесты, представленные в книге, вы найдете по адресу: **https://github.com/kyclark/tiny_python_projects**.

Системные требования

Все программы написаны и протестированы в Python 3.8, но для большинства подойдет и версия 3.6. Потребуется несколько дополнительных

модулей, например `pytest` для запуска тестов. В книге приведены инструкции, как их устанавливать с помощью модуля `pip`.

Ресурсы в интернете

Во многих материалах, обучающих программированию, отсутствуют наглядные примеры, демонстрирующие создание функционирующих программ. Работая со студентами, я подробно показываю, как написать программу, а затем добавить и протестировать новые функции. Я записал вспомогательные видеоролики ко всем главам и разместил их на канале **www.youtube.com/user/kyclark.** Для каждой главы вы найдете свой плей-лист. Видеоролики демонстрируют примеры из книги, в них ставятся проблемы и раскрываются возможности языка, которые могут помочь вам при написании собственных программ, после чего следует обсуждение решений.

Об авторе

Меня зовут Кен Юэнс-Кларк. Я работаю старшим научным программистом в Аризонском университете. Большая часть моей карьеры связана с биоинформатикой — анализом биологических данных с помощью информатики.

Я начал свое обучение с барабанщика на бакалавриате по специальности «джазовые исследования» в Университете Северного Техаса в 1990 году. Несколько раз поменяв специальность, в конце концов я получил степень бакалавра по английской литературе в 1995 году. Тогда я не имел представления о своей будущей карьере, знал только, что мне нравятся компьютеры.

Где-то в 1995 году я начал разбираться с базами данных и HTML на своей первой после колледжа работе. Там я занялся списком рассылки компании и разрабатывал веб-сайт. И тут я увидел свой дальнейший путь! Я выучил язык Visual Basic для Windows 3.1, затем какое-то время программировал на разных языках в нескольких компаниях, пока в 2001 году не попал в группу биоинформатики в лаборатории Колд-Спринг-Харбор под руководством Линкольна Штейна, известного автора книг и модулей для языка Perl и одного из первых сторонников открытого ПО, данных и науки. В 2014 году я переехал в Тусон, штат Аризона, и устроился на работу в Аризонский университет, попутно в 2019 году получив степень магистра в области биосистемной инженерии.

На досуге мне нравится барабанить, кататься на велосипеде, готовить, читать и проводить время с женой и детьми.

Об обложке

Изображение на обложке этой книги называется *Femme Turc allant par les rues*, или «Прогуливающаяся турчанка». Иллюстрация взята из коллекции костюмов разных стран Жака Грассе де Сен-Совера (1757–1810) под названием *Costumes de Différents Pays*, опубликованной во Франции в 1788 году. Каждая иллюстрация тщательно прорисована и раскрашена вручную. Богатое разнообразие коллекции Грассе де Сен-Совера ярко напоминает нам о том, насколько далекими друг от друга в культурном отношении были жители разных городов и стран каких-то два века назад. Они говорили на различных диалектах и языках. В городах и селах можно было по одежде легко определить, где живет человек, кем работает и сколько зарабатывает.

В настоящее время различия в одежде смазались и разнообразие, столь богатое в те времена, исчезло. Сейчас трудно отличить жителей разных континентов, не говоря уже о разных городах, регионах или странах. Вероятно, мы променяли культурное разнообразие на более разнообразную личную жизнь — и уж точно на более разнообразную и динамичную технологическую жизнь.

Большинство компьютерных книг сложно отличить друг от друга. Издательство Manning отдает дань изобретательности и инициативности IT-индустрии книжными обложками, отражающими богатое разнообразие представителей культур, живших два столетия назад. За это мы благодарны художнику Жаку Грассе де Сен-Соверу.

Начало работы. Введение и руководство по установке

Данная книга научит вас писать консольные Python-программы. Если вы никогда раньше не пользовались командной строкой, не волнуйтесь. Можно прибегнуть к таким инструментам, как PyCharm (рис. B.1) или Microsoft Visual Studio Code, чтобы с удобством создавать и запускать консольные приложения. Если вы новичок в программировании или не знакомы с языком Python, я постараюсь охватить все, что, по моему мнению, вам нужно знать. Но все же полезно сначала прочитать другую книгу, знакомящую с такими азами, как переменные и функции.

В этом введении мы обсудим:

— в чем польза умения разрабатывать консольные программы;
— инструменты и среды для написания кода;
— каким образом и для чего тестируется ПО.

Консольные программы

Почему я хочу научить вас писать консольные программы? Во-первых, на мой взгляд, в таких программах есть только самый необходимый код. Мы не будем разрабатывать сложные программы типа интерактивных 3D-игр, для работы которых требуется множество других модулей и компонентов. Все программы из этой книги запускаются с минимумом

входных данных и генерируют только текстовый вывод. Мы сосредоточимся на изучении основ языка Python и на том, как писать и *тестировать* программы.

Еще одна причина в пользу изучения именно консольных программ заключается в том, что они запускаются на любом компьютере, на котором установлен Python. Я пишу эту книгу на ноутбуке Mac и при этом могу запускать все описанные программы на рабочем компьютере под управлением операционной системы Linux или в Windows-среде, которой пользуются друзья. Любой компьютер с идентичной версией Python поддерживает запуск описанных здесь программ, и это круто!

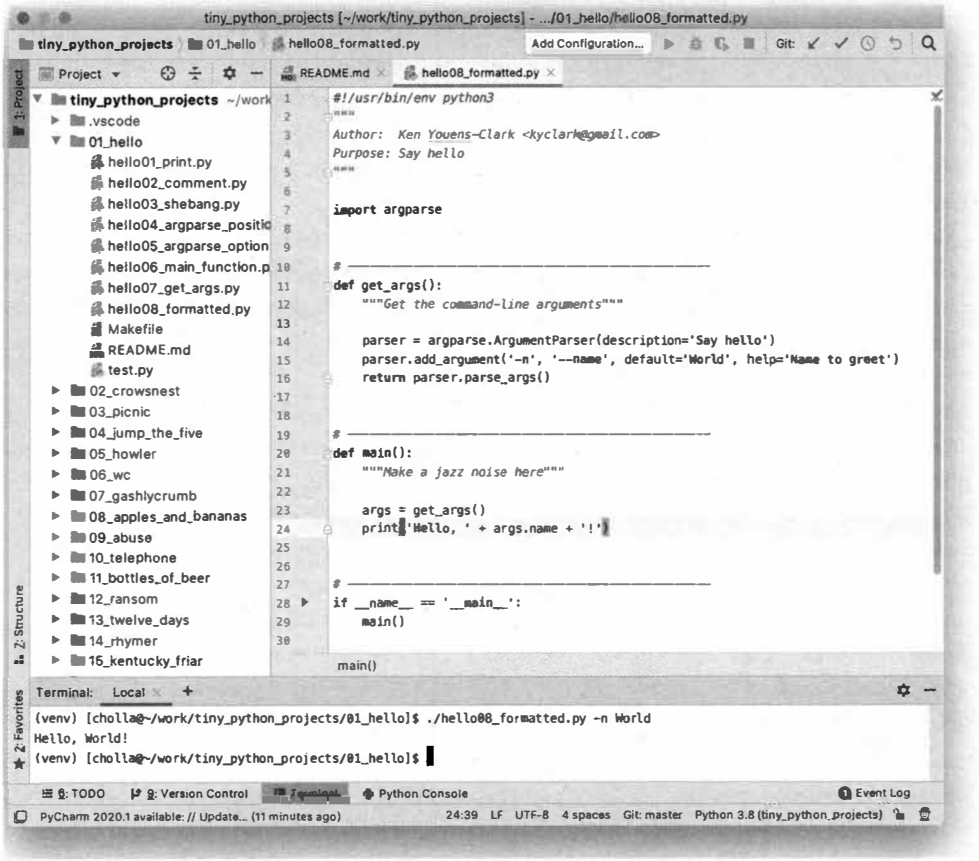


Рис. B.1. Инструмент PyCharm, используемый для редактирования кода и запуска программы *hello.py* из главы 1

Однако главная причина выбора консольных программ заключается в том, что с их помощью можно нагляднее проводить *тестирование*,

позволяющее убедиться в их работоспособности. Конечно, если я допущу ошибку, армагеддон не случится, но все же я очень и очень хочу быть уверенным, что мой код настолько совершенен, насколько это возможно.

Как провести тестирование? Что ж, если моя программа должна складывать два числа, мне нужно запустить ее со многими парами чисел и убедиться, что она выводит правильную сумму. Можно также передать ей число и слово и тем самым проверить, что она не пытается прибавить 3 к словосочетанию «морской конек», а жалуется, что я не ввел два числа. Тестирование формирует некоторую степень уверенности в коде и, надеюсь, поможет вам глубже понять программирование.

Упражнения, приведенные в книге, можно назвать дурашливыми. Они должны возбудить ваш интерес, но при этом каждое из них призвано научить вас решать реальные проблемы. Практически все написанные мной программы принимают определенные входные данные, будь то пользовательский ввод или файл, и генерируют некий вывод — текст на экране или, быть может, новый файл. Такие навыки вы и приобретете, работая с ними.

В каждой главе я описываю программы, которые вам нужно написать, и тесты, используемые для проверки корректности их работы. Затем я показываю и обсуждаю решения. По мере усложнения материала я предлагаю варианты самостоятельного создания тестов для анализа и проверки кода.

После прочтения книги вы сможете:

— писать и запускать консольные Python-программы;
— управлять аргументами ваших программ;
— писать и выполнять тесты для своих программ и функций;
— использовать структуры данных Python, такие как строки, списки и словари;
— читать и записывать текстовые файлы в своих программах;
— использовать регулярные выражения для поиска текстовых шаблонов;
— вносить в программы аспекты случайности для непредсказуемой работы приложений.

«Коды — это головоломки. Как игра, любая другая игра».

— Алан Тьюринг

Алан Тьюринг наиболее известен тем, что взломал код шифровальной машины «Энигма», которую нацисты использовали для защищенного обмена данными во время Второй мировой войны. Считается, что благодаря возможности дешифровки секретных сообщений нацистов

союзные войска смогли сократить войну и спасти миллионы жизней. *«Игра в имитацию»* — забавный фильм, согласно сюжету которого Тьюринг публикует в газетах головоломки, пытаясь найти людей, способных их решить, дабы с их помощью взломать считавшийся крайне стойким код.

Полагаю, мы можем многому научиться, создавая забавные программы, генерирующие случайные шуточки, сочиняющие рождественские стихи или позволяющие сыграть в «Крестики-нолики». Некоторые из программ, приведенных в книге, даже немного связаны с криптографией. Например, в главе 4 мы будем кодировать все числа в тексте, а в главе 18 создадим сигнатуры слов, суммируя числовые представления содержащихся в них букв. Я надеюсь, что данные программы покажутся вам одновременно и забавными, и увлекательными, вдохновляющими принять вызов.

Методы программирования в упражнениях не специфичны для Python. Практически в каждом языке программирования есть переменные, циклы, функции, строки, списки и словари, а также способы параметризации и тестирования программ. После того как вы напишете свои решения на Python, я рекомендую вам переписать их на другом знакомом вам языке программирования и сравнить, какие моменты решений в разных языках упрощаются, а какие усложняются. Если программы поддерживают идентичные параметры командной строки, вы даже можете использовать для их проверки приведенные здесь тесты.

Разработка через тестирование

Разработку через тестирование описал Кент Бек в 2002 году в одноименной книге* как метод создания более надежных программ. Основная идея заключается в том, что тесты нужно писать раньше основного кода. Тесты определяют, каким образом программа должна работать «правильно». *Сначала* мы пишем и запускаем тесты, проверяющие, что код не работает. Затем проводим самостоятельную работу для прохождения каждого теста. Мы всегда выполняем *все тесты*, чтобы, создавая новые, не привести к отказу тех, что успешно выполнялись до этого. Когда все тесты пройдены, у нас появляется уверенность (по крайней мере, частичная), что написанный нами код соответствует требуемой спецификации.

Каждая программа в этой книге сопровождается тестами, демонстрирующими, работает ли код должным образом. Первый тест в любом

---

* «Экстремальное программирование. Разработка через тестирование». *Кент Бек.*

из упражнений проверяет, существует ли ожидаемая программа. Второй — выводит ли программа инструкции, когда мы их запрашиваем. Затем программа запускается с различными входными данными и параметрами.

Поскольку я написал порядка 250 тестов для программ, описанных в данной книге, а вы еще ни одного, вы неизбежно столкнетесь с провалами в тестировании. Не волнуйтесь! На самом деле это очень хорошо, так как, преодолев все трудности и справившись с тестированием, вы приобретете уверенность в корректности кода созданных вами программ. Вы научитесь внимательно анализировать провальные тесты, чтобы понять, что нужно исправить. Затем будете править код и снова запускать тесты. И так снова и снова. До тех пор пока, наконец, все тесты не будут пройдены. И тогда вы сможете считать работу выполненной.

Неважно, решите вы проблемы так же, как я, или каким-то иным способом. Главное, чтобы вы нашли ошибки и успешно провели тесты.

Настройка среды программирования

Чтобы писать программы из этой книги, вам необходима версия Python 3.6 или более поздняя. Вам также понадобится инструмент, способный выполнить команду `python3`, называемый *командной строкой*, *терминалом* или *консолью*. Если вы используете компьютер под управлением операционной системы Windows, можно установить подсистему Windows для Linux (WSL). На Mac достаточно приложения Terminal, устанавливаемого по умолчанию. Вы также можете использовать такие инструменты, как VS Code (рис. B.2) или PyCharm, в которые встроены собственные терминалы.

Я писал и тестировал программы для этой книги в Python версии 3.8, но они должны работать и с версией 3.6 или новее. Поддержка Python 2 была прекращена в конце 2019 года, поэтому ее больше не рекомендуется использовать. Чтобы узнать, какая версия Python установлена у вас, откройте окно командной строки и выполните команду `python3 --version`. Если консоль выводит что-то вроде «команда „python3“ не найдена», вам нужно загрузить последнюю версию с сайта **www.python.org/downloads**.

Если на вашем компьютере нет возможности установить Python, можно выполнять примеры из этой книги на веб-сайте Repl.it (**http://repl.it**).

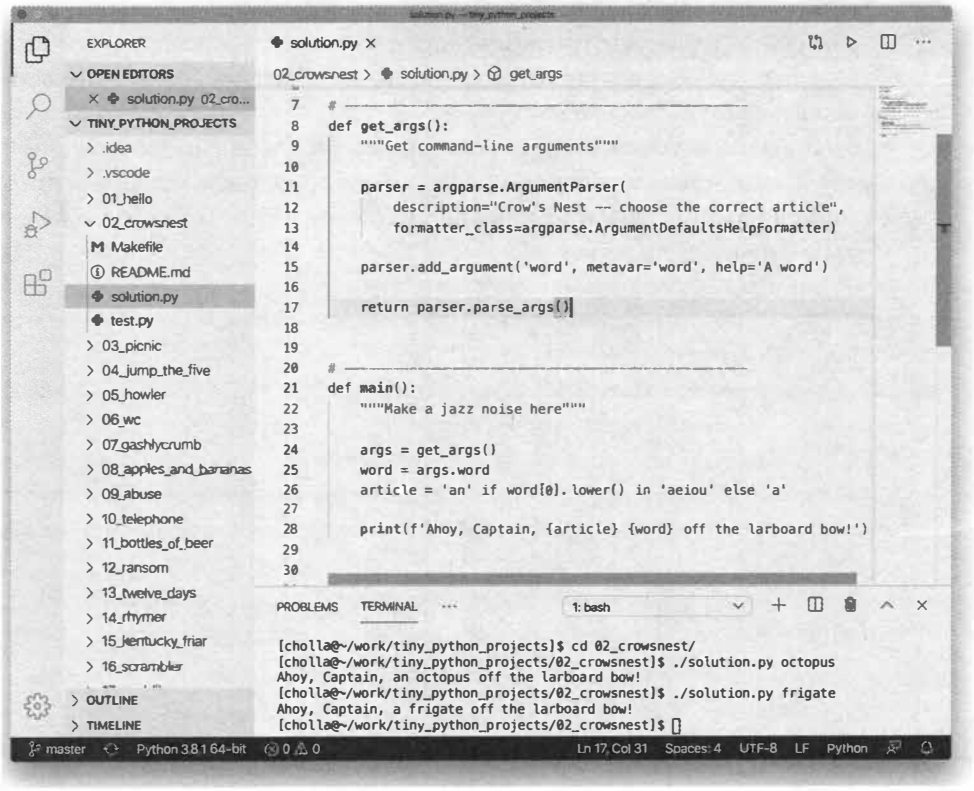


Рис. B.2. IDE, например VS Code, сочетает в себе текстовый редактор для написания кода, терминал (нижнее правое окно) для запуска программ и многие другие инструменты

Листинги

Все команды и код, присутствующие в книге, выделены с помощью моноширинного шрифта. Если тексту предшествует знак доллара (`$`), его можно ввести в командной строке. Например, программа `cat` (сокращение от concatenate — конкатенация) выводит на экран содержимое файла. Вот как я могу запустить ее для вывода на экран содержимого файла *spiders.txt*, который находится в каталоге *input*:

```
$ cat inputs/spiders.txt
Don't worry, spiders,
I keep house
casually.
```

Если вы хотите выполнить данную команду, *не копируйте* символ `$` в начале, а только следующий за ним текст `cat inputs/spiders.txt`. В противном случае вы, скорее всего, получите сообщение об ошибке в духе «$: команда не найдена».

В Python встроен IDLE — отличный инструмент, позволяющий напрямую взаимодействовать с интерпретатором ради экспериментов. Вы можете запустить его с помощью команды `idle3`. Откроется новое окно с приглашением типа `>>>` (рис. В.3).

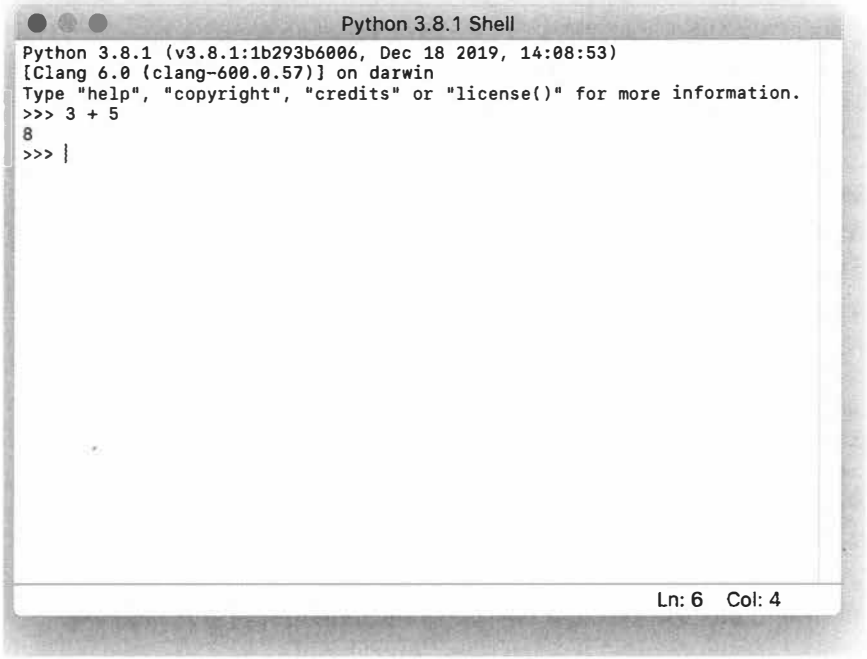


Рис. В.3. Приложение IDLE позволяет напрямую взаимодействовать с языком Python. Каждый введенный вами оператор выполняется по нажатию клавиши Enter, а результаты выводятся в окне

Вы можете ввести операторы Python, и они будут немедленно выполнены и выведены. Например, напечатайте строку `3 + 5` и нажмите клавишу **Enter**. В ответ вы увидите число `8`:

```
>>> 3 + 5
8
```

Данный интерфейс называется *REPL* (от англ. read-eval-print-loop — цикл «чтение — вычисление — вывод»). Вы можете открыть его, выполнив в командной строке команду `python3` (рис. В.4).

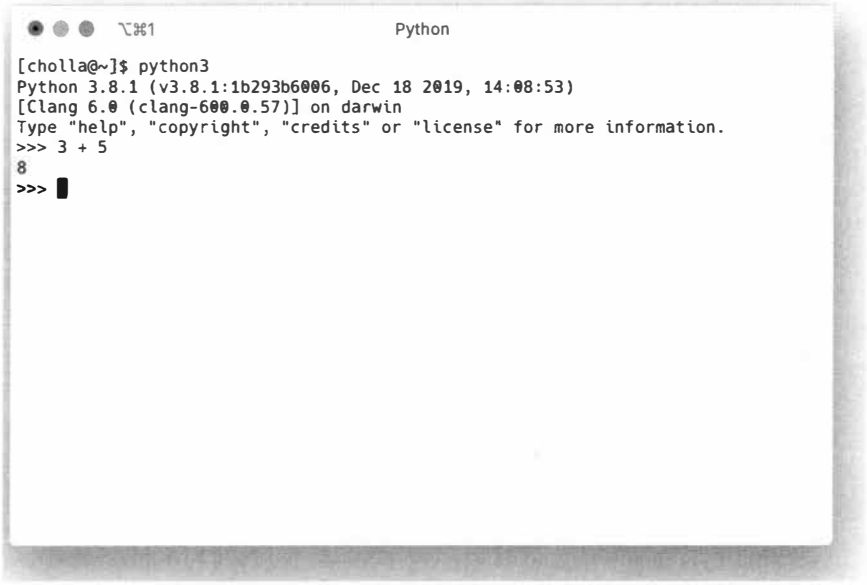


Рис. B.4. Выполнение команды `python3` в консоли открывает REPL-интерфейс, аналогичный IDLE

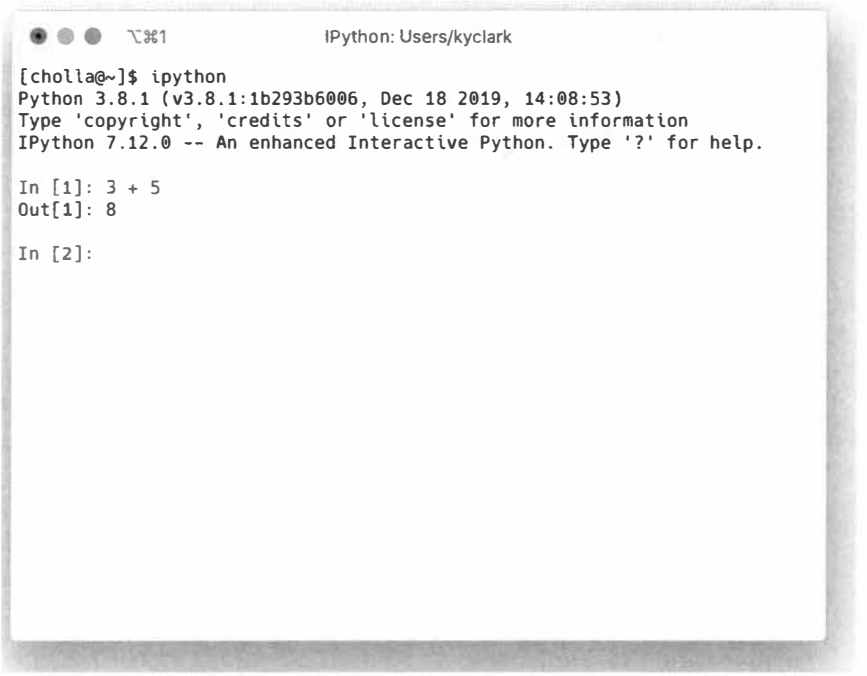


Рис. B.5. Инструмент IPython — это еще один REPL-интерфейс, в котором вы можете реализовать свои идеи на языке Python

Инструмент IPython — еще один интерактивный REPL-интерфейс Python, который имеет множество улучшений по сравнению с IDLE и программой `python3`. На рис. B.5 показано, как он выглядит в моем случае. Я также рекомендую вам освоить блокноты Jupyter Notebook, поскольку они позволяют в интерактивном режиме запускать код с дополнительной возможностью сохранять блокноты в файлы и предоставлять доступ к ним другим людям.

Какой бы REPL-интерфейс вы ни использовали, вы можете вводить операторы Python, такие как `x = 10`, и нажимать клавишу **Enter**, чтобы присвоить значение `10` переменной `x`:

```
>>> x = 10
```

Как и в случае с приглашением командной строки, символом `$`, не копируйте символы `>>>` в начале строки, иначе получите ошибку:

```
>>> >>> x = 10
  File "<stdin>", line 1
    >>> x = 10
     ^
SyntaxError: invalid syntax
```

REPL-интерфейс IPython обладает «магическим» режимом `%paste`, удаляющим приглашение `>>>`, чтобы вы могли правильно копировать и вставлять примеры кода:

```
In [1]: >>> x = 10

In [2]: x
Out[2]: 10
```

Какой бы способ взаимодействия с Python вы ни выбрали, я предлагаю вам *вручную набирать весь код* из этой книги, чтобы тренировать память и запоминать синтаксис языка программирования.

Примеры кода

Все тесты и решения доступны на сайте **https://github.com/kyclark/tiny_python_projects**. Вы можете использовать программу Git (ее нужно установить отдельно), чтобы скопировать примеры кода на свой компьютер с помощью следующей команды:

```
$ git clone https://github.com/kyclark/tiny_python_projects
```

В таком случае на вашем компьютере появится каталог *tiny_python_projects*.

Вы можете скопировать код в собственный репозиторий, чтобы отслеживать свои изменения и делиться решениями с другими пользователями. Данный процесс, так называемое разветвление, позволяет работать независимо от моего репозитория и добавлять программы в собственный. Если для выполнения упражнений вы планируете использовать Repl.it, я рекомендую вам создать ветвь моего репозитория в своем аккаунте GitHub и настроить Repl.it для работы со своими репозиториями. Для этого выполните следующие действия:

1) создайте учетную запись на сайте GitHub.com;
2) перейдите по адресу: **https://github.com/kyclark/tiny_python_projects**;
3) нажмите кнопку **Fork** (рис. B.6), чтобы скопировать репозиторий в свою учетную запись.

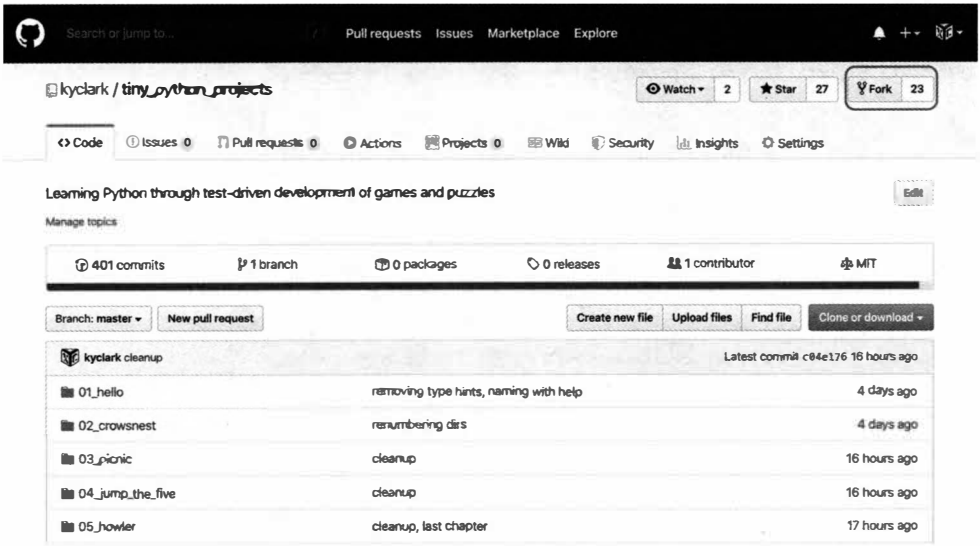


Рис. B.6. Кнопка Fork в репозитории GitHub создает копию файлов из него в вашей учетной записи

Теперь у вас есть копия всех моих файлов в собственном репозитории. Вы можете использовать инструмент Git, чтобы скопировать их на свой компьютер. Обязательно замените значение *ВАШ_GITHUB_ID* своим идентификатором GitHub:

```
$ git clone https://github.com/ВАШ_GITHUB_ID/tiny_python_projects
```

Я могу внести изменения в свой репозиторий после того, как вы скопируете его. Если вы хотите тоже получить эти обновления, вам нужно настроить Git так, чтобы мой репозиторий использовался в качестве исходного источника. Для этого после клонирования своего репозитория на компьютер перейдите в каталог *tiny_python_projects*:

```
$ cd tiny_python_projects
```

Затем выполните следующую команду:

```
$ git remote add upstream https://github.com/kyclark/tiny_python_
projects.git
```

Всякий раз, когда вам потребуется обновить свой репозиторий из моего, вы можете выполнять такую команду:

```
$ git pull upstream master
```

Установка модулей

Я рекомендую установить несколько инструментов, которые помогут при работе с книгой. Чтобы это сделать, воспользуйтесь модулем `pip`:

```
$ python3 -m pip install black flake8 ipython mypy pylint pytest
yapf
```

Я добавил файл *requirements.txt* в корень репозитория. Вы можете использовать его для установки всех рекомендуемых модулей и инструментов с помощью следующей команды:

```
$ python3 -m pip install -r requirements.txt
```

К примеру, если вы хотите выполнять упражнения в среде Repl.it, вам нужно выполнить данную команду, чтобы установить требуемые для работы модули.

Форматирование кода

В большинстве IDE и текстовых редакторов доступны средства форматирования кода, чтобы его было легче читать и находить проблемы.

Кроме того, сообщество Python сформулировало стандарт по оформлению кода, позволяющий программистам избегать трудностей при его чтении. В документе «PEP 8 — руководство по написанию кода на Python» на сайте **www.python.org/dev/peps/pep-0008** описаны передовые методы форматирования кода.

Отмечу, что большинство редакторов автоматически форматируют код. Для этого в интерфейсе Repl.it есть кнопка **Autoformat** (рис. B.7), в VS Code доступна команда **Format Document**, а в PyCharm используется команда **Reformat Code**.

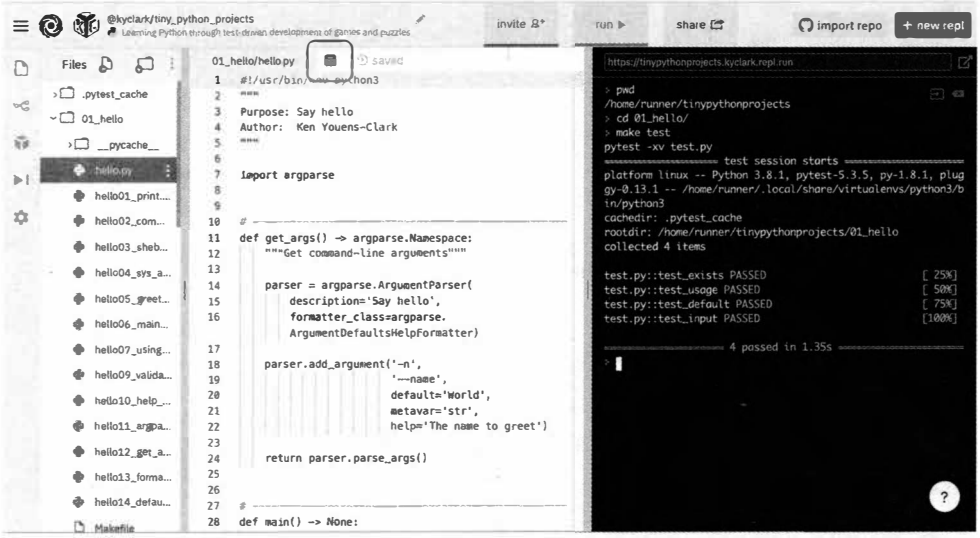


Рис. B.7. В окне Repl.it есть кнопка Autoformat для автоматического форматирования кода в соответствии со стандартами сообщества Python. Интерфейс также содержит командную строку для выполнения и тестирования кода

Кроме того, существуют консольные инструменты, которые интегрируются с разными редакторами. Я, к примеру, пользовался инструментом YAPF (Yet Another Python Formatter, **https://github.com/google/yapf**) для форматирования листингов в книге. Другое популярное средство форматирования кода — это Black (**https://github.com/psf/black**).

Какой бы инструмент вы ни использовали, главное — применяйте его *почаще*. К примеру, я могу с помощью YAPF отформатировать программу *hello.py*, которую мы напишем в главе 1, выполнив команду, приведенную ниже. Обратите внимание, что параметр `-i` позволяет перезаписать содержимое исходного файла отформатированным кодом «на месте».

```
$ yapf -i hello.py
```

Линтеры

Линтер — это инструмент, информирующий о проблемах в коде, например об объявлении неиспользуемой переменной. Два лучших, на мой взгляд, — это Pylint (**www.pylint.org**) и Flake8 (**http://flake8.pycqa.org/en/latest**). Оба ищут ошибки, на которые интерпретатор Python не обращает внимания.

В последней главе я покажу вам, как использовать в коде *аннотации типов*. С их помощью инструмент Муру (**http://mypy-lang.org**) способен обнаруживать такие проблемы, как указание программистом текста вместо чисел.

С чего начать

Думаю, гораздо проще начать писать код со стандартного шаблона, поэтому я создал программу *new.py*, генерирующую Python-программы с шаблонным кодом. Данный файл расположен в каталоге *bin*, поэтому, если вы находитесь в корне репозитория, вы можете запустить программу следующим образом:

```
$ bin/new.py
usage: new.py [-h] [-s] [-n NAME] [-e EMAIL] [-p PURPOSE] [-f]
program
new.py: error: the following arguments are required: program
```

Как видно из результата, *new.py* просит указать имя файла будущей программы, которую следует создать. Программа для каждой главы должна находиться в каталоге с соответствующим файлом *test.py*.

В качестве примера используем файл *new.py* для создания программы *crowsnest.py* из главы 2 в каталоге *02_crowsnest*:

```
$ bin/new.py 02_crowsnest/crowsnest.py
Done, see new script "02_crowsnest/crowsnest.py."
```

Открыв файл, вы увидите много строк кода, которые я объясню позже. А пока лишь запомните, что программа *crowsnest.py* может быть запущена следующим образом:

```
$02_crowsnest/crowsnest.py
usage: crowsnest.py [-h] [-a str] [-i int] [-f FILE] [-o] str
crowsnest.py: error: the following arguments are required: str
```

Позже вы узнаете, как изменить код программы, чтобы она успешно проходила тесты.

Вместо создания файла с помощью приложения *new.py* можно скопировать файл *template.py* из каталога с шаблонами в нужную папку и переименовать его. Например, файл *crowsnest.py* можно создать так:

```
$ cp template/template.py 02_crowsnest/crowsnest.py
```

Вам не понадобятся файлы *new.py* и *template.py* для запуска собственных программ. Они предназначены лишь для того, чтобы сэкономить время и предоставить начальную структуру кода, который можно переписывать так, как вам заблагорассудится.

А как же Jupyter Notebook?

Многие люди знакомы с интерактивными блокнотами Jupyter Notebook. Они позволяют интегрировать Python-код, текст и изображения в документ, который другие пользователи могут выполнять как программу. Мне очень нравится платформа Jupyter Notebook, особенно для интерактивного анализа данных, но использовать ее в обучении сложно по следующим причинам.

- Содержимое блокнота хранится в формате JSON (JavaScript Object Notation), а не в виде строкового текста. Это очень затрудняет сравнение содержимого разных блокнотов, чтобы понять, чем они отличаются.
- Код, текст и изображения хранятся в независимых ячейках. Ячейки можно запускать в любом порядке, что способно привести к очень хитрым и трудно обнаруживаемым проблемам в логике работы программы. Все программы, приведенные в этой книге, выполняются построчно сверху вниз и целиком, что, как мне кажется, упрощает их понимание.
- Блокноты не принимают разные значения при запуске. То есть, если вы тестируете программу с одним входным файлом, а затем хотите перейти к другому файлу, вам придется изменить саму программу. Из книги же вы узнаете, как передать файл в качестве аргумента программе, чтобы в дальнейшем можно было сменить файл без изменения кода.
- Сложно автоматизированно тестировать блокноты или содержащиеся в них функции. Мы воспользуемся инструментом `pytest`, чтобы запускать программы с разными входными данными и проверять, генерируют ли они корректный вывод.

Темы, которые мы рассмотрим

Цель данной книги — показать вам, насколько удивительно полезны встроенные функции языка Python. Благодаря упражнениям вы попрактикуетесь в работе со строками, списками, словарями и файлами. Я посвятил несколько глав регулярным выражениям, и все упражнения, кроме последнего, требуют, чтобы вы передавали и проверяли аргументы командной строки различных типов и чисел.

Каждый автор предвзято относится к темам своей книги, и я не исключение. Я выбрал такие темы, потому что они отражают идеи, лежащие в основе моей работы последние 20 лет. Например, я потратил намного больше часов, чем готов признать, на анализ хаотичных данных из бесчисленных электронных таблиц Excel и XML-файлов. Мир геномики, поглотивший львиную долю моей карьеры, основан в первую очередь на эффективном анализе текстовых файлов, и бóльшая часть моей деятельности в веб-разработке основана на понимании того, как текст кодируется и передается в веб-браузер и из него. Поэтому множество упражнений в книге связаны с обработкой текста и файлов, и вам предстоит задуматься о том, как преобразовывать входные данные в выходные. Если вы качественно проработаете все предложенные здесь задачи, вы намного глубже поймете основные концепции многих языков программирования.

Почему не ООП?

Обратите внимание: в книге отсутствует материал, касающийся написания объектно ориентированного Python-кода. Если вы не знакомы с *объектно ориентированным программированием* (ООП), можете пропустить текущий раздел. В противном случае читайте дальше.

Я считаю, что ООП — довольно сложная тема, выходящая за рамки данной книги. Я предпочел сосредоточиться на написании небольших функций и соответствующих тестов. Так код будет читабельнее, поскольку функции короткие, используют значения, только явно переданные в качестве аргументов, и имеют достаточно тестов, чтобы вы могли в полной мере понять поведение программ в любых условиях, как благоприятных, так и наоборот.

Язык Python сам по себе объектно ориентирован. Почти все элементы, от строк до списков и словарей, которыми мы воспользуемся, на самом деле являются *объектами*, так что вы серьезно попрактикуетесь в их использовании. Но не думаю, что необходимо создавать объекты для решения какой-либо из представленных проблем. Хотя я не один

год занимался написанием объектно ориентированного кода, в последние несколько лет я отказался от него. Я склонен черпать вдохновение из мира чисто функционального программирования и надеюсь, что к концу этой книги смогу убедить вас в том, что вы способны написать любую программу, комбинируя функции.

Несмотря на то что я отказался от ООП, я рекомендую все же познакомиться с ним. К настоящему моменту успело произойти несколько существенных сдвигов парадигмы от процедурного к объектно ориентированному, а теперь и к функциональному программированию. На полках магазинов полно книг про ООП в целом и про программирование объектов на языке Python в частности. Это глубокая и увлекательная тема, и я призываю вас пробовать писать объектно ориентированные решения и сравнивать их с моими.

О жаргоне

Часто в книгах по программированию в примерах используется слово *foobar*. Слово не является каким-то определенным термином и происходит, вероятно, от армейского акронима времен Второй мировой, FUBAR — Fouled Up Beyond All Recognition (англ. «Разбито в хлам»). Употребляя слово foobar в примере, я имею в виду не что-то конкретное, а лишь строку символов. Если мне нужен список элементов, обычно первым в нем идет foo, а следующим — bar. По традиции следом используются слова baz и quux, которые тоже ничего не значат. Не заморачивайтесь насчет foobar. Это просто заглушка для подходящего значения в ваших будущих проектах.

Программисты называют ошибки в коде «багами» (от англ. слова bug — насекомое). Название придумали во времена старых вычислительных машин до изобретения транзисторов. В ту пору в ЭВМ использовались радиолампы, тепло от которых привлекало насекомых, например мотыльков, а те, в свою очередь, садясь на контакты, вызывали короткое замыкание. Операторы (люди, управляющие ЭВМ) разбирали компьютеры и извлекали насекомых; отсюда и произошли термины «баг» и «отладка» (с англ. debug, т. е. удалить насекомое).

Глава 1

Пишем и тестируем приложение на Python

Прежде чем приступить к упражнениям, давайте обсудим, как писать программы, документировать и тестировать их. Вот что мы сделаем.

- Напишем программу на Python, выводящую строку `Hello, World!`
- Обработаем аргументы командной строки с помощью модуля `argparse`.
- Протестируем наш код, используя инструмент Pytest.
- Разберемся с переменной `$PATH`.
- Отформатируем код с помощью таких инструментов, как YAPF и Black.
- Поищем проблемы в коде с применением инструментов Flake8 и Pylint.
- Используя программу `new.py`, научимся создавать новые приложения.

1.1. Создание первой программы

Обычно начинающие разработчики, независимо от используемого языка, первым делом пишут программу, выводящую текст "Hello, World!". Давайте поступим так же. Однако предлагаю создать вариацию данной

программы. Она помимо приветствия будет выводить имя, переданное ей в качестве аргумента, а также некое сообщение по запросу. А еще мы воспользуемся тестами и проверим, что программа работает должным образом.

В папке *01_hello* есть несколько версий программы-приветствия. Еще там находится программа *test.py*, которой мы воспользуемся для тестирования нашего приложения-приветствия.

Начнем с создания текстового файла *hello.py* в данном каталоге. Если вы работаете в программе Visual Studio Code или PyCharm, можно воспользоваться командой меню **File ⇨ Open** (Файл ⇨ Открыть), чтобы открыть каталог *01_hello* в качестве проекта. Обе программы содержат команду меню в духе **File ⇨ New** (Файл ⇨ Создать), позволяющую создать новый файл в данном каталоге. Крайне важно создать файл *hello.py* именно в папке *01_hello*, чтобы программа *test.py* имела к нему доступ.

После создания нового файла добавьте в него следующую строку:

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

Настало время запустить новую программу! Откройте панель терминала в Visual Studio Code, PyCharm или иной терминал, которым вы пользуетесь, и перейдите в каталог с файлом *hello.py*. Вы можете запустить его с помощью команды `python3 hello.py` — таким образом интерпретатор Python версии 3 выполнит команды, указанные в файле *hello.py*. Вы увидите следующее:

```
$ python3 hello.py Hello, World!
```

На рис. 1.1 показано, как выглядит результат в интерфейсе Repl.it.

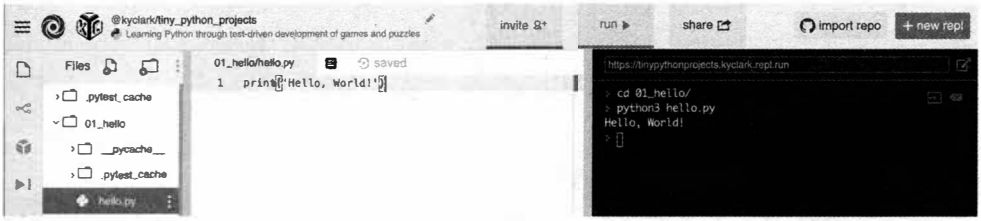


Рис. 1.1. Код и результат выполнения нашей первой программы в интерфейсе Repl.it

Если это первая программа на Python, которую вы написали и запустили, примите мои поздравления!

1.2. Комментирование кода

В языке Python символ # и все, что следует за ним в строке, игнорируется. Такое поведение полезно для добавления комментариев к коду и временного предотвращения выполнения строк кода при тестировании и отладке программ. Настоятельно рекомендуется документировать свои программы, указывая их предназначение, имя разработчика, адрес электронной почты и т. п. Мы можем написать такой комментарий:

```
# Предназначение программы: поприветствовать мир
print('Hello, World!')
```

Если вы запустите данную программу, то увидите тот же результат, что и раньше, потому что строка `# Предназначение...` игнорируется. Обратите внимание, что код *слева* от символа # выполняется, поэтому вы можете добавлять комментарии в конец строки, если сочтете нужным.

1.3. Тестирование программы

Самое главное, чему я хочу вас научить, это тестирование. Вот почему я добавил программу *test.py* в каталог *01_hello*. Ее можно использовать для тестирования приложения *hello.py*.

Мы воспользуемся инструментом pytest для выполнения команд и вывода результатов — сколько тестов пройдено. Параметр `-v` программы `pytest` генерирует «подробный» вывод. Если вы запустите ее с этим параметром, результат будет таким, как продемонстрировано ниже. В примере показано несколько первых строк. После них следует еще много информации о тестах, которые были провалены.

Примечание. Если вы получаете сообщение об ошибке «pytest: команда не найдена», вам необходимо установить модуль `pytest`. Обратитесь к разделу «Установка модулей» во введении к книге.

Второй тест запускает программу командой `python3 hello.py`, а затем проверяет, вывела ли программа текст "Hello, World!". Если вы пропустите символ(ы), например запятую, то в результатах теста увидите ошибку, так что читайте внимательно!

```
$ pytest -v test.py
====================== test session starts ======================
...
collected 5 items

test.py:: test_exists PASSED                                  [20%]
test.py:: test_runnable PASSED                                [40%]
test.py:: test_executable FAILED                              [60%]
test.py:: test_usage FAILED                                   [80%]
test.py:: test_input FAILED                                   [100%]

============================ FAILURES ============================
```

Первый тест всегда проверяет существование ожидаемого файла. В данном примере программа ищет файл *hello.py*.

Четвертый тест запрашивает инструкции к программе и ничего не получает. Тест провален. Мы добавим оператор, выводящий инструкции, как использовать нашу программу.

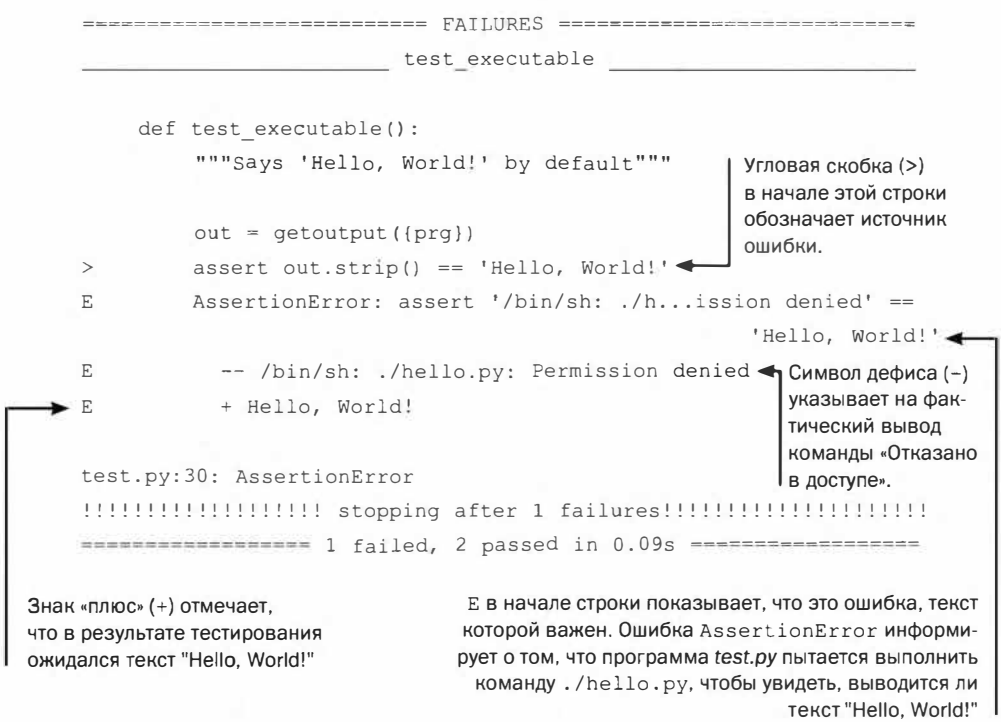
Последний тест проверяет, может ли программа выводить имя, которое мы передаем в качестве аргумента. Поскольку наша программа еще не принимает аргументы, необходимо реализовать и это.

Третий тест проверяет, является ли программа «исполняемой». Тест провален. Далее мы поговорим о том, как его пройти.

Я расположил тесты в том порядке, который, надеюсь, поможет вам написать программу логически верно. Если программа проваливает один из тестов, нет смысла выполнять последующие. Я рекомендую всегда запускать тесты с флагом `-x`, чтобы прекратить тестирование после первого же провала, и с флагом `-v`, чтобы вывести подробный результат. Можно комбинировать параметры, например `-xv` или `-vx`. Вот как выглядят результаты тестирования с указанными параметрами:

```
$ pytest -xv test.py
====================== test session starts ======================
...
collected 5 items

test.py:: test_exists PASSED                                  [20%]
test.py:: test_runnable PASSED                                [40%]
test.py:: test_executable FAILED                              [60%]
```

Данный тест не пройден. Последующие не выполняются, поскольку мы запустили программу `pytest` с параметром `-x`.



Обсудим, как исправить полученную ошибку.

1.4. Использование шебанга (#!)

Вы уже усвоили, что Python-программы — это обычные текстовые файлы, которые вы запускаете с помощью `python3`. Многие другие языки программирования, подобно Ruby и Perl, работают аналогично: мы добавляем операторы Ruby или Perl в текстовый файл и запускаем его в нужном интерпретаторе. В таких программах обычно размещают специальную строку комментария, указывающую, какой язык необходимо использовать для выполнения операторов в файле.

Данная строка начинается с символов # ! , т. н. «шебанга». Как и в случае с другими комментариями, интерпретатор Python игнорирует строку с указанными символами, а операционная система (например, macOS или Windows) использует эту информацию, чтобы определить, какую программу следует применить для запуска остальной части файла.

Ниже показан код, который вам нужно добавить:

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

Программа `env` сообщает о вашем «окружении». Выполняя `env` на своем компьютере, я вижу в выводе много строк типа `USER=kyclark` и `HOME=/Users/kyclark`. Данные значения доступны через переменные `$USER` и `$HOME`:

```
$ echo $USER
kyclark
$ echo $HOME
/Users/kyclark
```

Запустив программу `env` на своем компьютере, вы увидите собственный логин и путь к домашнему каталогу. Разумеется, эти значения будут отличаться от моих, но оба они должны быть настроены.

Можно использовать команду `env` для поиска и запуска программ. Выполнив команду `env python3`, вы запустите программу `python3`, если та будет обнаружена. Вот как это выглядит на моем компьютере:

```
$ env python3
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Программа `env` ищет в окружении приложение `python3`. Если Python не установлен, программа не сможет его найти. Случается, что есть несколько версий Python. Тогда можно использовать команду `which`, чтобы определить, какую версию программы `python3` находит `env`:

```
$ which python3
/Library/Frameworks/Python.framework/Versions/3.8/bin/python3
```

Если я выполню данный код на платформе Repl.it, то увижу, что `python3` расположен по другому адресу. А где он расположен на вашем компьютере?

```
$ which python3
/home/runner/.local/share/virtualenvs/python3/bin/python3
```

Точно так же, как отличается от вашего мое значение переменной `$USER`, вероятно, отличается и путь к каталогу `python3`. Если команда `env` определит путь `python3`, она запустит программу. Как было

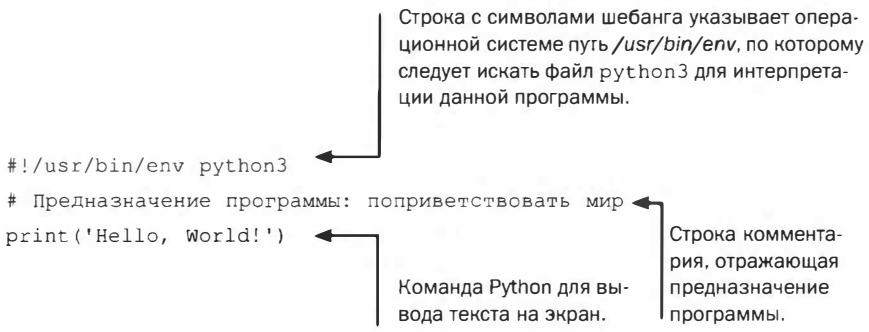
показано ранее, при самостоятельном запуске файла `python3` открывается REPL-интерфейс.

Если я укажу путь к `python3` с символами шебанга, вот так:

```
#!/Library/Frameworks/Python.framework/Versions/3.8/bin/python3
```

моя программа не будет работать ни на одном компьютере, где `python3` установлен по другому пути. Наверняка и на вашем устройстве она не запустится. Вот почему вы всегда должны использовать программу `env`, позволяющую найти путь к `python3`, уникальный для каждого компьютера.

Теперь ваша программа должна выглядеть так:



1.5. Исполнение программы

До сих пор мы целенаправленно указывали интерпретатору `python3` запускать нашу программу. Однако поскольку мы добавили символы шебанга, мы можем выполнить программу независимо и позволить ОС определить, что потребуется интерпретатор `python3`. Преимущество такого способа в том, что мы можем скопировать нашу программу в каталог с другими программами и выполнять ее из любого каталога на компьютере.

Первый шаг в достижении цели — сделать нашу программу «исполняемой» с помощью команды `chmod` (*change mode — изменить режим*). Представьте себе, что так вы «активируете» программу. Выполните показанную ниже команду, чтобы сделать файл *hello.py* исполняемым:

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

Параметр +x добавляет к файлу атрибут «исполняемый» (*eXecutable*).

Теперь можно запустить программу следующим образом:

```
$ ./hello.py
Hello, World!
```

./ — текущий каталог, и программу необходимо запускать, когда вы находитесь в том же каталоге, что и программа.

1.6. О переменной $PATH

Одна из главных причин для указания шебанга и создания исполняемой программы заключается в том, что благодаря этому вы можете устанавливать свои программы Python точно так же, как другие программы и команды. Ранее мы использовали команду `which`, чтобы определить расположение интерпретатора `python3` в экземпляре Repl.it:

```
$ which python3
/home/runner/.local/share/virtualenvs/python3/bin/python3
```

Как программа `env` нашла его? Дело в том, что операционные системы Windows, macOS и Linux содержат переменную `$PATH`, представляющую собой список каталогов, в которых ОС ищет требуемую программу. Например, вот содержимое переменной `$PATH` для моего экземпляра Repl.it:

```
> echo $PATH
/home/runner/.local/share/virtualenvs/python3/bin:/usr/local/bin:\
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Каталоги разделяются двоеточиями (`:`). Обратите внимание, что каталог, в котором находится интерпретатор `python3`, указан в `$PATH` первым. Строка получилась довольно длинной, поэтому для удобства чтения я разделил ее с помощью символа `\`. Если вы скопируете свою программу *hello.py* в любой из каталогов, перечисленных в переменной `$PATH`, то сможете выполнить ее и другие подобные программы без указания пути `./` и без необходимости находиться в том же каталоге, что и программа.

Чтобы понять работу переменной `$PATH`, представьте, что вы потеряли ключи в квартире. Как вы будете их искать? Начнете ли с подвесных, напольных и выдвижных ящиков на кухне, после чего обыщете

ванную комнату и спальню? Или сначала проверите те места, куда обычно кладете ключи, например тумбочку возле входной двери, карманы любимой куртки, сумки/рюкзака, а затем, допустим, посмотрите под диванными подушками? Ухватили суть?

Переменная `$PATH` — это способ указать вашему компьютеру искать исполняемые программы только в определенных местах. Другой вариант — поиск во *всех каталогах*, что может занять минуты, а то и часы! Вы способны управлять как именами каталогов в переменной `$PATH`, так и их порядком, чтобы ОС успешно и быстро находила нужные программы.

Очень часто программы устанавливаются в каталог */usr/local/bin*. Мы можем скопировать нашу программу в него с помощью команды `cp`. К сожалению, на Repl.it я не обладаю нужными разрешениями:

```
> cp 01_hello/hello.py /usr/local/bin
cp: cannot create regular file '/usr/local/bin/hello.py':
Permission denied
```

Но способен сделать это на своем ноутбуке:

```
$ cp hello.py /usr/local/bin/
```

И убедиться, что программа найдена:

```
$ which hello.py
/usr/local/bin/hello.py
```

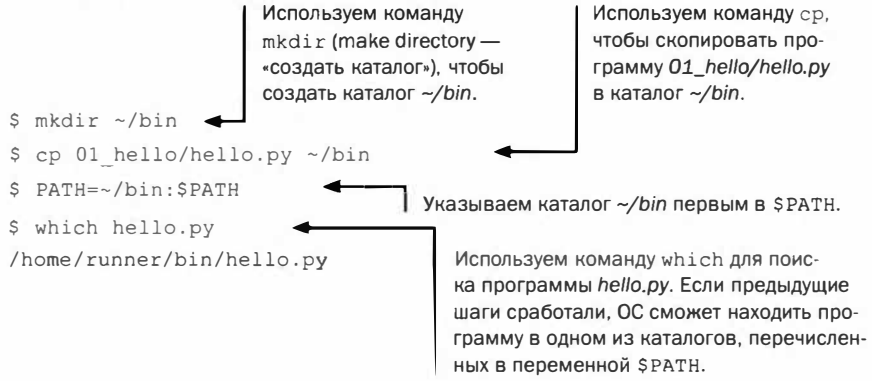
Теперь я могу выполнить ее из любого каталога на своем компьютере:

```
$ hello.py
Hello, World!
```

1.6.1. Настройка переменной $PATH

Бывают ситуации, не позволяющие устанавливать программы в каталоги, указанные в `$PATH`, как при работе на платформе Repl.it. Решение проблемы — изменить значение переменной `$PATH`, добавив каталог, в который вы сможете поместить программы. Например, я часто создаю каталог *bin* в своем домашнем каталоге и ссылаюсь на него с помощью тильды (~).

На большинстве компьютеров значение `~/bin` равно «каталог *bin* в моем домашнем каталоге». Также часто можно увидеть значение `$HOME/bin`, где `$HOME` — это имя вашего домашнего каталога. Ниже показано, как я создаю такой каталог на платформе Repl.it, копирую в него программу, а затем добавляю его в качестве значения переменной `$PATH`:



Теперь я могу перейти в любой каталог:

```
$ pwd
/home/runner/tinypythonprojects
```

и запустить программу:

```
$ hello.py
Hello, World!
```

Хотя использование символов шебанга и превращение файлов в исполняемые рискует показаться трудоемкой задачей, не стоит забывать о важном преимуществе, которое вы получаете. Так вы можете создавать программу Python, устанавливать ее на свой или любой другой компьютер и запускать так же, как и любую другую программу.

1.7. Аргументы и инструкции

На протяжении всей книги я привожу диаграммы для визуализации входных и выходных данных программ, которые мы пишем. Если создать такую для нашей программы, мы увидим, что она не принимает данные и всегда выдает один и тот же результат — текст "Hello, World!" (см. рис. 1.2).

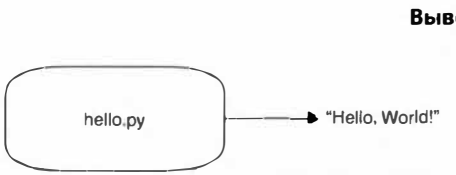


Рис. 1.2. Диаграмма, представляющая нашу программу hello.py, не принимающую входные данные

Согласитесь, довольно скучно, когда программа только и делает, что монотонно твердит: "Hello, World!". Было бы неплохо, если бы она умела здороваться не только с миром, но и, например, со всей вселенной. Мы можем реализовать это, изменив код следующим образом:

```
print('Hello, Universe')
```

Однако в таком случае нам придется менять код каждый раз, когда потребуется вывести другое имя. Гораздо лучше изменить *поведение* программы, а не менять постоянно *саму программу*.

Для этого нам нужно найти ту часть программы, которая должна меняться, например имя в приветствии, и предоставить программе данное значение в качестве *аргумента*. То есть нам нужно, чтобы программа работала так:

```
$ ./hello.py Terra
Hello, Terra!
```

Однако как человек, запустивший нашу программу, сможет узнать о необходимости ввода имени? *Программа сама предоставит пользователю инструкции!* Большинство консольных программ поддерживает такие аргументы, как `-h` и `--help`, позволяющие выводить инструкции о том, как ими пользоваться. Необходимо, чтобы наша программа выводила примерно следующее:

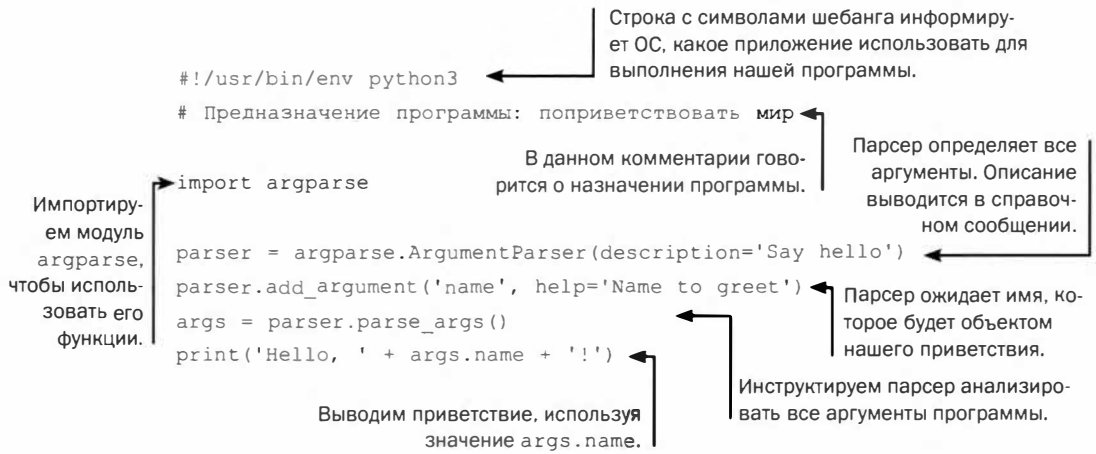
```
$ ./hello.py -h
usage: hello.py [-h] name

Say hello

positional arguments:
  name        Name to greet
```

Обратите внимание, что `name` — это позиционный аргумент.

```
optional arguments:
-h, --help  show this help message and exit
```

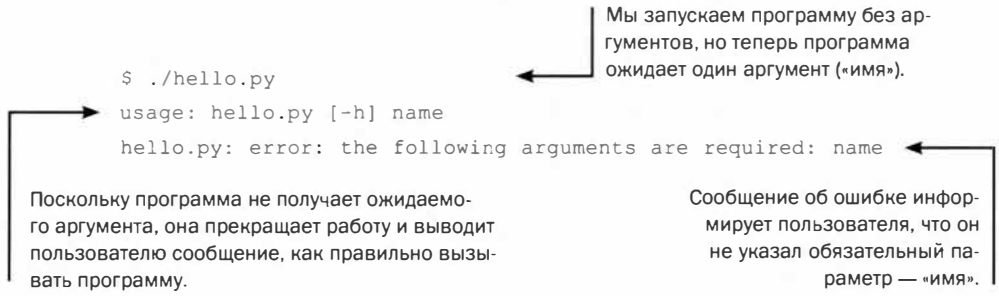
Для этого нам понадобится модуль `argparse`. Модули — это отдельные файлы с кодом, которые можно использовать в программах. Мы вправе создавать собственные модули, чтобы делиться ими с другими людьми. В Python можно использовать сотни, нет, тысячи модулей, и это одна из причин, почему так интересно изучать данный язык.

Модуль `argparse` «анализирует» «аргументы» программы. Чтобы использовать его, измените код так, как показано ниже. Я рекомендую вводить весь код самостоятельно, а не копировать из файлов примеров.



На рис. 1.3 показана диаграмма нашей программы.

Теперь, если вы запустите программу обычным способом, она выведет ошибку и справочную информацию:



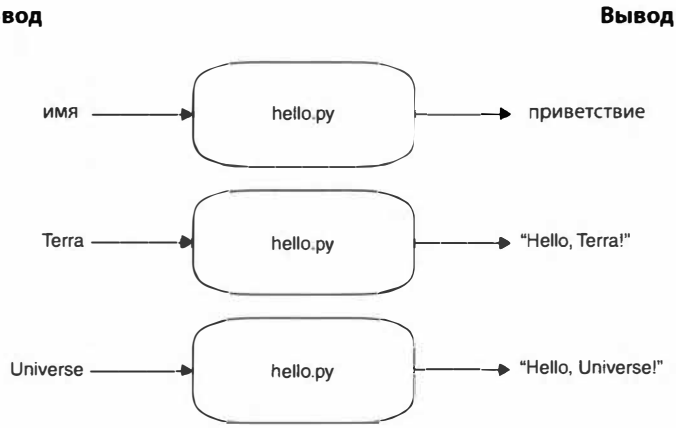


Рис. 1.3. Теперь диаграмма отражает, что программа принимает аргумент и создает сообщение на основе полученного значения

Мы изменили программу так, что для ее запуска необходимо имя, в противном случае она не заработает. Здорово! Давайте передадим ей имя для приветствия:

```
$ ./hello.py Universe
Hello, Universe!
```

Запустите программу с аргументом `-h` или `--help` и убедитесь, что документация выводится корректно.

Теперь программа работает великолепно и обучает пользователя правильному запуску, а все потому, что мы добавили несколько строк с помощью модуля `argparse`. Согласитесь, весомое улучшение.

1.8. Опциональные аргументы

Предположим, мы хотим запустить программу как раньше, без аргументов, и вывести "Hello, World!". Мы можем сделать аргумент `name` необязательным, изменив его имя на `--name`:

```
#!/usr/bin/env python3
# Предназначение программы: поприветствовать мир

import argparse

parser = argparse.ArgumentParser(description='Say hello')
```

```
parser.add_argument('-n', '--name', metavar='name',
                    default='World', help='Name to greet')
args = parser.parse_args()
print('Hello, ' + args.name + '!')
```

Единственное изменение в этой программе — добавление коротких и длинных имен параметров `-n` и `--name`. Мы также добавили дефолтное значение. `metavar` используется для описания аргумента.

Теперь мы можем запустить программу как раньше:

```
$ ./hello.py
Hello, World!
```

Или использовать вариант с параметром `--name`:

```
$ ./hello.py --name Terra
Hello, Terra!
```

Инструкции тоже изменились:

```
$ ./hello.py -h
usage: hello.py [-h] [-n NAME]

Say hello

optional arguments:
-h, --help            show this help message and exit
-n name, --name name  Name to greet
```

Аргумент теперь необязателен и больше не является позиционным. Обычно используются как короткие, так и длинные имена, упрощающие ввод параметров. Здесь применяется параметр `metavar` с «именем», описывающим, каким должно быть значение.

На рис. 1.4 показана диаграмма, иллюстрирующая нашу программу.

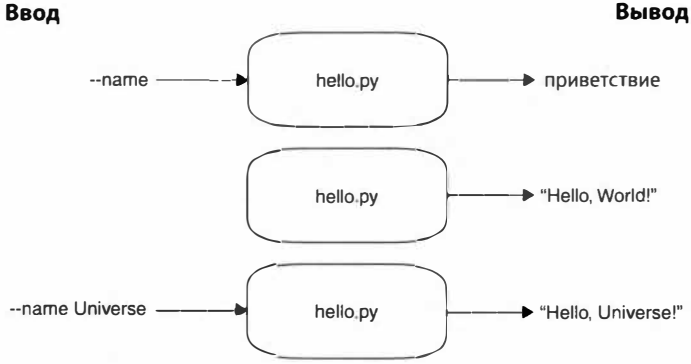


Рис. 1.4. Вводить имя теперь необязательно. Программа отобразит указанное имя или будет использовать дефолтное значение, если пользователь ничего не ввел

Наша программа стала гибкой. Она может выводить приветствие с дефолтным именем при запуске без аргументов или с именем, указанным пользователем. Помните, что параметры, начинающиеся с дефиса, *опциональны*, их необязательно указывать. Кроме того, они могут содержать дефолтные значения. Параметры, которые *не* начинаются с дефиса, являются *позиционными* и обычно обязательны, поэтому они не имеют значений по умолчанию (табл. 1.1).

Таблица 1.1. Два типа параметров командной строки

Тип	Пример	Обязателен	Значение по умолчанию
Позиционный	`name`	Да	Нет
Опциональный	`-n` (короткое имя), `--name` (длинное имя)	Нет	Да

1.9. Выполнение тестов

Давайте снова запустим тесты, чтобы проверить программу:

```
$ make test
pytest -xv test.py
================== test session starts ====================
...
collected 5 items

test.py:: test_exists PASSED                              [20%]
test.py:: test_runnable PASSED                            [40%]
test.py:: test_executable PASSED                          [60%]
test.py:: test_usage PASSED                               [80%]
test.py:: test_input PASSED                               [100%]

==================== 5 passed in 0.38s ====================
```

Ничего себе, мы прошли все тесты! Для меня это всегда волнительный момент, когда программы проходят все тесты, даже если я сам их писал. Чуть раньше у нас был печальный опыт. Мы провалили тесты использования и ввода. Благодаря модулю `argparse` нам удалось справиться с обеими проблемами, поскольку теперь программа способна принимать аргументы при запуске, а также содержит инструкции, как правильно ее запускать.

1.10. Добавление функции main()

Сейчас наша программа работает прекрасно, однако она не во всем соответствует стандартам и ожиданиям сообщества. Дело в том, что довольно часто компьютерные программы — не только написанные на Python — начинаются с так называемой функции `main()`. Большинство программ на Python определяет функцию `main()`, которая по традиции вызывается в конце программы. Выглядит это так:

```
#!/usr/bin/env python3
# Предназначение программы: поприветствовать мир

import argparse


def main():
    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('-n', '--name', metavar='name',
                        default='World', help='Name to greet')
    args = parser.parse_args()
    print('Hello, ' + args.name + '!')


if __name__ == '__main__':
    main()
```

Слово `def` отвечает за определение функции, имеющей в данном случае имя `main()`. Пустые скобки показывают, что эта функция не принимает аргументов.

У каждой программы или модуля в Python есть имя, к которому можно получить доступ с помощью переменной `__name__`. Когда программа выполняется, переменной `__name__` присваивается значение `__main__`.*

Если это так (истина), вызывается функция `main()`.

По мере усложнения наших программ мы начнем создавать больше функций. Программисты на Python подходят к этому по-разному, но в данной книге я всегда буду использовать функцию `main()`, чтобы быть последовательным. Нам предстоит помещать внутрь указанной функции основную часть нашей программы.

1.11. Добавление функции get_args()

Лично я предпочитаю помещать весь код `argparse` в отдельное место, которое всегда называю `get_args()`. В моем понимании получение и проверка аргументов — это единая концепция, представляющая собой

* Для дополнительной информации см. документацию Python: https://docs.python.org/3/library/__main__.html.

отдельную часть программы. Для некоторых программ данная процедура может занимать довольно много времени.

Я всегда указываю функцию `get_args()` первой для удобства чтения исходного кода. `main()` обычно идет следом. Разумеется, вы можете структурировать собственные программы по своему усмотрению.

Вот как программа выглядит сейчас:

```
#!/usr/bin/env python3
# Предназначение программы: поприветствовать мир

import argparse

def get_args():
    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('-n', '--name', metavar='name',
                        default='World', help='Name to greet')
    return parser.parse_args()

def main():
    args = get_args()
    print('Hello, ' + args.name + '!')

if __name__ == '__main__':
    main()
```

Функция `get_args()` предназначена для получения аргументов. Весь код `argparse` теперь расположен здесь.

Функция `main()` стала существенно короче.

Нам необходим оператор `return`, чтобы отправить результаты анализа аргументов обратно функции `main()`.

Вызываем функцию `get_args()`, чтобы получить проанализированные аргументы. Если есть проблема с аргументами или если пользователь указал флаг `--help`, программа никогда не доберется до этой точки, потому что код `argparse` приводит к ее завершению. Если же интерпретатор добрался так далеко, значит, с входными данными все в порядке.

В работе программы ничего не изменилось. Мы лишь структурировали код для группировки концепций — код, который относится к `argparse`, теперь располагается в функции `get_args()`, а все остальные операторы — в функции `main()`. Чтобы убедиться в корректности кода, протестируйте программу!

1.11.1. Проверка стиля и отсутствия ошибок

Наша программа работает прекрасно. Мы можем использовать такие инструменты, как Flake8 и Pylint, чтобы проверить, есть ли в ней

какие-то проблемы. Упомянутые инструменты называются *линтерами*. Их задача — предлагать способы улучшения программы. Если вы еще не установили их, то можете использовать модуль `pip` и выполнить следующую команду:

```
$ python3 -m pip install flake8 pylint
```

Программа Flake8 предлагает разделять определения функций двумя пустыми строками:

```
$ flake8 hello.py
hello.py:6:1: E302 expected 2 blank lines, found 1
hello.py:12:1: E302 expected 2 blank lines, found 1
hello.py:16:1: E305 expected 2 blank lines after class or function
definition, found 1
```

А Pylint сообщает, что у функций отсутствует документация (docstring — документационные строки):

```
$ pylint hello.py
************* Module hello
hello.py:1:0: C0114: Missing module docstring
      (missing-module-docstring)
hello.py:6:0: C0116: Missing function or method docstring
      (missing-function-docstring)
hello.py:12:0: C0116: Missing function or method docstring
      (missing-function-docstring)

-------------------------------------------------------------------
Your code has been rated at 7.00/10 (previous run: -10.00/10,
                                     +17.00)
```

Документационные строки указываются сразу после определения функции. Обычно в функции используется несколько документационных строк, поэтому программисты зачастую заключают их в тройные кавычки Python (одинарные или двойные), формируя многострочные комментарии. Ниже показана программа с документационными строками. Еще я отформатировал код и исправил проблемы с пробелами с помощью YAPF. Если этот инструмент вам не нравится, вы можете использовать Black или любое другое ПО по вкусу.

```
#!/usr/bin/env python3
"""
Разработчик:     Кен Юэнс-Кларк <kyclark@gmail.com>
Предназначение: Поприветствовать мир
"""

import argparse

# ---------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('-n', '--name', default='World',
                        help='Name to greet')
    return parser.parse_args()

# --------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    print('Hello, ' + args.name + '!')

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Многострочный документационный комментарий в тройных кавычках с описанием программы. Обычно документационную строку пишут сразу после кода с символами шебанга, чтобы задокументировать общее назначение функции. Я люблю указывать свое имя, адрес электронной почты и предназначение программы, чтобы человек, использующий ее, знал, для чего она, кем написана и как со мной связаться, если возникнут проблемы.

Длинный горизонтальный комментарий в виде «строки» помогает разделять функции. Можно не использовать его, если он вам не нравится.

Документационная строка для функции `get_args()`. Я использую тройные кавычки даже в однострочных комментариях, поскольку так документационные строки заметнее.

Функция `main()` — это лишь начало программы, поэтому в документационной строке особо нечего сказать. Забавно (чуточку) написать что-то вроде «Да грянет джаз!», но решать вам.

Чтобы узнать, как в командной строке использовать YAPF и Black, запустите их с аргументом `-h` или `--help` и прочтите документацию. Если вы пользуетесь IDE, например VS Code или PyCharm, или интерфейсом Repl.it, в них вы найдете команды для форматирования кода.

1.12. Тестирование программы hello.py

Мы внесли много изменений в программу. Теперь давайте убедимся, что она по-прежнему работает правильно. Вновь запустим наш тест.

Тестировать программы вы будете сотни раз, поэтому я предлагаю воспользоваться упрощенным методом. В каждый каталог я поместил файл Makefile, содержимое которого выглядит следующим образом:

```
$ cat Makefile
.PHONY: test

test:
    pytest -xv test.py
```

Если программа make установлена, можно выполнить команду make test из каталога *01_hello*. Программа make обнаружит Makefile в текущем рабочем каталоге, а затем найдет в данном файле сценарий test. Согласно ему команда для выполнения теста — pytest -xv test.py, ее программа make и запустит:

```
$ make test
pytest -xv test.py

======================== test session starts ========================
...
collected 5 items

test.py:: test_exists PASSED                                    [20%]
test.py:: test_runnable PASSED                                  [40%]
test.py:: test_executable PASSED                                [60%]
test.py:: test_usage PASSED                                     [80%]
test.py:: test_input PASSE                                      [100%]

======================== 5 passed in 0.75s ==========================
```

Если программа make не установлена, сделайте это и изучите, как использовать файлы Makefile для выполнения сложных наборов команд. Если же вы не хотите пользоваться программой make, вы можете вручную выполнять команду pytest -xv test.py. В обоих случаях результат будет одинаковым.

Важно отметить, что мы использовали тесты, чтобы проверить, что наша программа работает так, как задумывалось. Разрабатывая программы, вы можете пробовать разные решения. Тесты позволят проверить измененный код (так называемый рефакторинг) и убедиться, что все работает корректно.

1.13. Создание программы на основе файла new.py

`argparse` — это популярный стандартный модуль, устанавливаемый вместе с Python. Он позволяет сэкономить много времени при анализе и проверке аргументов программ. Мы будем пользоваться модулем `argparse` во всех программах из этой книги, преобразовывать с его помощью текст в числа, проверять и открывать файлы и т. д. и т. п. Вариантов так много, что я написал приложение *new.py*, помогающее создавать Python-программы с модулем `argparse`.

Файл *new.py* вы найдете в каталоге *bin* репозитория GitHub. Предлагаю использовать его для всех новых программ. Например, можно создать новую версию программы *hello.py* с помощью файла *new.py*. Перейдите в корень вашего репозитория и выполните следующее:

```
$ bin/new.py 01_hello/hello.py
"01_hello/hello.py" exists.  Overwrite? [yN] n
Will not overwrite. Bye!
```

Программа *new.py* не перезапишет исходный файл, если, конечно, вы не попросите ее об этом, в связи с чем вы можете использовать ее без страха потерять свои данные. Итак, создайте приложение с другим именем:

```
$ bin/new.py 01_hello/hello2.py
Done, see new script "01_hello/hello2.py."
```

Теперь запустите программу:

```
$01_hello/hello2.py
usage: hello2.py [-h] [-a str] [-i int] [-f FILE] [-o] str
hello2.py: error: the following arguments are required: str
```

Взглянем на исходный код новой программы:

Строка с шебангом содержит инструкцию по поиску программы `python3` с помощью `env`.

```
#!/usr/bin/env python3
"""
Разработчик: Кен Юэнс-Кларк <kyclark@gmail.com>
Дата: 2020-02-28
Предназначение: Раскачивание Касбы
"""
```

Данная документационная строка описывает программу.

```
import argparse
import os
import sys

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Rock the Casbah',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('positional',
                        metavar='str',
                        help='A positional argument')

    parser.add_argument('-a',
                        '--arg',
                        help='A named string argument',
                        metavar='str',
                        type=str,
                        default='')

    parser.add_argument('-i',
                        '--int',
                        help='A named integer argument',
                        metavar='int',
                        type=int,
                        default=0)

    parser.add_argument('-f',
                        '--file',
                        help='A readable file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        default=None)

    parser.add_argument('-o',
                        '--on',
                        help='A boolean flag',
                        action='store_true')
```

В этих строках импортируются различные модули, необходимые для работы программы.

Функция `get_args()` отвечает за синтаксический анализ (парсинг) и проверку аргументов.

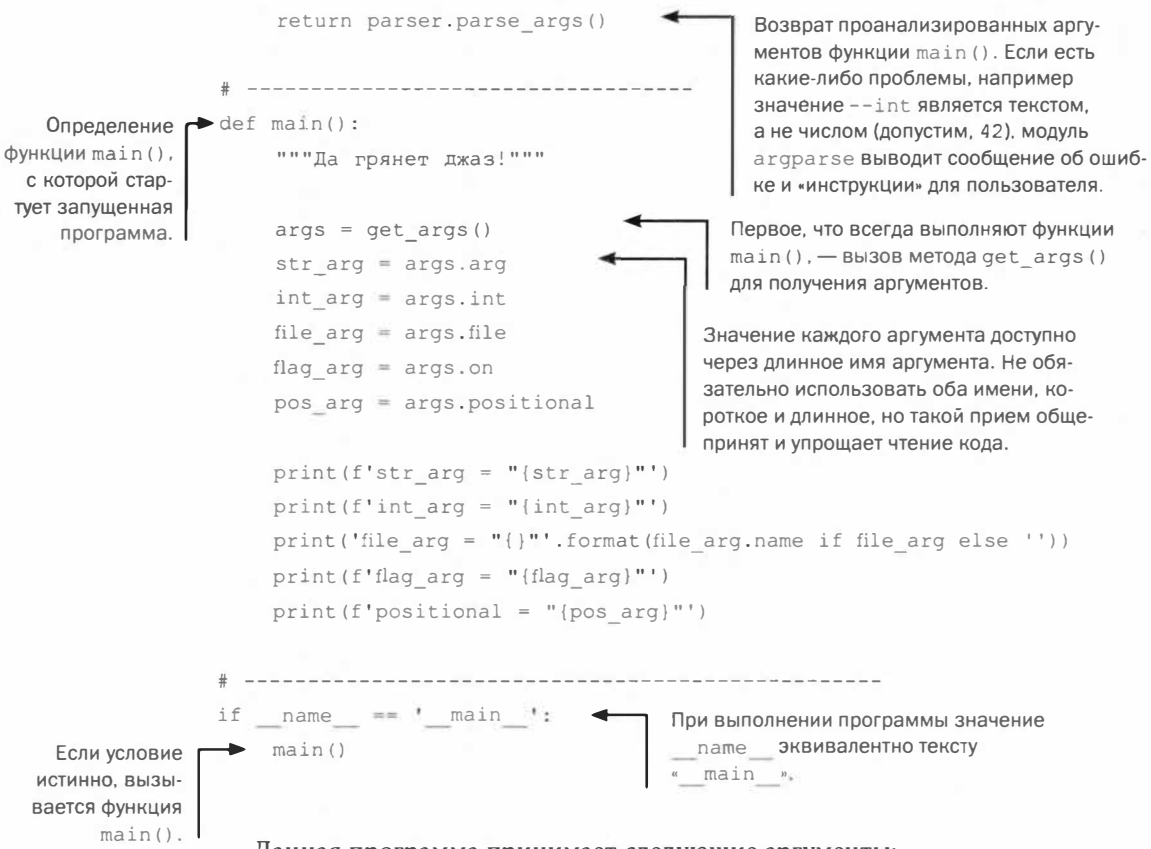
Определение позиционного аргумента в духе первой версии программы *hello.py*, в которой использовался аргумент `name`.

Определение опционального аргумента в духе использования параметра `--name`.

Определение опционального аргумента, который должен быть целым числом.

Определение опционального аргумента, который должен быть представлен файлом.

Определение «флага», который «включен» при наличии или «выключен» при отсутствии. Эту тему обсудим позже.



Данная программа принимает следующие аргументы:

- один позиционный аргумент типа `str`. «Позиционный» означает, что ему не предшествует флаг имени и ему присваивается значение согласно позиции в команде;
- встроенный флаг `-h` или `--help`, который обращается к модулю `argparse`, чтобы вывести инструкции для пользователя;
- строковый параметр `-a` или `--arg`;
- аргумент именованного параметра `-i` или `--int`;
- файловый параметр `-f` или `--file`;
- логический флаг (выкл./вкл.) `-o` или `--on`.

Проанализировав список, мы понимаем, что код в файле *new.py* выполнил следующее:

- создал Python-программу *hello2.py*;
- на основе шаблона сформировал рабочий код программы со строками документации, функцией `main()` с кодом программы, функцией

`get_args()` для анализа и документирования различных типов аргументов и с операторами запуска функции `main()`;
— сделал новую программу исполняемой, чтобы ее можно было запускать как `./hello2.py`

В результате получилась программа, готовая к немедленному исполнению и содержащая документацию с инструктажем для пользователя. Создав новую программу с помощью файла *new.py*, откройте ее в редакторе и измените имена и типы аргументов в соответствии с предназначением вашей программы. Например, прочитав главу 2, вы сможете удалить все аргументы, кроме позиционного, переименовав его с `'positional'` на что-то вроде `'word'` (потому что аргументом будет слово).

Обратите внимание, что вы можете использовать собственные значения переменных `name` и `email` в файле *new.py*, создав файл с именем *new.py* (не упустите из вида точку в начале имени!) в домашнем каталоге. В моем случае используются следующие значения:

```
$ cat ~/.new.py
name=Ken Youens-Clark
email=kyclark@gmail.com
```

1.14. Использование файла *template.py* вместо *new.py*

Если вам не хочется использовать файл *new.py*, найдите пример нашей программы в файле *template/template.py*, который можно скопировать. В главе 2 вам понадобится создать программу *02_crowsnest/crowsnest.py*.

Вы можете сделать это с помощью программы *new.py*, расположенной в корневом каталоге репозитория:

```
$ bin/new.py 02_crowsnest/crowsnest.py
```

или выполнив команду `cp` (copy — копировать), чтобы скопировать шаблон в новую программу:

```
$ cp template/template.py 02_crowsnest/crowsnest.py
```

Смысл в том, что вам не придется писать каждую программу с нуля. Я думаю, гораздо проще начать с готовой работающей программы и изменить ее код.

Примечание. Вы можете скопировать файл *new.py* в каталог *~/bin*, а затем обращаться к этому файлу из любого каталога, в котором требуется создать новую программу.

Обязательно прочитайте приложение к книге — в нем приведено много примеров программ с модулем `argparse`. Вы можете воспользоваться ими, чтобы эффективнее работать с упражнениями.

РЕЗЮМЕ

- Программа на языке Python — это простой текст, сохраненный в файл. Для интерпретации и выполнения такого файла необходима программа `python3`.
- Можно сделать программу исполняемой и скопировать ее в папку `$PATH`, чтобы запускать ее точно так же, как и любую другую программу на вашем компьютере. Обязательно укажите строку с символами шебанга для поиска корректной программы `python3` с помощью `env`.
- Модуль `argparse` помогает документировать и анализировать аргументы программы. Можно проверить типы и количество аргументов, позиционных, опциональных и/или флагов. Код будет сгенерирован автоматически.
- Мы используем инструмент `pytest` для запуска программ *test.py* в каждом упражнении. Можно выполнить команду `make test` для запуска `pytest -xv test.py` либо запустить эту команду вручную.
- Следует постоянно тестировать программы, чтобы убедиться в их корректной работе.
- Инструменты типа YAPF и Black автоматически форматируют код программы в соответствии со стандартами сообщества, упрощая его чтение и отладку.
- Линтеры кода, такие как Pylint и Flake8, помогают исправлять как программные, так и стилистические недочеты.
- Можно использовать файл *new.py* для создания новых Python-программ с модулем `argparse`.

Глава 2

Воронье гнездо: работа со строками

Эй ты, прохиндей! Полезай в воронье гнездо и заступай на вахту. Смекаешь, о чем я, пустоголовый?! Ах ты, крыса сухопутная!

Наблюдатель в вороньем гнезде

Итак, вы наблюдатель в вороньем гнезде — открытой бочке на топе мачты парусного судна. Ваша задача — отслеживать все интересное и опасное, например гражданские суда — потенциальные цели грабежа, или айсберги, столкновений с которыми следует избегать. Когда вы видите животное, похожее на нарвала, вы кричите: «Капитан, *нарвал* по левому борту!» А если заметили осьминога, то: «Капитан, *осьминог* по левому борту!» (Для нашего упражнения допустим, что все объекты находятся «по левому борту».)

Каждая глава данной книги содержит задачу по программированию, с которой вы должны справиться самостоятельно. Я объясню ключевые идеи, способные помочь вам в решении подобных задач, а также

подскажу, как тестировать свои приложения. Вам понадобится локальная копия моего репозитория Git (см. инструкции во введении). Кроме того, вам обязательно нужно создавать каждую программу в каталоге, соответствующем номеру главы. Например, программа из этой главы должна располагаться в каталоге *02_crowsnest*, в котором также находятся и соответствующие тесты.

В этой главе мы начнем работать со строками. К концу главы вы:

— создадите программу, принимающую позиционный аргумент и содержащую документацию для пользователя;
— научитесь генерировать вывод программы в соответствии с входными данными;
— проведете тестирование.

Программа называется *crowsnest.py*. Она принимает один позиционный аргумент и выводит данный аргумент внутри строки Ahoy вместе с артиклем `a` или `an` в зависимости от того, с какой буквы начинается аргумент: согласной или гласной*.

То есть в случае с нарвалом программа выведет следующий результат:

```
$ ./crowsnest.py narwhal
Ahoy, Captain, a narwhal off the larboard bow!
```

А в случае с осьминогом вот что:

```
$ ./crowsnest.py octopus
Ahoy, Captain, an octopus off the larboard bow!
```

Таким образом, вам нужно написать программу, которая принимает ввод от пользователя в командной строке, выбирает правильный артикль (`a` или `an`) и выводит соответствующую строку текста, поместив эти два значения в фразу `"Ahoy..."`.

* Задача построена на правиле употребления неопределенного артикля в английском языке. *Прим. ред.*

2.1. Создание программы

Итак, вам не терпится написать программу! Но подождите минутку, уймите свои руки. Необходимо обсудить, как проверять программу на работоспособность и с чего начать ее создание.

2.1.1. Как тестировать программы

«Лучший учитель — неудачи наши».
— Йода

В репозиторий я добавил тесты, которые помогут вам в написании программы. Еще не принимаясь за код, запустите тест, чтобы провалить его:

```
$ cd 02_crowsnest
$ make test
```

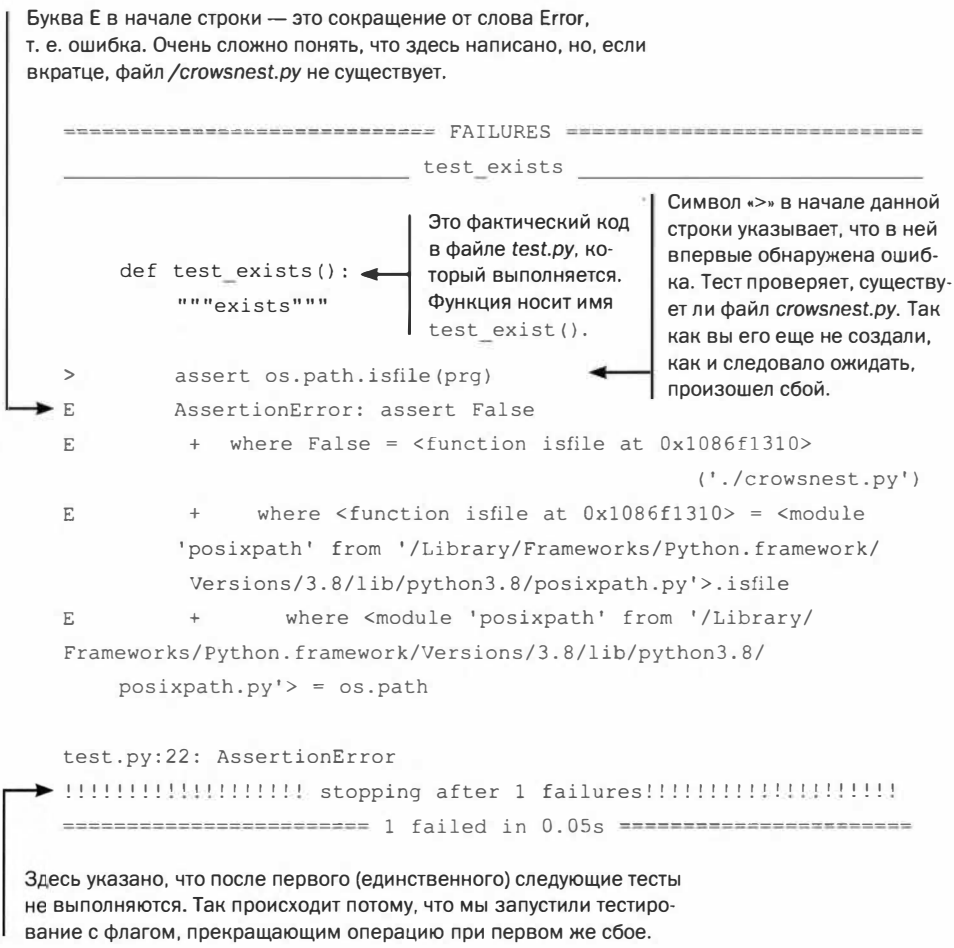
Вместо `make test` вы также можете выполнить прямую команду `pytest -xv test.py`. В выводе вы увидите следующую строку:

```
$ pytest -xv test.py
====================== test session starts ======================
...
collected 6 items

test.py:: test_exists FAILED                                  [16%]
```

Итак, данный тест провален. Последующие тоже не выполняются из-за флага `-x` в команде `pytest`.

Вы можете увидеть длинное сообщение о том, что необходимый файл *crowsnest.py* не обнаружен. Научиться анализировать результаты теста — важный навык, требующий немало практики, поэтому не расстраивайтесь, если с ходу чего-то не понимаете. В моем терминале (iTerm на Mac) результат выполнения команды `pytest` отформатирован разным цветом и полужирным шрифтом, таким образом обращая мое внимание на ключевые ошибки. Обычно подобный текст выделен полужирным шрифтом красного цвета, но настройки вашего терминала могут быть иными.

Давайте рассмотрим вывод. На первый взгляд ничего не понятно, но вскоре вы научитесь анализировать его и находить ошибки.




```
============================= FAILURES =============================
_____________________________ test_exists ______________________________

    def test_exists():
        """exists"""
>       assert os.path.isfile(prg)
E       AssertionError: assert False
E        +  where False = <function isfile at 0x1086f1310>
                                                    ('./crowsnest.py')
E        +    where <function isfile at 0x1086f1310> = <module
        'posixpath' from '/Library/Frameworks/Python.framework/
         Versions/3.8/lib/python3.8/posixpath.py'>.isfile
E        +      where <module 'posixpath' from '/Library/
Frameworks/Python.framework/Versions/3.8/lib/python3.8/
    posixpath.py'> = os.path

test.py:22: AssertionError
!!!!!!!!!!!!!!!!!!!! stopping after 1 failures!!!!!!!!!!!!!!!!!!!!
======================== 1 failed in 0.05s =========================
```

Итак, первый тест проверяет наличие необходимого файла, так что давайте его создадим!

2.1.2. Создание программ с помощью файла new.py

Чтобы пройти первый тест, вам нужно создать файл *crowsnest.py* в каталоге *02_crowsnest* с файлом *test.py*. Несмотря на то что писать код с нуля — вполне обычная практика, я предлагаю воспользоваться приложением *new.py* для вывода некоторого полезного шаблонного кода, который понадобится вам во всех упражнениях.

В корневом каталоге репозитория выполните показанную ниже команду, чтобы создать новую программу:

```
$ bin/new.py 02_crowsnest/crowsnest.py
Done, see new script "02_crowsnest/crowsnest.py."
```

Если вы не хотите использовать программу *new.py*, вы можете скопировать код из файла *template/template.py*:

```
$ cp template/template.py 02_crowsnest/crowsnest.py
```

Теперь у вас есть каркас рабочей программы, принимающей аргументы командной строки. Если вы запустите файл *crowsnest.py* без аргументов, он выведет короткую инструкцию, показанную ниже (обратите внимание, что usage — это первое слово в выводе):

```
$ ./crowsnest.py
usage: crowsnest.py [-h] [-a str] [-i int] [-f FILE] [-o] str
crowsnest.py: error: the following arguments are required: str
```

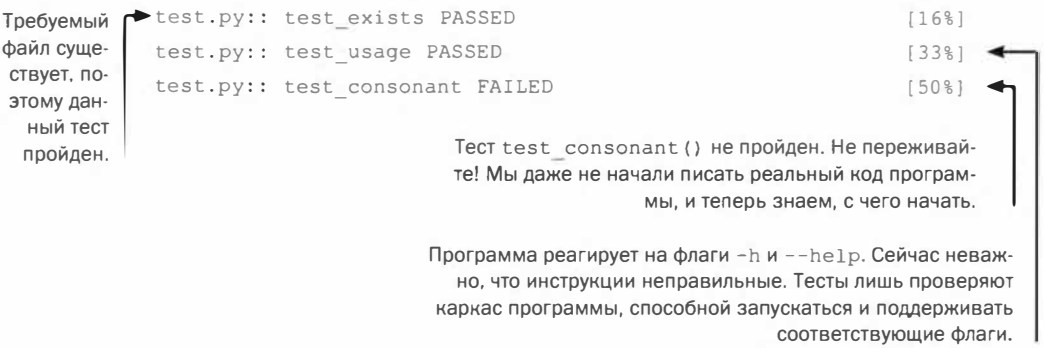
Теперь запустите файл следующим образом: `./crowsnest.py --help`. Вы увидите более длинное сообщение с инструкциями для пользователя.

Примечание. Я использую в коде демонстрационные дефолтные параметры, добавленные с помощью файла *new.py*. Вам нужно изменить параметры в соответствии с вашей программой.

2.1.3. Пишите, тестируйте, повторяйте

Вы только что создали программу, способную пройти первый тест. Полагаю, что вам по силам написать небольшой цикл, состоящий из пары строк кода, а затем запустить или протестировать программу. Выполним тестирование еще раз:

```
$ make test
pytest -xv test.py
========================= test session starts ==========================
...
collected 6 items
```



Как результат, программа, созданная с помощью файла *new.py*, успешно проходит первые *два* теста.

1. Программа существует? Да, вы только что создали ее.
2. Выводит ли программа инструкции, когда вы запрашиваете их? Да, вы запустили ее без аргументов и с флагом `--help`, и убедились, что она выводит справочные инструкции.

Теперь у вас есть рабочая программа, принимающая некоторые аргументы (но не те, что нужно). Осталось сделать так, чтобы она принимала значение `narwhal` или `octopus`, которое необходимо объявить. Для этого воспользуемся аргументами командной строки.

2.1.4. Определение аргументов

Рисунок 2.1 может напугать вас входными (*параметрами*) и выходными данными. Я привожу подобные диаграммы в каждой главе книги, чтобы показать, как взаимодействуют код и данные. В текущей программе вводом является некое *слово*, а выводом — фраза, содержащая это *слово* с правильным *артиклем*.

Необходимо изменить фрагмент кода, в котором программа получает аргументы, — в нашем случае функцию с соответствующим именем `get_args()`. Данная функция использует модуль `argparse` для анализа аргументов командной строки. Наша программа должна принимать один позиционный аргумент. Если вы не знаете, что значит позиционный аргумент, обязательно прочитайте Приложение, в частности раздел A.4.1.

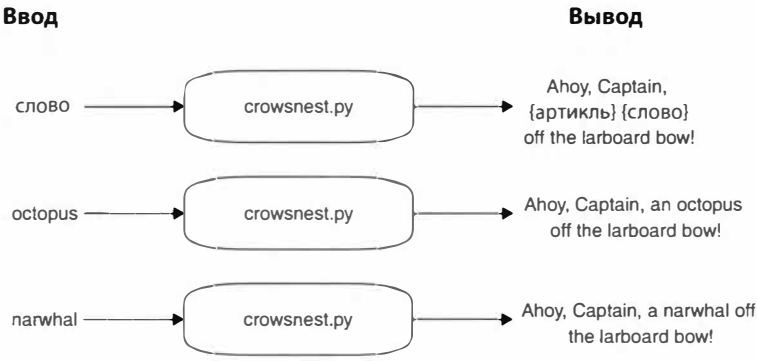


Рис. 2.1. Входные данные для программы — это слово (`word`), а выходные данные — слово (`word`) плюс соответствующий ему артикль (`article`) (и кое-что еще).

Функция `get_args()` согласно шаблону присваивает первому аргументу имя `positional`. Помните, что позиционные аргументы определяются их позициями, а их имена не начинаются с дефиса. Вы можете удалить все аргументы, кроме позиционного аргумента `word`. Измените код функции `get_args()` в программе так, чтобы при запуске файла выводилось следующее.

```
$ ./crowsnest.py
usage: crowsnest.py [-h] word
crowsnest.py: error: the following arguments are required: word
```

Аналогично должна выводиться более подробная документация с помощью флага `-h` или `--help`:

```
$ ./crowsnest.py -h
usage: crowsnest.py [-h] word

Crow's Nest -- choose the correct article

positional arguments:
  word        A word

optional arguments:
-h, --help  show this help message and exit
```

Вам нужно определить параметр `word`. Обратите внимание, что он используется как позиционный аргумент.

Флаги `-h` и `--help` создаются автоматически модулем `argparse`. Вы не можете использовать их опционально. Они применяются для создания документации к вашей программе.

Добейтесь, чтобы ваша программа работала точно также, прежде чем читать дальше!

Когда она выведет корректные инструкции, вы можете получить аргумент `word` внутри функции `main`. Измените код так, чтобы выводилось значение `word`:

```
def main():
    args = get_args()
    word = args.word
    print(word)
```

Затем проверьте, работает ли программа:

```
$ ./crowsnest.py narwhal
narwhal
```

А теперь снова проведите тестирование. Программа должна пройти два теста и провалить третий. Давайте разберемся:

Неважно, понимаете ли вы эту строку, но если вкратце, функция `getoutput()` запускает программу со значением `word`. В данной главе мы знакомимся с f-строками. Результат выполнения программы будет передан переменной `out`, используемой для проверки, сгенерировала ли программа корректный вывод для предоставленного слова (значения `word`). Не волнуйтесь, если пока не понимаете код этой функции.

```
============================ FAILURES ============================
_____________________________ test_consonant ______________________________

    def test_consonant():
        """brigantine -> a brigantine"""

        for word in consonant_words:
            out = getoutput(f' {prg} {word}')
>           assert out.strip() == template.format('a', word)
E           AssertionError: assert 'brigantine' == 'Ahoy, Captai...
                                                    larboard bow!'
E             - brigantine
E             + Ahoy, Captain, a brigantine off the larboard bow!
```

Строка, начинающаяся с символа >, указывает на код, вызвавший ошибку. Вывод программы сравнивается с ожидаемой строкой. Поскольку совпадений не обнаружено, выбрасывается исключение.

Данная строка начинается с символа `E`, что указывает на ошибку.

Строка, начинающаяся с символа «плюс» (+), выводит результаты теста: "Ahoy, Captain, a brigantine off the larboard bow!"

Строка, начинающаяся с символа «дефис» (–), содержит значение, полученное тестом. Так как он был выполнен с аргументом `brigantine`, то и вернул это слово в строке.

Итак, нам нужно подставить значение `word` в фразу `"Ahoy..."`. Как это сделать?

2.1.5. Конкатенация строк

Совмещение строк называется *конкатенацией*, или *объединением*. В качестве примера я введу некоторый код непосредственно в интерпретатор Python. Я хочу, чтобы вы набрали его вручную. Пожалуйста! Вводите все, что ввожу я, так вы сможете самостоятельно выполнять примеры.

Откройте терминал и введите `python3` или `ipython`, чтобы перейти к REPL-интерфейсу (от англ. read-eval-print loop — цикл «чтение — вычисление — вывод»). Благодаря ему Python будет *считывать* каждую строку ввода, *вычислять* ее и *выводить* результаты в *цикле*. Вот как это выглядит в моем случае:

```
$ python3
Python 3.8.1 (v3.8.1:1b293b6006, Dec 182019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

`>>>` — это приглашение, после которого можно ввести код. *Не* вводите сами скобки! Чтобы выйти из REPL-интерфейса, наберите `quit()` или нажмите сочетание клавиш **Ctrl+D** (одновременно нажмите клавишу **Ctrl** и буквенную клавишу **D**).

Примечание. Для взаимодействия с интерпретатором можно использовать инструмент Python IDLE (интегрированная среда разработки и обучения), IPython или Jupyter Notebooks. Для этой книги я выбрал REPL-интерфейс Python 3.

Начнем с присвоения переменной `word` значения `narwhal`. В REPL-интерфейсе введите строку `word = 'narwhal'` и нажмите клавишу **Enter**:

```
>>> word = 'narwhal'
```

Обратите внимание, что можно добавить (или вовсе опустить) сколько угодно пробелов слева и справа от символа =, но по соглашению и для удобства чтения кода (такие инструменты, как Pylint и Flake8,

помогают находить ошибки в коде) следует использовать только один пробел с каждой стороны символа =.

Если вы наберете слово `word` и нажмете клавишу **Enter**, Python выведет текущее значение переменной `word`:

```
>>> word
'narwhal'
```

Теперь введите слово `werd` и нажмите клавишу **Enter**:

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

Внимание! Переменной `werd` не существует, потому что мы не объявили ее ранее. Обращение к неопределенной (необъявленной) переменной вызывает *исключение*, приводящее к сбою программы. Python с удовольствием создаст переменную `werd`, если вы объявите ее и присвоите ей значение.

Необходимо добавить слово между двумя строками текста. Для соединения (конкатенации) строк используется оператор +:

```
>>> 'Ahoy, Captain, a ' + word + ' off the larboard bow!'
'Ahoy, Captain, a narwhal off the larboard bow!'
```

Если вы измените код программы так, чтобы функция `print()` выводила показанную выше строку вместо слова, программа успешно пройдет четыре теста:

```
test.py:: test_exists PASSED                                [16%]
test.py:: test_usage PASSED                                 [33%]
test.py:: test_consonant PASSED                             [50%]
test.py:: test_consonant_upper PASSED                       [66%]
test.py:: test_vowel FAILED                                 [83%]
```

Проанализировав ошибку, вы увидите следующее:

```
E               - Ahoy, Captain, a aviso off the larboard bow!
E               + Ahoy, Captain, an aviso off the larboard bow!
E               ?                  +
```

Мы жестко запрограммировали артикль «a» перед значением переменной `word`, хотя на самом деле необходимо использовать артикль «a» или «an» в зависимости от того, начинается хранимое в переменной `word` слово с гласной или нет. Как это сделать?

2.1.6. Типы переменных

Прежде чем двигаться дальше, давайте вернемся чуть назад и вспомним, что значение нашей переменной `word` является *строкой*. Каждая переменная в Python имеет некий *тип*, описывающий содержащийся в ней тип данных. Поскольку мы помещаем значение переменной `word` в кавычки (`'narwhal'`), интерпретатор понимает, что она содержит *строку*, которую Python представляет с помощью класса `str`. (*Класс* — это набор операторов и функций, которые мы можем использовать в своих программах.)

Функция `type()` позволяет узнать тип данных переменной, указанной в скобках:

```
>>> type(word)
<class 'str'>
```

Значение, заключенное в одинарные (`' '`) или двойные кавычки (`" "`), Python интерпретирует как строку:

```
>>> type("submarine")
<class 'str'>
```

Внимание! Если вы забудете поставить кавычки, Python начнет искать переменную или функцию с указанным именем. Если таковых не существует, интерпретатор выбросит исключение:

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

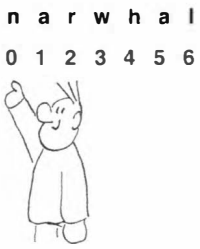
Исключения — это всегда плохо, и мы будем стараться избегать их или, по крайней мере, изящно обрабатывать.

2.1.7. Работа с фрагментом строки

Вернемся к нашей задаче. Необходимо добавить артикль «a» или «an» перед словом, хранящимся в переменной `word`, в зависимости от первой буквы этого слова, — гласная она или согласная.

Мы можем использовать квадратные скобки и числовой *индекс*, чтобы извлечь отдельный символ из строки. Индекс — это порядковая позиция элемента в последовательности. Важно помнить, что индексация начинается с `0`:

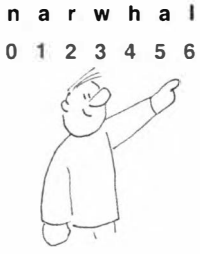
```
>>> word = 'narwhal'
>>> word[0]
'n'
```



Можно индексировать строковый литерал:

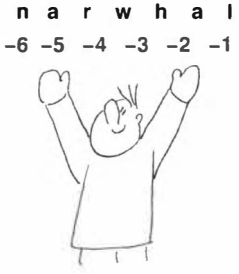
```
>>> 'narwhal'[0]
'n'
```

Поскольку индексация начинается с `0`, последний индекс на *единицу меньше длины* строки, что часто приводит к путанице. Длина (количество символов) строки `narwhal` равна 7, но последний символ расположен в позиции с индексом `6`:



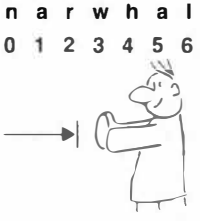

```
>>> word[6]
'l'
```

Можно также использовать отрицательные значения индексов для извлечения в обратном порядке, причем индекс последнего символа равен `-1`:




```
>>> word[-1]
'l'
```

Извлечь диапазон символов позволяют *срезы* `[start: stop]`. Оба индекса среза, и начальный (`start`), и конечный (`stop`), опциональны. Если начальный индекс не указан, срез извлекается, начиная с символа с индексом `0`. Символ, находящийся в позиции конечного индекса, в срез *не включается*:



```
>>> word[:3]
'nar'
```

Если конечный индекс не указан, срез извлекается до конца строки:

```
>>> word[3:]
'whal'
```

В следующей главе вы увидите, что такой же синтаксис используется и для нарезки списков. По сути строка — это своего рода список символов, так что ничего удивительного.

2.1.8. Поиск справочных сведений в REPL-интерфейсе

В классе `str` имеется масса функций для обработки строк, но как узнать каких? Львиная доля программирования заключается в постановке вопросов и поиске ответов. Чаще всего в ответ на вопрос вы услышите RTFM (Read The Following Manual — обратитесь к прилагаемому руководству). Сообществом пользователей Python создано огромное количество документации, доступной по адресу: **https://docs.python.org/3**. Вы будете постоянно обращаться к ней, чтобы вспомнить (или узнать), как использовать те или иные функции. Документация к классу `str` доступна по адресу: **https://docs.python.org/3/library/string.html.**

Я предпочитаю читать руководство прямо в REPL-интерфейсе, используя функцию `help()`:

```
>>> help(str)
```

Для перемещения по документации вверх и вниз используйте клавиши ↑ и ↓ на клавиатуре. Вы также можете нажать клавишу **Пробел** или **F** (в некоторых случаях **Ctrl+F**), чтобы перейти на следующую страницу, или клавишу **B** (в некоторых случаях **Ctrl+B**) для возвращения на предыдущую. Для поиска нажмите клавишу **/**, а затем введите текст поискового запроса. Воспользовавшись клавишей **N** (Next — далее) в процессе поиска, вы перейдете к следующему вхождению поискового запроса. Чтобы выйти из справочной системы, нажмите клавишу **Q** (Quit — выход).

2.1.9. Строковые методы

Теперь, когда мы знаем, что значение переменной `word` является строкой (`str`), мы можем использовать все невероятно полезные методы, доступные в Python, для работы с ней. (*Метод* — это функция, относящаяся к переменной, в нашем случае `word`.)

Например, если я захочу закричать, увидев нарвала, я могу вывести это слово ПРОПИСНЫМИ БУКВАМИ. Согласно документации, существует функция `str.upper()`. Вот так можно ее *вызвать* (*выполнить*):

```
>>> word.upper()
'NARWHAL'
```

Обязательно нужно добавить круглые скобки `()`, иначе интерпретатор выведет информацию о *самой функции*:

```
>>> word.upper
<built-in method upper of str object at 0x10559e500>
```

Данный прием пригодится нам позже, в процессе изучения таких функций, как `map()` и `filter()`. Сейчас же нам нужно, чтобы Python вызвал функцию `str.upper()` для переменной `word`, поэтому мы добавляем круглые скобки. Обратите внимание, что функция возвращает вариант слова прописными буквами, но *не* изменяет значение переменной `word`:

```
>>> word
'narwhal'
```

Существует еще одна строковая функция со словом upper в имени — `str.isupper()`. Согласно названию, она возвращает логическое (булево) значение — «истина/ложь». Давайте попробуем:

```
>>> word.isupper()
False
```

Мы можем связать данные методы следующим образом:

```
>>> word.upper().isupper()
True
```

В этом есть смысл. Если я преобразую значение переменной `word` в верхний регистр, метод `word.isupper()` вернет значение `True`.

Я нахожу странным, что класс `str` не содержит метод для извлечения длины строки. Для этого нужно использовать отдельную функцию `len()` (от слова length — длина):


```
>>> len('narwhal')
7
>>> len(word)
7
```

Не могу не поинтересоваться: вы набираете весь код в Python вручную? Я рекомендую вам делать именно так! Поищите другие методы на странице документации к классу `str` и опробуйте их.

2.1.10. Сравнение строк

Теперь вы знаете, как извлечь первую букву слова, хранящегося в переменной `word`, используя метод `word[0]`. Присвоим ее переменной `char`:

```
>>> word = 'octopus'
>>> char = word[0]
>>> char
'o'
```

Если вы проверите тип значения новой переменной `char`, оно будет строковым. Даже один символ по-прежнему рассматривается Python как строка:

```
>>> type(char)
<class 'str'>
```

Теперь необходимо выяснить, гласная или согласная буква хранится в переменной `char`. Допустим, буквы «a», «e», «i», «o» и «y» будут относиться к гласным. Можно использовать оператор == для сравнения строк:

```
>>> char == 'a'
False
>>> char == 'o'
True
```

Примечание. Будьте внимательны и всегда вводите один знак равенства (=) при *присвоении значения* переменной, например `word = 'narwhal'`, и два знака равенства (==, которые я читаю как «равно-равно») при *сравнении двух значений*, например `word == 'narwhal'`. В первом случае используется *оператор*, который изменяет значение переменной `word`, а во втором — *выражение*, возвращающее логическое значение `True` или `False` (рис. 2.2).

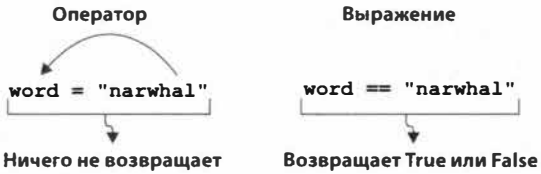


Рис. 2.2. Выражение возвращает значение, а оператор — нет

Необходимо сравнить наш символ со *всеми* гласными. В таких случаях можно использовать ключевые слова `and` и `or`, и выражения будут объединены по правилам булевой алгебры:

```
>>> char == 'a' or char == 'e' or char == 'i' or char == 'o' or
char == 'u'
True
```

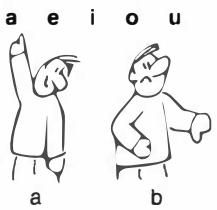
А вдруг переменной `word` присвоено значение `'Octopus'` или `'OCTOPUS'`?

```
>>> word = 'OCTOPUS'
>>> char = word[0]
>>> char == 'a' or char == 'e' or char == 'i' or char == 'o' or
char == 'u'
False
```

Должны ли мы выполнить 10 сравнений, чтобы проверить прописные буквы? Или если нужно вывести значение `word[0]` строчными буквами? Напомню, что метод `word[0]` возвращает строковое значение, поэтому мы можем связать разные строковые методы следующим образом:

```
>>> word = 'OCTOPUS'
>>> char = word[0].lower()
>>> char == 'a' or char == 'e' or char == 'i' or char == 'o' or
char == 'u'
True
```

Существует более простой способ определить, представлено ли значение переменной `char` гласной буквой, — использовать конструкцию `x in y`, которая сообщит нам, находится ли значение `x` в коллекции `y`. Мы можем запросить, есть ли буква `'a'` в длинной строке `'aeiou'`:




```
>>> 'a' in 'aeiou'
True
```

А буквы `'b'` там нет:

```
>>> 'b' in 'aeiou'
False
```

Давайте воспользуемся полученными данными для проверки первого символа слова, хранящегося в переменной `word`, в строчном регистре (это `'o'`):

```
>>> word = 'OCTOPUS'
>>> word[0].lower() in 'aeiou'
True
```

2.1.11. Условное ветвление

Выяснив, является первая буква гласной или нет, следует выбрать артикль. Мы воспользуемся очень простым правилом: если слово начинается с гласной, выбираем артикль `an`; в противном случае — `a`. Так мы обойдем исключения, такие как, например, непроизносимая первая буква h. Ведь мы говорим «a hat» и «an honor». Мы также не будем рассматривать случаи, когда первая гласная буква произносится как согласная, как, например, в слове union, где u произносится как y.

Итак, мы можем создать новую переменную с именем `article`, которой присвоим пустую строку, и воспользуемся оператором `if/else`, чтобы определять, что в нее вставлять:

```
article = ''
if word[0].lower() in 'aeiou':
    article = 'an'
else:
    article = 'a'
```

Инициализируем переменную `article` с пустой строкой.

Проверяем, является ли первая строчная буква слова гласной.

Присваиваем переменной `article` значение `'an'`, если первая буква слова — гласная.

Присваиваем переменной `article` значение `'a'`, если первая буква слова — не гласная.

Существует гораздо более простой способ сделать то же самое с помощью *выражения* `if` (выражения возвращают значения, а операторы — нет). Оно пишется, скажем, немного наоборот. Сначала указывается значение, выводимое, если условие (предикат) истинно, затем сам предикат, а затем значение, выводимое, если предикат ложен (рис. 2.3).

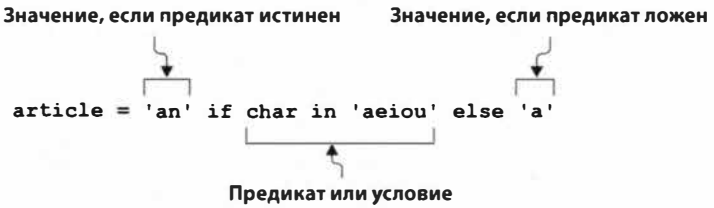


Рис. 2.3. Выражение `if` вернет первое значение, если предикат результирует как `True`, и второе значение в противном случае

Данный подход еще и безопаснее, поскольку выражение `if` *должно* включать случай `else`. Невозможно забыть обработать оба случая:

```
>>> char = 'o'
>>> article = 'an' if char in 'aeiou' else 'a'
```

Теперь проверим, есть ли у нас корректное значение переменной `article`:

Let's verify that we have the correct `article`:

```
>>> article
'an'
```

2.1.12. Форматирование строк

Теперь у нас есть две переменные, `article` и `word`, которые нужно включить в фразу `"Ahoy!.."`. Ранее мы использовали символ плюс (`+`) для конкатенации строк. Другой способ получения новых строк из существующих — и метод `str.format()`.

В таком случае вы создаете шаблон строки с фигурными скобками {}, в которые подставляются значения. Данные значения являются аргументами метода `str.format()` и подставляются в том же порядке, в котором указаны наборы скобок {} (рис. 2.4).

Рис. 2.4. Метод `str.format()` используется для подстановки значений переменных внутрь строк.

Вот как это выглядит в коде:

```
>>> 'Ahoy, Captain, {} {} off the larboard bow!'.format(article, word)
'Ahoy, Captain, an octopus off the larboard bow!'
```

Есть еще один способ объединения строк. Вы можете написать специальную «f-строку» и поместить в нее имена переменных в фигурных скобках { }. Какой подход выбрать, решать вам. Я предпочитаю именно этот способ, потому что не нужно заморачиваться, какая переменная какому набору скобок соответствует:

```
>>> f'Ahoy, Captain, {article} {word} off the larboard bow!'
'Ahoy, Captain, an octopus off the larboard bow!'
```

Примечание. В некоторых языках программирования необходимо объявлять и имя переменной, и *тип* данных, которые она будет содержать. Если переменная объявлена как числовая, ей нельзя присвоить значение другого типа, например строковое. Это так называемая *статическая типизация*, поскольку тип данных переменной нельзя изменить. Python — язык с *динамической типизацией*, то есть вам не нужно объявлять, какие данные будут храниться в переменной. Можно изменить значение и тип данных переменной в любой момент. Это может быть как хорошо, так и плохо. Как говорит Гамлет, «Нет ничего ни хорошего, ни плохого; это размышление делает все таковым».

2.1.13. Самостоятельная работа

Итак, несколько советов по решению задачи.

- Создайте программу с помощью файла `new.py` и поместите в функцию `get_args()` один позиционный аргумент с именем `word`.
- Можно извлечь первый символ слова, проиндексировав его как список, `word[0]`.
- Если вы не хотите проверять прописные и строчные буквы по отдельности, обратитесь к методу `str.lower()` или `str.upper()`, чтобы принудительно использовать один регистр для проверяемой первой гласной или согласной буквы.
- Гласных меньше (шесть, если вы помните), чем согласных, поэтому, думаю, проще проверить, является ли первая буква гласной.
- Можно использовать синтаксис `x in y` и определить, находится ли элемент `x` в коллекции `y`, причем коллекция здесь является списком (`list`).
- Обратитесь к функции `str.format()` или f-строкам, чтобы добавить правильный артикль к указанному слову и вставить все это в текстовую строку.
- Выполняйте команду `make test` (или `pytest -xv test.py`) после каждого изменения кода программы, проверяйте, что она компилируется и работает должным образом.

Теперь напишите программу, а затем изучите мое решение, представленное ниже. Пошевеливайся, лезь в воронье гнездо и за работу!

2.2. Решение

Ниже приведен один из вариантов программы, проходящей все тесты:

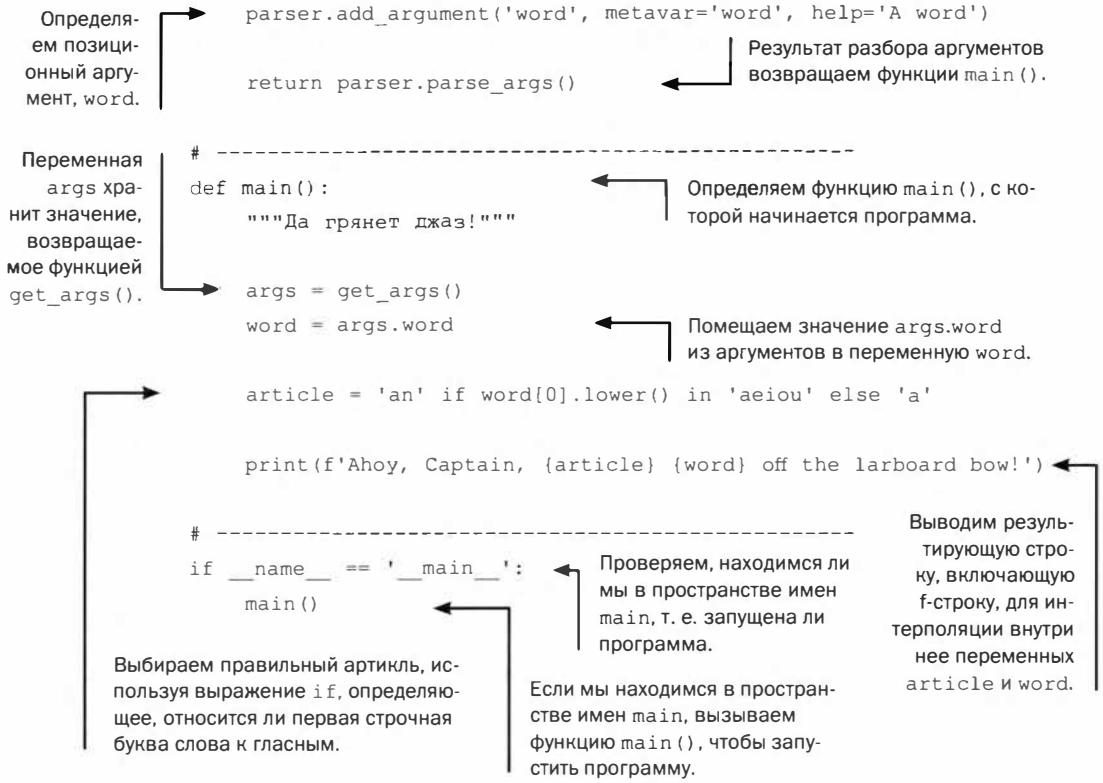
```
#!/usr/bin/env python3
"""Воронье гнездо"""

import argparse

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description="Crow's Nest -- choose the correct article",
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)
```

Определяем функцию `get_args()` для обработки аргументов командной строки. Я привык писать такие определения в первую очередь для удобства чтения исходного кода.

Парсер должен анализировать аргументы.

Инструкция описывает предназначение программы.

Показываем и описываем дефолтные значения для каждого параметра.



2.3. Обсуждение

Я хотел бы подчеркнуть, что приведенное мной решение *не единственное*. Есть много способов решить данную задачу средствами Python. Если ваш код проходит тесты, он правильный.

Я создал свою программу с помощью файла *new.py*, автоматически сгенерировавшего две функции:

— `get_args()`, благодаря которой я определяю аргументы для программы;
— `main()`, с которой начинается программа.

Давайте поговорим об этих двух функциях.

2.3.1. Определение аргументов с помощью get_args()

В первую очередь я пишу функцию `get_args()`, чтобы сразу было понятно, что программа ожидает в качестве входных данных. Вам необязательно оформлять данный код в виде отдельной функции — можно поместить его в функцию `main()`, ваше право. Однако со временем исходный код программ станет гораздо объемнее, и, я считаю, удобнее представить эту часть отдельной функцией. Каждая приведенная в книге программа содержит функцию `get_args()`, определяющую и проверяющую вводимые данные.

Согласно своим характеристикам (спецификациям) наша программа должна принимать один позиционный аргумент. Я изменил имя аргумента с `'positional'` на `'word'`, поскольку ожидается ввод одного слова:

```
parser.add_argument('word', metavar='word', help='Word')
```

Я не рекомендую вам называть позиционный аргумент дефолтным именем `'positional'`, потому что это совершенно не описательное название. Называя свои переменные в соответствии с *тем, что они собой представляют,* вы сделаете код гораздо читабельнее.

Программе не нужен прочий код, сгенерированный файлом *new.py*, так что можно удалить остальную его часть с вызовами функции `parser.add()`.

Функция `get_args()` вернет результат анализа (парсинга) аргументов командной строки, помещенных в переменную `args`:

```
return parser.parse_args()
```

Если модуль `argparse` не может проанализировать аргументы — например, если их нет, — он не вернет результат выполнения функции `get_args()`, а вместо этого выведет инструкции для пользователя и завершит работу, сообщив код ошибки, чтобы оповестить операционную систему о том, что программа завершилась безуспешно. (В командной строке выходное значение `0` означает 0 ошибок. Любое иное значение считается ошибкой.)

2.3.2. О функции main()

Программы на многих языках автоматически начинают работу с функции `main()`, поэтому я всегда определяю данную функцию и запускаю с ее помощью свои программы. Это опциональная, но чрезвычайно

распространенная практика среди программистов на Python. Все представленные в книге программы стартуют с функции `main()`, которая первым делом вызывает функцию `get_args()` для получения входных данных:

```
def main():
    args = get_args()
```

Теперь я могу получить доступ к переменной `word`, обратившись к значению `args.word`. Вы заметили отсутствие скобок? Я не написал `args.word()`, потому что это не вызов функции. Представьте `args.word` в виде ячейки, в которой находится значение переменной `word`:

```
word = args.word
```

Мне нравится реализовывать свои идеи с помощью REPL-интерфейса, поэтому я собираюсь притвориться, что присваиваю переменной `word` значение `'octopus'`:

```
>>> word = 'octopus'
```

2.3.3. Классификация первого символа слова

Чтобы выяснить, какой артикль следует выбрать, `an` или `a`, мне нужно взглянуть на первую букву слова, хранящегося в переменной `word`. Ранее мы использовали следующее:

```
>>> word[0]
'o'
```

Я могу проверить, совпадает ли первая буква с символом в последовательности гласных как строчных, так и прописных:

```
>>> word[0] in 'aeiouAEIOU'
True
```

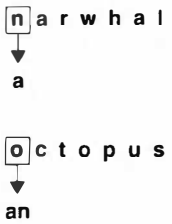
Можно упростить решение, воспользовавшись функцией `word.lower()`. В таком случае мне нужно проверить только строчные гласные:

```
>>> word[0].lower() in  'aeiou'
True
```

Напоминаю, что конструкция `x in y` позволяет узнать, находится ли элемент `x` в коллекции `y`. Попробуем прибегнуть к ней для поиска совпадений букв в более длинной строке (например, в полном списке гласных):

```
>>> 'a' in 'aeiou'
True
```

Можно использовать совпадение с символом в списке гласных как условие для выбора артикля `an`, а в противном случае — артикля `a`. Как упоминалось ранее, выражение `if` — самый короткий и надежный способ решения *дилеммы* (когда доступны только две опции):




```
>>> article = 'an' if word[0].lower() in  'aeiou' else 'a'
>>> article
'an'
```

Надежность выражения `if` обеспечивается тем фактом, что Python даже не запустит программу, если вы забыли написать код `else`. Попробуйте и посмотрите, какую ошибку выбросит интерпретатор.

Давайте изменим значение переменной `word` на `'galleon'` и убедимся, что программа все еще работает:

```
>>> word = 'galleon'
>>> article = 'an' if word[0].lower() in  'aeiou' else 'a'
>>> article
'a'
```

2.3.4. Вывод результатов

Наконец, необходимо вывести текстовую фразу с артиклем и указанным словом. Как отмечалось ранее, можно использовать функцию `str.format()` для добавления переменных в строку:

```
>>> article = 'a'
>>> word = 'ketch'
>>> print('Ahoy, Captain, {} {} off the larboard bow!'.
format(article, word))
Ahoy, Captain, a ketch off the larboard bow!
```

В Python f-строки позволяют разместить имена переменных *внутри* заглушек { }, при запуске выводя их значения:

```
>>> print(f'Ahoy, Captain, {article} {word} off the larboard bow!')
Ahoy, Captain, a ketch off the larboard bow!
```

Независимо от выбранного способа, программа выводит подходящий артикль и слово и проходит тесты. Хотя все зависит от вашего личного вкуса, я считаю, что f-строки удобнее, так как не нужно сверять позиции заглушек { } с переменными, значения которых будут выводиться в итоге.

2.3.5. Тестирование программы

> *«Компьютер похож на озорного джинна. Он даст вам именно то, о чем вы просите, но не всегда то, что вы хотите».*
>
> — Джо Сондоу

Компьютеры напоминают злых джиннов. Они делают именно то, что вы им говорите, но не обязательно то, что вы *хотите*. В одном из эпизодов сериала «*Секретные материалы*» агент ФБР Малдер желает мира на Земле, и джинн, повинуясь, удаляет всех людей, кроме него.

Мы можем использовать тесты, чтобы проверить, что наши программы делают *именно* то, что мы хотим. Тесты никогда не смогут доказать, что программа действительно свободна от ошибок, нет. Они лишь способны показать, что ошибки, которые мы предположили или обнаружили при работе над ней, исправлены. Тем не менее мы пишем и запускаем тесты, потому что они эффективны и все же лучше использовать их, нежели отказаться.

Ключевые идеи *разработки через тестирование*.

— Напишите тесты, *прежде чем* писать само ПО.
— Запустите тесты, чтобы проверить, что еще не написанное ПО не справляется с поставленной задачей.
— Напишите программу для решения задачи.
— Протестируйте программу, чтобы убедиться, что теперь *она* работает.
— Пройдите все тесты, позволяющие гарантировать, что при добавлении нового кода мы не испортим существующий.

Пока мы не будем обсуждать, как *писать* тесты. Вернемся к этому позже. На данный момент я сделал все тесты за вас. Я надеюсь, что к концу данной книги вы убедитесь в преимуществах тестирования и всегда будете начинать с разработки *сначала тестов, а потом самого ПО!*

2.4. Прокачиваем навыки

- Сделайте так, чтобы программа учитывала регистр входящего слова (например, `"an octopus"` и `"An Octopus"`). Скопируйте готовую функцию `test_` в файл test.py, убедитесь, что программа работает правильно и при этом проходит все остальные тесты. Попробуйте сначала написать тест, а затем программу, успешно проходящую его. Это как раз разработка через тестирование!
- Добавьте новый параметр, меняющий larboard (левый борт) на starboard (правый борт). Можно создать либо параметр `--side`, по умолчанию имеющий значение larboard, либо флаг `--starboard`, который, если присутствует, изменяет сторону на правый борт.
- Предоставленные тесты допускают лишь значения, начинающиеся с буквенного символа. Дополните код, чтобы обрабатывать значения, начинающиеся с цифр или знаков препинания. Должна ли ваша программа отклонять их? Напишите дополнительные тесты, позволяющие убедиться, что программа делает именно то, что вы от нее хотите.

РЕЗЮМЕ

- Вся документация по Python доступна по адресу: **https://docs.python.org/3** и с помощью команды `help` в REPL-интерфейсе.
- Переменные в Python типизируются динамически в соответствии с тем значением, которое вы им присваиваете, и они объявляются, когда вы присваиваете им значения.
- Строки поддерживают такие методы, как `str.upper()` и `str.isupper()`, которые можно вызвать, чтобы внести в строки изменения или получить о них информацию.

— Для извлечения части строк используйте квадратные скобки и индексы, например `[0]` для первой буквы или `[-1]` — для последней.
— Строки объединяются (конкатенируются) с помощью оператора `+`.
— Метод `str.format()` позволяет создать шаблон с заглушками (плейсхолдерами) `{}`, которые заполняются с помощью аргументов.
— f-строки, такие как `f'{article} {word}'`, допускают помещение переменных и кода непосредственно в скобки.
— Выражение `x in y` сообщает, присутствует ли значение `x` в коллекции `y`.
— Такие операторы, как `if`/`else`, не возвращают значения, а выражения типа `x if y else z` — возвращают.
— Разработка через тестирование — это способ убедиться, что код программы соответствует некоторым минимальным критериям корректности. Каждая функция программы должна тестироваться, а разработка и выполнение тестов должны стать неотъемлемой частью вашей работы по созданию программ.

Глава 3

Айда на пикник: работа со списками

Когда я пишу код, я жутко хочу есть! Давайте создадим программу, которая будет перечислять вкусняшки, от которых мы бы не отказались.

До сих пор мы имели дело с одиночными переменными, такими как `name` из примера с приветствием, или `word`, которой мы присваивали название животного в программе на морскую тематику. Сейчас мы будем работать с одним или несколькими продуктами и сохраним их названия в списке — переменной, способной хранить любое количество элементов. Мы постоянно используем списки в реальной жизни. Это может быть топ-лист любимых песен, список желаемых подарков на день рождения или подборка брендов, производящих стальные ведра.

В данной главе мы будем собираться на пикник и распечатаем список предметов, которые нужно взять с собой. Вот что вам предстоит сделать:

- написать программу, принимающую несколько позиционных аргументов;
- использовать ключевые слова `if`, `elif` и `else` для обработки условных ветвей с тремя или более параметрами;
- научиться искать и изменять элементы в списке;
- заняться сортировкой и реверсированием списков;
- отформатировать список в новую строку.

Элементы списка передаются как позиционные аргументы. Когда существует только один элемент, вы делаете следующее:

```
$ ./picnic.py salad
You are bringing salad.
```

Чтооо? Кто же приносит на пикник только салат? Если существуют два элемента, вы пишете *and* между ними:

```
$ ./picnic.py salad chips
You are bringing salad and chips.
```

Хм, чипсы. Уже лучше. Когда используются три или более элементов, следует разделять их запятыми:

```
$ ./picnic.py salad chips cupcakes
You are bringing salad, chips, and cupcakes.
```

Интересный поворот. Программа также должна принимать аргумент `--sorted`, позволяющий сортировать элементы перед выводом. Мы решим эту задачу.

Итак, ваша программа на Python должна уметь следующее:

— сохранять один или более позиционных аргументов в виде списка;
— подсчитывать количество аргументов;
— по возможности сортировать элементы;
— использовать содержимое списка для вывода новой строки, форматирующей аргументы в соответствии с количеством элементов.

С чего начнем?

3.1. Приступаем к созданию программы

Я рекомендую всегда начинать с запуска файла *new.py* или с копирования содержимого файла *template/template.py* в файл вашей программы. На этот раз программа будет называться *picnic.py*, и вам нужно создать ее в каталоге *03_picnic*.

Сделать это можно с помощью программы *new.py*, расположенной в корневом каталоге репозитория:

```
$ bin/new.py 03_picnic/picnic.py
Done, see new script "03_picnic/picnic.py."
```

Теперь перейдите в каталог *03_picnic* и выполните команду `make test` или `pytest -xv test.py`. Первые два теста должны быть пройдены успешно (программа существует, программа имеет инструкции), а третий — провален:

```
test.py:: test_exists PASSED                                    [14%]
test.py:: test_usage PASSED                                     [28%]
test.py:: test_one FAILED                                       [42%]
```

Остальная часть вывода сообщает, что тест ожидал строку `"You are bringing chips"`, но получил нечто иное:

```
============================ FAILURES ============================
_____________________________ test_one _____________________________

    def test_one():
        """one item"""

        out = getoutput(f' {prg} chips')
>       assert out.strip() == 'You are bringing chips.'
E       assert 'str_arg = ""...nal = "chips"' == 'You are
bringing
        chips.'
E         + You are bringing chips.
E         -- str_arg = ""
E         -- int_arg = "0"
E         -- file_arg = ""
E         -- flag_arg = "False"
E         -- positional = "chips"

test.py:31: AssertionError
=============== 1 failed, 2 passed in 0.56 seconds ===============
```

Программа запускается с аргументом `chips`.

Строка, начинающаяся со знака +, показывает ожидаемое значение.

Строки, начинающиеся со знака –, показывают, что нам вернула программа.

В данной строке возникает ошибка. Вывод проверяется на соответствие (==) строке "You are bringing chips".

Запустим программу с аргументом `chips` и посмотрим, что получится:

```
$ ./picnic.py chips
str_arg = ""
int_arg = "0"
file_arg = ""
flag_arg = "False"
positional = "chips"
```

Да ну, это вообще неправильно! Разумеется, ведь в шаблоне еще нет *правильных* аргументов, только несколько примеров. Поэтому первое, что необходимо сделать, это исправить код функции `get_args()`. Если *не заданы аргументы*, ваша программа должна выводить справочную информацию, подобную следующей:

```
$ ./picnic.py
usage: picnic.py [-h] [-s] str [str ...]
picnic.py: error: the following arguments are required: str
```

А вот инструкции при использовании флага `-h` или `--help`:

```
$ ./picnic.py -h
usage: picnic.py [-h] [-s] str [str ...]

Picnic game

positional arguments:
  str           Item(s) to bring

optional arguments:
-h, --help    show this help message and exit
-s, --sorted  Sort the items (default: False)
```

Нам нужен один или более позиционных аргументов и необязательный флаг `--sorted`. Измените код функции `get_args()` так, чтобы программа выводила показанный выше результат.

Обратите внимание, что допустим один или более параметров `item`, поэтому вы должны включить определение `nargs ='+'`. Подробную информацию смотрите в разделе A.4.5 в приложении.

3.2. Содержимое файла picnic.py

На рис. 3.1 показана прекрасная диаграмма входных и выходных данных для программы *picnic.py*, которую мы напишем.

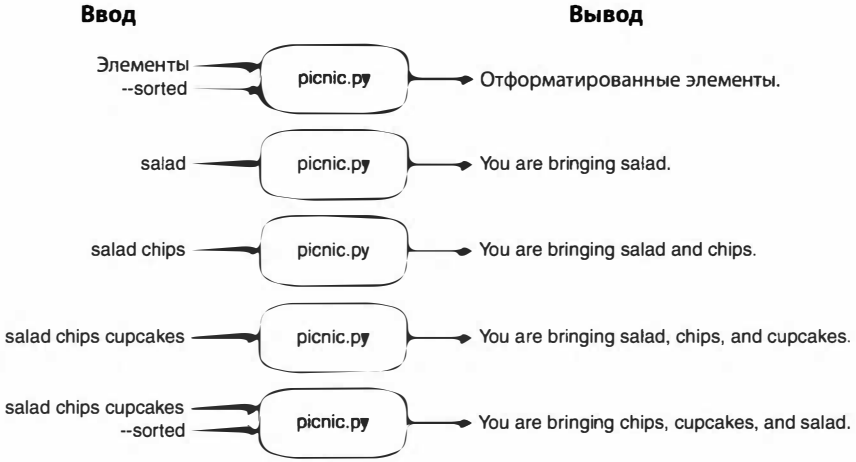


Рис. 3.1. Диаграмма программы picnic.py, отражающая различные входные и выходные данные для обработки

Программа должна принимать один или более позиционных аргументов для предметов, которые нужно принести на пикник, а также *флаг* `-s` или `--sorted`, позволяющий отсортировать элементы. Вывод будет начинаться со слов "You are bringing..." («Вы приносите...»), за которым следует список элементов, отформатированный в соответствии со следующими правилами:

— Если указан один элемент:

```
$ ./picnic.py chips
You are bringing chips.
```

— При наличии двух элементов между ними добавляется союз *and*. Обратите внимание, что "potato chips" — это *одна строка*, содержащая *два слова*. Если вы опустите кавычки, программа примет три аргумента. Не имеет значения, используете вы одинарные или двойные кавычки:

```
$ ./picnic.py "potato chips" salad
You are bringing potato chips and salad.
```

— Если элементов три и более, указываем запятую и пробел между ними, а перед последним — союз *and*. Не забудьте поставить запятую перед союзом *and* (называемую «серийной» или «оксфордской запятой»*), потому что ваш преданный слуга специализировался на английской литературе. И хотя я, наконец, перестал отбивать два пробела после точки в конце предложения**, вы вырвете серийную запятую только из моих окоченевших мертвых рук:

```
$ ./picnic.py "potato chips" salad soda cupcakes
You are bringing potato chips, salad, soda, and cupcakes.
```

Обязательно сортируйте элементы, если указан флаг `-s` или `--sorted`:

```
$ ./picnic.py --sorted salad soda cupcakes
You are bringing cupcakes, salad, and soda.
```

Чтобы посчитать количество элементов, выяснить, как их сортировать и создавать срезы, а также как форматировать выходную строку, необходимо поговорить о списках — типе `list` в Python.

3.3. Да здравствуют списки!

Пришло время узнать, как определять позиционные аргументы, чтобы они были доступны в виде списка. Если мы запустим программу вот так:

```
$ ./picnic.py salad chips cupcakes
```

аргументы `salad chips cupcakes` будут доступны в виде списка строк. Вызвав функцию `print()` для вывода списка, вы увидите следующее:

```
['salad', 'chips', 'cupcakes']
```

---

* Серийная запятая, также известная как оксфордская или гарвардская запятая, — это запятая, используемая в английском языке перед союзом (обычно and или or, а также nor) перед последним пунктом в списке из трех или более элементов. Она не является обязательной, но позволяет избегать неоднозначности. *Прим. ред.*

** Ставить два пробела в конце предложения учили на уроках машинописи в США еще с начала XX века, и традиция эта до сих пор жива, особенно среди пожилых американцев. Обосновывали двойной пробел тем, что на печатной машинке знаки являются моноширинными, поэтому не всегда легко было заметить, когда заканчивается одно предложение и начинается следующее. *Прим. ред.*

Квадратные скобки говорят о том, что мы имеем дело со списком, а кавычки вокруг элементов означают, что это строки. Обратите внимание, элементы отображаются в том же порядке, в котором они были указаны в командной строке. Списки всегда сохраняют порядок элементов.

Давайте перейдем в REPL-интерфейс и создадим переменную `items` для хранения вкуснейших блюд, предназначенных для похода на пикник. Мне очень хочется, чтобы вы собственноручно вводили команды, будь то `python3`, REPL-интерфейс, IPython или Jupyter Notebook. Очень важно взаимодействовать с языком в режиме реального времени.

Чтобы создать новый пустой список, можно использовать функцию `list()`:

```
>>> items = list()
```

Или пустые квадратные скобки:

```
>>> items = []
```

Проверим, что Python скажет о типе данных. Да, это список:

```
>>> type(items)
<class 'list'>
```

Первым делом необходимо узнать, сколько блюд у нас есть для пикника. Как и в случае с классом `str`, мы можем использовать функцию `len()` (length — длина), чтобы получить количество элементов в списке `items`:

```
>>> len(items)
0
```

Длина пустого списка равна 0.

3.3.1. Добавление элемента в список

Пустой список бесполезен. Давайте научимся добавлять в него элементы. В предыдущей главе мы использовали команду `help(str)` для чтения документации о строковых методах — функциях, доступных всем

строкам в Python. Теперь я хочу, чтобы вы воспользовались командой `help(list)`, чтобы узнать о методах списков:

```
>>> help(list)
```

Помните, что вы можете нажать клавишу **Пробел** или **F** (или **Ctrl+F**) для перехода на следующую страницу, либо клавишу **B** (в некоторых случаях **Ctrl+B**) для возвращения на предыдущую. Для поиска нажмите клавишу **/**, а затем введите текст поискового запроса.

Вы увидите множество методов с «двойным подчеркиванием», например `__len__`. Пропустите их все и увидите метод `list.append()`, который мы можем использовать для добавления элементов в конец списка.

Если мы выведем содержимое списка `items`, пустые скобки сообщат нам, что он пуст:

```
>>> items
[]
```

Добавим в конец списка «сэммичи»:

```
>>> items.append('sammiches')
```

Ничего не произошло. И как узнать, добавлен ли элемент? Проверим длину списка. Она должна быть равна `1`:

```
>>> len(items)
1
```

Ура! Сработало. В целях тестирования мы воспользуемся оператором `assert`, проверяющим, что длина списка равна `1`:

```
>>> assert len(items) == 1
```

То, что ничего не происходит, это хорошо. Когда утверждение оказывается ложным, индикатор выбрасывает исключение, приводящее к большому количеству сообщений.

Если в REPL-интерфейсе вы напечатаете `items` и нажмете клавишу **Enter**, Python выведет содержимое данного списка:

```
>>> items
['sammiches']
```

Круто, мы добавили один элемент.

3.3.2. Добавление нескольких элементов в список

Попробуем добавить в список «чипсы» и «мороженое»:

```
>>> items.append('chips', 'ice cream')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: append() takes exactly one argument (2 given)
```

Ой-ой, вот одно из надоедливых исключений, приводящее к *сбою* программы, которого мы стремимся избегать любой ценой. Нам дают понять, что функция `append()` принимает лишь один аргумент, а мы передали ей два. Если вы просмотрите содержимое списка `items`, то увидите, что ничего добавлено не было:

```
>>> items
['sammiches']
```

Хм, так, может, мы должны были передать для добавления список элементов? Давайте попробуем:

```
>>> items.append(['chips', 'ice cream'])
```

Ну, по крайней мере, не возникло исключения, так что, может быть, прием сработал? Предполагается, что элементов в списке будет три, поэтому давайте используем оператор `assert`, чтобы проверить, так ли это:

```
>>> assert len(items) == 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Получаем еще одно исключение, потому что `len(items)` не равно 3. Тогда чему же?

```
>>> len(items)
2
```

Только `2`? Смотрим на список:

```
>>> items
['sammiches', ['chips', 'ice cream']]
```

Поверить не могу, но это правда! Списки могут содержать данные любого типа, такие как строки, числа и даже другие списки (рис. 3.2). Мы предложили функции `items.append()` добавить элемент `['chips', 'ice cream']`, являющийся списком, и она сделала это! Разумеется, нам нужен несколько иной результат.

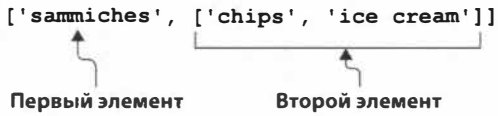


Рис. 3.2. Список может содержать элементы любого типа, например строку и список строк

Давайте вернемся к изначальному содержимому списка `items`, чтобы исправить ситуацию:

```
>>> items = ['sammiches']
```

Если покопаться в документации, вы обнаружите метод `list.extend()`:

```
|    extend(self, iterable, /)
|    Extend list by appending elements from the iterable.
```

Давайте попробуем его применить:

```
>>> items.extend('chips', 'ice cream')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: extend() takes exactly one argument (2 given)
```

Вот черт! Теперь Python говорит нам, что функция `extend()` принимает только один аргумент, который, согласно справке, должен быть итерируемым. Список — это то, что можно итерировать (перебирать содержимое от начала до конца), так что способ должен сработать:

```
>>> items.extend(['chips', 'ice cream'])
```

Ничего не произошло. Исключение не выброшено, так что, может быть, получилось? Проверим длину. Она *должна* быть равна 3:

```
>>> assert len(items) == 3
```

Ура! Давайте взглянем на элементы, которые мы добавили:

```
>>> items
['sammiches', 'chips', 'ice cream']
```

Великолепно! Предвкушаем довольно аппетитную программу.

Если известны все элементы, которые должны быть включены в список, можно создать его так:

```
>>> items = ['sammiches', 'chips', 'ice cream']
```

Методы `list.append()` и `list.extend()` добавляют новые элементы в *конец* заданного списка. Метод `list.insert()` позволяет размещать новые элементы в любой позиции списка путем указания индекса. Можно ввести индекс 0, чтобы поместить новый элемент в начало списка `items`:

```
>>> items.insert(0, 'soda')
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

Я рекомендую вам изучить все методы списков, чтобы понять, насколько мощна эта структура данных. В дополнение к команде `help(list)` вы найдете много отличной документации по адресу: **https://docs.python.org/3/tutorial/datastructures.html**.

3.3.3. Индексация списков

Теперь у нас есть список элементов. Мы научились использовать функцию `len()` для определения количества элементов в списке `items`, а сейчас пришло время узнать, как извлекать части списка для форматирования. Индексирование списков в Python осуществляется точно так же, как и строк (рис. 3.3). (К этому мне сложно привыкнуть, поэтому я представляю, что в случае с `str` список состоит из символов, так мне удобнее.)

```
   0          1           2           3
['soda', 'sammiches', 'chips', 'ice cream']
  -4         -3          -2          -1
```

Рис. 3.3. Индексация и списков, и строк выполняется одинаково. В обоих случаях она начинается с 0. Можно использовать отрицательные числа для индексации в обратном порядке

Индексация в Python начинается всегда с нуля, поэтому первый элемент в списке `items` расположен в позиции `[0]`:

```
>>> items[0]
'soda'
```

Если указан отрицательный индекс, Python выполняет обратный отсчет с конца списка. По индексу `-1` выдается последний элемент списка:

```
>>> items[-1]
'ice cream'
```

Следует быть очень внимательными, ссылаясь на элементы в списке с помощью индексов. Пример небезопасного кода:

```
>>> items[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Внимание! Ссылка на отсутствующий индекс вызовет исключение.

Вскоре вы научитесь безопасно *итерировать* (перебирать содержимое) списки, и индексы для доступа к элементам больше не понадобятся.

3.3.4. Нарезка списков

Можно извлекать фрагменты (подсписки) списков с помощью синтаксиса `list[start: stop]`. Чтобы извлечь первые два элемента, используется код `[0:2]`. Напоминаю, что 2 на самом деле является индексом *третьего* элемента, но срез не включает его, как показано на рис. 3.4.

```
>>> items[0:2]
['soda', 'sammiches']
```

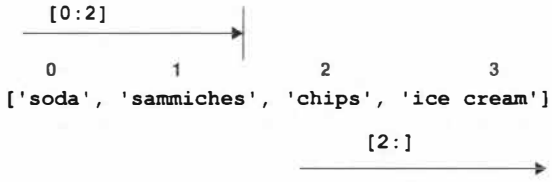


Рис. 3.4. Элемент с индексом `stop`-позиции среза списка не включается в результат. Если значение `stop` опущено, срез выполняется до конца списка

Если вы опустите позицию `start`, по умолчанию будет использоваться значение 0, поэтому строка `[:2]` делает то же самое, что и код `[0:2]`:

```
>>> items[:2]
['soda', 'sammiches']
```

Если вы опустите позицию `stop`, срез будет выполнен до конца списка:

```
>>> items[2:]
['chips', 'ice cream']
```

Как ни странно, для нарезки списков совершенно *безопасно* использовать несуществующие индексы. Например, мы можем запросить все элементы от позиции с индексом `10` до конца, хотя в индексе `10` ничего нет. Вместо выброшенного исключения получаем пустой список:

```
>>> items[10:]
[]
```

В упражнении к этой главе вам нужно будет вставить союз *and* в список элементов, если их насчитывается три и более. Можно ли для решения задачи воспользоваться индексацией списка?

3.3.5. Поиск элементов в списке

А мы не забыли взять с собой чипсы?

Часто требуется узнать, находится ли какой-либо элемент в списке. Метод `index` возвращает позицию элемента:

```
>>> items.index('chips')
2
```

Обратите внимание, что метод `list.index()` небезопасен, потому что вызывает исключение, если аргумент отсутствует в списке. Посмотрите, что произойдет, если мы запросим дымогенератор:

```
>>> items.index('fog machine')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'fog machine' is not in list
```

Ни при каких обстоятельствах не следует использовать метод `list.index()`, предварительно не проверив наличие элемента в списке. Это можно сделать с помощью конструкции `x in y`, к которой мы прибегали в главе 2, проверяя присутствие буквы в строке гласных. Если `x` находится в коллекции `y`, возвращается значение `True`:

```
>>> 'chips' in items
True
```

Хм, надеюсь, это чипсы с солью, а не с крабом...

Тот же код возвращает `False`, если элемент отсутствует в списке:

```
>>> 'fog machine' in items
False
```

Не поговорить ли с комитетом по планированию? Какой же пикник без дымогенератора?

3.3.6. Удаление элементов из списка

Метод `list.pop()` удаляет *и возвращает* элемент с указанным индексом, как показано на рис. 3.5. Без указанного индекса он удаляет *последний* элемент (`-1`).

```
>>> items.pop()
'ice cream'
```

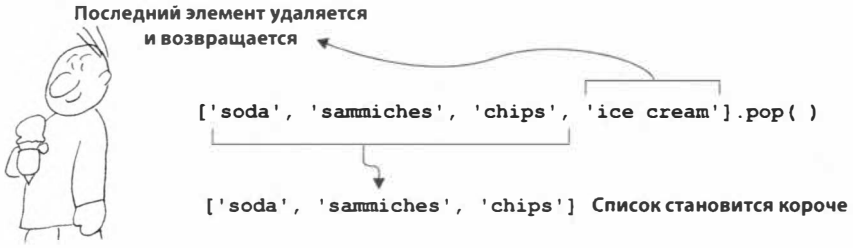


Рис. 3.5. Метод `list.pop()` удаляет элемент из списка

Если мы взглянем на содержимое списка `items`, мы увидим, что теперь он короче на один элемент:

```
>>> items
['soda', 'sammiches', 'chips']
```

Мы можем использовать индексы для удаления элементов в определенных позициях. Например, указать 0, чтобы избавиться от первого элемента (рис. 3.6):

```
>>> items.pop(0)
'soda'
```

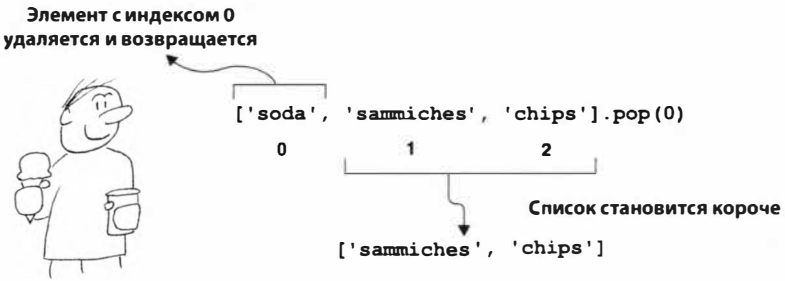


Рис. 3.6. Можно передать значение индекса методу `list.pop()`, чтобы удалить элемент в конкретной позиции

Теперь список `items` стал еще короче:

```
>>> items
['sammiches', 'chips']
```

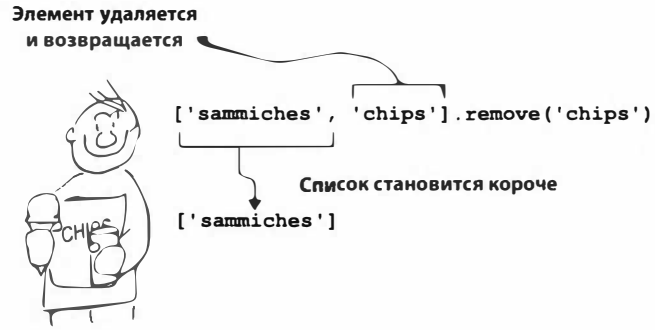


Рис. 3.7. Метод `list.remove()` удаляет элемент, соответствующий заданному значению

Можно также использовать метод `list.remove()` для удаления первого вхождения указанного в скобках элемента (рис. 3.7):

```
>>> items.remove('chips')
>>> items
['sammiches']
```

Внимание! Метод `list.remove()` выбросит исключение при отсутствии элемента.

Если мы вновь применим метод `items.remove()`, повторно удаляя чипсы, мы получим исключение:

```
>>> items.remove('chips')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Поэтому не используйте данный код, не убедившись, что искомый элемент находится в списке:

```
item = 'chips'
if item in items:
    items.remove(item)
```

3.3.7. Сортировка и обращение содержимого списка

Если при вызове программы используется флаг —`sorted`, необходимо отсортировать элементы в списке. В справочной документации написано, что для этого есть два метода, `list.reverse()` и `list.sort()`, которые работают *на месте*. Это означает, что список будет либо обращен, либо отсортирован, и ничего не будет возвращено. Итак, обработав наш список:

```
>>> items = ['soda', 'sammiches', 'chips', 'ice cream']
```

метод `list.sort()` ничего не вернет:


```
>>> items.sort()
```

Если вы проверите содержимое списка, то увидите, что элементы отсортированы в алфавитном порядке:

```
>>> items
['chips', 'ice cream', 'sammiches', 'soda']
```

Как и в случае с методом `list.sort()`, метод `list.reverse()` тоже ничего не возвращает:

```
>>> items.reverse()
```

а элементы списка теперь приведены в обратном порядке:

```
>>> items
['soda', 'sammiches', 'ice cream', 'chips']
```

Методы `list.sort()` и `list.reverse()` можно спутать с *функциями* `sorted()` и `reversed()`. Функция `sorted()` принимает *список* в качестве аргумента и *возвращает* обновленный список:

```
>>> items = ['soda', 'sammiches', 'chips', 'ice cream']
>>> sorted(items)
['chips', 'ice cream', 'sammiches', 'soda']
```

Важно отметить, что функция `sorted()` *не изменяет* предоставленный список:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

Обратите внимание, что Python сортирует список чисел *в числовом отношении*, что весьма приятно:

```
>>> sorted([4, 2, 10, 3, 1])
[1, 2, 3, 4, 10]
```

Внимание! Сортировка списка, содержащего и строки, и числа, вызовет исключение!

```
>>> sorted([1, 'two', 3, 'four'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

Метод `list.sort()` — это функция, принадлежащая списку. Он может принимать аргументы, влияющие на способ сортировки. Давайте выполним команду `help(list.sort)`:

```
sort(self, /, *, key=None, reverse=False)
    Stable sort *IN PLACE*.
```

Судя по всему, мы также можем отсортировать элементы в обратном порядке:

```
>>> items.sort(reverse=True)
```

Теперь список выглядит так:

```
>>> items
['soda', 'sammiches', 'ice cream', 'chips']
```

Функция `reversed()` работает несколько иначе:

```
>>> reversed(items)
<list_reverseiterator object at 0x10e012ef0>
```

Бьюсь об заклад, вы ожидали увидеть новый список с элементами, перечисленными в обратном порядке. Мы столкнулись с примером *ленивой* функции в Python. Процесс обращения списка может занять некоторое время, поэтому Python генерирует *итерируемый объект*, предоставляющий обращенный список, когда нам нужно обратиться к элементам.

Мы можем просмотреть содержимое нашего обращенного списка в REPL-интерфейсе, используя функцию `list()` для вычисления итерируемого объекта:

```
>>> list(reversed(items))
['ice cream', 'chips', 'sammiches', 'soda']
```

Как и в случае с функцией `sorted()`, исходный список `items` остается без изменений:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

Если вы используете метод `list.sort()` вместо функции `sorted()`, вы можете в итоге удалить свои данные. Представьте, что вы хотите сделать список `items` эквивалентным отсортированному списку `items`, к примеру:

```
>>> items = items.sort()
```

Итак, что у нас теперь находится в списке `items`? Если вы выведете его в REPL-интерфейсе, вы не увидите ничего полезного, так что выполните функцию `type()`:

```
>>> type(items)
<class 'NoneType'>
```

Это уже не список. Мы настроили его эквивалентным результату вызова метода `items.sort()`, который изменяет список `items` *на месте* и возвращает `None`.

Если программе передан флаг `--sorted`, вам нужно отсортировать элементы, чтобы пройти тест. Воспользуйтесь методом `list.sort()` или функцией `sorted()`?

3.3.8. Изменяемость списков

Как было показано ранее, мы с легкостью можем изменить список. Методы `list.sort()` и `list.reverse()` меняют содержимое списка полностью, но вы также можете изменить любой конкретный элемент, сославшись на него по индексу. Думаю, нам стоит позаботиться на пикнике о здоровом питании, заменив чипсы на яблоки:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
>>> if 'chips' in items:
...     idx = items.index('chips')
...     items[idx] = 'apples'
...
```

Смотрим, есть ли строка `'chips'` в списке `items`.

Присваиваем индекс элемента `'chips'` переменной `idx`.

Используем индекс `idx`, чтобы изменить элемент `'chips'` на `'apples'`.

Давайте взглянем на список `items`, чтобы проверить результат:

```
>>> items
['soda', 'sammiches', 'apples', 'ice cream']
```

Еще мы можем написать пару тестов:

```
>>> assert 'chips' not in items
>>> assert 'apples' in items
```

Проверяем, что в нашем меню больше нет «чипсов».

Проверяем, что взяли с собой несколько «яблок».

Теперь вам нужно добавить союз *and* в список перед последним элементом, если их два или более. Как реализовать эту идею?

3.3.9. Объединение элементов списка в строку

В упражнении данной главы вам нужно будет вывести строку, учитывающую количество элементов в предоставленном списке. Строка будет сопровождаться другими строками, такими как запятая с пробелом (`', '`), вставляемыми между элементами списка.

Показанный ниже синтаксис соединяет список элементов строкой, состоящей из запятой и пробела:

```
>>> ', '.join(items)
'soda, sammiches, chips, ice cream'
```

В указанном выше коде используется метод `str.join()`, получающий список `items` в качестве аргумента. Способ путаный, но так оно и есть.

В результате выполнения метода `str.join()` создается новая строка:

```
>>> type(', '.join(items))
<class 'str'>
```

Исходный список остается неизменным:

```
>>> items
['soda', 'sammiches', 'chips', 'apples']
```

В Python доступны и другие операции со списками, но изученного на данный момент достаточно для решения задач этой главы.

3.4. Условное ветвление с помощью if/elif/else

Нам потребуется реализовать концепцию условного ветвления, учитывающую количество элементов, чтобы правильно отформатировать вывод. В упражнении из главы 2 ставились два условия — либо гласная, либо нет, — поэтому мы использовали операторы `if/else`. Теперь у нас есть три варианта, значит, придется ввести еще и оператор `elif` (else-if).

Допустим, мы хотим классифицировать пользователей по возрасту, используя три условия.

1. Если возраст больше 0, пользователь допустим.
2. Если пользователю меньше 18 лет, он является несовершеннолетним.
3. В противном случае пользователю уже исполнилось 18 лет и у него есть право голоса.

Вот как мы могли бы написать такой код:

```
>>> age = 15
>>> if age < 0:
...     print('You are impossible.')
... elif age < 18:
...     print('You are a minor.')
... else:
...     print('You can vote.')
...
You are a minor.
```

Подумайте, сможете ли вы, руководствуясь данным примером, выяснить, как написать три условия для программы *picnic.py*. Сначала напишите ветвь, которая обрабатывает один элемент. Затем — ветвь, обрабатывающую два элемента. А потом — последнюю ветвь для трех и более элементов. Запускайте тесты *после каждого изменения кода программы*.

3.4.1. Самостоятельная работа

Теперь попробуйте написать программу самостоятельно, прежде чем подсматривать в мое решение. Приведу несколько советов.

- Перейдите в каталог *03_picnic* и выполните команду `new.py picnic.py` для создания программы. Затем выполните команду `make test` (или `pytest -xv test.py`). Программа должна успешно проходить первые два теста.
- Следом проработайте инструкции по флагу `--help`, как было показано в одном из примеров этой главы ранее. Очень важно правильно определить аргументы. Касательно списка `items` обратите внимание на параметр `nargs` модуля `argparse` (раздел A.4.5 приложения в конце книги).
- Если вы воспользовались файлом *new.py* для создания своей версии программы, обязательно сохраните логический флаг и измените его для флага `sorted`.
- Проходите тесты по порядку! Сначала обработайте один элемент, затем два, а затем три. После чего обработайте отсортированные элементы.

Пользы от книги будет гораздо больше, если вы попробуете самостоятельно писать программы и проходить тесты, прежде чем читать решения.

3.5. Решение

Покажу один из способов пройти тесты. Если ваш код отличается от моего, и при этом он успешно преодолевает тестирование, это здорово!

Функция `get_args()` помещена вначале, чтобы программисту, читающему исходный код, с первого взгляда было понятно, какие данные принимает программа. Обратите внимание, что для Python порядок функций не важен, все — только для нас, людей.

```
#!/usr/bin/env python3
"""Программа про пикник"""

import argparse
# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""
```

```
    parser = argparse.ArgumentParser(
        description='Picnic game',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('item',
                        metavar='str',
                        nargs='+',
                        help='Item(s) to bring')

    parser.add_argument('-s',
                        '--sorted',
                        action='store_true',
                        help='Sort the items')

    return parser.parse_args()

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    items = args.item
    num = len(items)

    if args.sorted:
        items.sort()

    bringing = ''
    if num == 1:
        bringing = items[0]
    elif num == 2:
        bringing = ' and '.join(items)
    else:
        items[-1] = 'and ' + items[-1]
        bringing = ', '.join(items)

    print('You are bringing {}.'.format(bringing))

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Среди аргументов элемента используется код `nargs='+'`, поэтому принимается один или более позиционных строковых аргументов.

Дефисы в коротких (`-s`) и длинных (`--sorted`) именах допускают возможность использования данной опции. С этим аргументом не связано какое-либо значение. Он либо присутствует (`True`), либо отсутствует (`False`).

Обрабатываем аргументы командной строки и возвращаем их вызывающей стороне.

С функции `main()` стартует программа.

Вызываем функцию `get_args()` и присваиваем возвращенное значение в качестве значения переменной `args`. Если при парсинге аргументов возникнет проблема, программа завершится ошибкой до того, как будут возвращены значения.

Копируем список элементов из переменной `args` в новую переменную `items`.

С помощью функции вычисления длины строки `len()` определяем количество элементов в списке. Список не может быть пуст, так как мы определили аргумент с помощью `nargs='+'`, а потому обязательно требуется хотя бы одно значение.

Значение `args.sorted` будет либо `True`, либо `False`.

Если необходимо отсортировать элементы, вызываем метод сортировки `items.sort()`, который сделает это на месте.

Используем пустую строку для инициализации переменной, хранящей объединяемые элементы.

Если количество элементов равно 1, мы назначаем один элемент для объединения.

Соединяем элементы в строке запятой с пробелом.

Если количество элементов равно 2, помещаем строку с союзом `'and'` между элементами.

В противном случае меняем последний элемент в списке `items`, добавляя строку `'and'` перед ним.

Выводим результирующую строку с помощью метода `str.format()` для интерполяции объединяющей переменной.

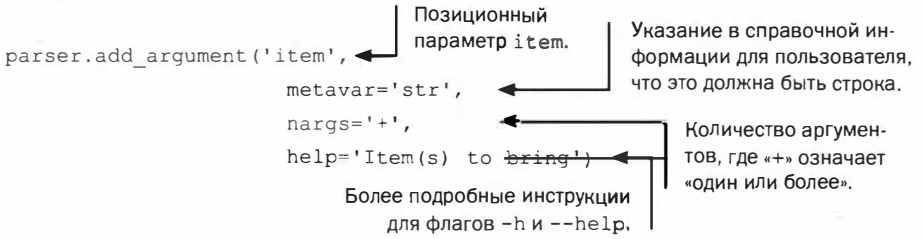
После запуска программы интерпретатор считывает все строки до этой позиции, но ничего не выполняет. Здесь определяется, находимся ли мы в «основном» пространстве имен. Если это так, вызывается функция `main()`, выполняющая код программы.

3.6. Обсуждение

Ну как? Долго трудились над своей программой? Ее код сильно отличается от приведенного выше? Давайте поговорим о моем решении. Не паникуйте, если ваше не совпадает с ним, главное, чтобы программа проходила тесты.

3.6.1. Определение аргументов

Моя программа принимает вариабельное количество строковых аргументов. В своем решении в методе `get_args()` я определяю элемент следующим образом:



Моя программа также принимает аргументы `-s` и `--sorted`. Это флаги, то есть они либо активны (`True`), либо нет (`False`). На опциональность указывают дефисы, предшествующие букве флага.

3.6.2. Присвоение и сортировка элементов

В функции `main()` я вызываю метод `get_args()`, чтобы получить аргументы, и присваиваю их переменной `args`. Затем я создаю переменную `items` для хранения значения (значений) `args.item`:

```
def main():
    args = get_args()
    items = args.item
```

Если значение `args.sorted` равно `True`, необходимо отсортировать элементы. Я выбрал метод сортировки на месте:

```
if args.sorted:
    items.sort()
```

Теперь у меня есть элементы (отсортированные при необходимости), и нужно отформатировать их для вывода.

3.6.3. Форматирование элементов

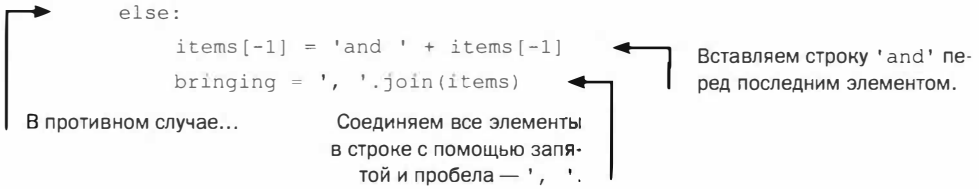
Я предложил проходить тесты по порядку. Предполагаются четыре условия.

1. Элементы отсутствуют.
2. Один элемент.
3. Два элемента.
4. Три и более элементов.

Первый тест обрабатывается модулем `argparse` — если пользователь не предоставил требуемые аргументы, он получает справочное сообщение с инструкциями:

```
$ ./picnic.py
usage: picnic.py [-h] [-s] str [str ...]
picnic.py: error: the following arguments are required: str
```

Итак, модуль `argparse` действует за нас в случае отсутствия аргументов, а мы должны обработать остальные три ситуации. Ниже один из способов сделать это.



Есть идеи, как еще можно решить эту задачу?

3.6.4. Вывод элементов

Наконец, в целях вывода результата с помощью функции `print()` я использовал форматирующую строку, в которой в качестве заглушки для значения выступают скобки {}:

```
>>> print('You are bringing {}.'.format(bringing))
You are bringing salad, soda, and cupcakes.
```

Если вам так удобнее, можете прибегнуть к `f''`-строке:

```
>>> print(f'You are bringing {bringing}.')
You are bringing salad, soda, and cupcakes.
```

В обоих случаях будет достигнут нужный результат.

3.7. Прокачиваем навыки

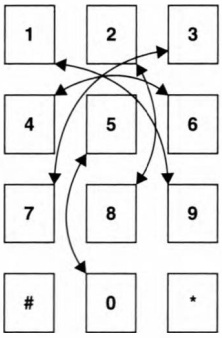
— Сделайте так, чтобы пользователь мог не использовать серийную запятую (хотя это и неприемлемый с моральной точки зрения вариант).
— Реализуйте возможность разделять элементы символом, переданным пользователем (например, точкой с запятой, если нужно, чтобы список элементов разделялся запятыми).
— Не забудьте добавить необходимые тесты в программу *test.py*, чтобы проверить корректность кода новых функций.

РЕЗЮМЕ

— Списки представляют собой упорядоченные последовательности других типов данных, таких как строки и числа.
— Существуют такие методы, как `list.append()` и `list.extend()`, предназначенные для добавления элементов в список. А методы `list.pop()` и `list.remove()` применяются для их удаления.
— Можно использовать конструкцию `x in y`, чтобы проверить, находится ли элемент `x` в списке `y`. Также есть метод `list.index()`, позволяющий определить индекс элемента, однако он вызывает исключение при поиске позиции отсутствующего элемента.
— Списки можно сортировать и обращать, а элементы в них изменять. Списки пригодятся, если важен порядок элементов.
— Строки и списки имеют много общего, например функцию `len()` для определения длины, индексацию с отсчетом от нуля, где `0` — первый элемент, а `-1` — последний, и использование срезов для извлечения меньших частей из бóльших.
— Для создания новой строки из списка применяется метод `str.join()`.
— Для ветвления кода в зависимости от условий используется конструкция `if/elif/else`.

Глава 4
Прыжок через пятерку: работа со словарями

«Когда я встаю, ничто не в силах сбить меня с ног».
— Дэвид Ли Рот



В одном из эпизодов сериала *«Прослушка»* дельцы предполагают, что полиция перехватывает их переписку. Поэтому в ходе преступного сговора они шифруют передаваемые на пейджер телефонные номера. Они используют алгоритм, который можно назвать «Прыжок через пятерку». Суть его в том, что каждая цифра номера заменяется на цифру на противоположной стороне кнопочного номеронабирателя путем перескока через кнопку 5. В данной главе мы обсудим, как шифровать сообщения с помощью такого алгоритма, и разберемся, как использовать его для расшифровки. Уже предвкушаете?

Если мы начнем с кнопки 1 и перепрыгнем через 5, то получим 9. С 6 прыгаем через 5 и получаем 4, и т. д. Кнопки 5 и 0 перепрыгивают через 8.

В текущем упражнении мы напишем программу *jump.py*, которая будет принимать некий текстовый ввод в качестве позиционного аргумента. Каждая цифра в тексте должна быть запрограммирована с помощью описанного выше алгоритма, а буквенные символы выводиться без изменений. Вот несколько примеров:

```
$ ./jump.py 867-5309
243-0751
$ ./jump.py 'Call 1-800-329-8044 today!'
Call 9-255-781-2566 today!
```

Вам нужно будет найти способ проверить каждый символ в вводе, чтобы определить цифры — для этого подойдет цикл `for`. Затем мы разберемся, как использовать цикл `for` для «представления списка»*. Вам понадобится связать цифру, например 1, с цифрой 9 и т. д. для всех цифр, с чем поможет структура данных Python «*словарь*».

В этой главе вы научитесь:

— создавать словари;
— использовать цикл `for` и представление списка для обработки текста посимвольно;
— проверять существование элементов в словаре;
— извлекать значения из словаря;
— выводить новую строку с цифрами, зашифрованными алгоритмом «Прыжок через пятерку».

Однако, прежде чем начать писать код, вам необходимо познакомиться со словарями Python.

4.1. Словари

Словарь Python позволяет связывать *одни* данные (ключи) с *другими* (значения). Вспомните обычную работу со словарем. Если найти в нем некий термин, скажем, «причуда» (**https://pishugramotno.ru/morfologiya/причуда**), то можно прочитать его определение, как показано на рис. 4.1. Представьте термин «ключом», а определение — «значением».

причуда ⇨ странный каприз, чудачество

Рис. 4.1. Можно прочитать определение слова, найдя его в словаре

* Мы приводим один из вариантов перевода функции list comprehension на русский язык. Устоявшегося перевода нет. *Прим. перев.*

Конечно, словари предоставляют гораздо больше информации. Не только определение, но и произношение, указание части речи, производные слова, историю, синонимы, альтернативные варианты написания, этимологию, первое известное использование и т. д. (Я очень люблю словари.) Каждый из этих атрибутов имеет значение, поэтому вполне можно рассматривать словарную статью для слова как еще один «словарь» (рис. 4.2).

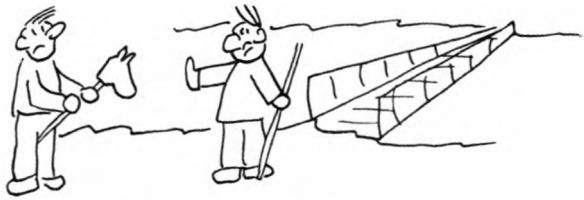
определение ⇨ странный каприз, чудачество
род ⇨ женский
часть речи ⇨ существительное

Рис. 4.2. Запись в словаре для слова «причуда» может содержать гораздо больше информации помимо самого определения

Давайте научимся пользоваться словарями Python и выберемся за рамки определений терминов.

4.1.1. Создание словаря

В фильме «*Монти Пайтон и Священный Грааль*» король Артур и его рыцари должны пересечь Мост Смерти. Тому, кто желает его перейти, нужно правильно ответить на три вопроса Хранителя. Тех, кто терпит неудачу, сбрасывают в Ущелье Вечной Опасности.



Давайте отправимся в Камелот... Ой, прошу прощения, давайте создадим словарь, чтобы обрабатывать вопросы и ответы как пары «ключ/значение». Итак, я хочу, чтобы вы выполнили команду `python3`, запустили IPython REPL-интерфейс или открыли блокнот Jupyter Notebook, и ввели все строки кода вручную.

Ланселот первый. Мы возьмем функцию `dict()` для создания пустого словаря, который будет хранить ответы Ланселота.

```
>>> answers = dict()
```

Как вариант, можно использовать пустые фигурные скобки (оба метода приведут к одному и тому же результату):

```
>>> answers = {}
```

Первый вопрос Хранителя: «Стой! Как твое имя?» Ланселот отвечает: «Сэр Ланселот из Камелота». Мы можем добавить ключ «имя» в словарь с ответами `answers`, используя квадратные (не фигурные!) скобки (`[]`) и строковый литерал `'name'`:

```
>>> answers['name'] = 'Sir Lancelot'
```

Если в REPL-интерфейсе вы введете слово `answers` и нажмете клавишу **Enter**, Python выдаст структуру в фигурных скобках (рис. 4.3), указывающих на то, что это словарь:

```
>>> answers
{'name': 'Sir Lancelot'}
```

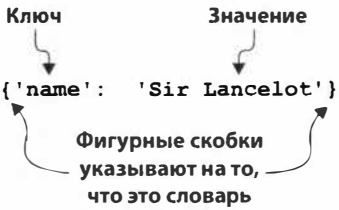


Рис. 4.3. Содержимое словаря выводится в фигурных скобках. Ключи отделены от значений двоеточием

Можно проверить тип данных с помощью функции `type()`:

```
>>> type(answers)
<class 'dict'>
```

Затем Хранитель спрашивает: «Какова твоя цель?», на что Ланселот отвечает: «Я ищу Священную чашу Грааля». Давайте добавим «цель» в словарь `answers`:

```
>>> answers['quest'] = 'To seek the Holy Grail'
```

Ничего не возвращается, поэтому предлагаю проверить успешность операции путем ввода слова `answers` и убедиться, что новая пара «ключ/значение» добавлена:

```
>>> answers
{'name': 'Sir Lancelot', 'quest': 'To seek the Holy Grail'}
```

Наконец Хранитель спрашивает: «Какой твой любимый цвет?», и Ланселот отвечает: «Синий».

```
>>> answers['favorite_color'] = 'blue'
>>> answers
{'name': 'Sir Lancelot', 'quest': 'To seek
the Holy Grail', 'favorite_color': 'blue'}
```

Примечание. В качестве ключа я использовал значение `'favorite_color'` (с подчеркиванием), хотя мог бы выбрать вариант `'favorite color'` (с пробелом), `'FavoriteColor'` или даже `'Favorite Color'`, каждый из которых стал бы прекрасным ключом. Но я предпочитаю придерживаться соглашения PEP 8 насчет имен переменных, функций и ключей в словарях. В документе «PEP 8 — руководство по написанию кода на Python» (**www.python.org/dev/peps/pep-0008**) рекомендуется использовать имена, набранные строчными буквами со словами, разделенными символами подчеркивания.

Если знать все ответы сразу, можно создать словарь `answers` с помощью функции `dict()`, как показано ниже. В таком случае ключи *не* нужно заключать в кавычки, а отделяются от значений они знаками равенства:

```
>>> answers = dict(name='Sir Lancelot', quest='To seek the Holy
Grail', favorite_color='blue')
```

Или можно использовать показанный ниже синтаксис. Здесь есть фигурные скобки {}, каждый ключ заключается в кавычки, а за ним следует двоеточие (`:`):

```
>>> answers = {'name': 'Sir Lancelot', 'quest': 'To seek the Holy
Grail', 'favorite_color': 'blue'}
```

Итак, наш словарь `answers` содержит карточки с парами «ключ/значение» с ответами Ланселота (рис. 4.4), подобно тому, как обычный словарь содержит всю информацию о термине «причуда».

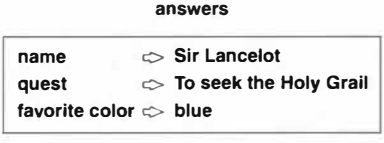


Рис. 4.4. Как и в случае со словарной статьей к слову «причуда», словарь Python может содержать множество пар «ключ/значение»

4.1.2. Доступ к значениям словаря

Для извлечения значения используется имя ключа в квадратных скобках (`[]`). К примеру, можно извлечь «имя» вот так:

```
>>> answers['name']
'Sir Lancelot'
```

Попробуем запросить «возраст»:

```
>>> answers['age']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'age'
```

Как видите, получаем исключение, поскольку передали несуществующий ключ.

Так же, как в случае со строками и списками, мы можем использовать конструкцию `x in y`, чтобы сначала выяснить, существует ли искомый ключ в словаре:

```
>>> 'quest' in answers
True
>>> 'age' in answers
False
```

Метод `dict.get()` — *безопасный* способ извлечь значение:

```
>>> answers.get('quest')
'To seek the Holy Grail'
```

Если запрошенный ключ в словаре отсутствует, возвращается специальное значение `None`:

```
>>> answers.get('age')
```

Вы не сможете его увидеть, так как в REPL-интерфейсе значения `None` не выводятся, однако вы в силах проверить тип данных с помощью функции `type()`. Обратите внимание, что тип `None` — это `NoneType`:

```
>>> type(answers.get('age'))
<class 'NoneType'>
```

Допустим опциональный второй аргумент, который можно передать методу `dict.get()`. Это значение, которое будет возвращено, *если ключ не существует*:

```
>>> answers.get('age', 'NA')
'NA'
```

Такое поведение важно для решения, нам предстоит работать только с символами 0–9.

4.1.3. Другие словарные методы

Чтобы узнать «объем» словаря, воспользуйтесь функцией `len()` (length — длина), она расскажет, сколько пар «ключ/значение» в нем хранится:

```
>>> len(answers)
3
```

Метод `dict.keys()` выдаст только ключи:

```
>>> answers.keys()
dict_keys(['name', 'quest', 'favorite_color'])
```

А метод `dict.values()` — только значения:

```
>>> answers.values()
dict_values(['Sir Lancelot', 'To seek the Holy Grail', 'blue'])
```

Часто бывает нужно и то и другое, поэтому вы можете встретить следующий код:

```
>>> for key in answers.keys():
...     print(key, answers[key])
...
name Sir Lancelot
quest To seek the Holy Grail
favorite_color blue
```

Существует и более простой способ решения данной задачи — использовать метод `dict.items()`, который возвращает содержимое словаря в виде отдельных списков с парами «ключ/значение»:

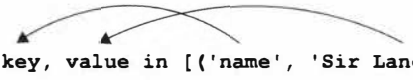
```
>>> answers.items()
dict_items([('name', 'Sir Lancelot'), ('quest', 'To seek the Holy
Grail'), ('favorite_color', 'blue')])
```

Продемонстрированный выше цикл `for` также может быть записан с применением метода `dict.items()`:

```
>>> for key, value in answers.items():
...     print(f' {key:15} {value}')
...
name            Sir Lancelot
quest           To seek the Holy Grail
favorite_color  blue
```

Распаковываем каждую пару «ключ/значение» в переменные с ключом и значением (рис. 4.5). Обратите внимание, нет необходимости называть их `key` и `value`. Можно использовать буквы `k` и `v` или слова `question` и `answer`.

Выводим ключ в выровненное по левому краю поле шириной 15 символов. Значение выводится нормально.



```
for key, value in [('name', 'Sir Lancelot'), ...]:
```

Рис. 4.5. Мы можем распаковать пары «ключ/значение», возвращаемые методом `dict.items()`, в переменные

Выполните в REPL-интерфейсе функцию `help(dict)`, чтобы просмотреть все доступные методы, такие как `dict.pop()`, удаляющий пары «ключ/значение», или `dict.update()`, объединяющий один словарь с другим.

Совет. Каждый ключ в словаре должен быть уникален.

Если вы присвоите значение одному и тому же ключу дважды:

```
>>> answers = {}
>>> answers['favorite_color'] = 'blue'
>>> answers
{'favorite_color': 'blue'}
```

в программе будут не две пары «ключ/значение», а одна, со *вторым* значением:

```
>>> answers['favorite_color'] = 'red'
>>> answers
{'favorite_color': 'red'}
```

Ключи необязательно должны быть представлены строками — вы спокойно можете использовать числа, например типы `int` и `float`. Суть в том, что какое бы значение вы ни выбрали, оно должно быть неизменным. К примеру, вам не подойдут списки, поскольку они изменяемы. Более подробно о том, какие типы являются неизменяемыми, поговорим чуть позже.

4.2. Создание файла jump.py

Итак, приступим. Вам нужно создать программу *jump.py* в каталоге *04_jump_the_five*, чтобы иметь возможность запускать там файл *test.py*. На рис. 4.6 показана диаграмма входных и выходных данных. Обратите внимание, что программа затрагивает в тексте только цифры. Все прочие данные, *не* являющиеся цифрами, не изменяются.

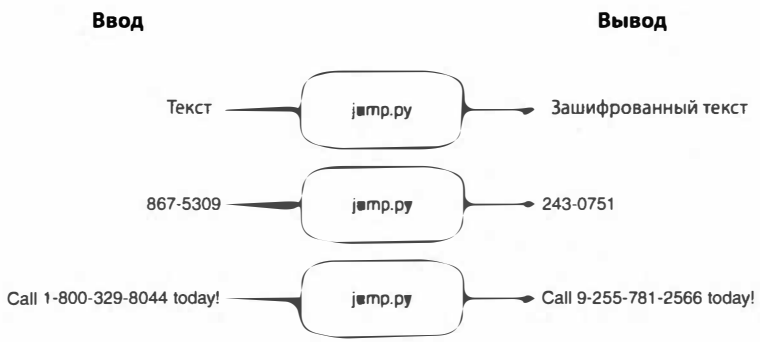


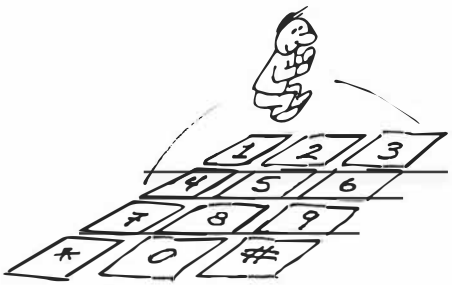
Рис. 4.6. Диаграмма программы jump.py. Все цифры во входном тексте будут заменены соответствующими цифрами в выводе

Если ваша программа запускается с аргументом `-h` или `--help`, она должна вывести инструкции для пользователя:

```
$ ./jump.py -h
usage: jump.py [-h] str
Jump the Five

positional arguments:
  str         Input text

optional arguments:
-h, --help  show this help message and exit
```

Важно отметить, что мы собираемся обрабатывать *текстовые* представления «цифр», поэтому строка `'1'` будет преобразовываться в строку `'9'`. Мы не будем приводить фактическое целочисленное значение `1` к целочисленному значению `9`. Имейте это в виду, когда будете думать над способом реализации замен из таблицы 4.1.

Таблица 4.1. Таблица замены цифровых символов в тексте

1 => 9 2 => 8 3 => 7 4 => 6 5 => 0 6 => 4 7 => 3 8 => 2 9 => 1 0 => 5	

Как вы решите данную задачу с помощью словаря? Попробуйте в REPL-интерфейсе создать словарь `jumper` с таблицей поиска с указанными выше парами «ключ/значение», а затем проверить, будут ли показанные ниже операторы `assert` выполняться с исключениями. Помните, что операторы `assert` ничего не возвращают, если утверждение истинно.

```
>>> assert jumper['1'] == '9'
>>> assert jumper['5'] == '0'
```

Далее вам понадобится обработать каждый символ. Я предлагаю воспользоваться циклом `for`:

```
>>> for char in 'ABC123':
...     print(char)
...
A
B
C
1
2
3
```

Попробуйте вывести значение символа из словаря `jumper` с таблицей поиска или сам символ. Обратите внимание на метод `dict.get()`! Кроме того, если вы прочтете документацию, `help(print)`, вы узнаете, что существует опция `end`, позволяющая заменить символ перевода строки в конце другим символом (символами).

Вот еще несколько советов.

- Цифры могут встречаться в любых позициях в тексте, поэтому я рекомендую обрабатывать ввод посимвольно с помощью цикла `for`.
- После получения символа как вы найдете его в словаре с таблицей поиска?
- Допустим, символ («ключ») находится в словаре, каким образом извлечь его «значение» (зашифрованную цифру)?
- Как вывести зашифрованную цифру (т. е. «значение») без перевода строки? В REPL-интерфейсе откройте документацию (`help(print)`), чтобы узнать про опции функции `print()`.
- Если вы читали документацию по классу `str` (`help(str)`), то знаете про метод `str.replace()`. Можно ли воспользоваться им в данном случае?

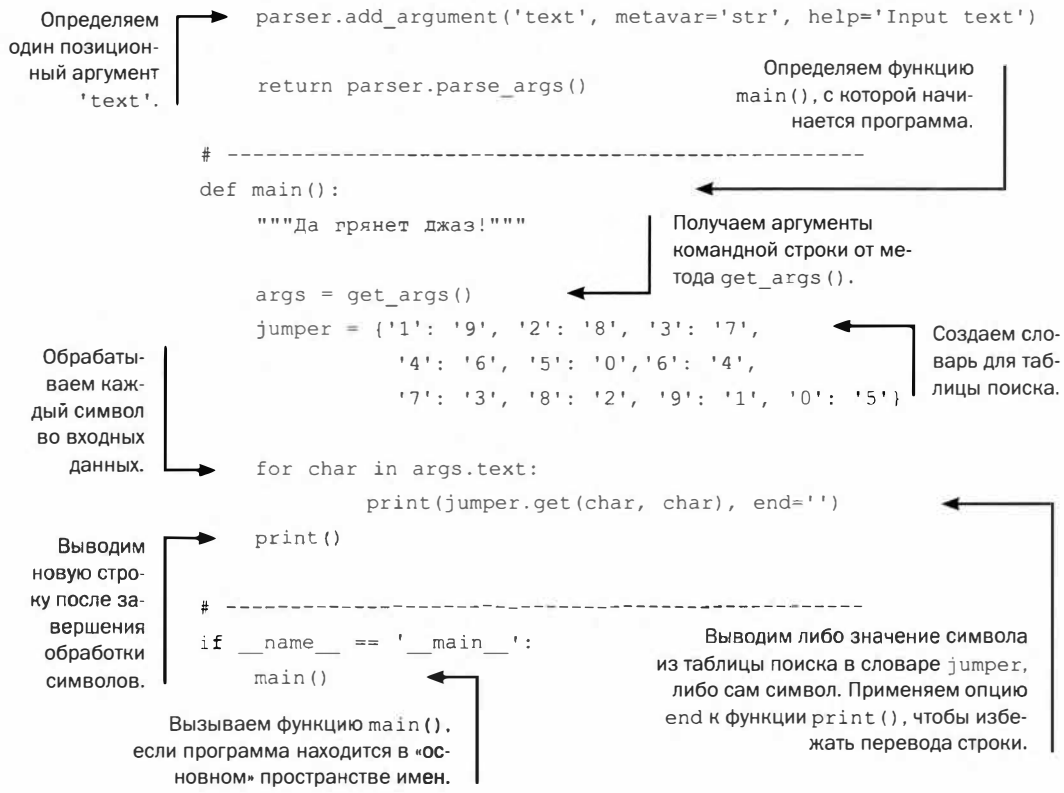
Теперь не поленитесь написать программу самостоятельно, прежде чем подглядывать в мое решение. И не забывайте про всемогущие тесты.

4.3. Решение

Покажу пример программы, который проходит все тесты. После того как обсудим данную версию, расскажу и о других способах решения задачи.

```
#!/usr/bin/env python3
"""Прыжок через пятерку"""

import argparse

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Jump the Five',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)
```

Первым делом определяем функцию `get_args()`, чтобы ее сразу было видно программисту, читающему исходный код.



4.4. Обсуждение

Давайте разобьем цель нашей программы на подзадачи, например такие: определение параметров, определение и использование словаря, обработку входных данных и, наконец, вывод выходных данных.

4.4.1. Определение параметров

Как правило, начинать нужно с функции `get_args()`. Нам необходимо определить один позиционный аргумент. Поскольку программа будет ожидать от пользователя некий «текст», присвоим аргументу имя `'text'`, а затем присвоим его переменной `text`:

```
parser.add_argument('text', metavar='str', help='Input text')
```

Еще раз подчеркну, что крайне важно называть элементы программы *в соответствии с их предназначением*. То есть не стоит использовать дефолтное имя аргумента — `'positional'`, так как оно не отражает его *предназначения*.

Применение модуля `argparse` в такой простой программе может показаться излишним, но помимо прочего он проверяет корректность *количества* и *типа* аргументов, а также формирует справочную документацию, поэтому не пренебрегайте им.

4.4.2. Создание словаря для алгоритма шифрования

Предлагаю создать таблицу поиска в виде словаря, где каждый ключ (исходная цифра) имеет соответствующее значение (зашифрованную цифру). Например, если я с 1 перепрыгну через кнопку 5, то приземлюсь на 9:

```
>>> jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
...           '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
>>> jumper['1']
'9'
```

Поскольку нужно запрограммировать всего 10 цифр, скорее всего, мы сейчас видим самый простой способ решения задачи. Обратите внимание, что цифры пишутся в кавычках, значит, на самом деле это строки, а не целочисленные значения. Мы так делаем потому, что будем считывать символы из строки. Если бы мы сохранили значения в числовом формате, нам пришлось бы приводить типы данных с помощью функции `int()`:

```
>>> type('4')
<class 'str'>
>>> type(4)
<class 'int'>
>>> type(int('4'))
<class 'int'>
```

4.4.3. Различные способы обработки элементов

Напоминаю, что строки и списки в Python схожи в том, как можно их индексировать. И строки, и списки, по существу, представляют собой

последовательности элементов: строки — это последовательности символов, а списки могут быть последовательностями чего угодно.

Существует несколько различных способов обработки любой последовательности — в нашем случае, символов в строке.

Способ 1: посимвольный вывод с помощью цикла for

Как я уже упоминал ранее, мы можем обрабатывать каждый текстовый символ, используя цикл `for`. Для начала с помощью конструкции `x in y` проверим, находятся ли нужные текстовые символы в словаре `jumper`:

```
>>> text = 'ABC123'
>>> for char in text:
...     print(char, char in jumper)
...
A False
B False
C False
1 True
2 True
3 True
```

Примечание. Когда функция `print()` получает более одного аргумента, между фрагментами текста добавляется пробел. Решается данная проблема с помощью аргумента `sep`. Подробная информация есть в документации (`help(print)`).

Теперь попробуем зашифровать цифры. Как вариант, можно использовать выражение `if`, чтобы вывести зашифрованное значение из словаря `jumper`, если символ в нем обнаружен, или неизменный символ, в противном случае:

```
>>> for char in text:
...     print(char, jumper[char] if char in jumper else char)
...
A A
B B
C C
1 9
2 8
3 7
```

Сложновато, но придется проверять каждый символ в словаре `jumper`. Ведь если в нем нет некоего символа, скажем, буквы «A», то при запросе такового программа выбросит исключение:

```
>>> jumper['A']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'A'
```

Метод `dict.get()` позволяет безопасно запрашивать имеющееся значение. Запрос отсутствующего значения не приведет к исключению, но и ничего не выведет в REPL-интерфейсе, а вернет в результате значение `None`:

```
>>> jumper.get('A')
```

Чтобы было понятнее, выведите значения с помощью функции `print()`:

```
>>> for char in text:
...     print(char, jumper.get(char))
...
A None
B None
C None
1 9
2 8
3 7
```

Можно передать второй опциональный аргумент методу `dict.get()`, отвечающему за дефолтное значение, возвращаемое в случае отсутствия ключа. В данном примере я хочу вывести сам символ, если его нет в словаре `jumper`. Допустим, в нем нет буквы «A», значит, и вывести надо «A»:

```
>>> jumper.get('A', 'A')
'A'
```

А вот если в словаре есть цифра «5», нужно вывести «0»:

```
>>> jumper.get('5', '5')
'0'
```

Данный прием подойдет для обработки всех символов:

```
>>> for char in text:
...     print(jumper.get(char, char))
...
A
B
C
9
8
7
```

Далее я передаю аргумент `end=''`, чтобы интерпретатор выводил в конце пустую строку, а не осуществлял перевод строки.

После выполнения кода в REPL-интерфейсе мы получаем несколько забавный результат, поскольку интерпретатор ожидает нажатия клавиши **Enter** для запуска цикла `for`. Так мы остаемся с последовательностью `ABC987` и приглашением `>>>` следом, без перевода строки:

```
>>> for char in text:
...     print(jumper.get(char, char), end='')
...
ABC987>>>
```

Видимо, нужно добавить еще одну функцию `print()`.

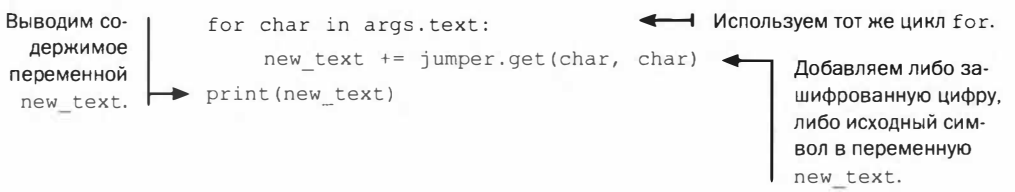
Удобно, что можно изменить символ в конце строки и использовать функцию `print()` без аргумента для перевода строки. У `print()` есть еще несколько реально крутых возможностей, поэтому я рекомендую вам прочитать документацию (`help(print)`) и опробовать их.

Способ 2: создание новой строки с помощью цикла for

Существует несколько других способов решить нашу задачу. Конечно, интересно реализовать все идеи с помощью функции `print()`, однако такой код будет перегружен. Я предлагаю создать переменную `new_text` и однократно вызвать функцию `print()` следующим образом:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    new_text = ''
```

Создаем пустую переменную `new_text`.



Итак, в данной версии программы я начинаю с присвоения переменной `new_text` пустой строки в качестве значения:

```
>>> new_text = ''
```

Затем использую тот же цикл `for` для обработки каждого символа в поступившем вводе. При всех итерациях цикла используется оператор +=. Он добавляет значение, расположенное справа, к переменной, находящейся слева:

```
>>> new_text += 'a'
>>> assert new_text == 'a'
>>> new_text += 'b'
>>> assert new_text == 'ab'
```

С правой стороны применяется метод `jumper.get()`. Все символы будут присвоены переменной `new_text`, как показано на рис. 4.7.

Рис. 4.7. Оператор += добавляет строку, расположенную справа, к переменной слева

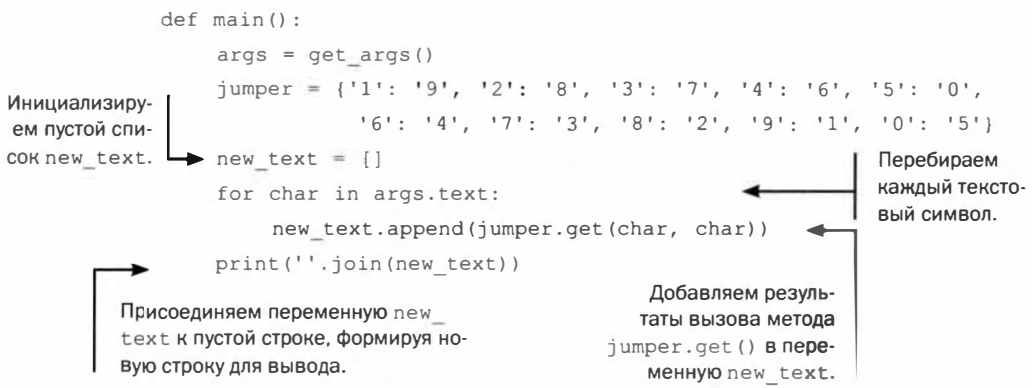
```
>>> new_text = ''
>>> for char in text:
...     new_text += jumper.get(char, char)
...
```

Теперь я могу однократно вызвать функцию `print()` и вывести новую строку:

```
>>> print(new_text)
ABC987
```

Способ 3: создание нового списка с помощью цикла for

Данный метод похож на предыдущий, однако теперь переменная `new_text` представляет собой список, а не строку:



На протяжении всей книги я буду постоянно напоминать, что Python одинаково обрабатывает строки и списки. В данном примере я использую переменную `new_text` точно так же, как и раньше, начав с пустой структуры и затем постепенно заполняя символами. Можно опять прибегнуть к оператору += вместо метода `list.append()`:

```
for char in args.text:
    new_text += jumper.get(char, char)
```

После завершения работы цикла `for` все новые символы собираем в новую строку, используя метод `str.join()`, а затем выводим с помощью функции `print()`.

Способ 4: превращение цикла for в представление списка

Более короткое решение требует *представления списка*. Выполнение однострочного цикла `for` внутри квадратных скобок (`[]`) результирует в новый список (рис. 4.8).

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    print(''.join([jumper.get(char, char) for char in args.text]))
```

Представление списка считывается из цикла `for` в обратном направлении, но все работает. Причем мы получаем одну строку кода вместо четырех!

```
>>> text = '867-5309'
>>> [jumper.get(char, char) for char in text]
['2', '4', '3', '-', '0', '7', '5', '1']
```

Рис. 4.8. Представление списка генерирует новый список с результатами итерирования цикла `for`

Можно использовать метод `str.join()` на пустой строке, чтобы превратить список в новую строку, впоследствии выводимую функцией `print()`:

```
>>> print(''.join([jumper.get(char, char) for char in text]))
243-0751
```

Результат представления списка — новый список, который ранее мы создавали с помощью цикла `for`. Метод представления списков гораздо понятнее и позволяет обходиться меньшим количеством строк кода.

Способ 5: решение задачи с помощью функции str.translate()

Теперь применим мощный метод из класса `str`, чтобы зашифровать все цифры одним махом:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    print(args.text.translate(str.maketrans(jumper)))
```

Аргументом функции `str.translate()` служит таблица поиска, описывающая, как следует менять каждую цифру. Такую таблицу как раз и содержит словарь `jumper`.

```
>>> text = 'Jenny = 867-5309'
>>> text.translate(str.maketrans(jumper))
'Jenny = 243-0751'
```

Рассмотрим более подробно в главе 8.

4.4.4. Подробнее о функции str.replace()

Ранее я спросил, можно ли использовать функцию `str.replace()`, чтобы изменить все цифры. Оказывается, нет, поскольку в итоге некоторые цифры изменятся дважды и вновь станут исходными.

Начнем со следующей строки:

```
>>> text = '1234567890'
```

Изменив цифру «1» на «9», мы получаем две девятки:

```
>>> text = text.replace('1', '9')
>>> text
'9234567890'
```

Теперь, когда вы замените все цифры «9» на «1», вы получите две единицы. Таким образом, единица в первой позиции меняется на девятку, а потом обратно на единицу:

```
>>> text = text.replace('9', '1')
>>> text
'1234567810'
```

В итоге, пробежавшись по каждой цифре в строке «1234567890» и изменив их с помощью функции `str.replace()`, вы получите значение «1234543215»:

```
>>> text = '1234567890'
>>> for n in jumper.keys():
...     text = text.replace(n, jumper[n])
...
>>> text
'1234543215'
```

А на самом деле правильно запрограммированная строка должна выглядеть так: «9876043215».

Функция `str.translate()` пригодна для изменения всех цифр за один ход, при этом оставляя неизменяемые символы в покое.

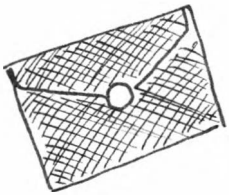
4.5. Прокачиваем навыки

— Создайте аналогичную программу, преобразующую числа в строки (например, чтобы «5» становилась «пятеркой», «7» — «семеркой»). Обязательно добавьте в файл *test.py* необходимые тесты, позволяющие проверить вашу работу.
— Что произойдет, если вы передадите вывод программы обратно самой программе? Например, если вы выполните команду `/jump.py 12345`, вы получите в результате строку `98760`. А если выполните команду `/jump.py 98760`, восстановятся ли исходные цифры? Речь идет о так называемом круговом обходе — распространенной операции, связанной с алгоритмами шифровки и дешифровки текста.

РЕЗЮМЕ

— Новый словарь можно создать с помощью функции `dict()` или фигурных скобок (`{}`).
— Значения словаря извлекаются путем указания ключей в квадратных скобках или с помощью метода `dict.get()`.
— Чтобы выяснить, есть ли некий ключ в словаре `x`, используется конструкция `'key' in x`.
— С помощью цикла `for` можно перебирать символы как строк, так и списков. Строки допустимо представить как списки символов.
— Функция `print()` принимает необязательные аргументы, такие как `end=''`, который позволяет выводить значения на экран без перевода строки.

Глава 5
Кричалка: файлы и потоки STDOUT



В романах о Гарри Поттере есть «Кричалка» (или «Громовещатель») — противная телеграмма, доставляемая в Хогвартс совами. Она раскрывается, выкрикивает гневное послание получателю, а затем сгорает. В данной главе мы напишем программу, своеобразную мягкую версию «Кричалки», которая способна преобразовывать текст, делая ВСЕ БУКВЫ ПРОПИСНЫМИ. Обрабатываемый текст будет передаваться как позиционный аргумент.

К примеру, если программе передается ввод: «Как ты посмел украсть эту машину!», она выдает в ответ: «КАК ТЫ ПОСМЕЛ УКРАСТЬ ЭТУ МАШИНУ!» Помните, что в командной строке аргументы с пробелами между словами считываются по отдельности, поэтому предложение нужно заключать в кавычки, чтобы оно считалось одним аргументом:

```
$ ./howler.py 'How dare you steal that car!'
HOW DARE YOU STEAL THAT CAR!
```

Аргумент программы также может содержать имя файла. Соответственно, входные данные считываются из файла:

```
$ ./howler.py ../inputs/fox.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Кроме того, программа принимает опцию `-o` или `--outfile` с выходным файлом, в который пишется результирующий текст. В таком случае в командной строке *ничего не* отображается:

```
$ ./howler.py -o out.txt 'How
   dare you steal that car!'
```

С помощью кода выше был создан файл *out.txt*, внутри которого записан следующий результат:

```
$ cat out.txt
HOW DARE YOU STEAL THAT CAR!
```

Прочитав данную главу, вы научитесь:

— принимать входные данные из командной строки и из файла;
— менять строчные буквы на прописные;
— выводить результат либо в командную строку, либо в предварительно созданный файл;
— превращать простой текст в файловый дескриптор.

5.1. Чтение файлов

Итак, начнем с чтения файлов. В качестве аргумента программе будет передан некоторый текст, который может включать имя файла. В таком случае интерпретатор откроет и прочитает содержимое файла. Если имя файла не указано, будет обработан сам текст аргумента.

Встроенный модуль `os` (operating system, «операционная система») содержит метод, определяющий имена файлов в передаваемых данных. Чтобы использовать этот модуль, его нужно импортировать:

```
>>> import os
```

Теперь проверим его в работе. Запросим, есть ли на устройстве файл с именем «blaargh»:

```
>>> os.path.isfile('blargh')
False
```

Как оказалось — нет.

Модуль `os` содержит множество полезных подмодулей и функций. Для получения подробной информации обратитесь к документации по адресу: **https://docs.python.org/3/library/os.html** или выполните команду `help(os)` в REPL-интерфейсе.

Например, такие функции, как `os.path.basename()` и `os.path.dirname()`, могут вернуть из пути соответственно имя файла и каталог (рис. 5.1):

```
>>> file = '/var/lib/db.txt'
>>> os.path.dirname(file)
'/var/lib'
>>> os.path.basename(file)
'db.txt'
```

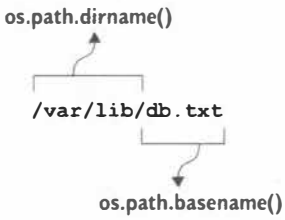


Рис. 5.1. Модуль `os` имеет удобные функции, такие как `os.path.dirname()` и `os.path.basename()` для извлечения путей и имен файлов

В корне исходного репозитория GitHub есть каталог *inputs*, содержащий несколько файлов, которыми мы воспользуемся в следующих упражнениях. Сейчас нам понадобится файл *inputs/fox.txt*. Обратите внимание, чтобы код работал, вам нужно находиться в основном каталоге репозитория.

```
>>> file = 'inputs/fox.txt'
>>> os.path.isfile(file)
True
```

Определив в аргументе имя файла, вы должны открыть (`open()`) его и прочитать (`read()`) содержимое. Функция `open()` вернет *файловый дескриптор*. Обычно я присваиваю соответствующей переменной «говорящее» имя `fh`, чтобы все сразу было понятно. Если же в программе есть сразу несколько открытых файловых дескрипторов, например ввода и вывода, я присваиваю им имена `in_fh` и `out_fh`.

```
>>> fh = open(file)
```

Примечание. Согласно документу «PEP 8 — руководство по написанию кода на Python» (**www.python.org/dev/peps/pep-0008/#function-and-variable-names**), в функциях и переменных рекомендуется использовать имена, набранные строчными буквами со словами, разделенными символами подчеркивания, что может быть необходимо для улучшения читабельности кода.

Если вы попробуете открыть несуществующий файл, то интерпретатор выбросит исключение. Ниже показан небезопасный код:

```
>>> file = 'blargh'
>>> open(file)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'blargh'
```

Всегда проверяйте наличие файла!

```
>>> file = 'inputs/fox.txt'
>>> if os.path.isfile(file):
...     fh = open(file)
```

Мы воспользуемся методом `fh.read()` для извлечения содержимого файла. Представьте файл в виде банки с консервированными помидорчиками. Имя файла, например «inputs/fox.txt», написанное на этикетке банки, не совпадает с *содержимым*. Чтобы добраться до текста внутри (ну то есть до помидорчиков), необходимо *открыть* банку.

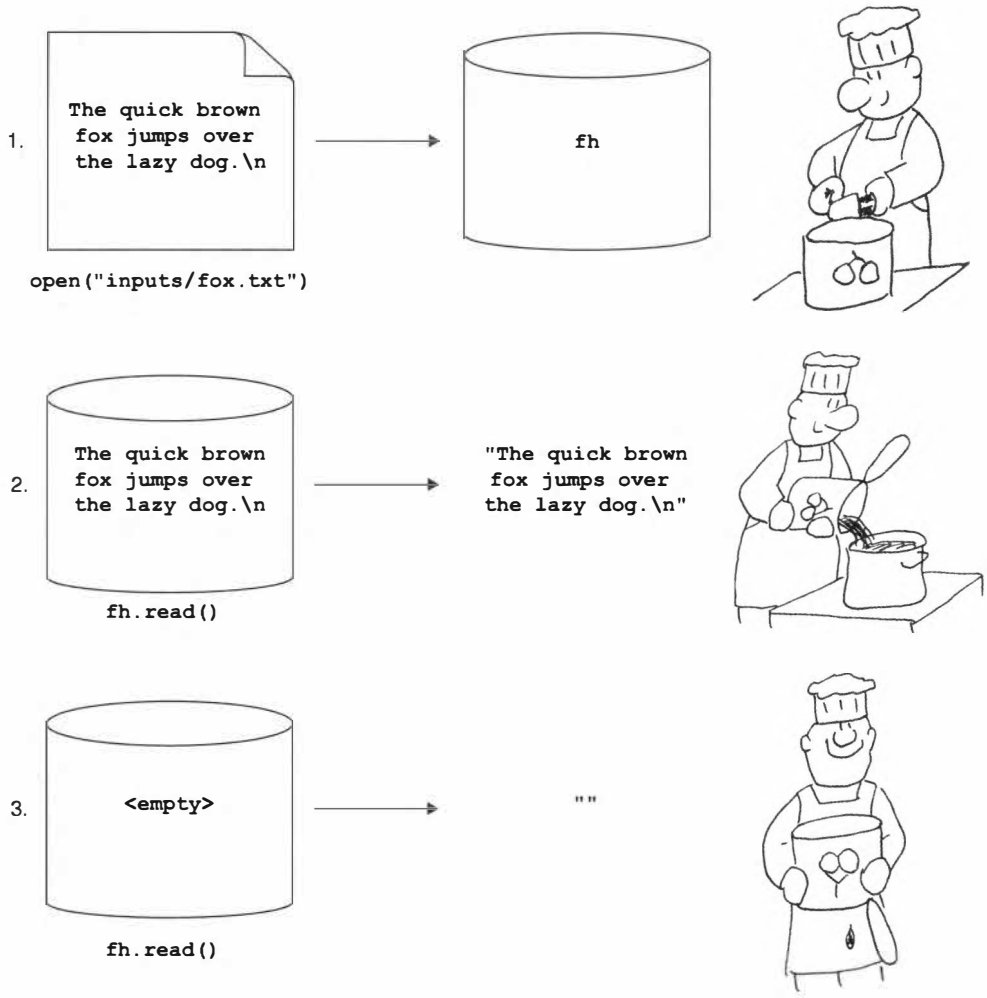


Рис. 5.2. Файл подобен банке с консервированными помидорчиками. Мы должны сначала открыть его, чтобы прочитать содержимое, после чего файловый дескриптор освобождается

Взгляните на рис. 5.2.

1. Файловый дескриптор (`fh`) — это механизм, который мы можем использовать для доступа к содержимому файла. Чтобы добраться до помидорчиков, необходимо открыть банку — `open()`.
2. Метод `fh.read()` возвращает содержимое файла. Открыв банку, мы получаем доступ к тому, что скрывается внутри.
3. После того как файловый дескриптор прочитан, он освобождается.

Примечание. Вы можете использовать метод `fh.seek(0)`, чтобы сбросить файловый дескриптор, если хотите вновь прочитать его.

Давайте посмотрим, к какому типу данных относится значение `fh`:

```
>>> type(fh)
<class '_io.TextIOWrapper'>
```

На компьютерном жаргоне «io» означает «input/output — ввод/вывод». Объект `fh` занимается обработкой операций ввода/вывода. Можно воспользоваться командой `help(fh)` (указав имя самой переменной), чтобы прочитать документацию к классу `TextIOWrapper`.

Вам предстоит часто пользоваться функциями `read()` и `write()`. Сейчас нас интересует `read()`. Рассмотрим ее:

```
>>> fh.read()
'The quick brown fox jumps over the lazy dog.\n'
```

Так-так, давайте повторим эту строчку еще раз. Что вы видите?

```
>>> fh.read()
''
```

Файловый дескриптор отличается от строк. Стоит его прочесть, он оказывается пуст. Это как высыпать помидорчики из банки. Если банка пуста, вы не можете ее снова опустошить.

На самом деле мы можем вместить методы `open()` и `fh.read()` в одну строку кода, *объединив* их. Метод `open()` возвращает файловый дескриптор, который можно использовать для вызова метода `fh.read()` (рис. 5.3). Проверим:

```
>>> open(file).read()
'The quick brown fox jumps over the lazy dog.\n'
```

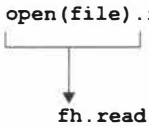


Рис. 5.3. Функция `open()` возвращает файловый дескриптор, поэтому мы можем связать ее с вызовом метода `read()`

А теперь запустим код снова:

```
>>> open(file).read()
'The quick brown fox jumps over the lazy dog.\n'
```

При каждом открытии файла с помощью метода `open()` вы создаете новый файловый дескриптор для метода `read()`.

Если вы хотите сохранить содержимое, необходимо присвоить его переменной:

```
>>> text = open(file).read()
>>> text
'The quick brown fox jumps over the lazy dog.\n'
```

Результат является строкой:

```
>>> type(text)
<class 'str'>
```

Можно добавить любой метод, поместив его в конец. Например, вы хотите избавиться от символа перевода строки. Метод `str.rstrip()` удаляет все пробельные символы (включая символы перевода строки) с *правого* конца строки (рис. 5.4).

```
>>> text = open(file).read().rstrip()
>>> text
'The quick brown fox jumps over the lazy dog.'
```

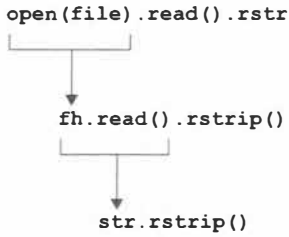


Рис. 5.4. Метод `open()` возвращает файловый дескриптор, к которому мы присоединили функцию `read()`, возвращающую строку с присоединенным методом `str.rstrip()`

Получив на входе текст — из командной строки или из файла, — представим его ПРОПИСНЫМИ буквами. Здесь нам пригодится метод `str.upper()`.

5.2. Запись файлов

Вывод программы нужно либо отобразить в командной строке, либо записать в файл. Вывод в командной строке называется *стандартным выводом*, или `STDOUT` (*STandarD OUTput — стандартное* расположение для *вывода*). Давайте посмотрим, как записать вывод в файл. Нам по-прежнему необходимо открыть файловый дескриптор, но теперь нужно использовать опциональный второй аргумент, строку `'w'`, информируя интерпретатор Python, что файл открывается для *записи*. К другим режимам обработки файлов относятся `'r'` — *чтение* (по умолчанию) и `'a'` — *добавление* или *дозапись*, как указано в таблице 5.1.

Таблица 5.1. Режимы обработки файлов

Режим	Обозначение
`w`	Запись
`r`	Чтение
`a`	Добавление

Можно дополнительно указать тип обрабатываемых данных: `'t'` для *текста* (по умолчанию) или `'b'` для *двоичных* данных (табл. 5.2).

Таблица 5.2. Тип содержимого файла

Режим	Обозначение
`t`	Текст
`b`	Двоичные данные

Вы способны комбинировать режим обработки и тип данных из указанных выше таблиц. Например, использовать `'rb'` для *чтения двоичных* данных или `'at'` для *добавления текстовых* данных в конец файла. В нашем примере мы прибегнем к аргументу `'wt'`, чтобы записать *текстовый* файл.

Присвоим переменной имя `out_fh`, чтобы не забыть, что это дескриптор выходного файла:

```
>>> out_fh = open('out.txt', 'wt')
```

Если файл не существует, он будет создан. Если существует, он будет *перезаписан,* а все имеющиеся данные уничтожены. Если вы не хотите, чтобы исходный файл перезаписывался, вы можете использовать функцию `os.path.isfile()`, которую видели ранее. Она поможет проверить, существует ли файл, и, если так, позволит осуществлять запись в файл в режиме «добавления». В данном упражнении мы воспользуемся режимом `'wt'` для записи текста.

Можно применить метод `write()` файлового дескриптора для записи текста в файл.

Напомню, что функция `print()` добавляет символ перевода строки (`\n`), если, конечно, вы не указываете обратное. А вот функция `write()` *не* добавляет символ перевода строки, поэтому его нужно указывать.

Если воспользоваться методом `out_fh.write()` в REPL-интерфейсе, он вернет количество записанных байтов. В примере каждый символ, включая символ перевода строки (`\n`), является байтом:

```
>>> out_fh.write('this is some text\n')
18
```

Проверим:

```
>>> len('this is some text\n')
18
```

В большинстве случаев возвращаемое значение игнорируется, то есть мы обычно не утруждаемся записью результата в переменную или проверкой того, что в результате возвращается ненулевое значение. Если метод `write()` завершается с ошибкой, обычно в коде присутствует гораздо более серьезная проблема.

Вы также можете использовать метод `print()` с опциональным аргументом `file`. Обратите внимание, что я не указываю символ перевода строки в значении, передаваемом функции `print()`, так как она сама его добавляет. В результате возвращается значение `None`:

```
>>> print('this is some more text', file=out_fh)
```

Закончив запись в файловый дескриптор, вы должны выполнить метод `out_fh.close()`, чтобы закрыть файл и освободить затрачиваемые на него ресурсы памяти. Данный метод также возвращает значение `None`:

```
>>> out_fh.close()
```

Давайте проверим, сохранился ли текст, который мы записали в файл *out.txt*. Для этого его необходимо открыть и прочитать. Запомните, что символ перевода строки отображается в виде `\n`. Нам нужно вывести строку, чтобы выполнить перевод на новую строку:

```
>>> open('out.txt').read()
'this is some text\nthis is some more text\n'
```

Содержимое, выводимое при открытом файловом дескрипторе, добавляется к существующим данным. Взгляните на следующий код:

```
>>> print("I am what I am an' I'm not ashamed.",
          file=open('hagrid.txt', 'wt'))
```

Как вы думаете, если выполнить код дважды, данная строка будет записана в файл *hagrid.txt* один или два раза? Разберемся:

```
>>> open('hagrid.txt').read()
"I am what I am an' I'm not ashamed\n"
```

Только один раз! Но почему? А потому, что всякий раз при вызове метода `open()` создается новый файловый дескриптор. При каждом выполнении данного кода файл открывается заново в режиме *записи*, и исходные данные *перезаписываются*. Чтобы избежать потенциальных проблем, я рекомендую писать код в таком духе:

```
fh = open('hagrid.txt', 'wt')
fh.write("I am what I am an' I'm not ashamed.\n")
fh.close()
```

5.3. Создание файла howler.py

Итак, вам нужно создать программу *howler.py* в каталоге *05_howler*. Вы можете использовать для этого программу *new.py* или скопировать содержимое файла *template.py*, как вам удобнее. Диаграмма на рис. 5.5 демонстрирует схему программы и примеры входных и выходных данных.

При запуске программы без аргументов должно выводиться короткое справочное сообщение:

```
$ ./howler.py
usage: howler.py [-h] [-o str] text
```

```
howler.py: error: the following arguments are required: text
```

При запуске с аргументом `-h` или `--help` — более подробная инструкция:

```
$ ./howler.py -h
usage: howler.py [-h] [-o str] text

Howler (upper-cases input)

positional arguments:
  text                  Input string or file

optional arguments:
-h, --help            show this help message and exit
-o str, --outfile str
                      Output filename (default:)
```

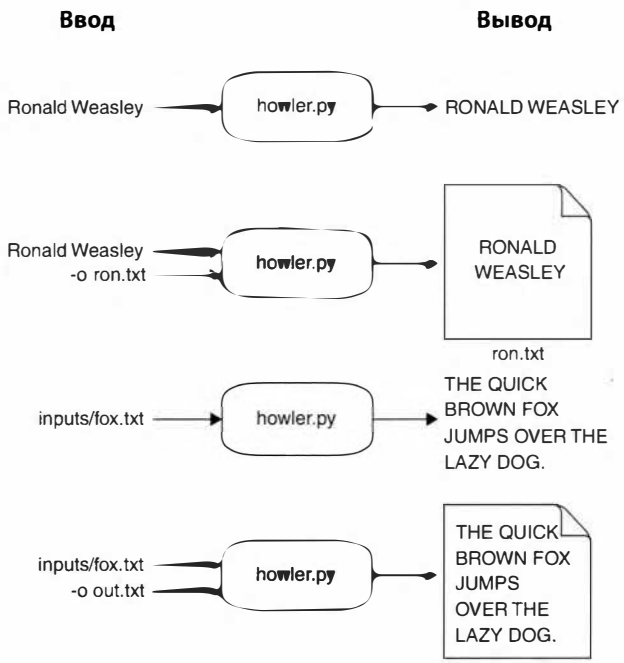


Рис. 5.5. Диаграмма демонстрирует, что программа *howler.py* принимает в качестве входных данных текст или файлы и, вероятно, имя выходного файла

Если аргумент представляет собой обычную строку, ее следует выводить прописными буквами:

```
$ ./howler.py 'How dare you steal that car!'
HOW DARE YOU STEAL THAT CAR!
```

Если в качестве аргумента передано имя файла, прописными буквами выводится *его содержимое*:

```
$ ./howler.py ../inputs/fox.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Если указано имя выходного файла, текст прописными буквами записывается в него, а в `STDOUT` ничего не выводится:

```
$ ./howler.py -o out.txt ../inputs/fox.txt
$ cat out.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Несколько советов.

- Создайте программу с помощью файла *new.py* и измените код в разделе `get_args()` согласно своим инструкциям по использованию программы.
- Запустите тестирование и попробуйте успешно завершить первый тест, который обрабатывает текст из командной строки и выводит его прописными буквами в `STDOUT`.
- Следующий тест должен проверять, возможна ли запись вывода в указанный файл. Придумайте, как это реализовать.
- Третий тест предназначен для чтения данных из файла. Не пытайтесь пройти весь набор тестов сразу!
- Существует специальный файловый дескриптор, доступный всегда, — так называемый стандартный вывод (обычно `STDOUT`). Если вызвать функцию `print()` без аргумента `file`, по умолчанию применяется дескриптор `sys.stdout`. Для его использования не забудьте импортировать модуль `sys`.

Не ленитесь, сначала попробуйте написать программу и пройти все тесты, а потом уже читать решение. Если что-то не получается, отвлекитесь, выпейте пузырек оборотного зелья и повеселите своих друзей.

5.4. Решение

Привожу решение, которое успешно проходит все тесты. Оно довольно короткое, потому что с помощью Python можно весьма лаконично реализовывать самые крутые идеи.

```
#!/usr/bin/env python3
"""Программа-кричалка"""

import argparse
import os
import sys

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Howler (upper-case input)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text',
                        metavar='text',
                        type=str,
                        help='Input string or file')

    parser.add_argument('-o',
                        '--outfile',
                        help='Output filename',
                        metavar='str',
                        type=str,
                        default='')

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text).read().rstrip()

    return args
```

Строковый аргумент `text` может содержать имя файла.

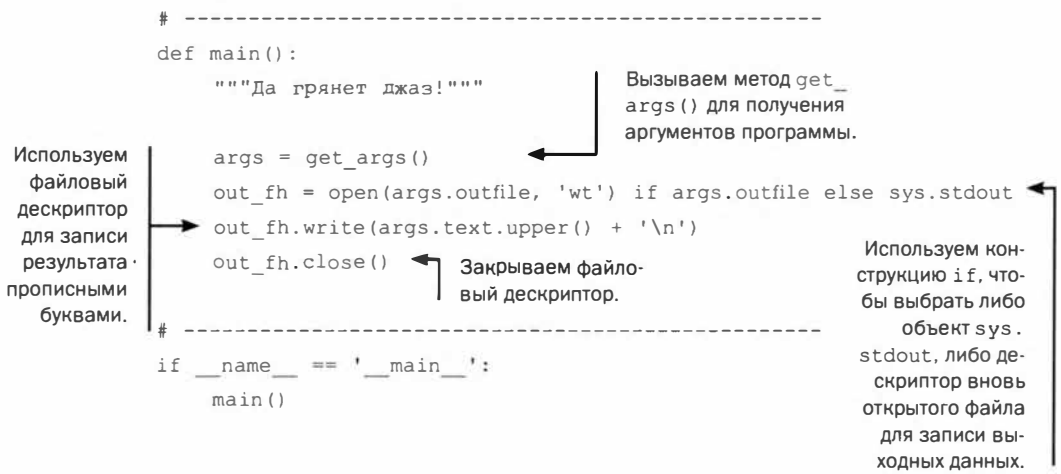
Строковый опциональный параметр `--outfile` также может содержать имя файла.

Анализируем аргументы командной строки в переменной `args`, чтобы можно было вручную проверить аргумент `text`.

Проверяем, содержит ли `args.text` имя существующего файла.

Если так, перезаписываем значение `args.text` содержимым файла.

Возвращаем аргументы вызывающей стороне.




```
# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout
    out_fh.write(args.text.upper() + '\n')
    out_fh.close()

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

5.5. Обсуждение

Как все прошло на этот раз? Надеюсь, вам все-таки не пришлось пробираться в кабинет Северуса Снегга. Вы же не хотите оказаться наказанными и провести очередную субботу, размышляя над своим поведением.

5.5.1. Определение аргументов

Функция `get_args()`, как всегда, впереди планеты всей. Тут я определяю два аргумента. Первый — позиционный аргумент `text`. Поскольку он может содержать как имя файла, так и текст, — данный аргумент содержит строковое значение.

```
parser.add_argument('text',
                    metavar='text',
                    type=str,
                    help='Input string or file')
```

Примечание. Если вы определяете несколько позиционных параметров, важно соблюдать их порядок *относительно друг друга.* Позиционный параметр, определенный первым, обрабатывает первый позиционный аргумент. Тут стоит упомянуть, что неважно, когда вы определяете позиционные параметры — до или после опций и флагов. Можно объявить их в любом порядке.

Второй аргумент опционален, и я присвоил ему короткое имя `-o` и длинное имя `--outfile`. Несмотря на то что по умолчанию все аргументы строковые, я предпочитаю указывать тип данных явно. Значение переменной `default` — пустая строка. Без проблем можно прибегнуть и к специальному типу `None`, который также является дефолтным, но я предпочитаю использовать определенный аргумент, в данном случае — пустую строку.

```
parser.add_argument('-o',
                    '--outfile',
                    help='Output filename',
                    metavar='str',
                    type=str,
                    default='')
```

5.5.2. Чтение входных данных из файла и командной строки

Наша на первый взгляд простая программа демонстрирует пару очень важных моментов, связанных с вводом и выводом в файл. При вводе переменная `text` может содержать как текст, так и имя файла. Такой способ представления данных будет неоднократно встречаться в книге:

```
if os.path.isfile(args.text):
    args.text = open(args.text).read().rstrip()
```

Функция `os.path.isfile()` помогает определить, содержится ли в тексте имя файла. Если она возвращает значение `True`, можно безопасно выполнить функцию `open(file)`, чтобы создать файловый дескриптор, содержащий метод `read` и возвращающий содержимое файла *полностью*.

Внимание! Учтите, что метод `fh.read()` возвращает содержимое файла *полностью* в виде одной строки. На вашем компьютере должен быть доступен объем памяти, превышающий размер файла. Во всех программах из этой книги используются небольшие файлы, поэтому у вас вряд ли возникнут проблемы. На практике же я часто оперирую гигабайтными файлами. Вызов метода `fh.read()` в таком случае способен при недостатке памяти привести к аварийному завершению программы, а то и всей системы.

Результатом выполнения `open(file).read()` является объект `str`, содержащий метод `str.rst rip()`. Он возвращает копию строки, *лишенную* любых пробельных символов с *правой* стороны (рис. 5.6).

Я вызываю его таким образом, чтобы передаваемый текст выглядел одинаково независимо от того, извлечен он из файла или непосредственно из командной строки. После ввода текста непосредственно в командной строке, вам нужно нажать клавишу **Enter**, чтобы завершить операцию. Нажатие данной клавиши передает символ перевода строки, и операционная система автоматически удаляет его перед передачей текста программе.

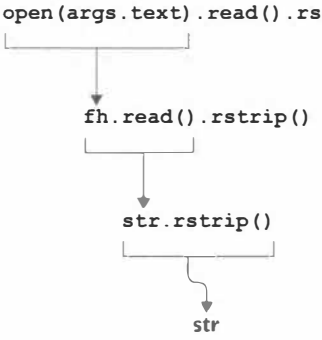


Рис. 5.6. Функция `open()` возвращает файловый дескриптор (`fh`). Функция `fh.read()` возвращает объект `str`, а функция `str.rstrip()` — новый объект `str` с удаленным пробельным символом с правой стороны. Перечисленные функции могут быть объединены

Более длинный способ решения представленной выше задачи выглядит так:

```
if os.path.isfile(text):
    fh = open(text)
    text = fh.read()
    text = text.rstrip()
    fh.close()
```

В своей версии я выполняю обработку в функции `get_args()`. Итак, можно перехватывать и изменять аргументы перед их передачей функции `main()`. Мы воспользуемся этим в следующих упражнениях.

Я привык проверять пользовательские аргументы функцией `get_args()`, хотя то же самое реализуемо в функции `main()` после вызова `get_args()`. Вопрос стиля.

5.5.3. Вывод данных в файл

Показанная ниже строка определяет, куда следует поместить выходные данные программы:

```
out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout
```

Конструкция `if` открывает `args.outfile` для записи текста (`wt`), если пользователь передал данный аргумент; в противном случае используется объект `sys.stdout` — файловый дескриптор для `STDOUT`. Обратите внимание, что мне не нужно вызывать функцию `open()` для объекта `sys.stdout`, потому что он доступен и открыт всегда (рис. 5.7).

Рис. 5.7. Конструкция `if` — лаконичное решение дилеммы. В нашем случае дескриптор выходного файла является результатом обработки аргумента `outfile`, если таковой передан; в противном случае используется файловый дескриптор `sys.stdout`

5.5.4. Вывод результата

Для написания текста прописными буквами мне понадобится метод `text.upper()`. Далее нужен способ, позволяющий передать его дескриптору выходного файла. Я поступил так:

```
out_fh.write(text.upper())
```

Как вариант, можно сделать следующее:

```
print(text.upper(), file=out_fh)
```

И, наконец, мне нужно закрыть файловый дескриптор с помощью метода `out_fh.close()`.

5.5.5. Версия для малых объемов памяти

В нашей программе притаилась потенциально серьезная проблема. С помощью метода `get_args()` мы считываем в память все содержимое файла:

```
if os.path.isfile(args.text):
    args.text = open(args.text).read().rstrip()
```

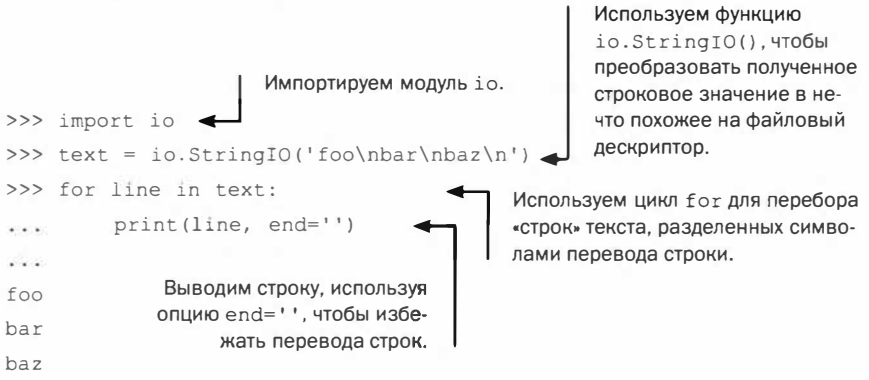
Вместо этого мы можем открыть файл:

```
if os.path.isfile(args.text):
    args.text = open(args.text)
```

А затем прочитать его содержимое построчно:

```
for line in args.text:
    out_fh.write(line.upper())
```

Проблема возникает в том случае, когда аргумент `text` содержит текст, а не имя файла. Модуль `io` (input/output — ввод/вывод) в Python позволяет представить текст в виде *потока*:



Перед вами пример, как можно обрабатывать обычное строковое значение подобно файловому дескриптору. Это полезный прием для тестирования кода, предназначенного для чтения файлов. Можно использовать результат выполнения метода `io.StreamIO()` в качестве «фиктивного» файлового дескриптора, чтобы считывать не *содержимое файла*, а значение, представляющее собой «строки» текста.

Для решения проблемы необходимо изменить способ обработки объекта `args.text`:

```
#!/usr/bin/env python3
"""Программа-кричалка для малых объемов памяти"""

import argparse
import os
import io
import sys
```

```
# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Howler (upper-cases input)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text',
                        metavar='text',
                        type=str,
                        help='Input string or file')

    parser.add_argument('-o',
                        '--outfile',
                        help='Output filename',
                        metavar='str',
                        type=str,
                        default='')

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text)
    else:
        args.text = io.StringIO(args.text + '\n')

    return args

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout
    for line in args.text:
        out_fh.write(line.upper())
    out_fh.close()

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Проверяем, является ли объект `args.text` файлом.

Если это так, заменяем `args.text` файловым дескриптором, создаваемым при открытии файла.

В противном случае заменяем `args.text` значением `io.StringIO()`, играющим роль дескриптора открытого файла. Обратите внимание на добавляемые к тексту символы перевода строки, позволяющие «представить» строки входных данных из реального файла.

Построчно считываем входные данные (`io.StringIO()` или файловый дескриптор).

Обрабатываем строку прежним методом.

5.6. Прокачиваем навыки

— Добавьте флаг, который преобразует входные данные в строковые символы. Пусть флаг называется `--ee` в честь Эдварда Эстлина Каммингса — поэта, писавшего стихи сплошь строчными буквами.
— Измените код программы таким образом, чтобы она поддерживала обработку нескольких файлов одновременно. Смените опцию `--outfile` на `--outdir` и записывайте содержимое каждого входного файла в такие же отдельные файлы в выходном каталоге.

РЕЗЮМЕ

— Чтобы читать или записывать файлы, вы должны сначала открыть их с помощью функции `open()`.
— По умолчанию функция `open()` выполняется в режиме чтения файлов.
— Чтобы записать текстовый файл, функции `open()` нужно передать значение `'wt'` в качестве второго аргумента.
— При выполнении функции `write()` файловому дескриптору по умолчанию передаются текстовые данные. Можно использовать флаг `'b'`, если требуется передать двоичные.
— Модуль `os.path` поддерживает много полезных функций, например `os.path.isfile()`, проверяющую, существует ли файл с указанным именем.
— Поток `STDOUT` (стандартный вывод) доступен, благодаря специальному, всегда открытому файловому дескриптору `sys.stdout`.
— Функция `print()` принимает опциональный аргумент `file`, указывающий, куда следует поместить выходные данные. Этот аргумент может быть файловым дескриптором, например `sys.stdout` (по умолчанию), либо результатом выполнения функции `open()`.

Глава 6

Подсчет слов: чтение файлов и потоки STDIN, итерирование списков, форматирование строк

«Я люблю считать!»
— Граф фон Знак

Подсчитывание — важная составляющая программирования. Например, вам может понадобиться вычислить количество продаваемых ежеквартально пицц или узнать, сколько раз встречается какое-то слово в пачке документов. Обычно данные, с которыми мы имеем дело в вычислениях, поступают к нам в виде файлов, поэтому в данной главе мы немного углубимся в чтение файлов и управление строками.

Мы напишем на Python версию почтенной UNIX-утилиты `wc` (word count — количество слов). Программа получит имя *wc.py* и займется подсчетом строк, слов и байтов входящих данных, поступающих в виде одного или нескольких позиционных аргументов. Результаты подсчета будут отображаться в столбцах шириной восемь символов, а следом

выводиться имя файла. К примеру, вот такой результат должна дать нам программа *wc.py* в случае анализа содержимого одного файла:

```
$ ./wc.py ../inputs/scarlet.txt
    7035   68061  396320 ../inputs/scarlet.txt
```

При анализе содержимого нескольких файлов появится дополнительная строка `total`, суммирующая значения в каждом столбце:

```
$ ./wc.py ../inputs/const.txt ../inputs/sonnet-29.txt
     865    7620   44841 ../inputs/const.txt
      17     118     661 ../inputs/sonnet-29.txt
     882    7738   45502 total
```

Кроме того, программа может быть запущена *без* аргументов. В таком случае мы будем считывать из *стандартного входа*, потока, который часто обозначается как `STDIN`. В предыдущей главе, когда мы использовали файловый дескриптор `sys.stdout`, я упоминал поток `STDOUT`. `STDIN` дополняет `STDOUT` и играет роль стандартного потока для чтения ввода из командной строки. Если программе не переданы *позиционные* аргументы, она будет считывать их из объекта `sys.stdin`.

`STDIN` и `STDOUT` — это универсальные файловые дескрипторы, поддерживаемые многими консольными программами. Мы можем связать поток `STDOUT` одной программы с потоком `STDIN` другой для создания специальных программ. Например, программа `cat` выводит содержимое файла в поток `STDOUT`. Мы можем использовать оператор конвейера (`|`) для направления этого вывода на вход программе через поток `STDIN`:

```
$ cat ../inputs/fox.txt | ./wc.py
       1       9      45 <stdin>
```

Еще вариант — использовать оператор `<` для перенаправления ввода из файла:

```
$ ./wc.py < ../inputs/fox.txt
       1       9      45 <stdin>
```

Один из самых удобных инструментов командной строки — это `grep`, который ищет в файлах текст по шаблону. Если, например, мы хотим найти все строки текста со словом `scarlet` во всех файлах в каталоге *input*, мы можем использовать следующую команду:

```
$ grep scarlet ../inputs/*.txt
```

Символ звездочки (*) в команде — это подстановочный знак, который предполагает любое значение, поэтому сочетание `*.txt` соответствует файлам с расширением *.txt* и с любым именем. Если выполнить данную команду, вы увидите довольно длинный вывод.

Чтобы подсчитать строки, найденные `grep`, мы можем направить этот вывод нашей программе *wc.py* следующим образом:

```
$ grep scarlet ../inputs/*.txt | ./wc.py
      108     1192     9201 <stdin>
```

Проверяем результат с помощью программы `wc`:

```
$ grep scarlet ../inputs/*.txt | wc
      108     1192     9201
```

Освоив данную главу, вы научитесь:

— работать с произвольным количеством аргументов;
— проверять входные файлы;
— считывать содержимое файлов и стандартного ввода;
— работать с уровнями циклов `for`;
— разделять содержимое файлов на строки, слова и байты;
— применять переменные-счетчики;
— форматировать выводимые строки.

6.1. Создание файла wc.py

Приступим! Создайте программу с именем *wc.py* в каталоге *06_wc* и измените аргументы таким образом, чтобы при запуске с флагами `-h` и `--help` программа выводила следующие инструкции:

```
$ ./wc.py -h
usage: wc.py [-h] [FILE [FILE ...]]

Emulate wc (word count)

positional arguments:
  FILE      Input file(s) (default: [<_io.TextIOWrapper name='<stdin>'
                mode='r' encoding='UTF-8'>])
```

```
optional arguments:
-h, --help  show this help message and exit
```

В случае отсутствия указанного файла программа должна выводить сообщение об ошибке и завершать работу:

```
$ ./wc.py blargh
usage: wc.py [-h] [FILE [FILE ...]]
wc.py: error: argument FILE: can't open 'blargh': \
[Errno 2] No such file or directory: 'blargh'
```

На рис. 6.1 показана диаграмма, отражающая функционал программы.

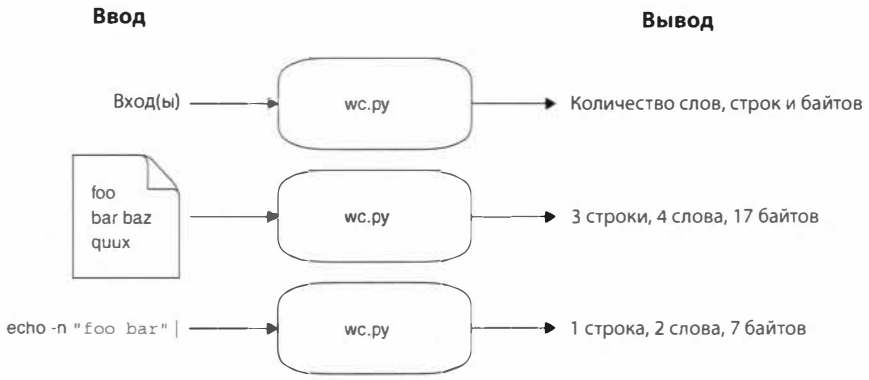


Рис. 6.1. Согласно диаграмме, программа *wc.py* считывает один или более входных файлов или опционально поток STDIN и выдает количество слов, строк и байтов, содержащихся в каждом фрагменте входных данных

6.1.1. Определение входных данных

Давайте разберемся, как определить параметры программы с помощью модуля `argparse`. Наша программа работает с произвольным количеством аргументов — с одним, с несколькими или вовсе без них. Напомню, что вам не нужно определять аргументы `-h` и `--help`, так как модуль `argparse` обрабатывает их автоматически.

В главе 3 мы использовали аргумент `nargs='+'`, чтобы обозначить один или более предметов для пикника. Здесь мы можем прибегнуть к аргументу `nargs='*'` для обозначения выражения «*ноль* или более». При отсутствии аргументов значением по умолчанию будет `None`. В данной программе мы планируем при отсутствии аргументов считывать поток `STDIN`.

Все допустимые значения переменной `nargs` перечислены в таблице 6.1.

Таблица 6.1. Допустимые значения переменной `nargs`

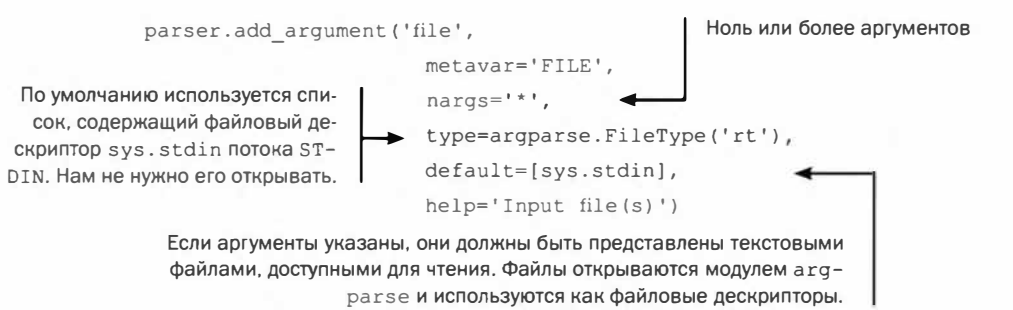
Символ	Обозначение
`?`	Ноль или один
`*`	Ноль или более
`+`	Один или более

Любые аргументы, передаваемые нашей программе, *должны быть файлами, доступными для чтения*. В главе 5 с помощью метода `os.path.isfile()` вы научились проверять, представлен ли входной аргумент файлом, а не текстом.

В данном случае входные аргументы должны быть представлены текстовыми файлами, доступными для чтения, поэтому мы можем определить аргументы, используя код `type=argparse.FileType('rt')`. Таким образом модуль `argparse` самостоятельно проверяет входные данные, передаваемые пользователем, и генерирует полезные сообщения об ошибках. Если пользователь передает допустимые данные, модуль `argparse` предоставляет список дескрипторов *открытых файлов*. Как можно догадаться, мы экономим немало времени. (Обязательно изучите раздел A.4.6 приложения.)

В главе 5 мы использовали файловый дескриптор `sys.stdout` для записи в поток `STDOUT`. Чтобы считать ввод из потока `STDIN`, мы прибегнем к файловому дескриптору `sys.stdin`. Как и `sys.stdout`, он не нуждается в открытии с помощью функции `open()`, так как открыт и доступен для чтения всегда.

Поскольку для определения аргумента мы применяем конструкцию `nargs='*'`, в результате мы получаем список. Чтобы дескриптор `sys.stdin` использовался по умолчанию, нужно поместить его в список следующим образом:



6.1.2. Итерирование списков

В программе есть список файловых дескрипторов, которые необходимо обработать. В главе 4 мы использовали цикл `for` для перебора символов во входных текстовых данных. Теперь мы можем применить его для итерирования `args.file`, содержащего дескрипторы открытых файлов:

```
for fh in args.file:
    # считываем каждый файл
```

Можно присваивать любые имена переменным в циклах `for`, но я считаю, что очень важно использовать семантически значимые имена. В нашем случае имя переменной `fh` указывает, что это открытый файловый дескриптор. В главе 5 вы научились открывать и считывать файлы с помощью функций `open()` и `read()`. В данном примере дескриптор `fh` уже открыт, поэтому мы можем сразу воспользоваться им для чтения содержимого.

Существует много способов прочитать содержимое файла. Метод `fh.read()` передает *полное содержимое* файла одним махом. Если файл большой — его размер превышает доступный объем памяти на вашем компьютере, — программа завершит работу аварийно. Поэтому я бы порекомендовал воспользоваться для `fh` другим циклом `for`. Так вы сможете считывать строки обрабатываемого файла по очереди:

```
for fh in args.file: # ОДИН ЦИКЛ!
    for line in fh:  # ДВА ЦИКЛА!
        # обрабатываем строку
```

Это двухуровневый цикл `for`: первый цикл служит для обработки файлового дескриптора, вложенный цикл — для обработки строк в файловом дескрипторе. ОДИН ЦИКЛ! ДВА ЦИКЛА! Я ЛЮБЛЮ СЧИТАТЬ!

6.1.3. Выполнение подсчета

Результатом анализа каждого файла выступает количество строк, слов и байтов (с учетом обычных и пробельных символов), причем каждое значение выводится в столбце шириной восемь символов, за которым следует пробел, а затем имя файла, доступное благодаря `fh.name`.

Давайте проанализируем вывод стандартной утилиты `wc`. Обратите внимание, что в данном примере она запускается с одним аргументом и выполняет подсчет только для указанного файла:

```
$ wc fox.txt
       1       9      45 fox.txt
```

Файл *fox.txt* маленький, поэтому вы сможете легко проверить, что он действительно содержит 1 строку, 9 слов и 45 байтов, включая обычные и пробельные символы, в том числе и завершающего символа перевода строки (рис. 6.2).

При запуске с несколькими файлами утилита `wc` помимо прочего выводит строку `total`:

```
$ wc fox.txt sonnet-29.txt
       1       9      45 fox.txt
      17     118     669 sonnet-29.txt
      18     127     714 total
```

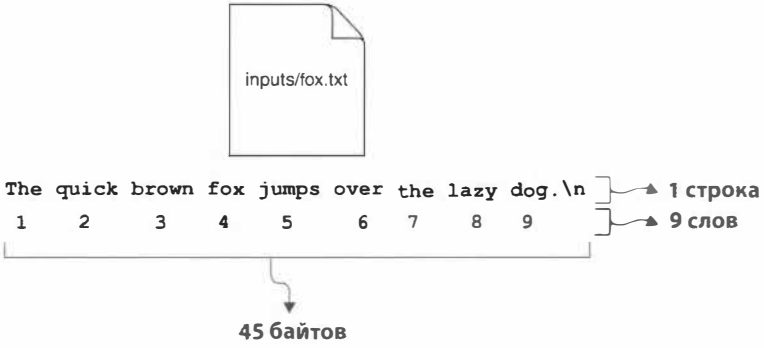


Рис. 6.2. Файл fox.txt содержит 1 текстовую строку, 9 слов и «весит» 45 байтов

Мы эмулируем поведение данной программы. Для каждого файла нужно создать переменные для хранения количества строк, слов и байтов. Если вы используете цикл `for line in fh`, вам понадобится переменная типа `num_lines`, значение которой инкрементируется на каждой итерации.

Проще говоря, сначала вы присваиваете ей значение 0, а затем, уже в цикле `for`, оно увеличивается на 1.

Программисты на Python применяют оператор +=, позволяющий добавить некое значение, указанное справа, к значению переменной слева (как показано на рис. 6.3):

```
num_lines = 0
for line in fh:
    num_lines += 1
```

Кроме того, вам нужно подсчитать количество слов и байтов, поэтому вам понадобятся переменные в духе `num_words` и `num_bytes`.

Чтобы подсчитать слова, мы воспользуемся функцией `str.split()`, позволяющей разбивать строки на слова по пробелам. Затем вы определяете количество слов по длине получившегося списка. Для вычисления количества байтов можно применить функцию `len()` (length — длина) к строке и добавить результат к значению переменной `num_bytes`.

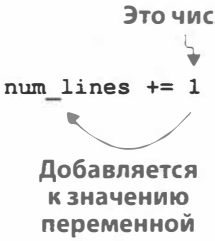


Рис. 6.3. Оператор += добавляет значение, указанное справа, к значению переменной, указанной слева

Примечание. Разбивка строк на слова по пробелам не позволяет оперировать словами в прямом смысле, так как в процессе от букв не отделяются знаки препинания, например запятые и точки. Однако для нашей программы такой вариант приемлем. В главе 15 мы рассмотрим регулярные выражения для выделения из строк именно слов.

6.1.4. Форматирование результатов

Перед вами первое упражнение, в котором выходные данные необходимо отформатировать определенным образом. Даже не пытайтесь решить данную задачу вручную — сойдете с ума. Бросьте. Вместо этого изучите магический метод `str.format()`. Документация к нему мало чем поможет, поэтому рекомендую ознакомиться с предложением PEP 3101 об усовершенствованном форматировании строк (**www.python.org/dev/peps/pep-3101**).

Метод `str.format()` предполагает использование фигурных скобок ({})для подстановки значений, передаваемых в качестве аргументов. Например, в нашей власти вывести необработанное значение `math.pi` следующим образом:

```
>>> import math
>>> 'Pi is {}'.format(math.pi)
'Pi is 3.141592653589793'
```

Форматирующие инструкции можно добавить после двоеточия (:), указав, как следует вывести значение. Если вы знакомы с функцией `printf()` из C-подобных языков, то тут суть та же. Например, я могу

вывести значение `math.pi` с двумя знаками после запятой, указав инструкцию `0.02f`:

```
>>> 'Pi is {:0.02f}'.format(math.pi)
'Pi is 3.14'
```

В предыдущем примере двоеточие (:) обозначает форматирующие инструкции, среди которых код `0,02f` указывает на использование двух знаков после запятой.

Вы также можете воспользоваться f-строкой, в которой переменная помещается *перед* двоеточием:

```
>>> f'Pi is {math.pi:0.02f}'
'Pi is 3.14'
```

В упражнении из данной главы вам нужно использовать форматирующую инструкцию `{:8}` для выравнивания всех строк, слов и символов по столбцам. Число `8` определяет ширину столбца. Текст выравнивается по левому краю:

```
>>> '{:8}'.format('hello')
'hello   '
```

А числа по правому краю:

```
>>> '{:8}'.format(123)
'     123'
```

Добавьте пробел между последним столбцом и именем файла, хранящимся в `fh.name`.

Приведу еще несколько советов.

— Начните с кода из файла new.py и удалите все непозиционные аргументы.
— Используйте аргумент `nargs='*'` для обозначения выражения «ноль или более» позиционных аргументов для аргумента `file`.
— Проходите тесты по очереди. Напишите программу, настройте вывод инструкций, пройдите первый тест, затем второй и т. д.
— Сравните код своей программы с кодом стандартной утилиты `wc` на вашем компьютере. Обратите внимание, что версии программы `wc` на разных компьютерах могут отличаться.

Традиционный совет: попробуйте сначала написать программу самостоятельно, прежде чем читать решение. Страх убивает разум. Все в ваших силах!

6.2. Решение

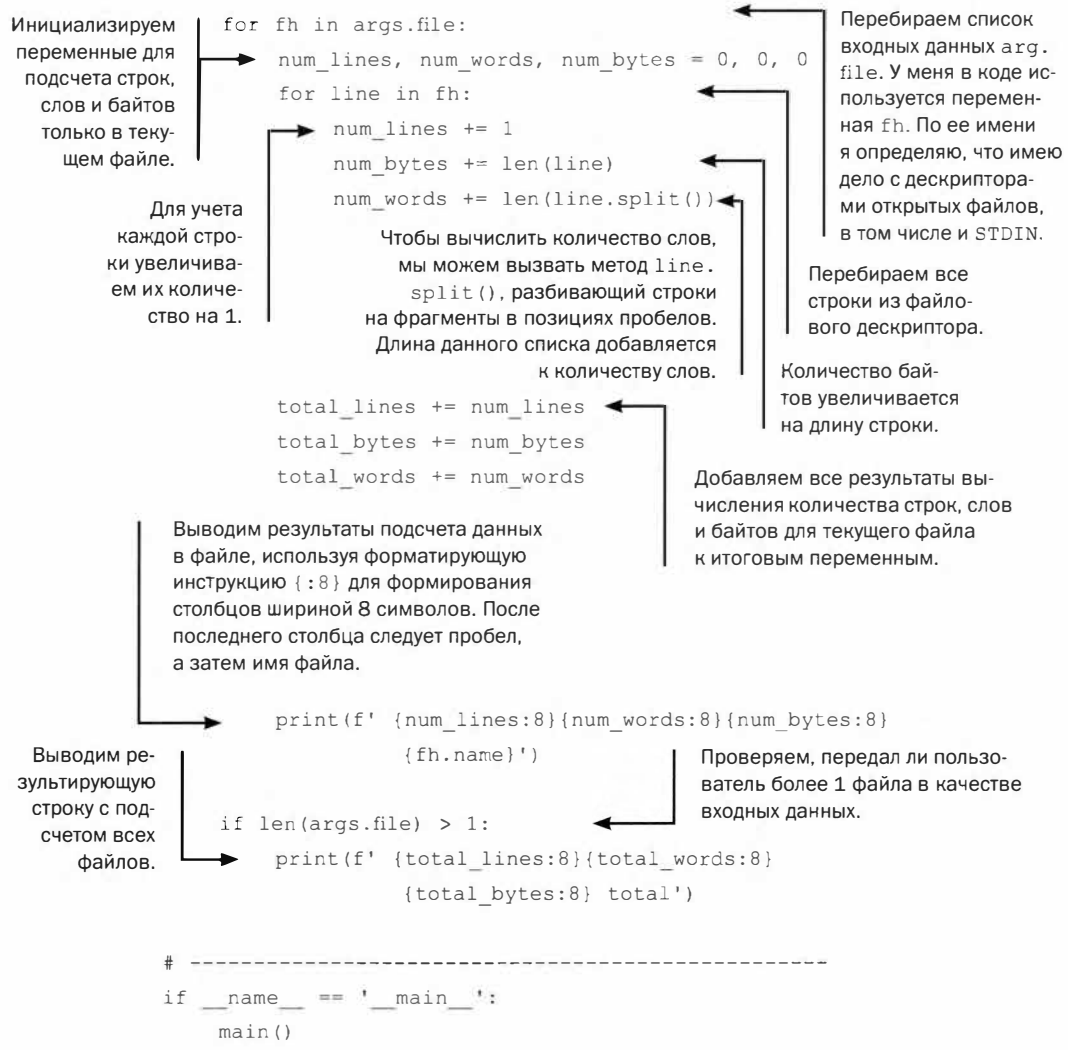
Перед вами один из вариантов, удачно проходящий тестирование. Напоминаю, что, если вы решили задачу по-другому, не стоит беспокоиться. Главное, чтобы программа работала, проходила тесты и чтобы вы понимали свой код.

```
#!/usr/bin/env python3
"""Эмулятор программы wc (подсчет слов)"""

import argparse
import sys

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Emulate wc (word count)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        nargs='*',
                        default=[sys.stdin],
                        type=argparse.FileType('rt'),
                        help='Input file(s)')

    return parser.parse_args()

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()

    total_lines, total_bytes, total_words = 0, 0, 0
```

Для работы с потоком STDIN по умолчанию устанавливаем список с помощью значения sys.stdin

Если пользователь передает аргументы, модуль argparse проверяет переданные файлы на допустимость. В случае возникновения проблемы модуль argparse прекращает выполнение программы и выводит сообщение об ошибке.

Данные переменные используются для вывода общего результата, если аргументов несколько.




```
# --------------------------------------------------
if __name__ == '__main__':
    main()
```

6.3. Обсуждение

Код программы небольшой и кажется простым, но это не совсем так. Разберем основные концепции.

6.3.1. Определение аргументов

В процессе выполнения данного упражнения вы познакомились с модулем `argparse` и узнали о его возможностях, упрощающих написание

кода. Основная идея заключается в определении параметра file. В программе используется строка type=argparse.FileType('rt'), с помощью которой мы указываем, что предоставляемые аргументы должны быть текстовыми файлами, доступными для чтения. Строка nargs ='*' применяется для обозначения выражения «ноль или более» аргументов, а переменной default в качестве значения присваивается список, содержащий дескриптор sys.stdin. Таким образом модуль argparse всегда будет предоставлять список с одним или несколькими файловыми дескрипторами.

В небольшом коде реализовано много алгоритмов. Прекрасно, что львиная доля задач по проверке входных данных, генерации сообщений об ошибках и обработке дефолтных значений выполняется за нас!

6.3.2. Чтение файлов с помощью цикла for

Модуль argparse возвращает args.file в список *файловых дескрипторов*. В REPL-интерфейсе мы тоже можем сделать такой список, чтобы имитировать значение args.file:

```
>>> files = [open('../inputs/fox.txt')]
```

Прежде чем использовать цикл for для итерации, необходимо создать три переменные для отслеживания *общего* количества строк, слов и символов. Мы можем определить их тремя отдельными строками:

```
>>> total_lines = 0
>>> total_words = 0
>>> total_bytes = 0
```

Или объявить в одной строке, как показано ниже:

```
>>> total_lines, total_words, total_bytes = 0, 0, 0
```

Технически с правой стороны получается *кортеж*, так как мы размещаем запятые между тремя нолями и «распаковываем» их в три переменные с левой стороны. О кортежах я расскажу чуть позже.

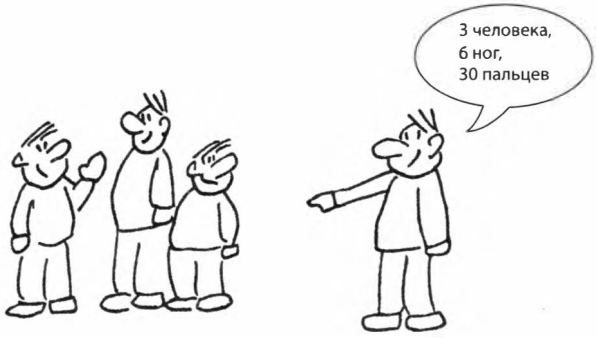
В цикле for для каждого файлового дескриптора мы инициализируем три дополнительные переменные для хранения количества строк, символов и слов *обрабатываемого файла*. Затем используем отдельный цикл for для перебора строк в файловом дескрипторе (fh). При подсчете строк мы можем добавлять 1 при каждой итерации цикла for.

При подсчете байтов — использовать длину строки (`len(line)`), чтобы отслеживать количество «символов» (которые могут быть как печатными, так и пробельными, поэтому удобнее называть их «байтами»). Наконец, при подсчете слов мы можем применить метод `line.split()`, позволяющий разбивать строку по пробельным символам, чтобы создать список «слов». Это не идеальный способ подсчета слов, хотя и достаточно точный. Мы можем использовать функцию `len()` на списке для определения значения переменной `words`.

Цикл `for` завершается при достижении конца файла. Затем наступает черед вывода (`print()`) результатов подсчета и имени файла с помощью форматирующей инструкции с плейхолдерами `{:8}` для получения столбцов шириной 8 символов:

```
>>> for fh in files:
...     lines, words, bytes = 0, 0, 0
...     for line in fh:
...         lines += 1
...         bytes += len(line)
...         words += len(line.split())
...     print(f' {lines:8}{words:8}{bytes:8} {fh.name}')
...     total_lines += lines
...     total_bytes += bytes
...     total_words += words
...
       1       9      45 ../inputs/fox.txt
```

Обратите внимание, что показанный вызов функции `print()` выполняется со *вторым циклом* `for`, поэтому осуществляется после того, как заканчивается перебор строк в `fh`. Я решил использовать f-строки для вывода значений переменных `lines`, `words` и `bytes` в столбцах шириной восемь символов, после которых следует один пробел, а затем `fh.name` (имя) файла.



Далее мы можем сложить значения переменных `lines`, `words` и `bytes`, чтобы получить общее количество строк, символов и слов предоставленных файлов.

Если количество аргументов больше 1, мы выводим итог:

```
if len(args.file) > 1:
    print(f' {total_lines:8}{total_words:8}{total_bytes:8} total')
```

6.4. Прокачиваем навыки

— По умолчанию утилита `wc` выводит столбцы с текстом, как и наша программа, а также принимает флаги команды вывода: `-c` для подсчета символов, `-l` — для подсчета строк и `-w` — для подсчета слов. При использовании перечисленных флагов отображаются только соответствующие столбцы, поэтому команда `wc.py -wc`, к примеру, выведет результаты подсчета только слов и символов. Добавьте подобные короткие и длинные флаги в свою программу, чтобы имитировать схожее с `wc` поведение.

— Напишите собственные реализации других системных инструментов, таких как `cat` (для вывода содержимого файла в `STDOUT`), `head` (для вывода первых *n* строк файла), `tail` (для вывода последних *n* строк файла) и `tac` (для вывода строк файла в обратный порядок).

РЕЗЮМЕ

— Параметр `nargs` (количество аргументов) модуля `argparse` позволяет проверить количество аргументов, передаваемых пользователем. Звездочка (`'*'`) означает «ноль или более», а символ «плюс» (`'+'`) — «один или более».

— Если определять аргумент методом `type=argparse.FileType('rt')`, модуль `argparse` проверяет, что пользователь предоставил текстовый файл, доступный для чтения, и представляет значение в качестве файлового дескриптора.

— Можно считывать/записывать данные из стандартных файловых дескрипторов ввода/вывода, используя потоки `sys.stdin` и `sys.stdout`.

— Допустимо вкладывать циклы `for` для реализации нескольких уровней обработки данных.

— Метод `str.split()` разбивает строки по пробелам.

— Функция `len()` может использоваться как для обработки строк, так и для обработки списков. В последнем случае она выводит количество элементов, содержащихся в списке.
— Метод `str.format()` и f-строки распознают синтаксис форматирования `printf`, позволяя выводить значения в желаемом формате.

Глава 7

Ужасная азбука: поиск в словаре

В данной главе мы будем искать в переданном файле текстовые строки, начинающиеся с букв, введенных пользователем. Для наших упражнений отлично подойдет текст книги Эдварда Гори, азбуки, описывающей различные ужасные причины гибели детей*. Ниже на рис. 7.1 показана иллюстрация из книги к букве H («H — это Невилл, почивший с тоски»).

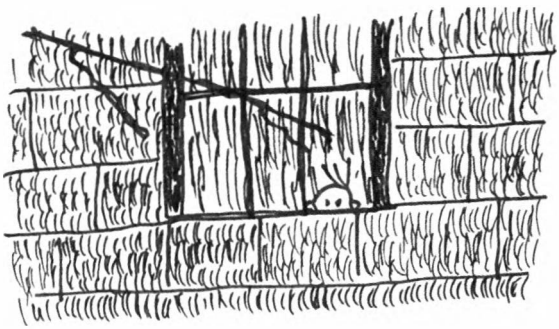


Рис. 7.1. H — это Невилл, почивший с тоски

* Речь о книге The Gashlycrumb Tinies, она представляет собой иллюстрированную азбуку в стихах в духе страшилок, где буквам английского алфавита соответствуют первые буквы имен детей. *Прим. ред.*

Программа *gashlycrumb.py* должна принимать одну и более букв в качестве позиционных аргументов и искать строки текста, начинающиеся с этой буквы (или букв), в *опциональном* входном файле. Поиск букв осуществляется без *учета регистра.*

Каждая строка входного файла начинается с буквы следующим образом:

```
$ head -2 gashlycrumb.txt
A is for Amy who fell down the stairs.
B is for Basil assaulted by bears.
```

Когда несчастный пользователь запустит нашу программу, он увидит:

```
$ ./gashlycrumb.py e f
E is for Ernest who choked on a peach.
F is for Fanny sucked dry by a leech.
```

В данном упражнении вам предстоит:

— принимать один или более позиционных аргументов, которые мы назовем `letter`;
— принимать опциональный аргумент `--file`, представляющий собой текстовый файл, доступный для чтения. Значение по умолчанию — `gashlycrumb.txt` (прилагается);
— считывать содержимое файла, находить первую букву каждой строки и выстраивать структуру данных, связывающую букву с соответствующей строкой текста (пригодны файлы, каждая строка в которых начинается с одной уникальной буквы, прочие текстовые файлы программа не поддерживает);
— для каждой буквы, предоставленной пользователем, выводить либо соответствующую букве строку текста, если такая буква обнаружена, либо сообщение об отсутствии совпадений;
— «красиво оформлять» вывод данных.

Здесь вам пригодится опыт, приобретенный в предыдущих главах.

— Из главы 2 вы узнали, как извлечь первую букву строки.
— Из главы 4 — как создать словарь и искать в нем значения.

— Из главы 6 — как принимать файловый ввод и считывать его содержимое построчно.

Объединив полученные знания, вы сможете продекламировать текст азбуки!

7.1. Создание файла gashlycrumb.py

Прежде чем писать код, я рекомендую выполнить тесты с помощью команды `make test` или `pytest -xv test.py` в каталоге *07_gashlycrumb*. Первый тест окажется провальным:

```
test.py:: test_exists FAILED
```

Судя по всему, чтобы его пройти, вам нужно создать файл *gashlycrumb.py*. Сделать это можно несколькими способами, например, выполнив команду `new.py gashlycrumb.py` в каталоге *07_gashlycrumb*, скопировав содержимое файла *template/template.py* или создав файл вручную. Выполните тестирование еще раз. Как минимум должен быть пройден первый тест, а если ваша программа содержит справочную информацию, то, возможно, и второй.

Затем разберитесь с аргументами. Измените параметры программы в функции `get_args()` таким образом, чтобы при запуске с флагами `-h` и `--help` или без аргументов она выводила следующие инструкции:

```
$ ./gashlycrumb.py -h
usage: gashlycrumb.py [-h] [-f FILE] letter [letter ...]

Gashlycrumb

positional arguments:
  letter                Letter(s)

optional arguments:
-h, --help              show this help message and exit
-f FILE, --file FILE  Input file (default: gashlycrumb.txt)
```

`letter` — обязательный позиционный аргумент, который принимает одно или несколько значений.

`-f` и `--file` — опциональный аргумент с дефолтным значением `gashlycrumb.txt`.

Аргументы `-h` и `--help` создаются автоматически с помощью модуля `argparse`.

На рис. 7.2 показана диаграмма, демонстрирующая работу программы.

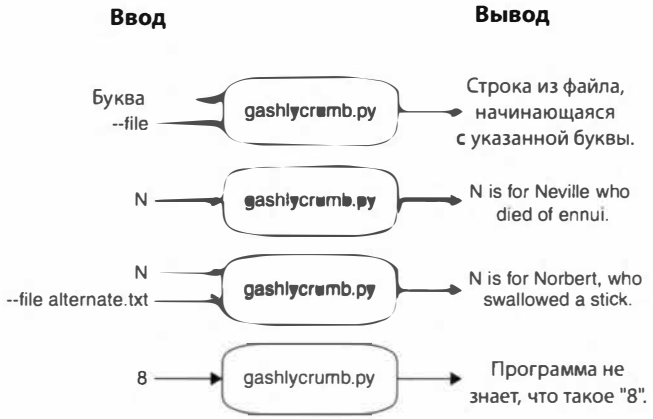


Рис. 7.2. Программа принимает некоторую букву (буквы) и опционально файл. Затем она ищет в файле текстовую строку (строки), начинающуюся с заданной буквы (или букв)

В методе `main()` начнем с вывода каждого аргумента `letter`:

```
def main():
    args = get_args()
    for letter in args.letter:
        print(letter)
```

Запустите программу, чтобы проверить, работает ли код:

```
$ ./gashlycrumb.py a b
a
b
```

Затем прочитайте содержимое файла построчно с помощью цикла `for`:

```
def main():
    args = get_args()
    for letter in args.letter:
        print(letter)

    for line in args.file:
        print(line, end='')
```

Обратите внимание, что в функции `print()` я использую опцию `end=''`, чтобы избежать перевода строк, так как в файле каждая строка и так начинается с новой строки:

Запустите программу и проверьте, считывается ли входной файл:

```
$ ./gashlycrumb.py a b | head -4
a

b
A is for Amy who fell down the stairs.
B is for Basil assaulted by bears.
```

Попробуйте «скормить» программе другой файл, например *alternate.txt*:

```
$ ./gashlycrumb.py a b --file alternate.txt | head -4
a

b
A is for Alfred, poisoned to death.
B is for Bertrand, consumed by meth.
```

Если программе передан несуществующий аргумент `--file`, она завершает работу с ошибкой и соответствующим сообщением. Обратите внимание, что при передаче методу `get_args()` параметра `type=argparse.FileType('rt')`, как мы делали это в предыдущей главе, ошибка обрабатывается автоматически модулем `argparse`:

```
$ ./gashlycrumb.py -f blargh b
usage: gashlycrumb.py [-h] [-f FILE] letter [letter ...]
gashlycrumb.py: error: argument -f/--file: can't open 'blargh': \
[Errno 2] No such file or directory: 'blargh'
```

Теперь придумайте, как использовать первую букву каждой строки для записи значения в словарь. Используйте функцию `print()` для просмотра его содержимого. Выясните, как проверить, присутствует ли данная буква в словаре.

Если программе передано значение, которого нет в списке первых символов строк входного файла (без учета регистра), следует вывести вот такое сообщение:

```
$ ./gashlycrumb.py 3
I do not know "3".
$ ./gashlycrumb.py CH
I do not know "CH".
```

В противном случае выводим соответствующее букве значение (рис. 7.3):

```
$ ./gashlycrumb.py a
A is for Amy who fell down the stairs.
$ ./gashlycrumb.py z
Z is for Zillah who drank too much gin.
```

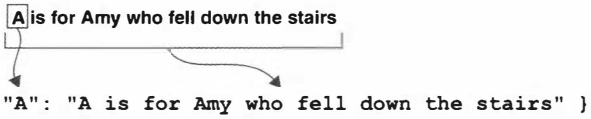


Рис. 7.3. Нам нужен словарь, в котором первая буква каждой строки является ключом, а строка — значением

Выполните тексты и проверьте, все ли из них программа проходит успешно. Внимательно проанализируйте ошибки, если таковые возникают, и внесите исправления в код.

Ниже я привел несколько советов.

- Сгенерируйте программу с помощью файла new.py и удалите лишний код за исключением позиционного аргумента `letter` и опционального параметра `--file`.
- Используйте код `type=argparse.FileType('rt')`, чтобы выполнить проверку аргумента `--file`.
- С помощью кода `nargs='+'` определите позиционный аргумент `letter`, требующий одно или несколько значений.
- Словарь — это натуральная структура данных для связывания буквы-ключа, такой как «A», со значением-фразой типа «A — это Аня, что скатилась по лестнице». Создайте пустой словарь.
- Обладая файловым дескриптором, можно считывать содержимое файла построчно с помощью цикла `for`.
- Каждая строка текста представляет собой строку. Как извлечь первый символ строки?
- Создайте запись в словаре, используя первый символ в качестве ключа и саму строку в качестве значения.
- Переберите буквы аргумента. Как проверить, что заданное значение есть в словаре?

Прошу вас, не читайте дальше, пока не напишете собственную версию программы. Если заглянете в решение раньше времени, вас ожидает страшный конец... возможности порадоваться личной победе.

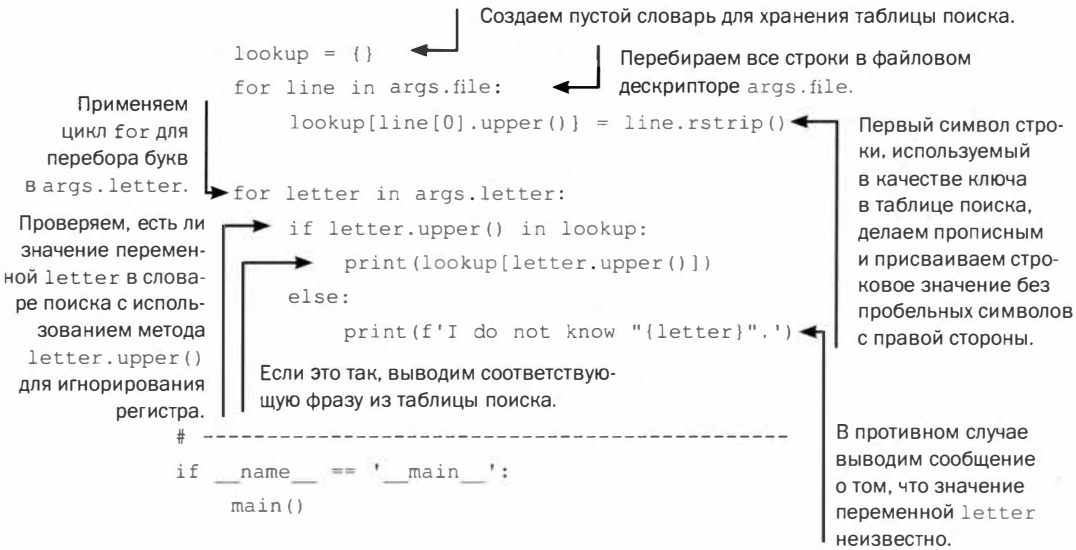
7.2. Решение

Ну что, изучили все иллюстрции Эдварда Гори к «кошмарной» азбуке? Теперь давайте обсудим, как создать словарь из файлового ввода:

```
#!/usr/bin/env python3
"""Таблицы поиска"""

import argparse

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Gashlycrumb',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('letter',
                        help='Letter(s)',,
                        metavar='letter',
                        nargs='+',
                        type=str)

    parser.add_argument('-f',
                        '--file',
                        help='Input file',
                        metavar='FILE',
                        type=argparse.FileType('rt'),
                        default='gashlycrumb.txt')
    return parser.parse_args()

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
```

Позиционный аргумент `letter` содержит код `nargs='+'`, поэтому принимаются один или более позиционных строковых аргументов.

Опциональный аргумент `--file` должен быть представлен файлом, доступным для чтения, на что указывает строка `type=argparse.FileType('rt')`. Значение по умолчанию — `gashlycrumb.txt`, файл, который точно существует.



7.3. Обсуждение

Ну что, выжили после написания нашей ужасной программы? Теперь послушайте, как я решил поставленную задачу. Напомню, что приведен один из нескольких возможных вариантов.

7.3.1. Обработка аргументов

Я предпочитаю помещать всю логику для парсинга и проверки аргументов командной строки в функцию `get_args()`. Модуль `argparse`, в числе прочего, отлично справляется с самостоятельной проверкой того, передан ли в качестве аргумента реально существующий текстовый файл, доступный для чтения. Поэтому я и использую код `type=argparse.FileType('rt')`. Если пользователь не передает допустимый аргумент, модуль `argparse` выбрасывает ошибку, выводит информационное сообщение вместе с краткими инструкциями, и завершает работу программы, сообщая код ошибки.

Добравшись до строки `args = get_args()`, я знаю, что есть один или более аргументов «буквы» и дескриптор поддерживаемого файла в `args.file`. В REPL-интерфейсе я могу использовать функцию `open()` для создания файлового дескриптора, которому обычно присваиваю имя `fh`. Чтобы соблюсти авторские права, я использую другой текстовый файл:

```
>>> fh = open('alternate.txt')
```

7.3.2. Чтение входного файла

Нам понадобится словарь, в котором ключами являются первые буквы каждой строки, а значениями — соответствующие ключам строки. Таким образом, необходимо начать с создания пустого словаря. Делается это либо с помощью функции `dict()`, либо с определения переменной, которой присваивается пара фигурных скобок (`{}`). Взгляните на вызов переменной `lookup`:

```
>>> lookup = {}
```

Можно применить цикл `for` для чтения каждой строки текстового файла. Создавая программу в главе 2, вы узнали про метод `line[0].upper()`, позволяющий извлечь первую букву строки и преобразовать ее в прописную. Мы можем использовать это значение как ключ в таблице поиска.

Каждая строка текста заканчивается символом перевода строки, который нужно удалить. Метод `str.rstrip()` ликвидирует пробельный символ с правой стороны строки (rstrip — это *right strip*, то есть *обрезка справа)*. В результате получается следующее:

```
for line in fh:
    lookup[line[0].upper()] = line.rstrip()
```

Давайте взглянем на результирующий словарь `lookup`. Мы можем вывести результат с помощью функции `print()` из программы или с помощью команды `lookup` в REPL-интерфейсе, однако он не будет отформатирован. Попробуйте.

К счастью, существует прекрасный модуль `pprint` для «форматирования вывода» структур данных. Ниже показано, как импортировать из него функцию `pprint()` с псевдонимом `pp`:

```
>>> from pprint import pprint as pp
```

На рис. 7.4 показан синтаксис.

Рис. 7.4. Мы можем выбирать, какие функции импортировать из модуля, и даже присваивать им псевдонимы

Теперь взглянем на таблицу поиска:

```
>>> pp(lookup)
{'A': 'A is for Alfred, poisoned to death.',
 'B': 'B is for Bertrand, consumed by meth.',
 'C': 'C is for Cornell, who ate some glass.',
 'D': 'D is for Donald, who died from gas.',
 'E': 'E is for Edward, hanged by the neck.',
 'F': 'F is for Freddy, crushed in a wreck.',
 'G': 'G is for Geoffrey, who slit his wrist.',
 'H': "H is for Henry, who's neck got a twist.",
 'I': 'I is for Ingrid, who tripped down a stair.',
 'J': 'J is for Jered, who fell off a chair,',
 'K': 'K is for Kevin, bit by a snake.',
 'L': 'L is for Lauryl, impaled on a stake.',
 'M': 'M is for Moira, hit by a brick.',
 'N': 'N is for Norbert, who swallowed a stick.',
 'O': 'O is for Orville, who fell in a canyon,',
 'P': 'P is for Paul, strangled by his banyan,',
 'Q': 'Q is for Quintanna, flayed in the night,',
 'R': 'R is for Robert, who died of spite,',
 'S': 'S is for Susan, stung by a jelly,',
 'T': 'T is for Terrange, kicked in the belly,',
 'U': "U is for Uma, who's life was vanquished,",
 'V': 'V is for Victor, consumed by anguish,',
 'W': "W is for Walter, who's socks were too long,",
 'X': 'X is for Xavier, stuck through with a prong,',
 'Y': 'Y is for Yoeman, too fat by a piece,',
 'Z': 'Z is for Zora, smothered by a fleece.'}
```

Похоже на удобную структуру данных. Прекрасная работа! Помните о преимуществах использования большого количества вызовов функции `print()`, когда пишете и разбираете программу, и плюсах функции `pprint()`, если нужно вывести сложную структуру данных.

7.3.3. Представление словаря

В главе 4 вы использовали представление списка для построения списка, поместив цикл `for` в квадратные скобки `[]`. Если мы изменим квадратные скобки на фигурные (`{}`), то получим «представление словаря»:

```
>>> fh = open('gashlycrumb.txt')
>>> lookup = {line[0].upper(): line.rstrip() for line in fh}
```

Взгляните на рис. 7.5, демонстрирующий, как можно организовать три строки цикла `for` в одну строку кода.

Если вывести таблицу поиска снова, вы увидите тот же результат, что и раньше. Не думайте, что писать одну строку кода вместо трех — это показуха, ведь на самом деле компактный идиоматический код — вещь очень и очень полезная. Чем больше кода в программе, тем выше риск появления ошибок. Помня об этом, я обычно стараюсь писать как можно более лаконичный код (но не более примитивный).

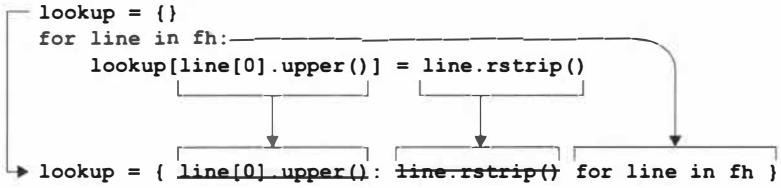
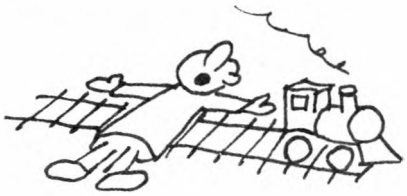


Рис. 7.5. Цикл `for`, который мы использовали для создания словаря, может быть написан с учетом представления словаря

7.3.4. Поиск по словарю

Теперь, имея таблицу поиска, я могу узнать, связаны ли с ключами какие-либо значения. В качестве ключей используются прописные буквы, и, поскольку пользователь может ввести строчную букву, я прибегаю к методу `letter.upper()`, позволяющему решить данную проблему:



```
>>> letter = 'a'
>>> letter.upper() in lookup
True
>>> lookup[letter.upper()]
'A is for Amy who fell down the stairs.'
```

Если буква обнаружена, я могу вывести связанную с ней строку текста; в противном случае — вывести сообщение о том, что введенный пользователем символ неизвестен:

```
>>> letter = '4'
>>> if letter.upper() in lookup:
...     print(lookup[letter.upper()])
... else:
...     print('I do not know "{}".'.format(letter))
...
I do not know "4".
```

Существует более короткий способ решить задачу — метод `dict.get()`:

```
def main():
    args = get_args()
    lookup = {line[0].upper(): line.rstrip() for line in args.file}

    for letter in args.letter:
        print(lookup.get(letter.upper(), f'I do not know
                         "{letter}".'))
```

Метод `lookup.get()` возвращает значение для `letter.upper()` или предупреждение о том, что значение в таблице поиска не найдено.

7.4. Прокачиваем навыки

— Напишите приложение с контактами, считывающее содержимое файла и создающее словарь из имен ваших друзей и адресов их электронной почты или номеров телефонов.
— Создайте программу, в которой словарь используется для подсчета количества вхождений каждого слова в документе.
— Напишите интерактивную версию программы, принимающую данные непосредственно от пользователя. С помощью конструкции `while True` создайте бесконечный цикл с функцией `input()` для запроса следующей буквы у пользователя:

```
$ ./gashlycrumb_interactive.py
    Please provide a letter [! to quit]: t
    T is for Titus who flew into bits.
    Please provide a letter [! to quit]: 7
    I do not know "7".
    Please provide a letter [! to quit]:!
    Bye
```

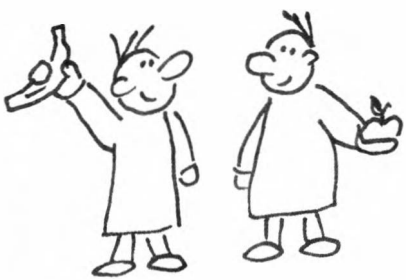
— Интерактивные программы писать интересно, но как вы будете их тестировать? В главе 17 я покажу один из вариантов.

РЕЗЮМЕ

— Представление словаря — это способ встроить словарь в однострочный цикл `for`.
— Определение входных аргументов файла с помощью метода `argparse.FileType` экономит время и сокращает код.
— Python-модуль `pprint` используется для форматирования вывода сложных структур данных.

Глава 8

Яблоки и бананы: поиск и замена



Вы когда-нибудь ошибались при наборе текста? Я — никогда, но знаю, что многие люди пишут с ошибками или опечатками. Компьютеры нам помогают находить и заменять слова с ошибками на правильные. Так можно, например, даже в оде любви заменить все упоминания имени бывшей на имя новой возлюбленной. «Поиск и замена» вам в руки.

Для начала давайте рассмотрим детскую фонетическую песенку "Apples and Bananas" («Яблоки и бананы»), состоящую из одной-единственной фразы, в которой постоянно меняются гласные:

```
I like to eat, eat, eat apples and bananas
```

В следующей строке гласные буквы заменяются долгим звуком «а» (как в слове hay):

```
I like to ate, ate, ate ay-ples and ba-nay-nays
```

Затем — сверхпопулярным долгим звуком «е» (как в слове knee):

```
I like to eat, eat, eat ee-ples and bee-nee-nees
```

И так далее. В данном упражнении мы напишем на Python программу *apples.py*, которая в качестве единственного позиционного аргумента принимает некоторый текст и заменяет все гласные в тексте подставляемой буквой (по умолчанию, `a`). Подставляемую букву можно изменить с помощью опции `-v` или `--vowel`.

Программа разместится в каталоге *08_apples_and_bananas* и будет обрабатывать текст в командной строке:

```
$ ./apples.py foo
faa
```

А также с учетом опции `-v` или `--vowel`:

```
$ ./apples.py foo -v i
fii
```

В данном случае должен *учитываться регистр* букв:

```
$ ./apples.py -v i "APPLES AND BANANAS"
IPPLIS IND BININIS
```

Как и в случае с «Кричалкой» из главы 5, в качестве аргумента может быть передан текстовый файл, и ваша программа должна суметь прочитать его содержимое:

```
$ ./apples.py ../inputs/fox.txt
Tha qaack brawn fax jamps avar tha lazy dag.
$ ./apples.py --vowel e ../inputs/fox.txt
The qeeck brewn fex jemps ever the lezy deg.
```

На рис. 8.1 показана диаграмма входных/выходных данных.

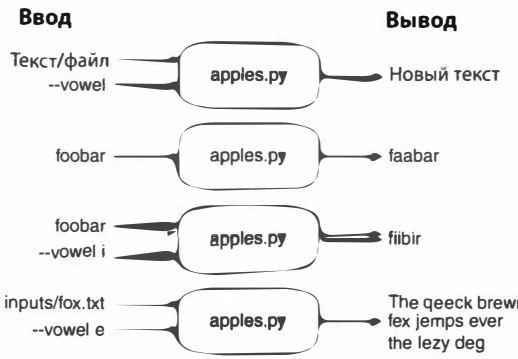


Рис. 8.1. Наша программа принимает текст и опционально гласную букву. Все гласные в данном тексте заменяются на одну и ту же гласную, что и делает песенку такой веселой

Ниже показаны инструкции, которые должна выводить программа при запуске без аргументов:

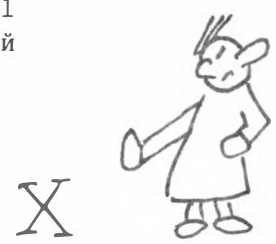
```
$ ./apples.py
usage: apples.py [-h] [-v vowel] text
apples.py: error: the following arguments are required: text
```

Кроме того, она должна выводить инструкции в случае использования флага `-h` или `--help`:

```
$ ./apples.py  -h
usage: apples.py [-h] [-v vowel] text

Apples and bananas

positional arguments:
  text                  Input text or file

optional arguments:
-h, --help            show this help message and exit
-v vowel, --vowel vowel
                        The vowel to substitute (default: a)
```

И ошибку, если в качестве аргумента `—vowel` передан символ, отличный от строчной гласной буквы:

```
usage: apples.py [-h] [-v str] str
apples.py: error: argument -v/--vowel: \
invalid choice: 'x' (choose from 'a', 'e',
                    'i', 'o', 'u')
```



Ваша программа должна уметь следующее:

— принимать позиционный аргумент, который может быть как обычным текстом, так и текстовым файлом;
— если в качестве аргумента передан файл, передавать его содержимое в качестве входных данных;
— принимать опциональный аргумент `-v` или `--vowel`, по умолчанию представленный буквой «a»;
— проверять, что значение `--vowel` соответствует одной из гласных из числа «a», «e», «i», «o» и «u»;

— заменять все гласные во входном тексте указанной с помощью аргумента `--vowel` буквой (или дефолтной «a»);
— выводить новый текст в поток `STDOUT`.

8.1. Изменение строк

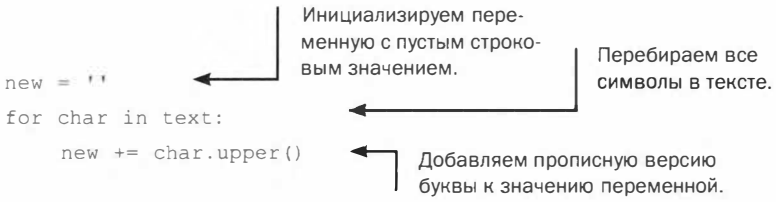
Работая, руководствуясь данной книгой, со строками, числами, списками и словарями Python, вы узнали, как *изменять* переменные. Проблема заключается в том, что *строки неизменяемы.* Предположим, что существует переменная `text`, которая содержит некоторый текст:

```
>>> text = 'The quick brown fox jumps over the lazy dog.'
```

Если попробовать превратить первую букву e (по индексу 2) в букву i, у нас ничего не получится:

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

Чтобы изменить текст, хранящийся в переменной `text`, необходимо присвоить ей совершенно новое значение. В главе 4 вы научились использовать цикл `for` для перебора символов в строке. Попробуем превратить каждую букву в строке в прописную следующим образом:



Далее проверим значение переменной `new`, чтобы определить регистр букв:

```
>>> new
'THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.'
```

Таким образом можно перебрать символы в переменной `text` и создать новую строку. Если символ является гласной буквой, его

необходимо заменить; в противном случае используется исходный символ. Мы работали с гласными в главе 2, можете вернуться к ней, чтобы освежить знания.

8.1.1. Метод str.replace()

В главе 4 мы говорили об использовании метода `str.replace()` для замены всех чисел в строке другим числом. Возможно, данный способ поможет решить и нашу проблему? Давайте заглянем в документацию к методу, для этого выполните команду `help(str.replace)` в REPL-интерфейсе:

```
>>> help(str.replace)
replace(self, old, new, count= -1, /)
    Return a copy with all occurrences of substring old replaced
    by new.

      count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.

    If the optional argument count is given, only the first count
    occurrences are replaced.
```

Ну что ж, попробуем. Заменим букву «T» на «X»:

```
>>> text.replace('T', 'X')
'Xhe quick brown fox jumps over the lazy dog.'
```

Кажется, работает! Сможете придумаеть способ заменить все гласные в строке подобным образом? Не забывайте, что данный метод не меняет заданную строку, а возвращает новую, которую следует присвоить переменной.

8.1.2. Метод str.translate()

Также в главе 4 мы изучали метод `str.translate()`. Тогда мы создали словарь, чтобы превращать одни строки, например `'1'`, в другие, скажем `'9'`. Символы, отсутствующие в словаре, пропускались.

Документация к данному методу чуть сложнее:

```
left untouched.  Characters mapped to None are del>>>
  help(str.translate) translate(self, table, /)
    Replace each character in the string using the given
    translation table.

      table
        Translation table, which must be a mapping of Unicode
        ordinals to Unicode ordinals, strings, or None.

    The table must implement lookup/indexing via __getitem__, for
    instance a dictionary or list.  If this operation raises
    LookupError, the character is eted.
```

В своем решении я создал следующий словарь:

```
jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
          '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
```

Вы видите аргумент для функции `str.maketrans()`, создающей таблицу замены, которая затем используется методом с `str.translate()` в целях замены всех символов, присутствующих в качестве ключей, на соответствующие им значения:

```
>>> '876-5309'.translate(str.maketrans(jumper))
'234-0751'
```

Подумайте, какие ключи и значения должны быть указаны в словаре, если нужно изменить все гласные, как строчные, так и прописные, на какое-то иное значение?

8.1.3. Другие способы изменения строк

Что вы знаете о регулярных выражениях? Они могли бы стать отличным ответом на волнующий нас вопрос. Если вы о них ничего не слышали, не волнуйтесь, я познакомлю вас с ними.

Суть в том, чтобы вы пробовали и экспериментировали. Я выявил восемь способов заменить все гласные новым символом, так что путей решения множество. Сколько *различных* способов вы сможете найти самостоятельно, прежде чем изучить мое решение?

Дам несколько советов:

— прочитайте документацию к модулю `argparse` в части использования переменной `choices` для ограничения опции `-vowel`, а также ознакомьтесь с разделом А.4.3 в приложении к этой книге;
— обязательно меняйте как строчные, так и прописные гласные, сохраняя регистр символов.

Начните с поиска собственного решения, а затем изучите мое.

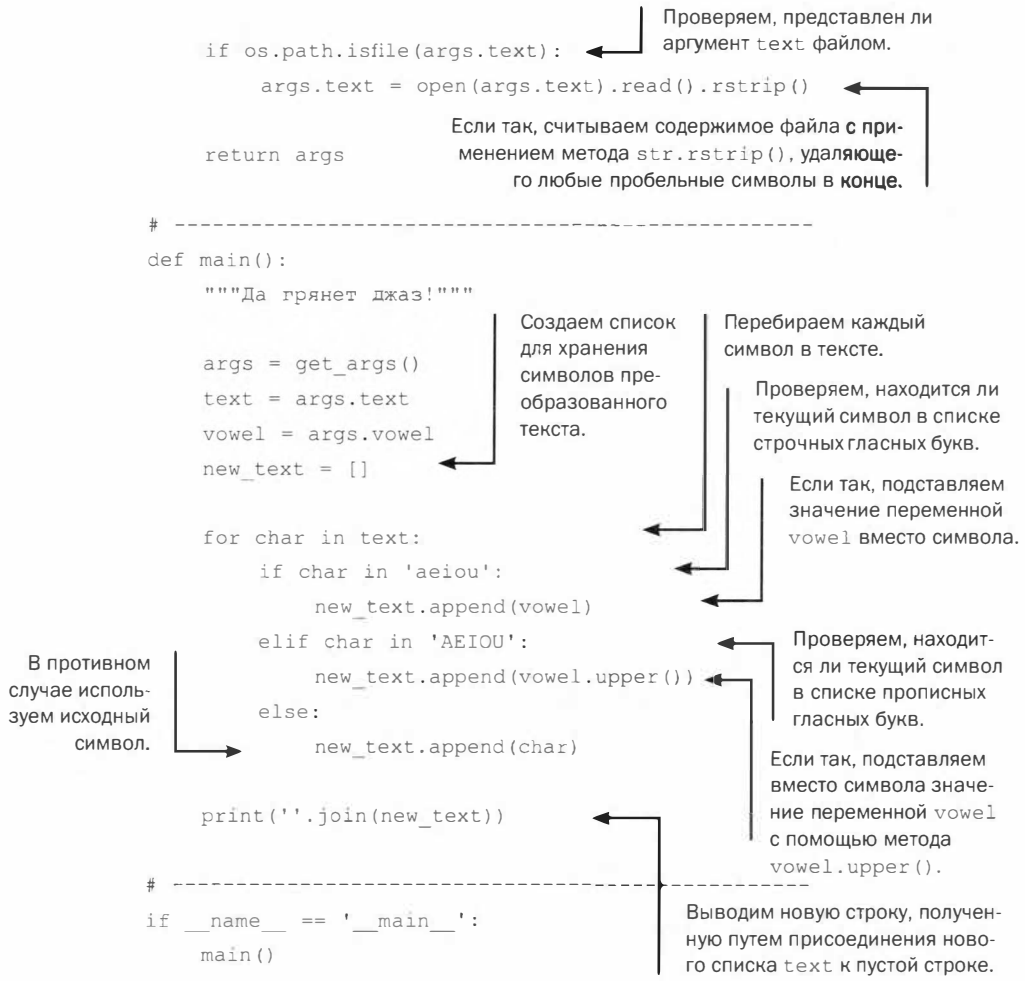
8.2. Решение

Вот первый вариант программы, которым я хотел бы поделиться. После него мы рассмотрим еще несколько решений.

```
#!/usr/bin/env python3
"""Яблоки и бананы"""

import argparse
import os

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Apples and bananas',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='text', help='Input text
                        or file')

    parser.add_argument('-v',
                        '--vowel',
                        help='The vowel(s) allowed',
                        metavar='vowel',
                        type=str,
                        default='a',
                        choices=list('aeiou'))

    args = parser.parse_args()
```

В качестве входных данных может поступать как простой текст, так и файл с указанным именем, поэтому я определяю их как строку.

Используем переменную `choices`, чтобы определить группу изменяемых гласных.



8.3. Обсуждение

Я придумал восемь способов решить задачу. Все они начинаются с одной и той же функции `get_args()`, поэтому рассмотрим ее в первую очередь.

8.3.1. Определение параметров

Вот одна из тех проблем, которые имеют много допустимых и интересных решений. Первая задача, которую необходимо решить, — это,

конечно же, получение и проверка ввода пользователя. Как обычно, я буду использовать модуль `argparse`.

В первую очередь я определяю необходимые параметры. Параметр `text` представляет собой позиционную строку, которая *может* быть именем файла:

```
parser.add_argument('text', metavar='str', help='Input text or file')
```

Параметр `--option` также является строкой, и я решил применить опцию `choices`, чтобы модуль `argparse` проверял, совпадают ли введенные пользователем символы со списком `'aeiou'`:

```
parser.add_argument('-v',
                    '--vowel',
                    help='The vowel to substitute'',
                    metavar='str',
                    type=str,
                    default='a',
                    choices=list('aeiou'))
```

То есть опции `choices` требуется список вариантов. Я мог бы воспользоваться кодом `['a', 'e', 'i', 'o', 'u']`, но, на мой взгляд, это слишком длинно. Гораздо проще набрать `list('aeiou')` и превратить строку «aeiou» в список символов. Оба подхода приводят к одному и тому же результату, поскольку метод `list(str)` создает список отдельных символов переданной строки.

Не забывайте, что нет разницы, какие кавычки вы используете, одинарные или двойные. Значение, заключенное в кавычки любого типа, является строкой, даже если речь идет всего об одном символе:

```
>>> ['a', 'e', 'i', 'o', 'u']
['a', 'e', 'i', 'o', 'u']
>>> list('aeiou')
['a', 'e', 'i', 'o', 'u']
```

Давайте протестируем это. Отсутствие ошибок подтверждает правильность кода:

```
>>> assert ['a', 'e', 'i', 'o', 'u'] == list('aeiou')
```

Следующая задача — определить, указано ли в качестве значения `text` имя файла, из которого следует извлечь текст, либо указан сам

текст. Тут сработает тот же код, который мы использовали в главе 5: я обрабатываю аргумент `text` в функции `get_args()`, поэтому значение `text` в методе `main()` уже будет обработано. На рис. 8.2 показано, как связать функцию `open()` с методом `read()` файлового дескриптора и строковым методом `rstrip()`.

```
if os.path.isfile(args.text):
    args.text = open(args.text).read().rstrip()
```

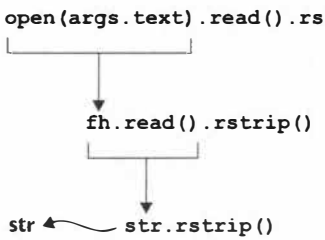


Рис. 8.2. Мы можем объединять методы в цепочки для создания конвейеров операций. Функция `open()` возвращает файловый дескриптор, доступный для чтения. Операция `read()` возвращает строку с удаленными пробельными символами

На данный момент пользовательские аргументы программы полностью проверены. Мы получаем текст либо непосредственно из командной строки, либо из файла, и мы убедились, что значение `--vowel` является одним из допустимых символов. В моем понимании данный код представляет собой единый «модуль», обрабатывающий аргументы. Теперь обработка может быть продолжена путем возврата аргументов:

```
return args
```

8.3.2. Восемь способов замены гласных

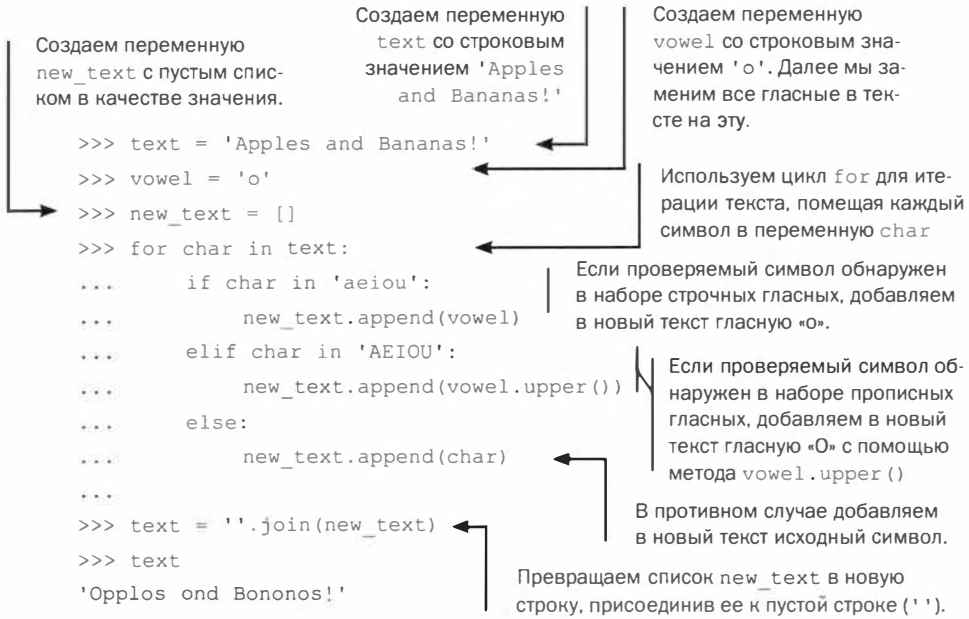
Сколько способов замены гласных удалось вам найти? Разумеется, чтобы пройти тесты, в программе нужно использовать только один, но я надеюсь, что вы попытались решить задачу разными методами, кайфуя от гибкости языка. Из Дзена Python:

> *Должен существовать один и желательно только один очевидный способ сделать это.*
>
> **www.python.org/dev/peps/pep-0020**

На самом деле мне близок обратный девиз из языка Perl «есть больше одного способа сделать это» (принцип TIMTOWTDI («Тим Тоуди»)).

Способ 1: перебор каждого символа

Первый метод аналогичен использованному нами в главе 4. Тогда мы применяли к строке цикл `for` для обработки каждого символа. Ниже показан код, который можно выполнить в IPython REPL-интерфейсе:



Обратите внимание, что можно начать с создания переменной `new_text` с пустой строкой в качестве значения, а затем конкатенировать новые символы. При таком подходе не нужно использовать метод `str.join()` в конце. Вам нравится данный вариант?

```
new_text += vowel
```

Далее я продемонстрирую вам несколько альтернативных решений. Все они функционально эквивалентны, потому что успешно проходят тесты. Суть здесь в том, чтобы изучить и понять язык Python. При обсуждении альтернативных решений я покажу код функции `main()`.

Способ 2: метод str.replace()

Ниже представлен вариант решения проблемы с помощью метода `str.replace()`:

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel

    for v in 'aeiou':
        text = text.replace(v, vowel).replace(v.upper(),
                            vowel.upper())

    print(text)
```

Перебираем список гласных. Нам не понадобится код `list('aeiou')` — Python автоматически обработает строку `'aeiou'` как список, потому что мы используем ее в контексте списка с циклом `for`.

Используем метод `str.replace()`, чтобы заменить как строчные, так и прописные гласные в тексте.

Ранее в этой главе я упоминал метод `str.replace()`, возвращающий новую строку, в которой все экземпляры одной строки заменены другой:

```
>>> s = 'foo'
>>> s.replace('o', 'a')
'faa'
>>> s.replace('oo', 'x')
'fx'
```

Обратите внимание, что исходная строка остается неизменной:

```
>>> s
'foo'
```

Вам не нужно связывать два метода `str.replace()`. Код можно оформить в виде двух отдельных операторов, как показано на рис. 8.3.

```
text = text.replace(v, vowel).replace(v.upper(), vowel.upper())

text = text.replace(v, vowel)
text = text.replace(v.upper(), vowel.upper())
```

Рис. 8.3. По желанию последовательность вызовов метода `str.replace()` можно записать как два отдельных оператора

Способ 3: метод str.translate()

Пригоден ли метод `str.translate()` для решения нашей проблемы? В главе 4 мы обсуждали, как для замены символа, к примеру 1 на 9, используется словарь `jumper`. Для решения текущей задачи необходимо заменить все строчные и прописные гласные (всего 10) на одну указанную гласную. Например, чтобы заменить все гласные буквой o, попробуем создать таблицу замены `t` следующим образом:

```
t = {'a': 'o',
     'e': 'o',
     'i': 'o',
     'o': 'o',
     'u': 'o',
     'A': 'O',
     'E': 'O',
     'I': 'O',
     'O': 'O',
     'U': 'O'}
```

Мы можем использовать таблицу замены `t` с методом `str.translate()`:

```
>>> 'Apples and Bananas'.translate(str.maketrans(t))
'Opplos ond Bononos'
```

В документации к методу `str.maketrans()` указан другой способ взаимодействия с таблицей замены, заключающийся в применении двух строк одинаковой длины:

```
maketrans(x, y=None, z=None, /)
    Return a translation table usable for str.translate().

    If there is only one argument, it must be a dictionary
    mapping Unicode ordinals (integers) or characters to Unicode
    ordinals, strings or None.
    Character keys will be then converted to ordinals.
    If there are two arguments, they must be strings of equal
    length, and in the resulting dictionary, each character in x
    will be mapped to the character at the same position in y. If
    there is a third argument, it must be a string, whose
    characters will be mapped to None in the result.
```

Первая строка должна содержать буквы, которые вы хотите заменить, то есть строчные и прописные гласные `'aeiouAEIOU'`. Вторая строка состоит из букв, подставляемых взамен. Мы можем использовать строку `'ooooo'` взамен `'aeiou'` и `'OOOOOO'` взамен `'AEIOU'`.

Применяем переменную `vowel` пять раз с помощью оператора `*`, обычно ассоциируемого с умножением чисел. Это (своего рода) «умножение» строки:

```
>>> vowel * 5
'ooooo'
```

Далее работаем с прописными буквами:

```
>>> vowel * 5 + vowel.upper() * 5
'oooooOOOOO'
```

Теперь мы можем использовать таблицу замены с помощью одной строки кода следующим образом:

```
>>> trans = str.maketrans('aeiouAEIOU', vowel * 5 +
                           vowel.upper() * 5)
```

Давайте рассмотрим таблицу `trans`. Воспользуемся функцией `pprint.pprint()` для форматирования вывода:

```
>>> from pprint import pprint as pp
>>> pp(trans)
{65: 79,
 69: 79,
 73: 79,
 79: 79,
 85: 79,
 97: 111,
 101: 111,
 105: 111,
 111: 111,
 117: 111}
```

Фигурные скобки `{}` говорят о том, что `trans` — это словарь. Каждый символ сопровождает его *порядковое* значение, отражающее расположение символа в таблице ASCII (**www.asciitable.com**).

Вы можете переключаться между символами и их порядковыми значениями с помощью функций `chr()` и `ord()`. Мы рассмотрим эти функции позже, в главе 18. Ниже показаны порядковые значения нужных нам гласных:

```
>>> for char in 'aeiou':
...     print(char, ord(char))
...
a 97
e 101
i 105
o 111
u 117
```

Можно получить тот же результат, используя порядковые значения для получения самих символов:

```
>>> for num in [97, 101, 105, 111, 117]:
...     print(chr(num), num)
...
a 97
e 101
i 105
o 111
u 117
>>>
```

Проверим порядковые значения всех печатаемых символов:

```
>>> import string
>>> for char in string.printable:
...     print(char, ord(char))
```

Я не привожу вывод, поскольку существует 100 печатаемых символов:

```
>>> print(len(string.printable))
100
```

Таким образом, принцип таблицы замены основан на сопоставлении одного символа с другим, как было показано в упражнении главы 4. Все строчные гласные (aeiou) сопоставляются с порядковым значением `111`, то есть строчной буквой o. Прописные гласные (AEIOU)

сопоставляются с порядковым значением 79, то есть прописной буквой O. Попробуйте применить метод `dict.items()` для перебора пар «ключ/значение» в словаре `trans`, чтобы убедиться, что это так:

```
>>> for x, y in trans.items():
...     print(f' {chr(x)} => {chr(y)}')
...
a => o
e => o
i => o
o => o
u => o
A => O
E => O
I => O
O => O
U => O
```

При использовании метода `str.translate()` исходное значение переменной `text` не меняется, поэтому мы можем перезаписать его новой версией текста. Вот как я реализовал эту идею:

```
def main():
    args = get_args()
    vowel = args.vowel
    trans = str.maketrans('aeiouAEIOU', vowel * 5 +
                          vowel.upper() * 5)
    text = args.text.translate(trans)
    print(text)
```

Создаем таблицу замены всех гласных, как строчных, так и прописных, на соответствующие символы. Строчные гласные сопоставляются со строчным символом, а прописные — с прописным.

На переменной `text` вызываем метод `str.translate()`, передав таблицу замены в качестве аргумента.

Я несколько раз упоминал функции `ord()` и `chr()`, словари и прочее, но взгляните, насколько данное решение простое и элегантное. Код намного короче, чем в способе 1. Меньше кода (метрика LOC) — меньше ошибок!

Способ 4: представление списка

Следуя методу 1, мы можем использовать *представление списка*, чтобы значительно сократить код цикла `for`. В главе 7 мы рассмотрели представление словаря. Здесь мы можем сделать практически то же самое:

```
def main():
    args = get_args()
    vowel = args.vowel
    text = [
        vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU'
        else c for c in args.text
]
    print(''.join(text))
```

С помощью представления списка обрабатываем все символы в `args.text` для создания списка `text`.

Выводим преобразованную строку, присоединив список `text` к пустой строке.

Используем сложную конструкцию `if` для обработки трех случаев: строчной гласной, прописной гласной и дефолтного значения.

Давайте еще немного поговорим о включении списков. Например, мы можем сгенерировать список возведенных в квадрат чисел от 1 до 4, используя функцию `range()` и указав через запятую начальное и конечное числа (не включая конечное). В REPL-интерфейсе нужно применить функцию `list()` для принудительной генерации значений, однако обычно в коде это не требуется:

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

Примечание. `range()` — еще один пример *ленивой* функции Python, то есть фактически она не выполняет вычислений, пока их результат не потребуется программе — ленивая она именно потому, что «обещает что-нибудь сделать». Если в вашей программе результаты вычислений появляются при необходимости, то есть без нужды лишние или повторные вычисления не выполняются, значит, ваш код работает эффективнее.

Напишем цикл `for` для вывода квадратов чисел:

```
>>> for num in range(1, 5):
...     print(num ** 2)
...
1
4
```

```
9
16
```

Вместо вывода самих значений создадим новый список, содержащий эти значения. Один из способов сделать это — создать пустой список, а затем с помощью метода `list.append()` передать каждое значение циклу `for`:

```
>>> squares = []
>>> for num in range(1, 5):
...     squares.append(num ** 2)
```

Проверим, есть ли у нас теперь список квадратов чисел:

```
>>> assert len(squares) == 4
>>> assert squares == [1, 4, 9, 16]
```

Мы можем достичь того же результата с меньшим количеством строк кода, использовав принцип «представления списка» для создания нового списка, как показано на рис. 8.4.

```
>>> [num ** 2 for num in range(1, 5)]
[1, 4, 9, 16]
```

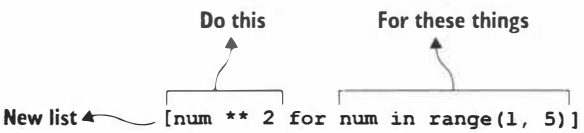


Рис. 8.4. С помощью представления списка создаем новый список, используя цикл `for` для перебора исходных значений

Мы можем присвоить полученный список переменной `squares` и проверить результат. Какой вариант кода вы хотите поддерживать в будущем: длинный с циклом `for` или короткий с представлением списка?

```
>>> squares = [num ** 2 for num in range(1, 5)]
>>> assert len(squares) == 4
>>> assert squares == [1, 4, 9, 16]
```

Для данной версии программы мы соберем конструкцию `if/elif/else` из метода 1 в составное выражение `if`. Сначала посмотрим, как можно сократить код цикла `for`:

```
>>> text = 'Apples and Bananas!'
>>> new = []
>>> for c in text:
...     new.append(vowel if c in 'aeiou' else vowel.upper() if c
       in 'AEIOU' else c)
...
>>> ''.join(new)
'Opplos ond Bononos!'
```

На рис. 8.5 показано соответствие частей выражения конструкции `if/elif/else`:

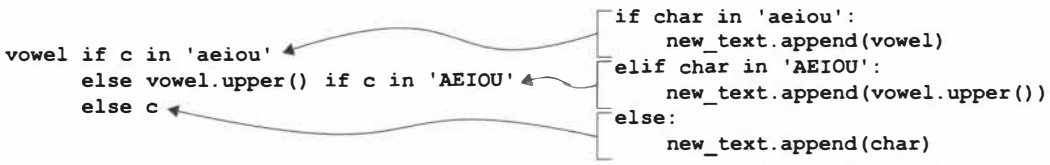


Рис. 8.5. Три условные ветви можно записать с помощью двух выражений `if`

Теперь давайте применим представление списка:

```
>>> text = 'Apples and Bananas!'
>>> new_text = [
...     vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU'
       else c
...     for c in text]
...
>>> ''.join(new_text)
'Opplos ond Bononos!'
```

Выбираем символ с помощью составного выражения `if`.

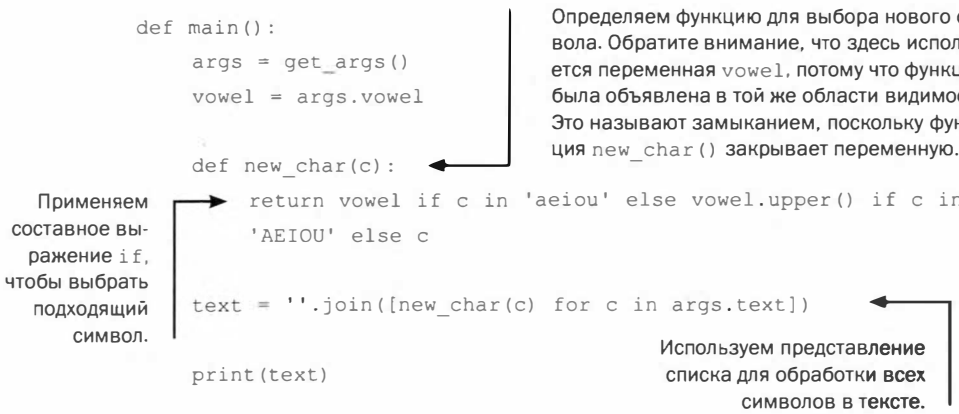
Выполняем данное действие для каждого символа в тексте.

Код получается плотнее, чем ранее рассмотренный цикл `for`, однако имеет следующие преимущества:

— представление списка короче и генерирует наш список, а не использует побочные эффекты метода `list.append()`;
— составное выражение `if` не будет компилироваться, если мы забудем про одну из условных ветвей.

Способ 5: представление списка с функцией

Составное выражение `if` в представлении списка достаточно сложное, поэтому его, вероятно, следует превратить в функцию. Мы можем *определить* новую функцию с помощью оператора `def` и присвоить ей имя `new_char()`. Она принимает символ, хранящийся в переменной `c`. Затем мы можем использовать то же составное выражение `if`, что и раньше:



Поэкспериментируем с функцией `new_char()` в REPL-интерфейсе:

```
vowel = 'o'
def new_char(c):
    return vowel if c in 'aeiou' else vowel.upper() if c in
    'AEIOU' else c
```

Она всегда должна возвращать букву o, если аргумент является строчной гласной:

```
>>> new_char('a')
'o'
```

Либо букву O, если аргумент является прописной гласной:

```
>>> new_char('A')
'O'
```

В противном случае возвращается исходный символ:

```
>>> new_char('b')
'b'
```

Применяем функцию `new_char()` для обработки всех символов в тексте, используя представление списка:

```
>>> text = 'Apples and Bananas!'
>>> text = ''.join([new_char(c) for c in text])
>>> text
'Opplos ond Bononos!'
```

Обратите внимание, что функция `new_char()` объявляется *внутри* функции `main()`. Да, мы можем так поступить! В данном случае функция «видна» только в пределах функции `main()`. Я так сделал, чтобы сослаться на значение переменной `vowel` внутри функции, не передавая ее в качестве аргумента.

Теперь давайте попробуем определить функцию `foo()`, внутри которой располагается функция `bar()`. Мы можем вызвать функцию `foo()`, и она, в свою очередь, вызовет функцию `bar()`. Но вне функции `foo()` функция `bar()` недоступна («не видна», «не находится в области видимости»).

```
>>> def foo():
...     def bar():
...         print('This is bar')
...     bar()
...
>>> foo()
This is bar
>>> bar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'bar' is not defined
```

Я объявил функцию `new_char()` внутри функции `main()`, чтобы сослаться на значение переменной `vowel` внутри функции, как показано на рис. 8.6. Поскольку функция `new_char()` «замыкается» вокруг переменной `vowel`, такая функция называется *замыкающей*.

Если не использовать замыкание, нам придется передать значение переменной `vowel` как аргумент:

```
def main():
    args = get_args()
    print(''.join([new_char(c, args.vowel) for c in args.text]))
```

Передаем значение `args.vowel` в качестве аргумента функции `new_char()`.

```
def new_char(char, vowel):
    return vowel if char in 'aeiou' else \
        vowel.upper() if char in 'AEIOU' else char
```

Переменная `vowel` доступна только внутри функции `main()`. Поскольку функция `new_char()` не объявляется в той же области видимости, необходимо использовать значение переменной `vowel` в качестве аргумента.

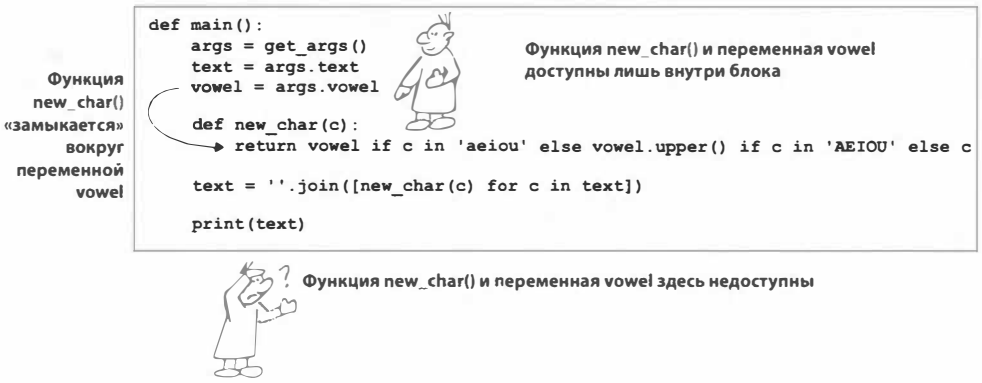


Рис. 8.6. Функция `new_char()` доступна только внутри функции `main()`. Она создает замыкание, так как ссылается на переменную `vowel`. Операторы вне функции `main()` не видят и не могут вызывать функцию `new_char()`

Замыкание выглядит интересно, и данную версию, возможно, легче понять. Кроме того, для нее проще написать модульный тест, чем мы вскоре и займемся.

Способ 6: функция map()

Еще один способ решить нашу задачу — использовать функцию `map()`, очень похожую на представление списка. Функция `map()` принимает два аргумента:

— функцию;
— итерируемый объект, например список, ленивую функцию или генератор.

Представьте функцию `map()` как покрасочную мастерскую, заправленную, допустим, синей краской. Заезжают некрашеные машины, а после нанесения краски выезжают синие.

Мы можем создать функцию для «окрашивания» машин, добавив в начало текста строку `'blue '`:

```
>>> list(map(lambda car: 'blue ' + car, ['BMW', 'Alfa Romeo',
            'Chrysler']))
['blue BMW', 'blue Alfa Romeo', 'blue Chrysler']
```

Первый показанный аргумент начинается с ключевого слова `lambda`, которое используется для создания *анонимной* (или *лямбда*-) функции. За ключевым словом `def` следует имя функции. У анонимной функции нет имени, только список параметров и тело.

В качестве примера ниже показана обычная именованная функция `add1()`, добавляющая `1` к значению:

```
def add1(n):
    return n + 1
```

Она работает предсказуемо:

```
>>> assert add1(10) == 11
>>> assert add1(add1(10)) == 12
```

Сравните предыдущий код с определением анонимной функции, которую мы присваиваем переменной `add1`:

```
>>> add1 = lambda n: n + 1
```

Определение `add1` эквивалентно первой версии функции. Вызов осуществляется точно так же, как и вызов функции `add1()`:

```
>>> assert add1(10) == 11
>>> assert add1(add1(10)) == 12
```

Тело анонимной функции представляет собой короткое (обычно однострочное) выражение. В нем нет оператора `return`, потому что окончательный результат вычисления выражения возвращается

автоматически. На рис. 8.7 показано, что анонимная функция возвращает результат `n + 1`.

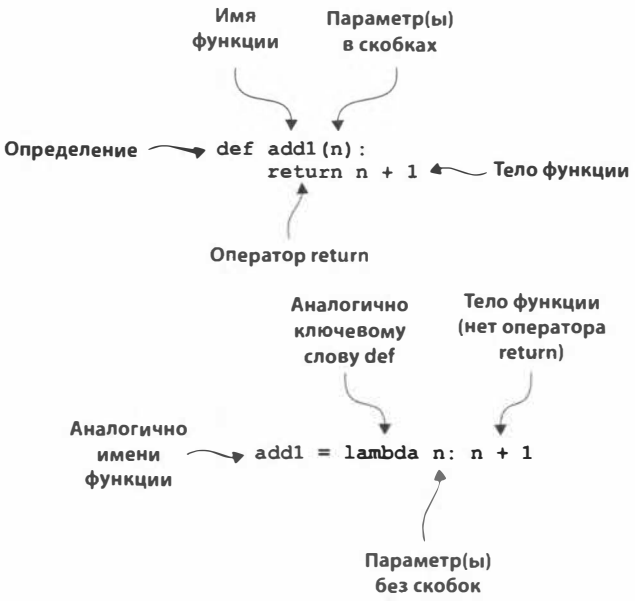


Рис. 8.7. Оба ключевых слова, и `def`, и `lambda`, используются для создания функций

В обеих версиях определения функции `add1`, как именованной, так и анонимной, аргументом функции является `n`. В обычной именованной функции `def add(n)` аргумент определяется в круглых скобках сразу после имени функции. У анонимной функции отсутствует как имя, так и круглые скобки вокруг параметра `n`.

Нет никакой разницы в том, как использовать эти функции. В обоих случаях:

```
>>> type(lambda x: x)
<class 'function'>
```

Если вам удобно иметь дело с функцией `add1()` в представлении списка, например так:

```
>>> [add1(n) for n in [1, 2, 3]]
[2, 3, 4]
```

вы легко сможете перейти к функции `map()`.

Функция `map()` — ленивая, как и функция `range()`, которую мы рассмотрели чуть ранее. Она не выполняет вычисления до тех пор, пока

вам не понадобится результат. Сравните данный подход с концепцией представления списка, при котором результирующий список создается сразу же. Если интересно мое мнение, для меня больше важна читабельность кода, нежели производительность. Поэтому для своих проектов я выбираю функцию `map()`. Мой совет: пишите код, который будет наиболее удобен для вас и вашей команды.

Чтобы в REPL-интерфейсе форсировать вычисления функции `map()`, необходимо использовать функцию `list()`:

```
>>> list(map(add1, [1, 2, 3]))
[2, 3, 4]
```

Мы можем реализовать представление списка с помощью функции `add1()` следующим образом:

```
>>> [n + 1 for n in [1, 2, 3]]
[2, 3, 4]
```

Очень похоже на анонимную функцию (как показано на рис. 8.8):

```
>>> list(map(lambda n: n + 1,
             [1, 2, 3]))
[2, 3, 4]
```

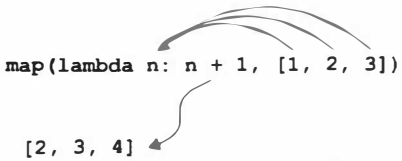


Рис. 8.8. Функция `map()` генерирует новый список, обрабатывая каждый элемент итерируемого объекта с помощью заданной функции

Вот как можно использовать функцию `map()`:

Функция `map()` ожидает функцию для первого аргумента и итерируемый объект для второго.

Указываем ключевое слово `lambda` для создания анонимной функции, принимающей символ, `c`.

```
def main():
    args = get_args()
    vowel = args.vowel
    text = map(
        lambda c: vowel if c in 'aeiou' else vowel.upper()
        if c in 'AEIOU' else c, args.text)

    print(''.join(text))
```

`args.text` — второй аргумент функции `map()`. Технически `args.text` — это строка, но, поскольку функция `map()` ожидает, что данный аргумент будет представлен списком, строка приводится к списку.

Функция `map()` возвращает новый список переменной `text`. Мы присоединяем результат к пустой строке, чтобы вывести ее.

Функции высшего порядка

`map()` — *функция высшего порядка* (HOF, higher-order function), потому что в качестве аргумента она принимает другую функцию. И это очень круто. Позже мы воспользуемся другой функцией высшего порядка, `filter()`.

Способ 7: map() и именованная функция

Необязательно использовать `map()` строго с анонимной функцией. Подойдет любая, так что давайте вернемся к функции `new_char()`:

```
def main():
    args = get_args()
    vowel = args.vowel

    def new_char(c):
        return vowel if c in 'aeiou' else vowel.upper() if c in
'AEIOU' else c

    print(''.join(map(new_char, args.text)))
```

Определяем функцию, возвращающую корректный символ. Обратите внимание, что я использую замыкание, чтобы сослаться на аргумент «гласной буквы».

Используем функцию `map()`, чтобы применить функцию `new_char()` ко всем символам в `args.text`. Результатом становится список символов, который благодаря методу `str.join()` преобразуется в новую строку для вывода с помощью функции `print()`.

Обратите внимание, что в функции `map()` имя функции `new_char` указывается в качестве первого аргумента *без скобок*. Если добавить круглые скобки, осуществится *вызов* функции, и вы увидите следующую ошибку:

```
>>> text = ''.join(map(new_char(), text))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: new_char() missing 1 required positional argument: 'c'
```

Как показано на рис. 8.9, функция `map()` извлекает каждый символ текста (`text`) и передает его в качестве аргумента функции `new_char()`, которая, в свою очередь, решает, что следует вернуть — подставляемую гласную букву или исходный символ. Результатом сопоставления этих

символов является новый список символов. Его мы соединяем с помощью метода `str.join()` с пустой строкой, чтобы создать результирующую строку текста.

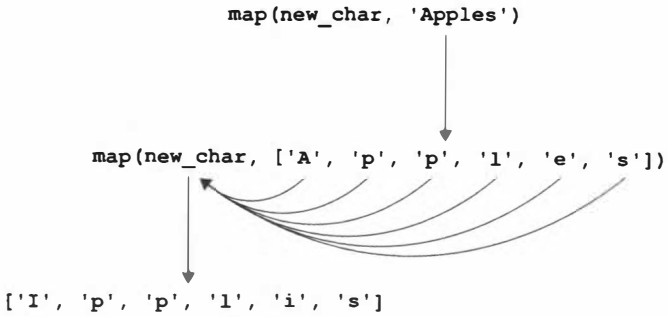


Рис. 8.9. Функция `map()` применяет заданную функцию к каждому элементу итерируемого объекта. Строка обрабатывается как список символов

Способ 8: регулярные выражения

Регулярные выражения — способ работы с шаблонами текста. Они представляют собой отдельный предметно-ориентированный язык (DSL, domain-specific programming language). На самом деле регулярные выражения не имеют ничего общего с Python. У них есть собственный синтаксис и правила, и еще они используются во многих областях, от инструментов командной строки до баз данных. Они невероятно мощны и стоят того, чтобы их изучить.

Для использования регулярных выражений вы должны импортировать в свое приложение модуль `re`:

```
>>> import re
```

Сейчас мы попробуем найти гласные буквы, которые определим как буквы a, e, i, o и u. Для реализации данной идеи с помощью регулярного выражения нам нужно заключить указанные символы в квадратные скобки:

```
>>> pattern = '[aeiou]'
```

Мы можем использовать функцию «замены» `re.sub()`, чтобы найти все гласные и заменить их заданной буквой. Квадратные скобки вокруг гласных `'[aeiou]'` создают *символьный класс,* определяющий набор символов в скобках, любой из которых может присутствовать во входной строке для успешного сопоставления.

Второй аргумент — это строка, предназначенная для замены найденных строк, — в нашем случае речь идет о гласной, введенной пользователем. Третий аргумент — это строка, которую мы хотим изменить, то есть сам текст, введенный пользователем:

```
>>> vowel = 'o'
>>> re.sub(pattern, vowel, 'Apples and bananas!')
'Applos ond bononos!'
```

Здесь не учитывается прописная буква A, поэтому нам придется обрабатывать и строчные, и прописные буквы. Вот как может выглядеть код:

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel
    text = re.sub('[aeiou]', vowel, text)
    text = re.sub('[AEIOU]', vowel.upper(), text)
    print(text)
```

Заменяем любую строчную гласную указанной в `vowel` буквой (которая является строчной из-за ограничений функции `get_args()`).

Заменяем любую прописную гласную указанной прописной буквой.

По желанию можно объединить два вызова функции `re.sub()`, как мы это делали с методом `str.replace()`:

```
>>> text = 'Apples and Bananas!'
>>> text = re.sub('[AEIOU]', vowel.upper(), re.sub('[aeiou]',
vowel, text))
>>> text
'Opplos ond Bononos!'
```

Кардинальное отличие данного решения от остальных заключается в том, что мы используем регулярные выражения для описания искомых значений. Нам не нужен код для определения гласных букв. Напоминает *декларативное* программирование. Мы пишем, что нам нужно, ну а компьютер делает за нас всю рутинную работу!

8.4. Рефакторинг с тестированием

Итак, существует много способов решить нашу задачу. Важнейший шаг — заставить программу работать правильно. В достижении этой цели отлично помогают тесты. С их помощью можно исследовать другие способы решения задачи и убеждаться в корректности их работы.

Тестирование предоставляет большую свободу для творчества. Никогда не забывайте о нем, чтобы, когда вы вносите изменения в код, приложения продолжали исправно работать.

Я показал множество способов решения, казалось бы, тривиальной задачи. Некоторые из них, включающие функции высшего порядка и регулярные выражения, являются весьма продвинутыми. Хотя у кого-то из читателей может возникнуть ощущение забивания гвоздя кувалдой, обратите внимание, что я лишь знакомлю вас с концепциями программирования, к которым буду возвращаться вновь и вновь в последующих главах.

Если вы разобрались лишь с несколькими способами, ничего страшного! Просто читайте внимательно дальше. Чем чаще вы будете сталкиваться с реализацей данных концепций в разных контекстах и программах, тем понятнее вам будет.

8.5. Прокачиваем навыки

Напишите версию программы, в которой одинаковые соседние гласные буквы схлопываются в одну. Например, слово quick должно стать qack, а не qaack.

РЕЗЮМЕ

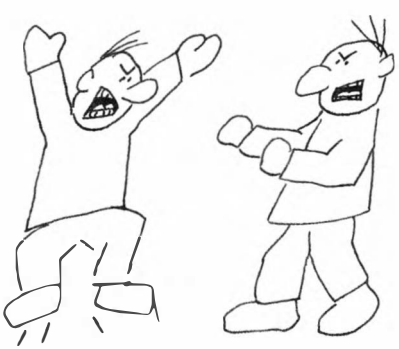
- С помощью модуля `argparse` можно ограничить значения аргумента списком определяемых вариантов.
- Строки неизменны, но благодаря методам `str.replace()` и `str.translate()` из исходной строки можно создать *новую измененную строку*.
- Цикл `for` перебирает символы «скормленной» ему строки.
- Представление списка — это лаконичный способ разместить цикл `for` внутри скобок `[]` для создания нового списка.
- Функции могут быть определены внутри других функций. Тогда их видимость ограничивается окружающей их функцией.

— Функции могут ссылаться на переменные, объявленные в той же области видимости, тем самым создавая замыкание.
— Функция `map()` напоминает представление списка. Она создает новый измененный список путем применения определенных функций к каждому элементу данного списка. Исходный список остается без изменений.
— Регулярные выражения предоставляют синтаксис шаблонов для работы с текстом и становятся доступны в Python путем импорта модуля `re`. Метод `re.sub()` заменяет найденные по шаблону фрагменты текста новыми значениями. Исходный текст остается без изменений.

Глава 9
Генератор ругательств: рандомные оскорбления из списков слов

«Он или она — питающийся илом мерзкий слизняк с лягушачьей мордой и черепашьими мозгами».

— Генератор ругательств



На случайных (рандомных) событиях основаны многие интересные игры и головоломки. Людям быстро надоедает рутина. Я думаю, что одна из причин, по которой люди заводят домашних животных и рожают детей, заключается в том, что они хотят внести в свою жизнь некоторую случайность. Давайте узнаем, как можно сделать наши программы более интересными путем изменения их поведения при каждом запуске.

Благодаря упражнению из этой главы вы узнаете, как выбрать один или более элементов из списков вариантов. Изучая принципы случайного выбора, мы создадим программу *abuse.py*, которая будет оскорблять пользователя, рандомно выбирая прилагательные и существительные для создания клеветнических эпитетов.

Однако, чтобы проверить корректность работы случайного выбора, необходимо взять его под контроль. На самом деле «случайные» события на компьютерах редко бывают по-настоящему случайными, а лишь *псевдослучайными*. То есть мы можем ими управлять с помощью некоего «зерна»*. Каждый раз, используя одно и то же зерно, вы получаете один и тот же «рандомный» выбор!

У Шекспира есть много прекрасных оскорблений, поэтому я планирую опираться на его произведения. Ниже показан список прилагательных, которые мы будем использовать:

bankrupt base caterwauling corrupt cullionly detestable dishonest false filthsome filthy foolish foul gross heedless indistinguishable infected insatiate irksome lascivious lecherous loathsome lubbery old peevish rascaly rotten ruinous scurilous scurvy slanderous sodden-witted thin-faced toad-spotted unmannered vile wall-eyed

А вот список существительных:

Judas Satan ape ass barbermonger beggar block boy braggart butt carbuncle coward coxcomb cur dandy degenerate fiend fishmonger fool gull harpy jack jolthead knave liar lunatic maw milksop minion ratcatcher recreant rogue scold slave swine traitor varlet villain worm

Соответственно, программа может выдавать следующее:

```
$ ./abuse.py
You slanderous, rotten block!
You lubbery, scurilous ratcatcher!
You rotten, foul liar!
```

Освоив данную главу, вы научитесь:

— пользоваться методом `parser.error()` из модуля `argparse`, чтобы выдавать ошибки;
— управлять фактором случайности с помощью рандомных зерен;
— в рандомном порядке выбирать элементы из списков Python;
— итерировать алгоритм заданное количество раз с помощью цикла `for`;
— форматировать результат работы программы.

---

* «Генерация случайных чисел слишком важна, чтобы оставлять ее на волю случая». *Роберт Р. Кавью.*

9.1. Создание файла abuse.py

Перейдите в каталог *09_abuse*, в нем мы создадим новую программу. Начнем с изучения справочной информации, которую та должна выводить:

```
$ ./abuse.py -h
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]

Heap abuse

optional arguments:
-h, --help            show this help message and exit
-a adjectives, --adjectives adjectives
                      Number of adjectives (default: 2)
-n insults, --number insults
                      Number of insults (default: 3)
-s seed, --seed seed  Random seed (default: None)
```

Все параметры опциональны и имеют дефолтные значения, так что наша программа может запускаться вообще без аргументов.

Например, опция `-n` или `--number` имеет дефолтное значение `3` и отвечает за количество оскорблений:

```
$ ./abuse.py --number 2
You filthsome, cullionly fiend!
You false, thin-faced minion!
```

Опция `-a` или `--adjectives` по умолчанию имеет значение `2` и определяет, сколько прилагательных используется в каждом оскорблении:

```
$ ./abuse.py --adjectives 3
You caterwauling, heedless, gross coxcomb!
You sodden-witted, rascaly, lascivious varlet!
You dishonest, lecherous, foolish varlet!
```

Наконец, опция `-s` или `--seed` контролирует рандомность выбора, определяя некое начальное значение — *зерно*. Значением по умолчанию должно быть специальное значение `None`, похожее на неопределенное значение.

Поскольку программа использует случайное зерно, следующий вывод должен быть таким же у любого пользователя на любом компьютере в любое время:

```
$ ./abuse.py --seed 1
You filthsome, cullionly fiend!
You false, thin-faced minion!
You sodden-witted, rascaly cur!
```

При запуске без аргументов программа генерирует оскорбления с дефолтными значениями:

```
$ ./abuse.py
You foul, false varlet!
You filthy, insatiate fool!
You lascivious, corrupt recreant!
```

Я рекомендую скопировать содержимое файла *template/template.py* в файл *abuse/abuse.py* или с помощью файла *new.py* создать программу *abuse.py* в каталоге *09_abuse* вашего репозитория.

На рис. 9.1 представлена диаграмма, иллюстрирующая работу программы.

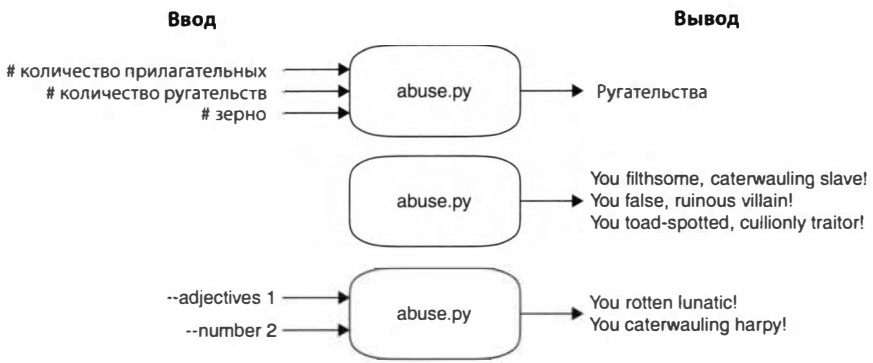


Рис. 9.1. Программа *abuse.py* опционально принимает количество создаваемых оскорблений, количество прилагательных в каждом оскорблении и случайное значение зерна

9.1.1. Проверка аргументов

Количество оскорблений и прилагательных, а также случайное зерно должны быть целочисленными значениями (`int`). Если вы определите параметры с помощью кода `type=int` (обратите внимание, что значение `int` не заключается в кавычки), модуль `argparse` будет автоматически проверять и преобразовывать значения аргументов в целочисленный тип. То есть, использовав код `type=int` и передав программе строку, вы увидите следующую ошибку:

```
$ ./abuse.py -n foo
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: argument -n/--number: invalid int value: 'foo'
```

Значение должно быть не просто числом, а *целым числом*, поэтому модуль `argparse` выдаст ошибку, если вы введете, к примеру, вещественное число (число с плавающей точкой*) (`float`). Кстати, вы можете использовать код `type=float`, если вам вдруг понадобится прибегнуть к вещественным числам:

```
$ ./abuse.py -a 2.1
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: argument -a/--adjectives: invalid int value: '2.1'
```

Кроме того, если значение `-number` и/или `--adjectives` меньше 1, программа завершит работу с кодом ошибки и сообщением:

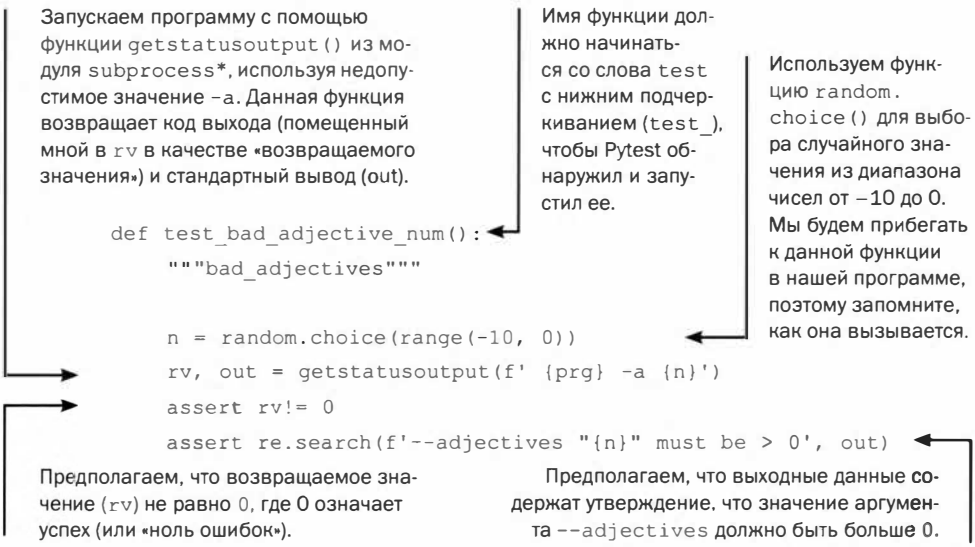
```
$ ./abuse.py -a -4
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: --adjectives "-4" must be > 0
$ ./abuse.py -n -4
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: --number "-4" must be > 0
```

В процессе работы над вашими программами тестирования рекомендую использовать мои тесты**. Взгляните на один из них в файле *test.py*. Давайте посмотрим, как тестируется программа:

---

* В программировании используется понятие «числа с плавающей точкой» вместо «чисел с плавающей запятой», принятых в России. *Прим. перев.*

** «Молодые поэты подражают, зрелые поэты воруют». *Томас Стернз Элиот.*



Нет простого способа сообщить модулю `argparse`, что количество прилагательных и оскорблений должно быть больше ноля, поэтому мы вынуждены вручную проверять эти значения. Мы воспользуемся концепциями из раздела A.4.7 приложения к данной книге. Применим функцию `parser.error()`, вызываемую внутри функции `get_args()`, чтобы выполнить следующее*.

1. Вывести краткую справочную информацию о программе.
2. Вывести пользователю сообщение об ошибке.
3. Прекратить выполнение программы.
4. Выйти из программы с кодом выхода, указывающим на ошибку.

Обычно функция `get_args()` завершается так:

```
return args.parse_args()
```

Вместо этого мы поместим аргументы в переменную и проверим значение аргумента `args.adjectives`, чтобы убедиться, что оно меньше 1. Если это так, вызываем функцию `parser.error()` с сообщением об ошибке:

```
args = parser.parse_args()
if args.adjectives < 1:
    parser.error(f'--adjectives "{args.adjectives}" must be > 0')
```

* Модуль subprocess позволяет выполнять команды внутри программы. Функция `subprocess.getoutput()` возвращает вывод команды, а функция `subprocess.getstatusoutput()` помимо вывода возвращает и код выхода.

Таким же образом можно поступить и с аргументом `args.number`. Если значения обоих аргументов допустимы, можно вернуть их вызывающей функции:

```
return args
```

9.1.2. Импорт и работа с модулем random

Определив и проверив все аргументы программы, мы можем приступить к следующему пункту — научить ее оскорблять пользователя. Для начала необходимо импортировать в программу модуль `random`, чтобы воспользоваться им для выбора прилагательных и существительных. Все операторы `import` рекомендуется перечислять в начале кода, импортируя модули по очереди.

Первое, что необходимо сделать в функции `main()`, это вызвать функцию `get_args()`, чтобы получить аргументы.

Следующий шаг — передать значение `args.seed` функции `random.seed()`:

```
def main()
    args = get_args()
    random.seed(args.seed)
```

Вызываем метод `random.seed()` для установки начального состояния модуля `random`. Метод `random.seed()` ничего не возвращает — все изменения происходят внутри модуля `random`.

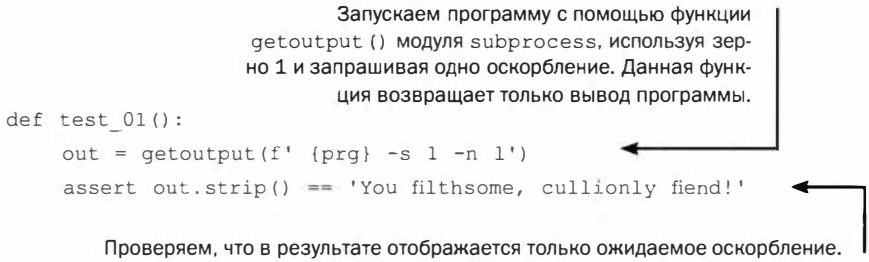
Вы можете прочитать документацию к методу `random.seed()` в REPL-интерфейсе:

```
>>> import random
>>> help(random.seed)
```

Из нее вы узнаете, что функция «инициализирует внутреннее состояние [модуля `random`] из хешируемого объекта». То есть мы задаем начальное значение [состояния] с помощью какого-то *хешируемого* типа Python. Оба типа, и `int`, и `str`, являются хешируемыми, но тесты написаны с расчетом на то, что аргумент `seed` определяется как целочисленное значение. (Помните, что *строка* `'1'` отличается от *целочисленного значения* `1`.)

Значение по умолчанию для `args.seed` — `None`. Если пользователь не указал зерно, то состояние `random.seed(None)` идентично не установленному вовсе начальному состоянию.

Открыв файл *test.py*, вы увидите, что все тесты, ожидающие определенного вывода, учитывают проверку аргумента `-s` или `--seed`. Вот первый тест вывода:



Файл *test.py* запускает программу и перехватывает вывод в переменную `out`:

```
$ ./abuse.py -s 1 -n 1
You filthsome, cullionly fiend!
```

Затем проверяется, действительно ли программа сгенерировала ожидаемое количество оскорблений с ожидаемым набором слов.

9.1.3. Определение прилагательных и существительных

Ранее в этой главе я предоставил длинный список прилагательных и существительных, которые мы можем использовать в программе. Вы вольны создать список, заключив каждое слово в кавычки:

```
>>> adjectives = ['bankrupt', 'base', 'caterwauling']
```

Или избавиться от трудоемкого набора кода вручную, применив метод `str.split()` для создания нового списка из строки, разбив ее по пробелам:

```
>>> adjectives = 'bankrupt base caterwauling'.split()
>>> adjectives
['bankrupt', 'base', 'caterwauling']
```

Если вы соберете одну строку из всех прилагательных, она получится очень длинной, будет переноситься на новую строку в окне редактора кода и выглядеть просто ужасно. Я рекомендую использовать тройные

кавычки (одинарные или двойные), чтобы разбивать прилагательные на строки:

```
>>> """
... bankrupt base
... caterwauling
... """.split()
['bankrupt', 'base', 'caterwauling']
```

Настроив переменные для хранения прилагательных и существительных, нужно проверить, правильное ли количество хранит каждая из них:

```
>>> assert len(adjectives) == 36
>>> assert len(nouns) == 39
```

Примечание. Чтобы программа успешно прошла тестирование, прилагательные и существительные должны быть приведены в алфавитном порядке, как указано в книге.

9.1.4. Выбор значений в случайном порядке

В дополнение к методу `random.seed()` мы также воспользуемся функциями `random.choice()` и `random.sample()`. В разделе 9.1.1 показан пример использования функции `random.choice()` в функции `test_bad_adjective_num()`. Аналогичным образом мы можем применить ее и сейчас, чтобы выбрать слово из списка существительных.

Обратите внимание, что данная функция возвращает один элемент, поэтому, используя список строк, мы получаем одну строку:

```
>>> random.choice(nouns)
'braggart'
>>> random.choice(nouns)
'milksop'
```

Для выбора прилагательных следует прибегнуть к методу `random.sample()`. В документации к нему (`help(random.sample)`) сказано, что данный метод принимает список элементов и параметр `k`, определяющий количество возвращаемых элементов:

```
sample(population, k) method of random.Random instance
Chooses k unique random elements from a population sequence or set.
```

Обратите внимание, что эта функция возвращает новый список:

```
>>> random.sample(adjectives, 2)
['detestable', 'peevish']
>>> random.sample(adjectives, 3)
['slanderous', 'detestable', 'base']
```

Существует также функция `random.choices()`, работающая аналогично, но позволяющая выбирать одни и те же элементы несколько раз, поскольку производит выборку «с заменой». Мы не будем ее использовать.

9.1.5. Форматирование вывода

Результатом работы программы является некое количество оскорблений, определяемое аргументом `--number`. Они генерируются с помощью цикла `for` и функции `range()`. Не имеет значения, что диапазон в этой функции начинается с ноля. Важно то, что она генерирует три значения:

```
>>> for n in range(3):
...     print(n)
...
0
1
2
```

Можно зациклить ее необходимое количество раз, выбрать прилагательные и существительные, а затем отформатировать вывод. Каждое оскорбление должно начинаться со строки «`You `». Последующие прилагательные разделены запятой и пробелом. Затем выводится существительное и восклицательный знак (рис. 9.2). Для форматирования вывода

можно использовать либо f-строку, либо метод `str.format()`, а затем передать его в поток `STDOUT`.

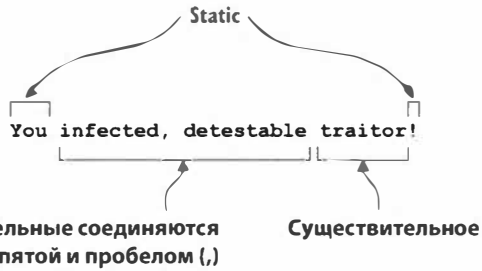


Рис. 9.2. Каждое оскорбление сочетает выбранные программой прилагательные, перечисленные через запятую, выбранное существительное и некоторый неизменный текст

Вот несколько советов.

- Выполняйте проверку положительных значений аргументов `--adjectives` и `--number` внутри функции `get_args()` и с помощью метода `parser.error()` выбрасывайте ошибки при выводе сообщения и справочной информации.
- Если для `args.seed` использовать дефолтное значение `None` и указать целочисленный тип с помощью `type=int`, значение можно передать непосредственно методу `random.seed()`. Присвоенное значение `None` равнозначно отсутствию значения.
- Используйте цикл `for` с функцией `range()` для создания цикла, который будет генерировать оскорбления нужное количество раз.
- Изучите документацию к `random.sample()` и `random.choice()` в части выбора прилагательных и существительных для нашей программы.
- Вы можете использовать три одинарные (`'''`) или двойные кавычки (`"""`) для формирования многострочной строки, а затем метод `str.split()` для получения списка строк. Такой прием реализовать проще, чем иметь дело с отдельными кавычками для длинного списка более коротких строк (например, списков прилагательных и существительных).
- Чтобы сформулировать оскорбление для вывода, можно использовать оператор + для конкатенации строк, метод слияния `str.join()` или f-строки.

Теперь потрудитесь хорошенько над своей программой, прежде чем подглядывать в мое решение!

9.2. Решение

В данном решении я впервые использую метод `parser.error()` для более надежной проверки аргументов. Еще в коде есть строки в тройных кавычках, а также импортируется модуль `random`, работающий довольно забавно.

```
#!/usr/bin/env python3
"""Грубое обращение"""

import argparse
import random

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Heap abuse',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-a',
                        '--adjectives',
                        help='Number of adjectives',
                        metavar='adjectives',
                        type=int,
                        default=2)

    parser.add_argument('-n',
                        '--number',
                        help='Number of adjectives',
                        metavar='adjectives',
                        type=int,
                        default=3)

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='seed',
                        type=int,
                        default=None)

    args = parser.parse_args()
```

Импортируем модуль `random`, чтобы вызывать соответствующие функции.

Определяем количество прилагательных, указываем целочисленный тип и дефолтное значение.

Аналогичным образом определяем количество оскорблений как целое число с дефолтным значением.

Дефолтное значение зерна — `None`.

Получаем результат разбора аргументов командной строки. Модуль `argparse` обрабатывает такие ошибки как нецелочисленные значения.

```
    if args.adjectives < 1:
        parser.error('--adjectives "{}" must be >
                     0'.format(args.adjectives))

    if args.number < 1:
        parser.error('--number "{}" must be >
                     0'.format(args.number))

    return args

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    random.seed(args.seed)

    adjectives = """
bankrupt base caterwauling corrupt cullionly detestable
dishonest false filthsome filthy foolish foul gross heedless
indistinguishable infected insatiate irksome lascivious
lecherous loathsome lubbery old peevish rascaly rotten ruinous
scurilous scurvy slanderous sodden-witted thin-faced
toad-spotted unmannered vile wall-eyed
""".strip().split()

    nouns = """
Judas Satan ape ass barbermonger beggar block boy braggart
butt carbuncle coward coxcomb cur dandy degenerate fiend
fishmonger fool gull harpy jack jolthead knave liar lunatic maw
milksop minion ratcatcher recreant rogue scold slave swine
traitor varlet villain worm
""".strip().split()
```

Проверяем, что значение `args.adjectives` больше `0`. Если возникает проблема, вызываем метод `parser.error()` с сообщением об ошибке.

Аналогичным образом проверяем значение `args.number`.

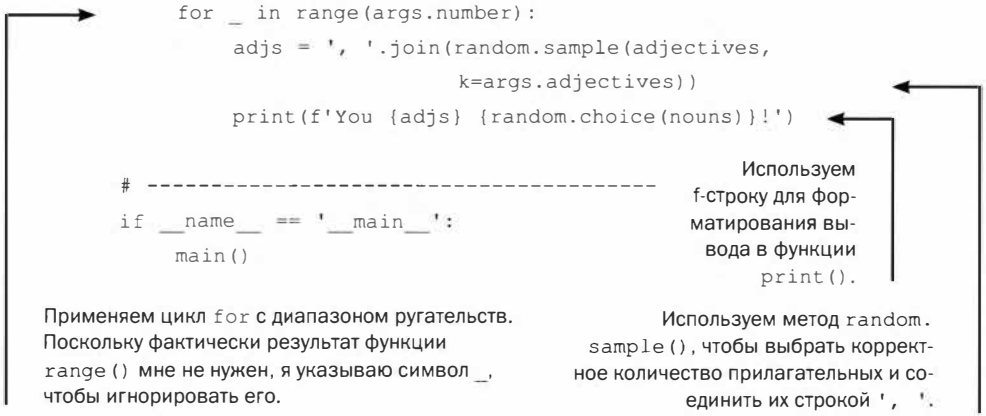
На данном этапе все аргументы пользователя проверены, поэтому возвращаем аргументы вызывающей стороне.

Именно отсюда начинается программа, так как это первая операция в функции `main()`. Все мои программы стартуют с получения аргументов.

Выполняем функцию `random.seed()` со значением, переданным пользователем. Допустимо любое целочисленное значение, и достоверно известно, что модуль `argparse` выполнил проверку и преобразование значения аргумента в целое число.

Создаем список прилагательных, разбивая излишне длинную строку с помощью тройных кавычек.

Повторяем операцию для списка существительных.



9.3. Обсуждение

Я надеюсь, что вы заглянули в решение уже после того, как ваша программа успешно прошла все тесты, иначе я на вас обижусь.

9.3.1. Определение аргументов

Львиная доля моего решения заключается в определении аргументов программы для модуля `argparse`. Но это того стоит. Поскольку я явно указал `type=int`, гарантируется, что каждый из аргументов имеет допустимое целочисленное значение. Обратите внимание, что в Python значение `int` не заключено в кавычки, так как это не строка `'int'`, а ссылка на класс:

Я выбрал разумные дефолтные значения для всех параметров программы, чтобы от пользователя не требовалось никаких действий.

Параметру `--seed` по умолчанию присвоено значение `None`, таким образом программа по умолчанию генерирует псевдослучайные оскорбления. Это важно для ее тестирования.

9.3.2. Метод parser.error()

Я обожаю модуль `argparse`, он мне реально помогает во многих проектах. К примеру, я часто использую метод `parser.error()`, когда возникает проблема с аргументами. Данный метод выполняет четыре действия.

1. Выводит пользователю краткий инструктаж по работе программы.
2. Выводит соответствующее сообщение о проблеме.
3. Прекращает выполнение программы.
4. Возвращает код ошибки операционной системе.

В данной программе метод `parser.error()` используется потому, что, хотя я и могу с помощью модуля `argparse` проверить, является ли заданное значение целым числом, мне неизвестно, положительное ли оно. Тем не менее я могу сам проверить значение и завершить работу программы в случае такой проблемы. Проверка осуществляется внутри функции `get_args()`, поэтому при получении аргументов в функции `main()` они уже проверены.

Я настоятельно рекомендую вам запомнить этот совет и пользоваться им. Вы сэкономите уйму времени на проверке пользовательского ввода и генерации полезных сообщений об ошибках. (При работе с готовой программой вы, несомненно, оцените свои старания.)

9.3.3. Коды выхода и поток STDERR

Я хочу заострить внимание на кодах выхода программы. В обычных условиях программы завершаются с кодом выхода `0`. В информатике частенько `0` представляется значением `False`, которое в данном случае играет положительную роль. Здесь мы говорим о «ноле (или отсутствии) ошибок».

Если вы используете в своей программе метод `sys.exit()` для ее преждевременного завершения, код выхода по умолчанию равен 0. Если требуется сообщить операционной системе или некой вызывающей программе, что ваша программа завершила работу с ошибкой, нужно вернуть *любой код выхода, отличный от 0*. Вы также можете вызвать

функцию, выводящую строку с сообщением об ошибке, и Python завершит работу программы с кодом выхода 1. Если вы выполните показанные ниже строки кода в REPL-интерфейсе, то вернетесь в командную строку:

```
>>> import sys
>>> sys.exit('You gross, thin-faced worm!')
You gross, thin-faced worm!
```

Стоит учитывать, что все сообщения об ошибках обычно выводятся в другой поток, не в STDOUT (стандартный вывод), а в STDERR (стандартная ошибка). Многие командные оболочки (например, Bash) сегрегируют эти два канала вывода, используя 1 для STDOUT и 2 для STDERR. Обратите внимание, как я могу в оболочке Bash использовать 2> для перенаправления потока STDERR в файл с именем *err*, чтобы в поток STDOUT ничего не передавалось:

```
$ ./abuse.py -a -1 2>err
```

Можно проверить, записаны ли ожидаемые сообщения об ошибках в файл *err*:

```
$ cat err
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: --adjectives "-1" must be > 0
```

Если вы хотите самостоятельно решить данную задачу, вам нужно написать следующий код:

```
if args.adjectives < 1:
    parser.print_usage()
    print(f'--adjectives "{args.adjectives}" must be > 0',
                          file=sys.stderr)
    sys.exit(1)
```

Выводим краткие инструкции по использованию программы. Вы также можете прибегнуть к методу `parser.print_help()` для подробного вывода программы при запуске с опцией `-h`.

Выходим из программы со значением, отличным от 0, чтобы указать на ошибку.

Выводим сообщение об ошибке в файловый дескриптор `sys.stderr`. Он похож на дескриптор `sys.stdout`, который мы использовали в главе 5.

Конвейеры

Вполне возможно, по мере разработки все большего количества программ вы начнете соединять их для реализации сложного функционала. Программисты часто называют такой процесс *конвейеризацией*. Это когда выходные данные одной программы передаются по конвейеру в качестве входных данных следующей. Если возникает ошибка в какой-либо части конвейера, мы, как правило, хотим, чтобы прекращалась вся операция и можно было исправить проблемы. Значение, отличное от ноля, возвращенное из любой программы конвейера, служит красным флагом для остановки операции.

9.3.4. Управление рандомностью с помощью метода random.seed()

Псевдослучайные события в модуле `random` следуют из некой заданной начальной точки. То есть каждый раз, когда вы начинаете с одного и того же заданного состояния, события будут происходить одинаково. Мы можем использовать метод `random.seed()`, чтобы установить начальную точку — зерно.

Значение зерна должно быть *хешируемым*. Согласно документации (**https://docs.python.org/3.1/glossary.html**), «все неизменяемые встроенные объекты Python являются хешируемыми, а любые изменяемые контейнеры (такие как списки или словари) — нет». В данной программе мы обязаны иметь дело с целочисленным значением, поскольку тесты были написаны с применением целочисленных зерен. Когда вы создаете свои собственные программы, можно использовать строку или другой хешируемый тип.

По умолчанию в нашей программе в качестве зерна применяется специальное значение `None`, которое напоминает неопределенное состояние. Вызов метода `random.seed(None)` эквивалентен отсутствующему значению зерна, поэтому допустимо написать следующее:

```
random.seed(args.seed)
```

9.3.5. Итерирование с помощью функции range() и одноразовые переменные

Мы можем применить функцию `range()`, чтобы сгенерировать некоторое количество оскорблений. Поскольку нам не нужны числа, возвращаемые функцией `range()`, мы используем символ подчеркивания (`_`) в качестве имени переменной, указывая таким образом, что это одноразовое значение:

```
>>> num_insults = 2
>>> for _ in range(num_insults):
...     print('An insult!')
...
An insult!
An insult!
```

Символ нижнего подчеркивания — допустимое имя переменной в Python. Вы можете присвоить ей значение и использовать в коде:

```
>>> _ = 'You indistinguishable, filthsome carbuncle!'
>>> _
'You indistinguishable, filthsome carbuncle!'
```

Программисты на Python пришли к соглашению, что имя переменной в виде нижнего подчеркивания указывает на то, что ее значение не будет использоваться. Таким образом, если мы прибегнем к коду `for num in range(...)`, некоторые инструменты, например Pylint, обнаружат, что переменная `num` не используется в программе, и выведут сообщение о вероятной ошибке (которая вполне может возникнуть). Имя `_` указывает на то, что хранимое в такой переменной значение отбрасывается. Это важно для внешних инструментов, для других разработчиков, изучающих ваш код, да и для вас самих, когда спустя некоторое время откроете код своей программы.

Обратите внимание, что можно использовать несколько переменных с именем `_` в одной строке кода. Например, я могу распаковать кортеж с тремя элементами, чтобы получить среднее значение:

```
>>> x = 'Jesus', 'Mary', 'Joseph'
>>> _, name, _ = x
>>> name
'Mary'
```

9.3.6. Конструирование оскорблений

Чтобы создать список прилагательных, я применил метод `str.split()` к длинной многострочной строке, заключенной в тройные кавычки. На мой взгляд, это самый простой способ обработать все строки. Тройные кавычки позволяют нам добавлять разрывы строк, что недопустимо с одинарными кавычками:

```
>>> adjectives = """
... bankrupt base caterwauling corrupt cullionly detestable
... dishonest false filthsome filthy foolish foul gross heedless
... indistinguishable infected insatiate irksome lascivious
... lecherous loathsome lubbery old peevish rascaly rotten ruinous
... scurilous scurvy slanderous sodden-witted thin-faced toad-
... spotted unmannered vile wall-eyed
... """.strip().split()
>>> nouns = """
... Judas Satan ape ass barbermonger beggar block boy braggart
... butt carbuncle coward coxcomb cur dandy degenerate fiend
... fishmonger fool gull harpy jack jolthead knave liar lunatic maw
... milksop minion ratcatcher recreant rogue scold slave swine
... traitor varlet villain worm
... """.strip().split()
>>> len(adjectives)
36
>>> len(nouns)
39
```

Так как необходимо выбрать одно или несколько прилагательных, прекрасно подойдет функция `random.sample()`. Она вернет список элементов, случайно выбранных из предоставленного списка:

```
>>> import random
>>> random.sample(adjectives, k=3)
['filthsome', 'cullionly', 'insatiate']
```

Функция `random.choice()` подходит для выбора только одного элемента из списка, например существительного для оскорбления:

```
>>> random.choice(nouns)
'boy'
```

Далее необходимо соединить эпитеты с помощью строки `', '` (запятая и пробел), по аналогии, как мы делали со списком вещей для пикника в главе 3. Для этой цели идеально подойдет функция `str.join()`:

```
>>> adjs = random.sample(adjectives, k=3)
>>> adjs
['thin-faced', 'scurvy', 'sodden-witted']
>>> ', '.join(adjs)
'thin-faced, scurvy, sodden-witted'
```

Чтобы сконструировать оскорбление, мы можем объединить прилагательные и существительные внутри нашего шаблона с помощью f-строки:

```
>>> adjs = ', '.join(random.sample(adjectives, k=3))
>>> print(f'You {adjs} {random.choice(nouns)}!')
You heedless, thin-faced, gross recreant!
```

Ха! Теперь у меня есть удобный способ нажить врагов.

9.4. Прокачиваем навыки

— Сделайте так, чтобы программа считывала прилагательные и существительные из файлов, которые передаются в качестве аргументов.
— Добавьте тесты, проверяющие, правильно ли обрабатываются файлы и обидны ли новые оскорбления.

РЕЗЮМЕ

— Метод `parser.error()` применяется для вывода краткой справочной информации, сообщений о проблемах и завершения работы программы с кодом ошибки.
— В тройных кавычках, в отличие от одинарных и двойных, можно использовать разрывы строк.
— Метод `str.split()` подойдет для создания списка строковых значений из длинной строки.
— Метод `random.seed()` можно использовать для псевдослучайного выбора значений при каждом запуске программы.
— Методы `random.choice()` и `random.sample()` полезны для выбора одного или нескольких случайных элементов из списка вариантов соответственно.

Глава 10
Испорченный телефон: рандомные изменения строк

«В данном случае мы имеем отсутствие взаимопонимания».
— Капитан

Теперь, когда мы поэкспериментировали с модулем `random`, давайте попробуем случайным образом изменять строки. Задача интересная, поскольку в Python строки *неизменны*. Мы должны найти способ обойти это ограничение.

Чтобы реализовать свои идеи, мы напишем версию игры «Испорченный телефон», в которой секретное сообщение передается шепотом через цепочку людей. В процессе передачи оно неким непредсказуемым образом меняется. Последний человек, получивший сообщение, произносит его вслух, после чего оно сравнивается с исходным. Часто результаты бессмысленны и комичны.

Мы напишем программу *phone.py*, которая имитирует описанную выше игру. Программа напечатает "You said:" и исходный текст, а затем "I heard:" с измененной версией сообщения. Как и в упражнении из главы 5, исходный текст может извлекаться из командной строки:

```
$ ./telephone.py 'The quick brown fox jumps over the lazy dog.'
You said: "The quick brown fox jumps over the lazy dog."
I heard: "TheMquick brown fox jumps ovMr t: e lamy dog."
```

Или из файла:

```
$ ./telephone.py ../inputs/fox.txt
You said: "The quick brown fox jumps over the lazy dog."
I heard: "The quick]b'own fox jumps ovek the la[y dog."
```

Программа должна принимать опцию `-m` или `--mutations`, представленную вещественным числом в диапазоне от 0 до 1 со значением по умолчанию `0.1` (10%). Она определяет процент букв, которые следует изменить. Например, `.5` означает, что 50% букв должны быть изменены:

```
$ ./telephone.py ../inputs/fox.txt -m .5
You said: "The quick brown fox jumps over the lazy dog."
I heard: "F#eYquJsY ZrHnna"o. Muz/$ Nver t/Relazy dA!."
```

Поскольку мы используем модуль `random`, принимается целочисленное значение параметра `-s` или `--seed`, чтобы воспроизвести псевдослучайные выборки:

```
$ ./telephone.py ../inputs/fox.txt -s 1
You said: "The quick brown fox jumps over the lazy dog."
I heard: "The 'uicq brown *ox jumps over the l-zy dog."
```

На рис. 10.1 показана диаграмма программы.

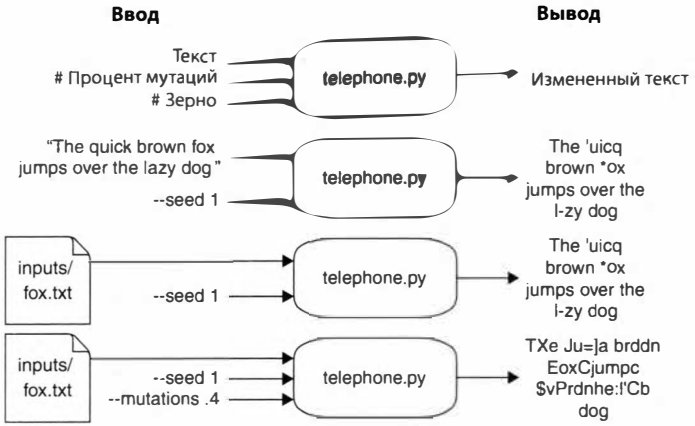


Рис. 10.1. Программа «Испорченный телефон» принимает текст и опционально процентное значение мутаций вместе с зерном случайности. На выходе появляется рандомно измененная версия исходного текста

Прочитав эту главу, вы научитесь:

— округлять числа;
— использовать модуль `string`;
— изменять строки и списки случайным образом.

10.1. Создание файла phone.py

Я советую использовать файл *new.py* для создания программы *phone.py* в каталоге *10_telephone*. Попробуйте сделать это из корня репозитория следующим образом:

```
$ ./bin/new.py 10_telephone/telephone.py
```

Вы также можете скопировать содержимое файла *template/template.py* в файл *10_telephone/telephone.py*. Измените код функции `get_args()` таким образом, чтобы вывод программы, запущенной с опцией `-h`, соответствовал показанному ниже. Я рекомендую определять тип значения процента мутаций как вещественное число. Сделать это можно с помощью кода `type=float`:

```
$ ./telephone.py -h
usage: telephone.py [-h] [-s seed] [-m mutations] text

Telephone

positional arguments:
  text                  Input text or file

optional arguments:
-h, --help              show this help message and exit
-s seed, --seed seed  Random seed (default: None)
-m mutations, --mutations mutations
                        Percent mutations (default: 0.1)
```

Теперь протестируйте программу. Она должна пройти как минимум два первых теста (файл *phone.py* существует и выводит справочную информацию при запуске с опцией `-h` или `--help`).

Следующие два теста проверяют, что параметры `-seed` и `--mutations` не принимают нечисловые значения. Это должно происходить автоматически, если вы определили данные параметры, явно

указав целое и вещественное числа, соответственно. Таким образом ваша программа обязана вывести вот что:

```
$ ./telephone.py -s blargh foo
usage: telephone.py [-h] [-s seed] [-m mutations] text
telephone.py: error: argument -s/--seed: invalid int value:
'blargh'
$ ./telephone.py -m blargh foo
usage: telephone.py [-h] [-s seed] [-m mutations] text
telephone.py: error: argument -m/--mutations: invalid float value:
'blargh'
```

Следующий тест проверяет, отклоняет ли программа значения параметра `--mutations`, находящиеся за пределами диапазона 0–1 (включительно). Такую проверку нелегко реализовать с помощью модуля `argparse`, поэтому я рекомендую вспомнить программу *abuse.py* из главы 9. Тогда мы вручную проверяли значения аргументов в функции `get_args()` и использовали функцию `parser.error()` для вывода ошибки. Обратите внимание, что значение параметра `--mutations`, равное `0`, допустимо, и в таком случае исходный текст выводится без изменений. Ваша программа должна выполнять следующее:

```
$ ./telephone.py -m -1 foobar
usage: telephone.py [-h] [-s seed] [-m mutations] text
telephone.py: error: --mutations "-1.0" must be between 0 and 1
```

Мы имеем дело с еще одной программой, принимающей входные данные либо из командной строки, либо из файла, поэтому, предполагаю, что вы вольны обратиться за подсказкой, взглянув на решение из главы 5. В функции `get_args()` можно использовать метод `os.path.isfile()`, чтобы определить, передал ли пользователь файл в качестве аргумента. Если это так, следует прочитать содержимое файла.

После того как все параметры приведены в порядок, выполните функцию `main()` с методом `random.seed()` и настройте вывод программы:

```
def main():
    args = get_args()
    random.seed(args.seed)
    print(f'You said: "{args.text}"')
    print(f'I heard: "{args.text}"')
```

Она должна обрабатывать текст из командной строки:

```
$ ./telephone.py 'The quick brown fox jumps over the lazy dog.'
You said: "The quick brown fox jumps over the lazy dog."
I heard: "The quick brown fox jumps over the lazy dog."
```

либо из файла, указанного пользователем:

```
$ ./telephone.py ../inputs/fox.txt
You said: "The quick brown fox jumps over the lazy dog."
I heard: "The quick brown fox jumps over the lazy dog."
```

Затем обязана выполняться функция `test_for_echo()`. Теперь перейдем к изменению входного текста и обсудим, как это сделать.

10.1.1. Вычисление процента мутаций

Количество букв, которые необходимо изменить, можно вычислить, умножив длину исходного текста на значение `args.mutations`. Если требуется изменить 20% символов в строке "The quick brown fox...", получается, что мы имеем дело с нецелым числом:

```
>>> text = 'The quick brown fox jumps over the lazy dog.'
>>> mutations = .20
>>> len(text) * mutations
8.8
```

Мы можем воспользоваться функцией `round()`, чтобы округлить значение до ближайшего целого числа. Выполнив команду `help(round)`, прочитайте документацию к этой команде и выясните, как округлять вещественные числа до определенного разряда:

```
>>> round(len(text) * mutations)
9
```

Обратите внимание, что вы также можете преобразовать вещественное число в целое с помощью функции `int()`, однако в этом случае вместо округления просто отсекается дробная часть числа:

```
>>> int(len(text) * mutations)
8
```

Полученное значение понадобится вам позже, поэтому давайте сохраним его в качестве значения переменной:

```
>>> num_mutations = round(len(text) * mutations)
>>> assert num_mutations == 9
```

10.1.2. Область мутаций

Что подставить взамен заменяемых символов? Давайте воспользуемся модулем `string`. Я рекомендую после импорта модуля выполнить команду `help(string)` и прочитать к нему документацию:

```
>>> import string
>>> help(string)
```

В качестве примера мы можем показанным ниже способом получить все строчные буквы ASCII. Обратите внимание, что мы имеем дело не с вызовом метода, так как в конце нет скобок `()`:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

Возвращается строка:

```
>>> type(string.ascii_lowercase)
<class 'str'>
```

Мы можем применить методы `string.ascii_letters` и `string.punctuation`, чтобы получить строки всех букв и знаков препинания. Для конкатенации строк пригодится оператор +. Воспользуемся данной строкой для случайного выбора символа под замену другим:

```
>>> alpha = string.ascii_letters + string.punctuation
>>> alpha
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
YZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Обратите внимание, что даже если использовать одно и то же зерно, мы получим разные результаты, так как буквы будут подставляться

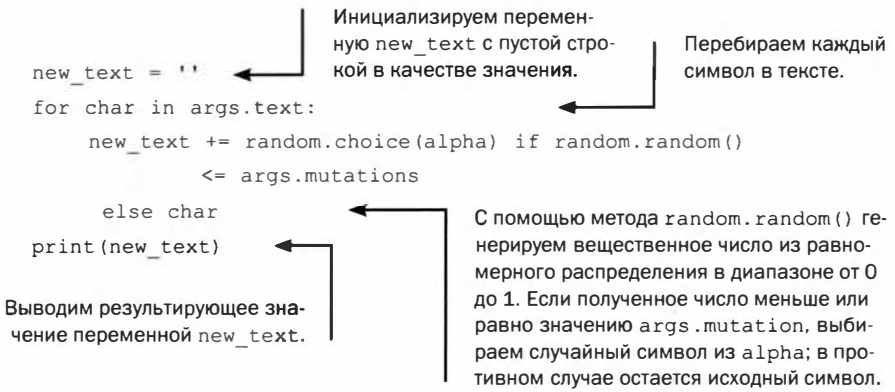
в разном порядке. Чтобы результаты были идентичны, нужно отсортировать символы в последовательном порядке.

10.1.3. Выбор символов под замену

Существует как минимум два подхода к выбору символов под замену: *детерминированный,* при котором результаты гарантированно будут одинаковыми, и *недетерминированный,* когда в вычисление результата вносится некая случайность. Сначала рассмотрим недетерминированный способ.

Недетерминированный подход

Один из способов выбрать символы, которые нужно изменить, — воспользоваться первым методом из главы 8. Мы можем перебирать все символы в тексте и на основе некоего случайного числа решать, оставить ли исходный символ или изменить его на некое случайное значение. Если данное случайное число меньше или равно значению процента мутаций, следует изменить символ:



Мы использовали метод `random.choice()` в файле *abuse.py* в главе 9 для рандомного выбора *одного* значения из списка вариантов. Мы можем прибегнуть к данному приему и здесь, чтобы выбрать символ из `alpha`, если результат работы метода `random.random()` попадает в диапазон значений `args.mutation` (которое, как мы знаем, представлено вещественным числом).

Проблема с данным подходом заключается в том, что к концу цикла `for` не гарантируется корректное количество изменений. То есть мы вычислили, что нужно изменить 9 символов из 44 при проценте мутаций в 20%. Ожидается, что с помощью этого кода в итоге изменится примерно пятая часть символов, потому что рандомное значение

из равномерного распределения в диапазоне от 0 до 1 должно быть меньше или равно 0,2 примерно в 20% случаев. То есть в итоге может измениться как 8 символов, так и 10. Из-за такой неопределенности данный подход считается *недетерминированным*.

Тем не менее это реально полезный прием, который следует запомнить. Представьте, что у вас на входе файл с миллионами или даже миллиардами строк текста, и вы хотите случайным образом выбрать, допустим, 10% строк. Упомянутый подход окажется достаточно быстрым и точным. Бóльший размер выборки позволит получить более точное количество мутаций.

Случайная выборка символов

Детерминированный подход в случае с файлом, содержащим миллионы/миллиарды строк, потребовал бы сначала прочитать весь ввод, чтобы подсчитать количество строк и определить, какие строки обрабатывать, а затем вернуться к файлу вновь для извлечения этих строк. Данный подход займет *гораздо* больше времени, чем недетерминированный метод, описанный выше. В зависимости от величины размера входного файла, кода программы и объема оперативной памяти компьютера программа может тормозить и даже сбоить.

```
>>> 'foo'.replace('o', 'a')
'faa'
```

В нашем случае входных данных довольно мало, поэтому мы воспользуемся детерминированным способом, поскольку среди его преимуществ — точность и тестируемость. Однако вместо того, чтобы делать акцент на строках текста, мы рассмотрим индексы символов. В главе 8 вы опробовали метод `str.replace()`, который позволяет менять все экземпляры одной строки на другую:

```
>>> text
'The quick brown fox jumps over the lazy dog.'
>>> len(text)
44
```

Сейчас нам не поможет метод `str.replace()`, потому что так мы изменим *каждое* вхождение выбранного символа, а нам нужно изменить лишь отдельные экземпляры разных символов. Поэтому мы

будем использовать метод `random.sample()` для выбора символов текста с определенными индексами. Первый аргумент метода `random.sample()` представляет собой нечто вроде списка. И мы можем указать диапазон чисел с учетом длины нашего текста.

Предположим, текст содержит 44 символа:

```
>>> text
'The quick brown fox jumps over the lazy dog.'
>>> len(text)
44
```

Вот так мы способны использовать функцию `range()` для конструирования списка чисел от 0 до 44:

```
>>> range(len(text))
range(0, 44)
```

Обратите внимание, что функция `range()` ленива. В результате в REPL-интерфейсе она не выдаст 44 значения, пока мы не попросим ее об этом, применив функцию `list()`:

```
>>> list(range(len(text)))
```

Ранее мы подсчитали, что значение `num_mutations` для изменения 20% символов в тексте равно 9. Вот один из вариантов выборки индексов, символы которых можно изменить:

```
>>> indexes = random.sample(range(len(text)), num_mutations)
>>> indexes
[13, 6, 31, 1, 24, 27, 0, 28, 17]
```

Я предлагаю использовать цикл `for` для перебора каждого значения с выбранными индексами:

```
>>> for i in indexes:
...     print(f' {i:2} {text[i]}')
...
13 w
 6 i
31 t
 1 h
24 s
```

```
27 v
 0 T
28 e
17 o
```

Нужно заменить каждый символ с выбранным индексом рандомным символом из `alpha`:

```
>>> for i in indexes:
...     print(f' {i:2} {text[i]} changes to {random.choice(alpha)}')
...
13 w changes to b
 6 i changes to W
31 t changes to B
 1 h changes to #
24 s changes to d
27 v changes to:
 0 T changes to C
28 e changes to %
17 o changes to,
```

Давайте еще учитывать тот момент, что замещающий символ не должен совпадать с исходным. Сможете разобраться, как получить подмножество `alpha`, которое *не* включает символ в позиции?

10.1.4. Изменение строки

Строковые значения переменных в Python *неизменны*, то есть мы не можем изменять их напрямую. Например, допустим, нам требуется изменить символ `'w'` в позиции `13` на `'b'`. Было бы удобно сделать это с помощью кода `text[13]`, но так будет создано исключение:

```
>>> text[13] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Единственный способ изменить строковое значение переменной — это перезаписать его новой строкой, то есть присвоить абсолютно новое значение. Мы можем создать новую строку, как показано на рис. 10.2, используя следующие данные.

1. Часть строки до позиции заменяемого символа.
2. Случайно выбранный символ из подмножества `alpha`.
3. Часть строки после позиции заменяемого символа.

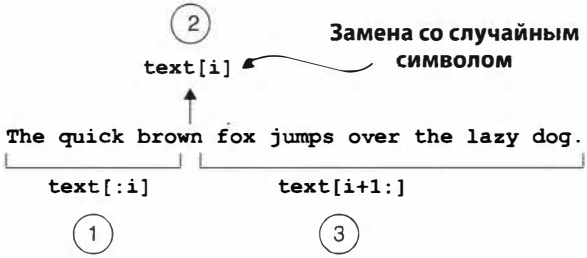


Рис. 10.2. Создание новой строки из части строки до позиции заменяемого символа, нового символа и части строки после позиции заменяемого символа.

В части данных 1 и 3 можно использовать *срезы строк*. Например, если индекс символа `i` равен 13, срез перед ним выполняется так:

```
>>> text[:13]
'The quick bro'
```

Соответственно, срез после него выполняется следующим образом:

```
>>> text[14:]
'n fox jumps over the lazy dog.'
```

Используя три части данных, перечисленных выше, цикл `for` выглядит так:

```
for i in index:
    text = 1 + 2 + 3
```

Понятно объяснил?

10.1.5. Самостоятельная работа

Ладно, урок окончен. Объявляется перемена. А потом возвращайтесь и напишите программу. Протестируйте ее. Решайте задачи по очереди. Я верю в вас.

10.2. Решение

Насколько ваше решение отличается от моего? Давайте рассмотрим один из вариантов программы, успешно проходящий все тесты:

```
#!/usr/bin/env python3
"""Испорченный телефон"""

import argparse
import os
import random
import string

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Telephone',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='text', help='Input text
                        or file')

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='seed',
                        type=int,
                        default=None)

    parser.add_argument('-m',
                        '--mutations',
                        help='Percent mutations',
                        metavar='mutations',
                        type=float,
                        default=0.1)

    args = parser.parse_args()
```

Импортируем модуль `string`, необходимый для выбора случайного символа.

Определяем позиционный аргумент. Это может быть строка текста или файл, содержимое которого нужно прочитать.

Параметр `--seed` представляет собой целое число с дефолтным значением `None`.

Параметр `--mutations` представляет собой вещественное число со значением по умолчанию `0.1`.

Обрабатываем аргументы из командной строки. Если модуль `argparse` определяет проблему, например нечисловое значение зерна или процента мутаций, программа завершает работу и выводится сообщение об ошибке. Если вызов выполнен успешно, значит, модуль `argparse` проверил аргументы и преобразовал значения.

Если значение `args.mutations` не находится в допустимом диапазоне от 0 до 1, используем метод `parser.error()`, чтобы завершить работу программы и вывести указанное сообщение. Обратите внимание на обратную связь для вывода пользователю неверного значения `args.mutation`.

```
    if not 0 <= args.mutations <= 1:
        parser.error(f'--mutations "{args.mutations}" must be
            between 0 and 1')

    if os.path.isfile(args.text):
        args.text = open(args.text).read().rstrip()

    return args
```

Если `args.text` ссылается на существующий файл, считываем содержимое этого файла и перезаписываем исходное значение `args.text`.

Возвращаем обработанные аргументы вызывающей стороне.

```
# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    text = args.text
    random.seed(args.seed)
    alpha = ''.join(sorted(string.ascii_letters
                           + string.punctuation))
    len_text = len(text)
    num_mutations = round(args.mutations * len_text)
    new_text = text

    for i in random.sample(range(len_text), num_mutations):
        new_char = random.choice(alpha.replace(new_text[i], ''))
        new_text = new_text[: i] + new_char + new_text[i + 1:]

    print(f'You said: "{text}"\nI heard: "{new_text}"')

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Передаем методу `random.seed()` пользовательское значение. Напоминаю, что по умолчанию для `args.seed` используется значение `None`, что идентично тому, что зерно не задано.

Создаем копию текста.

Выводим текст.

Перезаписываем текст, объединив его фрагмент перед символом с текущим индексом, значение переменной `new_char` и фрагмент после символа с текущим индексом.

Вычисляем значение `num_mutations`, умножив процент мутаций на длину текста.

Применяем метод `random.choice()` для выбора символа `new_char` из строки, созданной путем замены текущего символа (`text[i]`) в переменной `alpha` на заглушку. Так гарантируется, что новый символ не будет идентичен исходному.

Поскольку мы используем функцию `len(text)` более одного раза, присваиваем ее результат переменной.

Используем метод `random.sample()`, чтобы выбрать индексы `num_mutations`, символы с которыми будут изменены. В результате возвращается список, который мы можем перебирать с помощью цикла `for`.

Присваиваем переменной `alpha` значение, содержащее символы, которыми мы воспользуемся для замены. Функция `sorted()` возвращает новый список символов в нужном порядке, а затем метод `str.join()` преобразует его обратно в строку.

10.3. Обсуждение

Функция `get_args()` вам уже знакома. Аргумент `--seed` — это значение типа `int`, которое мы передаем функции `random.seed()`, чтобы контролировать рандомность в целях тестирования. Дефолтное значение зерна — `None`, поэтому мы можем вызывать метод `random.seed(args.seed)`, где `None` эквивалентно отсутствию значения. Параметр `--mutations` — это вещественное число с рациональным значением по умолчанию, и метод `parser.error()` выводит сообщение об ошибке, если значение не находится в указанном диапазоне. Как и в других программах, проверяется, откуда поступают входные данные, из командной строки или из файла, и в последнем случае считывается содержимое файла.

10.3.1. Изменение строки

Как уже обсуждалось, мы не можем просто изменить строку, хранящуюся в переменной `text`:

```
>>> text = 'The quick brown fox jumps over the lazy dog.'
>>> text[13] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Необходимо создать *новую* строку, используя текст до и после символа `i`. Строку мы делаем из фрагментов с помощью кода `text[start:stop]`. Если не указать значение `start`, Python начнет извлечение с позиции 0 (т. е. с начала строки), а если опустите `stop`, интерпретатор доберется до конца строки. Код `text[:]` выбирает всю строку полностью.

Если переменной `i` присвоено значение `13`, фрагмент строки до этой позиции будет выглядеть так:

```
>>> i = 13
>>> text[: i]
'The quick bro'
```

А фрагмент после `i+1` так:

```
>>> text[i+1:]
'n fox jumps over the lazy dog.'
```

Теперь о том, что мы помещаем в серединку. Нужно использовать метод `random.choice()` для выбора символа из последовательности `alpha`, содержащей комбинацию всех букв ASCII и знаков препинания *без* исходного символа. Я использую метод `str.replace()`, чтобы избавиться от исходной буквы:

```
>>> alpha = ''.join(sorted(string.ascii_letters + string.
punctuation))
>>> alpha.replace(text[i], '')
'!"#$%&\'()*+,-./:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvxyz{|}~'
```

Затем я получаю новую букву, которая не совпадает с исходной:

```
>>> new_char = random.choice(alpha.replace(text[i], ''))
>>> new_char
'Q'
```

Существует много способов объединить несколько строк в одну. Оператор +, пожалуй, самый простой из них:

```
>>> text = text[: i] + new_char + text[i+1:]
>>> text
'The quick broQn fox jumps over the lazy dog.'
```

Операция повторяется для каждого индекса в `random. sample()`, постоянно перезаписывая значение переменной `text`. После завершения цикла `for` исходная строка может быть выведена в измененном виде.

10.3.2. Списки вместо строк

Строки не изменяются, а вот списки да. Вы видели, что код в духе `text[13] = 'b'` выбрасывает исключение, но мы можем преобразовать значение переменной `text` в список и изменить его с помощью указанного кода:

```
>>> text = list(text)
>>> text[13] = 'b'
```

Затем мы способны превратить данный список обратно в строку, присоединив его к пустой строке:

```
>>> ''.join(text)
'The quick brobn fox jumps over the lazy d●g.'
```

Ниже показана версия функции `main()`, реализующая подобный подход:

```
def main():
    args = get_args()
    text = args.text
    random.seed(args.seed)
    alpha = ''.join(sorted(string.ascii_letters
                           + string.punctuation))
    len_text = len(text)
    num_mutations = round(args.mutations * len_text)
    new_text = list(text)

    for i in random.sample(range(len_text), num_mutations):
        new_text[i] = random.choice(alpha.replace(new_text[i], ''))

    print('You said: "{}"\nI heard: "{}"'.format(text,
          ''.join(new_text)))
```

Инициализируем `new_text` как список из исходного текста.

Теперь мы можем напрямую изменить символы в списке `new_text`.

Присоединяем значение переменной `new_list` к пустой строке, чтобы создать новую строку.

Особой разницы между двумя продемонстрированными подходами нет, но я выбрал бы второй вариант, потому что не люблю возиться с нарезкой строк. Мне гораздо проще и удобнее менять списки, чем многократно нарезать и собирать строки.

Мутации в ДНК

Как бы то ни было, наша программа имитирует (отчасти) мутации ДНК, происходящие с течением времени. Механизм копирования ДНК допускает ошибки, и хаотично возникают мутации. Зачастую они не оказывают пагубного воздействия на организм.

В нашем примере одни символы заменяются другими — в биологии это называется «точечными мутациями», «однонуклеотидными вариациями» (SNV) или «однонуклеотидным полиморфизмом» (SNP). Как вариант, мы могли бы написать версию программы, которая рандомно

удаляла бы или вставляла новые символы, выполняла «вставку/удаление». Мутации (не приводящие к гибели организма) возникают с определенной частотой, поэтому подсчет количества мугаций в консервативных последовательностях ДНК любых двух организмов позволяет оценить, как давно они отделились от общего предка.

10.4. Прокачиваем навыки

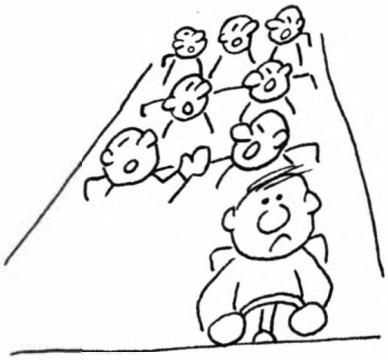
— Примените мутации к случайно выбранным словам, а не ко всей строке.
— Помимо мутаций выполняйте «вставки/удаления». Включите опциональные аргументы для указания процента каждой операции и выбора частоты добавления/удаления символов.
— Добавьте опцию для `-o` или `--output`, позволяющую указать имя файла для записи результирующего текста. По умолчанию результат следует выводить в поток `STDOUT`.
— Используйте флаг для ограничения замещающих символов буквами (без знаков пунктуации).
— Добавьте для каждой новой функции необходимые тесты в файл *test.py* и проверьте правильность работы вашей программы.

РЕЗЮМЕ

— Строку нельзя изменять напрямую. Однако переменная, содержащая строку, может перезаписываться новыми значениями.
— Списки изменяются напрямую, поэтому иногда полезно использовать функцию `list()` на строке и превращать ее в список. Затем изменять полученный список и с помощью метода `str.join()` превращать его обратно в строку.
— Модуль `string` имеет удобные функции для обработки строк.

Глава 11

99 бутылок пива: разработка и тестирование функций



Мало какие песни так раздражают, как «99 бутылок пива». Надеюсь, вам никогда не приходилось стоять в пробке в автобусе со школьниками, которые во всю мощь поют эту песню. У меня был такой опыт.

Воспользуемся этой простенькой песней для нашего упражнения и напишем алгоритм, генерирующий ее текст. Так мы сможем поэкспериментировать с подсчетом в обе стороны, с форматированием строк и — внимание, новинка — с написанием функций и тестов для них!

Наша программа будет носить имя *bottles.py*, принимать один параметр, `-n` или `--num`, представленный ***положительным*** целым числом (по умолчанию, `10`), и выводить все куплеты песенки с количеством бутылок от `--num` до 1. Между каждым куплетом должно быть два символа перевода строки, чтобы визуально их разделить, и только один символ перевода строки после последнего куплета (с одной бутылкой).

Причем нужно, чтобы в нем выводился текст «Нет бутылок пива на стене!», а не «0 бутылок пива на стене»:

```
$ ./bottles.py -n 3
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!

2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

Справившись с данной главой, вы научитесь:

— составлять списки чисел в обратном порядке;
— писать функции для создания куплета песни и проверять его корректность с помощью теста;
— использовать циклы `for` как представления списков, которые, в свою очередь, могут быть записаны с помощью функции `map()`.

11.1. Создание файла bottles.py

Мы будем работать в каталоге *11_bottles_of_beer*. Скопируйте содержимое файла *template.py* или используйте файл *new.py*, чтобы создать программу *bottles.py*. Затем измените код функции `get_args()` так, чтобы программа выводила показанные ниже справочные инструкции. Еще нужно определить целочисленную опцию `--num` с дефолтным значением `10`:

```
$ ./bottles.py -h
usage: bottles.py [-h] [-n number]

Bottles of beer song
```

```
optional arguments:
-h, --help            show this help message and exit
-n number, --num number
                      How many bottles (default: 10)
```

Если в качестве аргумента `—num` передано значение, отличное от целого числа, программа должна вывести сообщение об ошибке и завершить работу с ее кодом. Так должно происходить автоматически, если вы корректно определили свой параметр для модуля `argparse`:

```
$ ./bottles.py -n foo
usage: bottles.py [-h] [-n number]
bottles.py: error: argument -n/--num: invalid int value: 'foo'
$ ./bottles.py -n 2.4
usage: bottles.py [-h] [-n number]
bottles.py: error: argument -n/--num: invalid int value: '2.4'
```

Поскольку невозможно спеть ноль и меньше куплетов, необходимо проверить, является ли значение параметра `--num` меньше `1`. Для решения данной задачи предлагаю, как и в предыдущих проектах, воспользоваться методом `parser.error()` в функции `get_args()`:

```
$ ./bottles.py -n 0
usage: bottles.py [-h] [-n number]
bottles.py: error: --num "0" must be greater than 0
```

На рис. 11.1 показана диаграмма входных и выходных данных.

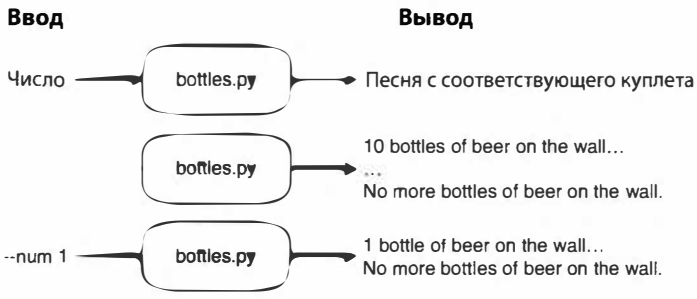
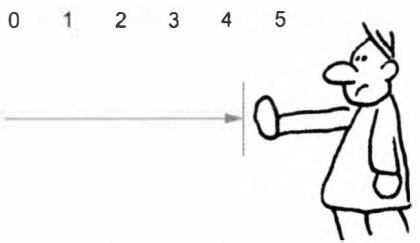


Рис. 11.1. Программа bottles.py принимает количество бутылок или, по умолчанию, начинает с куплета с 10 бутылками

11.1.1. Обратный отсчет

Песня начинается с переданного количества бутылок, `--num`, например 10, и поется с обратным отсчетом — 9, 8, 7 и т. д. Как реализовать это в Python? Ранее мы использовали функцию диапазона `range(start, stop)`, генерируя список целых чисел, значение которых *увеличивается*. Если вы передадите ей только одно число, оно будет принято в качестве параметра `stop`, а в качестве значения `start` применено число 0:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```



Поскольку мы имеем дело с ленивой функцией, в REPL-интерфейсе нужно использовать функцию `list()`, чтобы вывести результирующие числа. Помните, что значение `stop` не включается в вывод, поэтому в результате функция завершит вычисления числом 4, а не 5.

Если вы передадите функции `range()` два числа, они будут считаться начальным и конечным значениями, то есть `start` и `stop`:

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

Чтобы обратить последовательность чисел, вам может прийти в голову поменять местами значения `start` и `stop`. К сожалению, если значение `start` будет больше значения `stop`, функция выведет пустой список:

```
>>> list(range(5, 1))
[]
```

В главе 3 мы рассматривали, как использовать функцию `reversed()` для обращения списка. Она тоже ленивая, поэтому в REPL-интерфейсе я вновь прибегну к функции `list()` для принудительного вывода значений:

```
>>> list(reversed(range(1, 5)))
[4, 3, 2, 1]
```

Функция `range()` принимает опциональный третий аргумент, `step`, задающий шаг. Например, можно использовать шаг, равный пяти:

```
>>> list(range(0, 50, 5))
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

Другой способ обратного отсчета — поменять местами значения `start` и `stop` и применить значение `-1` в качестве шага:

```
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
```

Итак, теперь вы знаете несколько способов счета в обратном порядке.

11.1.2. Создание функции

До этого момента мы помещали весь код в функцию `main()`. И вот перед нами первое упражнение, в котором мы напишем отдельную функцию. Задумайтесь, как написать ее так, чтобы программа «пела» *только один куплет*. Функция может принимать количество бутылок (считайте, номер куплета) и возвращать соответствующий текст куплета.

Начнем с примера, показанного на рис. 11.2. Ключевым словом `def` «определяется» функция, после чего следует имя функции. Имена функций могут содержать только латинские буквы, цифры и знаки подчеркивания и не могут начинаться с цифры. После имени указываются круглые скобки, в которых содержатся параметры, принимаемые функцией. Наша функция будет называться `verse()` и принимать параметр `bottle` (или `number`, любое имя по вашему желанию). После параметров в скобках следует двоеточие, обозначающее конец *определения*. Затем с новой строки идет *тело* функции, все строки которого имеют отступ, кратный четырем пробелам.

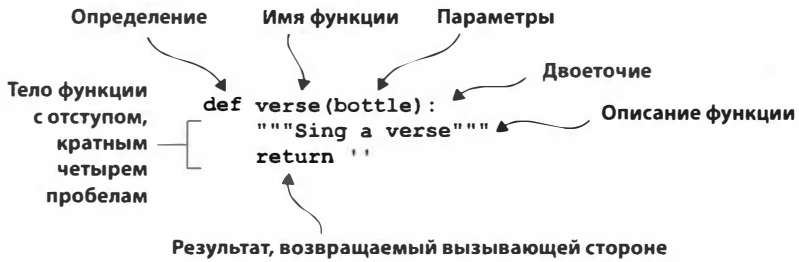


Рис. 11.2. Структура функции с определением и телом

Строка описания или документации (также называемая docstring) на рис. 11.2 располагается сразу после определения функции. Она отображается в справочной информации к ней.

Вы можете написать данную функцию в REPL-интерфейсе:

```
>>> def verse(bottle):
...     """Sing a verse"""
...     return ''
...
>>> help(verse)
```

Выполнив команду `help(verse)`, вы увидите следующий вывод с описанием функции:

```
Help on function verse in module __main__:

verse(bottle)
    Sing a verse
```

Оператор `return` сообщает Python, что следует вернуть из функции. Сейчас это не очень понятно, так как функция возвращает пустую строку:

```
>>> verse(10)
''
```

Еще программисты часто используют оператор `pass` в теле фиктивной функции. Он ничего не делает, и функция возвращает значение `None` вместо пустой строки, если ее выполнить так, как показано выше. В процессе разработки своих функций и тестов к ним вам понадобится оператор `pass`, чтобы заглушить новую функцию, пока вы не решите, что она будет делать.

11.1.3. Тестирование функции verse()

Придерживаясь концепции разработки *через тестирование,* прежде чем двигаться дальше, давайте напишем тест для функции `verse()`. Ниже показан код такого теста. Добавьте его в свою программу *bottles.py* сразу после функции `main()`:

```
def verse(bottle):
    """Sing a verse"""

    return ''

def test_verse():
    """Test verse"""

    last_verse = verse(1)
    assert last_verse == '\n'.join([
        '1 bottle of beer on the wall,', '1 bottle of beer,',
        'Take one down, pass it around,',
        'No more bottles of beer on the wall!'
])

    two_bottles = verse(2)
    assert two_bottles == '\n'.join([
        '2 bottles of beer on the wall,', '2 bottles of beer,',
        'Take one down, pass it around,', '1 bottle of beer on
         the wall!'
])
```

Существует множество способов написать данную программу. У меня на примете есть функция `verse()`, генерирующая один куплет песни, возвращая новое значение типа `str`, в котором строки куплета соединены символами перевода строки.

Вам необязательно следовать моему примеру, но я бы хотел, чтобы вы продумали, как написать функцию и *модульный тест* к ней. Если вы осведомлены о тестировании ПО, то, возможно, знаете, что существуют разные определения «модуля» кода. В данной книге я рассматриваю *функцию* как *модуль*, поэтому мои модульные тесты — это тесты отдельных функций.

Несмотря на то что в песне потенциально могут быть сотни куплетов, два наших теста должны охватывать все, что нуждается в проверке. Взгляните на ноты песни ниже (рис. 11.3). Рисунок отлично отражает структуру песни и, следовательно, нашей программы.

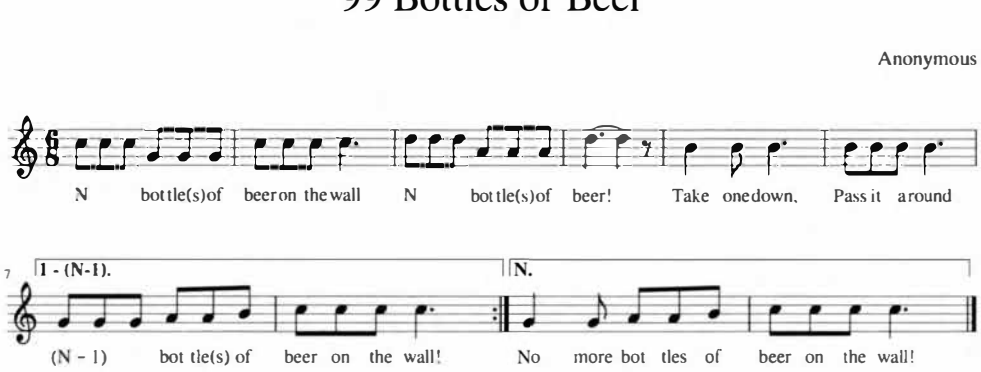


Рис. 11.3. По нотам песни «99 бутылок пива» понятно, что нужно обработать два случая: один для всех куплетов кроме последнего и один — для последнего

Я внес в нотную запись некоторые элементы, отражающие идеи программирования. Если вы не учились в музыкальной школе, позвольте мне кратко объяснить важные моменты. N — это текущее число бутылок, скажем «99», так что `(N-1)` будет равняться «98». Концовки куплетов отмечены строкой `1- (N-1)`, что слегка сбивает с толку, потому что мы используем дефис для обозначения как диапазона, так и операции вычитания в одном и том же «уравнении». Тем не менее первая концовка присутствует во всех куплетах, кроме последнего. Двоеточие перед тактовой чертой в первой концовке означает повтор песни с самого начала. В последнем куплете используется концовка N, а двойная тактовая черта указывает на окончание песни/программы.

По нотам видно, что необходимо обработать лишь два случая: последний куплет и все остальные. Сначала мы проверяем последний куплет. Мы ищем текст «1 bottle» (единственное число), а не «1 bottles» (множественное число). Также нужно проверить, чтобы в последней строке было написано «No more bottles» вместо «0 bottles». Второй тест для «нескольких бутылок пива» проверяет, идет ли сначала «2 bottles», а затем «1 bottle». Если удастся успешно пройти оба теста, наша программа сможет обрабатывать все куплеты.

Я написал функцию `test_verse()` для проверки только функции `verse()`. Имя функции важно, так как с помощью модуля `pytest` я ищу в коде все функции, имя которых начинается с `test_`, и запускаю их. Если в вашей программе *bottles.py* есть описанные выше функции `verse()` и `test_verse()`, можно протестировать файл *bottles.py*. В результате вы увидите примерно следующее:

```
$ pytest bottles.py
======================= test session starts ======================
...
collected 1 item

bottles.py F
[100%]

============================ FAILURES ============================
_____________________________ test_verse _____________________________

    def test_verse():
        """Test verse"""

        last_verse = verse(1)
>       assert last_verse == '\n'.join([
            '1 bottle of beer on the wall,', '1 bottle of beer,',
            'Take one down, pass it around,',
            'No more bottles of beer on the wall!'
])
E       AssertionError: assert '' == '1 bottle of beer on the
     wal...ottles of beer on the wall!'
E         + 1 bottle of beer on the wall,
E         + 1 bottle of beer,
E         + Take one down, pass it around,
E         + No more bottles of beer on the wall!

bottles.py:49: AssertionError
==================== 1 failed in 0.10 seconds ====================
```

Вызываем функцию `verse()` с аргументом `1`, чтобы сгенерировать последний куплет песни.

Символ > в начале данной строки указывает на источник ошибки. В рамках теста проверяется, присвоено ли переменной `last_verse` ожидаемое строковое значение. Поскольку это не так, данная строка выбрасывает исключение, приводящее к сбою утверждения.

Строки с буквой `E` отражают разницу между полученными и ожидаемыми данными. Переменной `last_verse` присвоена пустая строка (`''`), которая не соответствует ожидаемой строке «1 bottle of beer...» и т. д.

Чтобы пройти первый тест, можно скопировать ожидаемое значение `last_verse` прямо из теста. Измените код функции `verse()` следующим образом:

```
def verse(bottle):
    """Sing a verse"""
```

```
    return '\n'.join([
        '1 bottle of beer on the wall,', '1 bottle of beer,',
        'Take one down, pass it around,',
        'No more bottles of beer on the wall!'
])
```

Теперь снова запустите тест. Первый тест должен быть завершен успешно, а второй — провален. Ниже показаны соответствующие строки ошибок:

```
============================ FAILURES ============================
_____________________________ test_verse _____________________________

    def test_verse() -> None:
        """Test verse"""

        last_verse = verse(1)
        assert last_verse == '\n'.join([
            '1 bottle of beer on the wall,', '1 bottle of beer,',
            'Take one down, pass it around,',
            'No more bottles of beer on the wall!'
])

        two_bottles = verse(2)
>       assert two_bottles == '\n'.join([
          '2 bottles of beer on the wall,', '2 bottles of beer,',
          'Take one down, pass it around,', '1 bottle of beer on
        the wall!'
])
E       AssertionError: assert '1 bottle of ... on the wall!' ==
      '2 bottles of ... on the wall!'
E         - 1 bottle of beer on the wall,
E         ? ^
E         + 2 bottles of beer on the wall,
E         ? ^       +
E         - 1 bottle of beer,
E         ? ^
E         + 2 bottles of beer,...
E
E         ...Full output truncated (7 lines hidden), use '-vv' to show
```

Теперь данный тест завершается успешно.

Вызываем функцию `verse()` со значением 2, чтобы сгенерировать куплет «2 bottles...».

Утверждаем, что число в куплете эквивалентно ожидаемой строке.

В строках с буквой `E` заключается проблема. Функция `verse()` вернула `'1 bottle'`, а тест ожидал `'2 bottles'` и т. д.

Вернитесь к определению функции `verse()`. Взгляните на рис. 11.4 и подумайте, какие части кода нужно изменить. Первую, вторую и четвертую строки. Третья строка не меняется. Программа получает значение параметра `bottle`, которое, в зависимости от значения, используется в первых двух строках вместе со словом «bottles» или «bottle». (Подсказка: для значения `1` используется только единственное число; во всех остальных случаях — множественное число.) В четвертой строке нужно указать значение `bottle - 1` и, опять же, слово «bottles» или «bottle» в зависимости от полученного значения. Сможете написать подходящий код?

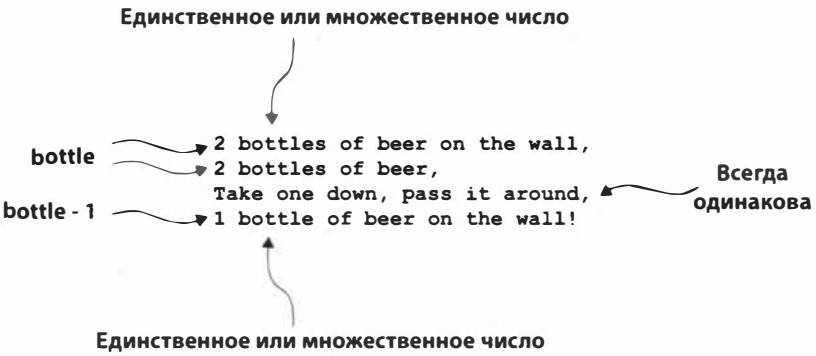


Рис. 11.4. Каждый куплет состоит из четырех строк, из которых первые две и последняя очень похожи. Третья строка не меняется. Определите части строк, которые различаются

Пройдите оба теста, прежде чем переходить к выводу всей песни целиком. В результате тестирования вы должны увидеть следующее:

```
$ pytest bottles.py
================== test session starts ====================
...

collected 1 item

bottles.py.                                              [100%]

=================== 1 passed in 0.05 seconds ===============
```

11.1.4. Использование функции verse()

К настоящему моменту вы узнали, что:

- значение `--num` представлено целым числом выше 0;
- программа выполняет обратный отсчет от значения `--num` до 0;
- функция `verse()` корректно выводит любой отдельный куплет.

Теперь вам нужно собрать все эти моменты вместе. Я предлагаю начать с цикла `for` с функцией `range()` для обратного отсчета. Применяйте каждое полученное значение для создания куплета с помощью функции `verse()`. После каждого куплета, кроме последнего, нужно использовать два символа перевода строки.

Примените команду `pytest -xv test.py` (или `make test`), чтобы протестировать программу на данном этапе. На языке программистов *test.py* — это *интеграционный тест*, поскольку он проверяет, работает ли программа *в целом*. С данного момента мы сосредоточимся на создании *модульных тестов* для проверки отдельных функций. Но и не забудем про интеграционные тесты, чтобы убедиться, что все функции исправно работают вместе.

Успешно пройдя набор тестов, попробуйте переписать код цикла `for`, используя представление списка или функцию `map()`. Вместо того, чтобы писать код с ноля, вы можете закомментировать рабочий код, добавив символ # в начало комментируемых строк, а затем опробовать другие способы написания алгоритма. Тестируйте программу в процессе разработки. Для того чтобы замотивировать вас, скажу, что мое решение состоит из одной строки. Получится ли у вас для получения ожидаемого результата объединить в одной строке кода функции `range()` и `verse()`?

Приведу несколько советов.

- Определите аргумент `--num` как целое число с дефолтным значением `10`.
- Используйте метод `parser.error()` модуля `argparse` для вывода сообщения об ошибке, если передано отрицательное значение параметра `--num`.
- Напишите функцию `verse()`. С помощью функции `test_verse()` и фреймворка `pytest` проверьте корректность кода.
- Объедините функции `verse()` с `range()` для генерации всех куплетов.

Попробуйте сначала создать свою программу, прежде чем читать мое решение. Кроме того, вы можете решить задачу совершенно иначе и даже написать собственные модульные тесты.

11.2. Решение

Я решил показать вам необычную версию программы с функцией `map()`. Позже я расскажу, как решить ту же задачу с помощью цикла `for` и представления списка.

```
#!/usr/bin/env python3
"""99 бутылок пива"""

import argparse

# --------------------------------------------------
def get_args():
"""Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Bottles of beer song',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-n',
                        '--num',
                        metavar='number',
                        type=int,
                        default=10,
                        help='How many bottles')

    args = parser.parse_args()

    if args.num < 1:
        parser.error(f'--num "{args.num}" must be greater than 0')

    return args

# --------------------------------------------------
def main():
    """Да грянет джаз!"""
```

Определяем аргумент `--num` как целое число с дефолтным значением `10`.

Разбираем аргумент командной строки и сохраняем в переменную `args`.

Если значение `args.num` меньше `1`, используем метод `parser.error()` для вывода сообщения об ошибке и выхода из программы с кодом ошибки.

`map()` ожидает функцию в качестве первого аргумента и итерируемый объект в качестве второго. Последовательность чисел в обратном порядке передается из функции `range()` в функцию `verse()`. Результатом функции `map()` является новый список куплетов, которые соединяются двумя символами перевода строки.

```
    args = get_args()
    print('\n\n'.join(map(verse, range(args.num, 0, -1))))

# --------------------------------------------------
def verse(bottle):
    """Поем куплет"""

    next_bottle = bottle - 1
    s1 = '' if bottle == 1 else 's'
    s2 = '' if next_bottle == 1 else 's'
    num_next = 'No more' if next_bottle == 0 else next_bottle
    return '\n'.join([
        f' {bottle} bottle{s1} of beer on the wall,',
        f' {bottle} bottle{s1} of beer,',
        f'Take one down, pass it around,',
        f' {num_next} bottle{s2} of beer on the wall!',
    ])

# --------------------------------------------------
def test_verse():
    """Test verse"""

    last_verse = verse(1)
    assert last_verse == '\n'.join([
        '1 bottle of beer on the wall,', '1 bottle of beer,',
        'Take one down, pass it around,',
        'No more bottles of beer on the wall!'
    ])

    two_bottles = verse(2)
    assert two_bottles == '\n'.join([
        '2 bottles of beer on the wall,', '2 bottles of beer,',
        'Take one down, pass it around,', '1 bottle of beer on
         the wall!'
    ])
```

Определяем переменную `s1` (первое окончание «s»), хранящую либо букву «s», либо пустую строку в зависимости от текущего значения переменной `bottle`.

Определяем значение переменной `next_bottle`, которое на единицу меньше текущего значения переменной `bottle`.

Определяем функцию, создающую один куплет.

Повторяем то же самое с переменной `s2` (второе окончание «s»), содержимое которой зависит от значения переменной `next_bottle`.

Определяем значение переменной `next_num` в зависимости от того, равно следующее значение `0` или нет.

Создаем результат, который будет возвращен, путем соединения четырех строк текста в одну с символами перевода строки. Подставляем значения переменных, чтобы сформировать корректный куплет.

Определяем модульный тест `test_verse()` для функции `verse()`. Префикс `test_` позволяет модулю `pytest` обнаружить и выполнить данную функцию.

Тестируем функцию `last_verse()` со значением 1.

Тестируем функцию `verse()` со значением 2.

```
# --------------------------------------------------
if __name__ == '__main__':
    main()
```

11.3. Обсуждение

Вам уже знакома функция `get_args()`. Вы изучили несколько способов определения опционального целочисленного параметра с дефолтным значением и научились применять функцию `parser.error()` для завершения работы программы, если пользователь передал недопустимый аргумент. Благодаря модулю `argparse`, избавляющему вас от решения рутинных задач, вы экономите массу времени, а также получаете уверенность, что в программе используются корректные данные. Теперь изучим новый функционал!

11.3.1. Обратный отсчет

Вы узнали, как вести обратный отсчет от заданного значения `—num` и как использовать цикл `for` для итерирования:

```
>>> for n in range(3, 0, -1):
...     print(f' {n} bottles of beer')
...
3 bottles of beer
2 bottles of beer
1 bottles of beer
```

Вместо создания каждого куплета с помощью цикла `for` я предлагаю использовать функцию `verse()` для генерации куплетов в заданном диапазоне чисел. В предыдущих проектах мы помещали весь функционал программы в функцию `main()`. Однако со временем ваши программы станут длиннее и будут содержать сотни и даже тысячи строк кода (LOC). Такие объемные программы и функции очень сложно тестировать и поддерживать, поэтому следует разбивать идеи на небольшие функциональные блоки, которые легко разобрать и протестировать. В идеале каждая функция должна делать что-то *одно*. Если вы умеете понимать и проверять свои маленькие простые *функции*, то с уверенностью можете составлять из них длинные и сложные *программы*.

11.3.2. Разработка через тестирование

Предлагаю вам добавить в свою программу функцию `test_verse()`, позволяющую с модулем `pytest` создать рабочую функцию `verse()`. Данная идея следует принципам, описанным Кентом Беком в книге «*Экстремальное программирование: разработка через тестирование*» (Питер, 2022).

1. Напишите тест для нереализованного функционального модуля.
2. Запустите все ранее написанные тесты и убедитесь, что созданный тест не пройден.
3. Напишите код, реализующий новую функциональность.
4. Запустите все тесты и убедитесь, что они успешно пройдены.
5. Выполните рефакторинг кода (перепишите его, чтобы улучшить читабельность или структуру).
6. Начните заново (повторите).

Например, предположим, что нам нужна функция, прибавляющая 1 к любому заданному числу. Мы присвоим ей имя `add1()` и определим в теле функции ключевое слово `pass`, чтобы интерпретатор Python воспринимал ее как «функцию-заглушку»:

```
def add1(n):
    pass
```

Далее пишем функцию `test_add1()`, которой передаются определенные аргументы и используются операторы `assert` для проверки возврата ожидаемого значения:

```
def test_add1():
    assert add1(0) = 1
    assert add1(1) = 2
    assert add1(-1) = 0
```

Запускаем `pytest` (или любую другую среду тестирования на ваш вкус) и проверяем, что функция *не работает* (разумеется это так, потому что в ней выполняется только оператор `pass`). Затем пишем *рабочий* код функции (`return n + 1` вместо оператора `pass`) и передаем всевозможные аргументы, в том числе `None`, ноль, единица и т. п.*.

* Однажды в ходе индивидуальной консультации профессор информатики велел мне обработать случаи со значениями 0, 1 и n (бесконечность), и это навечно врезалось мне в память.

11.3.3. Функция verse()

Я написал функцию `test_verse()`, которая точно отображает, что ожидается от первого и второго аргументов. В разработке через тестирование мне очень нравится возможность продумать, как следует использовать код, что передать в качестве аргументов и что получить в результате. Например, что *должна* вернуть функция `add1()` в случае запуска, если она:

- без аргументов;
- с более чем одним аргументом;
- со значением `None`;
- с любым нечисловым (`int`, `float` и `complex`) значением, например строкой или словарем.

Можно написать тесты, позволяющие передать как допустимые, так и недопустимые значения, и решить, как должна вести себя программа во всех этих ситуациях.

Ниже показана написанная мной функция `verse()`, которая проходит тестирование с помощью функции `test_verse()`:

```
def verse(bottle):
    """Поем куплет"""

    next_bottle = bottle - 1
    s1 = '' if bottle == 1 else 's'
    s2 = '' if next_bottle == 1 else 's'
    num_next = 'No more' if next_bottle == 0 else next_bottle
    return '\n'.join([
        f' {bottle} bottle{s1} of beer on the wall,',
        f' {bottle} bottle{s1} of beer,',
        f'Take one down, pass it around,',
        f' {num_next} bottle{s2} of beer on the wall!',
])
```

Данный код вы видели в разделе 11.2. По факту я изолирую фрагменты возвращаемой строки, которые будут изменены, и создаю переменные для подстановки значений в этих позициях. Я использую значения переменных `bottle` и `next_bottle` и определяю, следует ли указывать букву «s» для множественного числа после слова bottle. Мне

также нужно выяснить, следует ли выводить следующее количество бутылок в виде числа или необходимо вывести строку "No more" (если переменной `next_bottle` присвоено значение `0`). Так как для присвоения значений переменным `s1`, `s2` и `num_next` выполняются *двоичные* вычисления, то есть осуществляется выбор между *двумя* значениями, на мой взгляд, лучше всего применить конструкцию `if`.

Данная функция успешно проходит тест `test_verse()`, поэтому я могу использовать ее для конструирования песни.

11.3.4. Перебор куплетов

Вы можете применить цикл `for` для обратного отсчета и функцию `print()` для вывода каждого куплета:

```
>>> for n in range(3, 0, -1):
...     print(verse(n))
...
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!
2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!
1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

Результат *почти* достигнут, но нужно использовать два символа перевода строки между куплетами. Как вариант, можно прибегнуть к опции `end` функции `print()`, чтобы добавить два символа перевода строки для всех количеств бутылок больше 1:

```
>>> for n in range(3, 0, -1):
...     print(verse(n), end='\n' * (2 if n > 1 else 1))
...
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
```

```
2 bottles of beer on the wall!

2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

Я же предпочитаю метод `str.join()` для добавления двух символов перевода строки между элементами в списке. В данном примере элементами являются куплеты, и я могу превратить цикл `for` в представление списка, как показано на рис. 11.5.

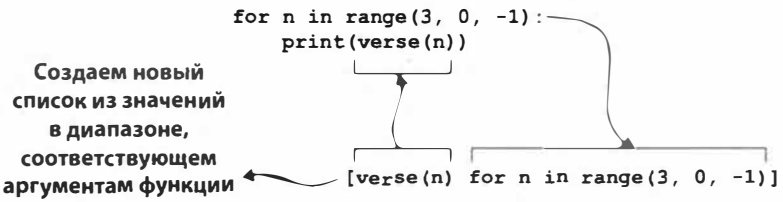


Рис. 11.5. Цикл `for` в сравнении с представлением списка

```
>>> verses = [verse(n) for n in range(3, 0, -1)]
>>> print('\n\n'.join(verses))
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!

2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

Прекрасное решение, но я хотел бы выделить следующую закономерность: функция применяется к каждому элементу последовательности, в чем заключается особенность `map()`. Как показано на рис. 11.6, наше представление списка можно лаконично переписать с помощью функции `map()`.

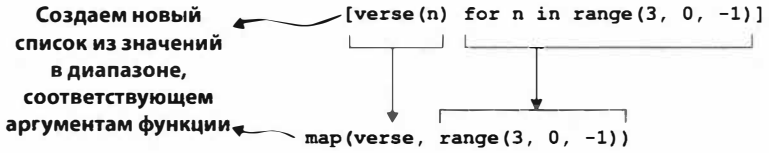


Рис. 11.6. Представление списка можно заменить функцией `map()`. В обоих случаях возвращается новый список

В нашем проекте последовательность представляет собой диапазон чисел в обратном порядке, и нам нужно применить функцию `verse()` к каждому значению и собрать полученные куплеты. Напоминает концепцию покрасочной мастерской из главы 8, где функция «окрашивает» автомобили в «синий цвет», добавляя слово «синий» в начало строки. Если требуется применить функцию к каждому элементу в последовательности, следует задуматься о рефакторинге кода с помощью функции `map()`:

```
>>> verses = map(verse, range(3, 0, -1))
>>> print('\n\n'.join(verses))
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!

2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

Если мне нужно преобразовать некоторую последовательность элементов, используя какую-либо функцию, я предпочитаю сначала

подумать о том, как буду обрабатывать *один* из элементов. На мой взгляд, гораздо проще написать и протестировать одну функцию с одним фрагментом входных данных, чем сразу огромный поток операций. Многие программисты на Python предпочитают использовать представления списков, а я остановил свой выбор на функции `map()`, благодаря которой код получается лаконичнее. Выполнив поиск по запросу «представление списка или map» в интернете, вы узнаете, что, по мнению ряда людей, у представлений списков понятнее код, чем у `map()`, а `map()` работает чуточку быстрее. Я не доказываю, что один подход лучше другого. Это дело вкуса или, по крайней мере, решение по результатам обсуждения с коллегами.

Если вы выбрали функцию `map()`, помните, что она требует *функцию* в качестве первого аргумента, а затем последовательность элементов, которые будут аргументами функции. Функция `verse()` (которую вы тестировали!) служит первым аргументом, а функция `range()` предоставляет список. Функция `map()` передает каждый элемент из списка `range()` в качестве аргумента функции `verse()`, как показано на рис. 11.7. В результате получается новый список с результатами всех этих вызовов. Многие циклы `for` лучше записывать в виде сопоставлений функции со списком аргументов!

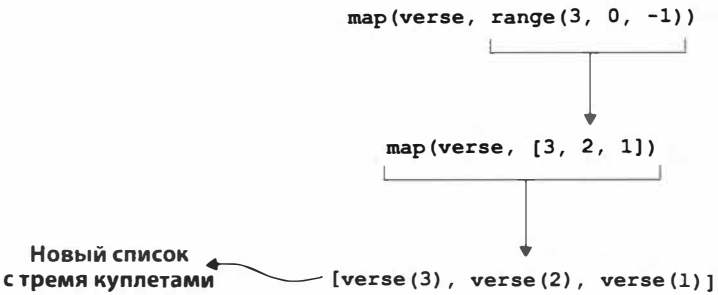


Рис. 11.7. Функция `map()` вызывает функцию `verse()` на каждом элементе, созданном функцией `range()`. Функция работает до последнего значения

11.3.5. 1500 других способов

Существуют буквально сотни способов решить описанную в этой главе задачу. На ресурсе **www.99-bottles-of-beer.net** представлено 1500 вариантов работы с нашей песней на разных языках. Сравните свое решение с размещенными там. Какой бы тривиальной ни была программа,

с помощью нее вы исследовали некоторые действительно интересные идеи Python.

11.4. Прокачиваем навыки

— Замените арабские цифры (1, 2, 3) текстом (один, два, три).
— Добавьте опцию `--step` (положительное целое число, дефолтное значение `1`), позволяющую пользователю шагать через числа, например кратно двум или пяти.
— Добавьте флаг `--reverse`, чтобы изменить порядок следования куплетов на прямой вместо обратного.

РЕЗЮМЕ

— Метод разработки через тестирование (TDD, Test-driven development) занимает важное место в написании надежного воспроизводимого кода. Благодаря тестам у вас развязаны руки в плане рефакторинга кода (реорганизации и совершенствования для улучшения производительности и/или читабельности) и вы можете быть уверены, что новая версия программы работает должным образом. Никогда не забывайте про тесты!
— Функция `range()` выполняет счет в обратном порядке, если поменять местами параметры `start` и `stop` и ввести опциональный третий параметр `step`, присвоив ему значение `-1`.
— Цикл `for` в большинстве случаев можно заменить представлением списка или функцией `map()` для более короткого и лаконичного кода.

Глава 12
Вымогатель: произвольная капитализация букв

Разработка программ так действует мне на нервы, что я пустился во все тяжкие! Я только что похитил соседскую кошку и хочу написать владельцам записку с требованием выкупа. В старые добрые времена я бы вырезал буквы из журналов и наклеил их на лист бумаги, чтобы изложить свои требования. Может быть, и сейчас так поступить? Но как же это утомительно! Нет уж, лучше я напишу Python-программу *ransom.py*, которая будет произвольно капитализировать буквы в тексте:

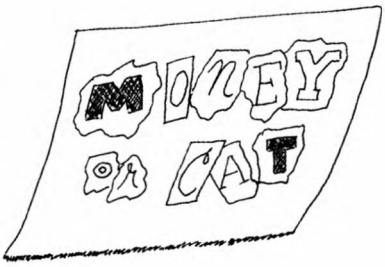
```
$ ./ransom.py 'give us 2 million dollars or the
cat gets it!'
gIVe US 2 milLION DoLlArs or ThE cAt GEts It!
```

Как видно из листинга, моя недобрая программа принимает на входе жуткий текст в качестве позиционного аргумента. Так как она работает с модулем `random`, я хочу использовать опцию `-s` или `--seed`, чтобы можно было реплицировать столь ужасный вывод:

```
$ ./ransom.py --seed 3 'give us 2 million dollars or the cat gets
it!'
giVE uS 2 MILlioN dolLaRS OR tHe cAt GETS It!
```

Зловредный позиционный аргумент может ссылаться на файл. В таком случае следует считывать его жестокое содержимое:

```
$ ./ransom.py --seed 2 ../inputs/fox.txt
the qUIck BROWN fOX JUmps ovEr ThE LAZY DOg.
```



Если моя противозаконная программа запускается без аргументов, она выводит краткие справочные сведения:

```
$ ./ransom.py
usage: ransom.py [-h] [-s int] text
ransom.py: error: the following arguments are required: text
```

Если же она запускается с флагом `-h` или `--help`, выводятся более длинные инструкции для злоумышленников:

```
$ ./ransom.py -h
usage: ransom.py [-h] [-s int] text

Ransom Note

positional arguments:
  text                  Input text or file

optional arguments:
-h, --help            show this help message and exit
-s int, --seed int    Random seed (default: None)
```

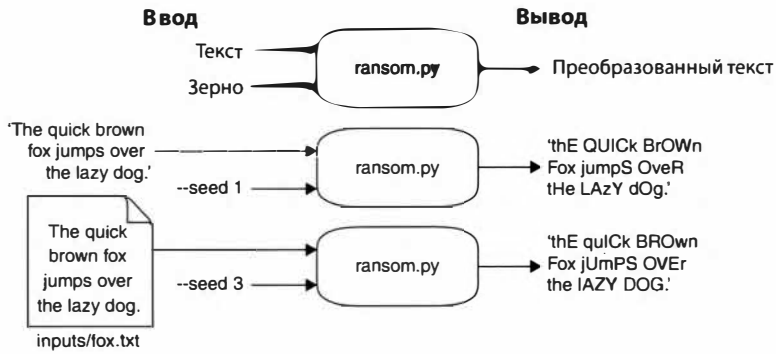


Рис. 12.1. Ужасная программа преобразует введенный текст в записку с требованием выкупа, произвольно капитализируя (превращая в прописные) буквы

На рис. 12.1 показана диаграмма программы с наглядным отображением входных и выходных данных.

Прочитав данную главу, вы научитесь:

— использовать модуль `random` как виртуальную монетку, которую можно «подбросить», чтобы выбрать один из двух вариантов;
— применять способы генерации новых строк из существующих, включая случайные вычисления;
— различать циклы `for`, представления списков и функцию `map()`.

12.1. Создание файла ransom.py

По традиции начнем с файла *new.py* или скопируем содержимое документа *template/template.py* для создания файла *ransom.py* в каталоге *12_ransom*. Данная программа, как и ряд предыдущих, принимает обязательный позиционный аргумент с текстом и опциональное целое число (по умолчанию `None`) в качестве зерна. Как и в прошлых упражнениях, вместо строки текста аргумент может указывать на имя файла, содержимое которого интерпретирует программа.

Сначала напишите следующий код функции `main()`:

```
def main():
    args = get_args()
    random.seed(args.seed)
    print(args.text)
```

Получаем обработанные аргументы командной строки.

Передаем методу `random.seed()` пользовательское значение. По умолчанию используется значение `None`, что равнозначно его отсутствию.

Начинаем с вывода пользовательских данных.

После запуска программы вы увидите в выводе входные данные из командной строки:

```
$ ./ransom.py 'your money or your life!'
your money or your life!
```

Или содержимое входного файла:

```
$ ./ransom.py ../inputs/fox.txt
The quick brown fox jumps over the lazy dog.
```

При написании программы важно продвигаться вперед понемногу. Рекомендуется запускать программу *после каждого внесенного изменения*, проверяя ее работу вручную и с помощью тестов.

После решения вопроса с получением входных данных пришло время продумать, как обрабатывать наше коварное сообщение.

12.1.1. Изменение текста

Вы уже знаете, что нельзя напрямую изменить строковое значение:

```
>>> text = 'your money or your life!'
>>> text[0] = 'Y'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

И как же тогда изменить регистр некоторых случайных букв?

Начните с поиска способа изменить *одну* букву, а не сразу несколько. Подумайте, как изменить букву, выбранную случайным образом, и вернуть ее в прописном или строчном регистре? Давайте создадим фиктивную функцию `choose()`, принимающую один символ и возвращающую его без изменений:

```
def choose(char):
    return char
```

Протестируем ее:

```
def test_choose():
    state = random.getstate()
    random.seed(1)
    assert choose('a') == 'a'
    assert choose('b') == 'b'
    assert choose('c') == 'C'
    assert choose('d') == 'd'
    random.setstate(state)
```

Состояние модуля `random` глобально влияет на всю программу. Любое внесенное здесь изменение способно отразиться на другом коде программы, поэтому мы сохраняем текущее состояние.

Присваиваем случайному зерну некое известное значение. Данное изменение имеет глобальное влияние на программу. Оно отражается на всех прочих вызовах функций из модуля `random`!

Функция `choose()` получает набор букв, и мы используем оператор `assert`, чтобы проверить, является ли возвращаемое функцией значение ожидаемой буквой.

Сбрасываем глобальное состояние на исходное.

Рандомные зерна

Вам интересно, как я узнал, каков будет результат выполнения функции `choose()` при указанном случайном значении зерна? Признаюсь, я создал функцию, затем указал зерно, запустил ее с заданными входными данными и записал результат в качестве показанных здесь утверждений. Вы должны получить такой же результат. Если это не так, вероятно, я допустил ошибку.

12.1.2. Подбрасывание монетки

Итак, нам нужно, чтобы программа рандомно выводила букву в прописном или строчном регистре. То есть она должна осуществлять так называемый *бинарный* выбор, когда допускается один из двух вариантов — по аналогии с подбрасыванием монетки. Орел или решка? Или, если говорить на языке программы, 0 или 1:

```
>>> import random
>>> random.choice([0, 1])
1
```

Также допустим вариант «Правда» или «Ложь»:

```
>>> random.choice([False, True])
True
```

Можно использовать конструкцию `if`, с помощью которой программа возвращает прописную букву в случае выбора значения 0 или `False` или строчную букву в противном случае. Весь код функции `choose()` умещается в одну строку.

12.1.3. Создание новой строки

Теперь необходимо применить функцию `choose()` к каждому символу в переданной пользователем строке. Надеюсь, данный прием вам уже знаком. Попробуйте начать с первого способа из главы 8, в котором мы использовали цикл `for` для перебора каждого символа исходного текста и замены всех гласных одной определенной буквой. В нашей программе можно также перебирать символы текста и передавать их в виде аргумента функции `choose()`. В результате мы получаем новый

список (или строку) преобразованных символов. После создания версии программы с циклом `for`, успешно проходящей тест, попробуйте переписать код с использованием представления списка, а затем функции `map()`.

Ваш ход! Разработайте программу, успешно проходящую тесты.

12.2. Решение

Мы рассмотрим несколько вариантов обработки всех символов во входном тексте. Начнем с цикла `for`, создающего новый список, и впоследствии убедимся, что представление списка — лучший способ справиться с задачей. Наконец, я покажу вам, как использовать функцию `map()` для крайне лаконичного (думаю, даже элегантного) решения.

```
#!/usr/bin/env python3
"""Программа-вымогатель"""

import argparse
import os
import random

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Ransom Note',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='text',
                        help='Input text or file')

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    args = parser.parse_args()
```

Текстовый аргумент представляет собой позиционное строковое значение.

Опция `--seed` представлена целым числом, с дефолтным значением `None`.

Обрабатываем аргументы командной строки в переменную `args`.

```
    if os.path.isfile(args.text):
        args.text = open(args.text).read().rstrip()

    return args

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    text = args.text

    random.seed(args.seed)
    ransom = []
    for char in args.text:
        ransom.append(choose(char))

    print(''.join(ransom))

# --------------------------------------------------
def choose(char):
    """Выводим случайным образом выбранную прописную
       или строчную букву"""

    return char.upper() if random.choice([0, 1]) else char.lower()

# --------------------------------------------------
def test_choose():
    """Test choose"""

    state = random.getstate()
    random.seed(1)
```

Возвращаем аргументы вызывающей стороне.

Если значение `args.text` ссылается на файл, берем его содержимое и присваиваем `args.text`.

Передаем методу `random.seed()` значение `args.seed`. По умолчанию используется значение `None`, что равнозначно его отсутствию. Это означает, что программа будет выдавать рандомный результат, когда зерно не задано, однако ее можно протестировать, если передать определенное значение зерна.

Создаем пустой список для хранения нового сообщения о выкупе.

Используем цикл `for` для перебора каждого символа в `args.text`.

Добавляем выбранную букву в список `ransom`.

Присоединяем список с текстом выкупа к пустой строке, формируя новую строку для вывода.

Используем метод `random.choice()` для выбора значения `0` или `1`, что с точки зрения логики конструкции `if` оценивается как `False` или `True`, соответственно.

Определяем функцию, которая случайным образом возвращает прописную или строчную версию данной буквы.

Сохраняем текущее состояние модуля `random`.

Определяем функцию `test_select()`, запускаемую с помощью модуля `pytest`. Функция не принимает аргументов.

Передаем методу `random.seed()` известное значение в целях тестирования.

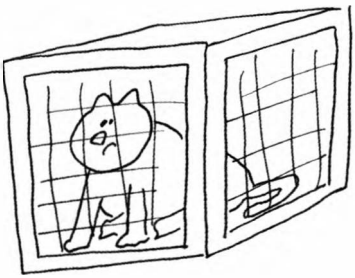
```
    assert choose('a') == 'a'
    assert choose('b') == 'b'
    assert choose('c') == 'C'
    assert choose('d') == 'd'
    random.setstate(state)

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Используем оператор `assert`, проверяя, что функция `choose()` с известным аргументом выдает ожидаемый результат.

Сбрасываем состояние модуля `random`, чтобы изменения не повлияли на остальной код программы.

12.3. Обсуждение

Мне очень нравится наша задача, потому что существует множество интересных вариантов ее решения. Знаю, знаю, программисты Python любят находить «один очевидный способ», но давайте исследуем разные методы, как вы на это смотрите? В коде функции `get_args()` нет ничего нового, да и разбирали мы ее уже неоднократно, поэтому предлагаю на этот раз ее пропустить.



12.3.1. Перебор элементов в последовательности

Предположим, что у нас есть следующее сообщение с требованием выкупа:

```
>>> text = '2 million dollars or the cat sleeps with the fishes!'
```

Я хочу случайным образом изменить регистр букв. Как упоминалось ранее, мы можем использовать цикл `for` для перебора каждого символа.

Один из способов вывести прописную версию текста — вывести в прописном регистре *каждую букву*:

```
for char in text:
    print(char.upper(), end='')
```

Таким образом мы получаем текст "2 MILLION DOLLARS OR THE CAT SLEEPS WITH THE FISHES!".

Вместо выполнения лишь функции `char.upper()` я могу случайным образом осуществлять вызовы функций `char.upper()` и `char.lower()`. Для этого пригодится функция `random.choice()`, позволяющая выбрать одно из двух значений, таких как `True` и `False` или `0` и `1`:

```
>>> import random
>>> random.choice([True, False])
False
>>> random.choice([0, 1])
0
>>> random.choice(['blue', 'green'])
'blue'
```

Следуя первому способу из главы 8, я создал новый список для хранения сообщения о выкупе и добавил фактор случайности следующим образом:

```
ransom = []
for char in text:
    if random.choice([False, True]):
        ransom.append(char.upper())
    else:
        ransom.append(char.lower())
```

Затем присоединил полученные символы к пустой строке, чтобы вывести новую:

```
print(''.join(ransom))
```

Решить задачу проще можно с помощью конструкции `if`, позволяющей выбрать, какие символы следует брать, прописные или строчные (рис. 12.2):

```
ransom = []
for char in text:
    ransom.append(char.upper() if random.choice([False, True])
else char.lower())
```

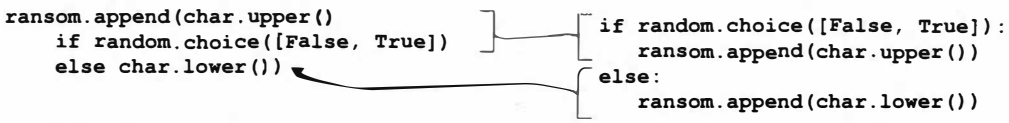


Рис. 12.2. Бинарная конструкция `if/else` выглядит лаконичнее с выражением `if`

Вам необязательно использовать логические значения `False` и `True`. Вполне подойдут значения `0` и `1`:

```
ransom = []
for char in text:
    ransom.append(char.upper() if random.choice([0, 1])
                  else char.lower())
```

Когда числа оцениваются *в логическом контексте* (то есть интерпретатор Python обрабатывает их как логические значения), `0` считается `False`, а все прочие числа — `True`.

12.3.2. Функция для выбора буквы

Конструкцию `if` можно расположить в функции, но, как по мне, трудно воспринимать ее внутри `ransom.append()`.

Поэтому, поместив конструкцию в функцию, я могу присвоить ей описательное имя и написать соответствующий тест:

```
def choose(char):
    """Выводим случайным образом выбранную прописную или строчную
       букву"""

    return char.upper() if random.choice([0, 1]) else char.lower()
```

Теперь выполним функцию `test_choose()`, чтобы проверить, предсказуемо ли результирует моя функция.

Показанный ниже код читать намного проще:

```
ransom = []
for char in text:
    ransom.append(choose(char))
```

12.3.3. Другой вариант метода list.append()

Решение из раздела 12.2 допускает создание пустого списка, к которому применяется метод `list.append()` с результатом вызова функции `choose()`. Другой вариант использования метода `list.append()` заключается в применении оператора += для добавления значения справа (добавляемого элемента) к левой части (списку) выражения, как показано на рис. 12.3.

```
def main():
    args = get_args()
    random.seed(args.seed)

    ransom = []
    for char in args.text:
        ransom += choose(char)

    print(''.join(ransom))
```

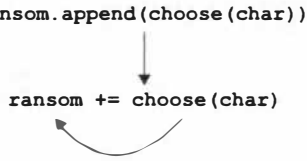


Рис. 12.3. С помощью оператора += можно написать другой вариант реализации метода `list.append()`

Такой же синтаксис используется в случаях конкатенации символа со строкой и сложения чисел.

12.3.4. Использование строк вместо списка

В двух предыдущих решениях списки соединяются с пустой строкой, формируя новую строку для вывода. Вместо этого мы можем начать с пустой строки и наращивать ее посимвольно с помощью оператора +=:

```
def main():
    args = get_args()
```

```
    random.seed(args.seed)

    ransom = ''
    for char in args.text:
        ransom += choose(char)

    print(ransom)
```

Как я упомянул, оператор += является еще одним способом добавления элемента в список. Хорошо это или плохо, но Python часто интерпретирует строки и списки взаимозаменяемо, зачастую неявно.

12.3.5. Представление списка

Во всех предыдущих вариантах пустая строка или список сначала инициализируется, а затем конструируется с помощью цикла `for`. Я считаю, что практически всегда лучше решать подобные задачи с помощью представления списка, поскольку все его предназначение заключается в возвращении нового списка. Мы способны сократить три строки кода до одной:

```
def main():
    args = get_args()
    random.seed(args.seed)
    ransom = [choose(char) for char in args.text]
    print(''.join(ransom))
```

А можно вообще не создавать переменную `ransom`. Как правило, я присваиваю значение переменной только в том случае, если использую ее не менее одного раза и/или в целях улучшения читабельности кода:

```
def main():
    args = get_args()
    random.seed(args.seed)
    print(''.join([choose(char) for char in args.text]))
```

Цикл `for` итерирует последовательности каких-либо значений, попутно выполняя *побочные действия*, например вывод значения или обработку строки в файле. Если вы преследуете цель создать новый список, думаю, лучше всего подойдет представление списка. Любой код, помещаемый в тело цикла `for` для обработки элемента, рекомендуется размещать в функции с тестом.

12.3.6. Функция map()

Я уже упоминал, что функция `map()` похожа на представление списка, но обычно состоит из меньшего количества строк кода. Оба подхода позволяют создать новый список из некоего итерируемого объекта, как показано на рис. 12.4. В данном случае результирующий список создается функцией `map()` путем обработки методом `choose()` каждого символа в `args.text`:

```
def main():
    args = get_args()
    random.seed(args.seed)
    ransom = map(choose, args.text)
    print(''.join(ransom))
```

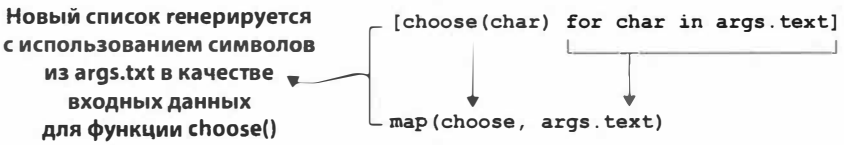


Рис. 12.4. С помощью функции `map()` можно лаконичнее выразить концепции представления списка

И опять же, можно не прибегать к переменной `ransom` и непосредственно применять список, возвращаемый функцией `map()`:

```
def main():
    args = get_args()
    random.seed(args.seed)
    print(''.join(map(choose, args.text)))
```

12.4. Сравнение способов

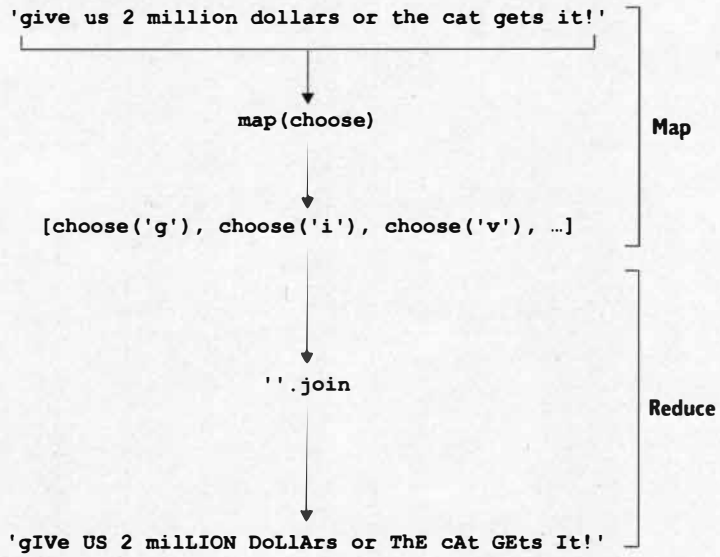
Вас может раздражать трата времени на поиск множества вариантов решения тривиальных задач, но напомню, что одна из целей данной книги — исследовать различные концепции, доступные в Python. Первый рассмотренный нами в разделе 12.2 способ весьма важен. Его, вероятно, взял бы на вооружение программист на C или Java. Версия с представлением списка крайне идиоматична для Python — это чисто «питонский» способ, как выразились бы поклонники данного языка. Решение с функцией `map()` может показаться привлекательным фанатам чисто функциональных языков типа Haskell.

Все рассмотренные нами подходы позволяют достичь одной и той же цели, но воплощают в себе разную эстетику и парадигмы программирования. Я предпочитаю решение с функцией `map()`, но это мой личный выбор. Вам следует придерживаться своего подхода.

Технология MapReduce

В 2004 году сотрудники корпорации Google опубликовали статью об алгоритме MapReduce. На стадии «Map» происходит предварительная обработка входных данных — всех элементов в коллекции, например страниц в интернете, которые необходимо проиндексировать для поиска. Данные операции могут выполняться параллельно, то есть с использованием множества компьютеров для обработки страниц отдельно друг от друга и в любом порядке. Затем на стадии «Reduce» происходит свертка предварительно обработанных данных, чтобы их можно было поместить в единую базу данных.

В нашей программе *ransom.py* на стадии «Map» осуществлялся в случайном порядке выбор регистра обрабатываемой буквы, а на стадии «Reduce» все фрагменты данных объединялись в новую строку. Предполагаю, функция `map()` может использовать ресурсы нескольких процессоров для *параллельного* выполнения функций, а не *последовательного* (как в случае с циклом `for`), что, вероятно, сократит время получения результатов.



Концепции технологии MapReduce «ложатся» на многие ситуации — от индексации веб-страниц до случая с нашей программой-вымогателем.

> Изучение MapReduce, на мой взгляд, похоже на прочтение статьи о новой птице. Раньше я ее даже не замечал, но как только узнал о ней побольше, стал видеть ее повсюду. Так и здесь. Стоит понять закономерность, вы разглядите ее повсюду.

12.5. Прокачиваем навыки

Напишите версию программы *ransom.py*, представляющую буквы другими способами, например, комбинируя символы ASCII, как показано ниже. Вы можете придумать свой собственный вариант. Главное, не забывайте вносить изменения в тесты.

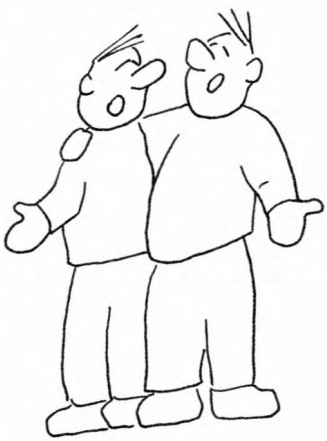
```
A  4      K  |<
B  |3     L  |_
C  (      M  |\/|
D  |)     N  |\|
E  3      P  |`
F  |=     S  5
G  (-     T  +
H  |-|    V  \/
I  1      W  \/\/
J  _|
```

РЕЗЮМЕ

- При обработке множества элементов сначала продумайте, как обработать один из них.
- Напишите тест, чтобы сконструировать функцию для обработки одного элемента. Что вы передадите функции и каков ожидаемый результат?
- Создайте функцию, успешно проходящую тест. Обязательно продумайте поведение кода при передаче допустимых и недопустимых входных данных.
- Чтобы применить функцию к каждому элементу из числа входных данных, используйте цикл `for`, представление списка или функцию `map()`.

Глава 13

Двенадцать дней Рождества: разработка алгоритмов



«Двенадцать дней Рождества» — для меня это одна из худших песен всех времен и народов. Когда я ее слышу, она гарантированно портит мне рождественское настроение. Хочется крикнуть: «Это когда-нибудь прекратится?!» Как бы то ни было, довольно интересно написать алгоритм для генерации текста этой песни до заданного пользователем дня, поскольку здесь нужно вести отсчет как *по* мере добавления каждого куплета (до указанного дня), так и *внутри* куплетов (учитывая подарки из предыдущих дней). Для решения данной задачи вам помогут знания, полученные при написании программы «99 бутылок пива».

Итак, нам нужна программа с названием *twelve_days.py*, генерирующая текст песни «Двенадцать дней Рождества» до заданного дня, указанного с помощью аргумента `-n` или `--num` (по умолчанию равного `12`). Обратите внимание, что между куплетами используются два символа перевода строки, а в конце лишь один:

```
$ ./twelve_days.py -n 3
On the first day of Christmas,
```

```
My true love gave to me,
A partridge in a pear tree.

On the second day of Christmas,
My true love gave to me,
Two turtle doves,
And a partridge in a pear tree.

On the third day of Christmas,
My true love gave to me,
Three French hens,
Two turtle doves,
And a partridge in a pear tree.
```

Текст по умолчанию выводится в поток `STDOUT`, а если указан аргумент `-o` или `--outfile`, записывается в файл с указанным именем. Обратите внимание, что песня целиком состоит из 113 строк:

```
$ ./twelve_days.py -o song.txt
$ wc -l song.txt
     113 song.txt
```

Освоив данную главу, вы научитесь:

- создавать алгоритм генерации песни «Двенадцать дней Рождества» до заданного дня в диапазоне от 1 до 12;
- обращать списки;
- работать с функцией `range()`;
- выводить текст в файл или поток `STDOUT`.

13.1. Создание файла twelve_days.py

Традиционно я предлагаю вам создать программу, запустив файл *new.py* или скопировав содержимое документа *template/template.py*. Присвойте программе имя *twelve_days.py* и разместите ее в каталоге *13_twelve_days*.

Ваша программа должна принимать два опциональных аргумента:

- целочисленный `-n` или `--num` со значением по умолчанию `12`;
- `--o` или `-- outfile` с именем файла для записи вывода.

За помощью с разработкой второй опции вы можете обратиться к главе 5, в которой мы реализовывали подобное в программе-кричалке. Та программа писала вывод в файл с указанным именем или в поток `sys.stdout`, если имя файла не указано. В нашей рождественской программе я предлагаю объявить параметр `--outfile` с кодом `type=argparse.FileType('wt')`, указав тем самым, что модулю `argparse` требуется аргумент с именем *текстового файла, доступного для чтения*. Когда пользователь указывает допустимый аргумент, `args.outfile` становится *открытым и доступным для записи файловым дескриптором*. Если вы также используете дефолтное значение `sys.stdout`, вы быстро разберетесь с вариантами записи в текстовый файл и в поток `STDOUT`!

Единственный недостаток такого подхода заключается в том, что справочная информация программы в части дефолтного значения параметра `--outfile` выглядит слегка чудно:

```
$ ./twelve_days.py -h
usage: twelve_days.py [-h] [-n days] [-o FILE]

Twelve Days of Christmas

optional arguments:
-h, --help            show this help message and exit
-n days, --num days   Number of days to sing (default: 12)
-o FILE, --outfile FILE
                      Outfile (default: <_io.TextIOWrapper
name='<stdout>'
                      mode='w' encoding='utf-8'>)
```

На данном этапе ваша программа должна успешно проходить первые два теста.

На рис. 13.1 показана диаграмма, которая поможет вам настроиться на написание оставшейся части кода.

Программа должна выводить ошибку, если значение `--num` не является целым числом в диапазоне от 1 до 12. Я предлагаю проверить данное обстоятельство с помощью функции `get_args()` и использовать метод `parser.error()` для прекращения работы программы с выводом сообщения об ошибке и справочной информации:

```
$ ./twelve_days.py -n 21
usage: twelve_days.py [-h] [-n days] [-o FILE]
twelve_days.py: error: --num "21" must be between 1 and 12
```

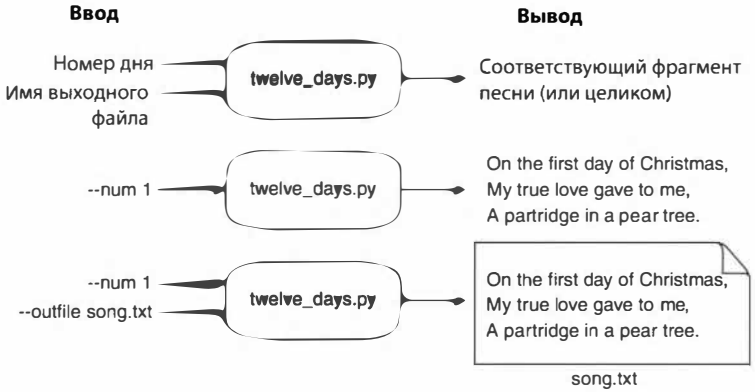


Рис. 13.1. Программа *twelve_days.py* принимает опции с днем окончания и именем выходного файла

Разобравшись с ошибкой, связанной с недопустимым значением `--num`, вы должны пройти первые три теста.

13.1.1. Отсчет

В песне «99 бутылок пива» необходимо было считать от заданного числа — количества бутылок. Здесь нужно сосчитать до значения `--num`, а затем обратно отсчитать подарки. Вы сможете решить данную задачу с помощью функции `range()`, но не забывайте, что начинать нужно с 1, поскольку бессмысленно петь «В нулевой день Рождества». Обратите внимание, что верхняя граница не включена:

```
>>> num = 3
>>> list(range(1, num))
[1, 2]
```

Вам нужно добавить 1 к значению `--num`:

```
>>> list(range(1, num + 1))
[1, 2, 3]
```

Давайте начнем с вывода первой строки каждого куплета:

```
>>> for day in range(1, num + 1):
...     print(f'On the {day} day of Christmas,')
...
```

```
On the 1 day of Christmas,
On the 2 day of Christmas,
On the 3 day of Christmas,
```

Предлагаю вспомнить, как мы писали программу «99 бутылок пива». Для нее мы создали функцию `verse()`, генерирующую нужный *куплет*. Затем мы использовали метод `str.join()`, чтобы соединить куплеты двумя символами перевода строки. Давайте и здесь осуществим тот же подход. Я помещаю код цикла `for` в отдельную функцию:

```
def verse(day):
    """Создаем куплет"""
    return f'On the {day} day of Christmas,'
```

Обратите внимание, что функция не выводит строку, а возвращает куплет, так что мы можем протестировать ее:

```
>>> assert verse(1) == 'On the 1 day of Christmas,'
```

Посмотрим, как мы способны использовать функцию `verse()`:

```
>>> for day in range(1, num + 1):
...     print(verse(day))
...
On the 1 day of Christmas,
On the 2 day of Christmas,
On the 3 day of Christmas,
```

Пример простенькой функции `test_verse()`, с которой можно начать:

```
def test_verse():
    """ Test verse """
    assert verse(1) == 'On the 1 day of Christmas,'
    assert verse(2) == 'On the 2 day of Christmas,'
```

Получается не совсем верно, ведь должно быть сказано: "On the *first* day" или "On the *second* day", а не "1 day" и "2 day". Тем не менее надо же с чего-то начинать. Добавьте функции `verse()` и `test_verse()` в программу *twelve_days.py*, а затем выполните команду `pytest twelve_days.py`, чтобы убедиться, что все работает.

13.1.2. Числа прописью

Думаю, первое, что нужно сделать, это написать числа прописью, то есть вместо «1» — «первый», вместо «2» — «второй» и т. д. Можно использовать словарь, как в программе «Прыжок через пятерку», и связать каждое целочисленное значение 1–12 с его строковым значением. Итак, мы создаем словарь `ordinal`:

```
>>> ordinal = {} # что здесь происходит?
```

В данном случае можно поступить следующим образом:

```
>>> ordinal[1]
'first'
>>> ordinal[2]
'second'
```

Вы также вольны использовать список, если придумаете, как превратить каждый день в диапазоне в строки порядковых значений.

```
>>> ordinal = [] # что здесь происходит?
```

Код функции `verse()` выглядит примерно так:

```
def verse(day):
    """Создаем куплет"""
    ordinal = [] # something here!
    return f'On the {ordinal[day]} of Christmas,'
```

Обновим тесты в соответствии с изменениями:

```
def test_verse():
    """ Test verse """
    assert verse(1) == 'On the first day of Christmas,'
    assert verse(2) == 'On the second day of Christmas,'
```

Теперь программа воспроизводит нечто такое:

```
>>> for day in range(1, num + 1):
...     print(verse(day))
...
On the day first day of Christmas,
```

```
On the day second day of Christmas,
On the day third day of Christmas,
```

После добавления в *twelve_days.py* функции `test_verse()`, вы способны проверить работоспособность функции `verse()` с помощью команды `pytest twelve_days.py`. Модуль `pytest` запускает все функции, имена которых начинаются со слова `test_`.

Затенение

У вас может возникнуть соблазн присвоить переменной имя `ord`. Проблема в том, что Python, хотя и не запретит вам создать переменную с таким именем, имеет функцию `ord()`, возвращающую «для указанного Юникод-символа целое число, представляющее его позицию»:

```
>>> ord('a')
97
```

Интерпретатор Python не выдает ошибку при определении переменной или собственной функции с именем `ord`:

```
>>> ord = {}
```

поэтому вы можете использовать ее:

```
>>> ord[1]
'first'
```

Но тем самым вы перезаписываете встроенную функцию `ord()` и прерываете вызов функции:

```
>>> ord('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict' object is not callable
```

Возникает опасная ситуация, называемая затенением. Затенение происходит, когда переменная/функция во внутренней области видимости объявляется с тем же именем, что и переменная/функция во внешней области. В данном случае переменная/функция во внутренней области видимости «затеняет» (маскирует) переменную/функцию во внешней области видимости.

Выявлять подобные проблемы в программах помогают такие инструменты, как Pylint. Предположим, у вас есть следующий код:

```
$ cat shadow.py
#!/usr/bin/env python3

ord = {}
print(ord('a'))
```

По результатам проверки Pylint сообщит:

```
$ pylint shadow.py
************* Module shadow
shadow.py:3:0: W0622: Redefining built-in 'ord' (redefined-
builtin)
shadow.py:1:0: C0111: Missing module docstring (missing-
docstring)
shadow.py:4:6: E1102: ord is not callable (not-callable)

-------------------------------------
Your code has been rated at -25.00/10
```

Рекомендую активно пользоваться такими инструментами, как Pylint и Flake8, и перепроверять свой код.

13.1.3. Создание куплетов

Сформировав базовую структуру программы, сосредоточимся на выводе *корректного* текста. Давайте передадим функции `test_verse()` фактические значения для первых двух куплетов. Можно, конечно, добавить больше куплетов, но, думаю, разобравшись с первыми двумя, мы справимся и со всеми остальными:

```
def test_verse():
    """Test verse"""

    assert verse(1) == '\n'.join([
        'On the first day of Christmas,', 'My true love gave to me,',
        'A partridge in a pear tree.'
])
```

```
    assert verse(2) == '\n'.join([
        'On the second day of Christmas,', 'My true love gave to me,',
        'Two turtle doves,', 'And a partridge in a pear tree.'
])
```

Включив показанный выше код в свою программу *twelve_days.py*, вы сможете выполнить команду `pytest twelve_days.py` и увидеть, что функция `verse()` не работает:

```
========================== FAILURES ==========================
_____________________________ test_verse _____________________________

    def test_verse():
        """Test verse"""
>       assert verse(1) == '\n'.join([
            'On the first day of Christmas,', 'My true love gave
             to me,',
            'A partridge in a pear tree.'
])
E       AssertionError: assert 'On the first...of Christmas,'
        == 'On the first... a pear tree.'
E         - On the first day of Christmas,
E         + On the first day of Christmas,
E         ?                               +
E         + My true love gave to me,
E         + A partridge in a pear tree.

twelve_days.py:88: AssertionError
-------------------- 1 failed in 0.11 seconds --------------------
```

Символ > указывает на строку, в которой возникает исключение. Мы вызываем функцию `verse(1)` и запрашиваем, эквивалентен ли результат ожидаемому куплету.

Текст, генерируемый функцией `verse(1)`. Выводится только первая строка куплета.

Далее следуют ожидаемые строки.

Теперь необходимо добавить остальные строки в каждый куплет. Все они начинаются одинаково:

```
On the {ordinal[day]} day of Christmas,
My true love gave to me,
```

Затем с каждым новым днем добавляются подарки.

1. Куропатка на грушевом дереве.
2. Две горлицы.
3. Три курицы французские.

4. Четыре птицы говорящие.
5. Пять колец золотых.
6. Шесть гусынь, яйца несущих.
7. Семь плавающих лебедей.
8. Восемь молодых кормилиц.
9. Девять танцующих леди.
10. Десять прыгающих лордов.
11. Одиннадцать играющих трубачей.
12. Двенадцать барабанщиков*.

Обратите внимание, что для всех дней, кроме первого, последняя строка начинается с союза *and* — например, "*And a partridge...*" вместо "*A partridge...*".

Каждый куплет нужно отсчитывать в обратном направлении от указанного дня. К примеру, если выбран третий день, то в списке участвуют следующие подарки:

1. Три курицы французские.
2. Две горлицы.
3. Куропатка на грушевом дереве.

В главе 3 мы разбирались, как обратить список с помощью метода `list.reverse()` или с помощью функции `reversed()`. Мы пользовались теми же концепциями и в главе 11, когда брали бутылки с пивом со стены, так что показанный код наверняка вам знаком:

```
>>> day = 3
>>> for n in reversed(range(1, day + 1)):
...     print(n)
...
```

---

* В оригинале:

1. A partridge in a pear tree.
2. Two turtle doves.
3. Three French hens.
4. Four calling birds.
5. Five gold rings.
6. Six geese a laying.
7. Seven swans a swimming.
8. Eight maids a milking.
9. Nine ladies dancing.
10. Ten lords a leaping.
11. Eleven pipers piping.
12. Twelve drummers drumming. *Прим. ред.*

```
3
2
1
```

Сделайте так, чтобы функция возвращала первые две строки, а затем отсчитывала дни в обратном порядке:

```
>>> print(verse(3))
On the third day of Christmas,
My true love gave to me,
3
2
1
```

Теперь вместо чисел 3, 2 и 1 добавьте названия подарков:

```
>>> print(verse(3))
On the third day of Christmas,
My true love gave to me,
Three French hens,
Two turtle doves,
And a partridge in a pear tree.
```

Когда справитесь с данным заданием, проверьте программу с помощью функции `test_verse()`.

13.1.4. Использование функции verse()

Теперь пришло время задуматься об окончательной структуре кода с функцией `verse()`. Это может быть цикл `for`:

```
verses = []
for day in range(1, args.num + 1):
    verses.append(verse(day))
```

Или представление списка — лучший вариант для создания списка куплетов:

```
verses = [verse(day) for day in range(1,
          args.num + 1)]
```

А может быть и функция `map()`:

```
verses = map(verse, range(1, args.num + 1))
```

13.1.5. Вывод результата

Итак, с куплетами мы разобрались, пора использовать метод `str.join()` для вывода результата. По умолчанию он выводится в «стандартный поток» (`STDOUT`). Однако программа принимает необязательный аргумент `--outfile`, который может содержать имя файла для записи вывода. Вы вольны скопировать код из главы 5, но я рекомендую все же потратить время на изучение способа объявления выходного файла с помощью кода `type=argparse.FileType('wt')`. Можно даже установить поток `sys.stdout` по умолчанию, и вам не придется открывать файл для записи ручную!

13.1.6. Самостоятельная работа

Вовсе необязательно решать задачу так, как описываю я. «Правильным» можно считать любое самостоятельно написанное и понятое вами решение, проходящее набор тестов. Не переживайте, если вам пришлась по душе идея создания функции для `verse()` и предоставленного теста. Я искренне радуюсь, когда вы пробуете идти своим путем, главное, не забывайте о концепции разработки отдельных небольших функций *и тестов* к ним для решения задачи по частям, а затем объединяйте их для реализации более крупной цели.

Если на разработку и тестирование уходит больше времени, чем вы планировали — сутки или даже несколько дней, — не торопитесь и не волнуйтесь. Зачастую хорошая прогулка или сон творят чудеса, помогая прийти к решению задачи. Не пренебрегайте гамаком* или чашкой крепкого чая.

13.2. Решение

В рассматриваемой нами песне виновник торжества получает в подарок почти 200 птиц! Для начала я попрошу вас познакомиться с моим

* Прочитайте доклад «Разработка через гамак» Ричарда Хикки, создателя языка Clojure, **https://sqrtt.pro/hammock-driven-development-ru**.

решением задачи, основанным на функции `map()`. После чего мы обсудим версии с циклом `for` и представлением списка.

```
#!/usr/bin/env python3
"""Двенадцать дней Рождества"""

import argparse
import sys

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Twelve Days of Christmas',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-n',
                        '--num',
                        help='Number of days to sing',
                        metavar='days',
                        type=int,
                        default=12)

    parser.add_argument('-o',
                        '--outfile',
                        help='Outfile',
                        metavar='FILE',
                        type=argparse.FileType('wt'),
                        default=sys.stdout)

    args = parser.parse_args()

    if args.num not in range(1, 13):
        parser.error(f'--num "{args.num}" must be between 1
                     and 12')

    return args
```

Опция `--num` представляет собой целое число с дефолтным значением 12.

Опция `--outfile` имеет тип `type=argparse.FileType('wt')` и дефолтное значение `sys.stdout`. Если пользователь использует данную опцию, он должен передать имя файла, разрешенного к записи, чтобы модуль `argparse` смог открыть его и применить для записи.

Проверяем, что переданное значение `args.num` находится в допустимом диапазоне от 1 до 12 включительно.

Захватываем в переменную `args` результаты синтаксического анализа аргументов командной строки.

Если передано недопустимое значение `args.num`, используем метод `parser.error()`, выводящий краткую справочную информацию и сообщение об ошибке в поток `STDERR`, после чего программа завершает работу с кодом ошибки. Обратите внимание, что сообщение об ошибке содержит неверное значение пользователя и явно указывает, что правильное значение должно находиться в диапазоне от 1 до 12.

Получаем аргументы командной строки. Напоминаю, что проверка всех аргументов происходит в функции `get_args()`. Если вызов успешен, пользовательские аргументы допустимы.

Соединяем куплеты двумя символами перевода строки и выводим в файловый дескриптор `args.outfile` или `sys.stdout`.

Генерируем куплеты для указанного в `args.num` количества дней.

```
# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    verses = map(verse, range(1, args.num + 1))
    print('\n\n'.join(verses), file=args.outfile)


# --------------------------------------------------
def verse(day):
    """Создаем куплет"""
```

Определяем функцию для создания одного куплета из заданного пользователем количества.

Прописные значения чисел перечислены в списке строк.

```
    ordinal = [
        'first', 'second', 'third', 'fourth', 'fifth', 'sixth',
        'seventh','eighth', 'ninth', 'tenth', 'eleventh', 'twelfth'
]
```

Ежедневные подарки также перечислены в списке строк.

```
    gifts = [
        'A partridge in a pear tree.',
        'Two turtle doves,',
        'Three French hens,',
        'Four calling birds,',
        'Five gold rings,',
        'Six geese a laying,',
        'Seven swans a swimming,',
        'Eight maids a milking,',
        'Nine ladies dancing,',
        'Ten lords a leaping,',
        'Eleven pipers piping,',
        'Twelve drummers drumming,',
]
```

Строки каждого куплета начинаются одинаково, меняется лишь номер (прописью) заданного дня.

```
    lines = [
        f'On the {ordinal[day - 1]} day of Christmas,',
        'My true love gave to me,'
]
```

Используем метод `list.extend()`, чтобы добавить подарки, нарезанные из списка `gifts` и приведенные в обратном порядке.

```
    lines.extend(reversed(gifts[: day]))
```

Проверяем, чтобы номер был больше 1.

```
    if day > 1:
```

```
        lines[-1] = 'And ' + lines[-1].lower()

    return '\n'.join(lines)

# --------------------------------------------------
def test_verse():
    """Test verse"""

    assert verse(1) == '\n'.join([
        'On the first day of Christmas,', 'My true love gave to me,',
        'A partridge in a pear tree.'
])

    assert verse(2) == '\n'.join([
        'On the second day of Christmas,', 'My true love gave to me,',
        'Two turtle doves,', 'And a partridge in a pear tree.'
])

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Возвращаем строковые значения с символами перевода строки.

Модульный тест для функции `verse()`

13.3. Обсуждение

В функции `get_args()` мало что покажется вам незнакомым, поэтому бросим на нее лишь беглый взгляд. Опция `--num` представлена целым числом с дефолтным значением `12`. Мы используем метод `parser.error()`, чтобы завершить работу программы в случае передачи пользователем недопустимого значения. Опция `--outfile` немного отличается, так как мы объявляем ее с помощью кода `type=argparse.FileType('wt')`, указывая, что в качестве значения пользователь должен передать файл, доступный для записи. Соответственно, значение, получаемое от модуля `argparse`, будет открытым, доступным для записи файлом. Мы установили дефолтное значение объекта `sys.stdout`, который также является открытым и доступным для записи файлом. Таким образом мы обрабатываем две опции вывода с помощью модуля `argparse`, что экономит массу времени!

13.3.1. Создание одного куплета

Я решил прибегнуть к функции `verse()` для генерации одного куплета с учетом переданного пользователем номера дня:

```
def verse(day):
    """Создаем куплет"""
```

В списке перечислены номера дней прописью:

```
ordinal = [
    'first', 'second', 'third', 'fourth', 'fifth', 'sixth',
'seventh',
    'eighth', 'ninth', 'tenth', 'eleventh', 'twelfth'
]
```

Отсчет в программе ведется с первого дня, а списки Python начинаются с 0 (рис. 13.2). Нужно это учитывать и вычитать 1:

```
>>> day = 3
>>> ordinal[day - 1]
'third'
```

```
ordinal = [          Индекс  День
    'first'            0      1
    'second'           1      2
    'third'            2      3
    'fourth'           3      4
    'fifth'            4      5
    'sixth'            5      6
    'seventh'          6      7
    'eighth'           7      8
    'ninth'            8      9
    'tenth'            9     10
    'eleventh'        10     11
    'twelfth'         11     12
]
```

Рис. 13.2. Дни в программе отсчитываются с 1, а списки Python индексируются с позиции 0

Как вариант, можно использовать и словарь:

```
ordinal = {
    1: 'first', 2: 'second', 3: 'third', 4: 'fourth',
    5: 'fifth', 6: 'sixth', 7: 'seventh', 8: 'eighth',
    9: 'ninth', 10: 'tenth', 11: 'eleventh', 12: 'twelfth',
}
```

В таком случае не нужно вычитать 1. Код прекрасно работает без этого:

```
>>> ordinal[3]
'third'
```

Еще я создал список для подарков:

```
gifts = [
    'A partridge in a pear tree.',
    'Two turtle doves,',
    'Three French hens,',
    'Four calling birds,',
    'Five gold rings,',
    'Six geese a laying,',
    'Seven swans a swimming,',
    'Eight maids a milking,',
    'Nine ladies dancing,',
    'Ten lords a leaping,',
    'Eleven pipers piping,',
    'Twelve drummers drumming,',
]
```

Данное решение удобно, так как можно нарезать список, чтобы извлекать подарки на указанный пользователем день (рис. 13.3):

```
>>> gifts[:3]
['A partridge in a pear tree.',
 'Two turtle doves,',
 'Three French hens,']
```

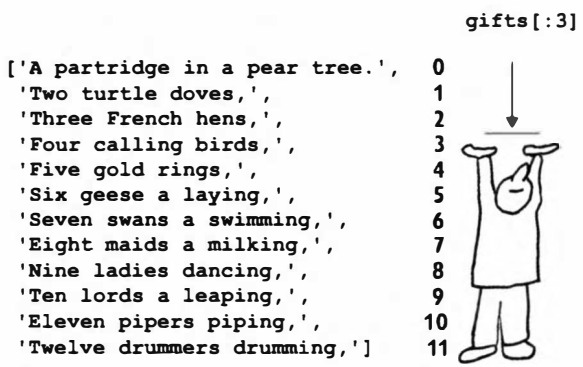


Рис. 13.3. Подарки перечислены согласно дням их дарения в порядке возрастания

Однако мне необходимо обратить порядок. Функция `reversed()` ленива, поэтому в REPL-интерфейсе мне понадобится функция `list()` для принудительного вывода значений:

```
>>> list(reversed(gifts[:3]))
['Three French hens,',
 'Two turtle doves,',
 'A partridge in a pear tree.']
```

Первые две строки каждого куплета одинаковы, меняется только номер (значение `ordinal`) дня:

```
lines = [
    f'On the {ordinal[day - 1]} day of Christmas,',
    'My true love gave to me,'
]
```

Мне нужны эти две строки вместе с подарками. Так как каждый куплет состоит из нескольких строк, имеет смысл использовать список, содержащий весь куплет.

Чтобы добавить подарки к остальным строкам куплета, я применил метод `list.extend()`:

```
>>> lines.extend(reversed(gifts[: day]))
```

Получилось пять строк:

```
>>> lines
['On the third day of Christmas,',
 'My true love gave to me,',
 'Three French hens,',
 'Two turtle doves,',
 'A partridge in a pear tree.']
>>> assert len(lines) == 5
```

Обратите внимание, что я не могу использовать метод `list.append()`. Его легко спутать с методом `list.extend()`, который принимает в качестве аргумента список, расширяет его и добавляет все отдельные элементы в исходный список. Метод `list.append()` предназначен для добавления *лишь одного* элемента, поэтому, если вы передадите ему список, он включит весь список целиком в конец исходного списка!

В конец добавляется итерируемый объект `reversed()`, чтобы выводились три элемента, а не пять:

```
>>> lines.append(reversed(gifts[:day]))
>>> lines
['On the third day of Christmas,',
 'My true love gave to me,',
 <list_reverseiterator object at 0x105bc8588>]
```

Ну что, задумали обратиться к объекту `reverse()` с помощью функции `list()`? Хорошая попытка, но, увы, так вы все равно добавите в конец новый список:

```
>>> lines.append(list(reversed(gifts[:day])))
>>> lines
['On the third day of Christmas,',
 'My true love gave to me,',
 ['Three French hens,', 'Two turtle doves,', 'A partridge in
a pear tree.']]
```

А еще у нас три строки вместо пяти:

```
>>> len(lines)
3
```

Если номер дня больше 1, нужно изменить последнюю строку, добавив союз *And*:

```
if day > 1:
    lines[-1] = 'And ' + lines[-1].lower()
```

Обратите внимание, что это еще одна веская причина для заключения строк в список, ведь элементы списка могут быть *изменены*. Вполне можно представить текст куплетов как строки, но так как они *неизменны*, было бы намного сложнее изменить последнюю строку.

Мне нужно вернуть единое строковое значение, поэтому я соединяю текст символами перевода строки:

```
>>> print('\n'.join(lines))
On the third day of Christmas,
My true love gave to me,
```

```
Three French hens,
Two turtle doves,
A partridge in a pear tree.
```

Функция возвращает соединенные строки и проходит предоставленную мной функцию `test_verse()`.

13.3.2. Создание куплетов

С функцией `verse()` я способен генерировать все необходимые куплеты, перебирая их номера от 1 до переданного пользователем в аргументе `--num`. Я могу собрать куплеты в список:

```
day = 3
verses = []
for n in range(1, day + 1):
    verses.append(verse(n))
```

И проверить, верно ли их количество:

```
>>> assert len(verses) == day
```

Работая с подобным шаблоном создания пустой строки или списка и планируя впоследствии добавить в него элементы с помощью цикла `for`, попробуйте поступить другим способом и реализовать представление списка:

```
>>> verses = [verse(n) for n in range(1, day + 1)]
>>> assert len(verses) == day
```

Лично я предпочитаю функцию `map()`, а не списки. На рис. 13.4 показано, как три упомянутых метода сочетаются друг с другом. В REPL-интерфейсе мне нужна функция `list()` для встряски ленивой функции `map()`. В коде программы «встряска» не потребуется:

```
>>> verses = list(map(verse, range(1, day + 1)))
>>> assert len(verses) == day
```

Все три способа выведут корректное количество куплетов. Выберите тот, что вам по душе.

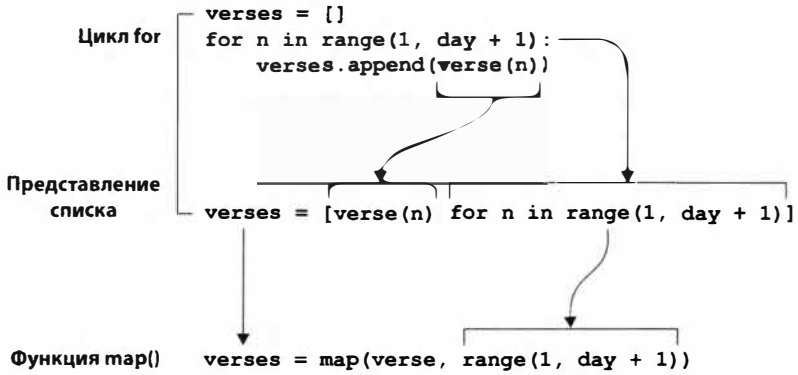


Рис. 13.4. Генерация списка с помощью цикла `for`, с помощью представления списка и с помощью функции `map()`

13.3.3. Вывод куплетов

Как и в случае с «99 бутылками пива» из главы 11, я хочу вывести куплеты с двумя символами перевода строки между ними. Опробуем метод `str.join()`:

```
>>> print('\n\n'.join(verses))
On the first day of Christmas,
My true love gave to me,
A partridge in a pear tree.

On the second day of Christmas,
My true love gave to me,
Two turtle doves,
And a partridge in a pear tree.

On the third day of Christmas,
My true love gave to me,
Three French hens,
Two turtle doves,
And a partridge in a pear tree.
```

Нам также подходит и функция `print()` с опциональным аргументом `file` для вывода текста в файловый дескриптор. Значением `args.outfile` может быть как имя файла, указанное пользователем, так и объект `sys.stdout`:

```
print('\n\n'.join(verses), file=args.outfile)
```

Вы также вольны воспользоваться методом `fh.write()`, главное не забыть добавить завершающий символ перевода строки вкупе с добавляемым функцией `print()`:

```
args.outfile.write('\n\n'.join(verses) + '\n')
```

Существуют сотни способов написать данный алгоритм. Если вы придумали совершенно иной подход, прошедший тест, примите мои поздравления! Пожалуйста, поделитесь им со мной. Я сосредоточился на написании, тестировании и использовании функции `verse()`, но мне хотелось бы увидеть и другие решения.

13.4. Прокачиваем навыки

Установите модуль `emoji` (**https://pypi.org/project/emoji**) и попробуйте вывести в качестве подарков различные эмодзи, а не текст. Например, можно применить код `': bird:'` для вывода значка 🐦 вместо птиц, таких, как куры и голуби. Я бы также использовал значки `': man:'`, `': woman:'` и `': drum:'`, но вы вольны выбирать сами:

```
On the twelfth day of Christmas,
My true love gave to me,
Twelve 🥁s drumming,
Eleven 🤴s piping,
Ten 🤴s a leaping,
Nine 👧s dancing,
Eight 👧s a milking,
Seven 🐦s a swimming,
Six 🐦s a laying,
Five gold 💍s,
Four calling 🐦s,
Three French 🐦s,
Two turtle 🐦s,
And a 🐦 in a pear tree.
```

РЕЗЮМЕ

- Существует множество способов разработки алгоритмов для выполнения рутинных задач. В данной главе я написал и протестировал функцию для обработки одной задачи, а затем сопоставил с ней диапазон входных значений.
- Функция `range()` возвращает целые числа в диапазоне между заданными начальным и конечным значениями, последнее из которых не включено.
- Для обращения значений, возвращаемых функцией `range()`, применяется функция `reverse()`.
- Если для определения аргумента с помощью модуля `argparse` прибегнуть к коду `type=argparse.FileType('wt')`, вы получите файловый дескриптор, открытый для записи текста.
- Файловый дескриптор `sys.stdout` всегда открыт и доступен для записи.
- Благодаря перечислению подарков в виде списка я смог нарезать список, чтобы извлечь все подарки на указанный день. Я использовал функцию `reversed()`, чтобы расположить подарки в правильном порядке, как и предусмотрено текстом песни.
- Строки песни, представленные в виде списка, в отличие от типа `str`, можно изменять. Данное свойство списков пригодилось нам для изменения последней строки всех куплетов, кроме первого.
- Затенением переменной или функции называется присвоение ей имени уже существующего в коде объекта. Если, к примеру, вы создадите переменную с именем существующей функции, данная функция фактически будет затенена. Чтобы избежать подобных ситуаций, пользуйтесь такими инструментами, как Pylint. Помимо затенения они выявляют и многие другие распространенные проблемы с кодом.

Глава 14

Рифмовальщик: генерация рифм с помощью регулярных выражений

Герои фильма «*Принцесса-невеста*» Иниго и Феззик любят придумывать рифмы, особенно когда обсуждают своего жестокого босса Виззини.

Иниго: Он ведь вправе негодовать.
Феззик: Негодовать, негодовать… Просто любит он орать.
Иниго: Может, он не так и зол…
Феззик: Очень просто: он — осел.

Когда я писал альтернативный текст для программы из главы 7, я брал слова вроде «цианид» и придумывал к ним рифмы. Начинал с первой согласной буквы алфавита и подставлял ее на место первой буквы слова: «б» — «бианид», затем «в» — «вианид» и т. д. Пропускал только ту букву, которую и пытался заменить. Метод эффективный, но утомительный. Поэтому я решил написать программу, выполняющую поиск рифм за меня.

По факту, здесь мы имеем дело с алгоритмом поиска/замены, отличным от замены чисел в строке, как было в программе из главы 4, или всех гласных в тексте, как мы делали в главе 8. Мы писали те программы, используя слабоавтоматизированные, *императивные* методы, такие как перебор всех символов в строках, сравнение их с определенным значением и вероятный возврат нового значения.

В финальном решении из главы 8 мы кратко коснулись «регулярных выражений», позволяющих *декларативно* описывать шаблоны текста. Материал данной главы может показаться немного сложным, но все же постарайтесь разобраться в регулярных выражениях, чтобы увидеть их потенциал.

Итак, мы планируем взять некое заданное слово и подобрать к нему рифмы. Например, слово bake рифмуется с такими словами, как cake, make и thrake, последнее из которых на самом деле не словарное слово, а лишь новая строка, созданная мной путем замены буквы b в слове bake на thr.

Мы воспользуемся алгоритмом, разбивающим каждое слово на начальные (ведущие) согласные и остальную часть. Так, слово bake разбивается на b и ake. Далее буква b заменяется остальными согласными из алфавита, а также следующими группами согласных:

```
bl br ch cl cr dr fl fr gl gr pl pr sc sh sk sl sm sn sp st
sw th tr tw thw wh wr sch scr shr sph spl spr squ str thr
```

Ниже показаны первые три слова, которые программа срифмует со словом cake:

```
$ ./rhymer.py cake | head -3
bake
blake
brake
```

А вот последние три:

```
$ ./rhymer.py cake | tail -3
xake
yake
zake
```

Вывод должен быть отсортирован в алфавитном порядке, это важно для прохождения тестов.

Мы заменим ведущие согласные списком других согласных, чтобы создать в общей сложности 56 слов:

```
$ ./rhymer.py cake | wc -l
      56
```

Обратите внимание, что мы меняем *все* ведущие согласные, а не только первую. Например, в слове chair нужно заменить буквы ch, а не букву c:

```
$ ./rhymer.py chair | tail -3
xair
yair
zair
```

Если слово начинается с гласной буквы, например apple, мы добавляем согласные буквы в начало слова, чтобы получились такие слова, как bapple и shrapple.

```
$ ./rhymer.py apple | head -3
bapple
blapple
brapple
```

Поскольку согласной под *замену нет,* слова, начинающиеся с гласной буквы, позволяют получить 57 рифмующихся слов:

```
$ ./rhymer.py apple | wc -l
      57
```

Чтобы слегка упростить задачу, мы выведем результат строчными буквами, даже если слова на входе содержат прописные:

```
$ ./rhymer.py GUITAR | tail -3
xuitar
yuitar
zuitar
```

Для слов, в которых нет гласных букв, должно выводиться сообщение, что они не могут быть зарифмованы:

```
$ ./rhymer.py RDNZL
Cannot rhyme "RDNZL"
```

Задачу поиска ведущих согласных букв можно значительно упростить с помощью регулярных выражений.

Прочитав данную главу, вы научитесь:

- составлять и использовать регулярные выражения;
- использовать представление списка с охранным выражением;
- различать представление списка с охранным выражением и функцию `filter()`;
- применять концепции «истинности» при оценке типов Python в логическом контексте.

14.1. Создание файла rhymer.py

Программа принимает позиционный строковый аргумент — слово, к которому следует подобрать рифму. На рис. 14.1 показана проектная, эффектная и корректная диаграмма.

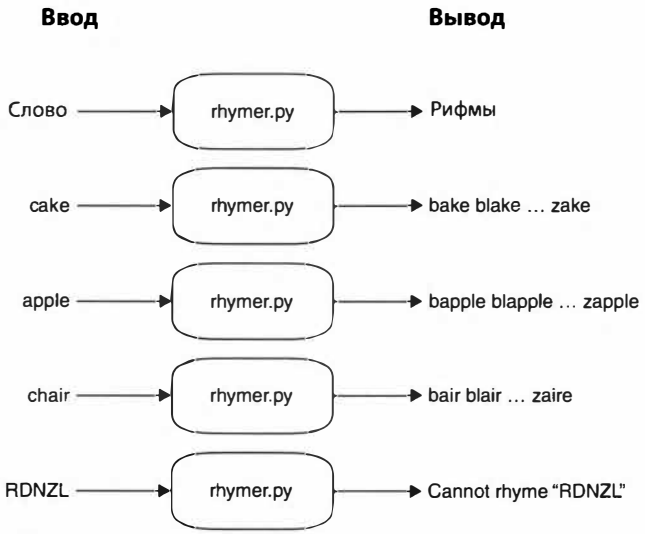


Рис. 14.1. На входе программа-рифмовальщик принимает слово, а на выходе выводит список рифм или ошибку

Если аргумент не передан или программа запущена с флагом `-h` или `--help`, она должна вывести справочную информацию:

```
$ ./rhymer.py -h
usage: rhymer.py [-h] word

Make rhyming "words"

positional arguments:
  word        A word to rhyme

optional arguments:
-h, --help  show this help message and exit
```

14.1.1. Разбиение слова

Как по мне, наиболее сложная задача заключается в том, чтобы разбить переданное пользователем слово на ведущие согласные и остальную часть.

Для начала определим функцию-заглушку, которой присвоим имя `stemmer()`. Сейчас она ничего не делает:

```
def stemmer():
    """Возвращаем ведущие согласные (если есть) и "остальную
       часть" слова."""
    pass
```

Оператор `pass` бездействует. Поскольку функция не возвращает значение, интерпретатор по умолчанию возвращает `None`.

Затем мы определяем функцию `test_stemmer()`, которая поможет нам подобрать значения, передаваемые функции, и продумать предполагаемый результат. Тест должен содержать как подходящие значения, такие как cake и apple, к которым можно подобрать рифмы, так и недопустимые, например пустую строку или число:

```
def test_stemmer():
    """ Test stemmer """
❶   assert stemmer('') == ('', '')
❷   assert stemmer('cake') == ('c', 'ake')
❸   assert stemmer('chair') == ('ch', 'air')
❹   assert stemmer('APPLE') == ('', 'apple')
```

```
❺ assert stemmer('RDNZL') == ('rdnzl', '')
❻ assert stemmer('123') == ('123', '')
```

Тесты проверяют следующие допустимые и недопустимые входные данные.

❶ Пустая строка.
❷ Слово с одной ведущей согласной буквой.
❸ Слово с несколькими ведущими согласными буквами.
❹ Слово, начинающееся с гласной буквы, но написанное прописными буквами для проверки, возвращается ли результат строчными буквами.
❺ Слово без гласных букв.
❻ Значение без букв, например число.

Функция `stemmer()` всегда будет возвращать кортеж из двух элементов (начало (`start`) и остальная часть (`rest`) слова). (Можно иначе записать данную функцию, но тогда не забудьте изменить тест, чтобы он соответствовал коду.) Второй элемент кортежа — остальную часть слова — мы планируем использовать для генерации рифм. Например, слово cake будет записано как кортеж `('c', 'ake')`, а chai — как `('ch', 'air')`. Аргумент `APPLE` не имеет начала и состоит исключительно из оставшейся части слова, написанной прописными буквами. В данном случае согласные добавляются ко всему слову, и рифмы выводятся строчными буквами.

Составляя тесты, я обычно передаю своим функциям и программам как допустимые, так и недопустимые данные. Три тестовых значения не рифмуются: пустая строка (`''`), строка без гласных (`'RDNZL'`) и строка без букв (`'123'`). При передаче подобных значений функция `stemmer()` все равно возвращает кортеж с переданным значением, написанным строчными буквами, в качестве первого элемента кортежа (начала слова) и пустой строкой в качестве второго (остальной части слова). За обработку значения, не содержащего части, пригодной для рифмования, отвечает вызывающий код.

14.1.2. Использование регулярных выражений

Разумеется, данную программу *можно* написать и без регулярных выражений, но я думаю, вам понравится то, насколько они эффективнее кода поиска/замены, набранного вручную.

Для начала необходимо импортировать модуль `re`:

```
>>> import re
```

Я рекомендую прочитать справку к указанному модулю (`help(re)`), чтобы научиться полноценно работать с регулярными выражениями. Это важная и сложная тема, которой посвящено множество книг и научных направлений (я советую ознакомиться с книгой «*Регулярные выражения*» Джеффри Фридла. Питер, 2018). Существует большое количество полезных веб-сайтов, посвященных приемам работы с регулярными выражениями, а некоторые ресурсы могут помочь вам научиться их составлять (например, **https://regexr.com**). Здесь мы коснемся только самых простых вещей.

Цель данной главы — написать регулярное выражение, с помощью которого мы будем находить согласные в начале строки. Мы можем определить согласные как символы английского алфавита, отличные от гласных букв (a, e, i, o и u). Функция `stemmer()` возвращает результат исключительно строчными буквами, поэтому необходимо определить только 21 согласную. Вы можете просто перечислить их, но лучше написать следующий код.

Начнем с `string.ascii_lowercase`:

```
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

Затем хорошо бы реализовать представление списка с охранным выражением, чтобы отфильтровать гласные. Поскольку нам нужна строка, а не список согласных, мы можем использовать функцию `str.join()`, чтобы создать новое строковое значение:

```
>>> import string as s
>>> s.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> consonants = ''.join([c for c in s.ascii_lowercase if c not in
'aeiou'])
>>> consonants
'bcdfghjklmnpqrstvwxyz'
```

Если применить цикл `for` и оператор `if`, получится более объемный код (рис. 14.2):

```
consonants = ''
for c in string.ascii_lowercase:
    if c not in 'aeiou':
        consonants += c
```

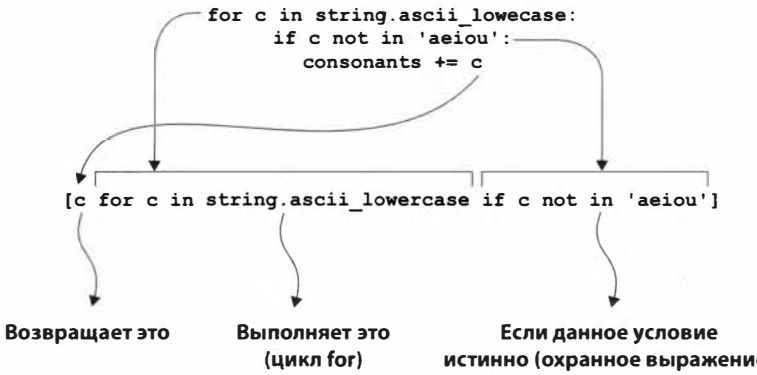


Рис. 14.2. Цикл for (сверху) можно записать как представление списка (снизу). Оно содержит охранное выражение, поэтому выбираются только согласные буквы по аналогии с оператором if (сверху)

В главе 8 мы создали «символьный класс» для сопоставления гласных букв, перечислив их в квадратных скобках: `[aeiou]`. Мы можем поступить так же в данной программе с согласными буквами:

```
>>> pattern = '[' + consonants + ']'
>>> pattern
'[bcdfghjklmnpqrstvwxyz]'
```

В модуле `re` есть две поисковые функции, `re.match()` и `re.search()`, которые я всегда путаю. Они обе ищут совпадение (первый аргумент) в переданном тексте, но функция `re.match()` ведет поиск *с начала* текста, тогда как функция `re.search()` ищет совпадения в *любом* месте текста.

В нашем случае идеально подойдет функция `re.match()`, поскольку мы ищем согласные буквы в начале строки (рис. 14.3).

Пробуем сопоставить данный шаблон с переданным текстом. Если это удается, мы получаем объект `re.Match`; в противном случае возвращается значение `None`.

```
>>> text = 'chair'
>>> re.match(pattern, text)
<re.Match object; span=(0, 1), match='c'>
```

Сопоставление завершено успешно, поэтому мы видим «приведенную к строке» версию объекта `re.Match`.

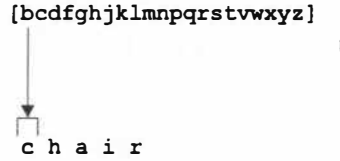


Рис. 14.3. Символьный класс согласных букв соответствует букве «c» в начале слова chair.

Строка `match='c'` указывает, что благодаря регулярному выражению найдена строка `'c'` в начале слова. Обе функции, `re.match()` и `re.search()`, вернут в случае успеха объект `re.Match`. Рекомендую прочитать справку (`help(re.Match)`) и больше узнать о возможностях функции `re.match()`:

```
>>> match = re.match(pattern, text)
>>> type(match)
<class 're.Match'>
```

Как сделать так, чтобы регулярное выражение соответствовало буквам `'ch'`? Мы можем указать знак + после символьного класса, чтобы дать знать интерпретатору, что нам нужен *один или более* символов (рис. 14.4). (Напоминает код `nargs='+'` — один или более аргументов, не правда ли?) Ниже я использую f-строку, чтобы создать шаблон:

```
>>> re.match(f'[{consonants}]+', 'chair')
<re.Match object; span=(0, 2), match='ch'>
```

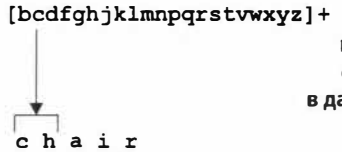


Рис. 14.4. Добавив знак «плюс» к классу, мы укажем на соответствие одному или более символам

Что произойдет при вводе слов строки без ведущих согласных, таких как, например, apple (рис. 14.5)?

```
>>> re.match(f' [{consonants}]+', 'apple')
```

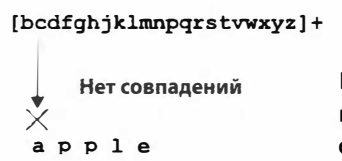


Рис. 14.5. Указанное регулярное выражение не соответствует слову, начинающемуся с гласной буквы

Результат отрицательный. Каков тип возвращенного значения?

```
>>> type(re.match(f' [{consonants}]+', 'apple'))
<class 'NoneType'>
```

Обе функции, `re.match()` и `re.search()`, возвращают значение `None`, указывая на отсутствие совпадений. Нам известно, что ведущие согласные буквы есть только в части слов, так что результат не удивляет. Вскоре вы узнаете, как такие совпадения сделать опциональными.

14.1.3. Группы захвата

Прекрасно, если в словах есть (или отсутствуют) ведущие согласные буквы, сейчас это не так важно. Сосредоточимся на разбиении слов на две части: согласные (если есть) и остальную часть слова.

Мы можем заключить шаблон регулярного выражения в круглые скобки, чтобы создать так называемую группу захвата. Если шаблон регулярного выражения совпадает, мы можем извлечь части, применяя метод `re.Match.groups()` (рис. 14.6):

```
>>> match = re.match(f' ([{consonants}]+)', 'chair')
>>> match.groups()
('ch',)
```

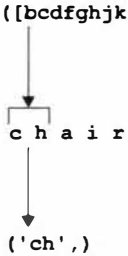


Рис. 14.6. Окружив шаблон круглыми скобками, мы превращаем соответствующий текст в «группу захвата»

Для захвата части слова после согласных используем точку (`.`), чтобы указать на соответствие, и знак плюс (`+`), чтобы обозначить «один или более». Помещаем шаблон в круглые скобки (рис. 14.7):

```
>>> match = re.match(f'([{consonants}]+)(.+)', 'chair')
>>> match.groups()
('ch', 'air')
```

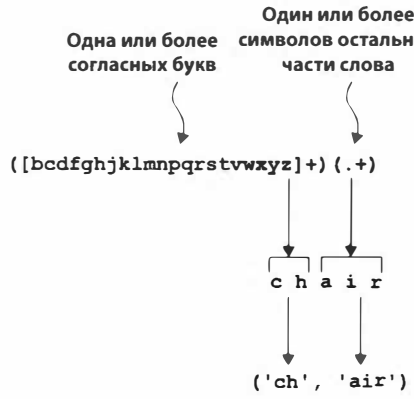


Рис. 14.7. Мы определяем две группы захвата для доступа к ведущей согласной букве и всем последующим

Что произойдет при обработке слова apple? Совпадения согласных не обнаруживается, поэтому *вся процедура сопоставления завершается ошибкой*, и возвращается значение None (рис. 14.8):

```
>>> match = re.match(f' ([{consonants}]+)(.+)', 'apple')
>>> match.groups()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'groups'
```

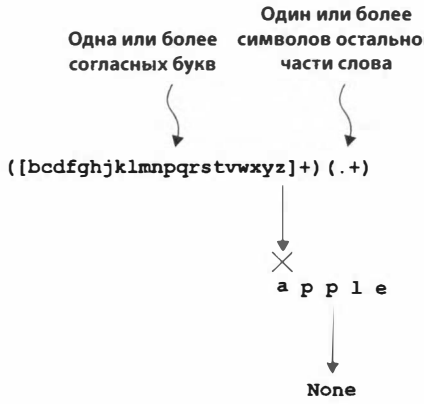


Рис. 14.8. Программа по-прежнему выдает ошибку, если текст начинается с гласной

Напомню, что при отсутствии совпадений по шаблону функция re.match() возвращает None. Мы можем добавить вопросительный знак (?) в конце шаблона consonants, чтобы сделать его необязательным (рис. 14.9):

```
>>> match = re.match(f' ([{consonants}]+)?(.+)', 'apple')
>>> match.groups()
(None, 'apple')
```

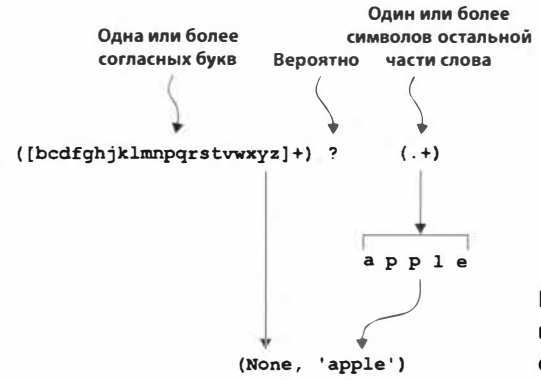


Рис. 14.9. Благодаря знаку вопроса шаблон становится опциональным

Функция `match.groups()` возвращает кортеж, содержащий совпадения для каждой группы захвата в скобках. Вы также можете использовать функцию `match.group()` (единственное число) с номером группы, чтобы получить соответствующую группу. Обратите внимание, что группы нумеруются с 1:

```
>>> match.group(1)
>>> match.group(2)
'apple'
```

Для слова apple нет совпадений для первой группы, поэтому в результате получается значение None.

Вторая группа захватывает оставшуюся часть — слово целиком.

Если передать слово chair, мы получим значения для обеих групп:

```
>>> match = re.match(f' ([{consonants}]+)?(.+)', 'chair')
>>> match.group(1)
'ch'
>>> match.group(2)
'air'
```

До сих пор мы имели дело только со словами, набранными строчными буквами, поскольку на них завязаны значения, выдаваемые нашей программой. Тем не менее давайте посмотрим, что произойдет при передаче слова прописными буквами:

```
>>> match = re.match(f' ([{consonants}]+)?(.+)', 'CHAIR')
>>> match.groups()
(None, 'CHAIR')
```

Неудивительно, что попытка провалилась. Согласно шаблону, определяются только строчные буквы. Мы вольны добавить в шаблон и все

прописные согласные буквы, но проще использовать третий опциональный аргумент функции `re.match()`, отключающий учет регистра символов:

```
>>> match = re.match(f' ([{consonants}]+)?(.+)', 'CHAIR',
                     re.IGNORECASE)
>>> match.groups()
('CH', 'AIR')
```

Либо можно принудительно преобразовать символы переданного тексты в строчные:

```
>>> match = re.match(f' ([{consonants}]+)?(.+)', 'CHAIR'.lower())
>>> match.groups()
('ch', 'air')
```

Что произойдет при передаче слова, состоящего исключительно из согласных?

```
>>> match = re.match(f' ([{consonants}]+)?(.+)', 'rdnzl')
>>> match.groups()
('rdnz', 'l')
```

Вы ожидали, что первая группа захватит *все* согласные, а вторая — ничего? Может показаться удивительным, что буква l была выделена в последнюю группу, как показано на рис. 14.10. Однако задумайтесь о том, что механизм регулярных выражений работает *буквально*. Мы описали опциональную группу из одной или более согласных, за которыми *должен следовать* один или более других символов. Символ l как раз и считается тем самым «одним или более» символом, поэтому регулярное выражение точно соответствует нашему запросу.

Рис. 14.10. Программа выдает именно то, что мы просим, но, возможно, не то, что хотим получить

Если мы изменим код `(.+)` на `(.*)`, то есть сделаем шаблон *равным нулю или более,* программа будет работать так, как ожидается:

```
>>> match = re.match(f' ([{consonants}]+)?(.*)', 'rdnzl')
>>> match.groups()
('rdnzl', '')
```

Наше регулярное выражение неполноценно, так как не обрабатывает сопоставление с небуквенными символами вроде `123`. Точнее, совпадение будет обнаружено, так как точка `(.)` соответствует цифрам, но нам это не нужно:

```
>>> re.match(f' ([{consonants}]+)?(.*)?', '123')
<re.Match object; span=(0, 3), match='123'>
```

Необходимо указать, что после согласных должна следовать *хотя бы одна гласная,* за которой могут идти другие символы. Мы можем использовать другой символьный класс для описания гласных букв. Поскольку их нужно захватывать, помещаем последовательность в круглые скобки: `([aeiou])`. Следом может располагаться *ноль или более* символов, которые также требуется захватить, поэтому используем код `(.*)`, как показано на рис. 14.11.

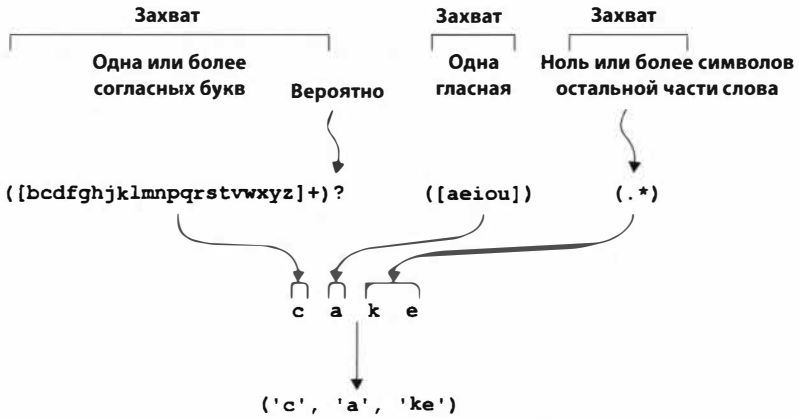


Рис. 14.11. Регулярное выражение теперь требует наличия гласной в переданном слове

Давайте опробуем данное решение на допустимых значениях:

```
>>> re.match(f' ([{consonants}]+)?([aeiou])(.*)', 'cake').groups()
('c', 'a', 'ke')
>>> re.match(f' ([{consonants}]+)?([aeiou])(.*)', 'chair').groups()
('ch', 'a', 'ir')
>>> re.match(f' ([{consonants}]+)?([aeiou])(.*)', 'apple').groups()
(None, 'a', 'pple')
```

А теперь на недопустимых. Как видите, программа не обнаруживает совпадений, если слово не содержит гласных букв или состоит из иных символов:

```
>>> type(re.match(f' ([{consonants}]+)?([aeiou])(.*)', 'rdnzl'))
<class 'NoneType'>
>>> type(re.match(f' ([{consonants}]+)?([aeiou])(.*)', '123'))
<class 'NoneType'>
```

14.1.4. Истинность

Итак, наша программа может получить входные данные, которые нельзя рифмовать, так что же должна делать с ними функция `stemmer()`? Некоторым разработчикам в подобных случаях нравится выбрасывать исключения. Мы уже сталкивались с ними при запросе несуществующего индекса списка или ключа словаря. Если исключения не перехватываются и не обрабатываются, они приводят к сбою программы!

Я стараюсь избегать кода, выбрасывающего исключения. Моя функция `stemmer()` в любом случае будет возвращать кортеж из двух элементов (начало (`start`) и остальная часть (`rest`) слова), и я использую пустую строку для обозначения отсутствующего значения, а не `None`. Приведу один из вариантов кода для возврата этих кортежей:

```
if match:
    p1 = match.group(1) or ''
    p2 = match.group(2) or ''
    p3 = match.group(3) or ''
    return (p1, p2 + p3)
else:
    return (word, '')
```

В результате получаем значение `None`, если совпадение не обнаружено, то есть «ложно». Если обнаружено, значит, «истинно».

Существуют три группы захвата, которые мы можем поместить в три переменные. Чтобы убедиться в отсутствии возврата значений `None`, используем оператор `or`, чтобы оценить выражение слева как «истинное», либо в противном случае передать пустую строку (справа).

Возвращаем кортеж, содержащий первую часть слова (вероятные согласные) и «остальную часть» слова (гласную плюс все остальные символы).

Если в результате получается значение `None`, возвращаем кортеж слова и пустую строку, указывая, что нет «остальной части» слова для рифмы.

Давайте на минутку остановимся на операторе `or`, используемом для выбора между кодом слева и справа. Оператор `or` вернет первое значение, которое согласно логике оценено как «истинное»:

Вы вольны применить данные концепции при написании функции `stemmer()`, тестируемой с помощью `test_stemmer()`. Напомню, что если обе эти функции есть в программе `rhymer.py`, можно проверить все функции с именем, начинающиеся со слова `test_` следующим образом:

```
$ pytest -xv rhymer.py
```

14.1.5. Формирование вывода

Итак, наша программа должна быть способна:

1) принимать позиционный строковый аргумент;
2) делить аргумент на две части: ведущие согласные буквы и остальную часть слова;

3) если разбиение выполнено успешно, объединять с прочими согласными буквами «остальную часть» слова (которой может быть представлено все слово при отсутствии ведущих согласных). Программа *не* должна включать исходную согласную букву и обязана сортировать рифмы;
4) если слово разбить невозможно, выводить сообщение `Cannot rhyme "<word>"`.

Настало время создать программу. Удачного мозгового штурма!

14.2. Решение

«Не рифмуйте больше здесь!»
«Хочет кто-нибудь поесть?»

Давайте рассмотрим один из способов решения задачи. Насколько ваш вариант отличается от моего?

```
#!/usr/bin/env python3
"""Генератор рифм"""

import argparse
import re
import string

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Make rhyming "words"',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('word', metavar='word', help='A word
                        to rhyme')

    return parser.parse_args()

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
```

Модуль `re` предназначен для работы с регулярными выражениями.

Получаем аргументы командной строки.

Делим слово из аргумента на две возможные части. Поскольку функция `stemmer()` всегда возвращает кортеж из двух элементов, мы можем распаковать значения в две переменные.

Определяем префиксы, которые будут добавлены для создания рифм.

```
prefixes = list('bcdfghjklmnpqrstvwxyz') + (
    'bl br ch cl cr dr fl fr gl gr pl pr sc '
    'sh sk sl sm sn sp st sw th tr tw thw wh wr '
    'sch scr shr sph spl spr squ str thr').split()

start, rest = stemmer(args.word)
if rest:
    print('\n'.join(sorted([p + rest for p in prefixes if
          p != start])))
else:
    print(f'Cannot rhyme "{args.word}"')
```

Проверяем, есть ли в аргументе часть слова, которую мы можем использовать для подбора рифм.

Если возможно, используем представление списка для перебора всех префиксов и добавления их к остальной части слова. Применяем охранное выражение, чтобы гарантировать, что нет префиксов, совпадающих с началом слова. Сортируем все значения и выводим их, объединив символами перевода строки.

Если в качестве «остальной части» слова рифмовать нечего, оповещаем об этом пользователя.

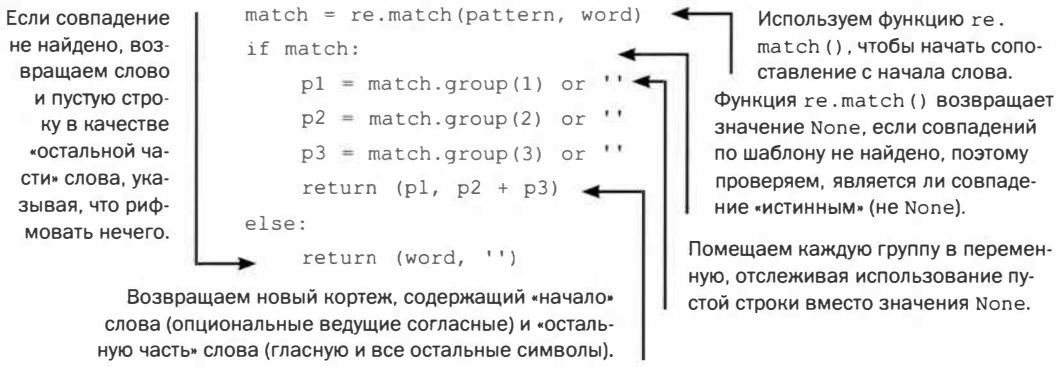
```
# --------------------------------------------------
def stemmer(word):
    """Возвращаем ведущие согласные (если есть) и "остальную
       часть" слова."""

    word = word.lower()
    vowels = 'aeiou'
    consonants = ''.join(
        [c for c in string.ascii_lowercase if c not in vowels])
    pattern = (
        '([' + consonants + ']+)?' # опционально захватываем одну
                                   # или более
        '([' + vowels      + '])'   # захватываем как минимум одну
                                   # гласную
        '(.*)'                     # захватываем ноль или более
                                   # знаков остальной части
    )
```

Пишем слово строчными буквами.

Поскольку мы пользуемся гласными более одного раза, присваиваем их переменной.

Согласные — все буквы, не являющиеся гласными. Сопоставляем только строчные буквы.

Шаблон определяется с помощью последовательных строковых литералов, которые интерпретатор объединяет в одну строку. Разбивая шаблон на отдельные строки, мы можем прокомментировать каждую часть регулярного выражения.




```
# -----------------------------------------------
def test_stemmer():
    """test the stemmer"""

    assert stemmer('') == ('', '')
    assert stemmer('cake') == ('c', 'ake')
    assert stemmer('chair') == ('ch', 'air')
    assert stemmer('APPLE') == ('', 'apple')
    assert stemmer('RDNZL') == ('rdnzl', '')
    assert stemmer('123') == ('', '')

# -----------------------------------------------
if __name__ == '__main__':
    main()
```

Тесты для функции `stemmer()`. Обычно я размещаю модульные тесты сразу после функций, которые нужно протестировать.

14.3. Обсуждение

Есть много вариантов нашей программы. Традиционно я решил разбить задачу на модули, которые мог бы написать и протестировать. Наша главная цель — разделить слова на ведущую согласную букву или буквы (которых может и не быть, если слово начинается с гласной) и остальную часть. Если разбиение выполняется, можно создавать рифмы; а если нет — необходимо оповестить об этом пользователя.

14.3.1. Выделение основной части слова

Для нашей программы «основной» является часть слова за исключением любого количества ведущих (в начале слова) согласных. Ее я определяю

с помощью представления списка с охранным выражением, обрабатывая только негласные буквы:

```
>>> vowels = 'aeiou'
>>> consonants = ''.join([c for c in string.ascii_lowercase
                          if c not in vowels])
```

Я уже неоднократно говорил, что представление списка — это краткий способ создания списка, более предпочтительный, чем цикл `for`, в плане добавления элементов к существующему списку. Мы используем оператор `if`, чтобы обрабатывать исключительно негласные буквы. Это так называемое *охранное выражение*. В результирующий список будут включены только «истинные» элементы.

Мы несколько раз рассматривали функцию `map()` и говорили о том, что это *функция высшего порядка* (HOF), поскольку она принимает *другую функцию* в качестве первого аргумента и применяет ее ко всем элементам из указанного *итерируемого объекта* (который можно *перебирать* или *итерировать* — например, список).

Теперь я расскажу еще об одной функции высшего порядка, `filter()`. Она тоже принимает функцию и итерируемый объект в качестве аргументов (рис. 14.12). Как и в случае с представлением списка с охранным выражением, в результирующий список допускаются только те элементы, которые возвращают «истинное» значение из функции.

Рис. 14.12. Функции `map()` и `filter()` принимают в качестве аргументов функцию и итерируемый объект и обе генерируют новый список

Вот еще один способ реализовать концепцию представления списка с помощью функции `filter()`:

```
>>> consonants = ''.join(filter(lambda c: c not in vowels, string.
ascii_lowercase))
```

Как и в случае с функцией `map()`, я использую ключевое слово `lambda` для создания *анонимной функции*. `c` — это переменная, которая будет содержать аргумент, представленный каждым символом из `string.ascii_lowercase`. Тело функции полностью представляет собой вычисление `c not in vowels`. Любая гласная оценивается как `False`:

```
>>> 'a' not in vowels
False
```

А любая согласная — как `True`:

```
>>> 'b' not in vowels
True
```

Следовательно, только согласные будут «пропускаться» функцией `filter()`. Вспомните «синие» автомобили из главы 8. Напишем код функции `filter()`, принимающей только те автомобили, названия которых начинаются со строки `blue`:

```
>>> cars = ['blue Honda', 'red Chevy', 'blue Ford']
>>> list(filter(lambda car: car.startswith('blue '), cars))
['blue Honda', 'blue Ford']
```

Если переменная `car` имеет значение `red Chevy`, анонимная функция возвращает `False`, и данное значение переменной отклоняется:

```
>>> car = 'red Chevy'
>>> car.startswith('blue ')
False
```

Важное замечание: если ни один из элементов исходного итерируемого объекта не принимается, функция `filter()` выдает пустой список (`[]`). К примеру, я мог бы отфильтровать числа больше 10. Обратите внимание, что функция `filter()` *ленива*, поэтому в REPL-интерфейсе для вывода результатов мне понадобится вызвать функцию `list()`:

```
>>> list(filter(lambda n: n > 10, range(0, 5)))
[]
```

Представление списка также вернет пустой список:

```
>>> [n for n in range(0, 5) if n > 10]
[]
```

На рис. 14.13 показаны сходства между вариантами создания списка `consonants`:

— императивным способом с циклом `for`;
— с помощью идиоматического представления списка с охранным выражением;
— чисто функциональным методом с функцией `filter()`.

Все они вполне приемлемы, хотя, пожалуй, самым «питоническим» приемом является представление списка. Цикл `for` должен быть хорошо знаком программисту на C или Java, в то время как подход с функцией `filter()` с ходу узнаваем для разработчика на Haskell или даже на Lisp-подобном языке. Код с функцией `filter()` может работать медленнее, чем представление списка, особенно если итерируемый объект велик. Выберите тот способ, который больше подходит под ваш стиль программирования и под ваши задачи.

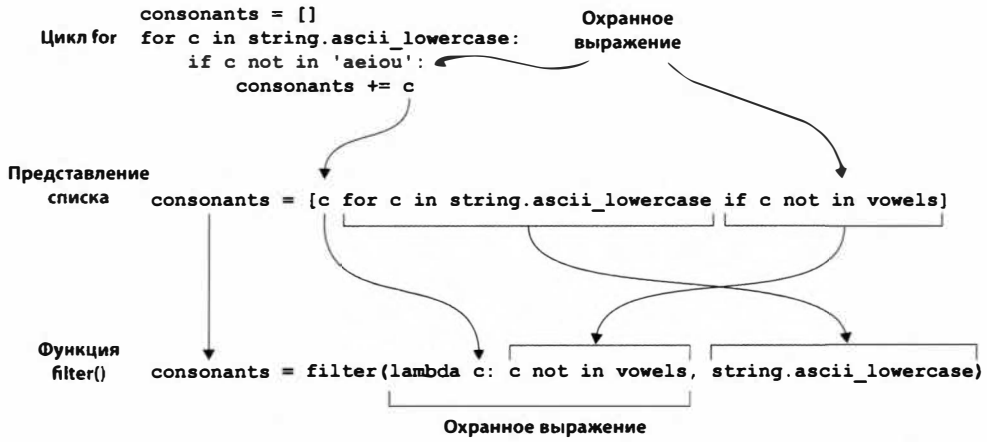


Рис. 14.13. Три способа создания списка с согласными: цикл `for` с оператором `if`, представление списка с охранным выражением и функция `filter()`

14.3.2. Форматирование и комментирование регулярного выражения

В начале главы мы говорили об отдельных частях использованного мной регулярного выражения. Пришло время рассказать, как я отформатировал его в коде. Я воспользовался интересной возможностью интерпретатора Python, в котором смежные строковые литералы неявно объединяются. Взгляните, как следующие четыре строки становятся одной:

```
>>> this_is_just_to_say = ('I have eaten '
... 'the plums '
... 'that were in '
... 'the icebox')
>>> this_is_just_to_say
'I have eaten the plums that were in the icebox'
```

Обратите внимание, что в конце строк нет запятых, иначе был бы создан кортеж с четырьмя отдельными строками:

```
>>> this_is_just_to_say = ('I have eaten ',
... 'the plums ',
... 'that were in ',
... 'the icebox')
>>> this_is_just_to_say
('I have eaten ', 'the plums ', 'that were in ', 'the icebox')
```

Преимущество оформления регулярного выражения отдельными строками заключается в том, что можно снабдить их комментариями, чтобы читатель разобрался в каждой части кода:

```
pattern = (
  '([' + consonants + ']+)?' # опционально захватываем одну или
                             # более
  '([' + vowels     + '])'   # захватываем как минимум одну гласную
  '(.*)'                     # захватываем ноль или более
                             # знаков остальной части
)
```

Отдельные строки будут конкатенированы интерпретатором Python в одну строку:

```
>>> pattern
'([bcdfghjklmnpqrstvwxyz]+)?([aeiou])(.*)'
```

Можно написать все регулярное выражение в одной строке. Однако подумайте, какую версию вам было бы удобнее читать и поддерживать, с комментариями или показанную ниже?*

```
pattern = f' ([{consonants}]+)?([{vowels}])(.*)'
```

14.3.3. Вызов функции stemmer() вне программы

Еще одна замечательная особенность Python: программа *rhymer.py* является своего рода общедоступным *модулем*. Конечно, вы не писали ее с целью применения в качестве контейнера многократно используемых (и проверенных!) функций, но это так. Вы даже можете вызывать функции прямо в REPL-интерфейсе.

Чтобы данный прием сработал, выполняйте команду `python3` в том же каталоге, в котором находится файл *rhymer.py*:

```
>>> from rhymer import stemmer
```

Теперь можно выполнить и протестировать функцию `stemmer()` вручную:

```
>>> stemmer('apple')
('', 'apple')
>>> stemmer('banana')
('b', 'anana')
>>> import string
>>> stemmer(string.punctuation)
('!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~', '')
```

Глубокий смысл выражения `if __name__ == '__main__':`

Обратите внимание, что если в файле rhymer.py изменить показанные ниже строки:

```
if __name__ == '__main__':
    main()
```

на код:

```
main()
```

* «Просматривать код, написанный вами две недели назад, все равно что видеть его впервые». *Дэн Хурвиц*.

функция `main()` будет вызываться при импорте модуля:

```
>>> from rhymer import stemmer
usage: [-h] str
: error: the following arguments are required: str
```

Происходящее связано с тем, что команда `import rhymer` выполняет код в файле *rhymer.py* полностью. Если в последней строке модуля указан вызов функции `main()`, то она будет выполнена!

Когда файл *rhymer.py* запускается как программа, переменной `__name__` присваивается значение `__main__`. Это единственная ситуация, при которой выполняется функция `main()`. Когда модуль импортируется другим модулем, переменной `__name__` присваивается значение `rhymer`.

Если вы явно не импортируете функции, можно указать полное имя функции с именем модуля:

```
>>> import rhymer
>>> rhymer.stemmer('cake')
('c', 'ake')
>>> rhymer.stemmer('chair')
('ch', 'air')
```

Есть много преимуществ в разработке программ из нескольких небольших функций, а не одной длинной. Во-первых, небольшие функции гораздо легче писать, поддерживать и тестировать. Во-вторых, можно превратить свои аккуратные, проверенные функции в модули и использовать их в разных программах.

С ростом опыта в программировании вы обнаружите, что постоянно решаете одни и те же проблемы. Гораздо проще создавать модули с многократно используемым кодом, чем копировать фрагменты кода из одной программы в другую. Если вы обнаружите ошибку в модуле, используемом в нескольких программах, легче исправить код только в нем, и все программы, в которых импортируется данный модуль, автоматически будут работать с исправленным кодом. При отсутствии модулей придется искать и изменять проблемный код в каждой программе (молясь, что не возникнет новых проблем, так как проблемный код тесно связан с остальным).

14.3.4. Создание строк с рифмами

Хочу, чтобы моя функция `stemmer()` возвращала кортеж из двух элементов (начало, остальная часть) для любого переданного пользователем слова. В таком случае я могу распаковать два значения в две переменные:

```
>>> start, rest = stemmer('cat')
>>> start
'c'
>>> rest
'at'
```

Если остальная часть слова выделена, добавляю к ней все свои префиксы:

```
>>> prefixes = list('bcdfghjklmnpqrstvwxyz') + (
...     'bl br ch cl cr dr fl fr gl gr pl pr sc '
...     'sh sk sl sm sn sp st sw th tr tw wh wr'
...     'sch scr shr sph spl spr squ str thr').split()
```

Я воспользуюсь другим представлением списка с охранным выражением, чтобы пропускать префиксы, совпадающие с началом слова. В результате получается новый список, который передается функции `sorted()` для получения строк в правильном порядке:

```
>>> sorted([p + rest for p in prefixes if p!= start])
['bat', 'blat', 'brat', 'chat', 'clat', 'crat', 'dat', 'drat',
'fat', 'flat', 'frat', 'gat', 'glat', 'grat', 'hat', 'jat', 'kat',
'lat', 'mat', 'nat', 'pat', 'plat', 'prat', 'qat', 'rat', 'sat',
'scat', 'schat', 'scrat', 'shat', 'shrat', 'skat', 'slat', 'smat',
'snat', 'spat', 'sphat', 'splat', 'sprat', 'squat', 'stat',
'strat', 'swat', 'tat', 'that', 'thrat', 'thwat', 'trat', 'twat',
'vat', 'wat', 'what', 'wrat', 'xat', 'yat', 'zat']
```

Затем я вывожу этот список, перемежая его символами перевода строки. Если у переданного слова нет «остальной части», выводится сообщение, что оно не может быть зарифмовано:

```
if rest:
    print('\n'.join(sorted([p + rest for p in prefixes if p!= start])))
else:
    print(f'Cannot rhyme "{args.word}"')
```

14.3.5. Функция stemmer() без регулярных выражений

Разумеется, можно решить задачу без регулярных выражений. Начнем с поиска позиции первой гласной в переданной строке. Если она обнаружена, нарезаем список, возвращая части строки до и после позиции гласной:

Приводим слово к строчным буквам, чтобы избежать проблем с прописными.

Отфильтровываем гласные «aeiou», найдя их в слове, а затем сопоставляем представленные гласные с `word.index`, чтобы определить их позиции. Это редкий случай, когда необходимо использовать функцию `list()`, чтобы ленивая функция `map()` вывела результат, который потребуется последующему оператору `if`.

```
def stemmer(word):
    """Возвращаем ведущие согласные (если есть) и "остальную
       часть" слова."""
    word = word.lower()
    vowel_pos = list(map(word.index, filter(lambda v: v in word,
                         'aeiou')))
    if vowel_pos:
        first_vowel = min(vowel_pos)
        return (word[: first_vowel], word[first_vowel:])
    else:
        return (word, '')
```

Проверяем, есть ли в слове гласные.

Определяем индекс первой гласной, используя минимальное (`min`) значение из числа позиций.

Возвращаем кортеж со срезом слова от начала до первой гласной и срезом от первой гласной буквы до конца слова.

В противном случае в слове не обнаружено ни одной гласной.

Возвращаем кортеж из двух элементов слова и пустую строку, указывая, что остальная часть слова не используется для генерации рифм.

Данная функция также будет тестироваться с помощью `test_stemmer()`. Протестировав ее в отдельности и изучив все ожидаемые значения, я могу заняться *рефакторингом* своего кода. На мой взгляд, функция `stemmer()` сродни «черному ящику». Происходящее в ней не имеет отношения к вызывающему ее коду. Пока функция проходит тесты, она работает «корректно» (в определенном смысле).

С небольшими функциями и *тестами* к ним вы можете легко улучшать свои программы. Сначала напишите код и отформатируйте его для удобства восприятия. Затем попробуйте его улучшить или

оптимизировать, не забывая при этом проводить тесты, чтобы убедиться в его работоспособности.

14.4. Прокачиваем навыки

- Добавьте опцию `--output` для записи рифм в указанный файл. По умолчанию вывод должен записываться в поток `STDOUT`.
- Прочитайте содержимое переданного пользователем файла и сгенерируйте рифмы для всех слов в файле. Вы можете воспользоваться программой из главы 6, чтобы прочитать файл и разбить его содержимое на слова, затем перебрать каждое слово и создать для него отдельный выходной файл с соответствующими рифмами.
- Напишите новую программу для поиска уникальных согласных букв в словаре из английских слов. (Среди примеров к книге вы найдете файл inputs/words.txt.zip, архив со словарем с моего компьютера. Распакуйте архив, чтобы использовать файл inputs/words.txt.) Выводите результат в алфавитном порядке и используйте его для замены согласных в вашей программе.
- Измените код своей программы так, чтобы она выдавала только рифмы, найденные в определенном словаре (например, в файле inputs/words.txt).
- Напишите программу для генерации поросячьей латыни*. В этом шутливом «тайном» языке ведущая согласная буква перемещается из начала слова в конец, предваряется дефисом, и к ней добавляется значение «ay». К примеру, слово cat превращается в at-cay. А если слово начинается с гласной, в конце добавляется -yay. Так слово apple становится apple-yay.
- Напишите программу для генерации спунеризмов**, в которых ведущие согласные буквы соседних слов в словосочетаниях меняются местами, так что вместо crushing blow получается blushing crow.

РЕЗЮМЕ

- Регулярные выражения позволяют обозначить шаблон, по которому будет выполняться поиск значений. Механизм регулярных выражений определяет, есть ли совпадения по шаблону. Это декларативный

---

* https://ru.wikipedia.org/wiki/Поросячья_латынь.

** https://ru.wikipedia.org/wiki/Спунеризм.

подход к программированию, отличный от императивного — ручного поиска совпадений путем самостоятельного написания кода.

— Можно заключить части шаблона в круглые скобки, чтобы «захватить» их в группы, извлекаемые из результатов функций `re.match()` и `re.search()`.

— К представлению списка допустимо добавить охранное выражение, позволяющее избежать обработки определенных элементов итерируемого объекта.

— Функция `filter()` является еще одним вариантом реализации представления списка с охранным выражением. Как и `map()`, это ленивая функция высшего порядка, принимающая функцию для обработки каждого элемента итерируемого объекта. Возвращаются только те элементы, которые функция оценивает как «истинные».

— Интерпретатор Python может оценивать в логическом контексте многие типы данных, включая строки, числа, списки и словари, реализуя концепцию «истинности». Таким образом, вы не ограничены значениями `True`/`False` в конструкциях `if`. Пустая строка `''`, целое число `0`, вещественное число `0.0`, пустой список `[]` и пустой словарь `{}` оцениваются как «ложные», поэтому любое неложное значение перечисленных типов, например непустая строка, список или словарь, а также любое числовое значение, отличное от ноля, будет считаться «истинным».

— Длинные строковые литералы можно разбить на более короткие соседние строки, которые интерпретатор затем объединит в одну длинную строку. Рекомендуется разбивать длинные регулярные выражения на более короткие фрагменты и добавлять комментарии к каждой строке, чтобы документировать предназначение каждой из них.

— Следует писать небольшие функции и тесты и публиковать их в виде модулей. Любой файл с расширением *.py* может быть модулем с возможностью импорта из него функций. Многократное использование небольших проверенных модулей лучше, чем написание длинных программ и копипаст кода.

Глава 15
Кентуккийский монах: вновь регулярные выражения

Ушел рыбалить*

Я провел детство на юге США, где жители склонны опускать финальную букву g в словах, оканчивающихся на ing, например cookin' вместо cooking. Там также любят говорить y'all вместо you all, используют данную конструкцию в качестве местоимения от второго лица во множественном числе, что имеет смысл, поскольку в английском языке отсутствует соответствующее отдельное слово. В этой главе мы напишем программу *friar.py*, которая на входе принимает позиционный аргумент и преобразовывает текст, заменяя финальную букву g в двусложных словах с окончанием ing апострофом ('), а также местоимение you на y'all. Разумеется, мы не можем знать, имеем ли дело с местоимением от первого или от второго лица. В любом случае перед нами забавная и интересная задача.

На рис. 15.1 показана диаграмма, отражающая входные и выходные данные. Программа, запущенная без аргументов или с флагом `-h` или `--help`, должна выводить следующие инструкции:

---

* Так говорят в Волгоградской области. *Прим. перев.*

```
$ ./friar.py -h
usage: friar.py [-h] text

Southern fry text

positional arguments:
  text        Input text or file

optional arguments:
-h, --help  show this help message and exit
```

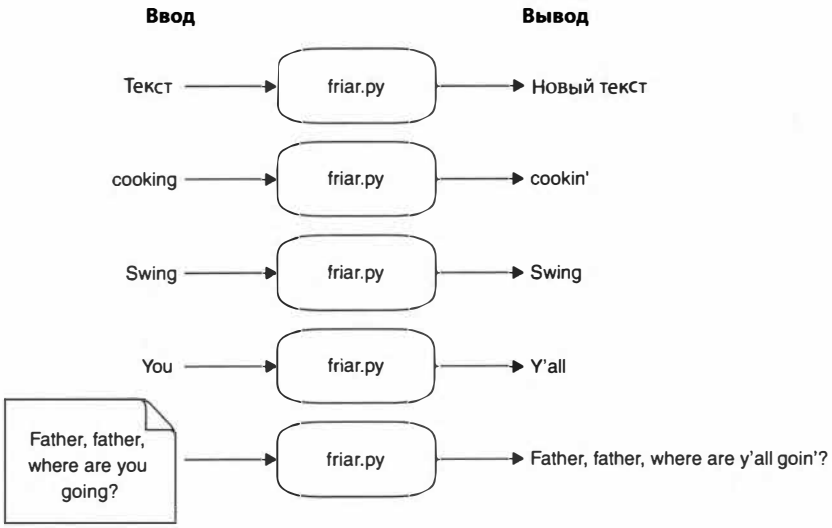


Рис. 15.1. Программа изменяет переданный текст, чтобы придать ему южноамериканский оттенок

Мы будем менять только *двусложные* слова с окончанием ing, поэтому слово cooking станет cookin', а swing останется без изменений. Чтобы определить подходящие под замену слова, нужно проверить части слова перед окончанием ing: содержится ли там гласная, в том числе и буква y*. Мы можем разбить слово cooking на cook и ing, а поскольку в части cook есть гласная буква o, мы способны опустить последнюю букву g:

* В английском языке буква Y может отвечать как за гласный, так и согласный звук, в зависимости от того, в каком месте слова она находится. Если буква Y является первой буквой в слоге, в котором есть другие буквы, кроме самой Y, — значит, это согласная. В любом другом месте слова — гласная. *Прим. перев.*

```
$ ./friar.py Cooking
Cookin'
```

Удалив окончание ing из слова swing, мы получим часть sw, в которой нет гласной буквы, поэтому мы не меняем это слово:

```
$ ./friar.py swing
swing
```

При замене you на y'all учитывайте регистр первой буквы. Так слово You должно превращаться в Y'all:

```
$ ./friar.py you
y'all
$ ./friar.py You
Y'all
```

Как и в предыдущих главах, в программе *friar.py* в качестве входных данных может указываться имя файла, и в этом случае вы должны прочитать содержимое файла для изменения. Для прохождения тестов необходимо соблюсти строковую структуру входных данных, поэтому рекомендуется считывать содержимое файла построчно. При обработке показанных ниже входных данных:

```
$ head -2 inputs/banner.txt
O! Say, can you see, by the dawn's early light,
What so proudly we hailed at the twilight's last gleaming -
```

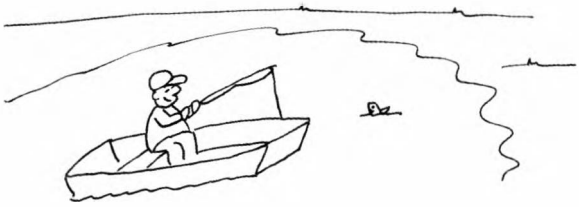
вывод должен иметь соответствующие переносы строк:

```
$ ./friar.py inputs/banner.txt | head -2
O! Say, can y'all see, by the dawn's early light,
What so proudly we hailed at the twilight's last gleamin' -
```

Мне интересно преобразовывать тексты таким способом, но, возможно, это просто чудачество:

```
$ ./friar.py inputs/raven.txt
Presently my soul grew stronger; hesitatin' then no longer,
"Sir," said I, "or Madam, truly your forgiveness I implore;
But the fact is I was nappin', and so gently y'all came rappin',
And so faintly y'all came tappin', tappin' at my chamber door,
```

```
That I scarce was sure I heard y'all" - here I opened wide the
door: - Darkness there and nothin' more.
```



Освоив данную главу, вы научитесь:

— новым приемам работы с регулярными выражениями;
— применять функции `re.match()` и `re.search()` для поиска совпадений в начале строк и в любых позициях строк соответственно;
— использовать в регулярном выражении символ `$` для поиска совпадений в конце строки;
— разрезать строки с помощью метода `re.split()`;
— вручную искать двусложные слова с окончанием ing и местоимения you.

15.1. Создание файла friar.py

Я вновь рекомендую вам создать файл *friar.py* с помощью программы *new.py* или скопировать содержимое документа *template/template.py* в файл *15_friar/friar.py*. Начните с упрощенной версии программы, возвращающей ввод из командной строки:

```
$ ./friar.py cooking
cooking
```

Или из файла:

```
$ ./friar.py inputs/blake.txt
Father, father, where are you going?
 Oh do not walk so fast!
Speak, father, speak to your little boy,
 Or else I shall be lost.
```

Нам нужно обрабатывать ввод построчно, а затем пословно. Можно воспользоваться методом `str.splitlines()`, чтобы извлечь строки

из входных данных, а затем методом `str.split()` для разбиения строки по пробелам на словоподобные элементы. Вот такой код:

```
for line in args.text.splitlines():
    print(line.split())
```

приводит к следующему выводу:

```
$ ./friar.py tests/blake.txt
['Father,', 'father,', 'where', 'are', 'you', 'going?']
['Oh', 'do', 'not', 'walk', 'so', 'fast!']
['Speak,', 'father,', 'speak', 'to', 'your', 'little', 'boy,']
['Or', 'else', 'I', 'shall', 'be', 'lost.']
```

Как видно из результата, данный прием не справляется с некоторыми словесными элементами, поскольку примыкающие знаки препинания также попадают в вывод, например в словах father и going? Метод разбиения текста по пробелам неэффективен, поэтому я научу вас разбивать текст *с помощью регулярных выражений*.

15.1.1. Разбиение текста с помощью регулярных выражений

Чтобы использовать регулярные выражения, необходимо импортировать модуль `re`, как мы поступали в главе 14:

```
>>> import re
```

В демонстрационных целях я присвою переменной `text` первую строку текста:

```
>>> text = 'Father, father, where are you going?'
```

Метод `str.split()` по умолчанию разбивает текст по пробелам. Обратите внимание, что символы, используемые для разбиения текста, отсутствуют в выводе, поэтому результат не содержит пробелов:

```
>>> text.split()
['Father,', 'father,', 'where', 'are', 'you', 'going?']
```

Методу `str.split()` можно передать опциональную строку, которая будет применяться для разбиения текста. Если указать запятую,

мы получим в результате три строки вместо шести. Отмечу, что в результирующем списке нет запятых, так как это аргумент метода `str.split()`:

```
>>> text.split(',')
['Father', ' father', ' where are you going?']
```

Модуль `re` содержит метод `re.split()`, работающий аналогичным образом. Я рекомендую вам прочитать справку к данному методу (`help(re.split)`), чтобы узнать, насколько он мощный и гибкий. Как и в случае с методом `re.match()` из главы 14, при выполнении метода `re.split()` нужно указать шаблон, по которому следует выполнить разбиение, и строку, которую требуется разбить в соответствии с шаблоном. Остальные аргументы опциональны. Мы можем передать методу `re.split()` запятую в качестве шаблона, чтобы получить тот же вывод, что и в случае с функцией `str.split()`. Как и в предыдущем примере, запятые в выводе будут отсутствовать:

```
>>> re.split(',', text)
['Father', ' father', ' where are you going?']
```

15.1.2. Сокращенная запись классов

Нам нужны элементы, похожие на «слова» в том смысле, что они состоят из символов, которые обычно встречаются в словах. Символы, не встречающиеся в словах (например, знаки препинания), следует использовать для разбиения строк. Мы уже рассматривали прием создания *символьного класса* путем заключения букв в квадратные скобки, например `'[aeiou]'` для гласных букв. А теперь давайте создадим символьный класс, в котором перечислим все небуквенные символы. Код будет выглядеть примерно так:

```
>>> import string
>>> ''.join([c for c in string.printable if c
             not in string.ascii_letters])
'0123456789!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

Не вижу необходимости в таком длинном коде, ведь практически любая реализация механизмов регулярных выражений допускает сокращенную запись символьных классов. В табл. 15.1 перечислены некоторые из наиболее распространенных сокращений классов и их развернутые версии.

Таблица 15.1. Сокращенные и полные записи классов регулярных выражений

Символьный класс	Сокращенная запись	Другие варианты записи
Цифры	`\d`	`[0123456789]`, `[0-9]`
Пробельные символы	`\s`	`[\t\n\r\x0b\x0c]`, то же, что и `string.whitespace`
Цифробуквенные символы, включая дефис и нижнее подчеркивание	`\w`	`[a-zA-Z0-9_-]`

Примечание. Существует базовый синтаксис регулярных выражений, распознаваемый многим ПО, от консольных инструментов Unix типа `awk` до поддержки языками программирования, такими, как Perl, Python и Java. Некоторое ПО расширяет поддержку регулярных выражений дополнениями, неподдерживаемыми другими инструментами. К примеру, была создана библиотека регулярных выражений в стиле Perl, реализующая много новых концепций, и в итоге превратившаяся в диалект PCRE (Perl Compatible Regular Expressions — Perl-совместимые регулярные выражения). Не существует ПО, поддерживающего все типы и разновидности регулярных выражений, но за годы работы с ними я редко сталкивался с проблемами.

Сокращение `\d` означает любую *цифру* и эквивалентно записи `'[0123456789]'`. Я могу использовать метод `re.search()` для поиска любой цифры в любой позиции строки. В следующем примере в строке `'abc123!'` найден символ `1`, поскольку это первая цифра в строке (рис. 15.2):

```
>>> re.search('\d', 'abc123!')
<re.Match object; span=(3, 4), match='1'>
```

Рис. 15.2. Сокращение `\d` соответствует любой отдельной цифре

С помощью длинной записи метод выдает тот же результат (рис. 15.3):

```
>>> re.search('[0123456789]', 'abc123!')
<re.Match object; span=(3, 4), match='1'>
```

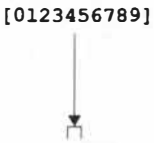


Рис. 15.3. Мы можем создать символьный класс, перечисляющий все цифры

Данный код аналогичен по результату регулярному выражению с диапазоном символов `'[0-9]'` (рис. 15.4):

```
>>> re.search('[0-9]', 'abc123!')
<re.Match object; span=(3, 4), match='1'>
```

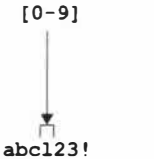


Рис. 15.4. Символьные классы могут содержать диапазон значений, например 0–9

Чтобы найти *одну и более цифр в строке*, укажите символ + (рис. 15.5):

```
>>> re.search('\d+', 'abc123!')
<re.Match object; span=(3, 6), match='123'>
```

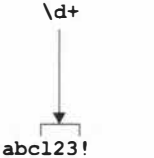


Рис. 15.5. Символ «плюс» допускает совпадение с одним или несколькими символами

Сокращение `\w` соответствует «цифробуквенным символам, включая дефис и нижнее подчеркивание». Оно включает в себя все арабские цифры, буквы английского алфавита, дефис (`'-'`) и знак подчеркивания (`'_'`). Первое совпадение в строке — буква a (рис. 15.6):

```
>>> re.search('\w', 'abc123!')
<re.Match object; span=(0, 1), match='a'>
```

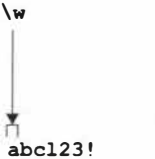


Рис. 15.6. Цифробуквенные символы, включая дефис и нижнее подчеркивание, ищутся сокращением \w

Если добавить символ + (рис. 15.7), вы обозначите соответствие одному или более символам в строке, включая `abc123`, но исключая восклицательный знак (`!`):

```
>>> re.search('\w+', 'abc123!')
<re.Match object; span=(0, 6), match='abc123'>
```

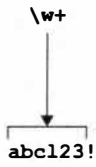


Рис. 15.7. Добавьте «плюс» для соответствия одному или более цифробуквенным символам, включая дефис и нижнее подчеркивание

15.1.3. Исключение символов из выборки

Можно «исключить» символы из шаблона, указав знак вставки (`^`) *непосредственно* в символьном классе, как показано на рис. 15.8. Согласно шаблону `'[^0-9]+'` будут найдены один или более любых символов, не являющихся цифрами, то есть `'abc'`:

```
>>> re.search('[^0-9]+', 'abc123!')
<re.Match object; span=(0, 3), match='abc'>
```

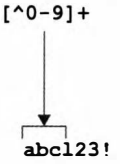


Рис. 15.8. Символ вставки в символьном классе исключает указанные символы. Продемонстрированное на рисунке регулярное выражение исключает из поиска цифры

В сокращении шаблон `[^0-9]+` также можно записать как `\D+` (рис. 15.9). Обратите внимание, что используется прописная буква `D`, а не строчная `d`:

```
>>> re.search('\D+', 'abc123!')
<re.Match object; span=(0, 3), match='abc'>
```

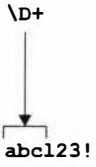


Рис. 15.9. Сокращение `\D+` исключает из поиска одну или более цифр

Сокращением для исключения «цифробуквенных символов, включая дефис и нижнее подчеркивание», служит обозначение \W. Показанный ниже код исключает любые символы, кроме восклицательного знака (рис. 15.10):

```
>>> re.search('\W', 'abc123!')
<re.Match object; span=(6, 7), match='!'>
```

Рис. 15.10. Шаблон \W ищет любые символы, кроме букв, цифр, символа нижнего подчеркивания и дефиса

В таблице 15.2 перечислены сокращенные и полные записи классов исключений.

Таблица 15.2. Сокращенные и полные записи классов регулярных выражений для поиска с исключением

Символьный класс	Сокращенная запись	Другие варианты записи
Исключая цифры	\D	[^0123456789], [^0-9]
Исключая пробельные символы	\S	[^ \t\n\r\x0b\x0c]
Исключая цифробуквенные символы, дефис и нижнее подчеркивание	\W	[^a-zA-Z0-9_-]

15.1.4. Использование метода re.split() с регулярным выражением

Мы можем передать методу `re.split()` аргумент \W:

```
>>> re.split('\W', 'abc123!')
['abc123', '']
```

Примечание. Анализатор кода Pylint обнаружит в регулярном выражении запись `'\W'` и вернет сообщение: «Аномальный обратный слеш в строке: `'\W'`. В строковой константе может отсутствовать префикс `r`». Мы вольны прибегнуть к префиксу `r` для создания «необработанной» строки, в которой Python не будет интерпретировать \W по аналогии с символами перевода строки (\n) или возврата каретки (\r). С данного момента я буду использовать синтаксис с префиксом `r` для создания необработанных строк.

Тут возникает проблема, поскольку метод `re.split()` *опускает строки, соответствующие шаблону.* В нашем примере теряется восклицательный знак! Внимательно прочитайте справку (`help(re.split)`), в ней можно найти решение:

> ***Если в шаблоне есть круглые скобки**, то текст из всех групп захвата в скобках возвращается в виде результирующего списка.*

В главе 14 мы использовали скобки групп захвата, чтобы механизм регулярных выражений «запоминал» определенные шаблоны, например согласную (или согласные), гласную и остальную часть слова. При обнаружении совпадений по шаблону мы применяли метод `match.groups()` для извлечения найденных и совпадающих строк. В данной главе мы окружим круглыми скобками шаблон для метода `re.split()`, поэтому строки, соответствующие шаблону, также будут возвращены:

```
>>> re.split(r' (\W)', 'abc123!')
['abc123', '!', '']
```

Применив код к демонстрационному тексту, в результате мы получаем список строк как соответствующих, так и не соответствующих шаблону регулярного выражения:

```
>>> re.split(r' (\W)', text)
['Father', ',', '', ' ', 'father', ',', '', ' ', 'where', ' ',
'are', ' ', 'you', ' ', 'going', '?', '']
```

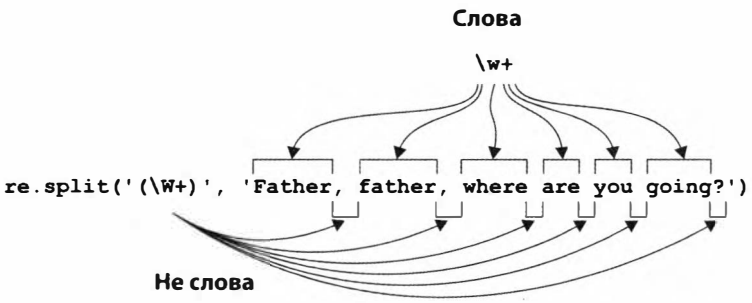


Рис. 15.11. Метод `re.split()` может использовать группы захвата для возврата как соответствующих регулярному выражению фрагментов строк, так и несоответствующих

Я могу сгруппировать все символы, не относящиеся к цифрам, буквам, дефису и нижнему подчеркиванию, добавив символ + к регулярному выражению (рис. 15.11):

```
>>> re.split(r' (\W+)', text)
['Father', ', ', 'father', ', ', 'where', ' ', 'are', ' ', 'you',
' ', 'going', '?', '']
```

Здорово! Теперь мы способны обрабатывать как *сами* слова, так и символы между ними.

15.1.5. Создание функции fry()

Наш следующий шаг — функция, определяющая, нужно ли модифицировать *одно слово* и каким образом это делать. Соответственно, вместо обработки всего текста одним махом, подумаем о том, как обрабатывать слова по очереди. Мы назовем нашу функцию `fry()`.

Чтобы было проще разобраться в ее устройстве, давайте напишем функцию `test_fry()` и заглушку для фактической функции `fry()`, содержащую лишь оператор `pass`, то есть не имеющую рабочего кода. Начнем со следующего фрагмента, который нужно добавить в нашу программу:

```
def fry(word):
    pass

def test_fry():
    assert fry('you') == "y'all"
    assert fry('You') == "Y'all"
    assert fry('fishing') == "fishin'"
    assert fry('Aching') == "Achin'"
    assert fry('swing') == "swing"
```

`pass` — оператор-заглушка, равноценный отсутствию операции. Он используется, когда синтаксически необходимо наличие какого-то оператора или выражения, но не требуется выполнение каких-либо действий. Можно сказать No OPeration — NOP — инструкция, которая предписывает ничего не делать. Мы определяем функцию `fry()` как заглушку, чтобы написать для нее тест.

Слово you становится y'all.

Соблюдаем регистр слов, начинающихся с прописной буквы.

Двусложное слово с окончанием ing, которое следует изменить, подставив апостроф вместо завершающей буквы g.

Двусложное слово с окончанием ing, начинающееся с гласной. Его также следует изменить.

Односложное слово с окончанием ing, которое не следует изменять.

Функция `test_fry()` будет передавать слова, предназначенные и непредназначенные для изменения. Проверка всех слов невозможна, поэтому полагаемся на выборочную проверку в основных ситуациях.

Выполните команду `pytest friar.py`. Как и ожидалось, тест провален:

```
============================ FAILURES ============================
_____________________________ test_fry _____________________________

    def test_fry():
>       assert fry('you') == "y'all"
E       assert None == "y'all"
E        +  where None = fry('you')

friar.py:47: AssertionError
=================== 1 failed in 0.08 seconds ===================
```

Первый тест провален.

Функция `fry('you')` выдает результат `None` вместо ожидаемого `y'all`.

Давайте изменим код функции нашего `fry()` для обработки данной строки:

```
def fry(word):
    if word == 'you':
        return "y'all"
```

Затем снова выполним тестирование:

```
============================ FAILURES ============================
_____________________________ test_fry _____________________________

    def test_fry():
        assert fry('you') == "y'all"
>       assert fry('You') == "Y'all"
E       assert None == "Y'all"
E        +  where None = fry('You')

friar.py:49: AssertionError
=================== 1 failed in 0.16 seconds ===================
```

Теперь первый тест успешно завершен.

Второй тест провален, так как слово `You` написано с прописной буквы.

Функция возвращает `None` вместо ожидаемого слова `Y'all`.

Давайте решим данную проблему:

```
def fry(word):
    if word == 'you':
        return "y'all"
    elif word == 'You':
        return "Y'all"
```

Если вы протестируете программу сейчас, она пройдет первые два теста, но я недоволен таким решением. Слишком много кода дублируется в программе. Существует ли более элегантный способ сопоставлять слова you и You и возвращать корректный результат с прописной или строчной буквы? Да, конечно.

```
def fry(word):
    if word.lower() == 'you':
        return word[0] + "'all"
```

Более того, мы можем написать регулярное выражение! Между словами you и You есть только одно различие — буквы y и Y. Мы способны его учесть, используя символьный класс (рис. 15.12). Ниже показан код для обработки строчной версии слова:

```
>>> re.match('[yY]ou', 'you')
<re.Match object; span=(0, 3), match='you'>
```

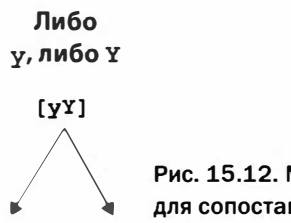


Рис. 15.12. Мы можем использовать символьный класс для сопоставления строчной и прописной буквы Y

Он также будет обрабатывать слово с прописной буквы (рис. 15.13):

```
>>> re.match('[yY]ou', 'You')
<re.Match object; span=(0, 3), match='You'>
```

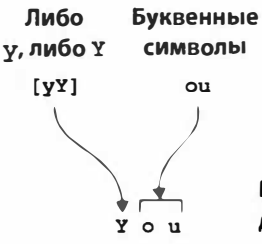


Рис. 15.13. Регулярное выражение для сопоставления слов you и You

Теперь мы в силах повторно использовать ведущую букву (либо y, либо Y) в возвращаемом значении. Можем *захватить* ее, поместив в круглые скобки. Измените код функции `fry()` для реализации

данного приема, и успешно пройдите первые два теста, прежде чем двигаться дальше:

```
>>> match = re.match('([yY])ou', 'You')
>>> match.group(1) + "'all"
"Y'all"
```

Теперь займемся обработкой такого слова, как fishing:

```
========================= FAILURES ==========================
_____________________________ test_fry _____________________________

    def test_fry():
        assert fry('you') == "y'all"
        assert fry('You') == "Y'all"
>       assert fry('fishing') == "fishin'"
E       assert None == "fishin'"
E        +  where None = fry('fishing')

friar.py:52: AssertionError
-------------------- 1 failed in 0.10 seconds ===================
```

Третий тест провален.

Функция `fry('fishing')` выдает результат `None` вместо ожидаемого слова `'fishin'`.

Как определить, что слово оканчивается на ing? С помощью метода `str.endswith()`:

```
>>> 'fishing'.endswith('ing')
True
```

В конце регулярного выражения для поиска окончания ing в строке содержится символ `$` («знак доллара»). Он указывает на *привязку* к концу строки (рис. 15.14):

```
>>> re.search('ing$', 'fishing')
<re.Match object; span=(4, 7), match='ing'>
```

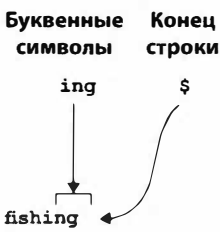


Рис. 15.14. Символ доллара указывает на конец слова

Как показано на рис. 15.15, мы можем нарезать строку, чтобы извлечь все символы от последнего с индексом −1, а затем добавить апостроф.

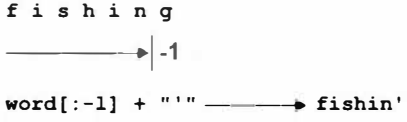


Рис. 15.15. Нарежьте строку, чтобы получить все буквы от последней и добавить апостроф

Включите показанный ниже код в функцию `fry()` и проверьте, сколько тестов она пройдет успешно:

```
if word.endswith('ing'):
    return word[:-1] + "'"
```

Как вариант, в регулярном выражении можно использовать группу захвата для извлечения первой части слова (рис. 15.16):

```
>>> match = re.search('(.+)ing$', 'fishing')
>>> match.group(1) + "in'"
"fishin'"
```

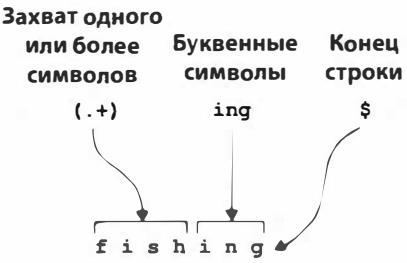


Рис. 15.16. Использование группы захвата для доступа к соответствующей строке

Функция должна выдать следующие результаты:

```
=========================== FAILURES ===========================
_____________________________ test_fry _____________________________

    def test_fry():
        assert fry('you') == "y'all"
        assert fry('You') == "Y'all"
        assert fry('fishing') == "fishin'"
        assert fry('Aching') == "Achin'"
>       assert fry('swing') == "swing"
```

Данный тест провален.

```
E       assert "swin'" == 'swing'
E         - swin'
E         ?      ^
E         + swing
E         ?      ^

friar.py:59: AssertionError
==================== 1 failed in 0.10 seconds ====================
```

Функция `fry('swing')` выдает результат `swin` вместо ожидаемого слова `swing`.

Иногда результаты тестов указывают на точную позицию проблемы. Тут отмечено, что вместо буквы g используется апостроф (').

Нам нужен способ идентификации двусложных слов. Как упоминалось ранее, попробуем воспользоваться эвристическим подходом и осуществим поиск гласной `'[aeiouy]'` в части слова *перед* окончанием ing (рис. 15.17). Поможет показанное ниже регулярное выражение:

Нам известно совпадающее значение, поэтому используем метод `match.group(1)` для обозначения первой группы захвата, которая может быть всем, чем угодно, перед окончанием ing. В рабочей версии программы нужно проверить, что в результате не выдается значение `None`, иначе будет выброшено исключение.

Группа захвата (.+) соответствует одному или более символам, за которыми следуют буквы ing. В результате метод `re.search()` выдаст либо объект `re.Match` (если есть совпадение), либо `None` (в противном случае).

```
>>> match = re.search('(.+)ing$', 'fishing')
>>> first = match.group(1)
>>> re.search('[aeiouy]', first)
<re.Match object; span=(1, 2), match='i'>
```

Используем метод `re.search()` для поиска гласной в первой части строки.

Поскольку метод `re.search()` вернул объект `re.Match`, значит, в первой части слова есть гласная и оно состоит из двух слогов.

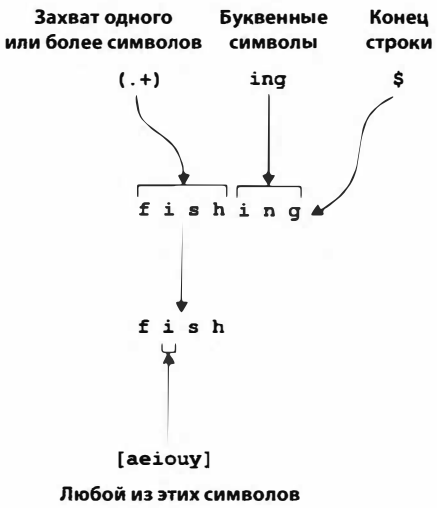


Рис. 15.17. Возможный способ определения двусложных слов с окончанием ing заключается в поиске гласной в первой части слова

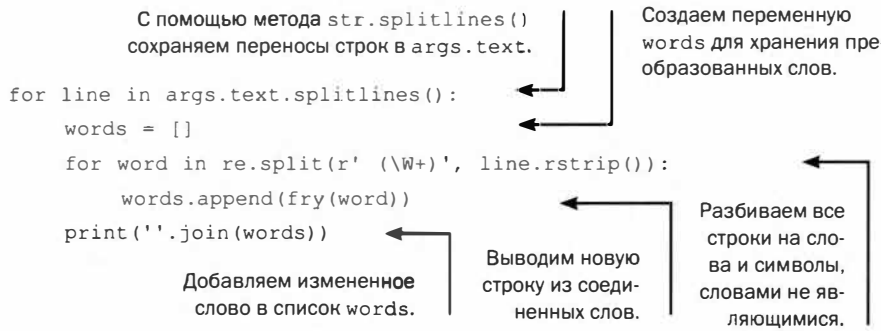
Если поиск совпадений в слове увенчался успехом, возвращаем слово с заменой последней буквы g на апостроф. В противном случае слово выводится без изменений. Я предлагаю вам доработать программу так, чтобы пройти все тесты с помощью функции `test_fry()`.

15.1.6. Применение функции fry()

Теперь ваша программа должна уметь следующее.

1. Считывать входные данные из командной строки или из файла.
2. Считывать входные данные построчно.
3. Разбивать все строки на слова и символы, словами не являющиеся.
4. Обрабатывать каждое слово с помощью функции `fry()`.

Давайте применим функцию `fry()` ко всем словоподобным единицам. Думаю, это знакомый вам прием. Вы можете использовать цикл `for`:



Программа с таким (или подобным) кодом должна успешно проходить все тесты. Написав рабочую версию, попробуйте заменить цикл `for` представлением списка, а затем и функцией `map()`.

Прекрасно! Пора напрячься и написать программу.

15.2. Решение

Вспомнился мне анекдот, в котором друг Робин Гуда, брат Тук, был схвачен шерифом Ноттингема. Монаха приговорили к варке в масле, на что он ответил: «Меня нельзя варить, я же монах!»*

```
#!/usr/bin/env python3
"""Кентуккийский монах"""

import argparse
import os
import re

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Southern fry text',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='text', help='Input
                        text or file')

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text).read()

    return args

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()

    for line in args.text.splitlines():
        print(''.join(map(fry, re.split(r' (\W+)', line.rstrip()))))
```

Если в качестве аргумента передано имя файла, помещаем в `text` его содержимое.

Сопоставляем фрагменты текста, разделенные регулярным выражением, с помощью функции `fry()`, возвращающей слова, измененные по мере необходимости. С помощью метода `str.join()` превращаем полученный список обратно в строку для вывода.

Получаем аргументы командной строки. К этому моменту значение `text` будет либо текстом из командной строки, либо содержимым файла.

С помощью метода `str.splitlines()` сохраняем переносы во входном тексте.

* Имя функции `fry()` переводится как жарка. *Прим. перев.*

Поиск значения ing в конце слова. Используем группу захвата для хранения части строки перед окончанием ing.

Определяем функцию `fry()`, предназначенную для обработки каждого слова.

```
# -----------------------------------------------
def fry(word):
    """Убираем букву 'g' из окончаний '-ing', меняем слово 'you'
       на 'y'all'"""

    ing_word = re.search('(.+)ing$', word)
    you = re.match('([Yy])ou$', word)

    if ing_word:
        prefix = ing_word.group(1)
        if re.search('[aeiouy]', prefix, re.IGNORECASE):
            return prefix + "in'"
    elif you:
        return you.group(1) + "'all"

    return word
```

Проверяем, обнаружено ли совпадение с шаблоном ing.

Получаем префикс (фрагмент перед ing), который находится в группе 1.

Проверяем, обнаружено ли совпадение с шаблоном you.

Добавляем значение in к префиксу и возвращаем его вызывающей стороне.

Возвращаем захваченный первый символ и «остальное».

Ищем в начале слова you и You. Учитываем регистр [yY] в группе.

В противном случае возвращаем слово без изменений.

Тесты для функции `fry()`.

```
# -----------------------------------------------
def test_fry():
    """Test fry"""

    assert fry('you') == "y'all"
    assert fry('You') == "Y'all"
    assert fry('fishing') == "fishin'"
    assert fry('Aching') == "Achin'"
    assert fry('swing') == "swing"

# -----------------------------------------------
if __name__ == '__main__':
    main()
```

Выполняем в префиксе поиск гласной (в том числе и буквы y, которая в английском языке может быть как гласной, так и согласной) без учета регистра. Если совпадений не обнаружено, возвращаем значение `None`, что в данном логическом контексте оценивается как `False`. Если совпадение обнаружено, значение, отличное от `None`, оценивается как `True`.

15.3. Обсуждение

В коде функции `get_arg s()` нет ничего нового, так что перейдем к разбиению текста на строки. В предыдущих упражнениях использовался метод считывания содержимого входного файла в значение `args.text`. Если в качестве входных данных выступает файл, каждая строка текста из него разделяется символами перевода строки. Я использую цикл `for` для обработки каждой строки исходного текста, возвращаемого методом `str.splitlines()`, чтобы сохранить в выводе переносы. Кроме того, рекомендуется начать со второго цикла `for` для обработки каждой словоподобной единицы, возвращаемой методом `re.split()`:

```
for line in args.text.splitlines():
    words = []
    for word in re.split(r' (\W+)', line.rstrip()):
        words.append(fry(word))
    print(''.join(words))
```

Пять строк кода можно сократить до двух, если заменить второй цикл `for` на представление списка:

```
for line in args.text.splitlines():
    print(''.join([fry(w) for w in re.split(r' (\W+)', line.
rstrip())]))
```

Укоротим код еще немного, воспользовавшись функцией `map()`:

```
for line in args.text.splitlines():
    print(''.join(map(fry, re.split(r' (\W+)', line.rstrip()))))
```

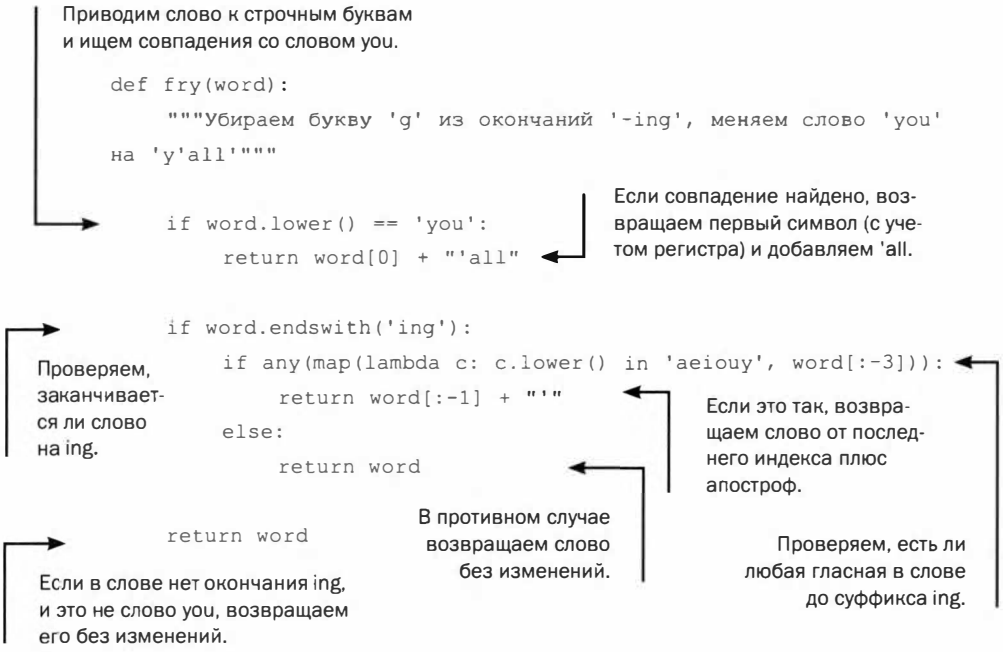
Чтобы улучшить читаемость кода, можно также применить метод `re.compile()` для компиляции регулярного выражения. При использовании метода `re.split()` в цикле `for` регулярное выражение необходимо компилировать заново при каждой итерации. А с методом `re.compile()` компиляция регулярного выражения выполняется однократно, поэтому код работает быстрее (хотя бы немного). Что еще более важно, такой код легче читать. Преимущества особенно заметны при использовании сложных регулярных выражений:

```
splitter = re.compile(r' (\W+)')
for line in args.text.splitlines():
    print(''.join(map(fry, splitter.split(line.rstrip()))))
```

15.3.1. Функция fry() без регулярных выражений

Разумеется, вам необязательно было писать именно функцию `fry()`. Так или иначе, вы разработали собственное решение и, будем надеяться, создали тесты для него!

Показанная ниже версия кода реализует некоторые концепции, о которых я говорил ранее в этой главе. Тем не менее в данной версии не используются регулярные выражения:



Давайте выделим минутку, чтобы по достоинству оценить потенциал моей любимой функции `any()`. В показанном коде используется функция `map()`, проверяющая, есть ли какая-нибудь из гласных в части слова перед окончанием ing:

```
>>> word = "cooking"
>>> list(map(lambda c: (c, c.lower() in 'aeiouy'), word[:-3]))
[('c', False), ('o', True), ('o', True), ('k', False)]
```

Первый символ слова cooking — буква c, и ее нет в строке с гласными. Следующие два символа («o») есть среди гласных, а буквы k — нет.

Давайте сократим задачу до логических значений `True` / `False`:

```
>>> list(map(lambda c: c.lower() in 'aeiouy', word[:-3]))
[False, True, True, False]
```

Мы можем использовать функцию `any()`, чтобы проверить, истинно ли какое-нибудь из значений:

```
>>> any([False, True, True, False])
True
```

Это то же самое, что использовать оператор `or`:

```
>>> False or True or True or False
True
```

Функция `all()` возвращает значение `True`, только если истинны *все* значения:

```
>>> all([False, True, True, False])
False
```

Это то же самое, что использовать оператор `and`:

```
>>> False and True and True and False
False
```

Если выражение, что одна из гласных присутствует в первой части слова, истинно, мы определяем, что имеем дело с двусложным словом (вероятно), а значит, можем вернуть его с заменой последней буквы g на апостроф. В противном случае мы возвращаем слово без изменений:

```
if any(map(lambda c: c.lower() in 'aeiouy', word[:-3])):
    return word[:-1] + "'"
else:
    return word
```

Данный подход прекрасен, но довольно трудоемок, так как необходимо написать довольно много кода по поиску совпадений.

15.3.2. Функция fry() с регулярными выражениями

Вернемся к версии функции `fry()` с регулярными выражениями:

С помощью метода `re.search()` ищем в префиксе любые гласные (в т. ч. и «y») без учета регистра. Напоминаю, что метод `re.match()` осуществляет поиск с начала слова, а это не то, что нам нужно.

Шаблон `'(.+)ing$'` соответствует одному или более символам, за которыми следует окончание ing. Знак доллара позволяет выполнить поиск в конце строки, то есть ищет строку с окончанием ing. При этом строка не может быть представлена просто буквами ing, так как перед ними должны быть еще символы. Круглые скобки формируют группу захвата части слова перед окончанием ing.

Метод `re.match()` приступает к поиску совпадений с начала заданного слова и ищет букву y, как строчную, так и прописную, за которой следуют символы ou, а также символ конца строки ($).

```
def fry(word):
    """Убираем букву 'g' из окончаний '-ing', меняем
       слово 'you' на 'y'all'"""

    ing_word = re.search('(.+)ing$', word)
    you = re.match('([Yy])ou$', word)

    if ing_word:
        prefix = ing_word.group(1)
        if re.search('[aeiouy]', prefix, re.IGNORECASE):
                return prefix + "in'"
    elif you:
        return you.group(1) + "'all"

    return word
```

Возвращаем префикс и строку in, отсекая финальную букву g.

Если метод `re.match()` не обнаруживает совпадений по шаблону you, возвращается значение `None`. Если значение отличается от `None`, получается, что совпадение обнаружено, а you — это объект `re.Match`.

Круглые скобки используются для захвата первого символа с целью учета регистра. То есть, если обнаружено слово You, нужно вернуть результат Y'all. Здесь мы возвращаем первую группу захвата вместе со строкой 'all.

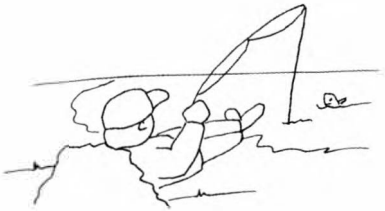
Префикс — это фрагмент перед окончанием ing, заключенный в круглые скобки. Поскольку мы имеем дело с первой парой скобок, мы можем работать с ней с помощью кода `ing_word.group(1)`.

Если `ing_word` имеет значение `None`, совпадение не обнаружено. Если значение отличается от `None` (т. е. «истинно»), мы получаем объект `re.Match` для дальнейшей обработки.

Если слово не соответствует ни шаблону двусложного слова с окончанием ing, ни слову you, возвращаем его без изменений.

Я использую регулярные выражения без малого лет 20, поэтому данная версия функции мне гораздо ближе, чем предыдущая. У вас же

может быть все наоборот. Если вы только осваиваете регулярные выражения, поверьте мне, они стоят того, чтобы их изучить. Без них я абсолютно точно не смог бы выполнять бóльшую часть своей работы.



15.4. Прокачиваем навыки

— Попробуйте научить программу заменять слово your на y'all's. К примеру, строка "Where are your britches?" может звучать как "Where are y'all's britches?"
— Замените слова getting ready или preparing на fixin'. Например, фразу "I'm getting ready to eat" на "I'm fixin' to eat". Потом измените строку think на reckon: пусть будет фраза "I reckon this is funny" вместо "I think this is funny". Еще можно заменить слово thinking на reckoning, а затем на reckonin'. В таком случае вам нужно выполнить либо две операции по изменению, либо искать и менять think и thinking одним махом.
— Напишите версию программы для другого регионального диалекта. Какое-то время я жил в Бостоне, и мне очень нравилось все время говорить wicked вместо very, вот так: "It's wicked cold out!"

РЕЗЮМЕ

— Регулярные выражения используются для поиска по шаблону совпадений в тексте. Шаблоны могут быть довольно сложными, например группировка специальных символов между группами цифробуквенных символов.
— Модуль `re` содержит очень удобные методы, такие как `re.match()` для поиска по шаблону совпадений в начале текста, `re.search()` — для поиска совпадений в любой позиции текста, `re.split()` — для разбиения текста согласно шаблону и `re.compile()` — для компиляции регулярных выражений, чтобы вы могли использовать их многократно.
— Если шаблон для метода `re.split()` содержит круглые скобки, захваченные совпадения будут включены в возвращаемый результат. Это позволяет восстановливать исходную строку со значениями, найденными по шаблону.

Глава 16

Скремблер: перемешивание букв внутри слов

Ваш мзог пяаелесвтрдт сбоой уьвилнеиодте стченаоие аоартанпгпо и пмггномраоро оенисбечпея. Вы чаеитте эотт тксет немрсотя на то, что бвкуы в сволах ппаруеенты. Ваш мзог вмопасирниет совла, помоту что пварея и пнсодяеля бувкы солв оастилсь на свиох мтаесх. На сомам длее ваш мзог севаитычт не кадужю бкуву в своле, а совло цклеиом. Совла с претнемыапуни баквмуи чтсаютия мделненее, но в лоюбм саучле, вы не псвтееелятрае бвукы в пранолиьвм пкоярде, не так ли? Вы птсроо чеаттие!

В данной главе вы напишете программу *scrambler.py*, шифрующую слова в тексте, переданном в качестве аргумента. Шифроваться должны слова с четырьмя и более буквами, причем в середине слов, а первая и последняя буквы оставаться на своих местах. Программа принимает опцию `-s` или `--seed` (целое число с дефолтным значением `None`) для метода `random.seed()`.

А также текст из командной строки:

```
$ ./scrambler.py --seed 1 "foobar bazquux"
faobor buuzaqx
```

Или из файла:

```
$ cat ../inputs/spiders.txt
Don't worry, spiders,
I keep house
casually.
$ ./scrambler.py ../inputs/spiders.txt
D'not wrory, sdireps,
I keep hsuoe
csalluay.
```

На рис. 16.1 показана диаграмма, которая поможет вам продумать структуру кода.

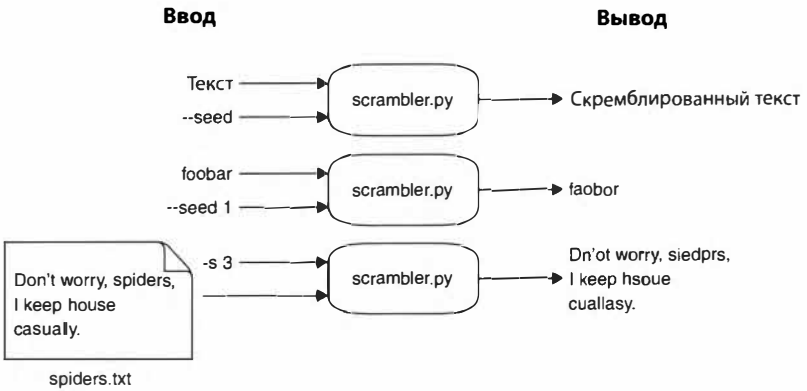


Рис. 16.1. Программа принимает пользовательский текст из командной строки или из файла и перемешивает буквы в словах из четырех и более символов

Прочитав данную главу, вы научитесь:

— использовать регулярные выражения для разбиения текста на слова;
— применять функцию `random.shuffle()` для перемешивания содержимого списка;
— создавать скремблированные версии слов, перемешивая буквы внутри них и оставляя первую и последнюю буквы на своих местах.

16.1. Создание файла scrambler.py

Выполните команду `new.py scrambler.py` для создания программы-скремблера в каталоге *16_scrambler*. Также можно скопировать содержимое документа *template/template.py* в файл *16_scrambler/scrambler.py*. При необходимости обратитесь к предыдущим упражнениям, например к главе 5, чтобы освежить знания по работе с позиционным аргументом, который может быть представлен как текстом, так и текстовым файлом.

При запуске без аргументов или с флагом `-h` или `--help` программа *scrambler.py* должна выводить следующие инструкции:

```
$ ./scrambler.py -h
usage: scrambler.py [-h] [-s seed] text

Scramble the letters of words

positional arguments:
  text                  Input text or file

optional arguments:
-h, --help              show this help message and exit
-s seed, --seed seed    Random seed (default: None)
```

Разобравшись со справочной информацией, измените определение функции `main()` следующим образом:

```
def main():
    args = get_args()
    print(args.text)
```

Затем проверьте, принимает ли программа текст из командной строки:

```
$ ./scrambler.py hello
hello
```

И из текстового файла:

```
$ ./scrambler.py ../inputs/spiders.txt
Don't worry, spiders,
I keep house
casually.
```

16.1.1. Разбиение текста на строки и слова

По аналогии с тем, как мы делали в программе из главы 15, нам нужно сохранить переносы в тексте из входного текстового файла с помощью метода `str.splitlines()`:

```
for line in args.text.splitlines():
    print(line)
```

Ниже показана первая строка хайку из файла *spiders.txt*:

```
>>> line = "Don't worry, spiders,"
```

Необходимо разбить ее на слова. В главе 6 мы использовали метод `str.split()`, однако в таком случае вместе со словами извлекаются знаки препинания — `worry,`, `spiders,`:

```
>>> line.split()
["Don't", 'worry,', 'spiders,']
```

В главе 15 мы применяли функцию `re.split()` с регулярным выражением `(\W+)` для разбиения текста на один или более символов, не являющихся словами. Давайте попробуем применить данный метод здесь:

```
>>> re.split('(\W+)', line)
['Don', "'", 't', ' ', 'worry', ', ', 'spiders', ',', '']
```

Прием не сработал, потому что слово Don't разделилось на три части: `Don`, `'` и `t`.

Вероятно, можно использовать символ `\b` для определения *границ слова*. Обратите внимание, что нужно указать код `r'\b'` перед первой кавычкой, чтобы обозначить, что далее следует «необработанная» строка.

И снова нерабочий прием. Так как согласно символу `\b` апостроф является границей слова, Don't опять разделяется:

```
>>> re.split(r'\b', "Don't worry, spiders,")
['', 'Don', "'", 't', ' ', 'worry', ', ', 'spiders', ',']
```

В процессе поиска в интернете регулярного выражения для правильного разбиения данного слова я нашел показанный ниже шаблон

на форуме Java-программистов. Он отлично отделяет *слова* от *прочего контента*:*

```
>>> re.split("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)", "Don't worry,
            spiders,")
['', "Don't", ' ', 'worry', ', ', 'spiders', ',']
```

Прелесть регулярных выражений в том, что это отдельный диалект, который используется во многих языках программирования, от Perl до Haskell. Давайте рассмотрим шаблон, показанный на рис. 16.2.

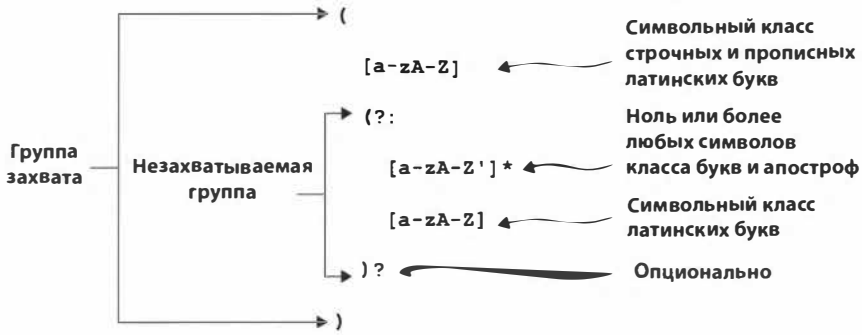


Рис. 16.2. Регулярное выражение, позволяющее извлекать слова с апострофом

16.1.2. Захват, группы без захвата и опциональное извлечение

На рис. 16.2 мы видим, что одни группы могут вкладываться в другие. К примеру, ниже показано регулярное выражение, захватывающее строку foobarbaz целиком, а также подстроку bar:

```
>>> match = re.match('(foo(bar)baz)', 'foobarbaz')
```

Группы захвата нумеруются по позиции их левой скобки. Поскольку первая левая скобка относится к группе, захватывающей буквы начиная с f и заканчивая z, это группа 1:

```
>>> match.group(1)
'foobarbaz'
```

* Подчеркиваю, что при работе над проектами значительную часть времени я трачу на поиск ответов как в купленных книгах, так и в интернете.

Левая скобка начинается непосредственно перед буквой b, и группа завершается буквой r. Это группа 2:

```
>>> match.group(2)
'bar'
```

Мы также можем сделать группу *без захвата,* использовав последовательность символов `?:` после левой скобки. Если указать данную последовательность во второй группе нашего примера, мы *не* захватим подстроку bar:

```
>>> match = re.match('(foo(?: bar)baz)', 'foobarbaz')
>>> match.groups()
('foobarbaz',)
```

Группы без захвата обычно используются с опциональными значениями. В таком случае после закрывающей скобки помещается символ?. Например, мы можем сделать подстроку bar опциональной к захвату, а затем сопоставить обе строки foobarbaz:

```
>>> re.match('(foo(?: bar)?baz)', 'foobarbaz')
<re.Match object; span=(0, 9), match='foobarbaz'>
```

а также строку foobaz:

```
>>> re.match('(foo(?: bar)?baz)', 'foobaz')
<re.Match object; span=(0, 6), match='foobaz'>
```

16.1.3. Компиляция регулярного выражения

В главе 15 я упомянул функцию `re.compile()` как способ оптимизации кода путем однократной компиляции регулярного выражения. При вызове метода `re.search()` или `re.split()` механизм регулярных выражений парсит предоставленное строковое значение в поддерживаемый формат. Данный синтаксический анализ выполняется при *каждом* вызове функции. Если же вы компилируете регулярное выражение и присваиваете его переменной, синтаксический анализ выполняется *до* вызова метода, что повышает производительность кода.

Я предпочитаю использовать метод `re.compile()` и присваивать регулярное выражение переменной с информативным именем и/или многократно применять регулярное выражение в разных позициях кода.

Поскольку регулярное выражение довольно длинное и сложное, лучше присвоить его переменной с именем `splitter`. В будущем, при чтении кода, это имя поможет вспомнить предназначение переменной:

```
>>> splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")
>>> splitter.split("Don't worry, spiders,")
['', "Don't", ' ', 'worry', ', ', 'spiders', ',']
```

16.1.4. Скремблирование слов

Научившись обрабатывать *строки*, а затем *слова* в них, давайте придумаем, как скремблировать последние. Начнем с перемешивания букв в *одном слове*. Нам понадобится успешно проходящий тесты алгоритм скремблирования слов, в котором реализованы следующие операции:

- если слово состоит из трех символов и меньше, слово возвращается без изменений;
- из строки вырезается фрагмент без первой и последней букв;
- для перемешивания букв в середине слова используется метод `random.shuffle()`;
- возвращается новое «слово», в котором объединены следующие части: первая буква, скремблированная серединка и последняя буква.

Напишите функцию с именем `scramble()`, выполняющую перечисленные операции, а также тест для нее. Функция может выглядеть примерно так:

```
def scramble(word):
    """Скремблируем слово"""
    pass
```

`pass` — оператор-заглушка, равноценный отсутствию операции. Он используется, когда синтаксически необходимо наличие какого-то оператора или выражения, но не требуется выполнение каких-либо действий. Мы определяем функцию `fry()` как заглушку, чтобы протестировать и убедиться, что она не работает.

```
def test_scramble():
    """Test scramble"""
    state = random.getstate()
    random.seed(1)
```

Передаем методу `random.seed()` известное значение в целях тестирования.

В следующей строке мы выполняем глобальное изменение состояния модуля `random`. Мы восстановим состояние после тестирования, поэтому здесь вызываем метод `random.getstate()` для определения текущего состояния модуля `random`.

```
assert scramble("a") == "a"
assert scramble("ab") == "ab"
assert scramble("abc") == "abc"
assert scramble("abcd") == "acbd"
assert scramble("abcde") == "acbde"
assert scramble("abcdef") == "aecbdf"
assert scramble("abcde'f") == "abcd'ef"
random.setstate(state)
```

Слова из трех букв и меньше возвращаются без изменений.

Данное слово не изменилось потому, что с зерном 1 скремблирование не привело к перемешиванию букв в середине слова.

Теперь очевидно, что слово скремблировано.

Восстанавливаем предыдущее состояние.

Допустим, функции `scramble()` передано слово worry. Мы можем извлечь часть строки методом нарезки списков. Так как в Python нумерация индексов ведется с `0`, мы используем `1` для обращения ко *второму* символу:

```
>>> word = 'worry'
>>> word[1]
'o'
```

Последний индекс любой строки равен `-1`:

```
>>> word[-1]
'y'
```

Чтобы получить срез, используем синтаксис `list[start: stop]`. Поскольку позиция `stop` не указана, мы можем извлечь среднюю часть слова следующим образом:

```
>>> middle = word[1:-1]
>>> middle
'orr'
```

Необходимо импортировать модуль `random`, чтобы получить доступ к методу `random.shuffle()`. Как и в случае с методами `list.sort()` и `list.reverse()`, аргумент перемешивается *на месте*, и функция возвращает значение `None`. Вам может прийти в голову написать следующий код:

```
>>> import random
>>> x = [1, 2, 3]
>>> shuffled = random.shuffle(x)
```

Какое значение будет присвоено переменной shuffled? Что-то в духе [3,1,2] или None?

```
>>> type(shuffled)
<class 'NoneType'>
```

Переменной shuffled присвоено значение None, а список x перемешан *на месте* (рис. 16.3):

```
>>> x
[2, 3, 1]
```

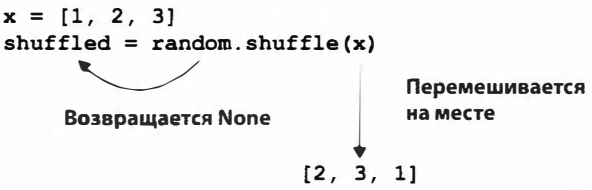


Рис. 16.3. В результате выполнения метода `random.shuffle()` получено значение `None`, которое и присвоено переменной `shuffled`

Надеюсь, вы не потеряли нить рассуждения. Мы не можем перемешать буквы в середине слова, как показано ниже:

```
>>> random.shuffle(middle)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/kyclark/anaconda3/lib/python3.7/random.py", line
278, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'str' object does not support item assignment
```

Потому что переменная middle содержит строковое значение:

```
>>> type(middle)
<class 'str'>
```

Метод random.shuffle() пытается напрямую изменить строковое значение на месте, но это невозможно! Строки в Python *неизменяемы*. Один из обходных путей — сгенерировать новый список middle с символами слова:

```
>>> middle = list(word[1:-1])
>>> middle
['o', 'r', 'r']
```

Выполняем скремблирование:

```
>>> random.shuffle(middle)
>>> middle
['r', 'o', 'r']
```

Теперь необходимо создать новую строку с первой исходной буквой, скремблированной серединкой и последней исходной буквой. Решите эту задачу самостоятельно.

Выполните команду `pytest scrambler.py`, чтобы с помощью Pytest вызвать функцию `test_scramble()` и протестировать программу. Проводите тестирование *после каждого изменения в коде программы.* Проверяйте, что она корректно компилируется и работает. Вносите изменения по очереди, сохраняйте программу и тестируйте.

16.1.5. Скремблирование всех слов в тексте

Теперь, вспоминая опыт из предыдущих упражнений, приступим к выполнению функции `scramble()` для скремблирования всех слов. Знакомый шаблон?

```
splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")
for line in args.text.splitlines():
    for word in splitter.split(line):
        # что здесь происходит?
```

Мы уже обсуждали, как применить функцию к каждому элементу последовательности. На выбор доступны: цикл `for`, представление списка и функция `map()`. Проанализируйте, как можно разбить текст на слова, передать их функции `scramble()`, а затем соединить все части вместе, чтобы вывести скремблированный текст.

Обратите внимание, что функцией `scramble()` обрабатываются как слова, так и символы, к словам не относящиеся (символы между словами). Вам не нужно изменять символы, не относящиеся к словам, поэтому найдите способ проверить, содержит ли аргумент именно слово. Возможно, получится сделать это с помощью регулярного выражения?

Пока все. Напишите собственное решение и используйте прилагаемые тесты для проверки программы.

16.2. Решение

Для меня корректность программы заключается в правильном разделении слов и правильном последующем выполнении функции `scramble()`. Функция скремблирует слова, а затем текст восстанавливается.

```
#!/usr/bin/env python3
"""Скремблирование слов"""

import argparse
import os
import re
import random

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Scramble the letters of words',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='text',
                        help='Input text or file')

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='seed',
                        type=int,
                        default=None)

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text).read().rstrip()
```

Текстовый аргумент может быть представлен как обычным текстом в командной строке, так и именем текстового файла.

Опция `seed` — это целое число, по умолчанию `None`.

Получаем аргументы, чтобы проверить значение `text`.

Если в `args.text` содержится имя существующего и доступного для чтения текстового файла, заменяем значение `args.text` содержимым файла.

```
    return args
```

Возвращаем аргументы вызывающей стороне.

```
# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    random.seed(args.seed)
    splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")

    for line in args.text.splitlines():
        print(''.join(map(scramble, splitter.split(line))))
```

Получаем аргументы командной строки.

Используем `args.seed` для установки значения `random.seed()`. Если `args.seed` присвоено дефолтное значение `None`, это равноценно отсутствию зерна.

Сохраняем скомпилированное регулярное выражение в переменной.

Применяем метод `str.splitlines()`, чтобы сохранить переносы во входном тексте.

Используем переменную `splitter`, позволяющую разбить строку на новый список, который функция `map()` передаст функции `scramble()`. Присоединяем полученный список к пустой строке, чтобы создать новую строку для вывода.

```
# --------------------------------------------------
def scramble(word):
    """Для слов длиной более 3 символов
       перемешиваем буквы посередине"""

    if len(word) > 3 and re.match(r'\w+', word):
        middle = list(word[1:-1])
        random.shuffle(middle)
        word = word[0] + ''.join(middle) + word[-1]

    return word
```

Определяем функцию `scramble()` для скремблирования одного слова.

Скремблируем лишь слова с четырьмя и более буквами, и только если они относятся к символам слов.

Перемешиваем буквы в середине слова.

Копируем содержимое в диапазоне от второго до предпоследнего символов слова в новый список `middle`.

Возвращаем слово, измененное при соответствии критериям.

Присваиваем переменной `word` значение в виде первого исходного символа, скремблированной середины слова и последнего исходного символа.

```
# --------------------------------------------------
def test_scramble():
    """Test scramble"""

    random.seed(1)
    assert scramble("a") == "a"
    assert scramble("ab") == "ab"
    assert scramble("abc") == "abc"
    assert scramble("abcd") == "acbd"
```

Проверка функции `scramble()`

```
    assert scramble("abcde") == "acbde"
    assert scramble("abcdef") == "aecbdf"
    assert scramble("abcde'f") == "abcd'ef"
    random.seed(None)

# -----------------------------------------------
if __name__ == '__main__':
    main()
```

16.3. Обсуждение

В коде функции `get_arg s()` нет ничего нового, поэтому я надеюсь, что он вам понятен. Обратитесь к главе 5, если хотите освежить знания по работе со значением `args.text` и текстом из командной строки или из файла.

16.3.1. Обработка текста

Как упоминалось ранее в этой главе, я обычно присваиваю переменной *скомпилированное* регулярное выражение. Здесь оно присваивается переменной `splitter`:

```
splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")
```

Еще мне нравится метод `re.compile()` потому, что, на мой взгляд, код с ним более читабельный. Без него мне пришлось бы написать следующее:

```
for line in args.text.splitlines():
    print(''.join(map(scramble, re.split("([a-zA-Z]
          (?:[a-zA-Z']*[a-zA-Z])?)", line))))
```

Получается строка кода длиной в 86 символов, в то время как документ «PEP 8 — руководство по написанию кода на Python» (**www.python.org/dev/peps/pep-0008**) рекомендует «не писать строки кода длиннее 79 символов». Мне гораздо удобнее читать показанный ниже код:

```
splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")
for line in args.text.splitlines():
    print(''.join(map(scramble, splitter.split(line))))
```

Не всем из вас данный код покажется понятным. На рис. 16.4 продемонстрирован процесс обработки данных.

1. Сначала Python разделяет строку `"Don't worry, spiders,"`.
2. Создается список из слов (соответствующих нашему регулярному выражению) и символов, не являющихся словами (промежуточные фрагменты).
3. Функция `map()` применяет функцию `scramble()` к каждому элементу списка.
4. При выполнении функции `map()` создается новый список с результатами каждого вызова функции `scramble()`.
5. Метод `str.join()` создает новую строку, передаваемую в виде аргумента функции `print()`.

Рис. 16.4. Наглядная иллюстрация процесса обработки данных функцией `map()`

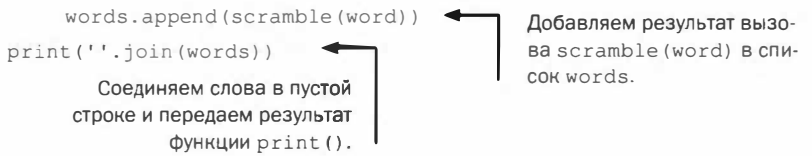
Цикл `for` позволяет решить ту же задачу с помощью более объемного кода:

```
for line in args.text.splitlines():
    words = []
    for word in splitter.split(line):
```

Используем метод `str.splitlines()`, чтобы сохранить переносы в тексте.

Для каждой входящей строки создается пустой список для хранения скремблированных слов.

Выполняем разбиение строки.



Поскольку наша цель заключается в том, чтобы создать новый список, лучше использовать представление списка:

```
for line in args.text.splitlines():
    words = [scramble(word) for word in splitter.split(line)]
    print(''.join(words))
```

Или же можно пойти другим путем и заменить все циклы `for` функцией `map()`:

```
print('\n'.join(
    map(lambda line: ''.join(map(scramble, splitter.split(line))),
        args.text.splitlines())))
```

Последнее решение напоминает мне о бывшем коллеге-программисте, который в шутку сказал: «Если это было трудно написать, то и читать будет трудно!» Впору задуматься, как перекомпоновать код. Обратите внимание, что парсер Pylint ругается на ключевое слово `lambda`, но тут я с ним не согласен и игнорирую «проблему»:

```
scrambler = lambda line: ''.join(map(scramble, splitter.
split(line)))
    print('\n'.join(map(scrambler, args.text.splitlines())))
```

Написание правильного, проверяемого и понятного кода — это не только ремесло, но и искусство. Выберите вариант программы, который вы (и ваши коллеги!) считаете наиболее понятным.

16.3.2. Скремблирование слов

Давайте поближе познакомимся с функцией `scramble()`. Я написал код таким образом, чтобы его было легко использовать с функцией `map()`:

Проверяем, годится ли слово к скремблированию. Во-первых, оно должно содержать не менее четырех символов. Во-вторых, оно должно содержать один или более цифро-буквенных символов, потому что функция обрабатывает как их, так и символы, не относящиеся к словам. Если хотя бы одно из этих условий не выполняется, слово возвращается без изменений.
Код `r'\w+'` необходим для создания «необработанной» строки. Обратите внимание, что регулярное выражение прекрасно работает как с необработанными, так и с обработанными строками, хотя Pylint и ругается на «недопустимую управляющую последовательность» в случае необработанной строки.

Копируем середину слова в новый список с именем `middle`.

```
def scramble(word):
    """Для слов длиной более 3 символов перемешиваем буквы
посередине"""

    if len(word) > 3 and re.match(r'\w+', word):
        middle = list(word[1:-1])
        random.shuffle(middle)
        word = word[0] + ''.join(middle) + word[-1]

    return word
```

Перемешиваем буквы в переменной `middle` на месте. Напоминаю, что данная функция возвращает значение `None`.

Возвращаем слово, которое может быть как скремблировано, так и нет.

Собираем слово, соединяя первый исходный символ, скремблированную середину и последний исходный символ.

16.4. Прокачиваем навыки

— Напишите версию программы, в которой функция `scramble()` сортирует буквы в середине слова в алфавитном порядке, а не перемешивает их.
— Создайте версию, в которой происходит обращение каждого слова, а не скремблирование.
— Напишите программу для расшифровки скремблированного текста. Для этого вам понадобится словарь английских слов, который вы найдете в архиве *inputs/words.txt.zip*. Вам нужно разделить скремблированный текст на слова и символы, не относящиеся к словам, а затем сравнить каждое «слово» со значениями в словаре. Я рекомендую начать со сравнения слов как анаграмм (то есть комбинаций, состоящих из одних и тех же букв), а затем, учитывая первую и последнюю буквы, точно определить расшифровываемое слово.

РЕЗЮМЕ

— Регулярное выражение, которое мы использовали для разбиения текста на слова, оказалось довольно непростым, но только так мы достигли результата. Создать программу без него было бы значительно труднее. Регулярные выражения, хотя иногда и поражают своей сложностью, играют роль чрезвычайно могущественной волшебной палочки, благодаря которой программы становятся невероятно гибкими и полезными.

— Функция `random.shuffle()` принимает список, изменяющийся на месте.

— Представление списка и функция `map()` частенько позволяют сократить объем кода. Главное не переборщить, чтобы не ухудшить его читабельность. Пишите код с умом.

Глава 17
Чепуха: и снова регулярные выражения

В детстве мы часами играли в «Чепуху». Еще не было компьютеров, телевизоров, радио и даже бумаги! Нет, бумага все-таки была. Неважно, главное, что мы днями напролет играли в «Чепуху», и нам это нравилось! А теперь и вы сыграете!

В данной главе мы собираемся написать программу *mad.py*, считывающую содержимое файла, имя которого передается в качестве позиционного аргумента, и занимающуюся поиском заглушек в угловых скобках, таких, как `<verb>` и `<adjective>`. Для замены каждой заглушки мы планируем запрашивать у пользователя соответствующую часть речи, например глагол ("Give me a verb") или прилагательное ("Give me an adjective"). (Обратите внимание, что необходимо подставлять правильный артикль, как мы это делали в главе 2.) Переданным пользователем словом заменяем заглушку в тексте. Например, если пользователь передает глагол drive, мы меняем заглушку `<verb>` в тексте на слово `drive`. Когда все заглушки заменены пользовательскими словами, программа выводит скомпилированный текст.

В каталоге *17_mad_libs/inputs* вы найдете примеры файлов, которые я подготовил, но можете создать и свои собственные. Давайте взглянем на файл *fox.txt*:

```
$ cd 17_mad_libs
$ cat inputs/fox.txt
The quick <adjective> <noun> jumps <preposition> the lazy <noun>.
```

После запуска программы с данным файлом она запрашивает слова для каждой заглушки, а затем выводит получившийся забавный текст:

```
$ ./mad.py inputs/fox.txt
Give me an adjective: surly
Give me a noun: car
Give me a preposition: under
Give me a noun: bicycle
The quick surly car jumps under the lazy bicycle.
```

По умолчанию программа должна работать в интерактивном режиме с приглашением, запрашивающим у пользователя ответы. В целях тестирования программа поддерживает опцию `-i` или `--inputs`, позволяющую обойти вызовы функции `input()` и в автоматическом режиме сформировать вывод:

```
$ ./mad.py inputs/fox.txt -i surly car under bicycle
The quick surly car jumps under the lazy bicycle.
```

Ознакомившись с данной главой, вы научитесь:

- применять метод `sys.exit()`, останавливающий выполнение программы с кодом ошибки;
- использовать «жадные» регулярные выражения;
- с помощью метода `re.findall()` искать все совпадения по шаблону регулярного выражения;
- применять метод `re.sub()` для замены совпадений новым текстом;
- решать задачу другими способами — без регулярных выражений.

17.1. Создание файла mad.py

Для начала создайте программу *mad.py* в каталоге *17_mad_libs*, использовав документ *new.py* или скопировав содержимое файла *template/template.py* в файл *17_mad_libs/mad.py*.

В программе есть строка `type=argparse.FileType('rt')`, с помощью которой мы указываем, что предоставляемые аргументы должны быть текстовыми файлами, доступными для чтения. Строка `nargs='*'`, доступная с помощью опции `-i` или `--inputs`, используется для определения списка с «нулем или более» строковых значений.

При запуске без аргументов или с флагом `-h`, или `--help` программа *mad.py* должна выводить следующие инструкции:

```
$ ./mad.py -h
usage: mad.py [-h] [-i [input [input ...]]] FILE

Mad Libs

positional arguments:
  FILE                  Input file

optional arguments:
-h, --help              show this help message and exit
-i [input [input ...]], --inputs [input [input ...]]
                        Inputs (for testing) (default: None)
```

Если указанный файл не существует, программа обязана выдавать ошибку:

```
$ ./mad.py blargh
usage: mad.py [-h] [-i [str [str ...]]] FILE
mad.py: error: argument FILE: can't open 'blargh': \
[Errno 2] No such file or directory: 'blargh'
```

При отсутствии в содержимом файла заглушек <> программа должна выводить соответствующее сообщение и завершать работу с кодом ошибки (отличным от `0`). Обратите внимание, что в случае возникновения данной ошибки не нужно выводить инструкции, поэтому нам не понадобится метод `parser.error()`, использовавшийся в предыдущих упражнениях:

```
$ cat no_blanks.txt
This text has no placeholders.
$ ./mad.py no_blanks.txt
"no_blanks.txt" has no placeholders.
```

На рис. 17.1 показана диаграмма, помогающая представить процессы обработки данных в программе.

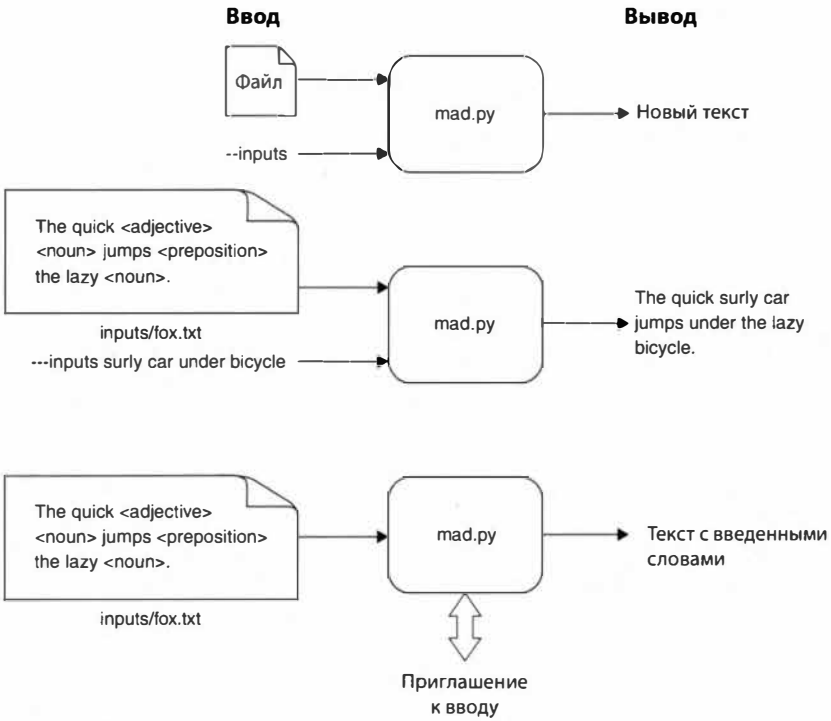


Рис. 17.1. Программа «Чепуха» запрашивает входной текстовый файл, который может содержать список строковых значений. Либо программа в интерактивном режиме запрашивает у пользователя слова для замены заглушек

17.1.1. Поиск угловых скобок с помощью регулярных выражений

Ранее я упоминал о возможных рисках при чтении и загрузке содержимого файла целиком в память. Тем не менее, поскольку мы ищем в тексте все вхождения заглушек <>, в данной программе нам придется прочитать файл целиком. Мы можем сделать это, связав соответствующие функции следующим образом:

```
>>> text = open('inputs/fox.txt').read().rstrip()
>>> text
'The quick <adjective> <noun> jumps <preposition> the lazy <noun>.'
```

Нам нужно искать совпадения текста в угловых скобках, поэтому воспользуемся регулярным выражением. Символ < разыскивается примерно так (рис. 17.2):

```
>>> import re
>>> re.search('<', text)
<re.Match object; span=(10, 11), match='<'>
```

Рис. 17.2. Сопоставление буквального знака «меньше»

А теперь попробуем отыскать пару нашей скобки. Символ точки (.) в регулярном выражении означает «любой символ». После него добавляем символ + для обозначения «один или более» символов. Ниже приведен пример кода:

```
>>> match = re.search('(<.+>)', text)
>>> match.group(1)
'<adjective> <noun> jumps <preposition> the lazy <noun>'
```

Как показано на рис. 17.3, данное выражение охватывает все символы до конца строки, не обращая внимания на первую скобку >. Как правило, с символом * или + (ноль, один или более) механизм регулярных выражений проявляет «жадность» в охвате совпадений. Результат нам не подходит, хотя технически шаблон точно описывает задачу. Так происходит потому, что точка (.) в регулярном выражении означает «любой символ», и правая угловая скобка (ну или математический знак «больше») тоже относится к числу «любых» символов. Согласно шаблону в выборку попадает максимальное количество символов до последней правой угловой скобки, на которой интерпретатор и останавливается. Вот поэтому данный шаблон называется «жадным».

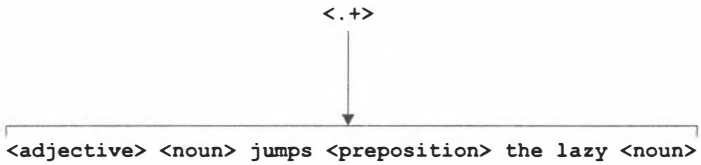


Рис. 17.3. Знак «плюс» в регулярном выражении позволяет выбирать один или более символов. Здесь вы видите пример «жадного» шаблона, направленного на выборку максимально возможного количества символов

Мы можем умерить аппетит регулярного выражения, добавив к символу + знак вопроса, чтобы шаблон соответствовал кратчайшей строке (рис. 17.4):

```
>>> re.search('<.+?>', text)
<re.Match object; span=(10, 21), match='<adjective>'>
```

Рис. 17.4. Благодаря символу ? после знака + в регулярном выражении интерпретатор останавливается на кратчайшей совпавшей строке

В регулярном выражении вместо точки (`.`) можно уточнить запрос, указав один или более символов (любых), *не являющихся ни той ни другой угловой скобкой.* Символьный класс `[<>]` соответствует как правой, так и левой скобке. Мы можем обратить класс, добавив впереди знак вставки (`^`), чтобы получился код `[^<>]` (рис. 17.5). Показанный ниже код соответствует любым символам, кроме левой/правой угловой скобки:

```
>>> re.search('<[^<>]+>', text)
<re.Match object; span=(10, 21), match='<adjective>'>
```

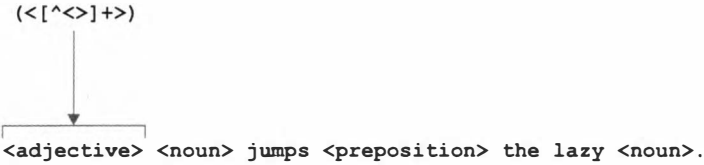


Рис. 17.5. Обращенный символьный класс, соответствующий любым символам, кроме левой/правой угловой скобки

А зачем внутри обращенного класса указаны обе скобки? Не хватит ли только правой? Поясню. Так мы избегаем проблем в случае использования *непарных* скобок. Если в коде указать лишь правую скобку, программа будет работать следующим образом (рис. 17.6):

```
>>> re.search('<[^>]+>', 'foo <<bar> baz')
<re.Match object; span=(4, 10), match='<<bar>'>
```

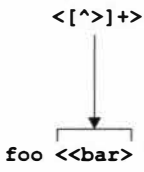


Рис. 17.6. Данное регулярное выражение находит совпадения среди непарных скобок

Если же в обращенном классе указаны *обе* скобки, интерпретатор будет обрабатывать исключительно пары скобок (рис. 17.7):

```
>>> re.search('<[^<>]+>', 'foo <<bar> baz')
<re.Match object; span=(5, 10), match='<bar>'>
```

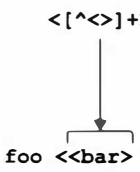


Рис. 17.7. Данное регулярное выражение находит корректные парные скобки и содержащийся в них текст

Мы добавим две пары круглых скобок ( ) . Первая захватывает *всю* заглушку со скобками (рис. 17.8):

```
>>> match = re.search('(<([^<>]+)>)', text)
>>> match.groups()
('<adjective>', 'adjective')
```

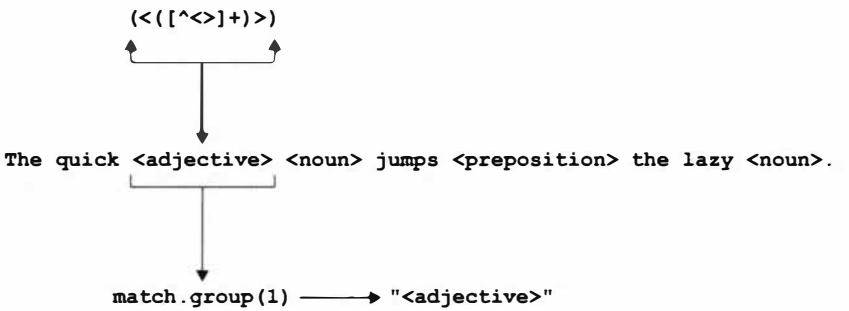


Рис. 17.8. Внешние круглые скобки захватывают угловые скобки и текст внутри них

Вторая — только текст внутри скобок < > (рис. 17.9):

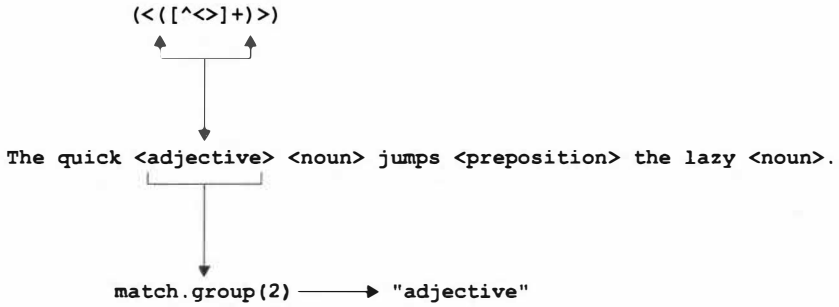


Рис. 17.9. Внутренние круглые скобки захватывают только текст внутри угловых скобок

Есть очень удобная функция `re.findall()`, возвращающая все текстовые совпадения в виде списка кортежей с элементами:

```
>>> from pprint import pprint
>>> matches = re.findall('(<([^<>]+)>)', text)
>>> pprint(matches)
[('<adjective>', 'adjective'),
 ('<noun>', 'noun'),
 ('<preposition>', 'preposition'),
 ('<noun>', 'noun')]
```

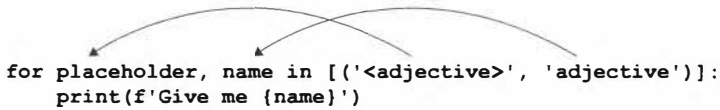


Рис. 17.10. Поскольку список содержит кортеж из двух элементов, мы способны с помощью цикла `for` распаковать их в две переменные

Обратите внимание, что совпадения возвращаются в порядке расположения открывающих круглых скобок, поэтому вся заглушка целиком является первым членом каждого кортежа, а содержащийся в скобках текст — вторым. Мы можем перебирать данный список, *распаковывая* каждый кортеж в переменные (рис. 17.10):

```
>>> for placeholder, name in matches:
...     print(f'Give me {name}')
...
Give me adjective
```

```
Give me noun
Give me preposition
Give me noun
```

Не забудьте в приглашение `input()` добавить правильный артикль (`a` или `an`, как мы делали это в главе 2).

17.1.2. Завершение работы с кодом ошибки

Если в переданном тексте нет заглушек, необходимо вывести сообщение об ошибке. Обычно сообщения об ошибках выводятся в поток `STDERR` (standard error — стандартная ошибка), а функция `print()` позволяет указать аргумент `file`. Мы воспользуемся объектом `sys.stderr` (помните главу 9?). Предварительно необходимо импортировать модуль `sys`:

```
import sys
```

Напомню, что `sys.stderr` — открытый файловый дескриптор, поэтому нет нужды открывать его дополнительно:

```
print('This is an error!', file=sys.stderr)
```

Если заглушек в тексте нет, нужно завершить работу программы с кодом ошибки, по которому операционная система «понимает», что программа не работает должным образом. При обычном завершении программы используется код `0`, что трактуется как «ноль ошибок». Нам же нужно выдать другое целочисленное значение, *отличное от* `0`. Я по привычке использую значение `1`:

```
sys.exit(1)
```

Один из тестов проверяет, определяет ли программа отсутствие заглушек и корректно ли она завершает работу.

Вы также можете вызвать метод `sys.exit()` с неким строковым значением, которое будет выведено в поток `sys.stderr`, и программа завершит работу с кодом ошибки `1`:

```
sys.exit('This will kill your program and print an error message!')
```

17.1.3. Запрос слов

Каждой части речи (заглушке) в тексте необходимо передать значение, которое мы получим либо с помощью аргумента `--inputs`, либо непосредственно от пользователя. Если опциональный аргумент `--inputs` не применяется, мы можем вызвать функцию `input()` для получения ответа от пользователя.

Данная функция принимает строковое значение для использования в качестве приглашения:

```
>>> value = input('Give me an adjective: ')
Give me an adjective: blue
```

И возвращает строковое значение, введенное пользователем перед нажатием клавиши **Enter**:

```
>>> value
'blue'
```

Как вариант, попробуйте передать программе подходящие значения и не вызывать функцию `input()`. (Я рекомендую использовать опцию `--inputs` только в целях тестирования.) Так вы сможете указать количество строковых значений по числу заглушек (рис. 17.11).

Например, для программы с аргументом `--inputs` можно использовать следующий список значений (для замены заглушек в тексте из файла *fox.txt*):

```
>>> inputs = ['surly', 'car', 'under', 'bicycle']
```

Требуется извлечь и удалить первую строку, «surly», из списка `inputs`. Тут пригодится метод `list.pop()`, но по умолчанию он удаляет *последний* элемент:

```
>>> inputs.pop()
'bicycle'
```

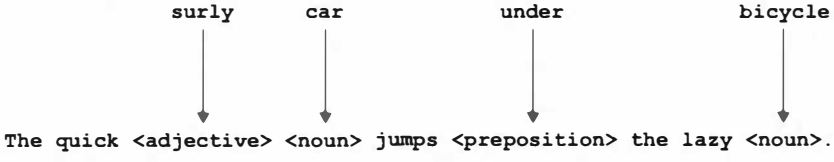


Рис. 17.11. Данные, вводимые из командной строки, совпадают с заглушками в тексте

Метод `list.pop()` принимает опциональный аргумент с индексом элемента, который нужно удалить. Сможете сами написать нужный код? Обязательно прочитайте справку (`help(list.pop)`), если вдруг застрянете.

17.1.4. Замена заглушек в тексте

После получения значений для всех заглушек, вам нужно заменить их в тексте. Я предлагаю обратить внимание на функцию `re.sub()` (от слова substitute — замещение), которая замещает другим значением текст, соответствующий заданному регулярному выражению. Для подробной информации прочитайте, пожалуйста, справку (`help(re.sub)`):

```
sub(pattern, repl, string, count=0, flags=0)
    Return the string obtained by replacing the leftmost
    non-overlapping occurrences of the pattern in string by the
    replacement repl.
```

Опуская подробности, вам нужно использовать аналогичный шаблон и заменить все заглушки соответствующим значением.

Кстати, для решения задачи из этой главы можно обойтись без функции `re.sub()`. Попробуйте написать код, в котором вообще не используется модуль `re`.

Итак, приступайте к написанию программы и не забывайте ее тестировать!

17.2. Решение

Ну что, удобно работать с регулярными выражениями? Признаю, их довольно сложно освоить, но, разобравшись, вы будете поражены их богатыми возможностями.

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

import argparse
import re
import sys
```

```
# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
    description='Mad Libs',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('rt'),
                        help='Input file')

    parser.add_argument('-i',
                        '--inputs',
                        help='Inputs (for testing)',
                        metavar='input',
                        type=str,
                        nargs='*')

    return parser.parse_args()

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    inputs = args.inputs
    text = args.file.read().rstrip()
    blanks = re.findall('(<([^<>]+)>)', text)

    if not blanks:
        sys.exit(f'" {args.file.name}" has no placeholders.')
```

Аргумент файла должен быть представлен текстовым файлом, доступным для чтения.

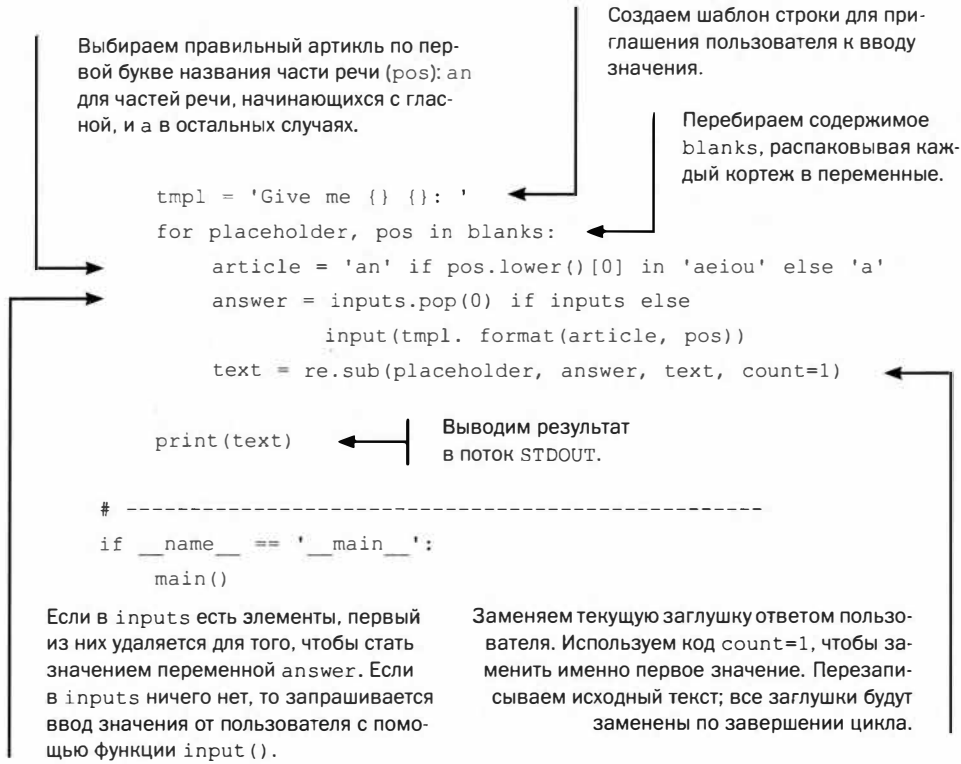
Опция `--inputs` может содержать ноль или более строк.

С помощью регулярного выражения ищем все вхождения левой угловой скобки, за которой следует один или более символов, отличных от левой или правой угловой скобок, а затем следует правая угловая скобка. Используем две группы захвата, чтобы захватить заглушку со скобками и сам текст уже без скобок.

Открываем и считываем входной файл, удаляя символы перевода строки в конце строк.

Проверяем текст на отсутствие заглушек.

Выводим в поток `STDERR` сообщение, что текст в указанном файле не содержит заглушек, и завершаем работу программы с кодом, отличным от `0`, чтобы указать операционной системе на ошибку.



17.3. Обсуждение

Начнем с определения аргументов. Входной файл должен объявляться с помощью кода `type=argparse.FileType('rt')`, чтобы модуль `argparse` проверял допустимость указанного в аргументе текстового файла. Аргумент `--inputs` опционален, поэтому мы можем прибегнуть к коду `nargs='*'` для указания ноля или более аргументов. Если входные данные не предоставлены, по умолчанию используется значение `None`, поэтому проверьте, что в таком случае вы не обрабатываете список и не выполняете соответствующие операции.

17.3.1. Замена заглушек с помощью регулярных выражений

Есть небольшая проблема, ожидающая вас при использовании метода `re.sub()`. Предположим, вы заменили первую заглушку `<adjective>` словом blue:

```
>>> text = 'The quick blue <noun> jumps <preposition>
           the lazy <noun>.'
```

Далее вы пробуете заменить заглушку <noun> словом dog:

```
>>> text = re.sub('<noun>', 'dog', text)
```

Давайте проверим значение переменной text на данном этапе:

```
>>> text
'The quick blue dog jumps <preposition> the lazy dog.'
```

Так как в тексте присутствовали две заглушки <noun>, они обе были заменены на слово dog (рис. 17.12).

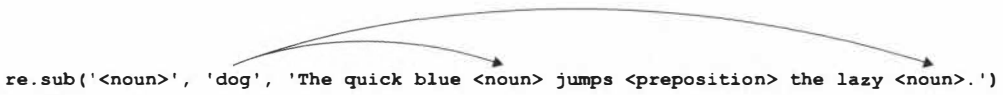


Рис. 17.12. Функция `re.sub()` заменяет все совпадения

Мы можем применить параметр count=1, чтобы изменить только первое вхождение (рис. 17.13):

```
>>> text = 'The quick blue <noun> jumps <preposition>
           the lazy <noun>.'
>>> text = re.sub('<noun>', 'dog', text, count=1)
>>> text
'The quick blue dog jumps <preposition> the lazy <noun>.'
```

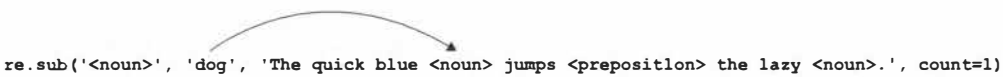


Рис. 17.13. Используйте в методе `re.sub()` параметр `count`, чтобы ограничить количество замен

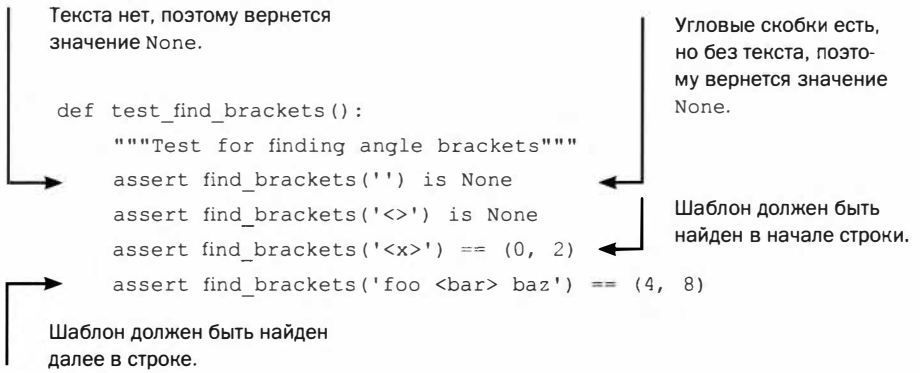
Теперь мы способны продолжить замену других заглушек.

17.3.2. Версия программы без регулярных выражений

Думаю, что решение с регулярными выражениями я объяснил достаточно подробно. Оно довольно элегантное, но нашу программу, безусловно, можно написать и без них. Вот как поступил бы я.

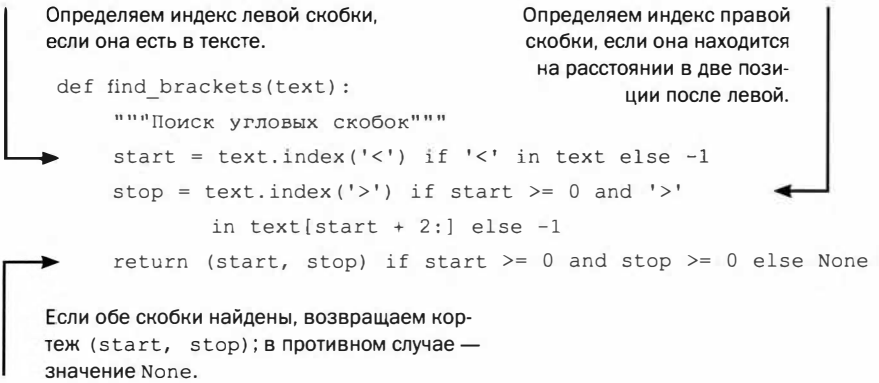
В первую очередь мне нужен способ поиска в тексте символов <>. Я начинаю с разработки теста, который поможет мне представить, что нужно передать функции и чего ожидать в ответ на допустимые и недопустимые значения.

Я решил возвращать значение `None`, если шаблон отсутствует, и кортеж индексов `(start, stop)`, если присутствует:




```
def test_find_brackets():
    """Test for finding angle brackets"""
    assert find_brackets('') is None
    assert find_brackets('<>') is None
    assert find_brackets('<x>') == (0, 2)
    assert find_brackets('foo <bar> baz') == (4, 8)
```

Теперь мне нужно написать код, который успешно пройдет данный тест. Вот что у меня получилось:




```
def find_brackets(text):
    """Поиск угловых скобок"""
    start = text.index('<') if '<' in text else -1
    stop = text.index('>') if start >= 0 and '>'
            in text[start + 2:] else -1
    return (start, stop) if start >= 0 and stop >= 0 else None
```

Данная функция неплохо справляется с поставленной задачей и проходит приведенные тесты, однако она не совсем корректна, поскольку не учитывает парность скобок:

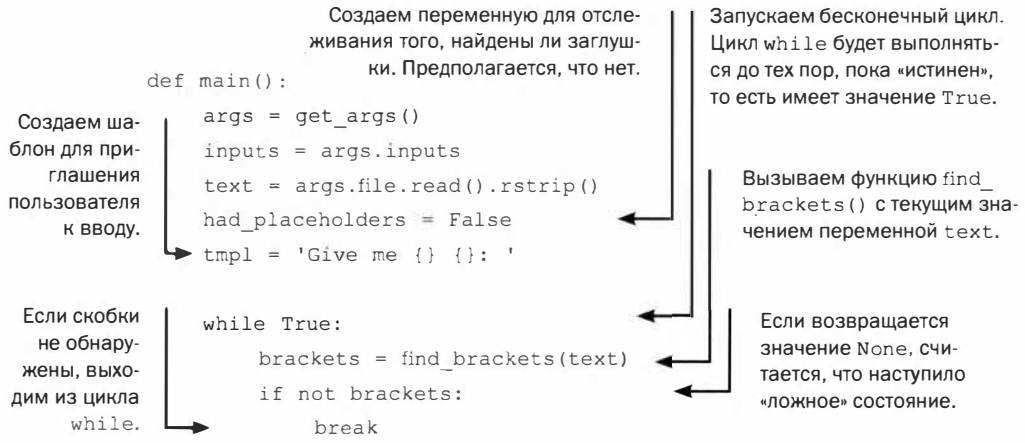
```
>>> text = 'foo <<bar> baz'
>>> find_brackets(text)
[4, 9]
>>> text[4:10]
'<<bar>'
```

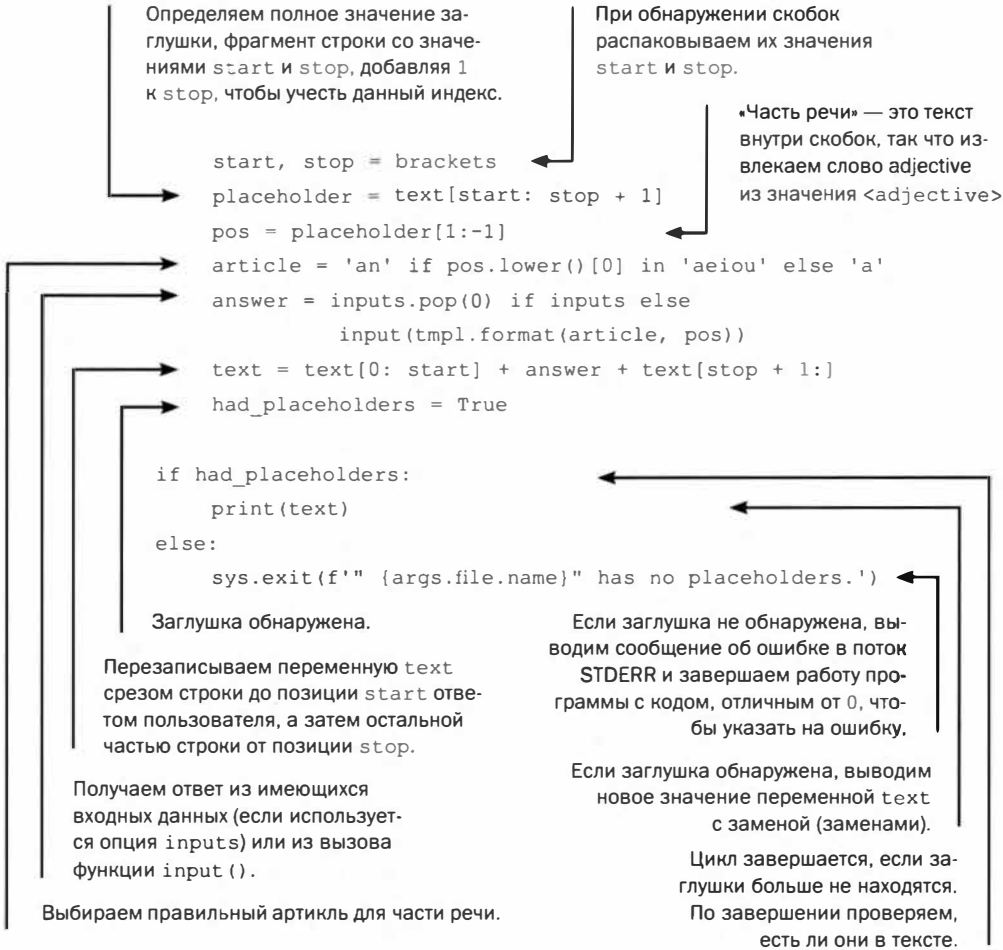
Возможно, вы догадались, что я выбрал угловые скобки для ассоциации с HTML-тегами, такими как `<head>` и `<img>`. Язык HTML печально известен своей некорректностью по причине ошибок в тегах, которые может допустить веб-дизайнер, или недочета в ПО для верстки веб-страниц. Фишка в том, что большинство веб-браузеров довольно спокойно реагируют на такие недочеты при анализе HTML-кода и проигнорируют ошибку в теге типа `<<head>` вместо `<head>`.

Версия программы с регулярными выражениями специально защищает от сопоставления непарных скобок, так как содержит класс `[^<>]` для определения текста, который не может содержать угловые скобки. Допустимо, конечно, написать функцию `find_brackets()`, которая ищет только парные скобки, но, честно говоря, игра не стоит свеч. Функция ссылалась бы на одну из сильных сторон механизма регулярных выражений и выполняла бы поиск частичных совпадений (первая левая скобка), убеждалась бы в отсутствии полного совпадения и искала бы далее заново (со следующей левой скобки). Писать такой код вручную весьма утомительно и, честно говоря, скучно.

Так или иначе, моя функция работает со всеми видами входных данных. Обратите внимание, что она возвращает пары скобок по очереди. Так как я меняю текст сразу после обнаружения пары скобок, из-за этого, скорее всего, меняются позиции всех последующих скобок. Именно поэтому лучше обрабатывать пары скобок по очереди.

Вот как я использовал бы свое решение в функции `main()`:





17.4. Прокачиваем навыки

— Создайте код, позволяющий найти все HTML-теги, заключенные в скобки < > и </ > на веб-странице, загруженной из интернета.
— Напишите программу, которая будет искать непарные группы круглых скобок (), квадратных скобок [] и фигурных скобок {}. Создайте входные файлы с текстом и парными/непарными скобками, а также напишите тесты, проверяющие, что ваша программа корректно определяет все эти скобки.

РЕЗЮМЕ

— Регулярные выражения похожи на функции, описывающие шаблоны значений, которые требуется найти. Механизм регулярных выражений будет пытаться найти шаблоны, обработать несоответствия и начать поиск заново.
— Шаблоны регулярных выражений с символами * и + являются «жадными» в том смысле, что включают в выборку максимально доступное количество символов. Символ? после знаков * и + уменьшает их «аппетит», добиваясь того, чтобы они включали в выборку как можно меньшее количество символов.
— Функция `re.findall()` возвращает список всех совпадающих строк или групп захвата для заданного шаблона.
— Функция `re.sub()` заменяет совпадения в тексте новыми значениями.
— Можно остановить программу в любой момент, применив функцию `sys.exit()`. Если ей не переданы аргументы, код выхода по умолчанию будет равен `0`, что означает, что ошибок не обнаружено. Если требуется указать, что произошла ошибка, используйте любое значение, отличное от `0`, например `1`. Или строковое значение, которое будет выведено в поток `STDERR`, а код ошибки, отличный от `0`, будет использоваться автоматически.

Глава 18

Гематрия: анализ текста с помощью ASCII-значений букв

Гематрия — это система анализа смысла слов и фраз на основе числовых значений, входящих в них букв (**https://ru.wikipedia.org/wiki/Гематрия**). В стандартной версии гематрии (*Mispar hechrechi — абсолютное значение)* каждому символу еврейского алфавита присваивается числовое значение в диапазоне от 1 до 400, но существует множество других методов вычисления числового значения букв. Для программирования слова данные значения складываются. В Откровении Иоанна, Главе 13, стихе 18 христианской Библии говорится: «Здесь нужна мудрость: человек сообразительный пусть вычислит число зверя. Это число человека, а число его 666». Некоторые ученые считают, что данное число образовано от суммы букв в имени «Нерон Кесарь» в еврейском написании и что оно использовалось как способ упомянуть римского императора, не называя его.

Мы напишем программу *gematria.py*, представляющую каждое слово из переданного текста в виде чисел путем складывания числовых значений букв в слове. Существует много вариантов подобного кодирования.

Например, можно присвоить букве a значение 1, b — значение 2 и т. д. Сделаем задачу интереснее. Давайте воспользуемся таблицей ASCII-символов (**https://ru.wikipedia.org/wiki/ASCII**) для получения числовых значений букв английского алфавита. Для символов, отличных от латиницы, можно взять значения Unicode. Однако в данном упражнении мы рассмотрим ASCII-коды только букв английского алфавита.

Входной текст может быть передан в командной строке:

```
$ ./gematria.py 'foo bar baz'
324 309 317
```

Или извлечен из файла:

```
$ ./gematria.py ../inputs/fox.txt
289 541 552 333 559 444 321 448 314
```

На рис. 18.1 показана диаграмма, иллюстрирующая устройство программы.

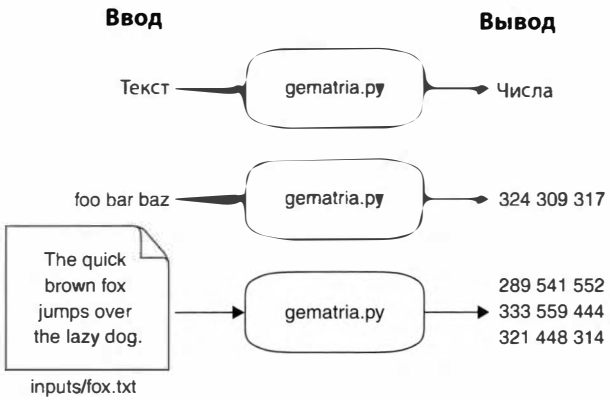


Рис. 18.1. Программа gematria.py принимает текст и выдает числовые значения каждого слова

Прочитав данную главу, вы научитесь:

- применять функции `ord()` и `chr()`;
- разбираться в организации символов в таблице ASCII;
- использовать в регулярных выражениях диапазоны символов;
- применять функцию `re.sub()`;
- деанонимизировать функцию `map()`;
- применять функцию `sum()`, в том числе в связке с функцией `reduce()`;
- сортировать строковые значения без учета регистра.

18.1. Создание файла *gematria.py*

Я не устаю рекомендовать вам создавать свои программы, избегая лишней работы по копированию шаблонного исходного кода. Пожалуйста, скопируйте содержимое документа *template/template.py* в файл *18_gematria/gematria.py* либо выполните команду `new.py gematria.py` в каталоге *18_gematria*, чтобы подготовить начальную версию программы *gematria.py*.

Измените в файле *gematria.py* код, чтобы программа при запуске без аргументов или с флагом `-h`/`--help` выводила следующие инструкции:

```
$ ./gematria.py -h
usage: gematria.py [-h] text

Gematria

positional arguments:
  text         Input text or file

optional arguments:
-h, --help  show this help message and exit
```

Как и в предыдущих примерах, входные данные могут поступать непосредственно из командной строки или из файла. Я предлагаю вам реализовать это самостоятельно, вспомнив решение из главы 5, а затем изменить код функции `main()` следующим образом:

```
def main():
    args = get_args()
    print(args.text)
```

Проверьте, что программа принимает и выводит текст из командной строки:

```
$ ./gematria.py 'Death smiles at us all, but all a man can do is
smile back.'
Death smiles at us all, but all a man can do is smile back.
```

и из файла:

```
$ ./gematria.py ../inputs/spiders.txt
Don't worry, spiders,
```

```
I keep house
casually.
```

18.1.1. Очистка слов от лишних символов

Давайте обсудим, как будет вычисляться гематрия одного слова, так как от этого зависит способ разбиения текста в следующем разделе. Чтобы работать только со значениями ASCII, предлагаю удалить любые символы, отличные от строчных и прописных букв английского алфавита и арабских цифр 0–9. Определить символьный класс поможет регулярное выражение `[A-Za-z0-9]`.

Мы можем использовать функцию `re.findall()` из главы 17, чтобы найти в слове все символы, соответствующие данному классу. Например, для слова Don't должны быть отобраны все символы, кроме апострофа (рис. 18.2):

```
>>> re.findall('[A-Za-z0-9]', "Don't")
['D', 'o', 'n', 't']
```

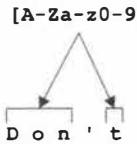


Рис. 18.2. Данный символьный класс допускает соответствие только цифробуквенным символам

Если мы укажем знак вставки (^) в начале регулярного выражения символьного класса `[^A-Za-z0-9]`, то найдем любые символы, *кроме* перечисленных в выражении. Теперь в результате совпадает *только* апостроф (рис. 18.3):

```
>>> import re
>>> re.findall('[^A-Za-z0-9]', "Don't")
["'"]
```

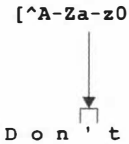


Рис. 18.3. Знак вставки исключает из выборки символы, перечисленные в регулярном выражении символьного класса, то есть все арабские цифры и буквы английского алфавита

Мы можем воспользоваться функцией `re.sub()` для замены любых символов во втором классе пустой строкой. Как вы узнали из главы 17, заменяются *все* вхождения, если не указан код `count=n`:

```
>>> word = re.sub('[^A-Za-z0-9]', '', "Don't")
>>> word
'Dont'
```

Таким образом мы планируем очищать все слова, к которым будет применяться гематрия, как показано на рис. 18.4.

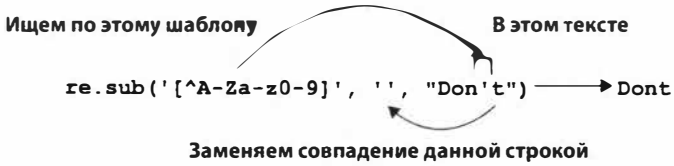


Рис. 18.4. Функция `re.sub()` заменяет все символы по шаблону указанным значением

18.1.2. Числовые значения и диапазоны символов

Для вычисления гематрии строки в духе "Dont" мы преобразуем *каждый символ* в строке в числовое значение, а затем сложим их. Для начала давайте выясним, как вычислять гематрию одного символа.

В языке Python есть функция `ord()`, возвращающая для указанного символа числовое значение, соответствующее позиции этого символа в таблице Американского стандартного кода для обмена информацией (ASCII*, American standard code for information interchange):

```
>>> ord('D')
68
>>> ord('o')
111
```

Функция `chr()` работает наоборот и возвращает символ, соответствующий введенному числовому значению** позиции в таблице ASCII:

```
>>> chr(68)
```

---

* Произносится как «аски».

** В десятичной системе исчисления. *Прим. перев.*

```
'D'
>>> chr(111)
'o'
```

Ниже приведена таблица ASCII. Для удобства я указал сокращение «УС» («управляющие символы») для символов с позициями от 0 до 31, а также 127, поскольку их нельзя вывести.

```
$ ./asciitbl.py
  0 УС    16 УС    32 SPACE 48 0    64 @    80 P     96 `    112 p
  1 УС    17 УС    33 !     49 1    65 A    81 Q     97 a    113 q
  2 УС    18 УС    34 "     50 2    66 B    82 R     98 b    114 r
  3 УС    19 УС    35 #     51 3    67 C    83 S     99 c    115 s
  4 УС    20 УС    36 $     52 4    68 D    84 T    100 d    116 t
  5 УС    21 УС    37 %     53 5    69 E    85 U    101 e    117 u
  6 УС    22 УС    38 &     54 6    70 F    86 V    102 f    118 v
  7 УС    23 УС    39 '     55 7    71 G    87 W    103 g    119 w
  8 УС    24 УС    40 (     56 8    72 H    88 X    104 h    120 x
  9 УС    25 УС    41 )     57 9    73 I    89 Y    105 i    121 y
 10 УС    26 УС    42 *     58 :    74 J    90 Z    106 j    122 z
 11 УС    27 УС    43 +     59 ;    75 K    91 [    107 k    123 {
 12 УС    28 УС    44 ,     60 <    76 L    92 \    108 l    124 |
 13 УС    29 УС    45 -     61 =    77 M    93 ]    109 m    125 }
 14 УС    30 УС    46 .     62 >    78 N    94 ^    110 n    126 ~
 15 УС    31 УС    47 /     63 ?    79 O    95 _    111 o    127 УС
```

Примечание. Файл *asciitbl.py* есть в каталоге *18_gematria* репозитория с примерами к данной книге.

Мы можем использовать цикл `for` для циклического перебора всех символов в строке:

```
>>> word = "Dont"
>>> for char in word:
...     print(char, ord(char))
...
D 68
o 111
n 110
t 116
```

Обратите внимание, что прописные и строчные буквы размещаются в таблице ASCII на разных позициях и имеют разные числовые значения. Вполне логично, ведь, к примеру, `D` и `d` — это две разные буквы:

```
>>> ord('D')
68
>>> ord('d')
100
```

Мы можем перебирать буквы от a до z, определяя их значения в таблице:

```
>>> [chr(n) for n in range(ord('a'), ord('z') + 1)]
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

В таблице ASCII буквы от a до z расположены по порядку. То же верно для букв A — Z и цифр 0–9, поэтому в качестве регулярного выражения можно использовать шаблон `[A-Za-z0-9]`.

Обратите внимание, что у прописных букв значения *меньше*, чем у строчных, в связи с чем диапазон `[a-Z]` недопустим. Попробуйте выполнить в REPL-интерфейсе следующий код:

```
>>> re.findall('[a-Z]', word)
```

И в результате вы получите соответствующую ошибку:

```
re.error: bad character range a-Z at position 1
```

Допустим диапазон `[A-z]`:

```
>>> re.findall('[A-z]', word)
['D', 'o', 'n', 't']
```

Обратите внимание, что буквы Z и a не смежные:

```
>>> ord('Z'), ord('a')
(90, 97)
```

Между ними есть другие символы:

```
>>> [chr(n) for n in range(ord('Z') + 1, ord('a'))]
```

```
['[', '\\', ']', '^', '_', '`']
```

Если мы попробуем вывести весь диапазон печатных символов от A до z, то в результат попадут не только буквы, но и некоторые специальные символы:

```
>>> import string
>>> re.findall('[A-z]', string.printable)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
 '[', '\\', ']', '^', '_', '`']
```

Вот почему надежнее указывать нужные символы в виде трех отдельных диапазонов, `[A-Za-z0-9]`, где помимо арабских цифр используются два диапазона букв «от a до z», строчных и прописных.

18.1.3. Гематрии букв и слов

Напомню нашу задачу: определить числовые значения всех символов в слове, а затем сложить полученные значения. Для этого в Python есть удобная функция `sum()`, которая складывает числа из списка:

```
>>> sum([1, 2, 3])
6
```

Мы можем вручную вычислить гематрию строки "Dont", вызвав функцию `ord()` для каждой буквы и передав результаты в виде списка функции `sum()`:

```
>>> sum([ord('D'), ord('o'), ord('n'), ord('t')])
405
```

Итак, как вычислить значение позиции каждой буквы в слове и составить список? Похожую задачу мы решали уже не раз. С какого варианта начнем? Давайте воспользуемся удобным циклом `for`:

```
>>> word = 'Dont'
>>> vals = []
```

```
>>> for char in word:
...     vals.append(ord(char))
...
>>> vals
[68, 111, 110, 116]
```

Затем вместим код в одну строку, применив метод представления списка:

```
>>> vals = [ord(char) for char in word]
>>> vals
[68, 111, 110, 116]
```

А теперь можем перейти к функции `map()`:

```
>>> vals = map(lambda char: ord(char), word)
>>> list(vals)
[68, 111, 110, 116]
```

Тут следует уточнить, что в коде функции `map()` нет необходимости указывать слово `lambda`, потому что функция `ord()` ожидает как раз то значение, которое получит от функции `map()`. Ниже показан пример этого прекрасного кода:

```
>>> vals = map(ord, word)
>>> list(vals)
[68, 111, 110, 116]
```

На мой взгляд, получилось очень аккуратно и красиво!

Теперь мы можем сложить числа из списка, чтобы вычислить гематрию слова:

```
>>> sum(map(ord, word))
405
```

Результат верен:

```
>>> sum([68, 111, 110, 116])
405
```

18.1.4. Метод functools.reduce()

Раз в Python есть функция `sum()`, то наверняка в нем должна быть функция `product()` для умножения чисел в списке. Увы, это не встроенная функция, хотя она и представляет собой распространенный способ *сведения значений в списке* к одному значению.

Функция `reduce()` из модуля `functools` предоставляет универсальный способ сведения списка. Давайте обратимся к документации о том, как ее использовать:

```
>>> from functools import reduce
>>> help(reduce)
reduce(...)
    reduce(function, sequence[, initial]) -> value

    Apply a function of two arguments cumulatively to the items
    of a sequence, from left to right, so as to reduce the
    sequence to a single value.
    For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
    calculates ((((1+2)+3)+4)+5). If initial is present, it is
    placed before the items of the sequence in the calculation,
    and serves as a default when the sequence is empty.
```

Мы имеем дело с еще одной функцией высокого порядка, которая принимает *другую функцию* в качестве первого аргумента по аналогии с `map()` и `filter()`. В документации указано, как написать функцию сведения списка:

```
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
```

Если мы изменим оператор `+` на `*`, у нас получится перемножать числа:

```
>>> reduce(lambda x, y: x * y, [1, 2, 3, 4, 5])
120
```

Вот как может выглядеть соответствующая функция:

```
def product(vals):
    return reduce(lambda x, y: x * y, vals)
```

И теперь мы ее вызываем:

```
>>> product(range(1,6))
120
```

Вместо собственной анонимной функции мы можем использовать любую функцию, принимающую два аргумента. Например, функцию `operator.mul`:

```
>>> import operator
>>> help(operator.mul)
mul(a, b, /)
    Same as a * b.
```

Проще написать так:

```
def product(vals):
    return reduce(operator.mul, vals)
```

К счастью, модуль `math` содержит похожую функцию, `prod()`, которая нам подойдет:

```
>>> import math
>>> math.prod(range(1,6))
120
```

В продолжение темы, метод `str.join()` также сводит список строк в одно строковое значение. Вот так выглядит код:

```
def join(sep, vals):
    return reduce(lambda x, y: x + sep + y, vals)
```

Мне по душе следующий синтаксис соединения строковых значений с помощью метода `str.join()`:

```
>>> join(', ', ['Hey', 'Nonny', 'Nonny'])
'Hey, Nonny, Nonny'
```

Итак, обрабатывая список значений, которые вы хотите свести в одно, не забывайте о функции `reduce()`.

18.1.5. Гематрии слов

Да, мы усердно потрудились, складывая числовые значения символов, но ведь это так увлекательно! Однако давайте вернемся к делу.

Теперь мы можем написать функцию для вычисления гематрии слова. Она будет складывать числовые значения символов из таблицы ASCII. Я выбрал для функции имя `word2num()`, и вот как выглядит тест для нее:

```
def test_word2num():
    """Test word2num"""
    assert word2num("a") == "97"
    assert word2num("abc") == "294"
    assert word2num("ab'c") == "294"
    assert word2num("4a-b'c,") == "346"
```

Обратите внимание, что моя функция возвращает строковое значение, а не целое число. Так происходит потому, что я планирую использовать результат с методом `str.join()`, принимающим исключительно строки, например, `'405'` вместо числа `405`:

```
>>> from gematria import word2num
>>> word2num("Don't")
'405'
```

Подводя итоги, следует упомянуть, что функция `word2num()` принимает слово, удаляет недопустимые символы, преобразует с помощью функции `ord()` оставшиеся символы в значения из таблицы ASCII и возвращает в виде строки сумму этих значений.

18.1.6. Разбиение текста

Тесты учитывают, что вы сохраняете те же переносы строк, что и в исходном тексте, поэтому я рекомендую прибегнуть к методу `str.splitlines()`, как мы делали в других упражнениях. В главах 15 и 16 мы использовали разные регулярные выражения для разбиения строк на слова. Данный процесс в программах, связанных с обработкой естественного языка (NLP), иногда называют токенизацией. Если вы работаете над функцией `word2num()`, проходящей приведенные мной тесты, можно применить метод `str.split()`, чтобы разбить строку по пробелам. Метод проигнорирует любые знаки, кроме буквенных

символов и цифр. Разумеется, вы можете разбивать строки на слова любыми способами, которые вам по душе.

Показанный ниже код сохраняет переносы строк и реконструирует текст. Можете ли вы изменить его, добавив функцию `word2num()` так, чтобы вместо этого он выводил гематрии слов, как показано на рис. 18.5?

```
def main():
    args = get_args()
    for line in args.text.splitlines():
        for word in line.split():
            # что здесь происходит?
            print(' '.join(line.split()))
```

Рис. 18.5. Каждое слово в тексте очищается и выводится его гематрия

Функция должна выводить по одному числу для каждого слова:

```
$ ./gematria.py ../inputs/fox.txt
289541 552333 559444 321448 314
```

Пора закругляться. Пишите программу и обязательно тестируйте ее! Увидимся чуть позже.

18.2. Решение

Мне нравится криптография и шифрование сообщений, и наша программа (как бы) шифрует входной текст, хотя и необратимо. Так или иначе, интересно придумывать разные способы преобразования и шифрования текста.

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

import argparse
import os
import re
```

```
# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Gematria',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='text',
                        help='Input text or file')

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text).read().rstrip()

    return args

# --------------------------------------------------
def main():
    """Да грянет джаз!" ""

    args = get_args()
    for line in args.text.splitlines():
        print(' '.join(map(word2num, line.split())))

# --------------------------------------------------
def word2num(word):
    """Складывание числовых значений всех символов"""

    return str(sum(map(ord, re.sub('[^A-Za-z0-9]', '', word))))

# --------------------------------------------------
def test_word2num():
    """Test word2num"""

    assert word2num("a") == "97"
    assert word2num("abc") == "294"
    assert word2num("ab'c") == "294"
    assert word2num("4a-b'c,") == "346"
```

Аргумент `text` принимает строковое значение, в том числе и имя файла.

Получаем проанализированные аргументы командной строки.

Проверяем, существует ли файл, указанный в аргументе `text`.

Пишем в `args.text` содержимое файла.

Возвращаем аргументы.

Получаем проанализированные аргументы.

Разбиваем значение `args.text` с символами перевода строки, чтобы сохранить переносы.

Определяем функцию для преобразования слова в число.

Разбиваем строку по пробелам, сопоставляем результат с `word2num()`, а затем соединяем его с пробелами.

С помощью метода `re.sub()` удаляем любые знаки, кроме цифробуквенных символов. Сопоставляем результирующую строку с помощью функции `ord()`, складываем числовые значения символов и возвращаем сумму в формате строки.

Код для тестирования функции `word2num()`.

```
# --------------------------------------------------
if __name__ == '__main__':
    main()
```

18.3. Обсуждение

Я надеюсь, вам понятен код функции `get_args()`, ведь мы использовали ее уже множество раз. Давайте перейдем к функции `word2num()`.

18.3.1. Функция word2num()

Мой первый вариант функции `word2num()`:

```
def word2num(word):
    vals = []
    for char in re.sub('[^A-Za-z0-9]', '', word):
        vals.append(ord(char))

    return str(sum(vals))
```

Инициализируем пустой список для хранения числовых значений символов.

Перебираем все символы, возвращенные методом `re.sub()`.

Складываем значения и возвращаем сумму в виде строки.

Определяем числовое значение символа и добавляем его в список `vals`.

Получилось четыре строки кода вместо одной. Пожалуй, воспользуюсь представлением списка и сокращу три строки кода в одну, как показано ниже:

```
def word2num(word):
    vals = [ord(char) for char in re.sub('[^A-Za-z0-9]', '', word)]
    return str(sum(vals))
```

Вот еще способ уместить код в одну строку, хотя, не буду спорить, данный вариант сложен для восприятия:

```
def word2num(word):
    return str(sum([ord(char) for char in re.sub('[^A-Za-z0-9]',
                '', word)]))
```

Так что я по-прежнему уверен, что версия с функцией `map()` — наиболее лаконичная и приятная для чтения:

```
def word2num(word):
    return str(sum(map(ord, re.sub('[^A-Za-z0-9]', '', word))))
```

На рис. 18.6 показано, как три версии кода соотносятся друг с другом.

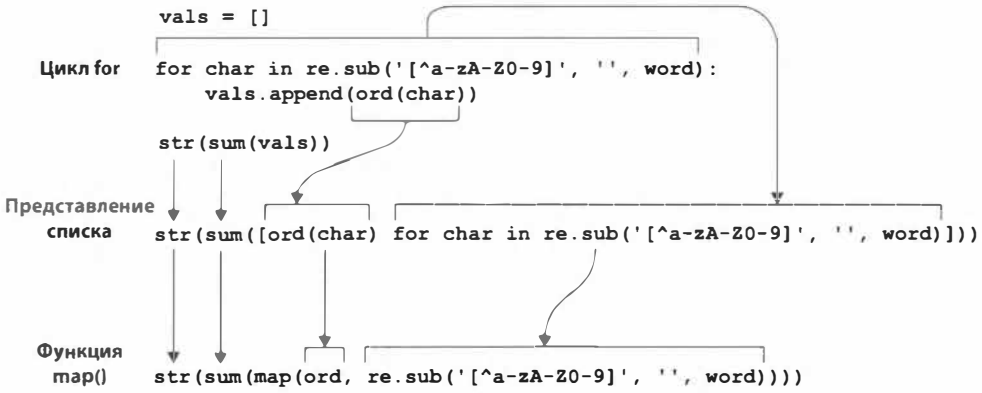


Рис. 18.6. Соотношение версий кода с циклом `for`, представлением списка и функцией `map()`

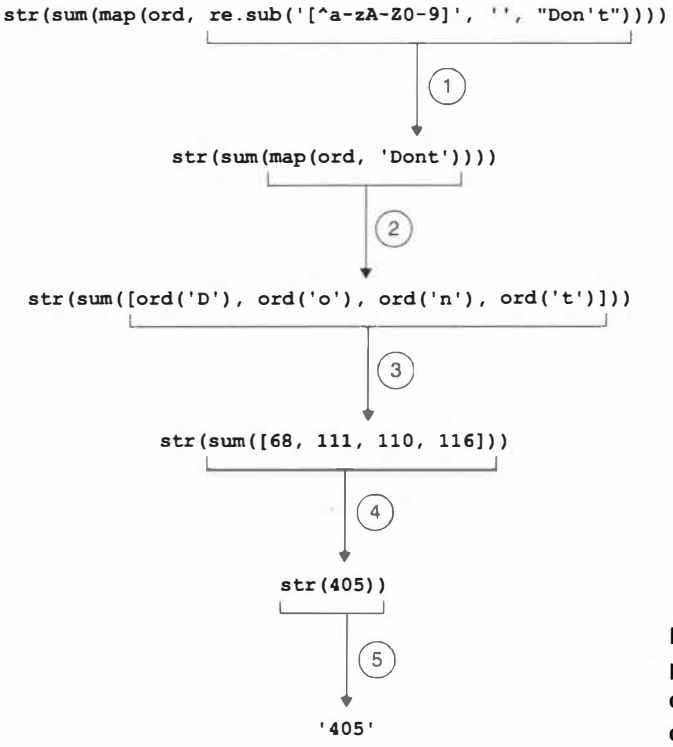


Рис. 18.7. Порядок операций по обработке текста в версии программы с функцией `map()`

На рис. 18.7 показан процесс обработки строки Don't в версии программы с функцией `map()`.

1. Функция `re.sub()` заменяет пустыми строками знаки вне символьного класса. Таким образом слово Don't превращается в Dont (без апострофа).
2. Функция `map()` применяет переданную ей функцию `ord()` к каждому элементу последовательности. В данном коде «последовательность» — это строка, поэтому она будет обрабатывать каждый символ слова.
3. В результате функция `map()` выдает новый список, где каждый символ слова Dont передан функции `ord()`.
4. Вызовы функций `ord()` результируют списком целых чисел, по одному на каждую букву.
5. Функция `sum()` сводит список чисел до одного значения, складывая их.
6. В конце концов функция результирует строкой, поэтому мы с помощью функции `str()` превращаем возвращенное функцией `sum()` значение в строковое представление числа.

18.3.2. Сортировка

Смысл данного упражнения не столько в изучении функций `ord()` и `chr()`, сколько в более глубоком понимании регулярных выражений, в обработке данных функциями и в представлении символов.

Так, при сортировке строк нужно учитывать регистр из-за позиций числовых значений символов (прописные буквы приведены в таблице ASCII перед строчными). Обратите внимание, что слова, начинающиеся с прописных букв, отсортировываются вперед слов со строчных:

```
>>> words = 'banana Apple Cherry anchovies cabbage Beets'
>>> sorted(words)
['Apple', 'Beets', 'Cherry', 'anchovies', 'banana', 'cabbage']
```

Так происходит потому, что числовые значения прописных букв меньше, чем у строчных. Чтобы отсортировать строки с учетом регистра, можно воспользоваться методом `key=str.casefold`. Функция `str.casefold()` возвращает «версию строки со сравнением значений без учета регистра». В коде ниже имя функции указано *без круглых скобок,* так как мы передаем *саму функцию* в качестве аргумента параметра `key`:

```
>>> sorted(words, key=str.casefold)
['anchovies', 'Apple', 'banana', 'Beets', 'cabbage', 'Cherry']
```

Если вы добавите скобки, будет вызвано исключение. Аналогичным способом мы передаем функциям в качестве аргументов функции `map()` и `filter()`:

```
>>> sorted(words, key=str.casefold())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor 'casefold' of 'str' object needs an argument
```

Похожий на метод `list.sort()` вариант — сортировка списка на месте:

```
>>> words.sort(key=str.casefold)
>>> words
['anchovies', 'Apple', 'banana', 'Beets', 'cabbage', 'Cherry']
```

Консольные инструменты типа программы `sort` обрабатывают данные одинаково из-за идентичного представления символов. К примеру, вот такое содержимое файла:

```
$ cat words.txt
banana
Apple
Cherry
anchovies
cabbage
Beets
```

программа `sort` на моем компьютере Mac* сортирует следующим образом:

```
$ sort words
Apple
Beets
Cherry
anchovies
banana
cabbage
```

* Программа `sort` из набора базовых системных утилит GNU Coreutils 8.30 на одном из моих компьютеров под управлением операционной системы Linux по умолчанию выполняет сортировку без учета регистра. А как ведет себя программа `sort` на вашем компьютере?

Сначала сортируются слова с прописной буквы, а затем со строчной. В руководстве к программе `sort` (команда `man sort`) указано, что нужно использовать флаг `-f` для сортировки без учета регистра:

```
$ sort -f words
anchovies
Apple
banana
Beets
cabbage
Cherry
```

18.3.3. Тестирование

Воспользуюсь моментом и расскажу, как я тестирую программы. Написав какую-нибудь функцию или программу, я провожу тест после каждого изменения, чтобы убедиться в отсутствии ошибок. Благодаря тестам я могу свободно и уверенно реорганизовывать код в своих программах, поскольку способен проверить их в любой момент. Обнаружив ошибку в коде, я сразу же пишу тест, чтобы убедиться, что ошибка действительно существует. Затем я исправляю ее и вновь тестирую код, проверяя, обработана ли ошибка. Так я могу быть уверен, что если случайно вновь допущу ту же ошибку, то при тестировании она выявится.

В данной книге я старался ограничить программы 100 строками. Частенько они разрастаются до тысяч строк, распределенных по десяткам модулей. Я рекомендую вам писать и использовать тесты, независимо от того, как вы программируете. Это отличная привычка, которую нужно привить себе на раннем этапе обучения, и она окажет вам бесценную услугу в будущем.

18.4. Прокачиваем навыки

— Скормите программе другие текстовые файлы и попробуйте найти еще слова, гематрия которых равна 666.
— Передав программе некоторый текст, найдите в результате вызова функции `word2num()` наиболее часто встречающееся значение и соответствующие им слова.
— Напишите версию программы с таблицей символов, в которой приведены ваши собственные числовые значения. Например, каждая буква может быть представлена числом, отражающим ее позицию

в алфавите, поэтому гематрии букв A и a будут равны 1, B и b — 2 и т. д. Или же можно вычислять любую согласную как 1, а гласную как –1. Придумайте собственную таблицу и напишите тесты, чтобы убедиться, что ваша программа работает так, как вам нужно.

РЕЗЮМЕ

— Функция `ord()` возвращает для указанного символа числовое значение, соответствующее позиции этого символа в таблице ASCII.
— Функция `chr()` возвращает для указанного числового значения соответствующий символ из таблицы ASCII.
— В регулярных выражениях можно использовать диапазоны символов, например `a-z`, если порядковые значения последовательности символов следуют по порядку друг за другом, как, например, в таблице ASCII.
— Функция `re.sub()` заменяет совпадающие по шаблону значения в строке новыми символами. Например, так можно заменить все знаки, не относящиеся к буквам, пустой строкой, чтобы удалить знаки препинания и пробелы.
— Функция `map()` может ссылаться на обычную функцию вместо анонимной, поскольку `map()` получает одно значение, и `ord()` ожидает то же самое.
— Функция `sum()` складывает числа из списка, сводя последовательность значений к одному. Можно вручную решить задачу с помощью метода `functools.reduce()`.
— Чтобы отсортировать строковые значения без учета регистра, используйте в функциях `sorted()` и `list.sort()` параметр `key=str.casefold`.

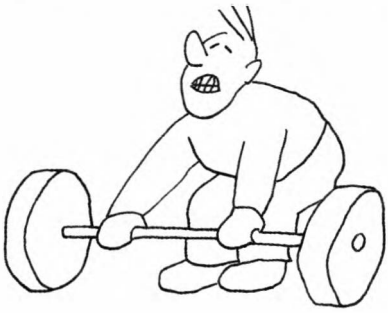
Глава 19

Тренировка дня: парсинг CSV-файлов и генерация текстовых таблиц

Несколько лет назад я присоединился к приверженцам здорового образа жизни. Мы встречаемся пару раз в неделю около фитнес-центра. Качаем мускулы и бегаем, откладывая смерть еще на денек. Вряд ли меня можно назвать образцовым спортсменом, тем не менее я нашел прекрасный способ потренироваться и повидаться с друзьями. Больше всего я люблю, когда тренер пишет на доске слова «Тренировка дня». Независимо от типа упражнений и нагрузки, я выполняю все. Неважно, хочу я сегодня двести раз отжаться или нет, я просто отжимаюсь, не обращая внимания на время*. Вдохновившись спортивным

* См. книгу Барри Шварца «Парадокс выбора. Как принимать решения, о которых мы не будем жалеть» (https://www.litres.ru/barri-shvarc/paradoks-vybora-kak-prinimat-resheniya-o-kotoryh-my-ne-budem/chitat-onlayn). Автор отмечает, что рост количества вариантов ведет к стрессу и в итоге к неудовлетворенности решением независимо от выбора. Примером может служить попытка выбрать мороженое в киоске, где представлены три вкуса: шоколадный, ванильный и клубничный. Если вы выберете шоколадное мороженое, то, скорее всего, окажетесь довольны своим выбором. А теперь представьте, что в магазине представлены 60 вкусов мороженого, в том числе 20 различных фруктовых шербетов и 12 различных вариаций шоколадного. Выбрав одну из вариаций последнего, вы будете переживать из-за 11 оставшихся, которые тоже могли бы выбрать. Иногда отсутствие выбора дает ощущение спокойствия. Можете называть это фатализмом или еще как-нибудь.

настроем, мы напишем программу *wod.py*, которая будет придумывать варианты «Тренировки дня» — набор упражнений для неукоснительного выполнения:



```
$ ./wod.py
Exercise                Reps
----------------------  ------
Pushups                   40
Plank                     38
Situps                    99
Hand-stand pushups         5
```

Примечание. При каждом запуске программы вы обязаны *неукоснительно выполнять все упражнения.* Да что говорить, даже просто *читая* их, вы уже обязываетесь их выполнять. *СРАЗУ ЖЕ.* Разговаривать не надо, не я устанавливаю правила. Приседайте до упада!

Мы выберем упражнения из списка, хранящегося в *текстовом файле с разделителями.* В данном случае в качестве «разделителя» используется запятая. Запятыми разделены значения в каждом поле. Файлы с данными, разделенными запятыми, часто упоминаются как файл в формате CSV (Comma-Separated Values — «значения, разделенные запятыми»). Обычно в первой строке файла указаны заголовки столбцов, а ниже располагаются строки с самими табличными данными:

```
$ head -3 inputs/exercises.csv
exercise, reps
Burpees,20-50
Situps,40-100
```

Освоив данную главу, вы научитесь:

— анализировать текстовые файлы с разделителями с помощью модуля `csv`;
— преобразовывать числа из текстового формата в числовой;
— выводить табличные данные с помощью модуля `tabulate`;
— выходить из ситуаций с отсутствующими и искаженными данными.

Эта и последующие главы будут существенно сложнее. Вам понадобится применять многие из навыков, полученных ранее, в предыдущих главах, так что приготовьтесь!

19.1. Создание файла *wod.py*

Создайте программу *wod.py* в каталоге *19_wod*. Измените ее код таким образом, чтобы при запуске с флагами `-h` и `--help` программа выводила следующие инструкции:

```
$ ./wod.py -h
usage: wod.py [-h] [-f FILE] [-s seed] [-n exercises] [-e]

Create Workout Of (the) Day (WOD)

optional arguments:
-h, --help            show this help message and exit
-f FILE, --file FILE  CSV input file of exercises (default: inputs/
                                                    exercises.csv)
-s seed, --seed seed  Random seed (default: None)
-n exercises, --num exercises
                        Number of exercises (default: 4)
-e, --easy            Halve the reps (default: False)
```

С помощью параметра `-f` или `--file` программа принимает текстовый файл, доступный для чтения (по умолчанию *inputs/exercises.csv*). Вывод будет содержать несколько видов упражнений, количество (видов) которых определяется с помощью параметра `-n` или `--num` (по умолчанию `4`). Еще при запуске может быть передан флаг `-e` или `--easy`, указывающий, что количество подходов в каждом упражнении должно быть уменьшено вдвое. Так как для выбора упражнений мы воспользуемся модулем `random`, понадобится опция `-s` или `--seed` (целое число с дефолтным значением `None`) для заглушки метода `random.seed()` в целях тестирования.

19.1.1. Чтение текстовых файлов с разделителями

Нам понадобится модуль `csv` для анализа входного файла. Это стандартный модуль, он уже должен быть установлен. Чтобы убедиться, что это так, откройте REPL-интерфейс `python3` и импортируйте данный модуль. Если команда выполнена успешно, модуль установлен:

```
>>> import csv
```

Мы собираемся рассмотреть еще два модуля, которые, вероятно, потребуется установить:

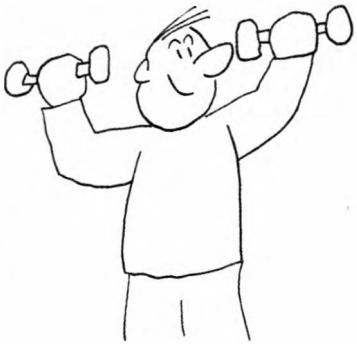
— модуль `csvkit` для обработки входного файла в консоли;
— модуль `tabulate` для форматирования результирующей таблицы.

Выполните показанную ниже команду, чтобы установить их:

```
$ python3 -m pip install csvkit tabulate
```

Еще можно воспользоваться файлом *requirements.txt*, в котором перечислены зависимости — необходимые модули для программы. Вместо предыдущей команды попробуйте установить все необходимые модули следующим образом:

```
$ python3 -m pip install -r requirements.txt
```



Несмотря на буквы csv в названии, модуль `csvkit` поддерживает не только CSV, но и многие другие форматы текстовых файлов с разделителями.

Кстати, в качестве разделителя вместо запятой используются и другие символы, например знак табуляции (`\t`). Модуль `csvkit` содержит множество инструментов, о которых можно прочитать в документации (**https://csvkit.readthedocs.io/en/1.1.0**). Я записал несколько файлов с разделителями в каталог *19_wod/inputs*, чтобы вы могли использовать их для тестирования своей программы.

После установки модуля `csvkit` попробуйте выполнить команду `csvlook`, чтобы преобразовать файл *inputs/exercises.csv* в таблицу, показанную ниже:

```
$ csvlook --max-rows 3 inputs/exercises.csv
| exercise | reps   |
| -------- | ------ |
| Burpees  | 20-50  |
| Situps   | 40-100 |
| Pushups  | 25-75  |
| ...      | ...    |
```

В столбце «reps» входного файла приведены два числа, разделенных дефисом, указывающих на количество подходов в каждом упражнении. Например, значение `10-20` означает «от 10 до 20 подходов». Чтобы выбрать конкретное количество из диапазона, вы воспользуетесь методом `random.randint()`. Данная функция применяется для выбора целочисленного значения из указанного диапазона. При запуске со значением `1` аргумента `--seed` и тремя видами упражнений вывод в вашем случае должен точно соответствовать показанному ниже:

```
$ ./wod.py --seed 1 --num 3
Exercise       Reps
----------  ------
Pushups          32
Situps           71
Crunches         27
```

При запуске программы с флагом `--easy` количество подходов уменьшится вдвое:

```
$ ./wod.py --seed 1 --num 3 --easy
Exercise       Reps
----------  ------
Pushups          16
Situps           35
Crunches         13
```

Опция `--file` по умолчанию ссылается на *файл inputs/exercises.csv*, но вы можете указать любой другой входной файл:

```
$ ./wod.py --file inputs/silly-exercises.csv
Exercise              Reps
----------------  ------
Hanging Chads           46
Squatting Chinups       46
Rock Squats             38
Red Barchettas          32
```

На рис. 19.1 показана диаграмма, которая поможет понять, что происходит с данными в программе.

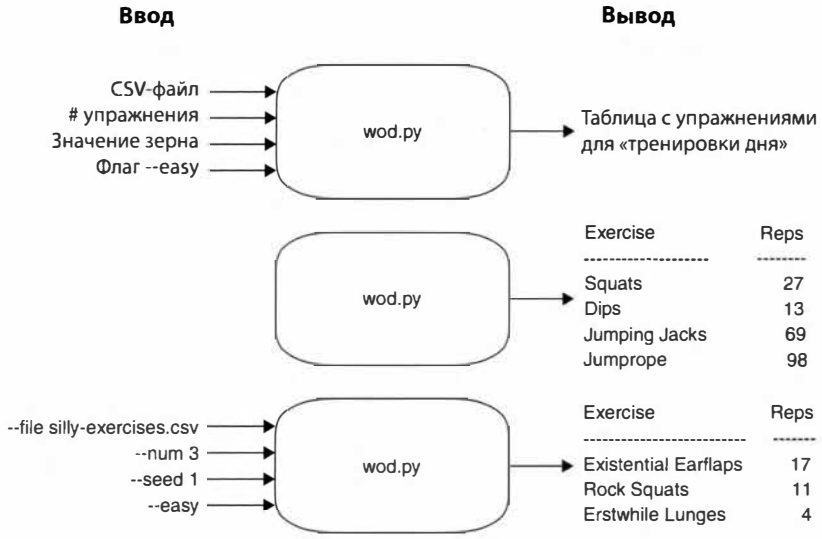


Рис. 19.1. Программа wod.py будет случайным образом выбирать количество подходов и виды упражнений из CSV-файла и формировать таблицу с содержимым «тренировки дня»

19.1.2. Чтение CSV-файла вручную

Сначала я покажу вам, как преобразовывать записи из CSV-файла в списки словарей вручную, а затем научу пользоваться модулем `csv`, позволяющим выполнять эту операцию гораздо быстрее. Мы планируем обратиться к словарям, поскольку для каждой тренировки нам нужно выводить виды упражнений и количество подходов (сколько раз повторять то или иное упражнение). Необходимо разбить количество подходов на малые и большие значения, чтобы сформулировать диапазон, из которого в случайном порядке будем выбирать конкретное количество подходов. Наконец, мы случайным образом выберем несколько упражнений и количество подходов, чтобы составить тренировку. Ого, даже просто описать задачу — все равно что сходить в фитнес-центр!

Обратите внимание, что количество подходов указано в виде диапазона чисел от минимального к максимальному с дефисом в виде разделителя:

```
$ head -3 inputs/exercises.csv
exercise, reps
Burpees,20-50
Situps,40-100
```

Удобно составить из этих данных список словарей, в котором заголовки столбцов в первой строке объединены с соответствующим значением в строках данных, например, так:

```
$ ./manual1.py
[{'exercise': 'Burpees', 'reps': '20-50'},
 {'exercise': 'Situps', 'reps': '40-100'},
 {'exercise': 'Pushups', 'reps': '25-75'},
 {'exercise': 'Squats', 'reps': '20-50'},
 {'exercise': 'Pullups', 'reps': '10-30'},
 {'exercise': 'Hand-stand pushups', 'reps': '5-20'},
 {'exercise': 'Lunges', 'reps': '20-40'},
 {'exercise': 'Plank', 'reps': '30-60'},
 {'exercise': 'Crunches', 'reps': '20-30'}]
```

Вы можете подумать, что излишне использовать словарь для записей, содержащих всего два столбца, но, поверьте мне, я регулярно имею дело с таблицами, состоящими из десятков и даже *сотен* столбцов, и вот тогда имена полей имеют важнейшее значение. Словарь — реально единственный толковый способ обработки большинства форматов текстовых файлов с разделителями, поэтому полезно изучить его на таком небольшом примере, как эта программа.

Давайте взглянем на код из файла *manual1.py*, решающий нашу задачу:

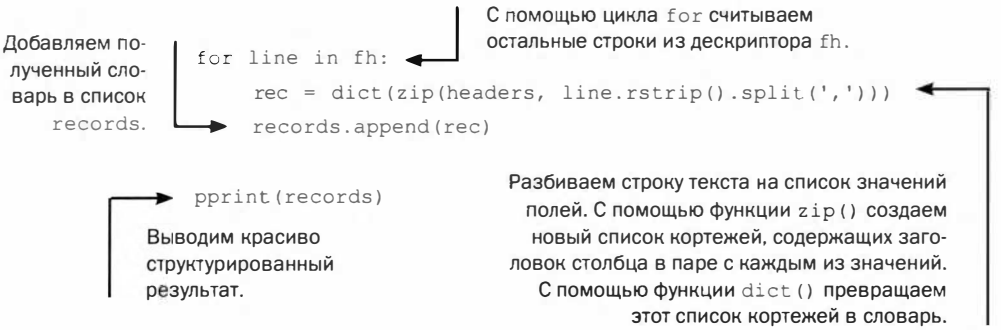
С помощью метода `fh.readline()` считываем первую строку файла. Удаляем пробельный символ с правой стороны (`str.rstrip()`) строки, а затем, используя модуль `str.split()`, нарезаем результирующую строку по запятым, чтобы создать список строк, являющихся заголовками столбцов.

Воспользуемся модулем `pprint` для вывода данных в структурированном виде.

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

from pprint import pprint

with open('inputs/exercises.csv') as fh:
    headers = fh.readline().rstrip().split(',')
    records = []
```

Инициализируем пустой список `records`.

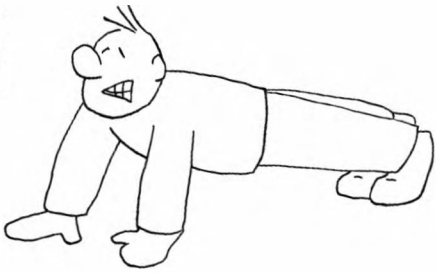
С помощью метода `fh.readline()` считываем первую строку файла. Удаляем пробельный символ с правой стороны (`str.rstrip()`) строки, а затем, используя модуль `str.split()`, нарезаем результирующую строку по запятым, чтобы создать список строк, являющихся заголовками столбцов.



Давайте еще немного покопаемся в данном коде. Сначала мы открываем файл и считываем первую строку:

```
>>> fh = open('exercises.csv')
>>> fh.readline()
'exercise, reps\n'
```

Строку все еще сопровождает символ перевода строки, поэтому мы применяем функцию `str.rstrip()`, чтобы удалить его:



```
>>> fh = open('exercises.csv')
>>> fh.readline().rstrip()
'exercise, reps'
```

Примечание. Обратите внимание, что мне нужно постоянно открывать данный файл для приведения примеров и при каждом вызове метода `fh.readline()` для считывания каждой последующей строки текста.

Теперь с помощью метода `str.split()` разделяем полученную строку по запятым с целью формирования списка строк:

```
>>> fh = open('exercises.csv')
>>> headers = fh.readline().rstrip().split(',')
>>> headers
['exercise', 'reps']
```

Таким же образом считываем следующую строку из файла, чтобы получить из полей список, состоящий из названия упражнения и количества подходов:

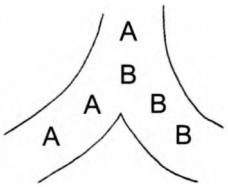
```
>>> line = fh.readline().rstrip().split(',')
>>> line
['Burpees', '20-50']
```

Затем используем функцию `zip()` для слияния двух списков в один, в котором элементы каждого списка сочетаются со своими аналогами в тех же позициях. Возможно, с ходу непонятно, но подумайте об окончании свадебной церемонии, когда жених и невеста поворачиваются лицом к собравшимся гостям. Обычно они берутся за руки и начинают идти по проходу, чтобы покинуть церемонию. Представьте себе трех друзей жениха (`G`) и трех подружек невесты (`B`), стоящих с двух сторон лицом друг к другу:

```
>>> groomsmen = 'G' * 3
>>> bridesmaids = 'B' * 3
```

Взяв две строки, каждая из которых содержит три человека, мы получим одну строку, содержащую три пары:

```
>>> pairs = list(zip(groomsmen, bridesmaids))
>>> pairs
[('G', 'B'), ('G', 'B'), ('G', 'B')]
>>> len(pairs)
3
```



Или представьте две дороги, сливающиеся в один поток на выезде с парковки. Как правило, одна машина с одной дороги (скажем, «A») вклинивается в поток, следом это делает машина с другой дороги (скажем, «B»). Автомобили входят в поток, как зубья застежки-молнии, и в результате чередуются «A», «B», «A», «B» и т. д.

Функция `zip()` помещает элементы списков в кортежи, группируя сначала все элементы в первой позиции, затем во второй и т. д., как показано на рис. 19.2. Обратите внимание, что это еще один представитель семейства *ленивых* функций, поэтому в REPL-интерфейсе мне понадобится функция `list()`, чтобы добиться от `zip()` результата:

```
>>> list(zip('abc', '123'))
[('a', '1'), ('b', '2'), ('c', '3')]
```

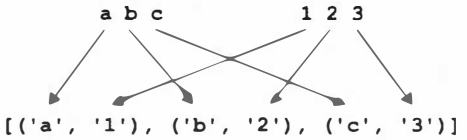


Рис. 19.2. Функция `zip()` объединяет содержимое двух списков в новый список с парами элементов

Функция `zip()` способна обрабатывать более двух списков. Отмечу, что при использовании списков разной длины она создает группы на основе самого короткого списка. В следующем примере первые два списка содержат четыре элемента («abcd» и «1234»), а третий — только три («xyz»), поэтому создаются только три кортежа:

```
>>> list(zip('abcd', '1234', 'xyz'))
[('a', '1', 'x'), ('b', '2', 'y'), ('c', '3', 'z')]
```

В нашем случае функция `zip()` объединяет заголовок столбца «exercise» со значением «Burpees» из второй строки, а затем заголовок столбца «reps» со вторым значением из второй строки — «20–50» (рис. 19.3):

```
>>> list(zip(headers, line))
[('exercise', 'Burpees'), ('reps', '20-50')]
```

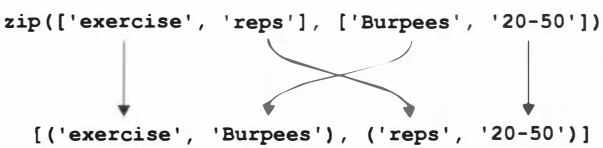


Рис. 19.3. Объединение заголовков столбцов и значений данных в список кортежей

Таким образом создается список кортежей.

Вместо функции `list()` мы можем использовать функцию `dict()` для создания словаря:

```
>>> rec = dict(zip(headers, line))
>>> rec
{'exercise': 'Burpees', 'reps': '20-50'}
```

Напомню, что с помощью метода `dict.items()` словарь превращается в список кортежей («ключ/значение»), поэтому эти две структуры данных вполне можно считать взаимозаменяемыми:

```
>>> rec.items()
dict_items([('exercise', 'Burpees'), ('reps', '20-50')])
```

Мы способны значительно сократить код, заменив цикл `for` представлением списка:

```
with open('inputs/exercises.csv') as fh:
    headers = fh.readline().rstrip().split(',')
    records = [dict(zip(headers, line.rstrip().split(',')))
               for line in fh]
    pprint(records)
```

Все еще необходимо разбить заголовки столбцов, прочитав первую строку.

Объединили три строки кода из цикла `for` в одну строку с представлением списка.

Или использовать функцию `map()` для написания эквивалентного кода:

```
with open('inputs/exercises.csv') as fh:
    headers = fh.readline().rstrip().split(',')
    mk_rec = lambda line: dict(zip(headers,
             line.rstrip().split(',')))
    records = map(mk_rec, fh)
    pprint(list(records))
```

Линтер Flake8 будет ругаться на присвоение анонимной функции. Обычно я разрабатываю код так, чтобы не получать предупреждений, но в данном случае не согласен с жалобой линтера. Я обожаю писать однострочные функции с присвоением лямбда-выражений.

Далее я расскажу, как использовать модуль `csv` для выполнения львиной части операций показанного в данном разделе кода, поэтому у вас наверняка возникнет вопрос, зачем я терроризировал вас ручной обработкой табличных данных. Дело в том, что довольно часто приходится обрабатывать отвратительно отформатированные данные, где, к примеру, первая строка не содержит заголовки столбцов или между строкой с заголовками и фактическими данными размещены строки с какой-то другой информацией. Если вам случится поработать с таким же количеством ужасно отформатированных файлов Excel, с каким довелось иметь дело мне, вы, несомненно, поймете, что иногда просто нет другого выбора, кроме как обрабатывать файл вручную.

19.1.3. Парсинг с помощью модуля csv

Описанный способ анализа текстовых файлов с разделителями чрезвычайно распространен, и нет смысла писать или копировать некий код каждый раз, когда требуется обработка файлов. На помощь приходит шикарный модуль `csv`, устанавливаемый вместе с Python и очень изящно избавляющий нас от головной боли.

Давайте посмотрим, как изменится код, если воспользоваться методом `csv.DictReader()` (файл *using_csv1.py* в репозитории):

```
#!/usr/bin/env python3
import csv
from pprint import pprint

with open('inputs/exercises.csv') as fh:
    reader = csv.DictReader(fh, delimiter=',')
    records = []
    for rec in reader:
        records.append(rec)

    pprint(records)
```

Импортируем модуль `csv`.

Применяем метод `csv.DictReader()`, создающий словарь для каждой записи в файле. Он объединяет заголовки столбцов в первой строке со значениями данных в последующих строках. Модуль использует разделитель для извлечения строк и разбиения столбцов текста.

Инициализируем пустой список `records` для хранения записей.

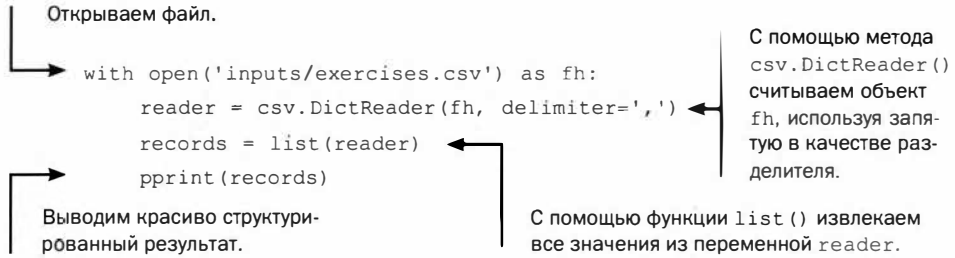
С помощью цикла `for` перебираем все записи, возвращаемые переменной `records`.

Получается словарь `records`, добавленный к списку `records`.

Показанный ниже код создает тот же список словарей, что и раньше, но содержит гораздо меньше строк. Обратите внимание, что каждая запись сопровождается словом `OrderedDict`, то есть упорядоченным словарем, в котором ключи хранятся в порядке их добавления:

```
$ ./using_csv1.py
[OrderedDict([('exercise', 'Burpees'), ('reps', '20-50')]),
 OrderedDict([('exercise', 'Situps'), ('reps', '40-100')]),
 OrderedDict([('exercise', 'Pushups'), ('reps', '25-75')]),
 OrderedDict([('exercise', 'Squats'), ('reps', '20-50')]),
 OrderedDict([('exercise', 'Pullups'), ('reps', '10-30')]),
 OrderedDict([('exercise', 'Hand-stand pushups'), ('reps', '5-20')]),
 OrderedDict([('exercise', 'Lunges'), ('reps', '20-40')]),
 OrderedDict([('exercise', 'Plank'), ('reps', '30-60')]),
 OrderedDict([('exercise', 'Crunches'), ('reps', '20-30')])]
```

Мы можем полностью удалить цикл `for` и использовать функцию `list()`, чтобы с помощью переменной `reader` получить тот же список. Данный код (см. файл *using_csv2.py*) выведет идентичный результат:



19.1.4. Функция для чтения CSV-файла

Давайте попробуем написать и протестировать функцию read_csv() для чтения входных данных. Начнем с определения функции-заглушки `read_csv()` (без рабочего кода) и функции `test_read_csv()`:

```
def read_csv(fh):
    """Чтение CSV-файла"""
    pass


def test_read_csv():
    """Test read_csv"""
    text = io.StringIO('exercise, reps\nBurpees,20-50\nSitups,40-100')
    assert read_csv(text) == [('Burpees', 20, 50),
                              ('Situps', 40, 100)]
```

Подтверждаем, что наша функция `read_csv()` превратит данный текст в список кортежей с названием упражнения и количеством подходов в диапазоне от минимального к максимальному значению. Обратите внимание, что данные значения преобразованы в целые числа.

С помощью метода `io.StringIO()` создаем фиктивный файловый дескриптор, чтобы обернуть допустимый текст, который мы могли бы прочитать из файла. Символы перевода строки, `\n`, разделяют строки во входных данных, а еще в каждой строке используются запятые для разделения значений в полях. Мы уже применяли метод `io.StringIO()` в главе 5 в версии программы для малых объемов памяти.

Итак, мы только что в поте лица составляли список словарей, так почему же теперь нужно создавать список кортежей?! Ну что сказать, так как мы планируем воспользоваться модулем `tabulate` для вывода результата, просто доверьтесь мне. Я знаю, что делаю!

Вернемся к анализу файла с помощью метода `csv.DictReader()` и подумаем о том, как превратить количество подходов в целые числа в диапазоне от минимального к максимальному значению:

```
reader = csv.DictReader(fh, delimiter=',')
exercises = []
for rec in reader:
    name, reps = rec['exercise'], rec['reps']
    low, high = 0, 0 # что здесь происходит?
    exercises.append((name, low, high))
```

В нашем распоряжении несколько способов. Представим количество подходов следующим образом:

```
>>> reps = '20-50'
```

Функция `str.split()` может разбить содержимое переменной `reps` на две строки: «20» и «50»:

```
>>> reps.split('-')
['20', '50']
```

Как вы превратите эти строки в целые числа?

Еще один способ, которым можно воспользоваться, — использовать регулярное выражение. В нем шаблон `\d` будет соответствовать любой цифре, а шаблон `\d+` — одной или более цифр. (Вернитесь к главе 15, чтобы освежить в памяти, как используется краткая запись `\d` символьного класса для выборки цифр.) Вы можете заключить данное выражение в круглые скобки, чтобы захватить и минимальное, и максимальное количество подходов:

```
>>> match = re.match('(\d+)-(\d+)', reps)
>>> match.groups()
('20', '50')
```

Сумеете написать функцию `read_csv()`, код которой проходит тест `test_read_csv()`, показанный в начале раздела?

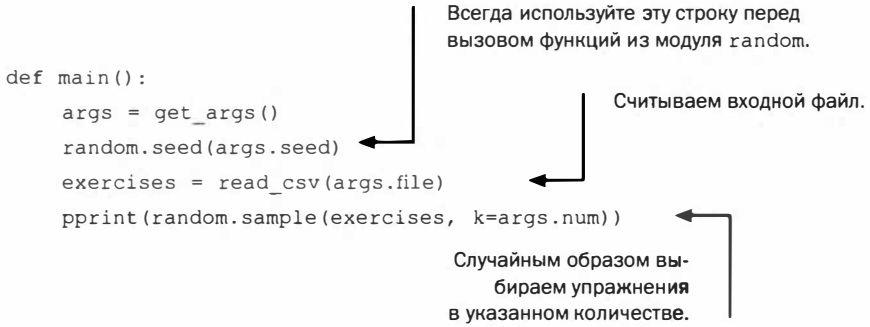
19.1.5. Выбор упражнений

Надеюсь, что к этому моменту вы уже написали код `get_args()`, а функция `read_csv()` успешно проходит тестирование. Теперь обратимся к методу `main()` и займемся выводом структурированных данных:

Если вы выполните показанный выше код, то получите следующий результат:

```
$ ./wod.py
[('Burpees', 20, 50),
 ('Situps', 40, 100),
 ('Pushups', 25, 75),
 ('Squats', 20, 50),
 ('Pullups', 10, 30),
 ('Hand-stand pushups', 5, 20),
 ('Lunges', 20, 40),
 ('Plank', 30, 60),
 ('Crunches', 20, 30)]
```

Воспользуемся функцией `random.sample()` для выбора количества видов упражнений, указанного пользователем с помощью опции `--num`. Импортируйте модуль `random` и измените код метода `main` следующим образом:



Теперь вместо того, чтобы выводить все доступные упражнения, программа выведет корректное количество видов упражнений, выбранных случайным образом. Напоминаю, ваш результат будет точно соответствовать моему, если вы укажете такое же значение зерна:

```
$ ./wod.py -s 1
[('Pushups', 25, 75),
 ('Situps', 40, 100),
 ('Crunches', 20, 30),
 ('Burpees', 20, 50)]
```

Необходимо перебрать подборку и выбрать одно значение «подходов» с помощью функции `random.randint()`. Первое упражнение — отжимания, количество подходов — от 25 до 75:

```
>>> import random
>>> random.seed(1)
>>> random.randint(25, 75)
33
```

Если переменной `args.easy` присвоено значение `True`, вам нужно уменьшить это значение вдвое. Как вы понимаете, количество подходов не может быть представлено вещественным числом:

```
>>> 33/2
16.5
```

Воспользуемся функцией `int()` для округления вещественного числа до целого:

```
>>> int(33/2)
16
```

19.1.6. Форматирование вывода

Модифицируйте свой код до тех пор, пока программа не выведет следующий результат:

```
$ ./wod.py -s 1
[('Pushups', 56), ('Situps', 88), ('Crunches', 27), ('Burpees', 35)]
```

Давайте воспользуемся функцией `tabulate()` из модуля `tabulate` для форматирования полученного списка кортежей в виде текстовой таблицы:

```
>>> from tabulate import tabulate
```

```
>>> wod = [('Pushups', 56), ('Situps', 88), ('Crunches', 27),
           ('Burpees', 35)]
>>> print(tabulate(wod))
--------  --
Pushups   56
Situps    88
Crunches  27
Burpees   35
--------  --
```

Выполнив команду `help(tabulate)`, вы узнаете, что в функции доступна опция `headers`, позволяющая указать список строк, которые следует использовать в качестве заголовков столбцов:

```
>>> print(tabulate(wod, headers=('Exercise', 'Reps')))
Exercise       Reps
----------  ------
Pushups          56
Situps           88
Crunches         27
Burpees          35
```

Если вы обдумаете и реализуете эти идеи, ваша программа пройдет все необходимые тесты.

19.1.7. Обработка плохих данных

Ни один из тестов не покажет, что вашей программе переданы плохие данные, поэтому я записал несколько «неправильных» CSV-файлов в каталог *19_wod/inputs*. Вам понравится придумывать способы их обработки.

- У файла *bad-headers-only.csv* верный формат, но в нем нет фактических данных. Только заголовки столбцов.
- Файл *bad-empty.csv* девственно чист. Он имеет нулевой размер. Я создал его с помощью команды `touch bad-empty.csv`, и в нем вообще нет никаких данных.
- Заголовки в файле *bad-headers.csv* написаны с прописной буквы, соответственно, вместо слова reps указано Reps и Exercise — вместо exercise.

— В файле *bad-delimiter.tab* в качестве разделителя в полях вместо запятой (`,`) используется символ табуляции (`\t`).
— Столбец «reps» в файле *bad-reps.csv* содержит неправильные данные: либо отсутствует диапазон, либо они не являются числовыми или целочисленными значениями.

После того, как ваша программа пройдет все тесты, запустите ее с «неправильными» файлами, чтобы посмотреть на ее поведение. Как должна поступить программа, если нет допустимых данных? Должна ли она выводить сообщения об ошибках, когда встречает недопустимые или отсутствующие значения, или же ей нужно спокойно игнорировать ошибки и выводить только обнаруженные корректные данные? Все это проблемы из жизни, с которыми вы наверняка столкнетесь, и вам думать, что стоит делать программе. Обсудив решение, я расскажу вам, как сам поступаю с такими файлами.

19.1.8. Самостоятельная работа

Ладно, оставим разговоры. Пора писать программу. С вас 10 отжиманий за каждую допущенную ошибку!

Традиционно несколько советов:

— для анализа входных CSV-файлов используется метод `csv.DictReader()`;
— разбивайте содержимое поля «reps» по дефисам, приведите минимальное и максимальное количество подходов к целым числам, а затем используйте метод `random.randint()`, чтобы выбрать случайное целое число в полученном диапазоне;
— с помощью `random.sample()` выберите количество видов упражнений;
— используя модуль `tabulate`, отформатируйте вывод и представьте его в виде текстовой таблицы.

19.2. Решение

Как успехи? Удалось написать программу, способную изящно обрабатывать «неправильные» входные файлы?

Давайте взглянем на мое решение.

```
#!/usr/bin/env python3
"""Тренировка дня"""
import argparse
import csv
import io
import random
from tabulate import tabulate

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Create Workout Of (the) Day (WOD)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-f',
                        '--file',
                        help='CSV input file of exercises',
                        metavar='FILE',
                        type=argparse.FileType('rt'),
                        default='exercises.csv')

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='seed',
                        type=int,
                        default=None)

    parser.add_argument('-n',
                        '--num',
                        help='Number of exercises',
                        metavar='exercises',
                        type=int,
                        default=4)

    parser.add_argument('-e',
                        '--easy',
                        help='Halve the reps',
                        action='store_true')
                        args = parser.parse_args()
```

Импортируем функцию `tabulate()` из модуля `tabulate`, благодаря которой мы отформатируем результирующую таблицу.

С помощью опции `--file` указывается текстовый файл, доступный для чтения.

```
    if args.num < 1:
        parser.error(f'--num "{args.num}" must be greater than 0')

    return args


# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    random.seed(args.seed)
    wod = []
    exercises = read_csv(args.file)

    for name, low, high in random.sample(exercises, k=args.num):
        reps = random.randint(low, high)
        if args.easy:
            reps = int(reps / 2)
        wod.append((name, reps))

    print(tabulate(wod, headers=('Exercise', 'Reps')))


# --------------------------------------------------
def read_csv(fh):
    """Чтение CSV-файла"""

    exercises = []
    for row in csv.DictReader(fh, delimiter=','):
        low, high = map(int, row['reps'].split('-'))
        exercises.append((row['exercise'], low, high))

    return exercises
```

Проверяем, что `args.num` присвоено положительное значение.

Инициализируем пустой список `wod`.

Считываем содержимое входного файла в список `exercises`.

Случайным образом выбираем количество подходов из указанного диапазона.

Если значение `args.easy` «истинно», сокращаем количество подходов вдвое.

Добавляем в список `wod` кортеж, содержащий название упражнения и количество подходов.

С помощью функции `tabulate()` форматируем список `wod` в текстовую таблицу, используя соответствующие заголовки.

Произвольно выбираем виды упражнений в заданном количестве. Результатом становится список кортежей, каждый из которых содержит три значения, распаковываемых в переменные `name`, `low` и `high`.

Определяем функцию для чтения открытого дескриптора CSV-файла.

Итерируем файловый дескриптор с помощью метода `csv.DictReader()` и генерируем словарь, объединяющий заголовки столбцов из первой строки с фактическими значениями полей из остальных строк. В качестве разделителя в полях используется символ , (запятая).

Инициализируем пустой список `exercises`.

Добавляем кортеж, содержащий название упражнения с минимальным и максимальным количеством подходов.

Возвращаем вызывающей стороне список `exercises`.

Разделяем диапазон «подходов» по дефису, преобразовываем полученные значения в целые числа и присваиваем переменным `low` и `high`.

```
# --------------------------------------------------
def test_read_csv():
    """Test read_csv"""

    text = io.StringIO('exercise, reps\nBurpees,20-50\nSitups,40-100')
    assert read_csv(text) == [('Burpees', 20, 50),
                              ('Situps', 40, 100)]


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Определяем функцию, используемую Pytest для тестирования функции `read_csv()`.

Создаем фиктивный файловый дескриптор, содержащий допустимые демонстрационные данные.

Проверяем, что функция `read_csv()` способна обрабатывать допустимые входные данные.

19.3. Обсуждение

Почти половину исходного кода программы занимает функция `get_args()`! Несмотря на то что весь код данной функции вам уже знаком, я обращаю ваше внимание, насколько много работы выполняется программой для проверки входных данных, предоставления дефолтных значений, вывода инструкций для пользователя и т. п. Давайте разберем код программы, начиная с функции `read_csv()`.

19.3.1. Чтение CSV-файла

Ранее в этой главе я предлагал вам написать код для разделения минимального/максимального значений диапазона подходов из столбца «reps» и преобразования результата в целые числа. Покажу один из способов:

```
def read_csv(fh):
    exercises = []
    for row in csv.DictReader(fh, delimiter=','):
        low, high = map(int, row['reps'].split('-'))
        exercises.append((row['exercise'], low, high))

    return exercises
```

Разделяем диапазон «подходов» в поле «reps» по дефису, преобразовываем с помощью функции `int()` полученные значения в целые числа и присваиваем переменным `low` и `high`.

В сопровожденной выноской строке кода происходит следующее. Предположим, что диапазон «подходов» выглядит так:

```
>>> '20-50'.split('-')
['20', '50']
```

Необходимо преобразовать каждое значение диапазона «подходов» в целое число, чем и займется функция `int()`. Можно использовать представление списка:

```
>>> [int(x) for x in '20-50'.split('-')]
[20, 50]
```

Но вариант с функцией `map()`, на мой взгляд, намного короче и понятнее:

```
>>> list(map(int, '20-50'.split('-')))
[20, 50]
```

Поскольку мы получаем два значения, мы можем присвоить их двум переменным:

```
>>> low, high = map(int, '20-50'.split('-'))
>>> low, high
(20, 50)
```

19.3.2. Вероятные ошибки во время выполнения

Код нашей программы весьма требователен к входным данным, и если они не соответствуют ожиданиям, ее запуск может с треском провалиться. К примеру, что произойдет, если в поле «reps» диапазон «подходов» не будет содержать дефис? Интерпретатор выдаст одно значение:

```
>>> list(map(int, '20'.split('-')))
[20]
```

Во *время выполнения,* когда программа попробует присвоить одно значение двум переменным, мы получим исключение:

```
>>> low, high = map(int, '20'.split('-'))
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 2, got 1)
```

А если одно или несколько значений не могут быть приведены к типу `int`? Это также вызовет исключение, и, опять же, вы не обнаружите проблему, пока не запустите программу с «неправильными» данными:

```
>>> list(map(int, 'twenty-thirty'.split('-')))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'twenty'
```

Что произойдет, если поле «reps» пусто или заголовки столбцов указаны с прописной буквы?

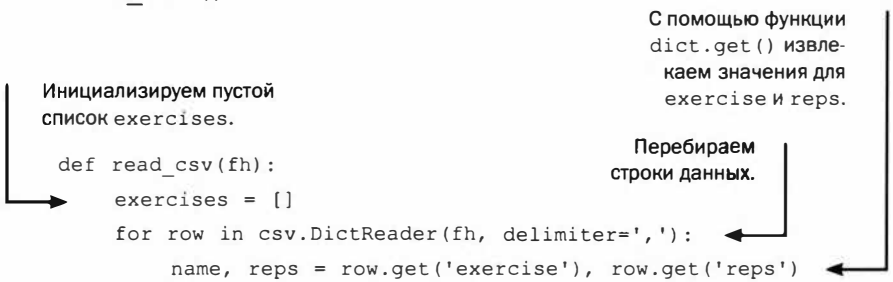
```
>>> rec = {'Exercise': 'Pushups', 'Reps': '20-50'}
```

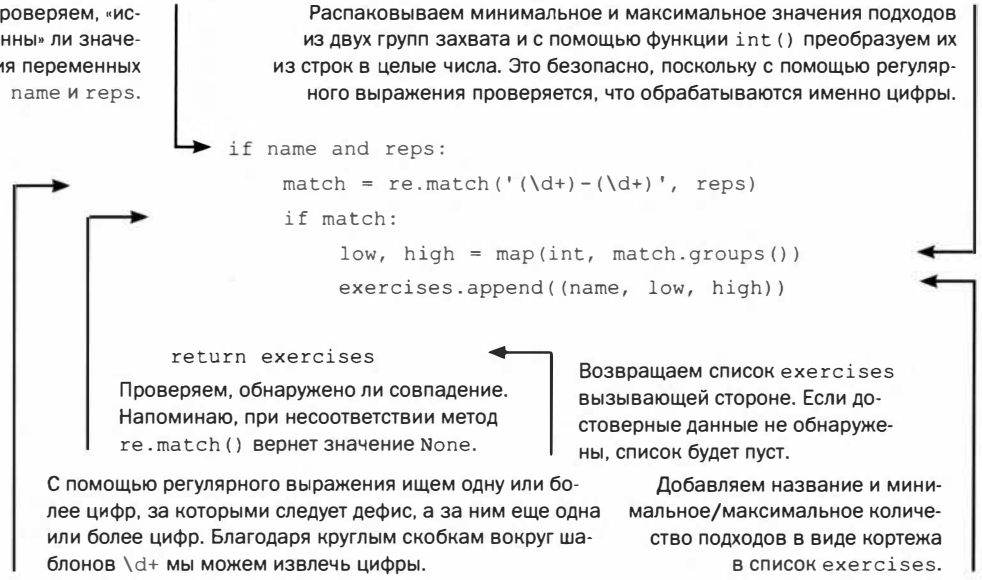
Доступ к словарю `rec['reps']` вызовет исключение:

```
>>> list(map(int, rec['reps'].split('-')))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'reps'
```

Предполагается, что функция `read_csv()` работает нормально, пока мы передаем ей «правильные» данные, но в реальности предоставляемые данные далеко не всегда идеальны. К сожалению, огромная часть моей работы заключается в поиске и исправлении подобных ошибок.

Ранее в этой главе я предложил использовать регулярное выражение для извлечения минимального/максимального значений диапазона подходов из поля «reps». Преимущество регулярного выражения состоит в том, что оно проверяет все поле, гарантируя, что данные в нем «правильные». Покажу более надежный способ реализации функции `read_csv()`:





19.3.3. Анализ файла с помощью метода pandas.read_csv()

Многим людям, знакомым со статистикой и наукой о данных, скорее всего, известен модуль `pandas`, имитирующий многие концепции из языка программирования R. Я специально выбрал имя функции `read_csv()`, так как оно похоже на встроенную в R функцию `read.csv`, которая, в свою очередь, использовалась в качестве модели для функции `pandas.read_csv()`. И R, и модуль `pandas` обрабатывают данные из файлов с разделителями/CSV в виде «фрейма данных» — двумерного объекта, позволяющего вам работать со столбцами и строками.

Чтобы запустить программу *using_pandas.py*, необходимо установить модуль `pandas`:

```
$ python3 -m pip install pandas
```

Теперь можно запустить программу:

```
import pandas as pd
df = pd.read_csv('inputs/exercises.csv')
print(df)
```

Вы увидите такой вывод:

```
$ ./using_pandas.py
              exercise     reps
0              Burpees    20-50
1                Situps   40-100
2              Pushups    25-75
3               Squats    20-50
4              Pullups    10-30
5   Hand-stand pushups     5-20
6                Lunges    20-40
7                 Plank    30-60
8             Crunches    20-30
```

Обучение использованию модуля `pandas` выходит за рамки данной книги. Я лишь хотел упомянуть, что есть крайне популярный способ анализа текстовых файлов с разделителями, особенно эффективный в случае, если вам необходим статистический анализ различных столбцов данных.

19.3.4. Форматирование таблицы

Взгляните на функцию `main()` из моего решения. Видите «спящее» исключение во время выполнения?

```
def main():
    args = get_args()
    random.seed(args.seed)
    wod = []
    exercises = read_csv(args.file)

    for name, low, high in random.sample(exercises, k=args.num):
reps = random.randint(low, high)
        if args.easy:
            reps = int(reps / 2)
        wod.append((name, reps))

    print(tabulate(wod, headers=('Exercise', 'Reps')))
```

Благодаря данной строке программа завершит работу с ошибкой, если значение `args.num` превысит количество элементов в списке `exercises`. Исключение будет выброшено, если, к примеру, функция `read_csv()` вернет значение `None` или пустой список.

Попробовав протестировать данную программу с файлом *bad-headers-only.csv*, вы увидите следующую ошибку:

```
$ ./wod.py -f inputs/bad-headers-only.csv
Traceback (most recent call last):
```

```
  File "./wod.py", line 93, in <module>
    main()
  File "./wod.py", line 62, in main
    for name, low, high in random.sample(exercises, k=args.num):
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/
        python3.8/rando
    m.py", line 363, in sample
    raise ValueError("Sample larger than population or is
                     negative")
ValueError: Sample larger than population or is negative
```

Более безопасный способ решить проблему — проверить, что функция `read_csv()` возвращает достаточно данных для передачи методу `random.sample()`. Существует вероятность двух возможных ошибок:

— во входном файле не найдено допустимых для обработки данных;
— выбирается слишком много записей из файла.

Показываю один из способов справиться с данными проблемами. Напоминаю, что вызов метода `sys.exit()` со строковым значением вынудит программу вывести сообщение в поток `sys.stderr` и завершить работу с кодом `1` (то есть с ошибкой):

```
def main():
    """Да грянет джаз!"""

    args = get_args()
    random.seed(args.seed)
    exercises = read_csv(args.file)

    if not exercises:
        sys.exit(f'No usable data in --file "{args.file.name}"')

    num_exercises = len(exercises)
    if args.num > num_exercises:
        sys.exit(f'--num "{args.num}" > exercises "{num_exercises}"')

    wod = []
    for name, low, high in random.sample(exercises, k=args.num):
        reps = random.randint(low, high)
```

Считываем входной файл в переменную `exercises`. Функция должна вернуть только список, вероятно, пустой.

Проверяем, не «ложно» ли значение переменной `exercises`, что может быть, к примеру, в случае присваивания ей пустого списка.

Проверяем, не слишком ли много записей выбирается.

Убедившись, что у нас достаточно достоверных данных, продолжаем работу.

```
        if args.easy:
            reps = int(reps / 2)
        wod.append((name, reps))

    print(tabulate(wod, headers=('Exercise', 'Reps')))
```

Версия программы в файле *solution2.py* содержит указанные обновленные функции и изящно обрабатывает все «неправильные» входные файлы. Обратите внимание, что я переместил функцию `test_read_csv()` в файл *unit.py*, потому что она стала намного длиннее из-за включения проверок различных «неправильных» входных данных.

Выполните команду `pytest -xv unit.py` для проведения модульного тестирования. Давайте разберем файл *unit.py*, содержащий более строгую схему тестирования:

```
import io
from wod import read_csv

def test_read_csv():
    """Test read_csv"""

    good = io.StringIO('exercise, reps\nBurpees,
                       20-50\nSitups, 40-100')
    assert read_csv(good) == [('Burpees', 20, 50),
                              ('Situps', 40, 100)]

    no_data = io.StringIO('')
    assert read_csv(no_data) == []

    headers_only = io.StringIO('exercise, reps\n')
    assert read_csv(headers_only) == []

    bad_headers = io.StringIO('Exercise, Reps\nBurpees,
                              20-50\nSitups,40-100')
    assert read_csv(bad_headers) == []

    bad_numbers = io.StringIO('exercise, reps\nBurpees,
                              20-50\nSitups, forty-100')
```

Помните, что можно импортировать функции из собственных модулей в другие программы. Здесь мы импортируем нашу функцию `read_csv()`. Если вместо данной строки указать код `import wod`, вызывать нашу функцию пришлось бы иначе — `wod.read_csv()`.

Исходный, допустимый ввод.

Тестирование без данных.

Файл правильного формата (правильные заголовки и разделитель), но без данных.

Заголовки написаны с прописной буквы, а ожидаются только строчные.

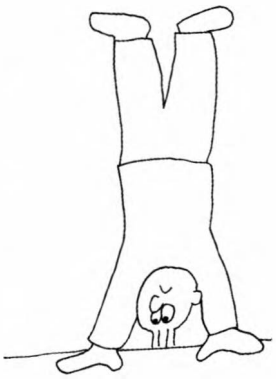
Строка (forty), которую функция `int()` не может привести к числовому значению.

```
assert read_csv(bad_numbers) == [('Burpees', 20, 50)]

no_dash = io.StringIO('exercise, reps\nBurpees, 20\nSitups,
      40-100')
assert read_csv(no_dash) == [('Situps', 40, 100)]

tabs = io.StringIO('exercise\treps\nBurpees\t20-40\nSitups\
                    t40-100')
assert read_csv(tabs) == []
```

Правильно сформированные данные с правильными заголовками, но с символом табуляции в качестве разделителя.

В столбце «reps» отсутствует дефис («20»).

19.4. Прокачиваем навыки



— Включите в программу поддержку другого разделителя, например символа табуляции. Либо добавьте поддержку входных файлов с расширением.tab (как у «неправильного» файла d-delimiter.tab), по которому программа поймет, что в файле используется разделитель в виде символа табуляции.
— Модуль `tabulate` поддерживает множество стилей таблиц, включая `plain`, `simple`, `grid`, `pipe`, `orgtbl`, `rst`, `mediawiki`, `latex`, `latex_raw` и `latex_booktabs`. Реализуйте возможность выбрать другой стиль форматирования таблицы, используя перечисленные стили в качестве допустимых вариантов. Выберите подходящее дефолтное значение.

РЕЗЮМЕ

— Модуль `csv` нужен для анализа текстовых данных с разделителями, таких как CSV и файлы с разделителями в виде символа табуляции.
— Текстовые записи чисел для обработки в программе должны быть приведены к числовым типам с помощью функции `int()` или `float()`.
— Модуль `tabulate` используется для создания текстовых таблиц при выводе форматирования табличных данных.

— Необходимо проявлять большую осторожность, предполагая и обрабатывая «неправильные» и отсутствующие данные. Тесты помогут спрогнозировать ситуации, в которых ваш код может дать сбой.

Глава 20
Сила шифра: создание надежного и запоминающегося пароля

Непросто создавать пароли, которые трудно взломать и при этом легко запомнить. На сайте веб-комикса XKCD опубликован алгоритм разработки пароля, являющегося одновременно и надежным, и запоминаемым. Для этого он должен состоять из «четырех случайных обычных слов». К примеру, согласно комиксу, пароль из слов «правильно», «лошадь», «батарея» и «скоба», обеспечивает «~ 44 бита энтропии». На его угадывание компьютеру потребуется около 550 лет при 1000 попыток в секунду (**https://xkcd.ru/936**).

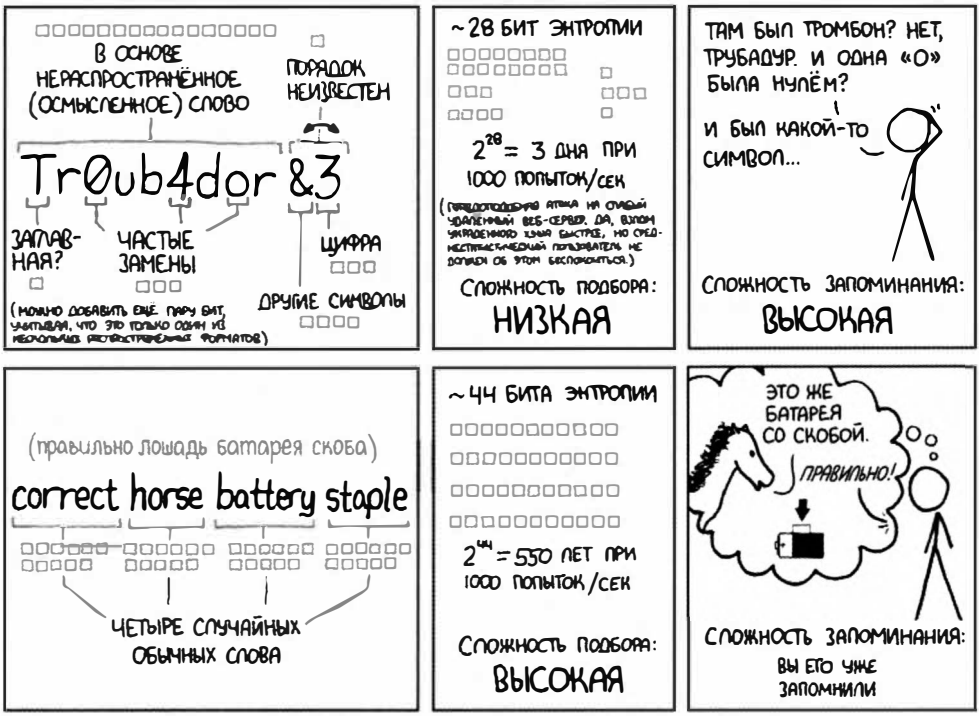
Мы напишем программу *password.py*, генерирующую пароли, случайным образом комбинируя слова из входного текстового файла. На многих компьютерах есть файлы с тысячами английских слов, перечисленными в отдельных строках. На своем компьютере я нашел такой файл в каталоге */usr/share/dict/words*. Он содержит более 235 тыс. слов! Поскольку файлы со словарными словами могут различаться в зависимости от используемой операционной системы, я добавил свой файл в репозиторий, чтобы вы могли работать с тем же файлом, что и я. Файл довольно большой, поэтому я запаковал его в архив *inputs/words.txt.zip*. Перед использованием вам нужно разархивировать его:

```
$ unzip inputs/words.txt.zip
```

Теперь и у меня, и у вас есть одинаковый файл *inputs/words.txt*, соответственно, показанный ниже код будет идентичен:

```
$ ./password.py ../inputs/words.txt --seed 14
CrotalLeavesMeeredLogy
NatalBurrelTizzyOddman
UnbornSignerShodDehort
```

Да, такие пароли не так-то просто запомнить! Может быть, проблема в источнике слов? Мы берем их из словаря объемом более 200 тыс. слов, а обыватель, как правило, использует в речи 20–40 тыс.



(Изображение приведено в книге с разрешения ресурса xkcd.ru.)

Мы можем генерировать более понятные пароли, взяв за источник фрагмент реального английского текста, например Конституции США. Обратите внимание, что для использования в программе входного текста необходимо удалить все знаки препинания, как мы делали в предыдущих упражнениях:

```
$ ./password.py --seed 8 ../inputs/const.txt
DulyHasHeadsCases
DebtSevenAnswerBest
ChosenEmitTitleMost
```

Другим источником запоминающихся слов могут стать романы или стихотворения, полные интересных существительных, глаголов и прилагательных. В репозитории есть программа *harvest.py*. В ней используется библиотека Python для обработки естественного языка с названием spaCy (**https://spacy.io**). Она извлекает в файлы части речи, которые можно взять в качестве входных данных для описываемой программы. Если вы хотите запустить данную программу со своим входным файлом, сначала потребуется установить модуль:

```
$ python3 -m pip install spacy
```

Я запускал программу *harvest.py* с разными текстами и записал результаты в текстовые файлы в подкаталоги папки *20_password* репозитория. Ниже в качестве примера показан результат выборки существительных из Конституции США:

```
$ ./password.py --seed 5 const/nouns.txt
TaxFourthYearList
TrialYearThingPerson
AidOrdainFifthThing
```

А здесь продемонстрированы пароли, сгенерированные из глаголов, извлеченных из «*Алой буквы*» Натаниеля Готорна:

```
$ ./password.py --seed 1 scarlet/verbs.txt
CrySpeakBringHold
CouldSeeReplyRun
WearMeanGazeCast
```

Ну а это пароли из прилагательных, взятых из сонетов Уильяма Шекспира:

```
$ ./password.py --seed 2 sonnets/adjs.txt
BoldCostlyColdPale
FineMaskedKeenGreen
BarrenWiltFemaleSeldom
```

На тот случай, если программа не сможет генерировать достаточно надежные пароли, она будет поддерживать флаг `--l33t`, чтобы еще сильнее обфусцировать (запутать) результат. В этом помогут следующие опции.

1. Передача сгенерированного пароля программе *ransom.py* из главы 12.
2. Перестановка различных символов по таблице, как мы это делали в файле *jump_the_five.py* из главы 4.
3. Добавление рандомного знака препинания в конец.

Вот как выглядят шекспировские пароли с такой обфускацией:

```
$ ./password.py --seed 2 sonnets/adjs.txt --l33t
B0LDco5TLYColdp@l3,
f1n3M45K3dK3eNGR33N[
B4rReNW1LTFeM4l3seldoM/
```

Освоив данную главу, вы научитесь:

— передавать программе список из одного или нескольких входных файлов в качестве позиционных аргументов;
— с помощью регулярного выражения удалять символы, не относящиеся к словам;
— фильтровать слова согласно допустимой минимальной длине слов;
— использовать множества для создания списков уникальных значений;
— генерировать заданное количество паролей, комбинируя указанное количество случайно выбранных слов;
— опционально обфусцировать пароли с помощью алгоритмов, которые мы написали в предыдущих главах.

20.1. Создание файла password.py

Программу нужно разместить в каталоге *20_password* и назвать *password.py*. Она поддерживает опцию `--num`, определяющую количество паролей (по умолчанию, `3`), каждый из которых генерируется случайным образом из опционального (`--num_words`) количества слов (по умолчанию, `4`), извлекаемых из одного или нескольких входных файлов. Поскольку мы используем модуль `random`, программа также принимает аргумент `--seed`, представленный целым числом со значением по умолчанию `None`. Длина слов (после удаления символов, не относящихся к словам) из входных файлов определяется опциями `--min_word_len`

(минимальная длина (по умолчанию 3)) и `--max_word_len` (максимальная длина (по умолчанию 6)).

Традиционно первым делом нужно навести порядок с входными данными для программы. Добейтесь, чтобы при запуске с флагами `-h` и `--help` она выводила показанные ниже инструкции и успешно проходила первые восемь тестов:

```
$ ./password.py -h
usage: password.py [-h] [-n num_passwords] [-w num_words]
                   [-m minimum][-x maximum] [-s seed] [-l]
                   FILE [FILE ...]

Password maker

positional arguments:
  FILE Input file(s)

optional arguments:
-h, --help show this help message and exit
-n num_passwords, --num num_passwords
Number of passwords to generate (default: 3)
-w num_words, --num_words num_words
Number of words to use for password (default: 4)
-m minimum, --min_word_len minimum
Minimum word length (default: 3)
-x maximum, --max_word_len maximum
Maximum word length (default: 6)
-s seed, --seed seed Random seed (default: None)
-l, --l33t Obfuscate letters (default: False)
```

Слова для паролей должны начинаться с прописной буквы (остальные буквы строчные), чего мы можем добиться с помощью метода `str.title()`. Так будет проще опознать и запомнить отдельные слова в выводе. Напоминаю, что мы можем настраивать количество слов, из которых состоит каждый пароль, а также и количество самих паролей:

```
$ ./password.py --num 2 --num_words 3 --seed 9 sonnets/*
QueenThenceMasked
GullDeemdEven
```

Опциональный аргумент `--min_word_len` поможет отфильтровать короткие и неинтересные слова, такие как «a», «I», «an», «of»

и прочие, а аргумент `--max_word_len`, наоборот, отсеять излишне длинные, иначе пароли будет сложно читать и запоминать. Если вы увеличите эти значения, пароли изменятся довольно сильно:

```
$ ./password.py -n 2 -w 3 -s 9 -m 10 -x 20 sonnets/*
PerspectiveSuccessionIntelligence
DistillationConscienceCountenance
```

Флаг `--l33t` — представляет собой стилизованное написание слова leet, при котором латинские буквы меняются на похожие цифры и символы. Так строка ELITE HACKER может выглядеть как `31337 H4X0R*`.

Если программа запущена с флагом `--l33t`, каждый пароль будет шифроваться тремя способами. Во-первых, мы прогоним его через алгоритм `ransom()`, созданный в главе 12:

```
$ ./ransom.py MessengerRevolutionImportune
MesSENGeRReVolUtIonImpoRtune
```

Во-вторых, воспользуемся показанной ниже таблицей поиска для замены символов по аналогии с программой из главы 4:

```
a => @
A => 4
O => 0
t => +
E => 3
I => 1
S => 5
```

В-третьих, с помощью метода `random.choice()` выберем рандомный символ из константы `string.punctuation` и добавим в конец:

```
$ ./password.py --num 2 --num_words 3 --seed 9 --min_word_len 10
--max_word_len 20 sonnets/* --l33t
p3RsPeC+1Vesucces5i0niN+3lL1Genc3$
D1s+iL1@+ioNconscleNc3coun+eN@Nce^
```

* См. страницу «Википедии» https://ru.wikipedia.org/wiki/Leet и инструмент Cryptii https://cryptii.com.

На рис. 20.1 показана диаграмма, отражающая обработку данных в программе.

20.1.1. Создание списка уникальных слов

Начнем с того, что выведем с помощью программы имена всех переданных ей файлов:

```
def main():
    args = get_args()
    random.seed(args.seed)

    for fh in args.file:
        print(fh.name)
```

Вызываем метод `random.seed()` для установки начального состояния модуля `random`, так как это глобально влияет на все операции с ним.

Перебираем файловые аргументы.

Выводим имя файла.

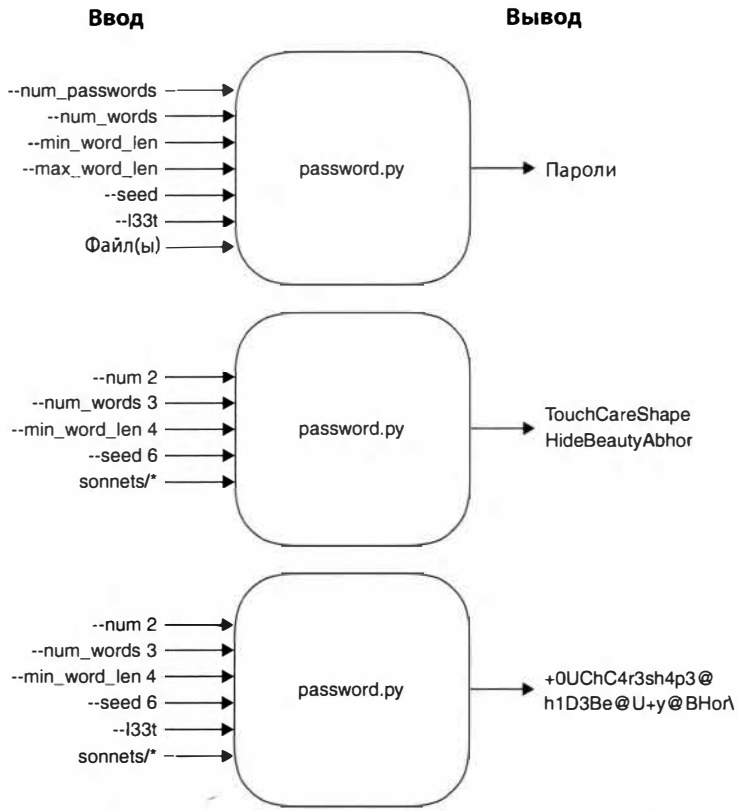


Рис. 20.1. Наша программа принимает много разных опций и требует один или несколько входных файлов. В результате она выдает надежные пароли

Давайте проверим работу программы с файлом *words.txt*:

```
$ ./password.py ../inputs/words.txt
../inputs/words.txt
```

А теперь с другими входными данными:

```
$ ./password.py scarlet/*
scarlet/adjs.txt
scarlet/nouns.txt
scarlet/verbs.txt
```

Наша первая задача — создать список уникальных слов, которые мы сможем использовать для генерации паролей. До сих пор мы имели дело со списками для хранения упорядоченных коллекций таких данных, как строки и числа. Причем элементы в списке необязательно должны были быть *уникальными*. Мы также использовали словари для хранения пар «ключ/значение», а ключи словаря *уникальны*. Поскольку нас не интересуют значения, мы могли бы присвоить каждому ключу какое-нибудь произвольное значение, например 1:

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = {}

    for fh in args.file:
        for line in fh:
            for word in line.lower().split():
                words[word] = 1

    print(words)
```

Создаем пустой словарь для хранения уникальных слов.

Итерируем файлы.

Итерируем строки в файле.

Пишем строку строчными буквами и разбиваем ее по пробелам на слова.

Присваиваем ключу `words[word]` значение `1`. Мы используем словарь только для создания уникальных ключей — слов для паролей. Нас не интересуют значения, поэтому можно использовать любое на свой вкус.

Если вы запустите программу и передадите ей текст Конституции США, вы увидите внушительный список слов (часть вывода опущена):

```
$ ./password.py ../inputs/const.txt
{'we': 1, 'the': 1, 'people': 1, 'of': 1, 'united': 1, 'states,
 ': 1, ...}
```

Здесь я вижу проблему в том, что к слову state приклеилась запятая. Если обработать данную строку в REPL-интерфейсе, мы увидим вот что:

```
>>> 'We the People of the United States,'.lower().split()
['we', 'the', 'people', 'of', 'the', 'united', 'states,']
```

Как избавиться от знаков препинания?

20.1.2. Очистка текста

Мы уже неоднократно убеждались, что разбиение предложений по пробелам оставляет знаки препинания. Поэтому лучше выполнять разбиение по символам, не относящимся к словам. Однако в таком случае возникает новая проблема: сокращенные слова, такие как Don't, разбиваются по апострофу на две части. Нам нужна функция, очищающая слова от лишних символов.

Сначала давайте прикинем тест для нее. Обратите внимание, что в данном упражнении все модульные тесты записаны в файл *unit.py*, запускающийся командой `pytest -xv unit.py`.

Ниже показан тест для нашей функции `clean()`:

```
def test_clean():
    assert clean('') == ''
    assert clean("states,") == 'states'
    assert clean("Don't") == 'Dont'
```

Не забывайте тестировать свои функции на пустом значении, просто чтобы убедиться, что они в принципе работают.

Функция не должна разбивать сокращенное слово на две части.

Функция обязана удалять знаки препинания в конце строки.

Чтобы очистить все слова после разбиения строк, я выбрал функцию `map()` — прекрасный инструмент для решения поставленной задачи. Программисты часто используют в качестве ее аргумента анонимную функцию, как показано на рис. 20.2.

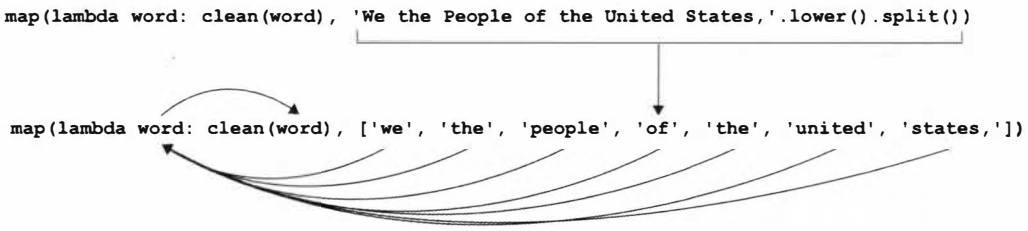


Рис. 20.2. Код функции `map()` с анонимной функцией для получения слов из разделенной строки

На самом деле не нужно писать слово `lambda` в функции `map()`, поскольку функция `clean()` ожидает один аргумент, как показано на рис. 20.3.

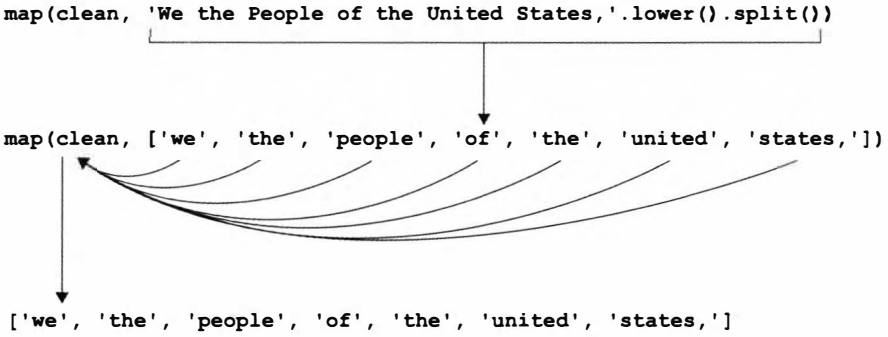


Рис. 20.3. Код функции `map()` без анонимной функции, поскольку функция `clean()` ожидает одно значение

Взгляните на пример интеграции функции очистки в код:

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = {}

    for fh in args.file:
        for line in fh:
            for word in map(clean, line.lower().split()):
                words[word] = 1

    print(words)
```

С помощью функции `map()` применяем функцию `clean()` к результатам разбиения строки по пробелам. Слово `lambda` не требуется, потому что функция `clean()` ожидает один аргумент.

Если мы проверим код с текстом из Конституции США, то увидим, что проблема со словом `states` решена:

```
$ ./password.py ../inputs/const.txt
{'we': 1, 'the': 1, 'people': 1, 'of': 1, 'united': 1,
 'states': 1, ...}
```

Попробуйте самостоятельно написать функцию `clean()`, проходящую данный тест. Вы можете воспользоваться представлением списка, функцией `filter()` или регулярным выражением. Выбор за вами.

20.1.3. Использование множеств

Мы вольны воспользоваться вместо словаря более удобной структурой данных — *множеством*. Множество можно охарактеризовать как «контейнер», содержащий не повторяющиеся элементы в случайном порядке, или просто как список ключей словаря. Перепишем код, чтобы применить множество для отслеживания *уникальных* слов:

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = set()

    for fh in args.file:
        for line in fh:
            for word in map(clean, line.lower().split()):
                words.add(word)

    print(words)
```

С помощью функции `set()` создаем пустое множество.

С помощью функции `set.add()` добавляем значение в множество.

Если вы запустите данный код, вы увидите, что структура данных указана в фигурных скобках ({ }). Отбросьте появившуюся мысль о словаре и обратите внимание на то, что содержимое в скобках больше похоже на список, а не на пары «ключ/значение» (как показано на рис. 20.4):

```
$ ./password.py ../inputs/const.txt
{'', 'impartial', 'imposed', 'jared', 'levying', ...}
```

Рис. 20.4. Множество похоже на нечто среднее между словарем и списком

Я выбрал множество, потому что таким образом гораздо проще сохранить список уникальных слов, а еще множества в нашем случае намного эффективнее других структур. Так, с помощью метода `set.intersection()` можно найти схожие значения в двух разных списках:

```
>>> nums1 = set(range(1, 10))
>>> nums2 = set(range(5, 15))
>>> nums1.intersection(nums2)
{5, 6, 7, 8, 9}
```

Выполните в REPL-интерфейсе команду `help(set)` или покопайтесь в онлайн-документации к множествам, чтобы узнать обо всех удивительных преимуществ этих прекрасных структур данных.

20.1.4. Фильтрация слов

Если мы вновь взглянем на полученный результат, то увидим, что в качестве первого элемента указана пустая строка:

```
$ ./password.py ../inputs/const.txt
{'', 'impartial', 'imposed', 'jared', 'levying', ...}
```

Необходим способ отфильтровать ненужные значения, например излишне короткие слова. В главе 14 мы рассмотрели `filter()` — функцию высшего порядка, принимающую два аргумента:

1) функцию, принимающую один элемент и возвращающую значение `True`, если элемент подходит, или `False` — в противном случае;
2) некоторые «итерируемые» объекты (например список или функцию `map()`), выдающие последовательность элементов для фильтрации.

Мы собираемся принимать только те слова, длина которых больше или равна значению аргумента `--min_word_len` и меньше или равна значению аргумента `--max_word_len`. В REPL-интерфейсе можно указать слово `lambda` для создания анонимной функции, принимающей слово и выполняющей требуемые сравнения. Результатом такого сравнения является либо значение `True`, либо `False`. Допускаются только слова длиной от 3 до 6 букв, поэтому короткие и скучные длинные слова будут отсеиваться. Обратите внимание на лень функции `filter()`. Чтобы добиться от нее результата, в REPL-интерфейсе вызывайте ее с помощью функции `list()`:

```
>>> shorter = ['', 'a', 'an', 'the', 'this']
>>> min_word_len = 3
>>> max_word_len = 6
>>> list(filter(lambda word: min_word_len <= len(word)
                <= max_word_len, shorter))
['the', 'this']
```

Функция `filter()` также отсеивает излишне длинные слова, из-за которых наши пароли стали бы слишком громоздкими:

```
>>> longer = ['that', 'other', 'egalitarian', 'disequilibrium']
>>> list(filter(lambda word: min_word_len <= len(word)
                <= max_word_len, longer))
['that', 'other']
```

Один из способов внедрения функции `filter()` заключается в создании функции `word_len()`, инкапсулирующей анонимную функцию. Обратите внимание, что я определил ее в функции `main()`, сделав *замыкание*, чтобы ссылаться на значения `args.min_word_len` и `args.max_word_len`:

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = set()

def word_len(word):
    return args.min_word_len <= len(word) <= args.max_word_len

    for fh in args.file:
```

Данная функция вернет True, если длина заданного слова находится в допустимом диапазоне.

```
        for line in fh:
            for word in filter(word_len, map(clean,
                                line.lower().split())):
                words.add(word)
print(words)
```

Мы можем использовать word_len (без круглых скобок!) в качестве аргумента функции filter().

Давайте вновь запустим программу, чтобы увидеть, как она себя поведет:

```
$ ./password.py ../inputs/const.txt
{'measures', 'richard', 'deprived', 'equal', ...}
```

Протестируйте ее с несколькими типами входных данных, например с существительными, прилагательными и глаголами из «Алой буквы»:

```
$ ./password.py scarlet/*
{'walk', 'lose', 'could', 'law', ...}
```

20.1.5. «Горбатый регистр»

Мы использовали функцию `line.lower()`, чтобы все слова во входных данных были написаны строчными буквами. Однако для генерируемых паролей нам нужен «горбатый регистр», при котором каждое слово внутри фразы пишется с прописной буквы. Сможете ли вы изменить код таким образом, чтобы программа выводила слова с прописной буквы?

```
$ ./password.py scarlet/*
{'Dark', 'Sinful', 'Life', 'Native', ...}
```

Теперь мы способны обрабатывать сколь угодно файлов для формирования списка уникальных слов в «горбатом регистре» без лишних символов и слишком коротких и длинных слов. Всего пара строк кода, а сколько мощи!

20.1.6. Выборка слов и генерация паролей

Пришло время воспользоваться методом `random.sample()` для рандомного выбора количества (согласно значению аргумента `--num`) слов

из множества и составления надежного, но легко запоминаемого пароля. Ранее я упоминал о важности указания значения зерна для проверки «псевдослучайной» выборки. Для тестирования важно, чтобы при одном и том же значении зерна всегда выбирались одни и те же значения из множества с целью генерации одинаковых паролей для проверки. Применив к множеству функцию `sorted()`, мы получим отсортированный список, идеально подходящий для обработки методом `random.sample()`.

Добавим следующую строку к предыдущему коду:

```
words = sorted(words)
print(random.sample(words, args.num_words))
```

Теперь, если запустить программу с текстом «Алой буквы», я увижу список слов, из которых может получиться интересный пароль:

```
$ ./password.py scarlet/*
['Lose', 'Figure', 'Heart', 'Bad']
```

На выходе метод `random.sample()` выдает список, который можно соединить с пустой строкой, чтобы получить пароль:

```
>>> ''.join(random.sample(words, num_words))
'TokenBeholdMarketBegin'
```

Вам нужно сделать столько паролей, сколько указал пользователь. Похоже на задачу из главы 9, в которой мы генерировали указанное пользователем число оскорблений. Как вы поступите?

20.1.7. Экстремальный режим

В последней части нашей программы мы имеем дело с функцией `133t()`, предназначенной для запутывания пароля. Сначала нужно преобразовать пароль с помощью алгоритма из программы *ransom.py*. Я пошел писать функцию `ransom()`, а пока привожу тест для нее из файла *unit.py*:

```
def test_ransom():
    state = random.getstate()
    random.seed(1)
    assert ransom('Money') == 'moNeY'
```

Сохраняем текущее глобальное состояние.

Передаем `random.seed()` известное значение в целях тестирования.

```
    assert ransom('Dollars') == 'DOLlaRs'
    random.setstate(state)                    ← Восстанавливаем состояние.
```

Надеюсь, вам по силам написать функцию, проходящую показанный тест.

Примечание. Можно выполнить команду `pytest -xv unit.py` для проведения модульных тестов. В целях тестирования программа импортирует различные функции из файла *password.py*. Откройте файл *unit.py* и изучите код, чтобы понять, как выполняется обработка.

Далее я заменю некоторые символы в соответствии с показанной ниже таблицей. Я рекомендую вам обратиться к главе 4, чтобы вспомнить, как это делается:

```
a => @
A => 4
O => 0
t => +
E => 3
I => 1
S => 5
```

Итак, я написал функцию `l33t()`, совмещающую алгоритм замены символов из таблицы выше, функцию `ransom()` и процедуру добавления знака препинания с помощью кода `random.choice(string.punctuation)`.

Ниже показана функция `test_l33t()`, на основе которой вам нужно создать свой вариант функции `l33t()`. Принцип работы данного теста схож с предыдущим, поэтому я воздержусь от комментариев:

```
def test_l33t():
    state = random.getstate()
    random.seed(1)
    assert l33t('Money') == 'moNeY{'
    assert l33t('Dollars') == 'D0ll4r5`'
    random.setstate(state)
```

20.1.8. Собираем все вместе

Опуская детали, я хотел бы предупредить вас, что нужно соблюдать *предельную осторожность* с порядком операций, относящихся к модулю

random. Первая реализация моей программы с флагом --l33t и с одним и тем же зерном выводила разные пароли. Вот такие, к примеру (простенькие):

```
$ ./password.py -s 1 -w 2 sonnets/*
EagerCarcanet
LilyDial
WantTempest
```

Я ожидал получить *точно такие же пароли*, только зашифрованные. И вот что выдала моя программа:

```
$ ./password.py -s 1 -w 2 sonnets/* --l33t
3@G3RC@rC@N3+{
m4dnes5iNcoN5+4n+|
MouTh45sl5T4nCe^
```

Первый пароль выглядит сносно, но что произошло с двумя другими?! Я изменил код, чтобы вывести как зашифрованный, так и исходный пароль. И нате вам:

```
$ ./password.py -s 1 -w 2 sonnets/* --l33t
3@G3RC@rC@N3+{ (EagerCarcanet)
m4dnes5iNcoN5+4n+| (MadnessInconstant)
MouTh45sl5T4nCe^ (MouthAssistance)
```

Модуль random использует глобальное состояние, чтобы осуществить каждый из своих «псевдослучайных» выборов. В первой реализации своей программы я изменил это состояние после вывода первого пароля, сразу же зашифровав его с помощью функции l33t(). Поскольку функция l33t() также задействует функционал модуля random, состояние для следующего пароля было иным. Я решил проблему, сгенерировав сначала *все* пароли, и уже затем изменив их с помощью функции l33t().

Вот и все, что вам понадобится для составления собственной программы. У вас есть модульные тесты, которые помогут проверить функции, и интеграционные тесты, чтобы убедиться, что она работает как цельный организм.

20.2. Решение

Я надеюсь, что вы будете использовать свою программу для генерации паролей. Не забудьте потом поделиться ими с автором этой книги — особенно паролями от банковского счета и от любимых интернет-магазинов!

```
#!/usr/bin/env python3
"""Генератор паролей, https://xkcd.com/936/"""

import argparse
import random
import re
import string

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Password maker',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('rt'),
                        nargs='+',
                        help='Input file(s)')

    parser.add_argument('-n',
                        '--num',
                        metavar='num_passwords',
                        type=int,
                        default=3,
                        help='Number of passwords to generate')

    parser.add_argument('-w',
                        '--num_words',
                        metavar='num_words',
                        type=int,
                        default=4,
                        help='Number of words to use for password')
```

```
    parser.add_argument('-m',
                        '--min_word_len',
                        metavar='minimum',
                        type=int,
                        default=3,
                        help='Minimum word length')

    parser.add_argument('-x',
                        '--max_word_len',
                        metavar='maximum',
                        type=int,
                        default=6,
                        help='Maximum word length')

    parser.add_argument('-s',
                        '--seed',
                        metavar='seed',
                        type=int,
                        help='Random seed')

    parser.add_argument('-l',
                        '--l33t',
                        action='store_true',
                        help='Obfuscate letters')
                        return parser.parse_args()
```

Передаем `random.seed()` указанное пользователем значение или дефолтное значение `None`, что равнозначно отсутствию настройки зерна.

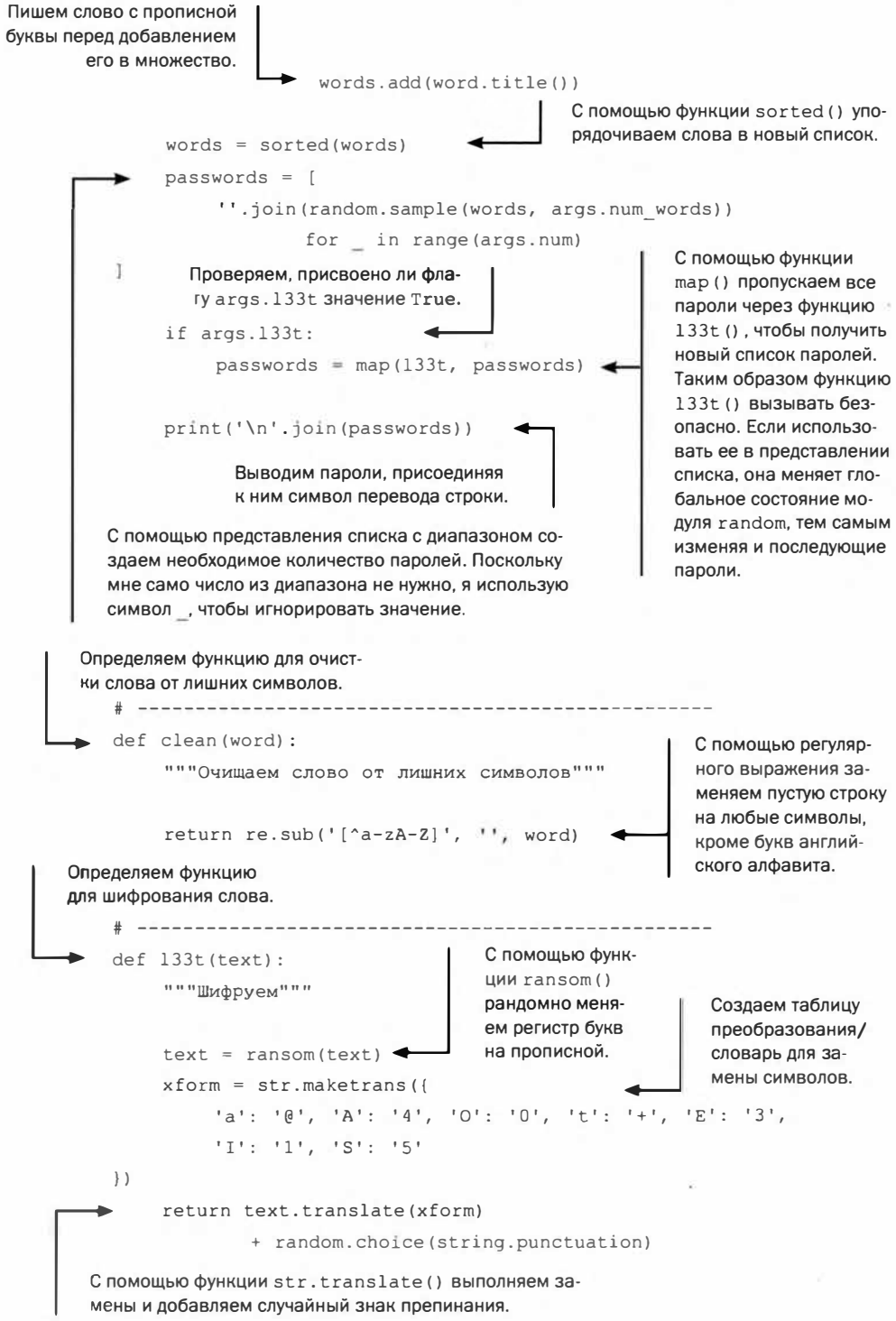
Перебираем все слова, полученные после разбиения строки по пробелам, удаления посторонних символов с помощью функции `clean()` и отсеивания слов нужной длины.

```
# -----------------------------------------------
def main():
    args = get_args()
    random.seed(args.seed)
    words = set()

    def word_len(word):
        return args.min_word_len <= len(word) <= args.max_word_len

    for fh in args.file:
        for line in fh:
            for word in filter(word_len, map(clean,
                                             line.lower().split())):
```

Создаем пустое множество для хранения уникальных слов, извлеченных из файлов.

Создаем функцию `word_len()` для функции `filter()`, возвращающей `True`, если длина слова находится в допустимом диапазоне, и `False` в противном случае.

Перебираем открытые файловые дескрипторы.


Пишем слово с прописной буквы перед добавлением его в множество.
words.add(word.title())
С помощью функции sorted() упорядочиваем слова в новый список.
words = sorted(words)
passwords = [
''.join(random.sample(words, args.num_words))
for _ in range(args.num)
]
Проверяем, присвоено ли флагу args.l33t значение True.
if args.l33t:
passwords = map(l33t, passwords)
С помощью функции map() пропускаем все пароли через функцию l33t(), чтобы получить новый список паролей. Таким образом функцию l33t() вызывать безопасно. Если использовать ее в представлении списка, она меняет глобальное состояние модуля random, тем самым изменяя и последующие пароли.
print('\n'.join(passwords))
Выводим пароли, присоединяя к ним символ перевода строки.
С помощью представления списка с диапазоном создаем необходимое количество паролей. Поскольку мне само число из диапазона не нужно, я использую символ _, чтобы игнорировать значение.
Определяем функцию для очистки слова от лишних символов.
# --
def clean(word):
"""Очищаем слово от лишних символов"""
return re.sub('[^a-zA-Z]', '', word)
С помощью регулярного выражения заменяем пустую строку на любые символы, кроме букв английского алфавита.
Определяем функцию для шифрования слова.
# --
def l33t(text):
"""Шифруем"""
text = ransom(text)
С помощью функции ransom() рандомно меняем регистр букв на прописной.
xform = str.maketrans({
'a': '@', 'A': '4', 'O': '0', 't': '+', 'E': '3',
'I': '1', 'S': '5'
})
Создаем таблицу преобразования/словарь для замены символов.
return text.translate(xform)
+ random.choice(string.punctuation)
С помощью функции str.translate() выполняем замены и добавляем случайный знак препинания.

```
# --------------------------------------------------
def ransom(text):
    """Выводим случайным образом выбранную прописную
       или строчную букву"""

    return ''.join(
        map(lambda c: c.upper() if random.choice([0, 1])
            else c.lower(), text))

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Определяем функцию для алгоритма `ransom()` из главы 12.

Возвращаем новую строку со словами с рандомными прописными и строчными буквами.

20.3. Обсуждение

Надеюсь, эта программа показалась вам не только сложной, но и интересной. Ничего нового в функции `get_args()` нет, но опять же примерно половина кода расположена именно в ней. Это говорит о том, насколько важно тщательно определять и проверять входные данные для программы!

Теперь обсудим дополнительные функции.

20.3.1. Очистка текста

Я написал регулярное выражение для удаления любых символов, отличных от строчных и прописных букв английского алфавита:

```
def clean(word):
    """Очищаем слово от лишних символов"""
    return re.sub('[^a-zA-Z]', '', word)
```

Функция `re.sub()` заменит любой текст, соответствующий шаблону (первый аргумент), найденному в указанном тексте (третий аргумент), на значение, заданное вторым аргументом.

Вспомним главу 18, можно написать символьный класс `[a-zA-Z]` для определения символов по таблице ASCII, ограниченных двумя диапазонами. Затем мы способны *обратить* этот класс, указав знак

вставки (^) в качестве *первого символа* в его коде. Шаблон `[^a-zA-Z]` можно прочитать как «любой символ, отличный от букв в диапазонах от a до z и от A до Z».

Чтобы было понятнее, откройте REPL-интерфейс и выполните там показанный ниже код. В следующем примере из строки `'A1b*C!d4'` останутся только буквы «AbCd»:

```
>>> import re
>>> re.sub('[^a-zA-Z]', '', 'A1b*C!d4')
'AbCd'
```

Если бы единственной нашей задачей было сопоставление букв из таблицы ASCII, мы решили бы ее путем сверки с константным перечислением `string.ascii_letters`:

```
>>> import string
>>> text = 'A1b*C!d4'
>>> [c for c in text if c in string.ascii_letters]
['A', 'b', 'C', 'd']
```

Представление списка с охранным выражением также можно записать с помощью функции `filter()`:

```
>>> list(filter(lambda c: c in string.ascii_letters, text))
['A', 'b', 'C', 'd']
```

Обе версии программы без регулярных выражений кажутся мне трудоемкими. Кроме того, если в будущем потребуется изменить функцию, чтобы допустить в пароли, скажем, цифры и определенные знаки препинания, версия с регулярным выражением окажется значительно проще как в написании, так и в поддержке.

20.3.2. Королевский метод с функцией ransom()

Функцию `ransom()` я взял из программы *ransom.py*, над которой мы работали в главе 12, так что о ней особо нечего сказать, кроме: эй, глядите, как далеко мы продвинулись! Код, бывший кульминацией целой главы, теперь представляет собой одну строку в гораздо более длинной и сложной программе:

```
def ransom(text):
    """Выводим случайным образом выбранную прописную или строчную
       букву"""
    return ''.join(
        map(lambda c: c.upper() if random.choice([0, 1])
        else c.lower(), text))
```

Соединяем результирующий список из функции `map()` с пустой строкой, чтобы создать новую строку.

С помощью функции `map()` перебираем каждый символ в тексте и выбираем регистр символа (прописной или строчный) путем «жеребьевки», применяя метод `random.choice()` для выбора между «истинным» (1) и «ложным» значениями (0).

20.3.3. Шифрование паролей

Функция `l33t()` основана на `ransom()`. Она перемешивает буквы — так же, как мы делали с цифрами в главе 4. Мне по душе вариант программы с методом `str.translate()`, поэтому я и использовал его здесь:

Рандомно капитализируем буквы полученного текста.

Создаем таблицу поиска из полученного словаря, описывающую, как изменить один символ на другой. Символы, отсутствующие среди ключей словаря, будут игнорироваться.

```
def l33t(text):
    """Шифруем"""
    text = ransom(text)
    xform = str.maketrans({
        'a': '@', 'A': '4', 'O': '0', 't': '+', 'E': '3',
        'I': '1', 'S': '5'
})
    return text.translate(xform)
           + random.choice(string.punctuation)
```

Используя метод `str.translate()`, заменяем все символы. С помощью метода `random.choice()` выбираем один знак препинания из константы `string.punctuation` и добавляем в конец.

20.3.4. Обработка файлов

Чтобы использовать описанные функции, необходимо подготовить множество уникальных слов из входных файлов. Я написал показанный ниже фрагмент кода с прицелом как на производительность, так и на стиль:

```
words = set()
for fh in args.file:
```

Перебираем открытые файловые дескрипторы.

```
    for line in fh:
        for word in filter(word_len, map(clean,
                                         line.lower().split())):
            words.add(word.title())
```

Производим построчное чтение из файлового дескриптора с помощью цикла `for`, а не метода типа `fh.read()`, который считывает все содержимое файла целиком.

Пишем слово с прописной буквы перед добавлением его в множество.

Читать данный листинг лучше с конца. Там я методом `line.lower()` разбиваю строку по пробелам. Каждое слово, переданное методом `str.split()`, передается функции `clean()`, а затем функции `filter()`.

На рис. 20.5 показана диаграмма данной строки кода с оператором `for`.

1. Метод `line.lower()` возвращает строку строчными буквами.
2. Метод str.split() разбивает текст по пробельным символам, возвращая отдельные слова.
3. Каждое слово передается функции `clean()` для удаления символов, отличных от букв английского алфавита.
4. Очищенные слова отфильтровываются функцией `word_len()`.
5. Полученное слово в переменной `word` преобразовано, очищено и отфильтровано.

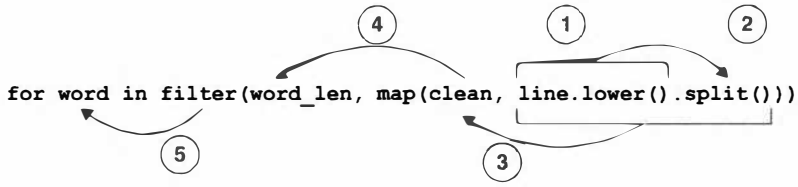


Рис. 20.5. Наглядная демонстрация порядка выполнения различных функций

Если вам не нравятся функции `map()` и `filter()` — не проблема, можно переписать код следующим образом:

```
words = set()
for fh in args.file:
    for line in fh:
        for word in line.lower().split():
            word = map(clean)
            if args.min_word_len <= len(word) <= args.max_word_len:
                words.add(word.title()
```

Перебираем открытые файловые дескрипторы.

Перебираем все строки из файлового дескриптора.

Перебираем каждое «слово», разбивая строку строчными буквами по пробелам.

Удаляем ненужные символы.

Проверяем, допустимая ли у слова длина.

Слово с прописной буквы добавляем в множество.

Независимо от способа обработки файлов, к данному моменту у вас должно быть полное множество всех уникальных слов с прописной буквы, извлеченных из входных файлов.

20.3.5. Выборка и создание паролей

Как уже упоминалось, в целях тестирования очень важно сортировать слова во множестве `words`, чтобы можно было убедиться в последовательности выборки. Если вам по душе рандомный выбор и вы ненавидите тестирование, пожалуй, тогда вам не нужно переживать о сортировке. Однако в таком случае я бы назвал вас весьма странным человеком. Так что отставить подобные мысли!

Я взял функцию `sorted()`, поскольку не знаю других способов сортировки множества:

```
words = sorted(words)
```

Функции `set.sort()` не существует. Множества упорядочиваются прямо в Python. Функция `sorted()` создает из множества новый отсортированный список.

Нам нужно создать определенное количество паролей, и, предполагаю, проще всего прибегнуть к циклу `for` с функцией `range()`. В своем решении я использовал код `for _ in range(...)`, так же, как и в главе 9, потому что мне не нужен результат при каждой итерации цикла. Символ подчеркивания (`_`) указывает на то, что вы игнорируете значение. Данный код можно написать и так: `for i in range(...)`, однако некоторые линтеры начнут ругаться из-за того, что переменная `i` объявляется, но не используется. Такая ситуация вполне может считаться ошибкой, поэтому лучше иметь дело с символом `_`, чтобы четко сказать, что вы игнорируете соответствующее значение.

Покажу первый вариант кода. Он вызвал ошибку при генерации разных паролей даже с одним и тем же зерном, о которой я упоминал ранее. Сможете найти, куда закралась эта ошибка?

Перебираем `args.num` паролей, которые нужно создать.

Каждый пароль будет основан на случайной выборке слов, и я выберу значение, указанное в `args.num_words`. Функция `random.sample()` возвращает список слов, которые я методом `str.join()` соединил с пустой строкой для создания новой строки.

```
for _ in range(args.num):
    password = ''.join(random.sample(words, args.num_words))
    print(l33t(password) if args.l33t else password)
```

Если флаг `args.l33t` «истинен», мы выводим зашифрованную версию пароля; в противном случае пароль выводится без шифрования. А вот и ошибка! Вызов функции `l33t()` изменяет глобальное состояние модуля `random`, поэтому при следующем вызове метода `random.sample()` я работаю уже с другой выборкой.

Суть в том, что нужно разделить операции создания паролей и их опционального шифрования:

```
passwords = [
    ''.join(random.sample(words, args.num_words))
    for _ in range(args.num)
]

if args.133t:
    passwords = map(133t, passwords)

print('\n'.join(passwords))
```

С помощью представления списка перебираем `range(args.num)`, чтобы сгенерировать правильное количество паролей.

Если флаг `args.leet` «истинен», используем функцию `133t()` для шифрования паролей.

Выводим пароли с символами перевода строки.

20.4. Прокачиваем навыки

— В части замены функция `133t()` изменяет каждый доступный символ, тем самым усложняя пароль для запоминания. Желательно менять лишь часть пароля, например 10%, как делали с текстом в упражнении с «испорченным телефоном» из главы 10.

— Создавайте программы, сочетающие в себе приемы, которым вы научились. Например, генератор стихов, случайным образом выбирающий строки из файлов с песнями ваших любимых исполнителей, затем кодирует текст по аналогии с программой из главы 15, потом заменяет все гласные на одну букву, как в главе 8, после чего выкрикивает результат, как в главе 5.

РЕЗЮМЕ

— Множество — это коллекция уникальных значений. Множества могут взаимодействовать друг с другом для создания различий, пересечений, объединений и многого другого.

— Изменение порядка операций с помощью модуля `random` может повлиять на вывод программы, поскольку глобальное состояние модуля `random` способно измениться.

— Короткие, протестированные функции используются для разработки более сложных проверенных программ. В данной главе мы объединили в кратких и мощных функциях многие идеи из предыдущих упражнений.

Глава 21

Крестики-нолики: все, что вы хотели знать о состояниях, но боялись спросить

Один из моих любимых фильмов — «*Военные игры» 1983 года* с Мэттью Бродериком в главной роли, чей персонаж, Дэвид, играет молодого хакера, взламывающего компьютерные системы — начиная со школьного компьютера с табелем успеваемости и заканчивая сервером Пентагона, контролирующим запуск межконтинентальных баллистических ракет.

Взломав сервер Пентагона, Дэвид начинает общаться с Джошуа, искусственным интеллектом (ИИ) компьютерной системы министерства обороны, который предлагает ему сыграть в одну из игр на выбор: «Крестики-нолики», «Шашки», «Шахматы», «Бридж», «Покер» и т. д. Среди игр встречаются и такие странные названия, как «Воздушные бои», «Военные действия в городских условиях», «Война в пустыне», «Война на нескольких театрах военных действий с применением химического и биологического оружия» и «Глобальная термоядерная

война». Заинтригованный, Дейв выбирает последнее название и начинает игру за СССР против США. В конце концов он догадывается, что Джошуа симулирует военную игру, а на самом деле обманом планирует нанести превентивный ядерный удар по Советскому Союзу. Понимая доктрину гарантированного взаимного уничтожения, Дэвид просит Джошуа сыграть в «Крестики-нолики», где выставляет число игроков «ноль», заставляя компьютер играть с самим собой, при этом постоянно терпя неудачи. Джошуа проигрывает вничью сотни и тысячи раундов в «Крестики-нолики», параллельно прогоняя все возможные сценарии ядерного конфликта, каждый из которых приводит к тому, что потери всех сторон оказываются настолько высоки, что победитель не вычисляется. В итоге он приходит к выводу, что «единственный выигрышный ход — не играть» и прекращает как игру в «Крестики-нолики», так и «Глобальную термоядерную войну».

Центральное место в сюжете фильма занимает игра «Крестики-нолики», настолько простая, что обычно заканчивается ничьей для обоих игроков. Предполагаю, вы уже знакомы с данной игрой, однако кратко расскажу о ней на тот случай, если ваше детство прошло без бесчисленных попыток обыграть в нее друзей.

Игра ведется на расчерченном квадратными клетками поле размером 3 на 3 клетки. Она рассчитана на двух игроков. Они по очереди отмечают символы X и O (у каждого свой символ). Выигрывает тот, кто нарисует свои символы в любых трех клетках по горизонтали, вертикали или диагонали. Обычно это невозможно, поскольку каждый игрок своими символами блокирует потенциальную победу противника.

Последние две главы данной книги посвящены созданию игры «Крестики-нолики». Мы рассмотрим концепции представления и отслеживания *состояния программы*. Компьютерные программы хранят данные в переменных, представляющих собой области хранения данных в памяти компьютера. Содержание этих областей памяти в любой момент исполнения программы называется *состоянием* программы*.

Например, начнем с пустого поля. Первым ходит игрок X. Игроки X и O ходят строго по очереди, занимая одну клетку за один ход, поэтому после каждого раунда на поле заняты две клетки обоими игроками. Необходимо отслеживать эти и другие ходы, чтобы в любой момент времени всегда знать состояние игры.

Возможно, вы помните, как скрытое состояние модуля `random` оказалось проблемой в главе 20, где одно из рассмотренных нами решений приводило к противоречивым результатам в зависимости от порядка операций, связанных с модулем. В данном упражнении мы подумаем

* https://ru.wikipedia.org/wiki/Состояние_(информатика)#Состояние_программы.

о том, как явно отслеживать состояние нашей игры и любые изменения в нем.

В этой главе мы напишем программу, выполняющую только один ход, а в следующей расширим ее до полноценной игры. Текущая версия программы будет получать строку, отражающую состояние игрового поля в любой момент игры. По умолчанию это пустое поле, имеющееся при запуске, до первого хода одного из игроков. Программа также выполнит один ход, добавив знак в клетку поля. Она выведет изображение поля и сообщит, есть ли победитель после сделанного хода.

В рамках этой программы необходимо отслеживать как минимум два момента в состоянии:

— поле, определяя, какой игрок какие клетки отметил;
— победителя, если таковой есть.

В следующей главе мы напишем интерактивную полную версию игры «Крестики-нолики», в которой необходимо будет отслеживать и обновлять еще несколько моментов состояния.

Прочитав данную главу, вы научитесь:

— использовать такие объекты, как строки и списки, для представления аспектов состояния программы;
— соблюдать правила игры с помощью кода, например, вы сможете запретить игроку ставить еще один знак в уже занятой клетке;
— с помощью регулярного выражения проверять исходное поле;
— применять операторы `and` и `or`, чтобы свести комбинации логических значений к одному значению;
— с помощью многомерных списков определять выигрыш на поле;
— используя функцию `enumerate()`, перебирать списки с индексами и значениями.

21.1. Создание файла tictactoe.py

Создайте программу *tictactoe.py* в каталоге *21_tictactoe*. Как всегда, я рекомендую сделать это с помощью файла *new.py* или *template.py*.

Давайте обсудим параметры программы. Начальное состояние игрового поля определяется опцией `-b` или `--board`, описывающей, какие клетки и какими игроками заняты. Поскольку клеток девять, мы воспользуемся строкой длиной в девять символов, состоящей из символов `X`, `O` и точка (`.`). Точка обозначает, что клетка пуста. Поле по умолчанию

представляет собой строку из девяти точек. При отображении поля вы будете выводить либо метку игрока в клетке, либо номер клетки в диапазоне от одного до девяти. В следующей версии игры этот номер будет использоваться игроком для определения клетки для своего хода. Поскольку на поле по умолчанию нет победителя, программа должна выводить текст "No winner":

```
$ ./tictactoe.py
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
No winner.
```

Опция `--board` определяет, какие клетки каким игроком заняты. Она принимает строку, в которой клетки поля указываются в порядке возрастания от 1 до 9. Согласно строке `XO.O..X` клетки 1 и 9 отмечены игроком X, а позиции 3 и 6 — игроком O. Остальные клетки пусты (рис. 21.1).

Вот как данная комбинация будет отображаться программой:

```
$ ./tictactoe.py -b X.O..O..X
-------------
| X | 2 | O |
-------------
| 4 | 5 | O |
-------------
| 7 | 8 | X |
-------------
No winner.
```

Мы также можем изменить параметр `--board`, передав опцию `-c` или `--cell` с цифрами в диапазоне `1-9` и опцию `-p` или `--player` с указанием буквы игрока `X` или `O`. Например, мы способны пометить первую клетку знаком «X» следующим образом:

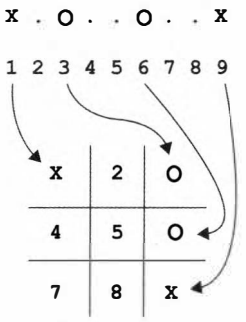


Рис. 21.1. Поле формируется из девяти символов, описывающих девять клеток поля

```
$ ./tictactoe.py --cell 1 --player X
-------------
| X | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
No winner.
```

Победителя, если он есть, следует объявить с восклицанием:

```
$ ./tictactoe.py -b XXXOO...
-------------
| X | X | X |
-------------
| O | O | 6 |
-------------
| 7 | 8 | 9 |
-------------
X has won!
```

По традиции мы воспользуемся набором тестов, проверяющих, что наша программа работает правильно. На рис. 21.2 показана диаграмма.

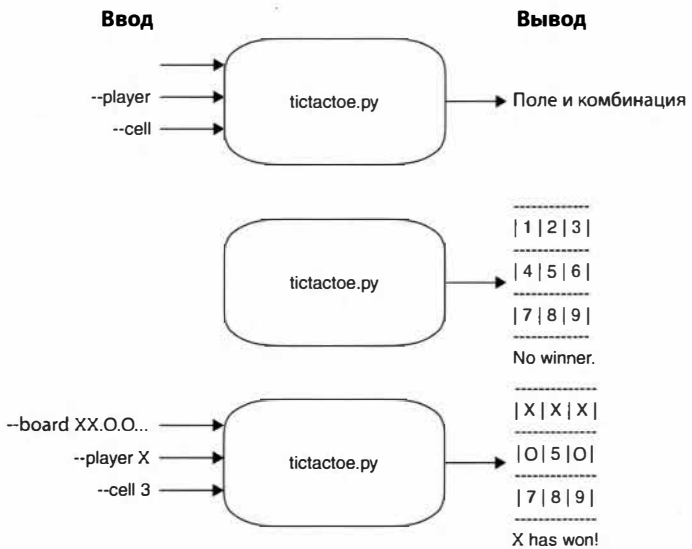


Рис. 21.2. Программа «Крестики-нолики» будет выполнять один ход, обрабатывая поле, игрока и клетку. Она выведет на поле комбинацию, а также победителя, если таковой есть

21.1.1. Проверка пользовательского ввода

Необходимо выполнить тщательную проверку ввода. Аргумент `--board` должен содержать точно 9 символов и только буквы `X` и `O`, а также точку `.`:

```
$ ./tictactoe.py --board XXXOOO..
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: --board "XXXOOO.." must be 9 characters
             of ., X, O
```

Аналогично опция `--player` поддерживает исключительно буквы `X` или `O`:

```
$ ./tictactoe.py --player A --cell 1
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: argument -p/--player: \
invalid choice: 'A' (choose from 'X', 'O')
```

А аргумент `--cell` принимает только целые числа в диапазоне от 1 до 9:

```
$ ./tictactoe.py --player X --cell 10
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: argument -c/--cell: \
invalid choice: 10 (choose from 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Оба аргумента, и `--player`, и `--cell`, могут использоваться лишь вместе:

```
$ ./tictactoe.py --player X
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: Must provide both --player and --cell
```

Наконец, если указанная в аргументе `--cell` клетка уже занята знаком `X` или `O`, программа должна выдать ошибку:

```
$ ./tictactoe.py --player X --cell 1 --board X..O....
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: --cell "1" already taken
```

Я рекомендую поместить весь код, отвечающий за проверку отсутствия ошибок, в функцию `get_args()`. Так вы сможете использовать

метод `parser.error()`, чтобы выбросить ошибки и завершить работу программы.

21.1.2. Изменение поля

Первоначальное поле после проверки описывает, какие клетки и каким игроком заняты. Комбинацию на поле можно изменить, передав аргументы `--player` и `--cell`. Кто-то, вероятно, подумает, что глупо передавать указанные аргументы для изменения параметра `--board`, однако это необходимо для разработки интерактивной версии игры.

Зная, что поле в игре, по сути, строковое значение типа `'XX.OO.X'`, и что вам нужно изменить клетку 3, например поместить в нее знак X, как вы это сделаете? Учтите, что, во-первых, клетка 3 не соответствует *индексу* 3 на данном поле — индекс на *единицу меньше* номера клетки. Во-вторых, строки неизменны. Как и в программе «Испорченный телефон» из главы 10, вам нужно найти способ изменить один символ в значении поля.

21.1.3. Вывод комбинации на поле

Чтобы вывести поле, вам нужно отформатировать его с помощью символов ASCII и сформировать сетку. Я рекомендую создать функцию `format_board()`, принимающую строку `board` в качестве аргумента и возвращающую строку с символами дефиса (`-`) и вертикальной черты (`|`) для конструирования таблицы. Я добавил в репозиторий файл *unit.py*, который содержит показанный ниже тест для дефолтной комбинации поля (пустого):

```
def test_board_no_board():
    """makes default board"""

    board = """
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
""".strip()

    assert format_board('.' * 9) == board
```

Здесь используются тройные кавычки, потому что в строке есть экранированные символы перевода строки. При завершающем вызове метода `str.strip()` символ перевода строки, применяющийся для форматирования, будет удален.

При перемножении строки с целочисленным значением интерпретатор будет повторять данную строку указанное число раз. Мы создаем строку из девяти точек в качестве входных данных для функции `format_board()`. В результате ожидается пустое поле, как показано здесь.

Теперь попробуйте вывести поле с другой комбинацией. Покажу еще один свой тест, который вы можете использовать (или написать и свой собственный):

```
def test_board_with_board():
    """makes board"""

    board = """
-------------
| 1 | 2 | 3 |
-------------
| O | X | X |
-------------
| 7 | 8 | 9 |
-------------
""".strip()

    assert format_board('...OXX...') == board
```

На данном поле первая и третья строки пусты, а вторая содержит символы OXX.

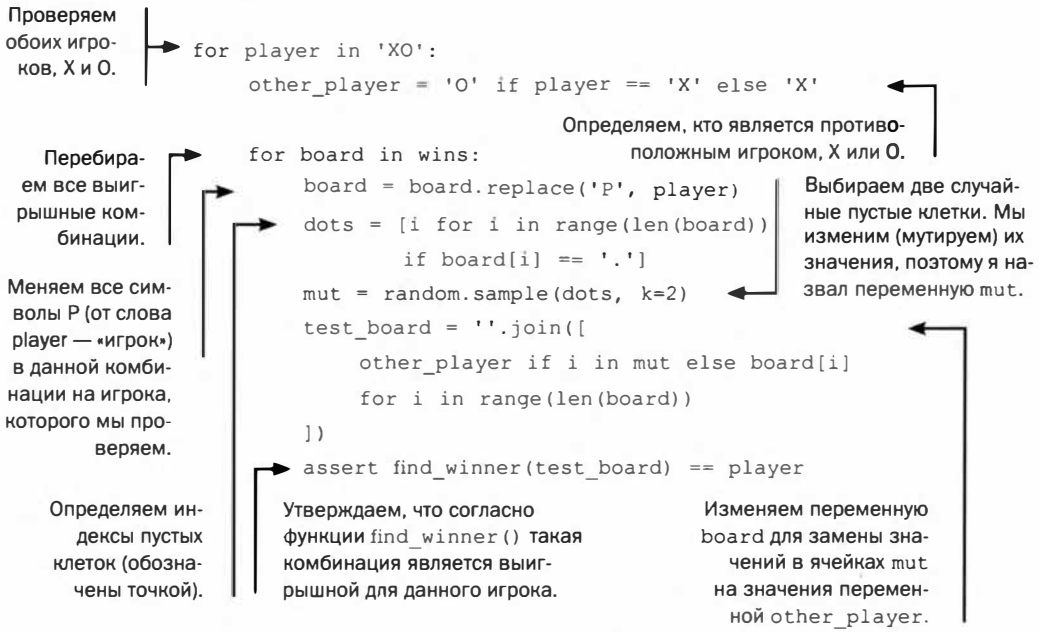
Непрактично тестировать все возможные комбинации на поле. В процессе работы над тестами вам часто придется проверять код выборочно. Я проверил пустое поле и непустое (с одной из комбинаций). Предположу, что если функция может обрабатывать эти два аргумента, она справится и с другими.

21.1.4. Определение победителя

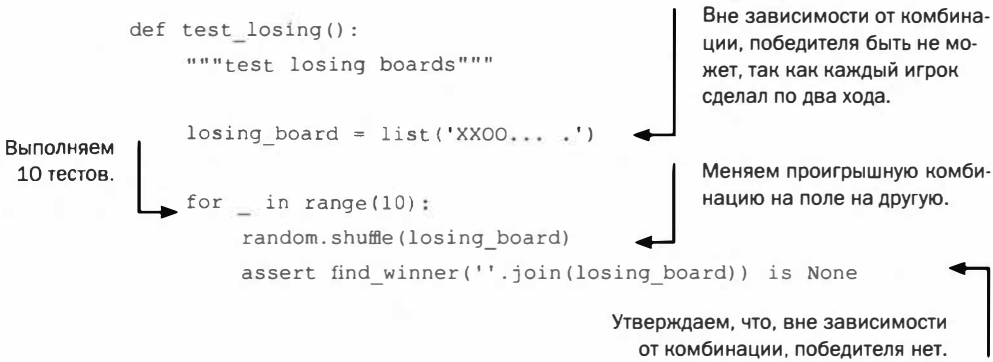
После проверки входных данных и вывода поля с комбинацией наша последняя задача — объявить победителя. Я написал функцию `find_winner()`, возвращающую либо `X`, либо `O`, если один из игроков победил, или же `None` — если победителя нет. Я записал все возможные выигрышные комбинации, чтобы проверить свою функцию с комбинациями для обоих игроков. Традиционно предлагаю проанализировать мой тест:

```
def test_winning():
    """test winning boards"""

    wins = [('PPP......'), ('...PPP...'), ('......PPP'),
            ('P..P..P..'), ('.P..P..P.'), ('..P..P..P'),
            ('P...P...P'), ('..P.P.P..')]
```

Перед вами — список комбинаций индексов клеток. Если любая из комбинаций клеток занята одним и тем же игроком, он выиграл.



Еще следует проконтролировать, чтобы проигрышная комбинация не считалась выигранной одним из игроков. Поэтому я написал следующий тест, проверяющий, что, если победитель отсутствует, возвращается значение None:



Если у вас в программе используются функции с теми же именами, что и у меня, вы можете выполнить команду `pytest -xv unit.py` для запуска разработанных мной модульных тестов. При выборе других функций напишите собственные модульные тесты в файле *tictactoe.py* (или в ином файле).

После вывода комбинации обязательно напечатайте текст «{Буква победителя} has won!» или "No winner" в зависимости от результата. Итак, приказ отдан! Шагом марш!

21.2. Решение

Маленькими шажками мы продвигаемся к полноценной интерактивной игре, которую создадим в следующей главе. А сейчас необходимо закрепить основные концепции выполнения одного хода. Полезно постепенно реализовывать функционал сложных программ — вы начинаете с наипростейшей версии и постепенно добавляете функции, позволяющие выстроить более сложную структуру.

```
#!/usr/bin/env python3
"""Крестики-нолики"""

import argparse
import re

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Tic-Tac-Toe',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-b',
                        '--board',
                        help='The state of the board',
                        metavar='board',
                        type=str,
                        default='.' * 9)

    parser.add_argument('-p',
                        '--player',
                        help='Player',
                        choices='XO',
                        metavar='player',
                        type=str,
                        default=None)
```

Аргумент `--board` по умолчанию содержит строку из девяти точек. Если применить оператор умножения (`*`) со строковым значением и целым числом (в любом порядке), в результате получится строковое значение, повторяющееся указанное количество раз. К примеру, код `'.' * 9` приведет к строке из девяти точек, `'.........'`.

Аргумент `--player` принимает либо `X`, либо `O`, что проверяется с помощью переменной `choices`.

```
        parser.add_argument('-c',
                            '--cell',
                            help='Cell 1-9',
                            metavar='cell',
                            type=int,
                            choices=range(1, 10),
                            default=None)

        args = parser.parse_args()

        if any([args.player, args.cell]) and not
                all([args.player, args.cell]):
            parser.error('Must provide both --player and --cell')

        if not re.search('^[.XO]{9}$', args.board):
            parser.error (f'--board "{args.board}" must be
                          9 characters of ., X, O')

        if args.player and args.cell and args.board[args.cell - 1]
            in 'XO':
            parser.error(f'--cell "{args.cell}" already taken')

        return args


    # --------------------------------------------------
    def main():
        """Да грянет джаз!"""

        args = get_args()
        board = list(args.board)

        if args.player and args.cell:
            board[args.cell - 1] = args.player

        print(format_board(board))
        winner = find_winner(board)
```

Аргумент `--cell` поддерживает целые числа в диапазоне от 1 до 9, что можно проверить, используя код `type=int` и `choices=range(1, 10)`, но не забывая, что верхнее значение (`10`) не включено.

С помощью регулярного выражения проверяем, что значение `--board` состоит именно из девяти допустимых символов.

Комбинация функций `any()` и `all()` позволяет проверить, что переданы / не переданы сразу оба аргумента, а не по одному.

Если оба аргумента, и `--player`, и `--cell`, переданы и допустимы, проверяем, что указанная в них клетка на поле в данный момент свободна.

Изменяем переменную `board`, если и `cell`, и `player` «истинны». Так как аргументы проверяются в функции `get_args()`, их можно использовать безопасно. То есть я не буду проверять значение индекса за пределами диапазона, потому что уже проверил значение клетки.

Так как нумерация клеток начинается с 1, вычитаем 1 из номера клетки, чтобы изменить на поле клетку с правильным индексом.

Поскольку нам может понадобиться изменить значение переменной `board`, проще всего преобразовать его в список.

Определяем победителя комбинации.

Выводим поле.

```
print(f' {winner} has won!' if winner else 'No winner.')
```

Выводим результат игры. Функция `find_winner()` возвращает либо `X`, либо `O`, если один из игроков выиграл, либо `None`, если победителя нет.

Определяем функцию для форматирования поля — содержимого переменной `board`. Функция не выводит значение данной переменной, потому что так усложняется ее тестирование. Функция возвращает новое строковое значение, которое можно вывести или протестировать.

Перебираем клетки поля и определяем, печатать ли букву игрока, если клетка занята, или номер клетки, если она свободна.

```
# ------------------------------------------------
def format_board(board):
    """Форматируем поле"""

    cells = [str(i) if c == '.' else c for i,
             c in enumerate(board, 1)]
    bar = '-------------'
    cells_tmpl = '| {} | {} | {} |'
    return '\n'.join([
        cells_tmpl.format(*cells[:3]), bar,
        cells_tmpl.format(*cells[3:6]), bar,
        cells_tmpl.format(*cells[6:]), bar
])
```

Результат функции — новая строка, созданная путем соединения всех строк сетки с символами перевода строки.

Определяем функцию, возвращающую букву победителя или значение `None`, если победителя нет. Опять же, функция не выводит букву победителя, а возвращает только ответ, который можно вывести или протестировать.

Существуют восемь выигрышных комбинаций, определенных как восемь списков с индексами клеток, которые должны быть заняты одним и тем же игроком. Обратите внимание, что индексы клеток считаются с 0, а не с 1, как номера клеток, указываемые пользователем.

```
# ------------------------------------------------
def find_winner(board):
    """Возвращаем победителя"""

    winning = [[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6],
               [1, 4, 7],[2, 5, 8], [0, 4, 8], [2, 4, 6]]

    for player in ['X', 'O']:
        for i, j, k in winning:
            combo = [board[i], board[j], board[k]]
            if combo == [player, player, player]:
                return player
```

Перебираем обоих игроков, X и O.

Выстраиваем комбинацию из значений занятых клеток, указанных в переменных `i`, `j` и `k`.

Перебираем каждую выигрышную комбинацию клеток, распаковывая их значения в переменные `i`, `j` и `k`.

Если в результате получаем `True`, возвращаем игрока. Если не получаем `True` ни для одной из комбинаций, выходим из функции, не возвращая значение, и поэтому по умолчанию возвращается `None`.

Проверяем, заняты ли одним и тем же игроком все клетки комбинации.

```
# ------------------------------------------------
if __name__ == '__main__':
    main()
```

21.2.1. Проверка аргументов и изменение переменной board

Бóльшая часть проверки эффективно выполняется с помощью модуля `argparse`. Оба параметра, `--player` и `--cell`, могут обрабатываться с помощью опции `choices`. Рассмотрим, как используются функции `any()` и `all()`:

```
if any([args.player, args.cell]) and not all([args.player,
      args.cell]):
    parser.error('Must provide both --player and --cell')
```

Давайте поэкспериментируем с этими функциями в REPL-интерфейсе. Функция `any()` аналогична логическому оператору `or`:

```
>>> True or False or True
True
```

Если *какой-либо* из элементов списка «истинный», все выражение будет «истинно»:

```
>>> any([True, False, True])
True
```

Если переменной `cell` присвоено значение, отличное от ноля, а переменной `player` — значение, отличное от пустой строки, они обе «истинны»:

```
>>> cell = 1
>>> player = 'X'
>>> any([cell, player])
True
```

Функция `all()` аналогична использованию оператора `and` между всеми элементами списка. Таким образом, *все* элементы должны быть «истинны», чтобы полное выражение стало истинным:

```
>>> cell and player
'X'
```

Почему возвращается значение X? Интерпретатор возвращает последнее «истинное» значение, присвоенное переменной player, поэтому, если мы поменяем местами аргументы, мы получим значение переменной cell:

```
>>> player and cell
1
```

Применяя функцию all(), мы оцениваем истинность обоих значений, в связи с чем получим в данном примере значение True:

```
>>> all([cell, player])
True
```

Наша задача — выяснить, предоставил ли пользователь лишь *один* из аргументов --player и --cell, так как программа принимает либо оба аргумента, либо ни одного. Итак, предположим, что переменной cell присвоено значение None (по умолчанию), а переменной player — значение X. И это правда, что одно из данных значений «истинно»:

```
>>> cell = None
>>> player = 'X'
>>> any([cell, player])
True
```

Но неправда, что «истинны» они *оба:*

```
>>> all([cell, player])
False
```

Таким образом, обе функции возвращают False:

```
>>> any([cell, player]) and all([cell, player])
False
```

Поскольку принцип тот же, что и в следующем выражении:

```
>>> True and False
False
```

По умолчанию в качестве аргумента `--board` используется строка в виде девяти точек, и мы можем применить регулярное выражение, чтобы убедиться в корректности данного значения:

```
>>> board = '.' * 9
>>> import re
>>> re.search('^[.XO]{9}$', board)
<re.Match object; span=(0, 9), match='.........'>
```

Регулярное выражение создает символьный класс с шаблоном из точки (`.`) и букв `X` и `O` — то есть `[.XO]`. Число в фигурных скобках `{9}` указывает, что должно быть ровно 9 символов, а символы `^` и `$` привязывают выражение к началу и концу строки, соответственно (рис. 21.3).

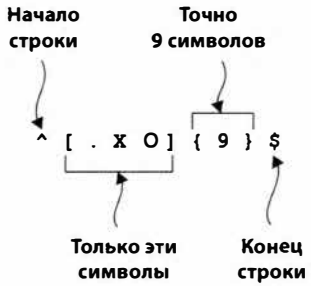


Рис. 21.3. Мы можем использовать регулярное выражение для точного описания допустимого значения аргумента `--board`

Вы можете вручную проверить выражение, снова воспользовавшись магической функцией `all()`.

— Составляет ли длина значение `board` ровно 9 символов?
— Правда ли, что каждый из символов относится к допустимым?

Вот один из способов написать код для проверки:

```
>>> board = '...XXXOOO'
>>> len(board) == 9 and all([c in '.XO' for c in board])
True
```

Функция `all()` проверяет следующее:

```
>>> [c in '.XO' for c in board]
[True, True, True, True, True, True, True, True, True]
```

Поскольку каждый символ `c` (от слова `cell` — «клетка») в значении `board` относится к числу допустимых, все сравнения результируют как `True`. Если мы изменим один из символов, то получим значение `False`:

```
>>> board = '...XXXOOA'
>>> [c in '.XO' for c in board]
[True, True, True, True, True, True, True, True, False]
```

Даже одно значение `False` в выражении `all()` вернет общее значение `False`:

```
>>> all([c in '.XO' for c in board])
False
```

Последняя часть теста проверяет, занята ли уже клетка, для которой передано значение `--player`:

```
if args.player and args.cell and args.board[args.cell - 1] in 'XO':
    parser.error(f'--cell "{args.cell}" already taken')
```

Поскольку номера клеток отсчитываются с 1, а не с 0, мы должны вычесть 1, если в аргументе `--board` ссылаемся на клетку по ее индексу. Согласно показанным ниже входным данным, первую клетку, уже занятую игроком `X`, хочет занять и игрок `O`:

```
>>> board = 'X..... .'
>>> cell = 1
>>> player = 'O'
```

С помощью кода `cell -1` можно выяснить, занята ли эта клетка в значении `board`:

```
>>> board[cell - 1] in 'XO'
True
```

Или проверить, что в данной позиции *нет* точки:

```
>>> boards[cell - 1] != '.'
True
```

Проверка входных данных — довольно утомительное занятие, но это единственный способ убедиться, что игра работает правильно.

Может возникнуть необходимость изменить значение `board` в функции `main()`, если переданы аргументы и `--cell`, и `--player`. Я решил использовать `board` как список именно потому, что мне может понадобиться изменить его следующим образом:

```
if player and cell:
    board[cell - 1] = player
```

21.2.2. Форматирование вывода

Теперь пришло время сформировать вывод. Я написал функцию, не сразу выводящую сетку, а возвращающую строковое значение, которое можно протестировать. Вот ее код:

```
def format_board(board):
    """Форматируем поле"""

    cells = [str(i) if c == '.' else c for i,
             c in enumerate(board, start=1)]
    bar = '-------------'
    cells_tmpl = '| {} | {} | {} |'
    return '\n'.join([
    bar,
        cells_tmpl.format(*cells[:3]), bar,
        cells_tmpl.format(*cells[3:6]), bar,
        cells_tmpl.format(*cells[6:]), bar
])
```

Звездочка (*) — краткая запись списка, возвращаемого после его нарезки на значения, принимаемые функцией `str.format()`.

Используем представление списка для перебора с помощью функции `enumerate()` каждой позиции и символа значения `board`. Поскольку предпочтительнее вести отсчет с номера клетки `1`, а не с позиции индекса 0, применяется опция `start=1`. Если символ — точка, выводится номер клетки; в противном случае — знак игрока, X или O.

Синтаксис со «звездочкой» позволил сократить строки кода, показанные ниже:

```
    return '\n'.join([
        bar,
        cells_tmpl.format(cells[0], cells[1], cells[2]), bar,
        cells_tmpl.format(cells[3], cells[4], cells[5]), bar,
        cells_tmpl.format(cells[6], cells[7], cells[8]), bar
])
```

Функция `enumerate()` возвращает список кортежей, содержащих индекс и значение каждого элемента списка (рис. 21.4). Поскольку это ленивая функция, в REPL-интерфейсе для просмотра значений нужно использовать функцию `list()`:

```
>>> board = 'XX.O.O...'
>>> list(enumerate(board))
[(0, 'X'), (1, 'X'), (2, '.'), (3, 'O'), (4, '.'), (5, 'O'),
 (6, '.'), (7, '.'), (8, '.')]
```

В данном случае я предпочитаю вести отсчет с 1, поэтому прибегаю к опции `start=1`:

```
>>> list(enumerate(board, start=1))
[(1, 'X'), (2, 'X'), (3, '.'), (4, 'O'), (5, '.'), (6, 'O'),
 (7, '.'), (8, '.'), (9, '.')]
```

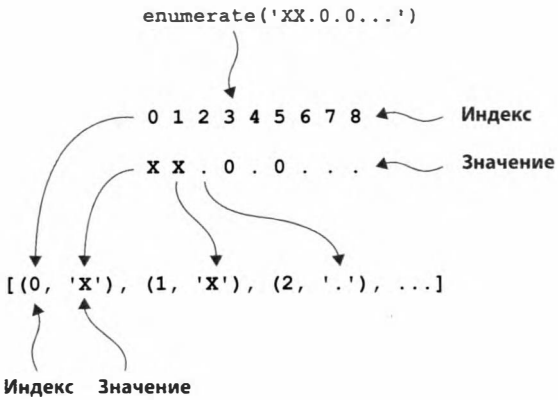


Рис. 21.4. Функция `enumerate()` возвращает индекс и значение элементов в последовательности. По умолчанию индекс первого элемента равен 0

В качестве альтернативы данное представление списка можно записать в виде цикла `for`:

```
cells = []
for i, char in enumerate(board, start=1):
    cells.append(str(i) if char == '.' else char)
```

Инициализируем пустой список для хранения значений клеток.

Распаковываем из кортежа каждого символа индекс (начиная с 1) в переменную i (от слова integer — «целое число») и значение в переменную `char`.

Если символ представляет собой точку, используем строковую версию значения `i`; в противном случае — значение типа `char`.

На рис. 21.5 показано, как результат функции `enumerate()` распаковывается в переменные `i` и `char`.

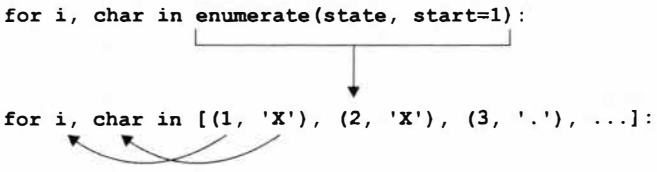


Рис. 21.5. Кортежи, содержащие индексы и значения, возвращаемые функцией `enumerate()`, присваиваются двум переменным в цикле for

Показанная версия функции `format_board()` проходит все тесты из файла *unit.py*.

21.2.3. Определение победителя

Последняя важная часть нашей программы — определение того, выиграл ли кто-нибудь из игроков, разместив три своих знака подряд по горизонтали, вертикали или диагонали.

```
def find_winner(board):
    """Возвращаем победителя"""
    winning = [[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6],
               [1, 4, 7],[2, 5, 8], [0, 4, 8], [2, 4, 6]]

    for player in ['X', 'O']:
        for i, j, k in winning:
            combo = [board[i], board[j], board[k]]
            if combo == [player, player, player]:
                return player
```

Существуют восемь выигрышных комбинаций — три горизонтальных строки, три вертикальных столбца и две диагонали, — поэтому я создал двумерный список, в котором каждый элемент также является списком, содержащим три клетки выигрышной комбинации.

Обычно переменным для хранения «целочисленных» значений присваивается имя `i`, особенно если их жизнь коротка, как в нашем коде. Если в той же области видимости необходимы похожие переменные, им обычно присваиваются имена `j`, `k`, `l` и т. д. Можно, конечно, использовать такие (более описательные) имена, как `cell1`, `cell2` и `cell3`, но их дольше набирать. Распаковка значений клеток происходит так же, как и распаковка кортежей в ранее показанном коде с функцией `enumerate()` (рис. 21.6).

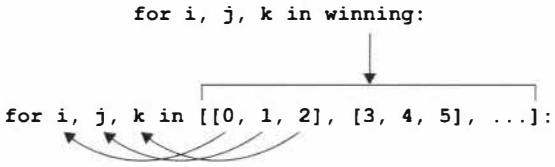
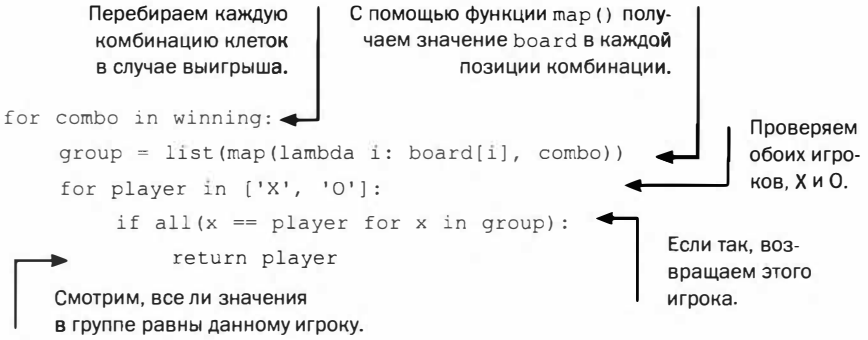


Рис. 21.6. Как и при распаковке кортежей функцией `enumerate()`, каждый список из трех элементов распаковывается в цикле `for` в три переменные

Остальная часть кода отвечает за проверку, что в каждой из трех позиций указан один определенный символ — или X, или O. Я придумал несколько способов решить данную задачу, но поделюсь лишь альтернативной версией, включающей мои любимые функции `all()` и `map()`:



Если функция не возвращает явный результат или вообще не возвращает его, как в данном случае, когда нет победителя, интерпретатор по умолчанию использует значение `None` в качестве результата. Значение `None` говорит о том, что победителя нет, о чем будет сообщено при выводе результатов игры:

```
winner = find_winner(board)
print(f' {winner} has won!' if winner else 'No winner.')
```

Вот и все, что касается варианта игры «Крестики-нолики», совершающей лишь один ход. В следующей главе мы расширим код и создадим интерактивную версию игры, которая начинается с пустого поля и динамически запрашивает у пользователя данные в процессе.

21.3. Прокачиваем навыки

— Напишите карточную игру для одной руки, например блек-джек («Двадцать одно») или «Пьяница».

РЕЗЮМЕ

— В программе используется строковое значение для представления поля игры «Крестики-нолики» с девятью символами из числа `X` и `O` (занятая клетка), а также `.` (пустая клетка). При необходимости строка преобразовывается в список для упрощения изменения поля.
— Регулярное выражение — удобный способ проверить исходное поле. Мы можем декларативно описать, что это должна быть строка длиной ровно девять символов, состоящая только из символов `.`, `X` и `O`.
— Функция `any()` похожа на группу операторов `or` между несколькими логическими значениями. Она вернет `True`, если любое из нескольких значений «истинно».
— Функция `all()` похожа на группу операторов `and` между несколькими логическими значениями. Она вернет `True`, только если все значения «истинны».
— Функция `enumerate()` возвращает индекс и значение каждого элемента в итерируемом объекте, например в списке.

Глава 22

Крестики-нолики — 2: интерактивная версия с аннотациями типов

Возвращаемся к игре «Крестики-нолики», начатой в предыдущей главе. Первая версия выполняла один ход, принимая начальное значение `--board`, а затем изменяя его с опциональными аргументами `--player` и `--cell`. Программа выводила одно поле и победителя, если он был.

Мы разовьем эти идеи и создадим версию игры, которая будет начинаться с пустого поля и совершать столько ходов, сколько необходимо для завершения партии победой одного из игроков или ничьей.

Данная программа заметно отличается от всех других упражнений книги, поскольку она не принимает аргументы командной строки. Игра всегда начинается с пустого «поля» и хода игрока X, выступающего первым. В программе используются функция `input()` для интерактивного

запроса хода у каждого игрока, сначала X, а затем O. Любой недопустимый ход, например на занятую или несуществующую клетку, должен отклоняться. После каждого хода игра может остановиться, если она определит, что возникла выигрышная комбинация или ничья.

Освоив данную главу, вы научитесь:

— запускать и выходить из бесконечного цикла;
— использовать в коде аннотации типов;
— применять и различать кортежи, именованные кортежи и типизированные словари;
— с помощью `mypy` анализировать код на наличие ошибок, в частности на неправильное использование типов.

22.1. Создание файла itictactoe.py

Это единственная программа, на которой я не запускаю интеграционный тест. Она не принимает аргументы, поэтому сложно написать тесты, динамически с ней взаимодействующие. А еще было довольно непросто выстроить диаграмму программы, поскольку ее вывод различен и зависит от сделанных ходов. Тем не менее на рис. 22.1 показано приблизительное представление о программе, начинающей работу без входных данных, а затем циклично выполняющейся до тех пор, пока не будет определен какой-либо результат или игрок не покинет игру.

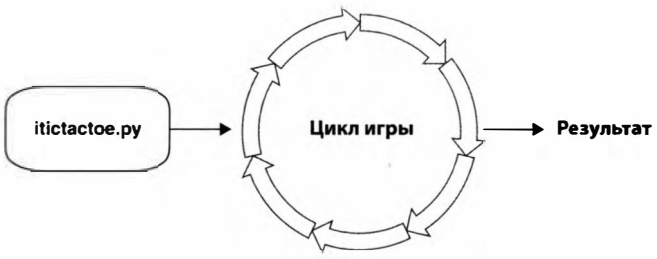


Рис. 22.1. Данная версия игры «Крестики-нолики» не принимает аргументы и воспроизводится в бесконечном цикле до достижения какого-либо результата, такого как выигрыш, ничья или поражение

Я рекомендую вам сначала запустить программу *solution1.py* и сыграть несколько раундов. Обратите внимание, что она очищает экран от любого текста и показывает поле с пустыми клетками с запросом хода игрока X. Я ввожу `1` и нажимаю клавишу **Enter**:

```
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
Player X, what is your move? [q to quit]: 1
```

Теперь клетка 1 содержит значок X, а ход запрашивается у игрока O:

```
-------------
| X | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
Player O, what is your move? [q to quit]:
```

Если я вновь выберу 1, то получу сообщение, что клетка уже занята:

```
-------------
| X | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
Cell "1" already taken
Player O, what is your move? [q to quit]:
```

Мы видим, что по-прежнему запрашивается ход игрока O, хотя предыдущий ход был недопустимым. То же самое произойдет, если ввести значение, которое нельзя преобразовать в целое число:

```
-------------
| X | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
```

```
-------------
Invalid cell "biscuit", please use 1-9
Player O, what is your move? [q to quit]:
```

Или ввести целое число, выходящее за пределы допустимого диапазона:

```
| X | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
Invalid cell "10", please use 1-9
Player O, what is your move? [q to quit]:
```

В данной версии игры используется много способов проверки ввода пользователя. Мы обсуждали их в главе 21.

Если один из игроков закроет своим знаком три клетки подряд, программа покажет выигрышную комбинацию и объявит победителя:

```
-------------
| X | O | 3 |
-------------
| 4 | X | 6 |
-------------
| 7 | O | X |
-------------
X has won!
```

22.1.1. Поговорим о кортежах

Итак, мы планируем написать интерактивную игру, которая будет начинаться с пустого поля и совершать столько ходов, сколько необходимо для завершения партии победой одного из игроков или ничьей. Тема «состояния» в предыдущей версии была ограничена игровым полем — какие игроки и какие клетки отметили. В данной версии необходимо отслеживать еще несколько переменных в состоянии игры:

— клетки поля, например `..XO..X.O`;
— текущего игрока, `X` или `O`;

— ошибки, например ввод игроком номера уже занятой или не существующей клетки или значения, которое не может быть преобразовано в число;
— выход пользователя из игры до завершения;
— ничью, когда все клетки поля заняты, но нет победителя;
— победителя, если таковой имеется, чтобы знать, когда игра окончена.

Вам необязательно писать свою программу в точности так, как это делаю я, но можно подглядывать в мое решение, поскольку придется отслеживать перечисленные переменные. Для хранения данных на ум приходит словарь, однако я хочу представить вам новую структуру данных, называемую «именованным кортежем», так как она хорошо сочетается с аннотациями типов Python, которые будут занимать важное место в моем решении.

Мы сталкивались с кортежами во многих программах книги. Они возвращаются методами типа `match.groups()`, когда регулярное выражение содержит круглые скобки групп захвата (главы 14 и 17); или функцией `zip()` при объединении двух списков (глава 19); или функцией `enumerate()` для получения списка индексов и элементов из списка. Кортеж — неизменяемый список, и мы рассмотрим, как данная особенность помогает не допускать неочевидных ошибок в программах.

Кортеж создается из значений с запятыми между ними:

```
>>> cell, player
(1, 'X')
```

Чаще всего для ясности их заключают в круглые скобки:

```
>>> (cell, player)
(1, 'X')
```

Мы можем присвоить кортеж переменной `state`:

```
>>> state = (cell, player)
>>> type(state)
<class 'tuple'>
```

А также извлекать значения из кортежа с помощью числовых индексов:

```
>>> state[0]
1
>>> state[1]
'X'
```

В отличие от списка, содержимое кортежа изменить нельзя:

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

Напомню, что в игре первая позиция — это номер клетки, а вторая — буква игрока. Когда мы добавим остальные поля, станет гораздо хуже. Можно, конечно, использовать словари и, соответственно, строки для обращения к значениям, но словари изменяемы, поэтому легко допустить ошибку в имени ключа.

22.1.2. Именованные кортежи

Было бы здорово совместить безопасность неизменяемого кортежа с именованными полями. Такими преимуществами как раз обладает функция `namedtuple()`. Для работы нужно импортировать ее из модуля `collections`:

```
>>> from collections import namedtuple
```

Функция `namedtuple()` позволяет описать новый класс для значений. Допустим, мы хотим создать класс, описывающий идею `State`. Класс — это группа переменных, данных и функций, которые могут использоваться вместе для представления объекта. В языке Python, к примеру, есть класс `str`, представляющий объект в виде последовательности символов, которая может быть присвоена переменной, имеющей некоторую длину (`len`), которая может быть преобразована в прописной регистр символов с помощью метода `str.upper()`, а также итерирована с помощью цикла `for` и т. п. Все эти возможности сгруппированы в класс `str`. Просмотреть документацию к нему можно выполнив команду `help(str)` в REPL-интерфейсе.

Первый аргумент, который мы передаем функции `namedtuple()`, — имя класса, а второй — список имен полей в классе. Обычно имена классов указываются с прописной буквы:

```
>>> State = namedtuple('State', ['cell', 'player'])
```

Мы только что создали новый тип данных с именем `State`!

```
>>> type(State)
<class 'type'>
```

Аналогично функции `list()`, применяемой для создания типа данных `list` (то есть списков), теперь мы можем использовать функцию `State()` для создания именованного кортежа типа `State`, имеющего два именованных поля, `cell` и `player`:

```
>>> state = State(1, 'X')
>>> type(state)
<class '__main__.State'>
```

Мы по-прежнему способны обращаться к полям по их индексам, как это делается в случае со списками и кортежами:

```
>>> state[0]
1
>>> state[1]
'X'
```

Однако теперь мы можем использовать и их имена, а это гораздо удобнее. Обратите внимание, что в конце строки кода нет круглых скобок, так как мы обращаемся к полю, а не вызываем метод:

```
>>> state.cell
1
>>> state.player
'X'
```

Поскольку `state` — это кортеж, после его создания мы не можем изменить его значение:

```
>>> state.cell = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Что на самом деле *прекрасно*. Зачастую довольно рискованно изменять значения после запуска программы. Поэтому рекомендую вам использовать обычные или именованные кортежи всякий раз, когда вам нужна структура, подобная словарю или списку, которую нельзя случайно изменить.

Однако есть проблема, заключающаяся в том, что ничто не мешает нам создать экземпляр кортежа `state` с полями не по порядку *и неправильными типами,* ведь `cell` — это обязательно целое число, а `player` — строка!

```
>>> state2 = State('O', 2)
>>> state2
State(cell='O', player=2)
```

Избежать этого можно использовав имена полей, чтобы их порядок больше не имел значения:

```
>>> state2 = State(player='O', cell=2)
>>> state2
State(cell=2, player='O')
```

Теперь у вас есть структура данных, похожая на словарь, но неизменная как кортеж!

22.1.3. Добавление аннотаций типа

Мы все еще не избавились от большой проблемы, заключающейся в том, что ничто не мешает нам присвоить строку переменной `cell`, которая должна хранить целое число, и, наоборот, целое число строковой переменной `player`:

```
>>> state3 = State(player=3, cell='X')
>>> state3
State(cell='X', player=3)
```

Начиная с версии Python 3.6, модуль `typing` позволяет добавлять *аннотации* (*или подсказки*) *типов* для описания типов данных для переменных. Обратитесь к документу PEP 484 (www.python.org/dev/peps/pep-0484) для получения исчерпывающей информации, но, если вкратце, суть в том, что мы можем использовать данный модуль для описания соответствующих типов для переменных и сигнатур для функций.

Я попробую улучшить класс `State` с помощью класса `NamedTuple` из модуля `typing`, применив его в качестве базового класса. Сначала необходимо импортировать из модуля `typing` необходимые классы, такие как `NamedTuple`, `List` и `Optional`. Последний описывает

тип, который может быть None или любым другим классом, например str:

```
from typing import List, NamedTuple, Optional
```

Теперь мы способны использовать класс State с именованными полями, типами и даже дефолтными значениями для представления начального состояния игры, когда игровое поле пусто (девять точек) и игрок X ходит первым. Обратите внимание, что board — это не строка, а список символов:

```
class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
    quit: bool = False
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None
```

Попробуем применить функцию State() для создания нового значения начального состояния:

```
>>> state = State()
>>> state.board
['.', '.', '.', '.', '.', '.', '.', '.', '.']
>>> state.player
'X'
```

Можно переопределить любое дефолтное значение, указав имя поля и значение. Например, мы способны начать игру с игрока O, указав код player='O'. Любое поле, которое мы опустим, будет иметь дефолтное значение:

```
>>> state = State(player='O')
>>> state.board
['.', '.', '.', '.', '.', '.', '.', '.', '.']
>>> state.player
'O'
```

Программа выбросит исключение, если мы ошибемся в имени поля, например playre вместо player:

```
>>> state = State(playre='O')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() got an unexpected keyword argument 'playre'
```

22.1.4. Проверка типа с помощью расширения mypy

Как бы ни прекрасно звучало все вышесказанное, Python *не сгенерирует ошибку выполнения, если мы присвоим неверный тип.* Например, я могу присвоить переменной `quit` строковое значение `'True'` вместо логического `True`, и вообще ничего не произойдет:

```
>>> state = State(quit='True')
>>> state.quit
'True'
```

Преимущество аннотаций типов заключается в возможности использования такого инструмента, как Муру, для проверки кода. Показанный ниже код вы найдете в репозитории в небольшой программе *typehints.py*:

```
#!/usr/bin/env python3
""" Пример аннотаций типов """

from typing import List, NamedTuple, Optional

class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
    quit: bool = False
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None

state = State(quit='False')

print(state)
```

Переменная `quit` определена как логическая, то есть принимающая только значение `True` или `False`.

Мы присваиваем переменной строковое значение `'True'` вместо логического `True`. Такую ошибку очень легко допустить, особенно в большой и сложной программе. Хотелось бы обнаруживать ошибки подобного рода!

Программа будет работать *без ошибок*:

```
$ ./typehints.py
State(board=['.', '.', '.', '.', '.', '.', '.', '.', '.'],
player='X', \
quit='False', draw=False, error=None, winner=None)
```

Но инструмент Муру сообщит об ошибке в типе данных:

```
$ mypy typehints.py
typehints.py:16: error: Argument "quit" to "State" has
incompatible type "str"; expected "bool"
Found 1 error in 1 file (checked 1 source file)
```

Если я исправлю код следующим образом:

```
#!/usr/bin/env python3
""" Пример аннотаций типов """

from typing import List, NamedTuple, Optional

class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
    quit: bool = False
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None

state = State(quit=True)

print(state)
```

Обратите внимание, теперь это логическое значение.

Присваиваем корректное логическое значение, чтобы пройти проверку инструментом Муру.

Инструмент Муру сообщит об отсутствии проблем:

```
$ mypy typehints2.py
Success: no issues found in 1 source file
```

22.1.5. Обновление неизменяемых структур

Если одним из преимуществ именованных кортежей является их *неизменяемость*, как нам отслеживать изменения в нашей программе? Рассмотрим начальное состояние пустого поля с игроком X, который ходит первым:

```
>>> state = State()
```

Представьте, что X ходит в клетку 1, поэтому необходимо сменить значение переменной `board` на `X.........`, а игрока (переменная `player`) на O.

Мы не можем непосредственно изменить состояние (переменную `state`):

```
>>> state.board=list('X.........')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Можно с помощью функции `State()` создать новое значение, перезаписав существующее переменной `state`. То есть, поскольку нам нельзя ничего изменить в переменной `state`, мы способны присвоить ей совершенно новое значение. Также мы поступали при рассмотрении второго способа в главе 8, когда необходимо было изменить строковое значение, ведь строки в Python также неизменяемы.

Итак, мы можем скопировать все текущие значения, которые не изменились, и объединить их с измененными значениями:

```
>>> state = State(board=list('X.........'), player='O',
\ quit=state.quit, draw=state.draw, error=state.error, \
winner=state.winner)
```

Однако есть гораздо более простой способ решить эту задачу — метод `namedtuple._replace()`. Изменяются только значения, которые мы предоставляем, и результатом является новое состояние — значение `State`:

```
>>> state = state._replace(board=list(
'X.........'), player='O')
```

Мы перезаписываем значение переменной `state` результатом, возвращенным методом `state._replace()`, точно так же, как неоднократно перезаписывали новыми значениями строковые переменные:

```
>>> state
State(board=['X', '.', '.', '.', '.', '.', '.', '.', '.', '.'], \
player='O', quit=False, draw=False, error=None, winner=None)
```

Согласитесь, это гораздо удобнее, чем перечислять все поля, — нужно указать лишь те, которые изменились. Кроме того, код защищен от случайного изменения прочих полей, а мы избавлены от лишней головной боли. Теперь нет риска что-нибудь забыть, неправильно написать имена каких-нибудь полей или присвоить им данные неправильного типа.

22.1.6. Добавление аннотаций типов в определения функций

Теперь рассмотрим, как добавить аннотации типов в определения наших функций. К примеру, добавив параметр `board` в функцию `format_board()`, мы можем указать что она принимает список строковых значений `List[str]`. Кроме того, функция возвращает строковое значение, поэтому после двоеточия в определении функции мы способны добавить код `-> str`, как показано на рис. 22.2.

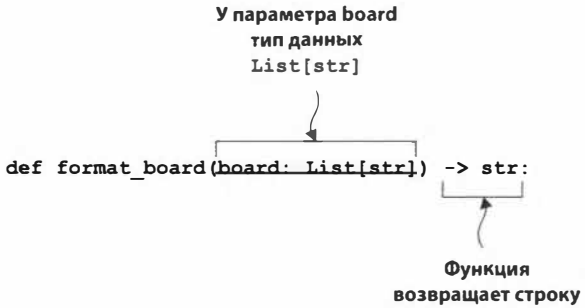


Рис. 22.2. Добавление аннотаций типов для указания типов параметра и возвращаемого значения

Аннотация в функции `main()` указывает, что возвращается значение `None`, как показано на рис. 22.3.

Рис. 22.3. Функция `main()` не принимает параметров и возвращает значение `None`

Но по-настоящему здорово, что мы можем определить функцию, принимающую значение типа `State`, и Муру проверит, действительно ли ей передается такое значение (рис. 22.4).

Сыграйте в мой вариант игры, а затем напишите свой, ведущий себя так же. Затем изучите мое решение для интерактивной версии игры, в котором реализуются концепции неизменности данных и безопасности типов.

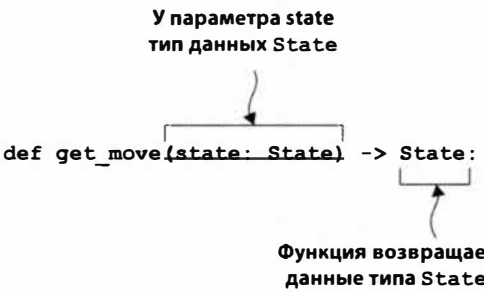


Рис. 22.4. В аннотациях типов мы можем указывать пользовательские типы данных. Показанная функция принимает и возвращает значения типа `State`

22.2. Решение

Все! Это последняя программа! Я надеюсь, что в процессе работы над простой версией игры в предыдущей главе у вас уже появились кое-какие мысли по поводу разработки интерактивного варианта. Что скажете, помогли вам аннотации типов и модульные тесты?

```
#!/usr/bin/env python3
""" Крестики-нолики - Интерактивная версия """

from typing import List, NamedTuple, Optional

class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
```

Импортируем необходимые классы из модуля `typing`.

Объявляем производный класс, созданный на основе базового класса `NamedTuple`. Данная процедура называется *наследованием классов*. Определяем имена полей, типы и дефолтные значения, которые может содержать производный класс.

```
    quit: bool = False
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None

# ------------------------------------------------
def main() -> None:
    """Да грянет джаз!"""

    state = State()

    while True:
        print("\033[H\033[J")
        print(format_board(state.board))

        if state.error:
            print(state.error)
        elif state.winner:
            print(f' {state.winner} has won!')
            break

        state = get_move(state)
        if state.quit:
            print('You lose, loser!')
            break
        elif state.draw:
            print("All right, we'll call it a draw.")
            break

# ------------------------------------------------
def get_move(state: State) -> State:
    """Ход игрока"""

    player = state.player
```

Запускаем бесконечный цикл. Если возникает необходимость остановиться, прерываем его.

Выводим специальную последовательность, которую большинство терминалов интерпретирует как команду очистки экрана.

Создаем экземпляр начального состояния — пустое поле и назначаем, что первым ходит игрок X.

Выводим текущее состояние поля.

Выводим все ошибки, например указание пользователем занятой или неправильной клетки.

Если определен победитель, объявляем его и прерываем цикл.

Получаем следующий ход от игрока. Функция `get_move()` принимает тип `State` и возвращает его же. Мы перезаписываем значение переменной `state` при каждой итерации цикла.

При преждевременном выходе пользователя из игры сообщаем ему о проигрыше и прерываем цикл.

В случае отсутствия в игре доступных ходов (все клетки заняты) и победителя, объявляем ничью и прерываем цикл.

Определяем функцию `get_move()`, принимающую и возвращающую данные типа `State`.

Копируем переменную `player` из `state`, так как мы будем ссылаться на нее несколько раз в теле функции.

Если это так, присваиваем переменной `quit` значение `True` и возвращаем новое состояние. Обратите внимание, что никакие другие значения в состоянии не изменяются.

Проверяем, ввел ли пользователь значение, которое можно преобразовать в число с помощью метода `str.isdigit()`, и находится ли данное число в допустимом диапазоне.

С помощью функции `input()` запрашиваем у игрока следующий ход. Инструктируем, как выйти из игры до завершения программы, чтобы не приходилось нажимать сочетание клавиш **Ctrl+C** для прерывания программы.

```
    cell = input(f'Player {player}, what is your move?
                 [q to quit]: ')

    if cell == 'q':
        return state._replace(quit=True)

    if not (cell.isdigit() and int(cell) in range(1, 10)):
        return state._replace(error=f'Invalid cell "{cell}",
                              please use 1-9')

    cell_num = int(cell)
    if state.board[cell_num - 1] in 'XO':
        return state._replace(error=f'Cell "{cell}" already taken')

    board = state.board
    board[cell_num - 1] = player
    return state._replace(board=board,
                          player='O' if player == 'X' else 'X',
                          winner=find_winner(board),
                          draw='.' not in board,
                          error=None)
```

Сначала проверяем, хочет ли пользователь выйти.

Если нет, возвращаем обновленное значение `state` с ошибкой. Опять же, больше ничего в состоянии не меняется, поэтому мы повторяем раунд с тем же игроком и с тем же стоянием.

Проверяем, пуста ли указанная клетка поля.

После проверки, что значение переменной `cell` является допустимым целым числом, преобразуем его в тип `int`.

Если нет, возвращаем обновленное значение `state` с ошибкой. Опять же, больше ничего в состоянии не меняется, поэтому мы повторяем раунд с тем же игроком и с тем же стоянием.

Возвращаем новое значение `state`, если изменено значение `board`, переменная `player` переключена на противоположного игрока и если есть победитель или произошла ничья.

Учитывая значение `cell`, заменяем значение `board` на текущее значение переменной `player`.

Копируем текущее значение `board`, потому что необходимо изменить его, а `state.board` неизменно.

```
# ------------------------------------------------
def format_board(board: List[str]) -> str:
    """Форматируем поле"""

    cells = [str(i) if c == '.' else c for i,
             c in enumerate(board, 1)]
    bar = '-------------'
    cells_tmpl = '| {} | {} | {} |'
    return '\n'.join([
        bar,
        cells_tmpl.format(*cells[:3]), bar,
        cells_tmpl.format(*cells[3:6]), bar,
        cells_tmpl.format(*cells[6:]), bar
])
```

Единственное отличие данной версии функции от предыдущей — добавленные аннотации типов. Функция принимает список строковых значений (текущее значение `board`) и возвращает отформатированную сетку состояния поля.

```
# ------------------------------------------------
def find_winner(board: List[str]) -> Optional[str]:
    """Возвращаем победителя"""

    winning = [[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6],
               [1, 4, 7], [2, 5, 8], [0, 4, 8], [2, 4, 6]]

    for player in ['X', 'O']:
        for i, j, k in winning:
            combo = [board[i], board[j], board[k]]
            if combo == [player, player, player]:
    return player

return None
```

Перед вами та же функция, что и раньше, но с аннотациями типов. Функция принимает значение `board` в виде списка строк и опционально возвращает строковое значение, т. е. она также может вернуть значение `None`.

```
# ------------------------------------------------
if __name__ == '__main__':
    main()
```

22.2.1. Версия программы с классом TypedDict

В версии Python 3.8 появился класс `TypedDict`, очень похожий на `NamedTuple`. Давайте посмотрим, как применение данного класса в качестве базового повлияет на код нашей программы. Одно из важных отличий — вы не можете (пока) назначить дефолтные значения полям:

```
#!/usr/bin/env python3
""" Крестики-нолики -- Интерактивная версия """

from typing import List, Optional, TypedDict

class State(TypedDict):
    board: str
    player: str
    quit: bool
    draw: bool
    error: Optional[str]
    winner: Optional[str]
```

Импортируем класс `TypedDict` вместо `NamedTuple`.

Наследуем производный класс `State` на базовом классе `TypedDict`.

Мы должны настроить начальные значения при присвоении переменной `state` экземпляра класса:

```
def main() -> None:
    """Да грянет джаз!"""

    state = State(board='.' * 9,
                  player='X',
                  quit=False,
                  draw=False,
                  error=None,
                  winner=None)
```

С точки зрения синтаксиса я предпочитаю использовать метод `state.board` с именованным кортежем, а не доступ к словарю `state['board']`:

```
while True:
    print("\033[H\033[J")
    print(format_board(state['board']))

    if state['error']:
        print(state['error'])
    elif state['winner']:
        print(f" {state['winner']} has won!")
        break

    state = get_move(state)
```

```
    if state['quit']:
        print('You lose, loser!')
        break
    elif state['draw']:
        print('No winner.')
        break
```

Помимо удобного доступа к полям, мне нравится неизменяемость класса `NamedTuple`, в отличие от изменяемого `TypedDict`. Обратите внимание, как в функции `get_move()` меняется состояние:

```
def get_move(state: State) -> State:
    """Ход игрока"""

    player = state['player']
    cell = input(f'Player {player}, what is your move? [q to
                 quit]: ')

    if cell == 'q':
        state['quit'] = True
        return state

    if not (cell.isdigit() and int(cell) in range(1, 10)):
        state['error'] = f'Invalid cell "{cell}", please use 1-9'
        return state

    cell_num = int(cell)
    if state['board'][cell_num - 1] in 'XO':
        state['error'] = f'Cell "{cell}" already taken'
        return state

    board = list(state['board'])
    board[cell_num - 1] = player

    return State(
        board=''.join(board),
        player='O' if player == 'X' else 'X',
        winner=find_winner(board),
        draw='.' not in board,
        error=None,
        quit=False,
)
```

Здесь мы напрямую вносим изменения в `TypedDict`, тогда как в `NamedTuple` используется метод `state._replace()` для возврата совершенно нового значения `state`.

Еще одна позиция, где значение `state` можно изменить напрямую. Вы можете выбрать и такой подход.

На мой взгляд, класс `NamedTuple` имеет ряд преимуществ — более приятный синтаксис, возможность настраивать дефолтные значения и неизменность по сравнению с `TypedDict`, так что я предпочитаю его. Независимо от того, что выберете вы, важно, чтобы мы старались явно указывать «состояние» программы, а также то, когда и как оно меняется.

22.2.2. Вкратце о состояниях программы

Идея состояния программы заключается в том, что она способна запоминать изменения переменных с течением времени. В предыдущей главе наша программа принимала значение `--board` и значения аргументов `--cell` и `--player`, позволяющие изменить значение переменной `board`. Тогда игра выводила представление поля. В интерактивной версии программы при запуске поле всегда пусто и значение переменной `board` меняется с каждым ходом игрока, что смоделировано нами как бесконечный цикл.

В подобных программах разработчики часто используют *глобальные переменные,* которые объявляются в верхней части кода программы вне каких-либо определений функций, чтобы они были *глобально* доступны во всей программе. Хотя это обычная практика, прибегать к ней не рекомендуется. Я бы не советовал вам использовать глобальные переменные, только если нет другого выхода. Лучше писать небольшие функции, которые принимают все требуемые значения и возвращают значение одного типа. Я бы также посоветовал использовать подходящие структуры данных, например типизированные именованные кортежи, для представления состояния программы, и очень тщательно защищать изменения состояния.

22.3. Прокачиваем навыки

— Попробуйте иначе «отругать» прерывающего игру пользователя. Вспомните генератор шекспировских оскорблений.
— Создайте версию, в которой пользователь может начать новую игру, не выходя из программы и не перезапуская ее.
— Напишите другие игры, например «Виселицу».

РЕЗЮМЕ

— Аннотации типов позволяют аннотировать типами значений переменные, а также параметры функций и возвращаемые значения.
— Интерпретатор Python игнорирует аннотации типов во время выполнения, поэтому следует использовать инструмент Муру для поиска ошибок, связанных с типами данных, в коде до того, как вы его запустите.
— Именованный кортеж слегка похож на словарь и немного на объект, и при этом обладает неизменностью кортежей.
— Оба класса, `NamedTuple` и `TypedDict`, позволяют создать новый тип с определенными полями и типами данных, которые можно использовать для аннотирования ваших собственных функций.
— В нашей программе применяется класс `NamedTuple` для создания сложной структуры данных, представляющей состояние программы. Состояние включает множество аннотированных переменных, таких как текущая комбинация на поле, текущий игрок, разные ошибки, победитель и проч.
— Хотя писать интеграционные тесты для интерактивных программ непросто, мы все же можем разбить программу на небольшие функции (например, `format_board()` или `get_winner()`) и создать для них модульные тесты.

Эпилог

Ну вот и все. Мы прошли долгий путь от написания программы «Вороньe гнездо» в главе 2 до интерактивной игры «Крестики-нолики» в главе 22, создав собственный класс на основе именованных кортежей и использовав аннотации типов. Надеюсь, теперь вам понятно, как много всего можно сотворить со строками, списками, кортежами, словарями, множествами и функциями Python. И надеюсь, что убедил вас писать программы:

— *гибкие,* принимающие аргументы командной строки;
— *документированные* с применением модулей типа `argparse` для анализа аргументов и формирования инструкций по использованию;
— *тестируемые* с помощью как *модульных* тестов для функций, так и *интеграционных* тестов для программы в целом.

Пользователи ваших программ обязательно оценят инструкции по их использованию и поведению. Они непременно оценят и то, что вы потратили время на тестирование своих программ. Но буду честным. Человеком, который, скорее всего, будет запускать и модифицировать ваши программы через несколько месяцев, будете вы сами. Я слышал, как говорят, что «документация — это любовное письмо самому себе в будущее». Все усилия, которые вы потратите на свои программы, весьма высоко будут оценены именно вами при возвращении к собственному коду.

Теперь, когда вы справились со всеми упражнениями и научились использовать написанные мной тесты, предлагаю открыть репозиторий и разобрать код в файлах *test.py*. Если вы намереваетесь следовать

аспектам разработки через тестирование, то наверняка сможете позаимствовать многие идеи и приемы из этих файлов.

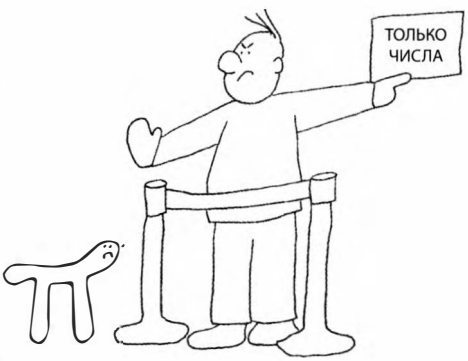
Кроме того, каждая глава содержит предложения по прокачке ваших навыков и расширению представленных идей и упражнений. Откройте их и подумайте, как можно использовать концепции, описанные в книге, для улучшения или расширения программ из разных глав. Ниже еще несколько идей.

- Глава 2 («Воронье гнездо»). Добавьте возможность рандомного выбора приветствия, отличного от Hello, из следующего списка: Привет, Hola, Salut и Ciao.
- Глава 3 («Айда на пикник»). Пусть программа сама выбирает один или несколько предметов для пикника и добавляет их к вашему списку с корректными знаками препинания и серийной запятой.
- Глава 7 («Ужасная азбука»). Скачайте книгу *The Devil's Dictionary* (*«Словарь Сатаны»*) Амброза Бирса с сайта Project Gutenberg. Напишите программу, которая будет искать определение слова, если оно встречается в тексте.
- Глава 16 («Скремблер»). Возьмите принцип скремблирования текста за основу для шифрования сообщений. Напишите скремблированные слова прописными буквами, удалите все знаки препинания и пробелы, а затем отформатируйте текст в «слова» из пяти символов, за которыми следует пробел, по пять «слов» на строку. Пусть последнюю строку текст заполняет полностью. Сможете понять вывод?
- *new.py*. Впервые я написал код для создания новой программы, будучи желторотым хакером Perl-приложений. Моя программа `new-pl` добавляла случайную цитату из произведений Уильяма Блейка (да-да, не вру — я также увлекался Бронте и Дикинсон). Сделайте так, чтобы в вашей версии программы *new.py* добавлялись случайные цитаты или шутки из тех произведений, которые нравятся вам, либо еще как-то настройте ее для создания других программ.

Надеюсь, вы получили такое же удовольствие от выполнения упражнений, какое при их разработке для вашего обучения испытывал я. Я счастлив, если вы рады тому, что у вас теперь есть десятки программ и тестов с идеями и функциями, которые можно использовать для создания еще более крутых приложений.

Всего вам наилучшего в путешествии по миру программирования!

Приложение. Модуль argparse



Зачастую получение правильных данных в программе становится страшной рутиной. Модуль `argparse` значительно упрощает проверку пользовательских аргументов и вывод полезных сообщений об ошибках, когда пользователи вводят неверные данные. Это как «вышибала» на входе вашей программы, позволяющий вводить в нее только корректные значения. Правильное определение аргументов с помощью модуля `argparse` — это важный первый шаг к тому, чтобы программы, описанные в данной книге, работали должным образом.

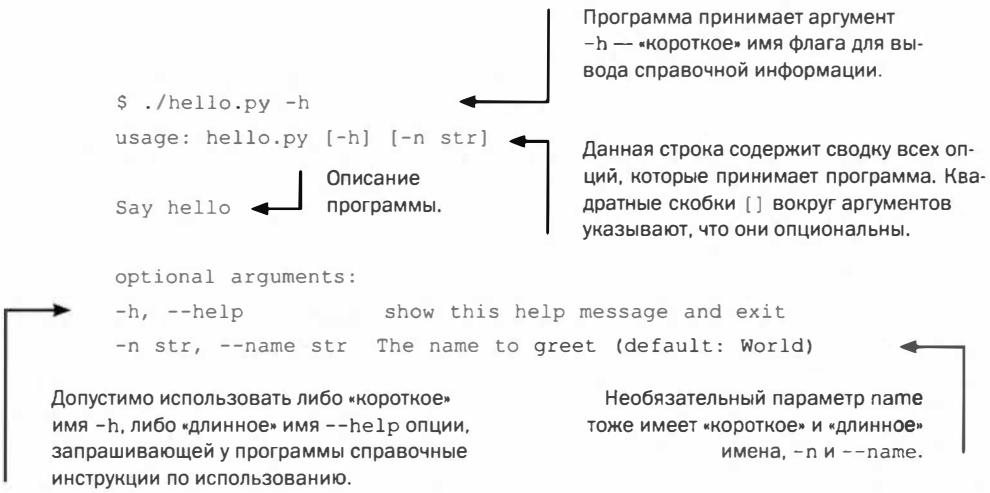
К примеру, в главе 1 обсуждается весьма гибкая программа, способная распространить свое теплое приветствие на опциональную именованную сущность, например на «Мир» или «Вселенную»:

```
$ ./hello.py
Hello, World!
$ ./hello.py --name Universe
Hello, Universe!
```

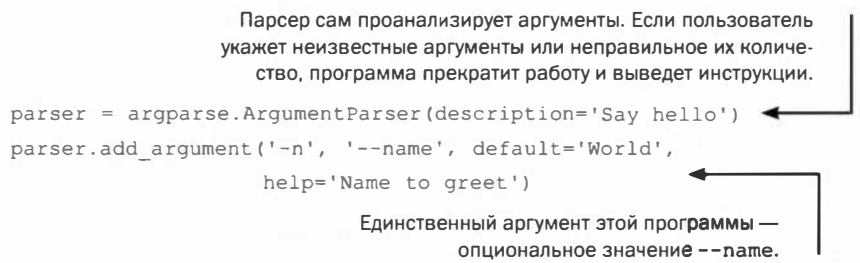
При запуске без входных данных используется слово World в качестве сущности для приветствия.

Программа принимает опциональное значение `--name`, переопределяющее дефолтное название сущности.

Она реагирует на флаг `-h` и `--help` полезными инструкциями:



Необходимый код умещается в две строки:



Примечание. Вам не нужно определять флаги `-h` или `--help`. Они генерируются автоматически силами модуля `argparse`. На практике никогда не следует использовать их в другом контексте, поскольку это привычные опции, поведение которых знакомо большинству пользователей.

Модуль `argparse` помогает определить синтаксический анализатор (парсер) для аргументов и генерирует справочные сообщения, экономя массу времени разработчикам и придавая программам профессиональный оттенок. Каждая программа из этой книги тестируется с различными входными данными, так что, прочтя ее всю, вы должны разобраться, как использовать данный модуль. Я рекомендую просмотреть документацию к модулю `argparse` (**https://docs.python.org/3/library/argparse.html**).

Теперь рассмотрим, от каких рутинных действий может избавить нас модуль. Освоив данное приложение, вы научитесь:

— применять модуль `argparse` для обработки позиционных параметров, опций и флагов;
— устанавливать дефолтные значения для параметров;
— использовать ключевое слово `type` для того, чтобы пользователь передавал данные корректного типа, например числа или файлы;
— с помощью ключевого слова `choices` ограничивать значения параметров.

A.1. Типы аргументов

Аргументы командной строки можно классифицировать следующим образом.

— *Позиционные аргументы* — порядок и количество аргументов определяют их предназначение. Некоторые программы ожидают, к примеру, имя файла в качестве первого аргумента и выходной каталог в качестве второго. Позиционные аргументы обычно обязательны (не опциональны). Сделать их опциональными сложно. Как бы вы написали программу, принимающую два-три аргумента, где второй и третий независимы и опциональны? В первой версии программы *hello.py* в главе 1 приветственное имя передается в качестве позиционного аргумента.
— *Именованные аргументы* — большинство консольных программ определяют *короткое* имя, допустим, `-n` (один дефис и одна латинская буква), и *длинное*, допустим, `--name` (два дефиса и слово), за которым следует некоторое значение, например имя в программе *hello.py*. Именованные аргументы можно указывать в любом порядке — их *позиции* не имеют значения. Поэтому именно их следует выбирать в случае, если программа может работать и без них (в конце концов, они же *опциональны)*. Рекомендуется присваивать опциональным параметрам разумные дефолтные значения. Когда мы изменили обязательный позиционный аргумент name в программе *hello.py* на опциональный `--name`, мы использовали слово World по умолчанию, чтобы программа могла работать без обязательного ввода имени пользователем. Обратите внимание, что некоторые другие языки программирования, такие как Java, поддерживают длинные имена аргументов с одним дефисом, например `-jar`.
— *Флаги* — логические параметры, имеющие одно значение из двух, например True или False. Они напоминают именованные аргументы, однако после имени не указывается значение, как, например, в случае с флагом `-d` или `--debug`, использующимся, чтобы включить

отладку. Обычно наличие флага указывает, что данному аргументу присвоено значение `True`, а его отсутствие — значение `False`. Так, наличие флага `--debug` включает отладку, а если его не указать, то и отладки нет.

A.2. Создание программ по шаблону

Довольно сложно запомнить синтаксис для определения параметров с помощью модуля `argparse`, поэтому я подготовил для вас шаблон, на основе которого вы можете создавать новые программы. Он включает показанную выше и другие структуры, упрощающие чтение кода и выполнение программ.

Один из способов создать новую программу — использовать файл *new.py*. В корне репозитория выполните следующую команду:

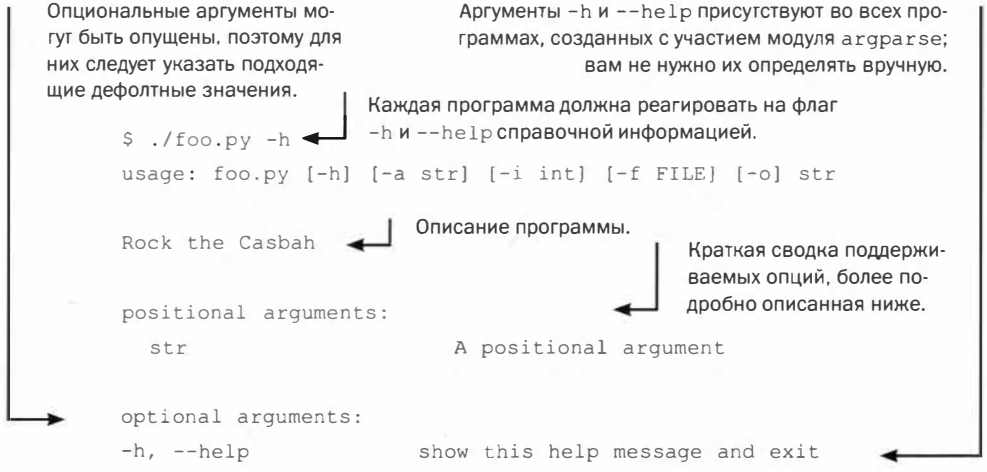
```
$ bin/new.py foo.py
```

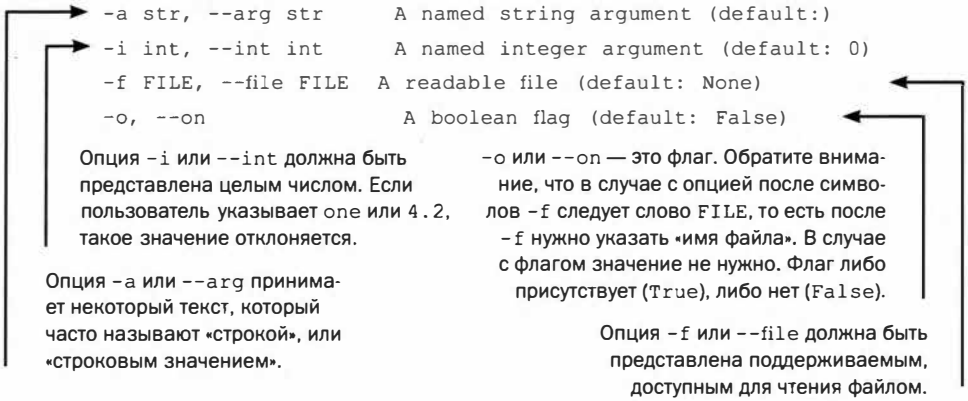
Кроме того, можно скопировать шаблон в новый файл:

```
$ cp template/template.py foo.py
```

Код получившейся программы будет идентичен шаблону независимо от того, как вы ее создадите, а еще в ней вы найдете объявления различных аргументов, описанных в предыдущем разделе. Кроме того, с помощью модуля `argparse` можно проверять входные данные, например чтобы убедиться, что один аргумент является числом, а другой — файлом.

Давайте взглянем на инструкции, сгенерированные новой программой:





A.3. Использование модуля argparse

Код для генерации инструкций, показанных выше, находится в функции `get_args()` и выглядит следующим образом:

```
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Rock the Casbah',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('positional',
                        metavar='str',
                        help='A positional argument')

    parser.add_argument('-a',
                        '--arg',
                        help='A named string argument',
                        metavar='str',
                        type=str,
                        default='')

    parser.add_argument('-i',
                        '--int',
                        help='A named integer argument',
                        metavar='int',
                        type=int,
                        default=0)
```

```
    parser.add_argument('-f',
                        '--file',
                        help='A readable file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        default=None)

    parser.add_argument('-o',
                        '--on',
                        help='A boolean flag',
                        action='store_true')

    return parser.parse_args()
```

Можно разместить данный код где угодно, однако определение и проверка аргументов иногда занимают довольно много времени. Обычно я помещаю такой код в функцию `get_args()` и всегда определяю ее самой первой в своей программе. Таким образом я сразу вижу ее, когда читаю исходный код.

Функция `get_args()` определяется следующим образом:

Ключевое слово `def` определяет новую (именованную) функцию, а аргументы функции перечисляются в скобках. Несмотря на то что функция `get_args()` не принимает аргументы, круглые скобки все равно обязательны.

```
def get_args():
    """Получаем аргументы командной строки"""
```

После определения функции в тройных скобках указана так называемая строка документации, которая документирует функцию. Строки документации указывать необязательно, но рекомендуется. Кроме того, если вы этого не сделаете, вас отругает линтер типа Pylint.

А.3.1. Создание парсера

Показанный ниже фрагмент кода создает *синтаксический анализатор* (*парсер*), обрабатывающий аргументы из командной строки. «Анализ (парсинг)» здесь означает извлечение какого-то значения по порядку и синтаксису текста, предоставленного в качестве аргументов:

Вызываем функцию `argparse.ArgumentParser()` для создания нового парсера.

```
parser = argparse.ArgumentParser(
    description='Argparse Python script',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)
```

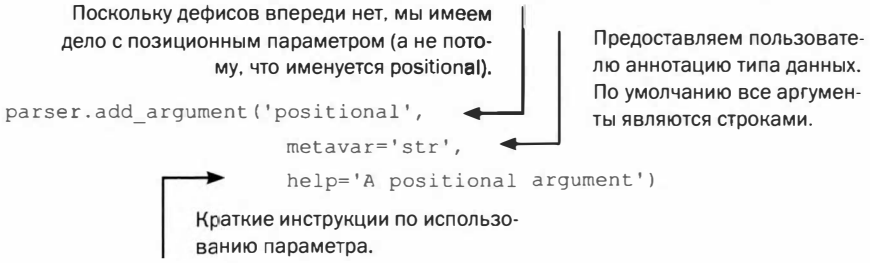
Короткое описание программы.

Благодаря аргументу `formatter_class` модуль `argparse` показывает дефолтные значения в документации.

Прочитайте документацию к модулю `argparse`, чтобы узнать про все прочие опции, которые можно использовать для определения парсера или параметров. Для этого выполните команду `help(argparse)` в REPL-интерфейсе или воспользуйтесь ссылкой **https://docs.python.org/3/library/argparse.html.**

A.3.2. Создание позиционного параметра

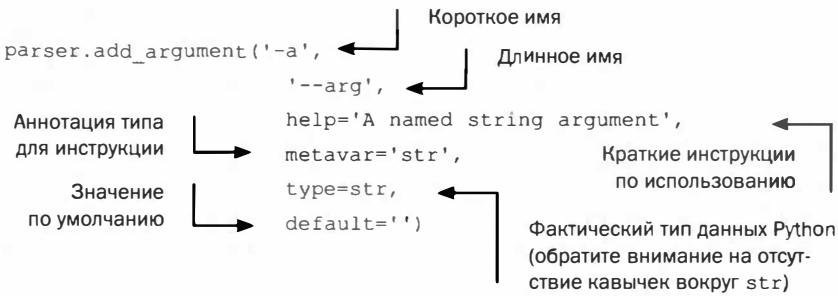
Показанная ниже строка создает новый *позиционный* параметр:



Помните, что позиционный параметр таков не потому, что имеет *имя* positional. В данном коде оно указано лишь для того, чтобы напомнить вам, что речь идет о *позиционном* параметре. Модуль `argparse` интерпретирует строку `position` как позиционный параметр не из-за его имени, а потому что *перед именем нет дефисов.*

A.3.3. Создание опционального строкового параметра

Показанная ниже строка создает *необязательный* (*опциональный*) параметр с коротким именем `-a` и длинным именем `--arg`. Это строковое (`str`) с дефолтным значением `''` (пустая строка).

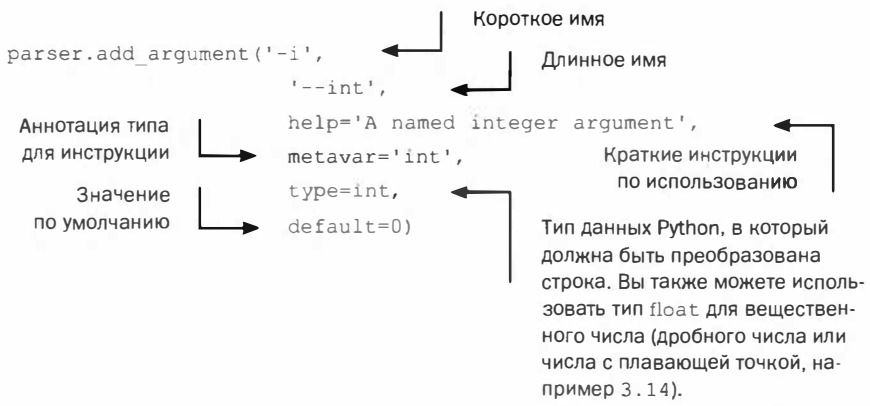


Примечание. Можно не указывать оба имени, короткое и длинное, но лучше все-таки это делать. Большинство тестов в данной книге проверяют программы в том числе на наличие как коротких, так и длинных имен опций.

Если вы хотите превратить опциональный параметр в обязательный именованный, нужно удалить строку `default` и добавить строку `required=True`.

A.3.4. Создание опционального числового параметра

Показанная ниже строка создает параметр с именем `-i` или `--int`, принимающий `int` (целое число) со значением по умолчанию 0. Если пользователь указывает значение, которое не может быть интерпретировано как целое число, модуль `argparse` прекращает обработку аргументов и выводит сообщение об ошибке и краткие инструкции.

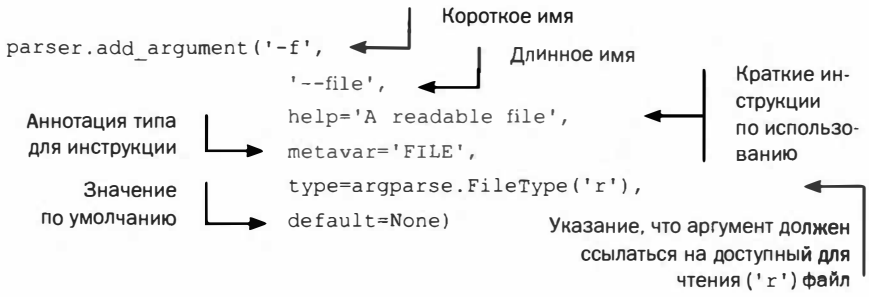


Почему важно определять числовые аргументы таким образом? Дело в том, что модуль `argparse` преобразует введенное пользователем значение в правильный тип данных. Все значения, поступающие из командной строки, являются строками, и задача программы — преобразовать их в фактические числовые значения. Если вы напишете, что параметр должен быть `type=int`, его значение, когда вы запросите его у парсера, будет преобразовано в фактическое целое число.

Если значение, предоставленное пользователем, не может быть преобразовано в целое число (тип `int`), оно будет отклонено. Обратите внимание, что вы также можете использовать код `type=float` для приема и преобразования введенного значения в вещественное число. Это сэкономит много времени и усилий.

A.3.5. Создание опционального файлового параметра

Продемонстрированная ниже строка создает параметр с именем `-f` или `--file`, который принимает только допустимый, доступный для чтения файл. Один только этот аргумент бесценен, так как сэкономит вам кучу времени на проверку ввода пользователя. Отмечу, что в данной книге почти все программы, принимающие файлы в качестве входных данных, сопровождаются тестами, включающими передачу *недопустимых* файловых аргументов, чтобы гарантировать, что программа их отклонит.



Пользователь, запускающий программу, отвечает за предоставление ей файла. Например, если вы создали программу *foo.py* в корне репозитория, там же будет размещен файл *README.md*. Мы могли бы использовать его в качестве входных данных для нашей программы, и он был бы принят как допустимый аргумент:

```
$ ./foo.py -f README.md foo
str_arg = ""
int_arg = "0"
file_arg = "README.md"
flag_arg = "False"
positional = "foo"
```

Если мы предоставим недопустимый файловый аргумент, например слово blargh, то получим сообщение об ошибке:

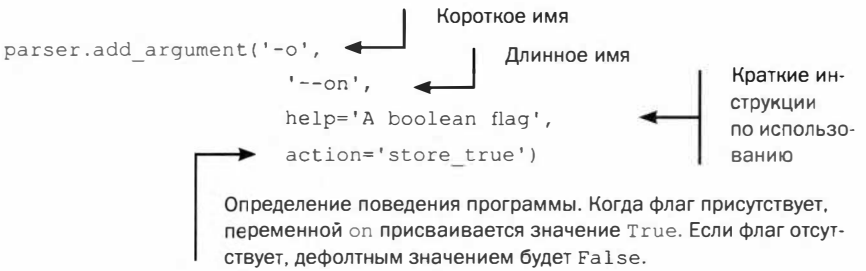
```
$ ./foo.py -f blargh foo
usage: foo.py [-h] [-a str] [-i int] [-f FILE] [-o] str
foo.py: error: argument -f/--file: can't open 'blargh': \
[Errno 2] No such file or directory: 'blargh'
```

A.3.6. Создание флага

Опция флаг не сопровождается значением типа строки или целого числа. Флаги либо присутствуют, либо нет, и *обычно они* указывают на то, что соответствующая опция либо «истинна» (`True`), либо «ложна» (`False`).

Вы уже знакомы с флагами `-h` и `--help`. Их не сопровождают какие-либо значения. Один из этих флагов либо присутствует, и тогда программа выводит инструкции по использованию, либо отсутствует, и программа не выводит инструкции. Во всех упражнениях данной книги я использую флаги для указания на значение `True`, если они присутствуют, и `False` в противном случае, что реализуемо с помощью кода `action='store_true'`.

Например, в файле *new.py* показан пример флага с именем `-o` или `--on`:



Не всегда присутствие такого «флага» следует интерпретировать как `True`. Вместо этого можно использовать код `action='store_false'`, и тогда переменной `on` будет присвоено значение `False` при наличии флага и дефолтное значением `True` при его отстутствии. Вы также можете применить одну или несколько констант при наличии флага.

Ознакомиться с более подробной информацией можно в документации к модулю `argparse`. В данной книге мы пользуемся флагами только для активации некоторого поведения программы.

A.3.7. Результат вызова функции get_args()

Последним в функции `get_args()` указывается оператор `return`, возвращающий результат анализа (парсинга) аргументов с помощью парсера. То есть код, вызвавший функцию `get_args()`, получит результат следующего выражения:

```
return parser.parse_args()
```

Данное выражение может привести к ошибке, если модуль `argparse` обнаружит, что пользователь предоставил недопустимые аргументы, например строку вместо вещественного числа, или, скажем, указал имя файла с ошибкой. Если синтаксический анализ завершится успешно, мы сможем из нашей программы получить доступ ко всем значениям, предоставленным пользователем.

Кроме того, значения аргументов должны иметь указанные нами *типы*. То есть если мы указываем, что аргумент `--int` должен быть целым числом, то, когда мы запрашиваем `args.int`, в результате передается целое число. Если мы определяем файловый аргумент, то получаем *открытый файловый дескриптор*. Возможно, вы не впечатлены, но это реально чрезвычайно полезная штука.

Если вы откроете сгенерированную нами программу *foo.py*, то увидите, что функция `main()` вызывает функцию `get_args()`, поэтому результат вызова функции `get_args()` возвращается в функцию `main()`. Оттуда мы можем получить доступ ко всем значениям, которые только что определили, используя имена позиционных параметров или длинные имена необязательных параметров:

```
def main():
    args = get_args()
    str_arg = args.arg
    int_arg = args.int
    file_arg = args.file
    flag_arg = args.on
    pos_arg = args.positional
```

A.4. Примеры программ с модулем argparse

Многие тесты из данной книги можно пройти, научившись с помощью модуля `argparse` эффективно проверять аргументы своих программ. Я представляю командную строку как таможенный досмотр. Вы должны быть внимательны и очень осторожны в отношении того, что допускать в свою программу, обязаны защищать ее от «неправильных» аргументов*. Программа *hello.py* из главы 1 служит примером использования одного позиционного и одного опционального аргумента. Давайте рассмотрим еще несколько примеров применения модуля `argparse`.

* При этом я всегда вспоминаю о ребятах, которые норовят ввести в качестве входных данных какое-нибудь неприличное слово.

A.4.1. Один позиционный аргумент

Перед вами первая версия программы *hello.py* из главы 1, которая требует один аргумент — имя для приветствия:

```
#!/usr/bin/env python3
"""Один позиционный аргумент"""

import argparse

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Один позиционный аргумент',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('name', metavar='name',
                        help='The name to greet')

    return parser.parse_args()

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    print('Hello, ' + args.name + '!')

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Параметр `name` не начинается с дефиса, соответственно, это позиционный параметр. Строка `metavar` должна отобразиться в инструкциях, чтобы пользователь понимал, что нужно передать в качестве аргумента.

Все, что предоставляется программе в качестве первого позиционного аргумента, будет доступно в слоте `args.name`.

Программа не выведет строку Hello, если ей не предоставить ни одного аргумента. В таком случае она выведет короткие инструкции по правильному способу вызова программы:

```
$ ./one_arg.py
usage: one_arg.py [-h] name
one_arg.py: error: the following arguments are required: name
```

Если указать более одного аргумента, программа снова откажется работать. В примере ниже Emily и Bronte — два разных аргумента, так как слова в командной строке разделены пробелами. Программе не понравится, что ей передан второй аргумент, который не был определен:

```
$ ./one_arg.py Emily Bronte
usage: one_arg.py [-h] name
one_arg.py: error: unrecognized arguments: Bronte
```

Только когда мы передадим программе ровно один аргумент, она запустится:

```
$ ./one_arg.py "Emily Bronte"
Hello, Emily Bronte!
```

Применение модуля `argparse` может показаться излишним для такой простой программы, но я лишь хотел показать, что данный модуль выполняет довольно много проверок аргументов и отсутствия ошибок самостоятельно.

A.4.2. Два разных позиционных аргумента

Представьте, что программе нужно передать два *разных позиционных* аргумента, например *цвет* и *размер* заказанного товара. Цвет должен быть указан в виде строки, а размер — в виде целого числа. В случае с позиционными аргументами нужно соблюдать порядок, в котором вы их объявили, и пользователь должен указывать аргументы строго в соответствии с ним. К примеру, сначала мы определяем в программе цвет, а потом размер:

```
#!/usr/bin/env python3
"""Два позиционных аргумента"""

import argparse

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Два позиционных аргумента',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)
```

Вот первый позиционный аргумент, так как он определен первым. Обратите внимание, что переменной `metavar` присвоено значение `'color'`, а не `'str'`, поскольку оно более точно описывает ожидаемую нами строку — «цвет» товара.

Вот второй позиционный аргумент. Здесь переменной `metavar` присвоено значение `'size'`, которое может быть как числом, например `4`, так и строкой, например `'small'`. Так что тут все неоднозначно.

```
    parser.add_argument('color',
                        metavar='color',
                        type=str,
                        help='The color of the garment')

    parser.add_argument('size',
                        metavar='size',
                        type=int,
                        help='The size of the garment')

    return parser.parse_args()

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    print('color =', args.color)
    print('size =', args.size)

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Доступ к аргументу «цвет» осуществляется через имя параметра `color`.

Доступ к аргументу «размер» осуществляется через имя параметра `size`.

Опять же, пользователь должен указать ровно два позиционных аргумента. Запуск программы без аргументов приведет к выводу кратких инструкций:

```
$ ./two_args.py
usage: two_args.py [-h] color size
two_args.py: error: the following arguments are
      required: color, size
```

Ввод лишь одного аргумента тоже не поможет. Программа сообщит, что не указан «размер»:

```
$ ./two_args.py blue
usage: two_args.py [-h] color size
two_args.py: error: the following arguments are required: size
```

При передаче программе двух строк, например «синий» в качестве цвета и «маленький» в качестве размера, значение размера будет отклонено, поскольку оно должно быть целым числом:

```
$ ./two_args.py blue small
usage: two_args.py [-h] color size
two_args.py: error: argument size: invalid int value: 'small'
```

Если мы передадим программе два аргумента, второй из которых можно интерпретировать как целое число, все будет хорошо:

```
$ ./two_args.py blue 4
color = blue
size = 4
```

Помните, что *все* аргументы, поступающие из командной строки, являются строками. Командная строка не требует указания кавычек вокруг значения `blue` или `4`, чтобы превратить их в строки, как это делает Python. В командной строке все данные являются строкой, и все аргументы передаются интерпретатору Python как строки.

Когда мы сообщаем модулю `argparse`, что второй аргумент должен быть целым числом, модуль `argparse` пытается преобразовать строку `'4'` в целое число `4`. Если же вы укажете вещественное число `4.1`, оно тоже будет отклонено:

```
$ ./two_args.py blue 4.1
usage: two_args.py [-h] str int
two_args.py: error: argument int: invalid int value: '4.1'
```

Позиционные аргументы требуют, чтобы пользователь соблюдал правильный порядок их передачи. Если по ошибке поменять строковый и целочисленный аргументы местами, модуль `argparse` обнаружит недопустимые значения:

```
$ ./two_args.py 4 blue
usage: two_args.py [-h] COLOR SIZE
two_args.py: error: argument SIZE: invalid int
value: 'blue'
```

Теперь представьте себе программу с двумя строковыми или числовыми аргументами, позволяющими указать два *разных* значения, например марку и модель автомобиля или рост и вес человека. Как определить, что аргументы перепутаны?

Честно говоря, я стараюсь писать программы, принимающие лишь один позиционный аргумент или один или несколько *одинаковых элементов*, например список файлов для обработки.

A.4.3. Ограничение значений с помощью опции choices

В предыдущем примере ничто не мешало пользователю указать *два целочисленных значения*:

```
$ ./two_args.py 1 2
color = 1
size = 2
```

1 — это строка. Вам может показаться, что это число, но на самом деле это *символ* `'1'`. Мы имеем дело с допустимым строковым значением, поэтому наша программа принимает его.

Она также приняла бы «размер» `-4`, что явно не является реальным размером:

```
$ ./two_args.py blue -4
color = blue
size = -4
```

Как проверить, предоставляет ли пользователь реальный цвет или размер? Допустим, мы предлагаем рубашки лишь нескольких цветов. Мы можем передать список допустимых значений, используя опцию `choices`.

В следующем примере мы ограничиваем цвет «красным», «желтым» или «синим». Кроме того, мы способны применить код `range(1, 11)`, чтобы ограничить диапазон числами от 1 до 10 (11 не включено!) и использовать их в качестве допустимых размеров для рубашек:

```
#!/usr/bin/env python3
"""Опция choices"""

import argparse
```

```
# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Опция choices',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('color',
                        metavar='str',
                        help='Color',
                        choices=['red', 'yellow', 'blue'])

    parser.add_argument('size',
                        metavar='size',
                        type=int,
                        choices=range(1, 11),
                        help='The size of the garment')

    return parser.parse_args()

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    print('color =', args.color)
    print('size =', args.size)

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Опция `choices` принимает список значений. Модуль `argparse` останавливает программу, если пользователь не вводит одно из них.

Пользователь должен выбрать из диапазона чисел от 1 до 10, иначе модуль `argparse` завершит работу программы с ошибкой.

Если интерпретатор доберется до данной строки, мы будем уверены, что `args.color` определенно будет присвоено одно из перечисленных выше значений цвета, а `args.size` — целочисленное значение в диапазоне от 1 до 10. Интерпретатор никогда не доберется до этой точки, пока оба аргумента не будут содержать допустимые значения.

Любое значение, отсутствующее в списке, отклоняется, и пользователю показываются допустимые варианты. Запуск программы без аргументов тоже не может быть успешным:

```
$ ./choices.py
usage: choices.py [-h] color size
choices.py: error: the following arguments are required: color, size
```

Если мы укажем «фиолетовый» цвет, он будет отклонен, потому что его нет в определенном списке `choices`. Сообщение об ошибке, выдаваемое модулем `argparse`, оповестит пользователя о проблеме («invalid choice») и даже перечислит допустимые цвета:

```
$ ./choices.py purple 1
usage: choices.py [-h] color size
choices.py: error: argument color: \
invalid choice: 'purple' (choose from 'red', 'yellow', 'blue')
```

Аналогично с отрицательным значением аргумента `size`:

```
$ ./choices.py red -1
usage: choices.py [-h] color size
choices.py: error: argument size: \
invalid choice: -1 (choose from 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Только если оба аргумента допустимы, программа продолжает выполнение:

```
$ ./choices.py red 4
color = red
size = 4
```

Да, действительно довольно много кода для обратной связи и проверки отсутствия ошибок, который вам никогда не придется писать. Ведь лучший код тот, который вы не пишете!

A.4.4. Два одинаковых позиционных аргумента

Разрабатывая программу, складывающую два числа, мы могли бы определить их как два позиционных аргумента, например `n1` и `n2`. Но так как это одни и те же аргументы (два числа, которые мы складываем), имеет смысл воспользоваться опцией `nargs` и сообщить модулю `argparse`, что программе нужно получить именно два значения:

```
#!/usr/bin/env python3
"""Опция nargs=2"""

import argparse

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Опция nargs=2',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('numbers',
                        metavar='int',
                        nargs=2,
                        type=int,
                        help='Numbers')

    return parser.parse_args()

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    n1, n2 = args.numbers
    print(f' {n1} + {n2} = {n1 + n2}')

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Согласно строке `nargs=2` программе требуются именно два значения.

Каждое значение должно быть пригодно к преобразованию в целое число, иначе программа выдаст ошибку.

Так как мы определили, что в качестве аргумента `numbers` переданы именно два значения, мы можем скопировать их в две переменные.

Поскольку это фактические целочисленные значения, результатом применения оператора + станет сумма чисел, а не конкатенация строк.

В инструкции указано, что программе нужно передать два числа:

```
$ ./nargs2.py
usage: nargs2.py [-h] int int
nargs2.py: error: the following arguments are required: int
```

Передав два прекрасных целочисленных значения, мы получим их сумму:

```
$ ./nargs2.py 3 5
3 + 5 = 8
```

Обратите внимание, что модуль `argparse` преобразует значения `n1` и `n2` в фактические целые значения. Если вы измените параметр `type=int` на `type=str`, вы увидите, что программа выведет `35` вместо `8`, потому что оператор `+` в Python не только складывает числа, но и конкатенирует строки!

```
>>> 3 + 5
8
>>> '3' + '5'
'35'
```

А.4.5. Множество одинаковых позиционных аргументов

Программу из предыдущего раздела можно расширить до сложения любого количества чисел, сколько бы ни ввел пользователь. Если вам нужен *один или более* аргументов, можно использовать параметр `nargs='+'`:

```
#!/usr/bin/env python3
"""Опция nargs=+"""

import argparse

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""
```

```
    parser = argparse.ArgumentParser(
        description='Опция nargs=+',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('numbers',
                        metavar='int',
                        nargs='+',
                        type=int,
                        help='Numbers')

    return parser.parse_args()
```

Символ + указывает на допустимость одного или более значений.

Значение int указывает, что все аргументы должны быть целыми числами.

```
# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    numbers = args.numbers
    print('{} = {}'.format(' + '.join(map(str, numbers)),
                           sum(numbers)))

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

numbers — это список, содержащий хотя бы один элемент.

Не волнуйтесь, если пока не понимаете эту строку кода. Поймете ближе к концу книги.

Обратите внимание, что `args.numbers` всегда будет списком. Даже если пользователь укажет только один аргумент, `args.numbers` все равно будет списком, содержащим одно значение:

```
$ ./nargs+.py 5
5 = 5
$ ./nargs+.py 1 2 3 4
1 + 2 + 3 + 4 = 10
```

Вы также можете применить параметр `nargs='*'`, чтобы указать *ноль или более* аргументов, и `nargs='?'` — *ноль или один* аргумент.

A.4.6. Файловые аргументы

До сих пор мы использовали аргументы типа `str` (по умолчанию), `int` или `float`. Но в книге также много упражнений, требующих в качестве

входных данных файл. Для этого переменной `type` можно присвоить значение `argparse.FileType('r')`. Так вы укажете, что аргумент должен быть *файлом*, доступным для *чтения* (о чем говорит буква `'r'` — read, «чтение»).

Кроме того, если вы хотите, чтобы файл был *текстовым* (а не *двоичным*), нужно добавить опцию `'t'`. Данные опции станут более понятными после того, как вы прочтете главу 5. Ниже показана реализация команды `cat -n`, где `cat` *конкатенирует* доступный для чтения текстовый файл, а `-n` отвечает за *нумерацию* строк вывода:

```
#!/usr/bin/env python3
"""Версия `cat -n` на Python"""

import argparse

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Версия `cat -n` на Python',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('rt'),
                        help='Input file')

    return parser.parse_args()

# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()

    for i, line in enumerate(args.file, start=1):
        print(f' {i:6}  {line}', end='')

# --------------------------------------------------
if __name__ == '__main__':
    main():
```

Аргумент будет отклонен, если он не указывает на допустимый, доступный для чтения текстовый файл.

Значение `args.file` — это открытый файловый дескриптор, из которого мы можем напрямую выполнять чтение. Опять же, не волнуйтесь, если не понимаете этот код. Мы обсудим файловые дескрипторы в главах книги.

Когда мы определяем аргумент с параметром `type=int`, мы возвращаем фактическое целочисленное значение. Здесь мы определяем тип аргумента `file` как `FileType`, поэтому получаем *открытый файловый дескриптор*. Если бы мы определили аргумент `file` как строку, нам пришлось бы вручную проверять, передал ли пользователь именно файл, а затем использовать функцию `open()`, чтобы открыть файловый дескриптор:

```
#!/usr/bin/env python3
"""Версия `cat -n` на Python - проверка файлового аргумента
  вручную"""

import argparse
import os

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Версия `cat -n` на Python',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', metavar='str', type=str,
                        help='Input file')

    args = parser.parse_args()

    if not os.path.isfile(args.file):
        parser.error(f'"{args.file}" is not a file')

    args.file = open(args.file)

    return args

# --------------------------------------------------
def main():
"""Да грянет джаз!"""

    args = get_args()

    for i, line in enumerate(args.file, start=1):
```

Перехватываем аргументы.

Проверяем, передано ли в качестве аргумента `file` что-то отличное от файла.

Выводим сообщение об ошибке и выходим из программы с ненулевым значением.

Заменяем `file` открытым файловым дескриптором.

```
        print(f' {i:6}  {line}', end='')

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

С определением `FileType` вам не нужно писать ничего подобного.

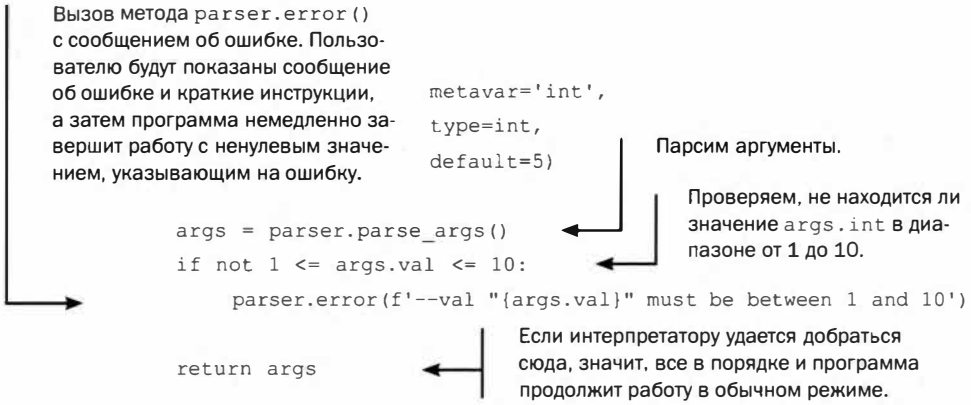
Вы вольны также использовать строку `argparse.FileType('w')`, чтобы указать на необходимость предоставления имени файла, доступного для *записи* (`'w'` — write, «запись»). Можно передать дополнительные аргументы, указывающие, как открыть файл, например кодировку. Для получения дополнительной информации смотрите документацию.

A.4.7. Проверка аргументов вручную

Мы способны вручную проверить аргументы, прежде чем получать результат от функции `get_args()`. Например, мы можем определить, что опция `--int` должна быть целым числом, но как ограничить диапазон допустимых значений числами от 1 до 10?

Довольно простой способ сделать это — вручную проверить значение. При возникновении проблемы можно вызвать функцию `parser.error()`, чтобы остановить выполнение программы, вывести сообщение об ошибке вместе с краткими инструкциями, а затем завершить ее работу с кодом ошибки:

```
#!/usr/bin/env python3
"""Проверка аргумента вручную"""

import argparse

# --------------------------------------------------
def get_args():
    """Получаем аргументы командной строки"""

    parser = argparse.ArgumentParser(
        description='Проверка аргумента вручную',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-v',
                        '--val',
                        help='Integer value between 1 and 10',
```




```
# --------------------------------------------------
def main():
    """Да грянет джаз!"""

    args = get_args()
    print(f'val = "{args.val}"')

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

Если мы предоставим корректное значение аргумента —`val`, все будет хорошо:

```
$ ./manual.py -v 7
val = "7"
```

При запуске программы со значением вне диапазона, например `20`, получаем сообщение об ошибке:

```
$ ./manual.py -v 20
usage: manual.py [-h] [-v int]
manual.py: error: --val "20" must be between 1 and 10
```

Нельзя не упомянуть, что метод `parser.error()` также завершает работу программы с ненулевым кодом выхода. В консольном мире статус выхода `0` означает «ноль ошибок», поэтому любые значения, отличные от `0`, расцениваются как ошибка. Возможно, вы еще не понимаете, насколько это круто, но это так, поверьте мне.

A.4.8. Автоматическое документирование программ

Определяя параметры программы с помощью модуля `argparse`, помните, что флаги `-h` и `--help` резервируются под создание и вывод справочной документации — инструкций по использованию программы. Вам не нужно добавлять их, и вы не сможете применять данные флаги для других целей.

Я представляю документацию чем-то вроде двери в программу. Через двери мы попадаем в здания, машины и т. п. Вы когда-нибудь сталкивались с дверью, которую непонятно как открыть? Или с дверью, на которой написано «ТОЛКАЙТЕ», хотя ручка явно предназначена, чтобы «ТЯНУТЬ»? В книге Дона Нормана *«Дизайн привычных вещей»** есть термин *аффордансы*, использующийся для описания свойств или функций объекта, которые подсказывают, что с ним можно сделать.

Справочная информация (инструкции) вашей программы похожа на ручку двери. Она должна точно сообщать пользователю, как пользоваться программой.

Когда я сталкиваюсь с совершенно новой программой, то запускаю ее либо без аргументов, либо с флагом `-h` или `--help`. Я *ожидаю* увидеть некие инструкции по использованию. Единственная альтернатива — открыть исходный код и посмотреть, как заставить программу работать и как ее настроить. Согласитесь, это совершенно неприемлемый способ разработки и распространения программного обеспечения!

Когда вы создаете новую программу с помощью команды `new.py foo.py`, в ней генерируются следующие инструкции:

```
$ ./foo.py -h
usage: foo.py [-h] [-a str] [-i int] [-f FILE] [-o] str

Rock the Casbah

positional arguments:
  str                   A positional argument

optional arguments:
-h, --help              show this help message and exit
-a str, --arg str      A named string argument (default:)
```

* Издано на русском языке. Норман, Д. Дизайн привычных вещей. М. : 2013.

```
-i int, --int int      A named integer argument (default: 0)
-f FILE, --file FILE  A readable file (default: None)
-o, --on               A boolean flag (default: False)
```

Не написав ни строчки кода, вы сразу же получаете:

— исполняемую Python-программу;
— различные аргументы командной строки;
— стандартные справочные инструкции по использованию программы.

Это «ручка» вашей программы, и вам не нужно писать ни строчки кода, чтобы получить ее!

РЕЗЮМЕ

— Позиционные параметры обычно обязательны. Если программа должна принимать два или более различных позиционных параметра, лучше использовать вместо них именованные параметры.
— Опциональные параметры могут иметь имя, например `--file fox.txt`, где `fox.txt` — это значение параметра `--file`. Рекомендуется всегда определять дефолтное значение для параметров.
— Модуль `argparse` может обрабатывать аргументы разных типов, в том числе числа, такие как `int` и `float`, и даже файлы.
— Такие флаги, как `--help`, не имеют ассоциированного значения. Они (обычно) считаются «истинными», если переданы с программой, и «ложными» в противном случае.
— Флаги `-h` и `--help` зарезервированы модулем `argparse`. С ним ваша программа автоматически ответит на эти флаги инструкцией по использованию.

Популярное издание

0+

Кен Юэнс-Кларк

PYTHON ДЛЯ НАЧИНАЮЩИХ

Учимся программировать с помощью мини-игр и загадок

Ответственный редактор *Екатерина Истомина*
Литературный редактор *Александра Голанцева*
Художник *Эрик Брегис*
Верстка *Эрик Брегис*
Корректура *Любовь Макарова*

Формат 70x100/16. Бумага офсетная.
Тираж 2000 экз. Заказ Х-1852.

Издатель и изготовитель: ООО «Феникс».
Юр. и факт. адрес: 344011, Россия, Ростовская обл.,
г. Ростов-на-Дону, ул. Варфоломеева, д. 150
Тел/факс: (863) 261-89-65, 261-89-50

Изготовлено в России. Дата изготовления: 09.2024.
Срок годности не ограничен.

Отпечатано в ООО «Экопейпер»
Юр. адрес: 420073, Россия, Республика Татарстан,
г. Казань, ул. Аделя Кутуя, дом 82, помещение 209
Адрес местонахождения: 420073, Россия, Республика Татарстан,
г. Казань, Проспект Ямашева, 36 Б

Свои пожелания и предложения по качеству и содержанию книг
вы можете сообщить по e-mail: idea@fenixrostov.ru

Python для начинающих

Кен Юэнс-Кларк

В этой книге вас ждут маленькие игры и головоломки, решая которые вы научитесь программировать на Python. Выполнять эти упражнения не только весело и интересно, но еще и полезно, ведь в процессе вы узнаете о работе с текстом, основных алгоритмах языка, списках и словарях.

Хотя проекты и маленькие, но вознаграждение большое: каждая глава книги бросает вам вызов. В процессе вы научитесь писать такие программы на языке Python, как генератор паролей, рифм и шекспировских оскорблений. По мере выполнения упражнений вы перейдете от новичка в Python к уверенному программисту, и это будет весело!

Что вы узнаете:

- Как писать консольные Python-программы
- Как управлять структурами данных Python
- Как использовать фактор случайности и управлять им
- Как создавать и выполнять тесты программ и функций
- Как и где скачать проекты для каждой главы

Кен Юэнс-Кларк — старший научный сотрудник-программист Аризонского университета. Программирует больше 20 лет и имеет степень магистра в области биосистемной инженерии.

> “Python для начинающих» — это легкое, забавное введение в Python, объясняющее ключевые концепции и не дающее заскучать”
>
> — Аманда Деблер, Schaeffler Technologies

> “Знание + юмор + лаконичность. Поистине бесценная книга”
>
> — Мафинар Хан, theScore

> “Обучение на маленьких проектах безумно эффективно, и поэтому эта книга — шедевр”
>
> — Марчин Сенк, e-Xim IT

> Отличный выбор для тех, кто хочет прокачать навыки программирования на Python”
>
> — Хосе Апаблаза, Steadfast

ISBN 978-5-222-37959-2

9 785222 379592

ИЗДАТЕЛЬСТВО ФЕНИКС ХОРОШИЕ КНИГИ

MANNING