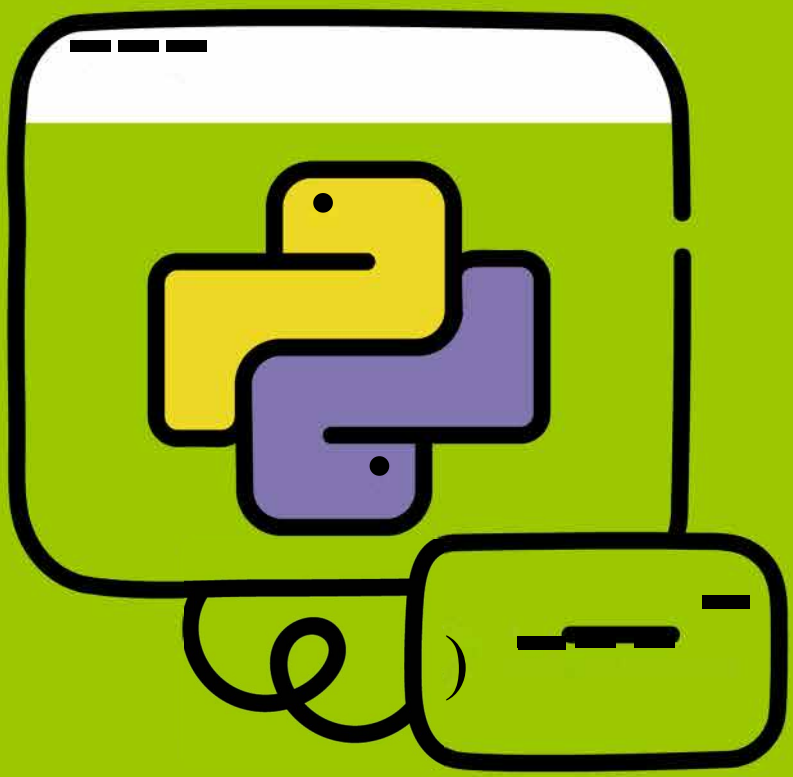
ЛЕРНЕР РЕУВЕН

PYTHON ИНТЕНСИВ

50 быстрых упражнений



REUVEN LERNER

PYTHON WORKOUT: 50 TEN-MINUTE EXERCISES

РЕУВЕН ЛЕРНЕР

PYTHON-ИНТЕНСИВ: 50 БЫСТРЫХ УПРАЖНЕНИЙ

УДК 004.43
ББК 32.973
Л49

Лернер, Реувен.
Л49 Python–интенсив: 50 быстрых упражнений / Р. Лернер; перевод с английского Г. Ярошенко — Москва: Издательство АСТ, 2024. – 368 с. – (Программирование для всех)

ISBN 978–5–17–155721–8

Автор, Реувен М. Лернер преподает Python и data science компаниям по всему миру.

«Python–интенсив: 50 быстрых упражнений» – пособие по программированию для продолжающих, тех, кто владеет теоретической базой языка Python.

Книга отлично подойдет всем, кто хочет применить свои знания на практике. Перед каждым упражнением вы найдете теоретическую выжимку, необходимую для успешного выполнения заданий. Пособие также содержит ссылки на разбор упражнений и полезные материалы.

С помощью этой книги вы освоите такие базовые понятия языка Python, как:

- основные структуры данных,
- функции,
- генераторы,
- объектно–ориентированное программирование
- итераторы.

УДК 004.43
ББК 32.973

ISBN 978–5–17–155721–8

Предисловие

Во многом изучение языка программирования похоже на изучение иностранного (человеческого) языка. Вы можете пройти курс, понять предмет и даже хорошо сдать выпускной экзамен. Но, когда приходит время использовать язык на практике, вы можете оказаться в замешательстве, не зная, какой синтаксис использовать, или каким наиболее подходящим способ выразить свою мысль — не говоря уже о том, что вы не можете понять носителей языка.

Вот тут–то и приходит на помощь практика. Практика иностранного языка повышает уровень беглости и уверенности, позволяя вам вести более глубокие и интересные беседы. Практика Python позволит вам быстрее и проще решать проблемы и одновременно писать более читаемый и сопровождаемый код. Улучшение навыков происходит со временем, по мере того как вы используете язык в новых и разнообразных ситуациях. Зачастую легко не заметить свой прогресс. Но если вспомнить, как вы писали код всего несколько месяцев назад, разница будет очевидной.

Эта книга не предназначена для того, чтобы научить вас Python. Скорее, ее цель — дать вам практику, необходимую для свободного владения Python. После выполнения упражнений в этой книге — а не просто беглого просмотра вопросов и подглядывания в ответы — вы будете писать более читабельный, более правильный и удобный в сопровождении код на Python.

Упражнения на Python появились в результате бесед с моими студентами на корпоративных занятиях по обучению Python. После окончания курса они часто спрашивали, где можно найти

упражнения для дополнительной практики и улучшения своих навыков. Эта книга основана на практических занятиях, которые я провожу для своих студентов, а также на обсуждениях, возникающих во время и после занятий.

Упражнения призваны помочь вам усвоить некоторые из основных понятий языка Python: основные структуры данных, функции, генераторы, объектно-ориентированное программирование и итераторы. Эти темы могут показаться простыми, возможно, даже слишком простыми для книги упражнений. Но весь Python, от самого большого приложения до самого маленького скрипта, основан на этих фундаментальных вещах. Чтобы чувствовать себя уверенным при разработке на Python, очень важно знать эти основы. Я часто говорю, что игнорирование фундаментальных основ в пользу более сложных тем сродни тому, как если бы студент-химик игнорировал элементы в пользу «настоящих» химических веществ.

Я могу лично подтвердить важность практики не только как преподаватель Python, но и как студент. В течение нескольких лет я изучаю китайский язык, во многом из-за того, что каждые несколько месяцев я езжу в Китай для проведения курсов на Python. Каждый урок, который я беру, и каждое упражнение, которое я делаю, кажется, не сильно продвигает мое свободное владение языком. Но когда я возвращаюсь в Китай после нескольких месяцев отсутствия, я понимаю, что практика действительно помогла и что мне легче общаться с местными.

Мне еще далеко до свободного владения китайским языком, но я делаю успехи, и мне приятно оглядываться назад и видеть, как далеко я продвинулся. Я надеюсь и рассчитываю, что Упражнения на Python сделает то же самое для вас, улучшая ваше понимание и свободное владение языком с каждым днем.

Благодарности

Возможно, это клише, что написание книги является совместной работой, но тем не менее это чистая правда. Поэтому я хочу поблагодарить и выразить признательность людям, без которых этой книги не было бы.

Прежде всего я хочу поблагодарить тысячи студентов, которым я имел честь преподавать на моих корпоративных курсах по Python. Именно благодаря их вопросам, предложениям, соображениям и исправлениям решения и объяснения приобрели свой нынешний вид.

Спасибо также многим подписчикам моей еженедельной рассылки «Лучшие разработчики», которые часто находят время, чтобы прокомментировать и исправить темы, о которых я писал. Я многому научился у них и часто использую эти знания в своей преподавательской деятельности.

Далее Филип Гуо — доцент кафедры когнитивных наук Калифорнийского университета в Сан-Диего. Он также является автором и владельцем сайта Python Tutor, бесценного инструмента, который я часто использую в своих курсах и к которому я рекомендую своим студентам обращаться в случае, когда у них возникают сложности с написанием кода. В этой книге я использовал много скриншотов с Python Tutor, и почти каждое решение содержит ссылку на этот сайт, чтобы вы могли самостоятельно разобраться в коде.

Спасибо всем, кто работает над Python, начиная с основных разработчиков, тех, кто пишет и ведет блог о языке, и заканчивая теми, кто создает пакеты. Экосистема Python — это впечат-

ляющее технологическое достижение, но меня также поразило огромное количество по–настоящему отзывчивых, порядочных и дружелюбных людей, стоящих за этими достижениями.

Множество людей из Manning внесли свой вклад в эту книгу и сделали ее намного лучше, чем я бы сделал в одиночку. (И вот доказательство: прототип этой книги, опубликованный самостоятельно, был не так хорош, как тот, что вы сейчас читаете!) Я тесно сотрудничал с несколькими людьми, и все они благодаря мастерству и терпению помогли этой книге воплотиться в жизнь. Майкл Стивенс увидел перспективность книги, ориентированной на отработку упражнений, и предложил мне поработать с Manning. Фрэнсис Лефковиц не только умело отредактировала текст и указала, где его можно улучшить, разбить или проиллюстрировать; она также была со мной во время всего процесса написания книги. Гэри Хаббард и Игнасио Белтран Торрес дали мне бесчисленное количество технических советов и правок, находя ошибки и помогая улучшить неудачные объяснения. А Карл Кеснель произвел на меня неизгладимое впечатление своими подробными правками окончательного текста.

Всем рецензентам: Аннет Девинд, Билл Бейли, Чарльз Дэниелс, Кристоффер Финк, Дэвид Криф, Дэвид Моравек, Дэвид Р. Снайдер, Гэри Хаббард, Джефф Крейг, Глен Сиракавит, Жан–Франсуа Морен, Джефф Смит, Йенс Кристиан Б. Мадсен, Джим Амрхайн, Джо Юстесен, Киран Кут–Динь, Марк Элстон, Майур Патил, Мередит Годар, Стефан Трост, Стив Лав, Сушант Бхосале, Тамара Л. Фульц, Тони Холдройд и Уоррен Майерс — ваши предложения помогли сделать эту книгу лучше.

Наконец, моя семья была терпелива на протяжении всей моей деловой и академической карьеры. Они поддерживали и помогали мне, пока я занимался развитием своих курсов, получал PhD и путешествовал по миру, преподавая Python. Они дважды проявили терпение касательно это книги: когда я самостоятельно опубликовал ее на своем сайте, и когда она была обновлена, дополнена и улучшена (довольно значительно), чтобы стать тем, что вы сейчас читаете. Спасибо моей жене, Шире, и моим детям, Атаре, Шикме и Амоцу, за их понимание и признательность.

Об этой книге

Python Workout не предназначена для обучения вас Python, хотя я надеюсь и ожидаю, что вы узнаете много нового. Она призвана помочь вам улучшить понимание языка Python и его использование для решения проблем. Воспринимайте ее как сборник упражнений, силу и потенциал которого вы можете использовать. Чем больше усилий вы вложите в эту книгу, тем больше вы получите от нее.

Другими словами, это книга, которую нужно не просто прочитать или пролистать. Чтобы обучение прошло успешно, вам придется потратить время на поиск ответов на вопросы, неизбежно совершая ошибки. Есть большая разница между тем, чтобы прочитать о поиске решения, и тем, чтобы написать его самому. Я надеюсь, что вы потратите время на то, чтобы ответить на эти вопросы; я обещаю, что это с лихвой окупится в будущем.

За время работы с *Python Workout* вы решите множество задач, связанных с основными структурами данных, функциями, генераторами, модулями, объектами и итераторами. Вы поймете, как использовать их эффективно, и узнаете, как применять различные идиоматические способы. После выполнения этих упражнений вам будет легче разрабатывать и писать программы на Python для работы и для развлечения.

Обратите внимание, что не стоит искать помощи в документации Python или даже на таких сайтах, как Stack Overflow. Ни один разработчик не может запомнить все, что ему нужно в повседневной работе. Я надеюсь, что по мере изучения книги и дальнейшей работы с Python вы будете обращаться к подобной документации реже или только для ознакомления с более сложными темами.

Для кого эта книга

Эта книга предназначена для разработчиков, которые прошли курс Python или, возможно, прочитали вводную книгу по этому языку. Фактически большая часть этих упражнений предназначена для тех, кто проходит мой вводный курс по Python или недавно закончил его. Вы должны иметь представление об основных конструкциях, таких как `if` и `for`, а также об основных структурах данных, таких как строки, списки, кортежи и словари.

Но есть разница между мимолетным знакомством с этими темами и умением применять их для решения реальных проблем. Если вы умеете работать с Python, но каждый день по многу раз обращаетесь к Stack Overflow, то эта книга поможет вам стать более уверенным и независимым в написании кода на Python. Думаю, что если вы регулярно программируете на Python не менее шести месяцев, то эта книга будет вам полезна.

Из чего состоит эта книга: дорожная карта

Эта книга состоит из десяти глав, каждая из которых посвящена отдельным аспектам Python. Однако в упражнениях каждой главы будут использоваться методы из других глав. Например, почти в каждом упражнении вас попросят написать функцию или класс, хотя функции представлены в главе 6, а классы — в главе 9. Воспринимайте эти названия как общие рекомендации, а не строгие правила для того, что вы будете практиковать и изучать в каждой главе.

Главы

1. **Числовые типы:** Целые числа и числа с плавающей запятой — преобразования значений между числами и строками.
2. **Строки:** Работая со строками, рассматривайте их не только как текст, но и как последовательности, которые можно итерировать.

3. **Списки и кортежи:** Создание, изменение (в случае списков) и извлечение из списков и кортежей.

4. Словари и множества: Различные способы использования словарей и некоторых их полезных методов. Кроме того, в некоторых случаях используются множества, связанные со словарями.

5. **Файлы:** Чтение и запись в файл.

6. **Функции:** Написание функций, включая вложенные функции. Изучение области видимости переменных в Python.

7. **Функциональное программирование с генераторами:** Решение задач при помощи списков, множеств и генераторов словарей.

8. **Модули и пакеты:** Написание и использование модулей в Python–программе.

9. **Объекты:** Создание классов, написание методов, использование атрибутов и объяснение наследования.

10. **Итераторы и генераторы:** Добавление протокола итератора в классы, написание функций–генераторов и представлений генераторов.

Об этой книге

Упражнения составляют основную часть каждой главы. Каждое упражнение состоит из пяти компонентов:

1. **Упражнение:** Постановка проблемы, которую вам предстоит решить.

2. **Обсуждение:** Подробное обсуждение проблемы и способов ее решения.

3. **Решение:** Код решения, а также ссылка на код на сайте Python Tutor [qr1], чтобы вы могли его запустить. Код доступен на GitHub [qr2].

4. **Скринкаст решения:** Короткое видео, представляющее собой запись экрана с объяснением решения. В видео будет показан

1

2

не только ответ, но и процесс поиска решения. При чтении книги на liveBook, платформе Manning, видео появляются сразу после каждого решения. В печатной и электронной книге есть ссылка на навигационную страницу [qr3], нужное упражнение можно найти по номеру и названию.

5. **После выполнения упражнения:** Три дополнительных, связанных между собой задания. На эти задания нет ни ответов, ни обсуждений в книге, но вы можете скачать соответствующий код. (Подробности см. в следующем разделе.) Вы можете обсудить эти дополнительные задания и сравнить решения с другими читателями Python Workout на онлайн-форуме, посвященному книге, платформы Manning — liveBook.

В дополнение к упражнениям приводятся многочисленные сноски со справочной информацией, в которых объясняется тема, часто ставящая разработчиков Python в тупик. Например, в сносках рассказывается про f-строки, определение области видимости переменной и о том, что происходит при создании нового объекта. В книге также содержится множество подсказок, советов и примечаний — все они призваны помочь вам улучшить навыки программирования на Python и предостеречь от повторения ошибок, которые я неоднократно совершал на протяжении многих лет.

О коде

В этой книге содержится большое количество кода на языке Python. В отличие от большинства книг, вы должны скорее написать код, чем просто прочитать его. Возможно, некоторые читатели (возможно, вы!) придумают решения лучше, чем у меня, более правильные или более элегантные. Если это так, то не стесняйтесь написать мне об этом.

Решения всех упражнений, включая задачи из «После выполнения упражнения», доступны в двух местах: на сайте Python

Tutor, который предоставляет среду для выполнения кода, или на GitHub по адресу, где вы можете скачать код. Этот репозиторий не только содержит все решения, но и включает тесты `pytest` для каждого из них. (Не знакомы с `pytest`? Я настоятельно рекомендую вам прочитать о нем здесь [qr4] и использовать его для проверки вашего кода.)

Между кодом в репозитории GitHub и тем, что опубликован в книге, есть небольшие различия. В частности, решения в книге не включают строки документации для функций, классов и модулей, в отличие от репозитория.

Эта книга содержит множество примеров исходного кода как в пронумерованных листингах, так и в строках обычного текста. В обоих случаях исходный код оформляется `моноширинным шрифтом`, в котором все знаки имеют одинаковую ширину, чтобы выделить его в тексте. Иногда код также выделяется **жирным шрифтом**, чтобы подчеркнуть изменения в коде, например, добавление новой функции к существующей строке кода.

Во многих случаях исходный код был переформатирован; мы добавили переносы строк и изменили отступы, чтобы уместить их на свободных страницах книги. В редких случаях этого было недостаточно, и мы включили в списки маркеры перевода строки (⍰). Кроме того, комментарии в исходном коде часто удалялись из листингов, если код описывался в тексте. В большинстве листингов содержатся аннотации к коду, подчеркивающие важные концепции.

Как я уже говорил ранее, приобретая эту книгу, вы также получаете доступ к скринкастам, в которых я показываю решения упражнений. Я надеюсь, что благодаря решениям задач (в печатном виде), объяснениям, ссылкам на Python Tutor, коду из репозитория, тестам pytest и скринкастам, вы сможете в полной мере понять каждое решение и применить его в своем собственном коде.

Требования к программному/ аппаратному обеспечению

Прежде всего, вам нужно установить python. Загрузить и установить его проще всего с сайта [qr5]. Я рекомендую установить последнюю доступную версию. Существуют также альтернативные способы установки Python, включая Windows Store или Homebrew для Mac.

Для работы с этой книгой подойдет любая версия Python 3.6 и выше. В нескольких местах текст описывает возможности, которые являются новыми в Python 3.7 и 3.8, но все решения используют методы, которые работают с 3.6. Все программы работают в разных операционных системах, поэтому независимо от того, какую платформу вы используете, упражнения в этой книге будут работать.

Технически вам не нужно устанавливать редактор или IDE (интегрированную среду разработки) для Python, но они обязательно пригодятся. Двумя наиболее популярными IDE являются PyCharm (от JetBrains) и VSCode (от Microsoft). Более консервативные разработчики Python используют vim или Emacs (мой личный фаворит). Но в конце концов, вы можете (и должны) использовать тот редактор, который вам больше подходит. Языку Python неважно, какую IDE вы используете.

Форум для обсуждений liveBook

Покупая книгу *Python Workout*, вы получаете бесплатный доступ к частному веб-форуму Manning Publications, где вы можете оставить комментарии о книге, задать технические вопросы и получить помощь от автора и других пользователей. Чтобы получить доступ к форуму, перейдите по этому адресу [qr6]. Вы также можете

узнать больше о форумах Manning и правилах поведения на сайте [qr7].

Manning предоставляет площадку, где наши читатели могут обсудить интересующие их вопросы, а также пообщаться с автором. Это не накладывает обязательства на автора участвовать в обсуждении, его вклад в форум остается добровольным (и неоплачиваемым). Мы предлагаем вам попробовать задать автору несколько сложных вопросов, чтобы его интерес не пропал! Форум и архивы предыдущих обсуждений будут доступны на сайте издательства до тех пор, пока книга находится в печати.

Об авторе

Реувен М. Лернер — преподаватель по Python на постоянной основе. Он преподает в компаниях США, Европы, Израиля, Индии и Китая, а также ведет онлайн-курсы для частных лиц по всему миру. Он часто пишет в соцсетях о Python и принимает участие в дискуссиях подкаста Business of Freelancing. Реувен живет в Модиине, расположенном в Израиле, вместе с женой и тремя детьми. Вы можете узнать больше о Реувене на сайте [qr8].

Об иллюстрации на обложке[1]

Рисунок на обложке *Python Workout* называется Homme de la Terre de Feu или «Человек с Огненной Земли». Иллюстрация взята из сборника костюмов разных стран Жака Грассе де Сен–Совера (1757–1810) под названием *Costumes civils actuel de tous les peoples connus*, опубликованного во Франции в 1784 году. Каждая иллюстрация превосходно нарисована и разукрашена вручную. Богатое разнообразие коллекции *Grasset de Saint-Sauveur* напоминает нам о культурных различиях между разными частями света всего 200 лет назад. Изолированные друг от друга, люди говорили на разных диалектах и языках. По одежде человека легко было определить, где он живет, чем занимается и какое положение занимает в обществе.

Мы стали одеваться по–другому, а когда–то такое богатое разнообразие регионов исчезло. Теперь трудно различить жителей разных континентов, не говоря уже о разных городах, регионах или странах. Возможно, мы обменяли культурное разнообразие на более насыщенную жизнь — определенно более разнообразную с быстроразвивающимися технологиями.

Во времена, когда трудно отличить одну компьютерную книгу от другой, Мэннинг проявляет изобретательность и инициативу компьютерного бизнеса в обложках книг, основанных на богатом разнообразии региональной жизни двухвековой давности, оживленной иллюстрациями Грассе де Сен–Совера.

[1] В оригинале произведения была обложка, отличная от издания на русском языке

1. Числовые типы

Неважно, рассчитываете ли вы зарплату, банковский процент или сотовые частоты, трудно представить программу, которая совсем не использует числа. В Python есть три разных числовых типа: `int`, `float` и `complex`. Большинству читателей достаточно будет знать `int` (целые числа) и `float` (числа с дробной частью).

Числа не только лежат в основе программирования, но и дают нам хорошее представление о том, как работает язык программирования. Понимание того, как присвоенные переменные и аргументы функций работают с целыми и плавающими числами, поможет вам рассуждать о более сложных типах, таких как строки, кортежи и словари.

В этой главе содержатся упражнения по работе с числами как на ввод, так и на вывод. Хотя работа с числами может быть довольно простой и понятной, иногда требуется время привыкнуть к преобразованию и взаимодействию с другими типами данных.

Полезные ссылки

Таблица 1.1. Что вам нужно знать

Понятие	Что это?	Пример	Чтобы узнать подробнее
`random`	Модуль для генерации рандомных чисел и выбора случайных элементов.	`number = random.randint (1, 100)`	

Окончание таблицы

Сравнение	Операторы для сравнения значений.	`x < y`	
f — строки	Строки, при помощи которых можно интерполировать выражения.	`f'It is currently {datetime.datetime.now()}'`	 
Циклы `for`	Итерации по элементам итерируемого объекта.	`for i in range(10):print(i*i)`	
`input`	Предлагает пользователю ввести строку, затем возвращает строку.	`input ('Введите ваше имя: ')`	
`enumerate`	Помогает нам пронумеровать итерируемые объекты.	`for index, item in enumerate ('abc'): print (f'{index}: {item}')`	
`reversed`	Возвращает итератор с элементами итерируемого объекта, расположенными в обратном порядке.	`r = reversed ('abcd')`	

Упражнение 1. Игра «Угадай число»

Цель первого упражнения — подготовить вас к работе с остальной частью книги. Оно также поможет познакомиться с рядом тем, которые вам понадобятся на протяжении всей карьеры при работе с Python: циклы, ввод данных пользователем, преобразование типов и сравнение значений.

Проще говоря, все программы должны получать что-то на вход, чтобы сделать что-то интересное, и этот ввод часто исходит от пользователя. Знание того, как запрашивать ввод у пользователя, не только полезно, но и позволит нам лучше понимать, какой тип данных мы получаем, как преобразовывать в формат, который мы можем использовать, и что это будет за формат.

Как вы, наверное, знаете, в Python предусмотрено только два вида циклов: `for` и `while`. Знание того, как их писать и использовать, пригодится вам на протяжении всей вашей карьеры при работе с Python. Почти каждый тип данных умеет работать внутри цикла `for`, благодаря чему такие циклы являются распространенными и весьма полезными. Если вы работаете с записями базы данных, элементами в XML-файле или результатами поиска текста с помощью регулярных выражений, вы будете использовать циклы `for` довольно часто. Для работы с этим упражнением:

1. Напишите функцию (`guessing_game`), которая не принимает аргументы.
2. При вызове функция выбирает случайное целое число от 0 до 100 (включительно).
3. Затем попросите пользователя угадать, какое число было выбрано.
4. Каждый раз, когда пользователь будет вводить значение, программа будет показывать одно из следующих сообщений:
   — Слишком большое
   — Слишком маленькое
   — То, что нужно

5. Если пользователь угадает число, программа завершит свою работу. В противном случае пользователю предложат повторить попытку.
6. Программа завершается только после того, как пользователь угадал.

Мы будем использовать функцию `randint` из модуля `random` для генерации случайного числа. Таким образом, вы можете написать:

```
import random
number = random.randint (10, 30)
```

и `number` будет хранить число от 10 до 30 включительно. Затем мы можем делать с `number` все, что захотим: печатать его, хранить, передавать в функцию или использовать в вычислениях.

Мы также предложим пользователю ввести текст с помощью функции `input`. В этой книге мы будем довольно часто использовать `input`, чтобы попросить пользователя сообщить нам что-нибудь. Функция будет принимать в качестве аргумента строку, которая затем будет выведена на экран. После чего функция возвращает строку, содержащую все, что ввел пользователь, например:

```
name = input ('Введите ваше имя: ')
print (f'Привет, {name}!')
```

ПРИМЕЧАНИЕ Если пользователь нажимает на `Enter`, то значением, возвращаемым input, будет пустая строка, а не `None`. Фактически возвращаемое значение `input` всегда будет строкой, независимо от того, что ввел пользователь.

ПРИМЕЧАНИЕ В Python 2 вы бы попросили пользователя ввести данные с помощью функции `raw_input`. Использование функции input в Python 2 считалось рискованным, поскольку она запрашивала у пользователя ввод, а затем оценивала полученную строку с помощью функции `eval`. (Если вам интересно, ознакомьтесь с дополнительной информацией по ссылке). В Python 3 опасная функция исчезла, и теперь ввод осуществляется при помощи `input`.

Обсуждение

В основе этой программы лежит простое применение операторов сравнения ((==, < и >), чтобы пользователь мог угадать случайное целое число, которое выбрал компьютер. Однако стоит уделить внимание некоторым особенностям данной программы.

Первое и наиболее важное — мы используем модуль `random` для генерации случайного числа. После импортирования `random` мы можем вызвать `random.randint`, которая принимает два параметра и возвращает случайное целое число. В целом модуль `random` является полезным инструментом для тех случаев, когда вам нужно выбрать случайное значение.

Обратите внимание, что максимальное число в `random.randint` является инклюзивным, т.е. входит в диапазон, что является необычным для Python. В большинстве случаев крайние значения диапазонов являются эксклюзивными, т.е. не входят в диапазон.

> **СОВЕТ** Модуль `random` используется не только для генерации случайных чисел. Он также содержит ряд функций для выбора одного или нескольких элементов из последовательности Python.

Теперь, когда компьютер «загадал» число, пользователь должен его угадать. Ниже мы запускаем бесконечный цикл в Python, который проще всего создать с помощью `while True`. Несомненно важно, чтобы существовал способ выйти из цикла; в данном случае это произойдет, когда пользователь правильно угадает значение `answer`. Когда это произойдет, внутренний цикл завершится при помощи команды `break`.

Функция `Input` всегда возвращает строку. Это означает, что, если мы хотим угадать число, мы должны превратить введенную строку в целое число. Это делается так же, как и все преобразования в Python: целевой тип передается как параметр в соответствующую функцию. Таким образом, `int ('5')` вернет целое число 5, а `str (5)` — строку `'5'`. Вы также можете создавать но-

вые экземпляры более сложных типов, вызывая класс как функцию, например, `list ('abc')` и dict `([('a', 1), ('b', 2), ('c', 3)])`.

В Python 3 вы не можете использовать < и > для сравнения различных типов. Если вы пренебрежете преобразованием введенного пользователем значения в целое число, программа завершится с ошибкой, сообщив, что она не может сравнить строку (т.е. введенное пользователем значение) с целым числом.

> **ПРИМЕЧАНИЕ** В Python 2 сравнение объектов разных типов не вызывает ошибку. Результаты могут быть неожиданными, если вы не знаете, что именно хотите получить. Это потому, что Python сначала сравнивает их по типу, а затем внутри этого типа. Другими словами, все целые числа меньше всех списков, а все списки меньше всех строк. Так ли нужно использовать < и > для объектов разных типов? Скорее всего, нет, и я обнаружил, что эта функциональность больше путает людей, чем помогает им. В Python 3 вы не можете выполнить такое сравнение: попытка проверить `1 < [10, 20, 30]` приведет к исключению `TypeError`.

В этом упражнении и в остальных частях этой книги я использую *f-строки* для вставки значений из переменных в наши строки. Я большой поклонник f–строк, советую вам попробовать поработать с ними. (См. сноски про f–строки в этой главе.)

Спасительный моржовый оператор

Люди, которые перешли на Python с других языков, часто удивляются, что в циклах `while True` мы обрабатываем пользовательский ввод и прерываем его. Неужели нет способа получше? Некоторые предлагают использовать следующий код:

```
while s = input ('Введите ваши мысли:'):
print (f'Ваши мысли: {s}')
```

Действительно, в этом больше смысла — мы просим пользователя ввести данные и присваиваем их `s`. Однако значение, присвоенное `s`, затем будет передано в `while`, который определит его как логическое значение. Если мы получаем пустую строку, то логическое значение равно `False`, и мы выходим из цикла.

Есть только одна проблема с этим кодом: он не будет работать. Это потому, что присваивание в Python не является выражением, т.к. оно не возвращает значение. Если оно не возвращает значение, то его нельзя использовать в цикле `while`.

Начиная с версии Python 3.8, ситуация несколько изменилась. В этой версии появился оператор «присваивания выражения», который выглядит как `:=` (двоеточие, за которым следует знак равенства). Но на самом деле никто не называет его «оператором присваивания выражения», с самого начала его прозвали «моржовым оператором». Также с самого начала этот оператор вызывал много споров. Некоторые люди говорили, что он привнес в язык ненужную сложность и потенциальные ошибки.

Вот как предыдущий цикл будет выглядеть в Python 3.8:

```
while s:= input ('Введите ваши мысли:'):
print (f'Ваши мысли: {s}')
```

Благодаря моржовому оператору мы можем наконец-то избавиться от циклов `while True` и возможного хаоса с ними! Но подождите: разве нам не нужно беспокоиться о странных последствиях присваивания в условии цикла `while`? Возможно, и это часть противоречия. Но меня убедил, в немалой степени, тот факт, что обычное присваивание и оператор присваивания не являются взаимозаменяемыми; там, где можно использовать одно, другое нельзя. Я думаю, это уменьшает вероятность неправильного использования.

Если вы хотите узнать больше про моржовый оператор, его противоречиях и о том, почему он на самом деле весьма полезен, я предлагаю вам ознакомиться со следующим докладом с PyCon 2019, в котором Дастин Ингрэм приводит весомые аргументы: [qr20].

Вы также можете прочитать больше об этом операторе в PEP 572, где он был представлен и определен: [qr21].

Решение

```
import random
def guessing_game ():
answer = random.randint (0, 100)

    while True:
    user_guess = int (input ('Каковы ваши предпо-
    ложения?'))

    if user_guess == answer:
    print (f'Правильно! Ответ {user_guess}')
    break

    if user_guess < answer:
         print (f'Ваше число {user_guess} слишком
         маленькое!')

    else:
         print (f'Ваше число {user_guess} слишком
         большое!')

guessing_game ()
```

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr22].

ПРИМЕЧАНИЕ В рамках данного упражнения мы будем считать, что наш пользователь будет вводить только правильные данные, а именно целые числа. Помните, что функция `int` обычно предполагает, что мы передаем ей десятичное число, а это значит, что ее аргумент может содержать только цифры. Если вы предпочитаете педантично подходить к работе, вы можете использовать метод `str.isdigit` для проверки того, что строка содержит только цифры. Или вы можете обработать исключение `ValueError`, которое вы получите, если выполните `int` для того, что не может быть превращено в целое число.

Пройдитесь по своему коду с помощью Python Tutor

В этой книге я использую многие диаграммы из Python Tutor, удивительного онлайн-ресурса для преподавания и изучения Python. (Я часто использую его на своих очных занятиях). Вы можете ввести практически любой код Python на сайте, а затем пройтись по его выполнению, фрагмент за фрагментом. Большинство решений в этой книге содержат ссылку, указывающую на код в Python Tutor, чтобы вы могли запустить его, не вводя на сайте.

В Python Tutor глобальные переменные (включая функции и классы) отображаются в *глобальном фрейме*. Помните, что, если вы определяете переменную вне функции, вы создаете глобальную переменную. Любые переменные, которые вы создаете внутри функции, являются локальными переменными и отображаются в Python Tutor в затененных областях. Простые структуры данных, такие как целые числа и строки, отображаются вместе с переменными, указывающими на них, в то время как списки, кортежи и словари отображаются в графическом формате.

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr25].

После выполнения упражнения

Вы часто будете работать с вводом, и поскольку он представляет собой строку, вам, скорее всего, придется преобразовывать его в другие типы, такие как целые числа (в данном упражнении). Вот несколько дополнительных вариантов, как реализовать эту идею на практике:

1. Измените эту программу так, чтобы она давала пользователю только три шанса угадать правильное число. После трех безуспешных попыток программа сообщит, что попытки закончились, и завершится.
2. Вы должны не только выбрать случайное число, но и выбрать случайно основание системы счисления от 2 до 16, в которой пользователь должен представить свой ввод. Если пользователь введет число «10», вам нужно будет учесть систему счисления: «10» может означать 10 (десятичная система), или 2 (двоичная система), или 16 (шестнадцатеричная система).
3. Попробуйте сделать то же самое, но попросите программу выбрать случайное слово из словаря, а затем попросите пользователя угадать это слово. (Возможно, вы захотите ограничиться словами из двух–пяти букв, чтобы не усложнять задачу.) Вместо того, чтобы говорить пользователю, что он должен угадать меньшее или большее число, попросите его выбрать предыдущее или следующее из словаря.

f-строки

Большинство людей при выполнении упражнения «Угадай число» пытаются напечатать комбинацию строки и числа, например, «Вы угадали 5». Однако они быстро осознают, что в Python нельзя складывать (используя +) строки

и целые числа. Как же тогда можно вывести на экран строку, содержащую оба типа?

Эта проблема давно беспокоит тех, кто перешел на Python с других языков. Раньше проблему решали при помощи оператора%:

```
'Hello,%s'% 'world'
```

В то время как программисты, пишущие на C, радовались аналогу printf, все остальные считали этот метод неудачным. Среди прочего,% не был суперинтуитивно понятным для новичков, требовал использовать круглые скобки при передаче более одного аргумента и не позволял легко ссылаться на повторяющиеся значения.

Поэтому огромным шагом вперед стало появление в Python метода str.format:

```
'Hello, {0}'.format ('world')
```

Мне нравился str.format, однако многие новички находили его немного сложным в использовании и очень длинным. В частности, им не понравилась идея ссылаться на переменные слева и давать значения справа. А синтаксис внутри фигурных скобок был специфичным для Python, что огорчало всех.

В Python 3.6 появились f–строки, похожие на строки с двойными кавычками, которыми программисты пользовались десятилетиями в Perl, PHP, Ruby и в оболочках Unix. f–строки работают практически так же, как str.format, но без необходимости передачи параметров:

```
name = 'world'
f'Hello, {name}'
```

На самом деле это даже лучше. Вы можете поместить любое выражение внутри фигурных скобок, и оно будет обработано после анализа строки, например:

```
  name = 'world'
  x = 100
  y = 'abcd'
  f'x * 2 = {x*2}, and y.capitalize () is
{y.capitalize ()}'
```

В Python 3.6 появились *f-строки*, похожие на строки с двойными кавычками, которыми программисты пользовались десятилетиями в Perl, PHP, Ruby и в оболочках Unix. f–строки работают практически так же, как `str.format`, но без необходимости передачи параметров:

```
  name = 'world'
  f'Hello, {name}'
```

На самом деле это даже лучше. Вы можете поместить любое выражение внутри фигурных скобок, и оно будет обработано после анализа строки, например:

```
  name = 'world'
  x = 100
  y = 'abcd'
  f'x * 2 = {x*2}, and y.capitalize () is
{y.capitalize ()}'
```

Вы также можете управлять форматированием каждого типа данных, поместив код после двоеточия (`:`) внутри фигурных скобок. Например, можно выровнять строку влево или вправо на поле из 10 хэш–знаков (`#`), следующим образом:

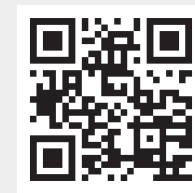
```
  name = 'world'
  first = 'Reuven'
  last = 'Lerner'

  f'Hello, {first:#<10}
  {last:#>10}'
```

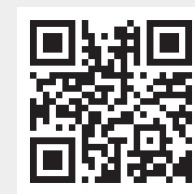
Код формата #<10 означает, что строка, выровненная по левому краю, будет располагаться в поле из 10 символов, при этом # заполнит пустое пространство.
Код формата #>10 означает то же самое, но с правым выравниванием.

Я определенно рекомендую вам обратить внимание на f–строки и использовать их. Это одно из моих любимых усовершенствований Python за последние несколько лет.

Для получения дополнительной информации об f–строках ознакомьтесь со следующими ресурсами:


26

1. Сравнение возможностей форматирования в Python, включая информацию об f–строках: [qr26]


27

2. Длинная статья о f–строках и о том, как их можно использовать: [qr27]
3. PEP, в котором были введены f–строки: [qr28]


28

Что делать, если вы все еще используете Python 2 и не можете использовать f–строки? Вы можете прибегнуть к строковому методу `str.format`, который работает примерно так же, но с меньшей гибкостью. Кроме того, вы должны вызвать метод и сослаться на аргументы по номеру или имени.

Упражнение 2. Сложение чисел

Один из моих любимых типов упражнений включает в себя повторную реализацию функциональности, которую мы уже видели в Python и в Unix. Это подготовка к следующему упражнению, в котором вам предстоит реализовать функцию `sum`. Эта функция принимает последовательность чисел и возвращает сумму этих чисел. Таким образом, если вы вызовете `sum([1,2,3])`, то получите в результате `6`.

Задача состоит в том, чтобы написать функцию `mysum`, которая делает то же самое, что и встроенная функция `sum`. Однако вместо того чтобы принимать в качестве параметра одну после–

довательность, он должен принимать переменное количество аргументов. В результате вместо `sum ([1,2,3])` вы вызовете `mysum (1,2,3)` или `mysum (10,20,30,40,50)`.

> **ПРИМЕЧАНИЕ** Встроенная функция `sum` принимает также необязательный второй аргумент, который мы здесь игнорируем.

И нет, вы не должны использовать для этого встроенную функцию `sum`! (Вы удивитесь, как часто мне задают этот вопрос, когда я читаю лекции.)

Это упражнение призвано помочь вам задуматься не только о числах, но и об устройстве функций. В частности, вам следует подумать о том, какие типы параметров могут принимать функции в Python. Во многих языках вы можете определять функции несколько раз, каждая из которых имеет свою сигнатуру типа (т.е. количество параметров и типы параметров). В Python сохраняется только одно определение функции (т.е. последний раз, когда функция была определена). Гибкость достигается за счет правильного использования различных типов параметров.

> **СОВЕТ** Если вы не знакомы с оператором *splat* (звездочка), вам вероятно захочется узнать больше о нем в этом учебнике по Python: [qr30].

Обсуждение

Функция `mysum` — простой пример того, как можно использовать оператор Python *splat* (он же `*`), чтобы позволить функции принимать любое количество аргументов. Добавляя к `numbers` символ `*`, мы говорим Python, что этот параметр должен получать все аргументы и что `numbers` всегда будут кортежем.

Даже если в нашу функцию не будет передано никаких аргументов, `numbers` все равно будут кортежем. Это будет пустой кортеж, но тем не менее кортеж.

Оператор splat особенно полезен, когда вы хотите получить неизвестное количество аргументов. Обычно предполагается, что все аргументы будут одного типа, хотя в Python это правило не соблюдается. По моему опыту, вы берете кортеж (в данном случае `numbers`) и перебираете каждый элемент с помощью цикла `for` или генератора списка.

ПРИМЕЧАНИЕ Если вы получаете элементы из `*args` с числовыми индексами, то, вероятно, вы делаете что-то не так. Используйте отдельные именованные параметры, если вы хотите отбирать их по одному.

Поскольку мы ожидаем, что все аргументы будут числовыми, мы устанавливаем локальной переменной `output` значение 0, помещаем ее в начало функции, а затем добавляем к ней все отдельные числа с помощью цикла `for`. Мы можем вызвать эту функцию для любого списка, множества или кортежа чисел.

Хотя `sum` (или ее аналоги) используется не очень часто, `*args` является чрезвычайно распространенным способом (для функции) принять неизвестное количество аргументов.

Преобразование итерируемых объектов в аргументы

Что если у нас есть список чисел, например `[1,2,3]`, и мы хотим использовать mysum? Мы не можем просто вызвать `mysum([1,2,3])`, в результате аргумент numbers будет кортежем, первым и единственным элементом которого является список `[1,2,3]`, что выглядит следующим образом:

```
([1,2,3],).
```

Python будет итерировать наш одноэлементный кортеж, пытаясь добавить `0` к `[1,2,3]`. Это приведет к появлению исключения `TypeError`, причем Python сообщит, что не может добавить целое число в список.

Решением в таком случае является предварительное обозначение аргумента символом * при вызове функции. Если мы вызовем `mysum (* [1,2,3])`, наш список превратится в три отдельных аргумента, что позволит вызвать функцию обычным способом.

Это обычно верно при вызове функций. Если у вас есть итерируемый объект и вы хотите передать его элементы в функцию, просто добавьте * в вызов функции.

Решение

```
def mysum (*numbers):
    output = 0
    for number in numbers:
         output += number
    return output

print (mysum (10, 20, 30, 40))
```


31

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr31].

Скринкаст решения


32

Посмотрите короткое видео с объяснением решения: [qr32].

После выполнения упражнения

Очень часто приходится перебирать элементы списка или кортежа, выполняя операцию над каждым элементом, а затем, например, суммировать их. Вот несколько примеров:

1. Встроенная функция `sum` принимает опциональный второй аргумент, который используется в качестве начальной точки для суммирования. (Вот почему она принимает список чисел в качестве первого аргумента, в отличие от нашей реализации `mysum`.) Так `sum ([1,2,3], 4)` возвращает

10, потому что 1+2+3 — это 6, которые будут добавлены к начальному значению 4. Доработайте свою функцию `mysum` так, чтобы она работала подобным образом. Если второй аргумент не указан, то по умолчанию он равен 0. Обратите внимание, что, хотя в Python 3 можно написать функцию, которая определяет параметр после `*args`, я бы предложил избегать этого и просто принимать два аргумента: список и опциональную начальную точку.

2. Напишите функцию, которая принимает список чисел. Он должен вернуть среднее значение (т.е. среднее арифметическое) этих чисел.
3. Напишите функцию, которая принимает список слов (строк). Она должна возвращать кортеж, содержащий три целых числа: длина самого короткого слова, самого длинного слова и средняя длина слова.
4. Напишите функцию, которая принимает список объектов Python. Суммируйте те объекты, которые либо являются целыми числами, либо могут быть преобразованы в целые числа, остальные игнорируйте.

Упражнение 3. Время выполнения

Системные администраторы часто используют Python для выполнения различных задач, включая создание отчетов на основе пользовательских данных и файлов. Нет ничего необычного в том, чтобы сообщить, как часто возникает то или иное сообщение об ошибке, или какие IP-адреса обращались к серверу в последнее время, или какие пользователи чаще всего вводят неправильный пароль. Таким образом, важно уметь накапливать информацию и создавать некоторые базовые отчеты (включая среднее время). Кроме того, важно знать, как работать со значениями с плавающей запятой и чем они отличаются от целых чисел.

Для этого упражнения мы предположим, что вы каждый день пробегаете 10 км согласно плану тренировок. Вы хотите знать, сколько времени в среднем занимает бег.

Напишите функцию (`run_timing`), которая спрашивает, сколько времени вам понадобилось, чтобы пробежать 10 км. Функция продолжит спрашивать, сколько времени (в минутах) потребовалось для дополнительных пробежек, пока пользователь не нажмет Enter. После этого функция вычислит и отобразит среднее время, потраченное на бег 10 км, а затем завершит работу.

Например, вот, как будет выглядеть вывод, если пользователь введет три примера данных:

```
Введите время пробега 10 км: 15
Введите время пробега 10 км: 20
Введите время пробега 10 км: 10
Введите время пробега 10 км:<enter>

Средний показатель 15.0, более 3 пробежек
```

Обратите внимание, что числовые входные и выходные значения должны быть значениями с плавающей точкой. Это упражнение призвано помочь вам потренироваться в преобразовании входных данных в соответствующие типы, а также в отслеживании информации во времени. Вы, вероятно, будете отслеживать данные более сложные, чем время и дистанция, но идея накопления данных с течением времени часто встречается в программах, и важно понять, как это сделать в Python.

Обсуждение

В предыдущем упражнении мы увидели, что `input` — это функция, которая возвращает строку, содержащую введеные пользователем данные. Однако пользователь может ввести как и число, так и пустую строку.

Пустые строки, как и число 0, оператором `if` будут рассматриваться как `False`. Обычно в программах на Python используют следующий способ:

```
if not one_run:
    break
```

Это необычно, и было бы немного странно сказать

```
if len (one_run) == 0:
    break
```

Такой код будет работать, однако, согласно общепринятым договоренностям, такой стиль считается плохим. Следование этим договоренностям сделает ваш код более питоничным, а значит, более читабельным для других разработчиков. В данном случае рекомендуется указать `not` перед переменной, которая может быть пустой, таким образом, значение будет `False`.

В реальном приложении Python, если вы принимаете ввод от пользователя и вызываете `float`, вам, вероятно, следует обернуть его в `try`, на случай, если пользователь введет недопустимое значение:

```
try:
    n = float (input ('Введите число: '))
    print (f'n = {n}')
except ValueError as e:
    print ('Эй! Это недопустимое число!')
```

Также помните, что значения чисел с плавающей точкой не являются абсолютно точными. Они подойдут для задач, связанных с измерением времени, затрачиваемого на бег, но не для чувствительных измерений, таких как научные или финансовые расчеты.


33

Если вы ранее не знали об этом, то я предлагаю открыть ваш локальный интерактивный интерпретатор Python и спросить у него значение `0.1 + 0.2`. Вы можете быть удивлены результатами. (Вы также можете зайти на сайт [qr33] и посмотреть, как это работает в других языках программирования.)

Одним из распространенных решений этой проблемы является использование целых чисел. Вместо того чтобы вести учет долларов и центов (как `float`), вы можете просто вести учет центов (как `int`).

Контролирование при помощи f-строк

Если вы хотите напечатать в Python число с плавающей точкой, то лучше использовать f–строку. Почему? Потому что таким образом можно указать количество цифр, которые будут выведены на печать. Приведем пример:

```
>>> s = 0.1 + 0.7
>>> print (s)
0,7999999999999999
```

Это, вероятно, не то, чего вы хотите. Однако, поместив `s` внутрь f–строки, вы можете ограничить вывод:

```
>>> s = 0.1 + 0.7
>>> print (f'{s:.2f}')
0.80
```

Здесь я сообщил f–строке, что хочу взять значение `s` и отобразить его как число с плавающей точкой (`f`) с максимум двумя цифрами в дробной части числа. См. справочную таблицу (таблица 1.1) в начале этой главы для получения полной документации по f–строкам и кодам форматирования, которые вы можете использовать для различных типов данных.

Решение

```
def run_timing ():
"""Просим пользователя несколько раз ввести числовые данные. Печатает среднее время и количество запусков."""

    number_of_runs
    total_time = 0

    while True:
    one_run = input ('Enter 10 km run time: ')
```

Смотрите, это бесконечный цикл! Может показаться странным наличие while True, действительно, это очень плохая идея не использовать в цикле оператор break для завершения работы при достижении соответствующего условия. Но я думаю, подойдет как общий способ получить неизвестное количество входных данных от пользователей.

```
    if not one_run:
        break
```

Если one_run — пустая строка, остановитесь.

```
number_of_runs += 1
total_time += float (one_run)

    average_time = total_time / number_of_runs

    print (f'Среднее значение {average_time}, для
    {number_of_runs} пробежек')

run_timing ()
```

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr34].

34

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr35].

35

После выполнения упражнения

Числа с плавающей точкой одновременно необходимы и одновременно потенциально опасны в мире программирования: необходимы, потому что многие вещи могут быть представлены только дробными числами, но потенциально опасны, потому что они не точны. Таким образом, вы должны задумываться о том, где и когда использовать их. Вот два упражнения, в которых вы захотите использовать `float`:

- **R** Напишите функцию, которая принимает число `float` и два целых числа (`before` и `after`). Функция должна возвращать `float`, состоящее из `before` (цифры до десятичной точки) и `after` (цифры после). Таким образом, если мы вызываем функцию с `1234.5678`, `2` и `3`, возвращаемое значение будет равно `34.567`.
- **R** Изучите класс `Decimal`, который содержит альтернативное представление с плавающей точкой, настолько точное,

насколько может быть точным любое десятичное число. Напишите функцию, которая принимает от пользователя две строки, превращает их в экземпляры `decimal`, а затем печатает сумму двух введенных пользователем чисел с плавающей точкой. Другими словами, сделайте так, чтобы пользователь ввел `0.1` и `0.2`, а мы получили `0.3`.

Упражнение 4. Шестнадцатеричный вывод

В Python циклы встречаются повсюду, а тот факт, что большинство встроенных структур данных являются итерируемыми объектами, позволяет легко работать с ними по одному элементу за раз. Однако обычно мы итерируем от первого элемента к последнему. Более того, Python не предоставляет нам автоматически индексы элементов. В этом упражнении вы увидите, как немного творчества, а также встроенные функции `reversed` и `enumerate` помогут вам обойти эти проблемы.

Шестнадцатеричные числа довольно распространены в мире компьютеров. На самом деле это не совсем так. Некоторые программисты используют их постоянно. Однако другие программисты, обычно использующие языки высокого уровня и занимающиеся, например, веб–разработкой, даже не помнят, как ими пользоваться.

Дело в том, что я почти не использую шестнадцатеричные числа в своей повседневной работе. И даже если бы они мне понадобились, я мог бы воспользоваться встроенной в Python функцией `hex` и префиксом `0x`. В первом варианте на вход принимается целое число и возвращается шестнадцатеричная строка; во втором — вводится число в шестнадцатеричной системе счисления, что может быть более удобным. Так, `0x50` равно 80, и `hex (80)` вернет строку `0x50`.

Для этого упражнения вам нужно написать функцию (`hex_output`), которая принимает шестнадцатеричное число и возвращает его десятичный эквивалент. То есть, если пользователь введет

`50`, вы будете считать, что это шестнадцатеричное число (равное 0x50), и на экран выведется значение 80. И нет, вы не должны преобразовывать число сразу, используя функцию `int`, хотя допустимо использовать int по одной цифре за раз.

Это упражнение не предназначено для проверки ваших математических навыков, вы можете получить шестнадцатеричный эквивалент целых чисел с помощью функции `hex`, при этом большинству людей это даже не нужно в повседневной жизни. Тем не менее, это касается преобразования (различными способами) типов, которое мы можем выполнять в Python благодаря тому, что последовательности (например, строки) являются итерируемыми объектами. Рассмотрим также встроенные функции, использовать которые проще, чем писать все с нуля.

СОВЕТ В Python оператором возведения в степень является `**`. Поэтому результатом `2**3` будет целое число `8`.

Обсуждение

Ключевым аспектом строк Python является то, что они представляют собой последовательности символов, над которыми мы можем выполнять итерации в цикле `for`. Однако циклы `for` в Python, в отличие от их аналогов в C, не дают нам (и даже не используют) индексы символов. Скорее, они выполняют итерацию над самими символами.

Если нам нужен числовой индекс каждого символа, мы можем использовать встроенную функцию `enumerate`. Эта функция возвращает кортеж из двух элементов на каждой итерации, используя синтаксис Python для множественного присваивания («распаковки»), мы можем захватить каждое из этих значений и поместить их в наши переменные `power` и `digit`.

Вот пример того, как мы можем использовать `enumerate` для вывода первых четырех букв алфавита вместе с индексами букв в строке:

```
for index, one_letter in enumerate ('abcd'):
    print (f'{index}: {one_letter}')
```

ПРИМЕЧАНИЕ Почему в Python вообще есть `enumerate`? Потому что во многих других языках, таких как C, цикл `for` выполняет итерации по последовательностям чисел, которые используются для извлечения элементов из последовательности. Но в Python наши циклы `for` получают элементы напрямую, без необходимости в явной индексной переменной. `enumerate` таким образом создает индексы на основе элементов — в точности противоположно тому, как это работает в других языках.

Вы также видите здесь использование `reversed`, т.е. мы начинаем с последней цифры и идем к первой. `reversed` — это встроенная функция, которая возвращает новую строку, значение которой является обратным значению старой. Мы могли бы получить тот же результат, используя синтаксис среза (slice), hexnum [::–1], но я пришел к выводу, что многие люди не понимают этого синтаксиса. Кроме того, срез возвращает новую строку, тогда как `reversed` возвращает итератор, который потребляет меньше памяти.

Нам нужно преобразовать каждую цифру нашего десятичного числа, которое было введено как строка, в целое число. Мы делаем это с помощью встроенной функции `int`, которую можно представить как создание нового экземпляра класса или типа `int`. Мы также видим, что `int` принимает два аргумента. Первый является обязательным и представляет собой строку, которую мы хотим превратить в целое число. Второй является необязательным и содержит основание числа. Поскольку мы конвертируем из шестнадцатеричной системы счисления (т.е. по основанию 16), мы передаем `16` в качестве второго аргумента.

Решение

```
def hex_output ():
    decnum = 0
    hexnum = input ('Введите
    шестнадцатеричное число
    для преобразования: ')

    for power, digit in
    enumerate (reversed
    (hexnum)):
         decnum += int (digit, 16)
         * (16 ** power)

print (decnum)

hex_output ()
```

reversed возвращает новый итерируемый объект, который возвращает элементы другого итерируемого объекта в обратномпорядке. Вызвав enumerate на выходе из reversed, мы получим каждый элемент hexnum, по одному за раз, вместе с его индексом, начиная с 0.

Оператор ** в Python используется для возведения в степень.

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr36].

36

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr37].

37

После выполнения упражнения

Каждый разработчик Python должен хорошо понимать протокол итератора, который используют циклы `for` и многие другие функции. Комбинирование циклов `for` с другими объектами, такими как `enumerate` и срезы, может помочь сделать ваш код короче и удобнее в обслуживании.

1. Реализуйте решение этого упражнения так, чтобы оно вообще не использовало функцию `int`, а использовало встроенные функции `ord` и `chr` для идентификации символа. Эта реализация должна быть более надежной,

т.к. игнорирует символы, которые не являются допустимыми для введенного основания чисел.

2. Напишите программу, которая запрашивает у пользователя его имя, а затем выдает «треугольник имен»: первая буква имени, затем первые две буквы, затем первые три и так далее, пока все имя не будет записано на последней строке.

Подводя итоги

Трудно представить себе программу на Python, в которой не используются числа. Будь то числовые индексы (в строке, списке или кортеже), подсчет количества появлений IP-адреса в лог-файле или расчет процентных ставок по банковским кредитам, вы будете постоянно использовать числа.

Помните, что Python сильно типизирован, а это значит, что, например, целые числа и строки — это разные типы. Вы можете превратить строки в целые числа с помощью `int`, а целые числа в строки с помощью `str`. И вы можете превратить любой из этих типов в число с плавающей точкой с помощью `float`.

В этой главе мы рассмотрели несколько способов работы с числами различных типов. Вы вряд ли будете писать программы, использующие числа только таким образом, но чувствовать уверенность в том, что они работают и вписываются в большую экосистему Python, очень важно.

2. Строки

Строки в Python — это способ работы с текстом. Слова, предложения, абзацы и даже целые файлы читаются и обрабатываются с помощью строк. Поскольку большая часть нашей работы связана с текстом, неудивительно, что строки являются одним из самых распространенных типов данных.

Вы должны помнить о двух важных вещах касательно строк в Python: (1) они неизменяемы, и (2) в Python 3 они содержат символы Unicode, закодированные в UTF–8. (См. сноски по каждому из этих вопросов.)

В Python нет такого понятия, как «символьный» тип. Мы можем говорить об «односимвольной строке», но это означает лишь строку, длина которой равна 1.

Строки в Python интересны и полезны не только потому, что они позволяют работать с текстом, но и потому, что они являются последовательностью Python. Это означает, что мы можем выполнять итерации по ним (символ за символом), получать их элементы с помощью числовых индексов и выполнять поиск с помощью оператора `in`.

В этой главе содержатся упражнения, которые помогут вам работать со строками различными способами. Чем лучше вы знакомы с техникой работы со строками в Python, тем легче вам будет работать с текстом.

Полезные ссылки

Таблица 2.1. Что вам нужно знать

Понятие	Что это?	Пример	Чтобы узнать подробнее
`in`	Оператор для поиска в последовательности.	`'a' in 'abcd'`	
`Срез`	Извлекает подмножество элементов из последовательности.	`# возвращает 'bdf' 'abcdefg' [1:7:2]`	
`str.split`	Разбивает строки на части, возвращая список.	`# возвращает ['abc', 'def', 'ghi'] 'abc def ghi'.split()`	
`str.join`	Объединяет строки для создания новой строки.	`# возвращает 'abc*def*ghi' '*'.join(['abc', 'def', 'ghi'])`	
`list.append`	Добавляет элемент в список.	`mylist.append('hello')`	
`sorted`	Возвращает отсортированный список, основанный на входной последовательности.	`# возвращает [10, 20, 30] sorted ([10, 30, 20])`	
Итерация над файлами	Открывает файл и итерирует построково.	`for one_line in open(filename):`	

Упражнение 5. Поросячья латынь

Поросячья латынь — распространенный детский «секретный» язык в англоязычных странах. (Он обычно является секретным для детей, которые забывают, что их родители когда-то сами были детьми.) Правила перевода слов с английского на поросячью латынь довольно просты:

1. Если слово начинается с гласной (*a*, *e*, *i*, *o* или *u*), добавьте way к концу слова. Таким образом, air становится *airway*, а *eat — eatway*.
2. Если слово начинается с любой другой буквы, то мы берем первую букву, ставим ее в конец слова, а затем добавляем ay. Следовательно, *python* становится *ythonpay*, а *computer — omputercay*.

(И да, я понимаю, что правила можно сделать более сложными. Для целей данного упражнения давайте оставим их простыми.)

Для этого упражнения напишите функцию Python (`pig_latin`), которая принимает на вход строку, предположительно являющуюся английским словом. Функция должна возвращать перевод этого слова на поросячью латынь. Вы можете работать со словами, не содержащими заглавных букв и знаков препинания.

Это упражнение не предназначено для того, чтобы помочь вам перевести документы на поросячью латынь для вашей работы. (Если это ваша работа, то я должен поставить под сомнение ваш выбор профессии.) Однако оно демонстрирует некоторые мощные приемы, которые вы должны знать при работе с последовательностями, включая поиск, итерацию и срезы. Трудно представить себе программу на Python, которая не содержит любую из вышеупомянутых техник.

Обсуждение

Это упражнение уже давно является одним из моих любимых упражнений для студентов на моих вводных занятиях по программированию. Оно было вдохновлено Брайаном Харви, чья превосходная серия *Computer Science Logo Style* давно стала одной из моих любимых для начинающих программистов.

Прежде всего рассмотрим проверку того, чтобы `word [0]`, первая буква в слове, была гласной. Я часто встречал людей, которые начинают использовать цикл следующим образом:

```
starts_with_vowel = False
for vowel in 'aeiou':
    if word [0] == vowel:
        starts_with_vowel = True
        break
```

Даже если этот код будет работать, он уже начинает выглядеть немного громоздким и запутанным.

Еще одно решение, которое я часто встречаю:

```
if (word [0] == 'a' or word [0] == 'e' or
        word [0] == 'i' or word [0] == 'o' or
        word [0] == 'u'):
    break
```

Я люблю говорить своим студентам: «К сожалению, этот код работает». Почему мне так не нравится этот код? Он не только длиннее, чем нужно, но и сильно повторяется. Правило «не повторяйся» (DRY) всегда должно быть на задворках вашего сознания при написании кода.

Более того, программы на Python, как правило, короткие. Если вы обнаружите, что повторяетесь и пишете необычно длинное выражение или условие, то, скорее всего, вы пропустили более питоновский способ выполнения задач.

Мы можем воспользоваться тем, что Python воспринимает строку как последовательность, и использовать встроенный оператор `in` для поиска `word [0]` в строке, содержащей гласные буквы:

```
if word [0] in 'aeiou':
```

Преимущество этой единственной строки в том, что она читабельна, коротка, точна и достаточно эффективна. Правда,

время, необходимое для поиска по строке или любой другой последовательности в Python, растет вместе с длиной последовательности. Но такое линейное время, иногда выражаемое как `O(n)`, часто достаточно хорошо, особенно когда строки, в которых мы будем искать, довольно короткие.

> **СОВЕТ** Оператор `in` работает со всеми последовательностями (строками, списками и кортежами) и многими другими коллекциями Python. Он эффективно запускает цикл `for` для элементов. Соответственно, оператор `in` для словаря будет выполняться, но при этом будет выполняться поиск только по ключам, игнорируя значения.

После того как мы определили, начинается ли слово с гласной, мы можем применить соответствующее правило поросячьей латыни.

Срезы

Все последовательности Python — строки, списки и кортежи — поддерживают срезы. Идея заключается в том, что если я напишу,

```
s = 'abcdefgh' print (s [2:6])
```
← **Возвращает cdef**

я получу все символы из `s`, начиная с индекса 2 и до индекса 6 (не включая), а именно строку `cdef`. Срез также может указывать размер шага:

```
s = 'abcdefgh' print (s [2:6:2]
```
← **Возвращает ce**

Этот код выведет строку `ce`, поскольку мы начинаем с индекса 2 (`c`), продвигаемся на два индекса вперед к `e`, а затем доходим до конца.

Срезы — это питоновский способ получить подмножество элементов из последовательности. Вы можете даже опустить начальный и/или конечный индекс, чтобы ука–

зать, что вы хотите начать с первого элемента последовательности или закончить ее последним элементом. Например, мы можем получить каждый второй символ из нашей строки с помощью функции

```
s = 'abcdefgh'
print (s [::2]) ← Возвращает aceg
```

Решение

```
def pig_latin (word):
    if word [0] in 'aeiou':
         return f'{word} way'
    return f'{word [1:]} {word [0]} ay'

print (pig_latin ('python'))
```

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr45].

45

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr46].

46

После выполнения упражнения

Трудно преувеличить, насколько часто вам придется работать со строками в Python. Более того, Python часто используется для анализа и работы с текстом. Вот несколько способов, которыми вы можете дополнить это упражнение, чтобы продвинуться дальше:

1. *Поработайте со словами, написанными с заглавной буквы.* Если слово написано с заглавной буквы (т.е. первая буква написана с заглавной, а остальная часть слова нет), то перевод на поросячью латынь должен быть написан аналогичным образом.

2. *Поработайте с пунктуацией.* Если слово заканчивается пунктуацией, то эту пунктуацию следует перенести в конец переведенного слова.
3. *Рассмотрите альтернативную версию поросячьей латыни.* Мы не проверяем, является ли первая буква гласной, мы проверяем, содержит ли слово две разные гласные. Если да, то мы не переносим первую букву в конец. Поскольку слово *wine* содержит две разные гласные (i и e), мы добавим к нему way, что даст нам *wineway*. Слово *wind*, напротив, содержит только одну гласную, поэтому мы перенесем первую букву в конец и добавим ay, получим *indway*. Как бы вы проверили наличие двух разных гласных в слове? (Подсказка: здесь могут пригодиться множества.)

Неизменяемость?

Одним из наиболее важных концептов в Python является различие между изменяемыми и неизменяемыми структурами данных. Основная идея проста: если структура данных неизменяема, то она никогда не может быть изменена.

Например, вы можете определить строку, а затем попытаться изменить ее:

```
s = 'abcd'
s [0] = '!'
```

Вы получите исключение при выполнении этого кода.

Но этот код не будет работать: вы получите исключение, и Python сообщит вам, что вам не разрешено изменять строку.

Многие структуры данных в Python неизменяемы, включая такие базовые, как целые числа и булевы значения. Но именно на строках люди чаще всего спотыкаются, отчасти потому, что строки часто используются, а отчасти потому, что во многих других языках есть изменяемые строки. Почему в Python именно так? Есть несколько причин, глав–

ная из которых заключается в том, что это делает реализацию более эффективной. Но это также связано с тем, что строки являются наиболее распространенным типом, используемым в качестве ключей словарей. Если бы строки были изменяемыми, их нельзя было бы использовать в качестве ключей словарей, или нам пришлось бы разрешить использование изменяемых ключей в словарях, что создало бы целый ряд других проблем.

Поскольку неизменяемые данные не могут быть изменены, мы можем сделать ряд предположений о них. Если мы передаем неизменяемый тип в функцию, то функция не будет его изменять. Если мы разделяем неизменяемые данные между потоками, то нам не нужно беспокоиться об их блокировке, поскольку они не могут быть изменены. А если мы вызываем метод для неизменяемого типа, то получаем обратно новый объект — потому что мы не можем изменять неизменяемые данные.

Обучение работе с неизменяемыми строками занимает некоторое время, но компромиссы в целом оправдывают себя. Если вам понадобится изменяемый тип строки, обратите внимание на StringIO, который обеспечивает файлоподобный доступ к неизменяемому типу в памяти.

Многие новички в Python думают, что неизменяемость — это просто другое слово для обозначения *константы*, но это не так. Константы, которые есть во многих языках программирования, постоянно связывают имя со значением. В Python нет такого понятия, как константа: вы всегда можете переназначить имя, указывающее на новое значение. Но вы не можете изменить строку или кортеж, как бы вы ни старались, например:

```
s = 'abcd'
s [0] = '!'
t = s
```

Не допускается, так как строки неизменяемы.

Теперь переменные s и t относятся к одной и той же строке.

```
s = '!bcd'
```

Переменная s теперь ссылается на новую строку, но t продолжает ссылаться на старую строку без изменений.

Упражнение 6. Предложения на поросячьей латыни

Теперь, когда вы успешно написали переводчик для одного английского слова, давайте усложним задачу: переведите ряд английских слов на поросячью латынь. Напишите функцию `pl_sentence`, которая принимает строку, содержащую несколько слов, разделенных пробелами. (Чтобы упростить задачу, мы не будем запрашивать реальное предложение. В частности, в нем не будет заглавных букв и знаков препинания.) Итак, если кто-то вызовет

```
pl_sentence ('this is a test translation'
```

вывод будет таким:

```
histay isway away estay ranslationtay
```

Печать вывода на одной строке, а не каждого слова на отдельной строке.

Это упражнение может показаться поверхностным, похожим на предыдущее. Но здесь акцент делается не на переводе с поросячьей латыни. Скорее, речь идет о том, как мы обычно используем циклы в Python, и как циклы сочетаются с разбиением строк на части и собиранием их обратно. Также часто возникает необходимость взять последовательность строк и вывести их в одной строке. Есть несколько способов сделать это, мы рассмотрим преимущества и недостатки каждого из них.

Обсуждение

Суть решения практически идентична решению из предыдущего раздела, в котором мы переводили одно слово на поросячью латынь. И снова мы получаем от пользователя текстовую строку. Разница в том, что в данном случае мы не рассматриваем строку как отдельное слово, а рассматриваем ее как предложение, то есть нам нужно разделить ее на отдельные слова. Это можно сделать с помощью функции `str.split`. `str.split` может принимать аргумент, который определяет, какая строка должна использоваться в качестве разделителя между полями.

Часто бывает так, что для разделения полей необходимо использовать все символы пробелов, независимо от их количества. В таком случае не передавайте аргумент вообще;

Python будет рассматривать любое количество пробелов, табуляций и новых строк как один символ разделения. Разница может быть значительной:

```
s = 'abc def ghi' ←Два разделяющих пробела.
s = 'abc def ghi' ←Возвращает ['abc', '', 'def', '', 'ghi'].
s.split ()        ←Возвращает ['abc', 'def', 'ghi'].
```

ПРИМЕЧАНИЕ Если вы не передаете никакие аргументы в `str.split`, это фактически то же самое, что передать `None`. Вы можете передать в `str.split` любую строку, а не только строку с одним символом. Это означает, что если вы хотите разделить строку на `::`, вы можете это сделать. Однако вы не можете разделить более чем на одну, используя одновременно `,` и `::` в качестве разделителей полей. Для этого нужно использовать регулярные выражения и функцию `re.split` из стандартной библиотеки Python.

Таким образом, мы можем разбить ввод пользователя на слова — опять же, предполагая, что нет знаков препинания, — и затем перевести каждое отдельное слово в поросячью латынь. Если однословная версия нашей программы могла просто сразу вывести результат, то эта программа должна хранить

результаты, а затем вывести их сразу. Конечно, можно использовать для этого строку и вызывать += на строке с каждой итерацией. Но, как правило, не стоит строить строки таким образом. Лучше добавлять элементы в список с помощью `list.append`, а затем вызывать `str.join`, чтобы превратить элементы списка в длинную строку.

Это связано с тем, что строки неизменяемы, а `+=` в строке заставляет Python создавать новую строку. Если мы продолжим добавлять к строке, то каждый раз будет создаваться новый объект, содержимое которого будет больше, чем на предыдущей итерации. Однако, списки являются изменяемыми, и добавление к ним с помощью `list.append` относительно не требует больших затрат как памяти, так и вычислений.

Решение

```
def pl_sentence (sentence):
    output = []
    for word in sentence.split ():
         if word [0] in 'aeiou':
              output.append (f'{word} way')
         else:
              output.append (f'{word [1:]} {word
              [0]} ay')

    return ' '.join (output)

print (pl_sentence ('this is a test'))
```

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr47].

47

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr48].

48

После выполнения упражнения

Разделение, объединение и управление строками — обычные действия в Python. Вот некоторые дополнительные действия, которые вы можете попробовать, чтобы продвинуться еще дальше:

1. Возьмите текстовый файл, создайте (и распечатайте) бессмысленное предложение из n–го слова в каждой из первых 10 строк, где n — номер строки.
2. Напишите функцию, которая транспонирует список строк, в котором каждая строка содержит несколько слов, разделенных пробелами. В частности, она должна работать таким образом, что, если вы передаете функции список, например, `['abc def ghi', 'jkl mno pqr', 'stu vwx yz']`, она возвращает `['abc jkl stu', 'def mno vwx', 'ghi pqr yz']`.
3. Прочитайте лог–файл Apache. Если там есть ошибка 404 — вы можете просто поискать `' 404 '`, если вы хотите отобразить IP–адрес, который должен быть первым элементом.

Упражнение 7. Убби–Дубби

Услышав, что строки в Python неизменяемы, многие задаются вопросом, как можно использовать этот язык для обработки текста. В конце концов, если строки нельзя изменять, то как можно с ними работать?

Более того, бывают случаи, когда простой цикл `for`, который мы использовали в примере с поросячьей латынью, не подходит. Если мы изменяем каждое слово только один раз, то все хорошо, но, если мы потенциально изменяем его несколько раз, мы должны быть уверены, что каждая из модификаций не повлияет на будущие модификации.

Это упражнение призвано помочь вам научиться думать таким образом. Здесь вы будете использовать переводчик с английского на другой секретный детский язык — Убби–Дубби. (Этот язык был популяризирован в замечательной американской детской программе Zoom, которая шла по телевидению во времена

моего детства.) Правила Убби–Дубби даже проще, чем правила поросячьей латыни, хотя программировать переводчик сложнее и требует немного больше размышлений.

В Убби–Дубби перед каждой гласной (a, e, i, o или u) ставится `ub`. Таким образом, `milk` становится `mubilk` (`m — ub — ilk`), а `program` — `prubogrubam` (`prub — ogrub — am`). Теоретически вы ставите `ub` только перед гласным *звуком*, а не перед каждой гласной буквой. Учитывая, что это книга о Python, а не о лингвистике, я надеюсь, что вы простите небольшое расхождение в определении.

На языке Убби–Дубби очень весело говорить, и это в какой–то степени волшебно, когда вы начинаете понимать другого человека, говорящего на нем. Даже если вы не понимаете его, Убби–Дубби звучит очень забавно.

Для этого упражнения вы напишете функцию `ubbi_dubbi`, которая принимает в качестве аргумента одно слово (строку). Она возвращает строку — перевод этого слова на язык Убби–Дубби. Таким образом, если вызвать функцию с аргументом `octopus`, она вернет строку `uboctubopubus`. А если пользователь передаст аргумент `elephant`, то будет выведен `ubelubephubant`.

Как и в оригинальном переводчике с поросячьей латыни, вы можете игнорировать заглавные буквы, пунктуацию и угловые случаи, например, когда несколько гласных сочетаются, образуя новый звук. Когда две гласные находятся рядом друг с другом, перед каждой из них ставится `ub`. Таким образом, `soap` станет `suboubap`, несмотря на то что `oa` объединяется в один гласный звук.

Подобно заданию «Поросячья латынь», в этом упражнении применяются различные способы, которыми нам часто приходится сканировать строки в поисках определенных шаблонов или переводить из одной структуры данных Python или шаблона в другую; а также показывается, как итерации могут играть центральную роль в этом.

Обсуждение

Задача состоит в том, чтобы спросить у пользователя слово, а затем перевести это слово в Убби–Дубби. Это немного дру–

гая задача, чем в случае с поросячьей латынью, потому что нам нужно работать побуквенно. Мы не можем просто проанализировать слово и выдать результат на основе всего слова. Более того, мы не должны попасть в бесконечный цикл, в котором мы пытаемся добавить `ub` перед `u` в `ub`.

Решение заключается в итерационном переборе каждого символа в `слове`, добавлении его в список, `выводе`. Если текущий символ является гласным, то мы добавляем `ub` перед буквой. В противном случае мы просто добавляем букву. В конце программы мы соединяем, а затем печатаем буквы вместе. На этот раз мы соединяем буквы не с помощью символа пробела (`' '`), а с помощью пустой строки (`''`). Это означает, что результирующая строка будет состоять из букв, соединенных между собой ничем, или, как мы часто называем такие коллекции, — *word*.

Решение

```
def ubbi_dubbi (word):
    output = []
    for letter in word:
         if letter in 'aeiou':
              output.append (f'ub {letter}')
         else:
              output.append (letter)

    return ''.join (output)

print (ubbi_dubbi ('python'))
```

Почему нужно добавлять к списку, а не к строке? Чтобы не выделять слишком много памяти. Для коротких строк это не имеет большого значения.
Но для длинных циклов и больших строк это будет плохой идеей.

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr49].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr50].

После выполнения упражнения

Часто возникает необходимость заменить в строке одно значение на другое. В Python есть несколько различных способов сделать это. Вы можете использовать `str.replace` или `str.translate`, два строковых метода, которые переводят строки и множества символов, соответственно. Но иногда нет другого выбора, кроме как просмотреть строку итерациями, найти нужный нам шаблон, а затем добавить измененную версию в список, который будет расти со временем:

1. *Работа словами, написанными с заглавной буквы.* Если слово написано с заглавной буквы (т.е. первая буква заглавная, а остальная часть слова — нет), то перевод на Убби–Дубби должен быть написан с такой же заглавной буквы.
2. *Удаление имен авторов.* В научных кругах принято удалять имена авторов из статьи, представленной на рецензирование. Получив строку со статьей и отдельный список строк с именами авторов, замените все имена в статье символами _.
3. *URL-кодирование символов.* В URL–адресах мы часто заменяем специальные и непечатные символы знаком `%`, за которым следует значение ASCII символа в шестнадцатеричной системе. Например, если в URL должен быть символ пробела (ASCII 32, он же 0x20), мы заменим его на `%20`. Получив строку, закодируйте в URL любой символ, не являющийся буквой или цифрой. Для целей этого упражнения мы будем считать, что все символы являются ASCII (т.е. длиной в один байт), а не многобайтовыми символами UTF–8. Возможно, вам будет полезно узнать о функциях `ord` и `hex`.

Упражнение 8. Сортировка строк

Если строки неизменяемы, значит ли, что они навсегда останутся в том виде, в каком существуют сейчас? Отчасти. Мы не можем изменить сами строки, но мы можем создавать новые строки на их основе, используя комбинацию встроенных функций и строковых методов. Умение обходить неизменяемость строк и создавать функции, которые эффективно изменяют строки, несмотря на их неизменяемость, — полезный навык.

В этом упражнении вы рассмотрите эту идею, написав функцию `strsort`, которая принимает на вход строку и возвращает строку. Возвращаемая строка должна содержать те же символы, что и входная, за исключением того, что ее символы должны быть отсортированы в порядке от наименьшего значения к наибольшему Unicode. Например, результатом вызова команды `strsort ('cba')` будет строка `abc`.

Обсуждение

В реализации решения `strsort` используется тот факт, что строки Python являются последовательностями. Обычно мы считаем, что это уместно в цикле `for`, поскольку можем перебирать символы в строке. Однако нам не нужно ограничиваться такими ситуациями.

Например, мы можем использовать встроенную функцию `sorted`, которая принимает итерируемый объект, — что означает не только последовательность, но и все, что мы можем итерировать, например, множество файлов — и возвращает его элементы в отсортированном порядке.

В результате `sorted` отсортирует символы в порядке Unicode. Однако она возвращает список, а не строку.

Чтобы превратить наш список в строку, мы используем метод `str.join`. Используем пустую строку (`''`) в качестве клея, который будем использовать для соединения элементов, таким образом возвращая новую строку, символы которой будут такими же, как и во входной строке, но в отсортированном порядке.

Unicode

Что такое Unicode? Идея проста, но ее реализация может показаться чрезвычайно сложной и сбить с толку многих разработчиков.

Идея Unicode заключается в том, что мы должны использовать компьютеры для представления любого символа, задействованного в любом языке в любое время. Это очень важная цель, поскольку она означает, что у нас не будет проблем с созданием документов, в которых мы хотим отобразить русский, китайский и английский языки на одной странице. До появления Unicode смешивать множества символов из нескольких языков было сложно или даже невозможно.

Unicode присваивает каждому символу уникальный номер. Но эти номера могут (как вы понимаете) быть очень большими. Поэтому мы должны взять номер символа Unicode (*кодовую точку*) и перевести его в формат, который можно хранить и передавать в виде байтов. В Python и многих других языках используется UTF–8, который представляет собой кодировку переменной длины, то есть для разных символов может потребоваться разное количество байт. Символы, существующие в ASCII, кодируются в UTF–8 с тем же номером, который они используют в ASCII, в одном байте. Французский, испанский, иврит, арабский, греческий и русский языки используют два байта для своих символов, не входящих в ASCII. А китайский, как и ваши детские эмодзи, — три байта или больше.

Насколько это важно для нас? Одновременно очень и не очень. С одной стороны, хорошо иметь простой способ работы с разными языками. С другой стороны, легко забыть, что существует разница между байтами и символами и что иногда (например, при работе с файлами на диске) нужно переводить из байтов в символы, или наоборот.

Для получения более подробной информации о символах и строках, а также о том, как Python хранит символы в наших строках, я рекомендую ознакомиться с выступлением Неда Батчелдера с PyCon 2012: [qr51].

Решение

```
def strsort (a_string):
    return ''.join (sorted (a_string))

print (strsort ('cbjeaf'))
```

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr52].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr53].

После выполнения упражнения

Это упражнение призвано дополнительно напомнить вам, что строки — это последовательности, и поэтому их можно использовать везде, где можно использовать другие последовательности (списки и кортежи). Мы нечасто думаем о сортировке строк, но нет никакой разницы в использовании `sorted` для строк, списков или кортежей. Элементы (в случае строки — символы) возвращаются в отсортированном порядке.

Однако `sorted` возвращает список, а мы хотели получить строку. Поэтому нам нужно превратить полученный список обратно в строку — то, для чего предназначен `str.join`. `str.split` и `str.join` — два метода, с которыми вы должны быть хорошо знакомы, потому что они очень полезны и помогают во многих случаях.

Рассмотрим несколько других вариаций и дополнений к этому упражнению, в которых используются `str.split` и `str.join`, а также `sorted`:

1. Возьмите строку Tom Dick Harry и разбейте ее на отдельные слова, а затем отсортируйте эти слова по алфавиту. После сортировки напечатайте их с запятыми (`,`) между именами.
2. Какое слово является последним в текстовом файле?
3. Какое самое длинное слово в текстовом файле?

Обратите внимание, что для второй и третьей задач вам, возможно, захочется почитать о `ключевом` параметре и типах значений, которые можно ему передавать. Хорошее введение с примерами находится здесь: [qr54].

Подводя итоги

Программисты Python постоянно имеют дело с текстом. Будь то чтение из файлов, вывод на экран или просто использование словарей, строки — это тип данных, с которым мы, скорее всего, знакомы по другим языкам.

В то же время строки в Python необычны тем, что они также являются последовательностями, а значит, мышление в Python требует учета их свойств, подобных последовательностям. Это означает поиск (с помощью `in`), сортировку (с помощью sorted) и использование срезов. Также необходимо подумать о том, как можно превратить строки в списки (используя `str.split`) и как превратить последовательности обратно в строки (используя `str.join`). Хотя эти задачи могут показаться простыми, они регулярно встречаются в производственном коде Python. Тот факт, что эти структуры данных и методы написаны на языке C и существуют уже много лет, означает, что они по-прежнему очень эффективны и нет необходимости в том, чтобы изобретать их заново.

3. Списки и кортежи

Рассмотрим программу, которая работает с документами, отслеживает пользователей, регистрирует IP–адреса, которые заходили на сервер, или хранит имена и даты рождения детей в школе. Во всех этих случаях нам нужно хранить много информации. Мы хотим отображать, искать, дополнять и изменять эту информацию.

Это настолько распространенные задачи, что каждый язык программирования поддерживает работу с *коллекциями* — структурами данных, предназначенных для работы с такими случаями. Списки и кортежи — это встроенные коллекции Python. Технически они отличаются тем, что списки являются изменяемыми, а кортежи — неизменяемыми. Но на практике списки предназначены для последовательностей одного типа, а кортежи — для последовательностей разных типов.

Например, серию документов, пользователей или IP–адресов лучше всего хранить в списке, потому что у нас много объектов одного типа. Запись, содержащую чье–то имя и дату рождения, лучше всего хранить в кортеже, поскольку имя и дата рождения относятся к разным типам. Однако множество таких кортежей с именами и датами рождения можно хранить в списке, поскольку он будет содержать последовательность кортежей и все кортежи будут одного типа.

Поскольку списки являются изменяемыми, они поддерживают работу с гораздо большим числом методов и операторов. В конце концов, с кортежем мало что можно сделать, кроме как передать его, получить его элементы и сделать несколько запросов о его содержи–

мом. Списки, напротив, можно расширять, сокращать и изменять, а также искать, сортировать и заменять. Поэтому вы не можете добавить размер обуви человека к кортежу «имя–дата рождения», который вы для него создали. Но вы можете добавить множество дополнительных кортежей имя–дата рождения в созданный вами список, а также удалить элементы из этого списка, если они больше не являются учениками школы.

Потребуется время, чтобы научиться определять, в каких случаях стоит использовать списки, а в каких кортежи. Если вы еще не поняли разницу, то это не ваша вина!

И списки, и кортежи являются *последовательностями* Python, это означает, что мы можем использовать по отношению к ним циклы `for`, искать с помощью оператора `in` и извлекать из них данные как с помощью отдельных индексов, так и с помощью срезов. Третий тип последовательности в Python — это строки, которые мы рассматривали в предыдущей главе. Я считаю, что полезно думать о последовательностях в таком ключе.

Таблица 3.1. Сравнение последовательностей

Тип	Изменяем?	Содержит	Синтаксис	Использование
`str`	Нет	Одноэлементные строки.	`s = 'abc'`	`s [0] # возвращает 'a'`
`Список`	Да	Любой тип Python.	`mylist = [10, 20, 30, 40, 50]`	`mylist [2] # возвращает 30`
`Кортеж`	Нет	Любой тип Python.	`t = (100, 200, 300, 400, 500)`	`t [3] # возвращает 400`

В этой главе мы будем практиковаться в работе со списками и кортежами. Мы рассмотрим, как создавать, изменять (в случае списков) и использовать их для отслеживания данных. Мы также будем использовать *генератор списков* — синтаксис, который многим кажется непонятным, но который позволяет нам

взять один итерируемый объект Python и создать на его основе новый список. В этой и последующих главах мы будем много говорить про генераторы, если вы не знакомы с ними, ознакомьтесь с ссылками, приведенным в таблице 3.2.

Таблица 3.2. Что вам нужно знать

Понятие	Что это?	Пример	Чтобы узнать подробнее
`Список`	Упорядоченная, изменяемая последовательность.	`[10, 20, 30]`	
`Кортеж`	Упорядоченная, неизменяемая последовательность.	`(3, 'clubs')`	
Генераторы списков	Возвращает список на основе итерируемого объекта.	`# возвращает ['10', '20', '30'] [str(x) for x in [10,20, 30]]`	
`range`	Возвращает итерируемую последовательность целых чисел.	`# каждое третье целое число, от 10 до (и не включая) 50 numbers = range (10, 50, 3)`	
`operator.itemgetter`	Возвращает функцию, которая работает как квадратные скобки.	`# final ('abcd') == 'd' final = operator.itemgetter (-1)`	

Окончание таблицы

Понятие	Что это?	Пример	Чтобы узнать подробнее
`collections.Counter`	Подкласс словаря, полезный для подсчета элементов итерируемого объекта.	`# примерно то же самое, что {'a':2, 'b':2, 'c':1, 'd':1} c = collections.Counter ('abcdab')`	
`max`	Встроенная функция, возвращающая самый большой элемент итерируемого объекта.	`# вернет 30 max ([10, 20, 30])`	
`str.format`	Строковый метод, возвращающий новую строку, основанную на шаблоне (похоже на f-строки).	`# вернет 'x = 100, y = [10, 20, 30] ' 'x = {0}, y = {1}' .format (100, [10, 20, 30])`	

Упражнение 9. Первый–последний

Для многих программистов, имеющих опыт работы на Java или C#, динамическая природа Python является довольно странной. Как язык программирования может не знать, какому типу присвоена определенная переменная? Поклонники динамических языков, таких как Python, отвечают, что это позволяет нам писать общие функции, которые обрабатывают множество различных типов.

Действительно, нам необходимо это сделать. Во многих языках функцию можно определять несколько раз, при условии, что каждое определение имеет разные параметры. В Python функцию можно определить только один раз — точнее, определение функции во второй раз перезапишет первое определение, поэтому нам нужно использовать другие методы для работы с разными типами входных данных.

В Python можно написать одну функцию, которая работает со многими типами, а не множество почти одинаковых функций, каждая из которых предназначена для определенного типа. Такие функции демонстрируют элегантность и мощь динамической типизации.

Тот факт, что последовательности — строки, списки и кортежи реализуют многие из одинаковых API, не случаен. Python позволяет нам писать общие функции, которые могут применяться ко всем из них. Например, все три типа последовательностей можно искать с помощью `in`, возвращать отдельные элементы с помощью индекса и возвращать несколько элементов с помощью среза.

Мы отработаем эти идеи при помощи данного упражнения. Напишите функцию `firstlast`, которая принимает последовательность (строку, список или кортеж) и возвращает первый и последний элементы этой последовательности как двухэлементную последовательность того же типа. Так `firstlast ('abc')` вернет строку `ac`, а `firstlast ([1,2,3,4])` вернет список [1,4].

Обсуждение

Это упражнение настолько же сложное, насколько и короткое. Однако я считаю, что это поможет продемонстрировать разницу между извлечением отдельного элемента из последовательности и между фрагментом из этой последовательности. Это также показывает мощь динамического языка: нам не нужно определять несколько различных версий `firstlast`, каждая из которых работает с разными типами. Вместо этого мы

можем определить одну функцию, которая работает не только со встроенными последовательностями, но и с любыми новыми типами, которые мы можем определить для работы с индексами и срезами.

В первую очередь программисты Python изучают способы извлечения элемента из последовательности — строки, списка или кортежа — с помощью квадратных скобок и числового индекса. Так вы можете получить первый элемент `s` с помощью `s [0]` и последний элемент `s` с помощью `s [-1]`.

Но это еще не все. Вы также можете получить срез, или подмножество элементов последовательности, используя двоеточие внутри квадратных скобок. Самый простой и очевидный способ — это написать что-то вроде `s [2:5]`, означающее, что вы хотите получить строку, которая начинается с `s`, следуя от индекса 2 до индекса 5, но не включая его. (Помните, что в срезе конечное число всегда «до, но не включая».)

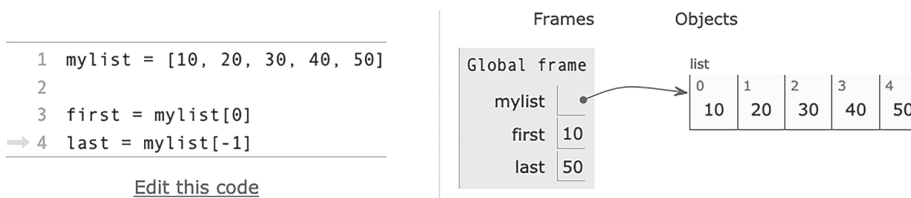


Рисунок. 3.1.

При извлечении одного элемента из последовательности (рисунок 3.1), вы можете получить любой тип. Строковые индексы возвращают односимвольные строки, а списки и кортежи могут содержать что угодно. Напротив, когда вы используете срез, вы гарантированно получите тот же тип обратно — срез кортежа будет кортежем, независимо от размера среза или элементов, которые он содержит. А срез списка вернет список. Обратите внимание, что на рисунках 3.2 и 3.3, взятых из Python Tutor, структуры данных различны, а значит, и результаты извлечения из каждого типа будут разными.

Рисунок 3.2. Извлечение срезов из списка (из Python Tutor). Frames — фреймы, Objects — объекты, Global frame — глобальный фрейм, list — список.

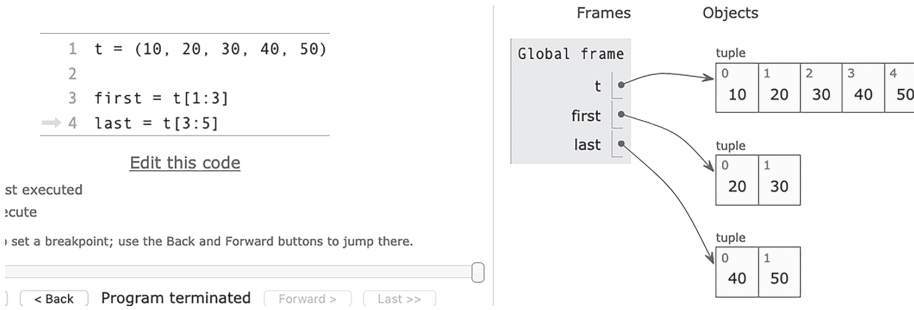


Рисунок 3.3. Извлечение срезов из кортежа (из Python Tutor). Frames — фреймы, Objects — объекты, Global frame — глобальный фрейм, list — список.

Не выходите за пределы

При извлечении одного индекса нельзя выходить за границы:

```
s = 'abcd'
s [5] # вызывает исключение IndexError
```

Однако при извлечении среза, Python более великодушен, игнорирует любой индекс, выходящий за границы структуры данных:

```
s = 'abcd'
s [3:100] # возвращает 'd'
```

На рисунках 3.2 и 3.3 индекс 5 отсутствует. И все же Python закрыл глаза на это, показав данные до конца. Мы так же легко могли бы опустить последнее число.

Учитывая, что мы пытаемся получить первый и последний элементы `sequence`, а затем соединить их вместе, может показаться разумным взять их оба (с помощью индексов), а затем сложить их вместе:

```
# не является реальным решением!
def firstlast (sequence):
    return sequence [0] + sequence [-1]
```

Но именно это и происходит на самом деле (рис. 3.4):

```
def firstlast (sequence):
    return sequence [0] + sequence [-1]

t1 = ('a', 'b', 'c')
output1 = firstlast (t1)
print (output1)

t2 = (1,2,3,4)
output2 = firstlast (t2)
print (output2)
```

Не является реальным решением!

Печатает строку 'ac', а не ('a', 'c').

Печатает целое число 5, а не (1, 4).

Мы не можем просто использовать `+` для отдельных элементов наших кортежей. Как показано на рисунке 3.4, если элементы являются строками или целыми числами, то использование `+` для этих двух элементов даст нам неправильный ответ. Мы хотим добавлять кортежи — или `последовательность` любого типа.

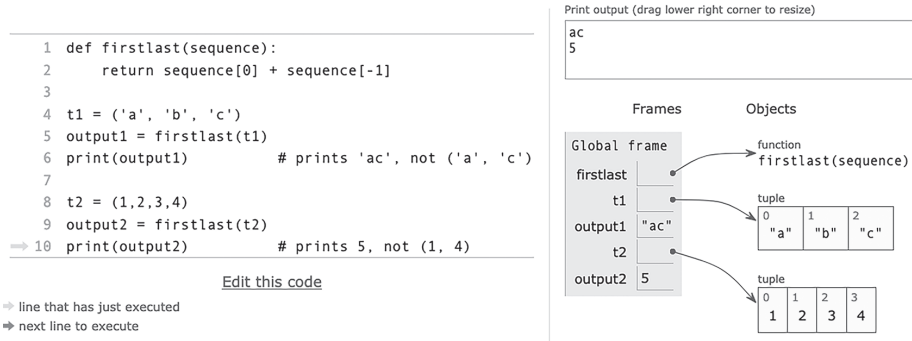


Рисунок 3.4. Наивное, неправильное добавление фрагментов (из Python Tutor). (line that has just executed — строка, которая только что была выполнена, next line to execute — следующая строка для выполнения, # prints the string 'ac', not ('a', 'c') — печатает строку 'ac', а не ('a', 'c'), # prints 5, not (1, 4) — печатает целое число, Frames — фреймы, Objects — объекты, Global frame — глобальный фрейм, list — список, function — функция, print output (drag lower right corner to resize) — вывод на печать (потяните за правый нижний угол, чтобы изменить размер).)

Самый простой способ сделать это — использовать срез, используя `s [:1]` для получения первого элемента и `s [−1:]` для получения последнего элемента (рисунок 3.5). Обратите внимание, что мы должны написать `s [−1:]`, чтобы последовательность началась с элемента по адресу –1 и закончилась в конце самой последовательности.

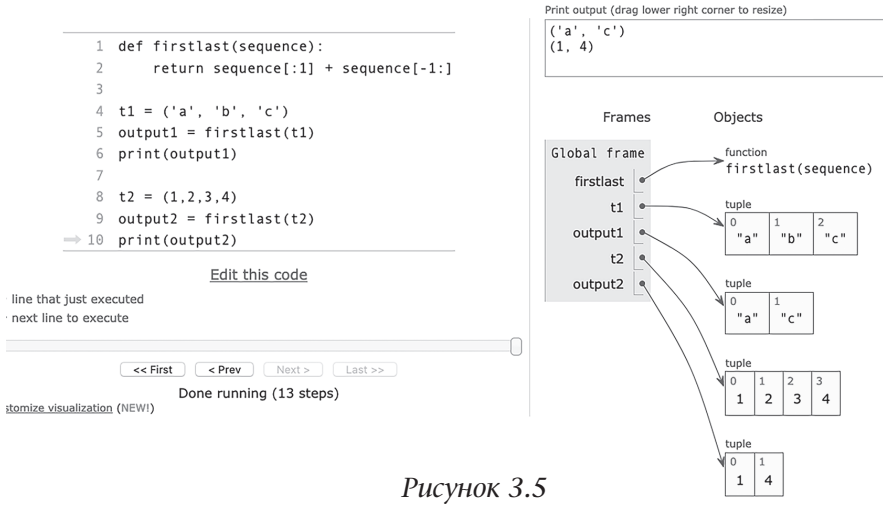


Рисунок 3.5

Суть в том, что, когда вы извлекаете срез из объекта x, вы получаете новый объект того же типа, что и x. Но если вы извлекаете отдельный элемент из x, то получаете все, что было сохранено в x — что может быть того же типа, что и x, но вы не можете быть в этом уверены.

Решение

```
def firstlast(sequence):          # В обоих случаях мы используем срезы, а не индексы.
    return sequence[:1] + sequence[-1:]

print(firstlast('abcd'))
```

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr63].

63

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr64].

64

После выполнения упражнения

Мы можем воспользоваться преимуществами динамической типизации Python, то есть, хотя данные сильно типизированы, переменные не имеют типов. Это означает, что мы можем написать функцию, которая примет любой индексируемый тип (т.е. такой, который может получить в качестве аргумента либо одиночный индекс, либо срез) и затем вернет что-то соответствующее. Это распространенная техника в Python, с которой вы должны быть хорошо знакомы, например:

1. Не пишите одну функцию, которая возводит в квадрат целые числа, и другую, которая возводит в квадрат числа с плавающей точкой. Напишите одну функцию, которая обрабатывает все числа.
2. Не пишите одну функцию, которая находит наибольший элемент строки, другую, которая делает то же самое для

списка, и третью, которая делает то же самое для кортежа. Напишите одну функцию для обоих случаев.

3. Не пишите одну функцию для поиска самого большого слова в файле, которая работает с файлами, и другую, которая работает с симуляторами файлов io.StringIO, используемыми при тестировании. Напишите одну функцию для обоих случаев.

Срезы — это отличный способ получить часть данных. Будь то подстрока или часть списка, срезы позволяют получить только часть последовательности. Студенты на моих курсах часто спрашивают, как они могут выполнить итерацию только по последним n элементам списка. Когда я напоминаю им, что это можно сделать с помощью среза `mylist [-3:]` и цикла `for`, они несколько удивляются и смущаются, что не додумались до этого сами: они были уверены, что это должно быть сложнее.

Вот несколько идей для других задач, которые вы можете попробовать решить, используя индексы и срезы:

1. Напишите функцию, которая принимает список или кортеж чисел. Функция должна возвращать двухэлементный список, содержащий сумму чисел с четным индексом и сумму чисел с нечетным индексом соответственно. Вызвав функцию `even_odd_sums ([10, 20, 30, 40, 50, 60])`, вы получите `[90, 120]`.
2. Напишите функцию, которая принимает список или кортеж чисел. Функция должна возвращать результат поочередного сложения и вычитания чисел друг из друга. Вызвав функцию `plus_minus ([10, 20, 30, 40, 50, 60])`, вы получите результат `10+20-30+40-50+60` или `50`.
3. Напишите функцию, которая частично эмулирует встроенную функцию `zip`, принимая любое количество итерируемых объектов и возвращая список кортежей. Каждый кортеж будет содержать по одному элементу итерируемого объекта, переданного в функцию.

Следовательно, если я вызову `myzip ([10, 20,30], 'abc')`, то получу `[(10, 'a'), (20, 'b'), (30, 'c')]`. Вы можете вернуть список (не итератор) и предположить, что все итерируемые объекты имеют одинаковую длину.

Являются ли списки массивами?

Те, кто только знакомится с Python, часто ищут такой тип, как `массив`. Но в Python разработчики используют списки, если им необходимы массивы или похожая структура.

Списки — это не массивы: массивы имеют фиксированную длину, а также тип. И хотя потенциально можно утверждать, что списки Python обрабатывают только один тип, а именно все, что наследуется от встроенного класса `object`, определенно неверно, что списки имеют фиксированную длину. Упражнение 9 демонстрирует это довольно наглядно, но без использования методов `list.append` или `list.remove`.

ПРИМЕЧАНИЕ В стандартной библиотеке Python есть тип `array`, а специалисты по анализу данных обычно используют массивы NumPy. Однако по большей части массивы в Python не нужны и не используются. Они не соответствуют динамической природе языка. Вместо этого мы обычно используем списки и кортежи.

По сути, списки Python реализуются как массивы указателей на объекты Python. Но если массивы имеют фиксированный размер, то как Python может использовать их для реализации списков? Ответ заключается в том, что Python выделяет некоторое дополнительное пространство в массиве списка, чтобы мы могли добавить в него несколько элементов. Но в определенный момент, когда мы добавим достаточно элементов в наш список и эти свободные места будут

использованы, Python выделит новый массив и переместит все указатели в это место. Это делается для нас автоматически, но это показывает, что добавление элементов в список не полностью освобождает от вычислительных затрат. Вы можете увидеть это в действии, используя `sys.getsizeof`, который показывает количество байт, необходимых для хранения списка (или любой другой структуры данных):

```
>>> import sys
>>> mylist = []
>>> for i in range (25):
… →   l = len (mylist)
… →   s = sys.getsizeof (mylist)
… →   print (f'len = {l}, size = {s}')
… →   mylist.append (i)
```

Выполнение этого кода дает нам следующий результат:

```
len = 0, size = 64
len = 1, size = 96
len = 2, size = 96
len = 3, size = 96
len = 4, size = 96
len = 5, size = 128
len = 6, size = 128
len = 7, size = 128
len = 8, size = 128
len = 9, size = 192
len = 10, size = 192
len = 11, size = 192
len = 12, size = 192
len = 13, size = 192
len = 14, size = 192
len = 15, size = 192
```

```
len = 16, size = 192
len = 17, size = 264
len = 18, size = 264
len = 19, size = 264
len = 20, size = 264
len = 21, size = 264
len = 22, size = 264
len = 23, size = 264
len = 24, size = 264
```

Как видите, список растет по мере необходимости, но всегда содержит свободное место, что позволяет ему не увеличиваться, если вы добавляете всего несколько элементов.

ПРИМЕЧАНИЕ Различные версии Python, а также различные операционные системы и платформы могут распределять память иначе, чем я показал здесь.

Насколько это важно для повседневной разработки на Python? Как и в случае с распределением памяти и реализацией языка Python, я считаю это полезными дополнительными знаниями, которые пригодятся вам, когда вы столкнетесь с проблемами в оптимизации, или просто для того, чтобы лучше понять процессы Python.

Но если вы регулярно беспокоитесь о размере ваших структур данных или о том, как Python выделяет память, то я бы сказал, что вы, вероятно, беспокоитесь не о тех вещах — или используете не тот язык для работы. Python потрясающе подходит под решение множества задач, кроме того, его сборщик мусора работает достаточно хорошо большую часть времени. Но вы не можете контролировать работу сборщика мусора, и Python в значительной степени предполагает, что вы передадите управление языку.

Упражнение 10. Суммируем что угодно

Мы рассмотрели, как можно написать функцию, которая принимает несколько различных типов. Вы также узнали, как можно написать функцию, которая получает на входе аргумент и на выходе возвращает различные типы.

В этом упражнении вы увидите, как можно еще работать с типами. Что произойдет, если вы будете применять методы не для самого аргумента, а для элементов внутри аргумента? Например, что, если вы хотите просуммировать сумму элементов списка — независимо от того, являются ли эти элементы целыми числами, числами с плавающей точкой, строками или даже списками?

Эта задача просит вас переопределить функцию `mysum` из главы 1 так, чтобы она могла принимать любое количество аргументов. Все аргументы должны быть одного типа и подходить под работу с оператором `+`. (Таким образом, функция должна работать с числами, строками, списками и кортежами, но не с множествами и словарями.)

ПРИМЕЧАНИЕ Python 3.9, выпуск которого был запланирован на осень 2020 года, очевидно включает поддержку `|` для словарей. Более подробную информацию см. в PEP 584 [qr65].

65

Результатом должна быть новая, более длинная последовательность типа, заданного параметрами. Следовательно, результатом `mysum ('abc', 'def')` будет строка `abcdef`, а результатом `mysum ([1,2,3], [4,5,6])` будет шестиэлементный список `[1,2,3,4,5,6]`. Конечно, он также должен вернуть целое число `6`, если мы вызовем `mysum (1,2,3)`.

Выполнение этого упражнения даст вам возможность подумать о последовательностях, типах и о том, как нам проще всего создавать возвращаемые значения разных типов из одной и той же функции.

Обсуждение

Реализация новой версии `mysum` сложнее той, которую мы видели ранее. Она по–прежнему принимает любое количество аргументов, которые помещаются в кортеж `items` благодаря оператору `splat (*)`.

> **СОВЕТ** Хотя мы традиционно называем параметр, который «принимает любое количество аргументов», как `*args`, вы можете использовать любое имя. Важной частью является `*`, а не имя параметра: он по–прежнему работает одинаково и всегда является кортежем.

Первое, что мы делаем, это проверяем, получили ли мы какие–либо аргументы. Если нет, мы возвращаем `items` — пустой кортеж. Это необходимо, потому что остальная часть функции требует, чтобы мы знали тип переданных аргументов и чтобы у нас был элемент с индексом 0. Без аргументов ни то, ни другое работать не будет.

Обратите внимание, что мы не проверяем пустой кортеж, сравнивая его с `()` или проверяя, что его длина равна 0. Вместо этого мы можем написать `if not items`, которое запросит булево значение нашего кортежа. Поскольку пустая последовательность в Python в булевом контексте равна `False`, мы получим `False`, если `args` пуст, и `True` в противном случае.

В следующей строке мы берем первый элемент `items` и присваиваем его `output` (рис. 3.6). Если это число, то `output` будет числом, если строка, то строка, и так далее. Это дает нам базовое значение, к которому мы будем добавлять (используя `+`) все последующие значения в `items`.

После этого мы делаем то же самое, что и в оригинальной версии `mysum`, но вместо итерации по всем `items` мы можем теперь итерировать по `items [1:]` (рисунок 3.7), то есть по всем элементам, кроме первого. Здесь мы снова видим ценность фрагментов Python и то, как мы можем использовать их для решения проблем.

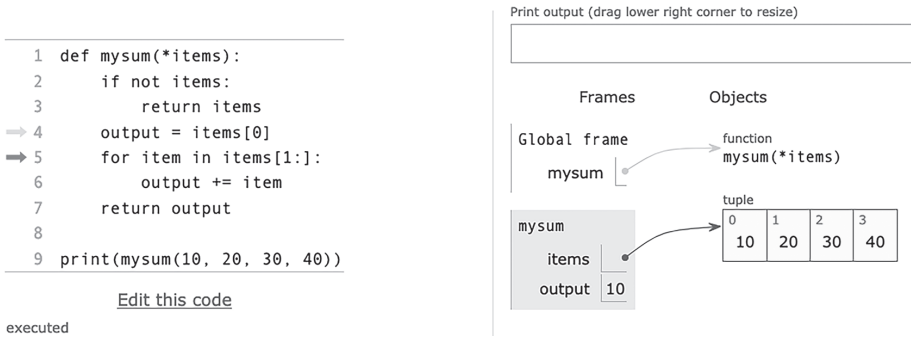


Рисунок 3.6. После присваивания первого элемента `output` *(из Python Tutor).*

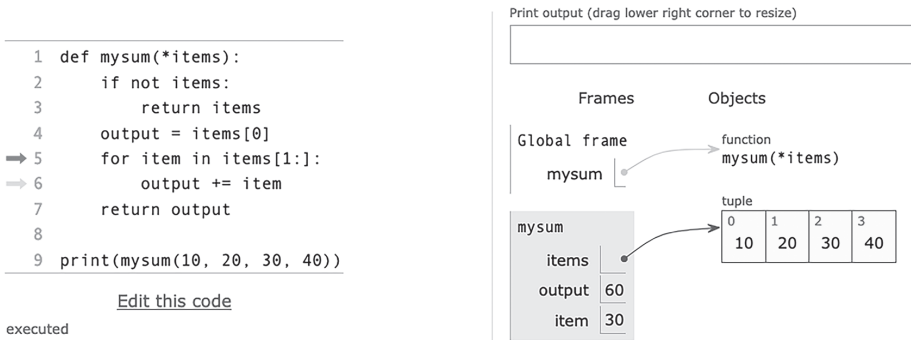


Рисунок 3.7. После добавления элементов в `output` *(из Python Tutor).*

Вы можете считать эту реализацию `mysum` такой же, как и нашу первоначальную версию, за исключением того, что вместо добавления каждого элемента к 0, мы добавляем каждый элемент к `items [0]`.

Но подождите, а что, если человек передал нам только один аргумент, и поэтому `args` не содержит ничего в индексе 1? К счастью, срезы прощают нам это и позволяют указывать индексы за границами последовательности. В таком случае мы получим пустую последовательность, по которой цикл `for` будет выполняться ноль раз. Это означает, что мы просто получим значение `items [0]`, возвращенное нам в качестве `output`.

Решение

```
def mysum (*items):
    if not items:
        return items
    output = items [0]

    for item in items [1:]:
        output += item
    return output
```

В Python все считается True в if, кроме None, False, 0 и пустых коллекций. Поэтому если кортеж items пуст, мы просто вернем пустой кортеж.

Мы предполагаем, что элементы items могут быть сложены вместе.

```
print (mysum ())
print (mysum (10, 20, 30, 40)) print (mysum ('a', 'b', 'c', 'd'))
print (mysum ([10, 20, 30], [40, 50, 60], [70, 80]))
```

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr66].

66

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr67].

67

После выполнения упражнения

Это упражнение демонстрирует некоторые способы использования преимуществ динамической типизации Python для создания функции, которая работает с различными типами входных данных и даже производит различные типы выходных данных. Вот несколько других задач, которые вы можете попробовать решить и которые имеют схожие цели:

1. Напишите функцию `mysum_bigger_than`, которая работает так же, как `mysum`, за исключением того, что она принимает первый аргумент, предшествующий `*args`. Этот аргумент задает максимальное значение аргумента, которое можно добавить в сумму. Таким образом, вызов

`mysum_bigger _than (10, 5, 20, 30, 6)` вернет `50` — потому что `5` и `6` не больше, чем `10`. Эта функция аналогично работает с любым типом и предполагает, что все аргументы имеют одинаковый тип. Обратите внимание, что `>` и `<` работают с различными типами в Python, а не только с числами. Для строк, списков и кортежей это относится к их порядку сортировки.

2. Напишите функцию `sum_numeric`, которая принимает любое количество аргументов. Если аргумент является целым числом или может быть преобразован в целое число, то он должен быть добавлен к сумме. Аргументы, которые не могут быть преобразованы в целые числа, должны быть проигнорированы. Результатом является сумма чисел. Соответственно, `sum_numeric (10, 20, 'a', '30', 'bcd')` вернет `60`. Обратите внимание, что даже если строка `30` является элементом списка, она преобразуется в целое число и добавится к сумме.
3. Напишите функцию, которая принимает список словарей и возвращает один словарь, объединяющий все ключи и значения. Если ключ встречается в более чем одном аргументе, то значением должен быть список, содержащий все значения из аргументов.

Упражнение 11. Упорядочение имен по алфавиту

Предположим, что у вас есть данные телефонного справочника в списке словарей, как показано ниже:

```
PEOPLE = [{'first':'Reuven', 'last':'Lerner',
    'email':'reuven@lerner.co.il'},
    {'first':'Donald', 'last':'Trump',
         'email':'president@whitehouse.gov'},
    {'first':'Vladimir', 'last':'Putin',
         'email':'president@kremvax.ru'}
         ]
```

Прежде всего, если это единственные люди в вашей телефонной книге, то вам следует переосмыслить, действительно ли программирование на Python является лучшей тратой вашего времени и связей.

В любом случае, напишите функцию `alphabetize_names`, которая предполагает существование константы `PEOPLE`, определенной, как показано в коде. Функция должна возвращать список словарей, отсортированных по фамилии и имени.

> **ПРИМЕЧАНИЕ** В Python нет констант, за исключением некоторых внутренних типов и структур данных, каждая переменная, функция и атрибут всегда могут быть изменены. Тем не менее переменные, определенные вне любой функции, обычно называются «константами» и обозначаются ЗАГЛАВНЫМИ БУКВАМИ.


68


69

Вы можете решить это упражнение несколькими способами, но все они потребуют использования метода `sorted`, который вы видели в прошлой главе, вместе с функцией, переданной в качестве аргумента ее `ключевого` (key) параметра. Подробнее о `sorted` и о том, как его использовать, включая пользовательские сортировки с помощью `key`, вы можете прочитать по ссылке [qr68]. Один из вариантов решения этого упражнения предполагает использование `operator.itemgetter`, о котором вы можете прочитать здесь: [qr69].

Обсуждение

Хотя структуры данных Python полезны сами по себе, они становятся еще более мощными и практичными, если их объединить вместе. Списки списков, списки кортежей, списки словарей и словари словарей — все они встречаются довольно часто. Умение работать с этими структурами позволяет уверенно програм-

мировать на Python. Это упражнение показывает, как можно не только хранить данные в таких структурах, но и извлекать их, управлять ими, сортировать и форматировать.

Решение, которое я предлагаю, состоит из двух частей. В первой части мы сортируем наши данные в соответствии с предложенными мною критериями, а именно: сначала фамилия, а затем имя. Во второй части решения рассматривается, как мы будем выводить данные для конечного пользователя.

Рассмотрим сначала вторую задачу. У нас есть список словарей. Это означает, что, когда мы итерационно просматриваем наш список, на каждой итерации `person` присваивается словарь. Словарь имеет три ключа: `first`, `last` и `email`. Мы хотим использовать каждый из этих ключей для отображения каждой записи телефонной книги.

Таким образом, мы можем написать:

```
for person in people:
   print (f'{person ["last"]}, {person ["first"]}:
   {person ["email"]}')
```

Пока все хорошо. Но мы все еще не решили первую проблему, а именно сортировку списка словарей по фамилии, а затем по имени. По сути, мы хотим, чтобы функция сортировки Python не сравнивала словари. Скорее, она должна сравнивать значения `last` и `first` внутри каждого словаря.

Другими словами, мы хотим, чтобы

```
{'first':'Vladimir', 'last':'Putin',
'email':'president@kremvax.ru'}
```

стало

```
['Putin', 'Vladimir']
```

Мы можем сделать это, воспользовавшись `ключевым` параметром `sorted`. Значение, передаваемое этому параметру, должно

быть функцией, принимающей один аргумент. Функция будет вызываться один раз для каждого элемента, а возвращаемое значение функции будет использоваться для сортировки значений.

Таким образом, мы можем отсортировать элементы списка:

```
mylist = ['abcd', 'efg', 'hi', 'j']
mylist = sorted (mylist, key=len)
```

После выполнения этого кода `mylist` теперь будет отсортирован в порядке возрастания длины, поскольку встроенная функция `len` будет применена к каждому элементу перед сравнением с другими. В случае с нашим упражнением по составлению алфавита мы могли бы написать функцию, которая принимает на вход словарь и возвращает нужный список:

```
def person_dict_to_list (d):
return [d ['last'], d ['first']]
```

Затем мы можем применить эту функцию при сортировке нашего списка:

```
print (sorted (people, key=person_dict_to_list))
```

После этого мы могли бы пройтись по уже отсортированному списку и отобразить имена людей.

Но подождите секунду: зачем нам писать специальную функцию (`person_dict_to_list`), которая будет использоваться только один раз? Разумеется, должен быть способ создать временную встраиваемую функцию. И он действительно есть, вы можете использовать `lambda`, которая возвращает новую анонимную функцию. Используя `lambda`, мы получаем следующее решение:

```
for p in sorted (people,
         key=lambda x: [x ['last'], x ['first']]):
   print (f'{p ["last"]}, {p ["first"]}: {p
   ["email"]}')
```

Многие из разработчиков Python, с которыми я говорил на эту тему, не в восторге от использования `lambda`, так как код становится менее читабельным и более запутанным. (Дополнительные соображения по поводу лямбд см. в сноске на странице.)

К счастью, в модуле `operator` есть функция `itemgetter`. `itemgetter` принимает любое количество аргументов и возвращает функцию, которая применяет каждый из этих аргументов в квадратных скобках. Например, если я скажу:

```
s = 'abcdef'
t = (10, 20, 30, 40, 50, 60)

get_2_and_4 = operator.itemgetter (2, 4)
print (get_2_and_4 (s)
print (get_2_and_4 (t))
```

Обратите внимание, что itemgetter возвращает функцию.

Возвращает кортеж ('c', 'e').

Возвращает кортеж (30, 50).

Если мы вызовем `itemgetter ('last', 'first')`, мы получим функцию, которую можно применить к каждому из наших словарей. Она вернет кортеж, содержащий значения, связанные с `last` и `first`.

Другими словами, мы можем просто писать:

```
from operator import itemgetter
for p in sorted (people,
         key=itemgetter ('last', 'first')):
   print (f'{p ["last"]}, {p ["first"]}: {p
   ["email"]}')
```

Решение

```
import operator

PEOPLE = [{'first': 'Reuven', 'last': 'Lerner',
         'email': 'reuven@lerner.co.il'},
         {'first': 'Donald', 'last': 'Trump',
```

```
        'email': 'president@whitehouse.gov'},
        {'first': 'Vladimir', 'last': 'Putin',
        'email': 'president@kremvax.ru'}]
```

Параметр key для sorted получает функцию, результат которой указывает, как мы будем сортировать.

```
def alphabetize_names (list_of_dicts):
    return sorted (list_of_dicts, key=operator.
    itemgetter ('last', 'first'))
print (alphabetize_names (PEOPLE))
```

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr70].

70

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr71].

71

После выполнения упражнения

Для Python–разработчика очень важно уметь работать с сортировкой структур данных Python и особенно с комбинациями встроенных структур данных Python. Недостаточно просто использовать встроенную функцию `sorted`, но понимать, как работает сортировка и как можно использовать `ключевой` параметр, также очень важно. В этом упражнении мы познакомились с этой идеей, но рассмотрим еще несколько возможностей сортировки:

1. Учитывая последовательность положительных и отрицательных чисел, отсортируйте их по абсолютной величине.
2. Задав список строк, отсортируйте их по количеству содержащихся в них гласных.
3. Если дан список списков, каждый из которых содержит ноль или более чисел, отсортируйте его по сумме чисел каждого внутреннего списка.

Что такое лямбда?

Очень часто Python–разработчики спрашивают меня о том, что такое `лямбда`, что она делает и где ее можно использовать.

Ответ заключается в том, что `лямбда` возвращает объект функции, позволяя нам создать анонимную функцию. И мы можем использовать ее везде, где мы могли бы использовать обычную функцию без необходимости использования имени переменной.

Рассмотрим следующий код:

```
glue = '*'
s = 'abc'
print (glue.join (s))
```

Этот код выводит строку `a*b*c`, полученную в результате вызова `glue.join` для `s`. Но зачем вам нужно определять либо `glue`, либо `s`? Разве вы не можете просто использовать строки без каких–либо переменных? Конечно, можете:

```
print ('*'.join ('abc'))
```

Этот код дает тот же результат, что и раньше. Разница в том, что вместо переменных мы используем строковые литералы. Они создаются, когда они нужны, и исчезают после выполнения нашего кода. Можно сказать, что это анонимные строки. Анонимные строки, также известные как строковые литералы, совершенно нормальны и естественны, и мы используем их постоянно.

Теперь рассмотрим следующее: когда мы определяем функцию с помощью `def`, мы фактически делаем две вещи: создаем объект функции и присваиваем этот объект функции переменной. Мы называем эту переменную функцией, но она настолько же функция, насколько `x` — целое после объявления `x=5`. Присвоение в Python всегда означает,

что имя ссылается на объект, а функции — это объекты, как и все остальное в Python. Например, рассмотрим следующий код:

```
mylist = [10, 20, 30]

def hello (name):
    return f'Hello, {name}'
```

Если мы выполним этот код в Python Tutor, то увидим, что мы определили две переменные (рисунок 3.8). Одна (`mylist`) указывает на объект типа `list`. Вторая (`hello`) указывает на объект функции.

Рисунок 3.8. И `mylist`, и `hello` указывают на объекты (из Python Tutor).

Поскольку функции являются объектами, их можно передавать в качестве аргументов другим функциям. Сначала это покажется странным, но вы быстро привыкнете к идее передачи всех объектов, включая функции.

Например, я собираюсь определить функцию (`run_func_with_world`), которая принимает функцию в качестве аргумента. Затем она вызывает эту функцию, передавая ей в качестве аргумента строку `world`:

```
def hello (name):
      return f'Hello, {name}'
```

```
def run_func_with_world (func): return
      func ('world')

print (run_func_with_world (hello))
```

Обратите внимание, что теперь мы передаем `hello` в качестве аргумента функции `run_func_with` (рисунок 3.9). С точки зрения Python, это абсолютно разумно и нормально.

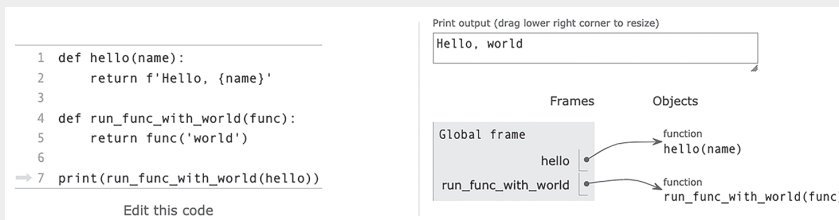


Рисунок 3.9. Вызов `hello` из другой функции (из Python Tutor).

Во многих случаях мы хотим написать функцию, которая принимает другую функцию в качестве аргумента. Одним из таких примеров является `sorted`.

Какое отношение это имеет к `lambda`? Ну, мы всегда можем создать функцию с помощью `def`, но тогда мы создаем новую переменную. И для чего? Чтобы использовать ее один раз? Не обращая внимания на экологические проблемы, вы, вероятно, не захотите покупать металлические вилки, ножи и ложки для случайного пикника: скорее, вы можете просто купить пластиковую посуду. Точно так же, если функция нужна только один раз, то зачем мне определять ее формально и давать ей имя? Именно здесь в дело вступает `lambda` — она позволяет нам создать анонимную функцию, идеально подходящую для передачи другим функциям. Она исчезает, удаляется из памяти, как только в ней отпадает необходимость.

Если мы думаем о `def` как о: (а) создании объекта функции и (б) определении переменной, которая ссылается на этот объект, то мы можем думать о `lambda` как о выполнении только первой из этих двух задач. То есть `lambda` создает и возвращает объект функции. Код, который я написал, вызвав `run_func_with_world` и передав ей `hello` в качестве аргумента, можно переписать с помощью `lambda` следующим образом:

```
def run_func_with_world (f):
      return f ('world')

      print (run_func_with_world (lambda
      name: f'Hello, {name}'))
```

Здесь (рисунок 3.10) я удалил определение функции `hello`, которая делает то же самое, используя `lambda`.

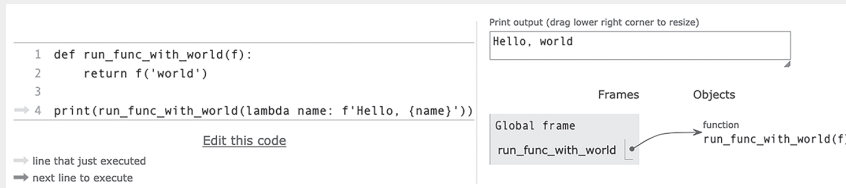


Рисунок 3.10 Вызов анонимной функции из функции (из Python Tutor).

Чтобы создать анонимную функцию с помощью `lambda`, используйте зарезервированное слово `lambda`, а затем перечислите все параметры перед двоеточием. Затем напишите однострочное выражение, которое будет возвращать `lambda`. И действительно, в Python `lambda` ограничена одним выражением — присваивание не допускается, и все должно быть на одной строке.

В настоящее время многие разработчики Python предпочитают не использовать `lambda`, отчасти из-за их ограни-

ченного синтаксиса, а отчасти потому, что доступны более читабельные варианты, такие как `itemgetter`, которые делают то же самое. Я все еще либерален, когда дело касается `lambda`, и люблю использовать их, когда могу, но я также понимаю, что для многих разработчиков это делает код более трудным для чтения и сопровождения. Вы должны сами решить, сколько `lambda` будет в вашем коде.

Упражнение 12. Слово с наибольшим количеством повторяющихся букв

Напишите функцию `most_repeating_word`, которая принимает на вход последовательность строк. Функция должна возвращать строку, содержащую наибольшее количество повторяющихся слов. Другими словами:

1. Для каждого слова найдите букву, которая встречается наибольшее количество раз.
2. Найдите слово, в котором самая повторяющаяся буква встречается чаще, чем любая другая.

То есть, если `words` представляет собой

```
words = ['this', 'is', 'an', 'elementary',
'test', 'example']
```

то ваша функция должна возвращать `elementary`. Это происходит потому, что:

1. `this` не содержит повторяющихся букв.
2. `is` содержит повторяющиеся буквы.
3. `an` не содержит повторяющихся букв.
4. `elementary` содержит одну повторяющуюся букву — `e`, которая появляется три раза.
5. `test` содержит одну повторяющуюся букву — `t`, которая появляется дважды.

6. `example` содержит одну повторяющуюся букву — `e`, которая появляется дважды.

Таким образом, самая распространенная буква в `elementary` встречается чаще, чем самые распространенные буквы в любом из других слов. (Если это ничья, то можно вернуть любое из подходящих слов.)

Скорее всего, вы захотите использовать `Counter` из модуля `collections`, который подходит для подсчета количества элементов в последовательности. Более подробную информацию можно найти здесь: [qr72]. Обратите особое внимание на метод `most_common` [qr73], использование которого здесь будет актуальнее.

72

73

Обсуждение

Это решение объединяет несколько моих любимых техник Python в коротком фрагменте кода:

1. `Counter`, подкласс `dict`, определенный в модуле `collections`, который позволяет легко подсчитывать предметы.
2. Передача функции в качестве `ключевого` параметра в `max`.

Чтобы наше решение работало, нам нужно найти способ определить, сколько раз каждая буква встречается в слове. Самый простой способ сделать это — `Counter`. Это правда, что `Counter` наследуется от `dict` и поэтому может делать все, что может делать `dict`. Но обычно мы создаем экземпляр `Counter`, инициализируя его последовательностью, например:

```
>>> Counter ('abcabcabbbc')
Counter ({'a': 3, 'b': 5, 'c': 3})
```

Таким образом, мы можем передать `Counter` слово, и он скажет нам, сколько раз каждая буква встречается в этом слове. Ко-

нечно, мы можем использовать `Counter` и узнать, какая буква встречается чаще всего. Но зачем так напрягаться, если можно вызвать `Counter.most_common`?

```
>>> Counter('abcabcabbbc').most_common()
[('b', 5), ('a', 3), ('c', 3)]
```

Показывает, как часто каждый элемент встречается в строке, от наиболее распространенного к наименее распространенному, в виде списка кортежей.

Результатом вызова `Counter.most_common` является список кортежей с именами и значениями счетчика в порядке убывания. В примере `Counter.most_common` мы видим, что `b` встречается пять раз, `a` и `c` встречаются по три раза. Если бы мы вызвали `most_common` с целочисленным аргументом `n`, мы бы увидели только `n` наиболее часто встречающихся элементов:

```
>>> Counter ('abcabcabbbc').most_common (1)
[('b', 5)]
```

Показывает только самый распространенный элемент и сколько раз он встречается.

Это идеально подходит для наших целей. Действительно, я думаю, что было бы полезно обернуть это в функцию, которая будет возвращать, сколько раз наиболее повторяющаяся буква встречается в слове:

```
def most_repeating_letter_count (word):
    return Counter (word).most_common (1) [0] [1]
```

`(1) [0] [1]` в конце выглядит немного запутанным. Это означает следующее:

1. Нам нужна только наиболее повторяющаяся буква, возвращаемая в виде одноэлементного списка кортежей.
2. Затем нам нужен первый элемент из этого списка, кортеж.
3. Затем нам нужен счетчик для наиболее повторяющегося элемента с индексом 1 в кортеже.

Помните, что нам неважно, какая именно буква повторяется. Нам просто важно, как часто встречается наиболее повторяемая буква. И да, мне тоже не нравятся множественные индексы в конце вызова функции, отчасти поэтому я хочу «завернуть» это в функцию. Мы можем вызвать `most_common` с аргументом `1`, чтобы сказать, что нас интересует только буква с наибольшим количеством баллов, затем, что нас интересует первый (и единственный) элемент этого списка, а затем, что нам нужен второй элемент (т.е. счетчик) из кортежа.

Чтобы найти слово с наибольшим количеством совпадающих букв, мы захотим применить `most_repeating_letter_count` к каждому элементу `WORDS`, указывая, какой из них имеет наибольший результат. Один из способов сделать это — воспользоваться `sorted`, используя `most_repeating_letter_count` в качестве `ключевой` функции. То есть мы будем сортировать элементы `WORDS` по количеству повторяющихся букв. Поскольку `sorted` возвращает список, отсортированный от наименьшего к наибольшему значению, последний элемент (т.е. с индексом –1) будет самым повторяющимся словом.

Но мы можем сделать еще лучше: встроенная функция `max` будет принимать `ключевую` функцию, как и `sorted`, и возвращать элемент, получивший наибольшую оценку. Следовательно, мы можем сэкономить немного времени на кодировании, используя однострочную версию `most_repeating_word`:

```
def most_repeating_word (words):
    return max (words,
          key=most_repeating_letter_count)
```

Решение

```
from collections import Counter
import operator

WORDS = ['this', 'is', 'an',
         'elementary', 'test', 'example']
```

Какая буква встречается чаще всего и сколько раз?

```
def most_repeating_letter_count (word):
    return Counter (word).most_common (1) [0] [1]
```

Counter.most_common возвращает список двухэлементных кортежей (value и count) в порядке убывания.

```
def most_repeating_word (words):
    return max (words,
                key-most_repeating_letter_count (1) {0} {1}
```

Точно так же, как вы можете передать ключ в sorted, вы можете передать его в max и использовать другой метод сортировки.

```
print (most_repeating_word (WORDS))
```

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr74].

74

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr75].

75

После выполнения упражнения

Сортировка, управление сложными структурами данных и передача функций другим функциям — все это объемные темы, заслуживающие вашего внимания и практики. Вот несколько вещей, которые вы можете сделать, чтобы выйти за рамки этого упражнения и изучить эти темы еще раз:

1. Вместо того, чтобы искать слово с наибольшим количеством повторяющихся букв, найдите слово с наибольшим количеством повторяющихся гласных.
2. Напишите программу для чтения `/etc/passwd` на компьютере Unix. Первое поле содержит имя пользователя, а последнее — *оболочку пользователя*, командный интерпретатор. Выведите оболочки в порядке убывания популярности

так, чтобы самая популярная оболочка была показана первой, вторая по популярности — второй, и так далее.

3. Для дополнительной сложности после отображения каждой оболочки также покажите имена пользователей (отсортированные по алфавиту), которые используют каждую из этих оболочек.

Упражнение 13. Печать записей кортежей

Кортежи обычно используются в качестве записей, подобно структурам в некоторых других языках. И, конечно, отображение этих записей в таблице — стандартная задача программ. В этом упражнении мы сделаем и то, и другое: прочитаем список кортежей и превратим их в форматированный вывод для пользователя.

Например, предположим, что мы отвечаем за проведение международного саммита в Лондоне. Мы знаем, за сколько часов каждый мировой лидер прибудет на мероприятие:

```
PEOPLE = [('Donald', 'Trump', 7.85),
    ('Vladimir', 'Putin', 3.626),
    ('Jinping', 'Xi', 10.603)]
```

План саммита должен содержать список мировых лидеров, которые примут участие, а также время, которое они потратят на дорогу. Однако в плане поездки не нужно указывать степень точности, которую обеспечивает компьютер, нам достаточно будет двух цифр после запятой.

Для этого упражнения напишите функцию Python, format_sort_records, принимающую на вход список PEOPLE и возвращающую отформатированную строку, которая выглядит следующим образом:

```
Trump → Donald   → 7.85
Putin → Vladimir → 3.63
Xi    → Jinping  → 10.60
```

Обратите внимание, что фамилия печатается перед именем (с учетом того, что китайские имена обычно отображаются именно так), а затем следует указание с десятичным выравниванием сколько времени потребуется каждому лидеру, чтобы прибыть в Лондон. Каждое имя должно быть напечатано в 10–символьном поле, а время — в 5–символьном поле, с одним пробелом между столбцами. Время в пути должно отображаться числом с двумя цифрами после десятичной точки, что означает, что даже если входные данные для рейса Си Цзиньпина составляют 10.603 часа, то отображаемое значение должно быть `10.60`.

Обсуждение

Кортежи часто используются в контексте структурированных данных и записей базы данных. В частности, вы можете ожидать получить кортеж при получении одной или нескольких записей из реляционной базы данных. Затем вам нужно будет получить отдельные поля с помощью числовых индексов.

Это упражнение состояло из нескольких частей. Прежде всего нам нужно было отсортировать людей в алфавитном порядке по фамилии и имени. Я использовал встроенную функцию `sorted` для сортировки кортежей при помощи алгоритма, аналогичный тому, который мы использовали со списком словарей в предыдущем упражнении. Таким образом, цикл `for` итерировал каждый элемент нашего отсортированного списка, получая кортеж (который он назвал `person`) на каждой итерации. Вы можете часто вспоминать о словаре как о списке кортежей, особенно когда дело касается итерирования при помощи метода `items` (рисунок 3.11).

Затем необходимо вывести содержимое кортежа в строгом формате. Хотя часто лучше использовать f–строки, `str.format` все же может быть полезен в некоторых обстоятельствах. Здесь я использую тот факт, что `person` — это кортеж, а `*person` при передаче в функцию становится не кортежем, а элементами этого кортежа. Это означает, что мы передаем три отдельных аргумента в `str.format`, доступ к которым мы можем получить через `{0}`, `{1}` и `{2}`.

```
import operator
PEOPLE = [('Donald', 'Trump', 7.85),
          ('Vladimir', 'Putin', 3.626),
          ('Jinping', 'Xi', 10.603)]

def format_sort_records(list_of_tuples):
    output = []
    template = "{1:10} {0:10} {2:5.2f}"
    for person in sorted(list_of_tuples,
            key=operator.itemgetter(1, 0)):

        output.append(template.format(*person))
    return output

print('\n'.join(format_sort_records(PEOPLE)))
```

```
Putin      Vladimir    3.63
Trump      Donald      7.85
Xi         Jinping    10.60
```

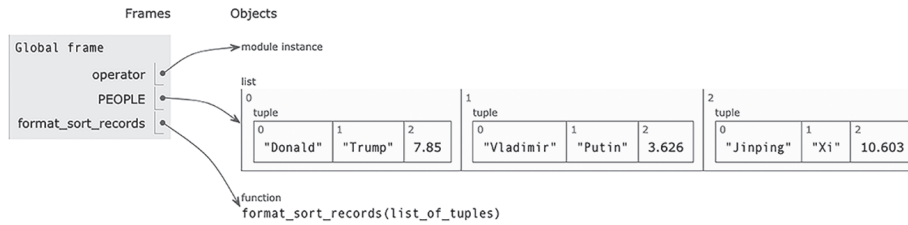


Рисунок 3.11. Итерация над списком кортежей (из Python Tutor).

В случае с фамилией и именем мы хотели использовать 10–символьное поле, заполненное пробелами. Мы можем сделать это в `str.format`, добавив символ двоеточия (:) после индекса, который мы хотим отобразить. Таким образом, `{1:10}` указывает Python отобразить элемент с индексом 1, добавляя пробелы, если данные содержат менее 10 символов. Строки по умолчанию выравниваются влево, поэтому имена будут отображаться по левому полю в своих столбцах.

Третий столбец немного сложнее, поскольку мы хотели отобразить только две цифры после запятой, максимум пять символов, чтобы выровнять десятичную запятую времени в пути и (как будто этого было недостаточно) заполнить столбец символами пробела.

В `str.format` (и в f–строках) каждый тип обрабатывается по–разному. Так, если мы просто укажем `{2:10}` в качестве параметра форматирования для наших чисел с плавающей точкой (например, `person [2]`), число будет выровнено

по правому краю. Мы можем отобразить его как число с плавающей точкой, если поставим в конце символ `f`, как в `{2:10f}`, но в этом случае после десятичной точки будут стоять нули. Спецификатор для вывода двух цифр после десятичной точки, максимум пяти цифр в сумме, будет `{5.2f}`, который выводит то, что мы хотели.

Решение

```
import operator
PEOPLE = [('Donald', 'Trump', 7.85),
          ('Vladimir', 'Putin', 3.626),
          ('Jinping', 'Xi', 10.603)]
def format_sort_records (list_of_tuples):
    output = []
    template = '{1:10} {0:10} {2:5.2f}'
    for person in sorted (list_of_tuples,
            → key=operator.itemgetter (1, 0)):
```

Вы можете использовать operator.itemgetter с любой структурой данных, которая принимает квадратные скобки. Вы также можете передать ему более одного аргумента, как показано здесь.

```
        output.append (template.format (*person))
    return output

print ('\n'.join (format_sort_records (PEOPLE)))
```

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr76].


76

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr77].


77

После выполнения упражнения

Вот несколько идей для дополнительных заданий к упражнению, которые помогут узнать больше о подобных структурах данных:

1. Если кортежи раздражают вас тем, что в них используются числовые индексы, вы не одиноки! Реализуйте это упражнение, используя объекты `namedtuple`, определенные в модуле `collections`. Многим нравится использовать именованные кортежи, потому что они обеспечивают правильный баланс между читабельностью и эффективностью.
2. Определите список кортежей, в котором каждый кортеж содержит название, продолжительность (в минутах) и режиссера фильмов, номинированных на премию «Оскар» за лучшую картину в прошлом году. Спросите пользователя, хочет ли он отсортировать список по названию, длине или имени режиссера, а затем представьте список, отсортированный по выбранному пользователем параметру.
3. Расширьте это упражнение, позволив пользователю сортировать не по одному, а по двум или трем из этих полей. Пользователь может указать поля, введя их через запятую. Вы можете использовать `str.split`, чтобы превратить их в список.

Подводя итоги

В этой главе мы рассмотрели несколько способов использования списков и кортежей и управления ими в программах на Python. Трудно преувеличить, насколько распространены списки и кортежи и насколько хорошо вы должны быть с ними знакомы. Вкратце, вот некоторые из наиболее важных моментов, которые следует помнить о них:

1. Списки являются изменяемыми, а кортежи — неизменяемыми, но реальная разница между ними заключается в том, как они используются: списки предназначены для последовательностей одного типа, а кортежи — для записей, содержащих разные типы.
2. Вы можете использовать встроенную функцию `sorted`

для сортировки списков или кортежей. В результате вызова `sorted` вы получите список.

3. Вы можете изменить порядок сортировки, передав функцию в `ключевом` параметре. Эта функция будет вызвана один раз для каждого элемента в последовательности, и вывод функции будет использован при упорядочивании элементов.
4. Если вы хотите подсчитать количество элементов, содержащихся в последовательности, попробуйте использовать класс `Counter` из модуля `collections`. Он не только позволяет быстро и легко подсчитывать количество элементов и предоставляет нам метод `most_common`, но и наследует его от `dict`, предоставляя нам все функции словарей, которые мы знаем и любим.

4. Словари и множества

Словари являются одними из самых мощных и важных структур данных в Python. Вы можете встретить их в других языках программирования, в которых они известны как «хэши», «ассоциативные массивы», «хэш–карты» или «хэ–ш–таблицы».

В словаре мы не вводим отдельные элементы, как в списке или кортеже. Скорее, мы вводим пары данных, где первый элемент известен как ключ, а второй — как *значение*. В то время как индекс в строке, списке или кортеже всегда является целым числом и всегда начинается с 0, ключи словаря могут быть самых разных типов Python — как правило, целые числа или строки.

Это, казалось бы, небольшое различие — то, что мы можем использовать произвольные ключи для поиска наших значений, а не целочисленные индексы, — на самом деле имеет решающее значение. Во многих задачах программирования используются пары имя–значение — например, имена пользователей/идентификаторы пользователей, IP–адреса/имена хостов, адреса электронной почты/зашифрованные пароли. Более того, большая часть самого языка Python реализована с использованием словарей. Поэтому знание того, как работают словари и как их лучше использовать, даст вам представление о фактической реализации Python.

Я использую словари тремя основными способами:

1. *В качестве небольших баз данных или записей.* Часто удобно использовать словари для хранения пар имя–значение. Мы

можем загрузить файл конфигурации в Python в виде словаря, получая значения, связанные с опциями конфигурации. Мы можем хранить информацию о файле, или предпочтениях пользователя, или множество других вещей со стандартными именами и неизвестными значениями. При таком использовании вы определяете словарь один раз, часто в начале программы, и он не изменяется.

2. *Для хранения тесно связанных имен и значений.* Вместо того чтобы создавать несколько отдельных переменных, вы можете создать словарь с несколькими парами ключ–значение. Я делаю так, когда хочу сохранить (например) несколько частей информации о веб–сайте, таких как его URL, мое имя пользователя и последняя дата посещения. Конечно, можно использовать несколько переменных для отслеживания этой информации, но словарь позволяет проще управлять ею — а также передавать ее в функцию или метод сразу, через одну переменную.
3. *Для накопления информации с течением времени.* Если вы отслеживаете, какие ошибки произошли в вашей программе и сколько раз произошла каждая ошибка, то словарь отлично подойдет для этой задачи. Вы также можете использовать один из классов, наследующих от `dict`, например, `Counter` или `defaultdict`, которые определены в модуле `collections`. При таком использовании словарь растет со временем, добавляя новые пары ключ–значение и обновляя значения по мере выполнения программы.

Несомненно, вы найдете другие способы использования словарей в своих программах, но я перечислил три наиболее часто встречающихся в моей работе.

Хэширование и словари

Из того, что я написал до сих пор, может показаться, что любой объект Python может быть использован в качестве ключа или значения в словаре. Но это не так. Хотя в Python в значениях можно хранить абсолютно все, в качестве ключей можно

использовать только хэшируемые типы, то есть те, к которым можно применить хэш–функцию. Эта же `хэш-функция` гарантирует, что ключи словаря уникальны, а поиск ключа может быть достаточно быстрым.

Что такое хэш–функция? Зачем она в Python? И как она влияет на то, что мы делаем?

Основная идея заключается в следующем. Предположим, что у вас есть здание с 26 офисами. Если посетитель приходит, чтобы встретиться с г–жой Смит, как ему узнать, где ее найти? Без секретаря или офисного справочника посетителю придется пройти через все офисы один за другим в поисках кабинета г–жи Смит.

Именно таким образом происходит поиск в строке, списке или кортеже в Python. Время, необходимое для поиска значения в такой последовательности, описывается в литературе по информатике как `O (n)`. Это означает, что по мере увеличения длины последовательности поиск искомого значения занимает пропорционально больше времени.

Теперь давайте представим себе нашу офисную среду. Здесь по–прежнему нет ни справочника, ни приемной, но есть табличка, гласящая, что если вы ищете сотрудника, то просто зайдите в офис, номер которого совпадает с первой буквой его фамилии — по схеме a=1, b=2, c=3 и так далее.

Поскольку посетитель хочет найти мисс Смит, он вычисляет, что S — это 19–я буква английского алфавита, идет в комнату 19 и с радостью обнаруживает, что она там. Если бы посетитель искал мистера Джонса, он, конечно, пошел бы в комнату 10, поскольку J — 10–я буква алфавита.

Такой поиск, как видите, не требует много времени. Действительно неважно, сколько сотрудников в нашей компании — 2, 25 или даже 250 — если компания существенно вырастет, посетители смогут найти офисы наших сотрудников за то же время. В мире программирования это известно как `O (1)`, или *постоянное время*, и его довольно трудно переоценить.

Конечно, есть одна загвоздка: что, если у нас есть два человека, чьи фамилии начинаются на S? Мы можем решить эту проблему

несколькими разными способами. Например, мы можем использовать первые две буквы фамилии или попросить всех людей, чьи фамилии начинаются на S, работать в одном офисе. Тогда нам придется перебрать всех людей в данном офисе, что, как правило, не так уж страшно.

Описание, которое я вам здесь дал, является упрощенной версией хэш–функции. Такие функции используются в самых разных случаях в мире программирования. Например, они особенно популярны в криптографии и компьютерной безопасности, потому что, хотя их отображение входов на выходы детерминировано, его практически невозможно вычислить без использования самой хэш–функции. Они также играют ключевую роль в работе со словарями в Python.

Запись в словарь состоит из пары ключ–значение. Ключ передается хэш–функции в Python, которая возвращает адрес, в котором хранится пара ключ–значение. Так, если вы скажете `d ['a'] = 1`, Python выполнит `hash ('a')` и использует результат для хранения пары ключ–значение. А когда вы запросите значение `d ['a']`, Python может вызвать `hash ('a')` и немедленно проверить в указанном слоте памяти, есть ли там пара ключ–значение. В мире Python словари называются *отображениями*, потому что хэш–функция отображает наш ключ в целое число, которое мы затем можем использовать для хранения пар ключ–значение.

Я опускаю здесь ряд деталей, включая важные внутренние изменения, которые произошли в Python 3.6. Данные изменения гарантировали, что пары ключ–значения будут храниться (и извлекаться) в хронологическом порядке, при этом использование памяти сократится на одну треть. Данная мысленная модель должна помочь объяснить, как словари достигают времени поиска `O (1)` (постоянное время), независимо от количества добавленных пар ключ–значение и почему они используются не только разработчиками Python, но и самим языком. Вы можете узнать больше о новой реализации в потрясающем докладе Рэй–Монда Хеттингера, перейдя по ссылке [qr78].

78

Хэш–функция объясняет, почему словари в Python

1. всегда хранят пары ключ–значение вместе
2. гарантируют очень быстрый поиск ключей
3. обеспечивают уникальность ключей
4. не гарантируют ничего в отношении поиска значений

Что касается того, почему списки и другие изменяемые встроенные типы считаются в Python «нехэшируемыми», то причина проста: если ключ изменится, то изменится и результат выполнения `хэша` над ним. Это означает, что пара ключ–значение может находиться в словаре, но быть ненайденной. Чтобы избежать таких проблем, Python гарантирует, что наши ключи не могут меняться. Термины *хэшируемость* и *неизменяемость* — не одно и то же, но они во многом совпадают, и когда вы только начинаете знакомиться с языком, не стоит сильно беспокоиться о различиях.

Множества

Со словарями тесно связаны множества, которые можно представить как словари без значений. (Я часто шучу, что это означает, что множества на самом деле являются аморальными словарями.) Множества чрезвычайно полезны, когда вам нужно найти что–то в большой коллекции, например, имена файлов, адреса электронной почты или почтовые индексы, потому что поиск выполняется за `O (1)`, как и в словаре. Я также все чаще стал использовать множества для удаления дублирующихся значений из входного списка — например, IP–адресов в файле журнала или номерных знаков автомобилей, проехавших через въезд на парковку за определенный день.

В этой главе вы будете использовать словари и множества различными способами для решения задач. Можно с уверенностью сказать, что почти каждая программа на Python использует словари, или, возможно, альтернативные словари, такие как `defaultdict` из модуля `collections`.

Таблица 4.1. Что вам нужно знать

Понятие	Что это?	Пример	Чтобы узнать подробнее
`input`	Предлагает пользователю ввести строку и возвращает строку.	`input ('Введите ваше имя: ')`	
`dict`	Тип dict в Python для хранения пар ключ–значение. dict также можно использовать для создания нового словаря.	`d = {'a':1, 'b':2}` или `d = dict'a', 1), ('b', 2`	
`d [k]`	Извлекает значение, связанное с ключом k в словарь d.	`x = d [k]`	
`dict.get`	Аналогично d [k] за исключением того, что возвращает None (или второй необязательный аргумент), если k отсутствует в d.	`x = d.get (k)` или `x =d.get (k, 10)`	
`dict. items`	Возвращает итератор, который на каждой итерации возвращает пару ключ–значение (в виде кортежа).	`for key, value ind. items ():`	
`set`	Тип set в Python используется для хранения уникальных, хэшируемых элементов. set также можно использовать для создания нового множества.	`s = {1,2,3} # создает 3–элемент- ное множе- ство`	
`set.add`	Добавляет один элемент во множество.	`s.add (10)`	

Окончание таблицы

Понятие	Что это?	Пример	Чтобы узнать подробнее
`set.update`	Добавляет элементы одной или нескольких итерируемых объектов в множество.	`s.update([10, 20, 30, 40, 50])`	
`str.isdigit`	Возвращает True, если все символы в строке являются цифрами 0–9.	`'12345'.isdigit() # Возвращает True`	

Упражнение 14. Ресторан

Одно из распространенных применений словарей — это небольшая база данных в нашей программе. Мы определяем словарь в верхней части программы, а затем ссылаемся на нее во всей программе.

Например, вы можете создать словарь месяцев, в котором названия месяцев будут ключами, а числа — значениями. Или, возможно, у вас будет словарь пользователей, где в качестве ключей будут идентификаторы пользователей, а в качестве значений — адреса электронной почты.

В этом упражнении я хочу, чтобы вы создали новый константный словарь под названием MENU, содержащий блюда, которые вы можете заказать в ресторане. Ключами будут строки, а значениями — цены (т.е. целые числа). Затем необходимо написать функцию `restaurant`, которая попросит пользователя ввести заказ:

1. Если пользователь вводит название блюда в меню, программа печатает цену и итоговую сумму. Затем она снова спрашивает пользователя о заказе.
2. Если пользователь вводит название блюда, которого нет в меню, программа ругает пользователя (мягко). Затем она снова спрашивает пользователя о заказе.

3. Если пользователь вводит пустую строку, программа прекращает запрос и печатает общую сумму.

Например, сеанс работы с пользователем может выглядеть следующим образом:

```
Заказ: сэндвич
сэндвич стоит 10, общая сумма 10 Заказ: чай
чай стоит 7, итого 17
Заказ: слон
Извините, но сегодня у нас нет слона.
Заказ: <enter>
Ваша сумма равна 17
```

Обратите внимание, что вы всегда можете проверить наличие ключа в словаре с помощью оператора `in`. Он возвращает `True` или `False`.

Обсуждение

В этом упражнении словарь определяется один раз и остается неизменным на протяжении всей программы. Конечно, мы могли бы использовать список списков или даже список кортежей, но, когда у нас есть пары имя–значение, для нас более естественно поместить их в словарь, а затем извлекать элементы из словаря через ключи.

Итак, что происходит в этой программе? Во–первых, мы создаем наш словарь (`menu`) с его ключами и значениями. Мы также задаем общую сумму, к которой будем суммировать в дальнейшем. Затем мы просим пользователя ввести строку. Мы вызываем функцию `strip` для строки пользователя, так что, если он введет кучу символов пробела (но больше ничего), мы будем считать это пустой строкой.

Если пользователь вводит пустую строку, мы выходим из цикла. Как обычно, мы проверяем пустую строку не с помощью явного `if order == ''`, и даже не с помощью проверки `len (order) == 0`, а с помощью `if not order`, как это принято в Python.

Но если пользователь ввел строку, то мы будем искать ее в словаре. Оператор `in` проверяет, существует ли там строка, если да, то мы можем получить цену и добавить ее к общей.

Если `order` не пустой, но это не ключ в `menu`, мы сообщаем пользователю, что товара нет на складе.

С одной стороны, такое использование словарей не является очень продвинутым или сложным для понимания. С другой стороны, оно позволяет нам работать с нашими данными достаточно простым способом, используя преимущества быстрого поиска, который обеспечивают словари, и используя связанные данные в наших программах.

Решение

Определяет константу словаря с названиями позиций меню (строки) и ценами (целые числа).

```
MENU = {'сэндвич': 10, 'чай': 7, 'салат': 9}
```

Продолжает запрашивать ввод данных у пользователя до явного «прерывания» цикла.

```
def restaurant ():
    total = 0
    while True:
        order = input ('Заказ: ').strip ()
```

Получает входные данные пользователя и использует str.strip для удаления пробелов в начале и конце.

```
        if not order:
            break
```

Если order (заказ) — пустая строка, выходим из цикла.

Если блюдо из заказа есть в меню, то получаем его цену и добавляем к общей сумме.

```
        if order in MENU:
            price = MENU [order]
            total += price
            print (f'{order} стоит {price},
            общая сумма {total}')
```

```
        else:
            print (f'Сегодня у нас только что
            закончился {order}')

        print (f'Общая стоимость {total}')

restaurant ()
```

Если order не пуст и не содержится в словаре, то мы не принимаем этот заказ.

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr88].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr89].

После выполнения упражнения

Поначалу может показаться странным рассматривать хранилище ключевых значений (например, словарь) как базу данных. Но оказывается, что существует множество примеров того, где и как можно использовать такую структуру данных. Вот несколько дополнительных заданий, которые вы можете выполнить для улучшения своих навыков в этой области:

1. Создайте словарь, в котором ключами являются имена пользователей, а значениями — пароли, представленные в виде строк. Создайте крошечную систему входа, в которой пользователь должен ввести имя пользователя и пароль. Если они совпадают, то укажите, что пользователь успешно вошел в систему. Если нет, то откажите ему во входе. (Примечание: это хорошее небольшое упражнение, но, пожалуйста, никогда не храните пароли в незашифрованном виде. Это серьезный риск для безопасности.)
2. Задайте словарь, ключами которого будут являться даты (представленные строками) за последнюю неделю, а значениями — температура. Попросите пользователя ввести дату

и выведите температуру для этой даты, а также в предыдущую и последующую даты, если таковые имеются.

3. Определите словарь, ключами которого являются имена людей в вашей семье, а значениями — даты их рождения, представленные в Python `date` объектами. Попросите пользователя ввести имя человека из вашей семьи и попросите программу вычислить, сколько дней ему исполнилось.

Упражнение 15. Дождевые осадки

Еще одно применение словарей — накопление данных в течение жизни программы. В этом упражнении вы будете использовать словарь именно для этого.

В частности, напишите функцию `get_rainfall`, которая отслеживает количество дождевых осадков в ряде городов. Пользователи вашей программы будут вводить название города, если название города пустое, то функция распечатает отчет (который я опишу) перед выходом.

Если название города не пустое, то программа также должна спросить пользователя, сколько осадков выпало в этом городе (обычно измеряется в миллиметрах). После того как пользователь введет количество осадков, программа снова спросит название города, количество осадков и так далее — до тех пор, пока пользователь не нажмет Enter вместо того, чтобы ввести название города.

Когда пользователь вводит пустое название города, программа завершает свою работу, но сначала она сообщает, сколько всего осадков выпало в каждом городе. Таким образом, если я введу

```
Бостон
5
Нью — Йорк
```

```
7
Бостон
5
[Enter; пустая строка]
```

программа должна вывести

```
Бостон: 10
Нью-Йорк: 7
```

Порядок, в котором появляются города, неважен, и города неизвестны программе заранее.

Обсуждение

В этой программе словари используется классическим образом как крошечная база данных имен и значений, которая растет по ходу работы программы. В случае с этой программой мы используем словарь с `количеством дождевых осадоков`, чтобы отслеживать города и количество осадков, выпавших в них на сегодняшний день.

Мы используем бесконечный цикл, который легче всего реализовать в Python с помощью `while True`. Только когда программа дойдет до `break`, она выйдет из цикла.

В начале каждого цикла мы получаем название города, для которого пользователь сообщает об осадках. Как мы уже видели, программисты Python обычно не проверяют, пуста ли строка, проверяя ее длину. Скорее, они проверяют, содержит ли строка булевое значение `True` или `False`. Если строка пуста, то она примет значение `False` в операторе `if`. Наше выражение `if not city_name` означает: «если переменная `city_name` содержит значение `False`», или, если говорить более простым языком: «если `city_name` пустая».

Давайте разберем работу программы с помощью примеров, приведенных ранее в этом разделе, и посмотрим, как она функционирует. Когда пользователя просят ввести данные в первый

раз, перед ним появляется командная строка (рисунок 4.1). Словарь `rainfall` уже определен, и мы хотим заполнить его парой ключ–значение.

```
1  def get_rainfall():
2      rainfall = {}
3      while True:
4          city_name = input('Enter city name: ')
5          if not city_name:
6              break
7
8          mm_rain = input('Enter mm rain: ')
9          rainfall[city_name] = rainfall.get(city_name,
10                         0) + int(mm_rain)
11
12     for city, rain in rainfall.items():
13         print(f'{city}: {rain}')
14
15 get_rainfall()
```

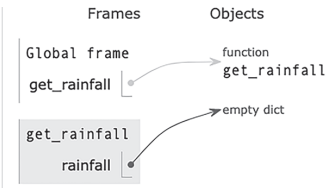


Рисунок 4.1. Запрашиваем у пользователя первый ввод.

После того как мы ввели название города (`Бостон`), мы вводим количество выпавших осадков (`5`). Поскольку Бостон впервые указан как город, мы добавляем новую пару ключ–значение для `rainfall`. Для этого мы задаем ключ `Boston` и значение `5` нашему словарю (рисунок 4.2).

Обратите внимание, что этот код использует `dict.get` по умолчанию, чтобы получить либо текущее значение, связанное с `Бостоном` (если оно есть), либо `0` (если его нет). В первый раз, когда мы спрашиваем о городе, нет ключа с именем `Бостон`, и уж тем более нет информации о предыдущих осадках.

Это упражнение содержит две части, которые зачастую удивляют, либо расстраивают новичков в Python. Первая заключается в том, что функция `input` возвращает строку. Это хорошо, когда пользователь вводит город, но не так хорошо, когда пользователь вводит количество выпавших осадков. Хранение количества осадков в виде строки работает относительно хорошо, если город вводится только один раз. Однако если город вводится более одного раза, то программа столкнется с необходимостью сложить (с помощью оператора `+`) две строки вместе.

Python с удовольствием сделает это, но результатом будет новая конкатенированная строка, а не значение сложенных целых чисел.

Рисунок 4.2. После добавления пары ключ-значение в словарь.

По этой причине мы вызываем `int` для `mm_rain`, чтобы получить целое число. Если вы хотите, вы можете заменить `int` на `float`, и таким образом получить обратно значение с плавающей точкой. Тем не менее, если вы используете `input` для получения ввода от пользователя и если вы хотите использовать числовое значение, а не строку, вы должны выполнить преобразование.

Перехват ошибок при вводе

В моем решении намеренно не проверяется, может ли пользовательский ввод быть преобразован в целое число. Это означает, что если пользователь введет строку, содержащую не цифры от 0 до 9, то вызов `int` вернет ошибку. Я не хотел слишком усложнять код решения.

Если вы хотите перехватывать такие ошибки, то у вас есть два основных варианта. Первый — обернуть вызов `int` внутри блока `try`. Если вызов int завершится ошибкой, вы можете поймать исключение, например:

```
try:
      mm_rain = int (input ('Введите коли-
      чество осадков в мм: '))
except ValueError:
      print ('Вы ввели недопустимое целое
      число; попробуйте еще раз.')
      continue

rainfall [city_name] = rainfall.get (city_
name, 0) + mm_rain
```

В этом коде мы позволяем пользователю вводить все, что он захочет. Если мы сталкиваемся с ошибкой (исключением) при преобразовании, мы отправляем пользователя обратно в начало нашего цикла `while`, когда мы запрашиваем название города. В более сложной реализации пользователь просто повторно вводит значение `mm_rain`.

Второе решение — использовать метод `str.isdigit`, который возвращает `True`, если строка содержит только цифры 0–9, и `False` в противном случае, например:

```
mm_rain = input ('Введите количество осадков
в мм: ').strip ()
if mm_rain.isdigit ():
      mm_rain = int (mm_rain)
else:
      print ('Вы ввели недопустимое целое
      число; попробуйте еще раз.')
      continue
```

И снова это вернет пользователя к началу цикла `while`, попросив его еще раз ввести название города. Также предполагается, что нас интересуют только целые значения, потому что `str.isdigit` возвращает `False`, если вы зададите ему число с плавающей запятой.

Вы могли заметить, что у строк Python есть три метода с похожими названиями: `isdigit`, `isdecimal` и `isnumeric`. В большинстве случаев эти три метода взаимозаменяемы.

Однако вы можете узнать больше о том, чем они отличаются друг от друга, по ссылке [qr90]

Вторая сложная часть этого упражнения заключается в том, что вы должны обработать случай, когда город был назван первый раз (т.е. до того, как название города стало ключом в `rainfall`), а также последующие разы.

В первый раз, когда кто-то вводит `Бостон` в качестве названия города, нам нужно будет добавить в наш словарь пару ключ–значение для этого города и количества осадков. Во второй раз, когда кто-то введет `Бостон` в качестве названия города, нам нужно будет добавить новое значение к уже существующему.

Одним из простых решений этой проблемы является использование метода `dict.get` с двумя аргументами. При одном аргументе `dict.get` либо возвращает значение, связанное с именованным ключом, либо `None`. Но с двумя аргументами `dict.get` возвращает либо значение, связанное с ключом, либо второй аргумент (рисунок 4.3).

Рисунок 4.3. Добавляем к существующей паре имя-значение.

Таким образом, когда мы вызываем `rainfall.get (city_name, 0)`, Python проверяет, существует ли уже ключ `city_name` в `rainfall`. Если да, то вызов `rainfall.get` вернет значение, связанное с этим ключом. Если `city_name` отсутствует в `rainfall`, то мы получим `0`.

В качестве альтернативного решения можно использовать `defaultdict`, класс, определенный в модуле `collections`, который позволяет вам определить словарь, работающий так же, как и обычный — до тех пор, пока вы не запросите у него несуществующий ключ. В таких случаях `defaultdict` вызывает функцию, с помощью которой он был определен, например:

```
from collections import defaultdict
rainfall = defaultdict (int)
rainfall ['Boston'] += 30
rainfall → # defaultdict (<type 'int'>,
{'Boston': 30})
```

defaultdict(int) означает, что, если мы зададим rainfall[k] и k не входит в rainfall, функция int выполнится без аргументов, вернув нам int 0.

```
rainfall ['Boston'] += 30
rainfall → # defaultdict (<type 'int'>, {'Boston':
60})
```

Решение

Мы не знаем, какие города введет пользователь, поэтому мы создаем пустой словарь, готовый к заполнению.

```
def get_rainfall ():
    rainfall = {}

while True:
    city_name = input ('Введите название города: ')
    if not city_name:
        break
```

Если вы живете в США, то, возможно, вы удивитесь, узнав, что в других странах осадки измеряются в миллиметрах.

```
mm_rain = input ('Введите количество осадков
в мм: ') rainfall [city_name] = rainfall.get
(city_name,
              0) + int (mm_rain)

for city, rain in rainfall.items ():
   print (f'{city}: {rain}')

get_rainfall ()
```

При первом упоминании города, мы добавляем 0 к его текущему количеству осадков. В следующие разы мы добавим текущее количество осадков к ранее сохраненному. dict.get делает это возможным.

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr91].

91

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr92].

92

После выполнения упражнения

Довольно стандартно использовать словари для отслеживания накопленных значений (таких как количество событий или сумма денег), связанных с произвольными значениями. Ключи могут представлять то, что вы отслеживаете, а значения могут отслеживать данные, имеющие отношение к ключу. Вот некоторые дополнительные возможности:

1. Вместо того чтобы печатать только общее количество осадков для каждого города, напечатайте общее количество осадков и среднее количество осадков за определенные дни. Таким образом, если бы вы ввели 30, 20 и 40 для Бостона, вы бы увидели, что общее число равно 90, а среднее значение равно 30.
2. Откройте файл журнала из системы Unix/Linux, например, из сервера Apache. Для каждого кода ответа (т.е.

трехзначного кода, указывающего на успех или неудачу HTTP–запроса) сохраните список IP–адресов, которые выдали этот код.

3. Чтение через текстовый файл на диске. Используйте словарь для отслеживания количества слов каждой длины в файле — то есть, сколько трехбуквенных слов, четырехбуквенных слов, пятибуквенных слов и так далее. Отобразите результаты.

Упражнение 16. Dictdiff

Умение работать со словарями имеет решающее значение для вашей карьеры Python–разработчика. Более того, как только вы научитесь эффективно использовать `dict.get`, вы обнаружите, что ваш код стал короче, элегантнее и удобнее в обслуживании.

Напишите функцию `dictdiff`, которая принимает два словаря в качестве аргументов. Функция вернет новый словарь, который будет представлять собой разницу между двумя словарями.

Если между словарями нет различий, `dictdiff` вернет пустой словарь. Для каждой отличающейся пары ключ–значение возвращаемое значение `dictdiff` будет иметь пару ключ–значение, в которой значение представляет собой список, содержащий значения из двух разных словарей. Если один из словарей не содержит этого ключа, он должен содержать `None`. Ниже приведены некоторые примеры:

Выведет «{}», потому что мы сравниваем d1 с самим собой.

```
d1 = {'a':1, 'b':2, 'c':3}
d2 = {'a':1, 'b':2, 'c':4}
print (dictdiff (d1, d1))
print (dictdiff (d1, d2))
```

Выведет «{'c': [3, 4]}», потому что у d1 есть c:3, а у d2 — c:4.

```
d3 = {'a':1, 'b':2, 'd':3}
d4 = {'a':1, 'b':2, 'c':4}
print (dictdiff (d3, d4))

d5 = {'a':1, 'b':2, 'd':4}
print (dictdiff (d1, d5))
```

Выведет «{'c': [3, 4]}», потому что у d1 есть c:3, а у d2 — c:4.

Выведет «{'c': [3, None], 'd': [None, 4]}», потому что у d1 есть c:3, а у d5 — d:4.

Обсуждение

Давайте начнем с размышлений об общей идее этой программы:

1. Мы создаем пустой `output` словарь.
2. Мы проходим по каждому из ключей в `first` и `second`.
3. Для каждого ключа мы проверяем, существует ли он в другом словаре.
4. Если ключ существует в обоих, то проверяем, одинаковы ли значения.
5. Если значения одинаковы, то мы ничего не делаем для `output`.
6. Если значения различны, то мы добавляем к output пару ключ–значение, содержащую текущий ключ и список значений из `first` и `second`.
7. Если ключ не существует ни в одном словаре, то в качестве значения мы задаем `None`.

Все это звучит хорошо, но есть проблема с этим подходом: это означает, что мы перебираем все ключи в `first`, а затем все ключи в `second`. Учитывая, что по крайней мере некоторые ключи, как мы надеемся, будут пересекаться, такой подход кажется неэффективным. Лучше и рациональнее было бы собрать все ключи из `first` и `second`, поместить их в множество (гарантируя, что каждый из них появится только один раз), а затем выполнить итерацию по ним.

Обратите внимание, `dict.keys ()` возвращает специальный объект типа `dict_keys`. Но этот объект реализует несколько тех же методов, которые доступны для множеств, включая `|` (объединение) и `&` (пересечение)! Результатом яв–

ляется множество, содержащее уникальные ключи из обоих словарей вместе:

```
all_keys = first.keys () | second.keys ()
```

ПРИМЕЧАНИЕ В Python 2 `dict.keys` и многие подобные методы возвращали списки, поддерживающие оператор `+`. В Python 3 почти все подобные методы были модифицированы для возврата итераторов. Когда возвращаемый результат мал, разница между реализациями практически отсутствует. Но когда возвращаемый результат большой, то разница велика, и большинство предпочитает использовать итератор. Таким образом, логика работы в Python 3 предпочтительнее, даже если она непривычна для людей, перешедших с Python 2.

Поскольку множество — это фактически словарь без значений, мы точно знаем, что, помещая эти списки в наше множество `all_keys`, мы пройдем через каждый ключ только один раз. Вместо того чтобы проверять, существует ли ключ в каждом словаре, затем извлекать его значение, а затем проверять, одинаковы ли значения, я использовал метод `dict.get`. Это спасет нас от получения исключения `KeyError`. Более того, если в одном из словарей отсутствует нужный ключ, мы получим обратно `None`. Мы можем использовать это не только для проверки того, одинаковы ли словари, но и для извлечения значений.

Теперь давайте пройдемся по каждому из приведенных примеров и посмотрим, что произойдет:

```
d1 = {'a':1, 'b':2, 'c':3}
print (dictdiff (d1, d1))
```

Мы видим этот пример на рисунке 4.4. На рисунке видно, что локальные переменные `first` и `second` указывают на один и тот же словарь — `d1`.

Рисунок 4.4 Вычитаем `d1` из самого себя.

Когда мы выполняем итерацию над объединенным множеством ключей (рисунок 4.5), мы фактически выполняем итерацию над ключами `d1`. Поскольку мы никогда не находим никаких различий, возвращаемое значение (`output`) — `{}` (пустой словарь).

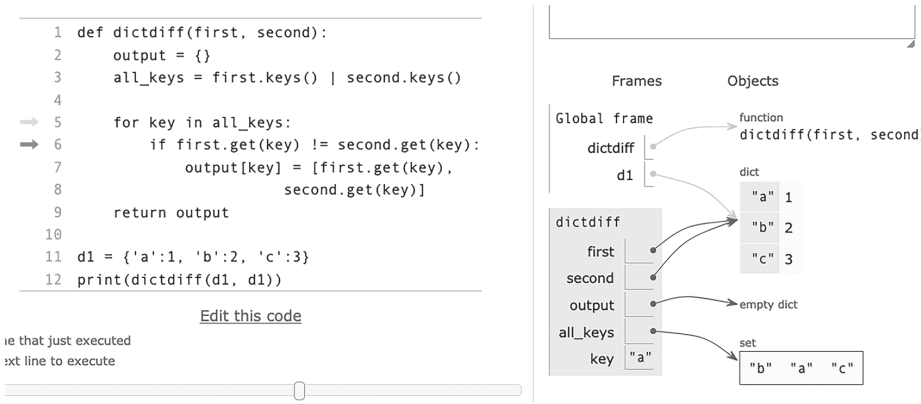


Рисунок 4.5. Итерация по ключам `d1`.

Сравнивая `d1` и `d2`, мы видим, что `first` и `second` указывают на два разных словаря (рисунок 4.6). Они также имеют одинаковые ключи, но разные значения для ключа c. На рисунке 4.7 показано, как наш словарь output получает новую пару ключ–значение, представляющую различные значения ключа `c`.

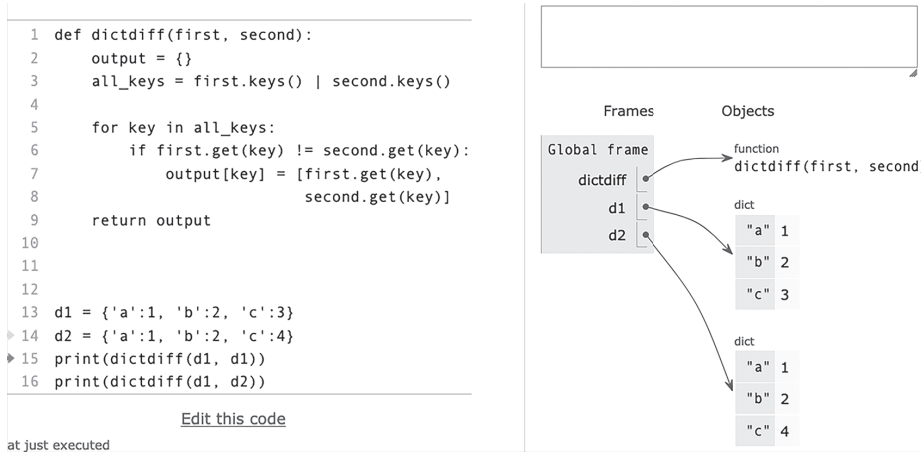
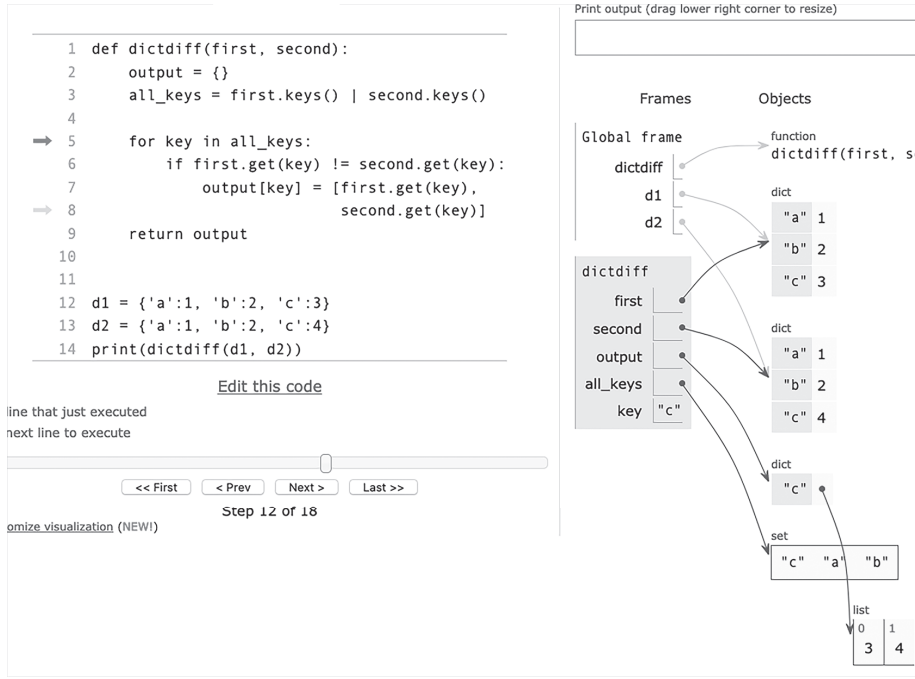


Рисунок 4.6 Сравнение `d1` *и* `d2`.



Сравнивая `d3` и `d4`, мы видим, как все усложняется. Теперь наш результирующий словарь будет содержать две пары ключ–значение, и каждое значение будет (как указано) списком. Таким образом, вы можете увидеть, как мы создаем наш словарь из ничего, чтобы поместить в него различия между двумя аргументами.

Решение

```
def dictdiff (first, second):
output = {}
all_keys = first.keys () | second.keys ()
```

Получаем все ключи как из первого, так и из второго, без повторений.

```
    for key in all_keys:
         if first.get (key)!= second.get (key):
              output [key] = [first.get (key),
                   second.get (key)]
    return output

d1 = {'a':1, 'b':2, 'c':3}
d2 = {'a':1, 'b':2, 'd':4}
print (dictdiff (d1, d2))
```

Используется тот факт, что dict.get возвращает None, если ключ не существует.

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr93].

93

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr94].

94

После выполнения упражнения

Функции Python могут возвращать любой объект, в том числе и словари. Зачастую полезно написать функцию, которая создает словарь: функция может объединять или суммировать другие словари (как в этом упражнении), или превращать другие объекты в словари. Вот несколько идей, которые вы можете реализовать:

1. Метод `dict.update` объединяет два словаря. Напишите функцию, которая принимает любое количество словарей и возвращает словарь, представляющий собой комбина–

цию из них. Если один и тот же ключ появляется более чем в одном словаре, то в выходных данных должно появиться значение самого последнего объединенного словаря.

2. Напишите функцию, которая принимает любое четное количество аргументов и возвращает на их основе словарь. Аргументы с четным индексом становятся ключами словаря, а аргументы с нечетным номером становятся значениями словаря. Таким образом, вызов функции с аргументами ('a', 1, 'b', 2) приведет к возвращению словаря {'a':1, 'b':2}.
3. Напишите функцию `dict_partition`, которая принимает один словарь (`d`) и функцию (`f`) в качестве аргументов. `dict_partition` вернет два словаря, каждый из которых содержит пары ключ–значение из `d`. Решение о том, куда поместить каждую из пар ключ–значение, будет приниматься в соответствии с выходными данными `f`, которые будут выполняться для каждой пары ключ–значение в `d`. Если `f` возвращает `True`, то пара ключ–значение будет помещена в первый результирующий словарь. Если `f` возвращает `False`, то пара ключ–значение будет помещена во второй результирующий словарь.

Упражнение 17. Сколько всего разных чисел?

В моей консультационной работе мне иногда интересно найти сообщения об ошибке, IP–адреса или имена пользователей в файле журнала. Но если сообщение, адрес или имя пользователя появляется дважды, то никакой пользы от этого нет. Таким образом, я хотел бы убедиться, что я просматриваю каждое значение только один раз без возможности повторения.

В этом упражнении вы можете предположить, что ваша программа Python содержит список целых чисел. Мы хотим напечатать, сколько различных целых чисел содержится в этом списке. Таким образом, рассмотрим следующее:

```
numbers = [1, 2, 3, 1, 2, 3, 4, 1]
```

При таком определении выполнение `len (numbers)` вернет `7`, поскольку список содержит семь элементов. Как мы можем получить в результате `4`, указывающую на то, что список содержит четыре различных значения? Напишите функцию с именем `how_many_different_numbers`, которая принимает один список целых чисел и возвращает количество различных целых чисел, которые он содержит.

Обсуждение

Множество по определению содержит уникальные элементы — точно так же, как и ключи словаря являются гарантированно уникальными. Таким образом, если у вас есть список значений, из которого вы хотите удалить все дубликаты, вы можете просто создать множество. Вы можете создать множество, как показано в коде решения

```
unique_numbers = set (numbers)
```

или вы можете создать пустое множество, а затем добавить в него новые элементы:

```
numbers = [1, 2, 3, 1, 2, 3, 4, 1] unique_
numbers = set ()
for number in numbers:
unique_numbers.add (number)
```

В этом примере используется функция `set.add`, которая добавляет один новый элемент во множество. Вы можете добавлять большое количество элементов с помощью `set.update`, которая принимает в качестве аргумента итерируемый объект:

```
numbers = [1, 2, 3, 1, 2, 3, 4, 1]
unique_numbers = set ()
```

```
unique_numbers.update (numbers)
```

Вы можете использовать set.update только с итерируемым объектом. Считайте это сокращением выполнения цикла for для каждого из элементов числа, вызывая set.add для текущего элемента итерации.

Наконец, у вас может возникнуть соблазн использовать синтаксис фигурных скобок для множеств:

```
numbers = [1, 2, 3, 1, 2, 3, 4, 1]
unique_numbers = {numbers}
```

Не работает!

Этот код не будет работать, потому что Python думает, что вы хотите добавить список `numbers` к множеству как один элемент. А так как списки не могут быть ключами словаря, они также не могут быть элементами множества.

Но, конечно, мы не хотим добавлять `numbers`. Скорее, мы хотим добавить элементы внутри `numbers`. Здесь мы можем использовать оператор `*` (`splat`), но несколько иначе, чем мы видели раньше:

```
numbers = [1, 2, 3, 1, 2, 3, 4, 1]
unique_numbers = {*numbers}
```

Мы сообщаем Python, что он должен брать элементы `numbers` и передавать их (в своем роде цикл `for`) в фигурные скобки. И действительно, это прекрасно работает.

Что лучше: использовать `set` без `*` или `{}` с `*`? На ваше усмотрение. Я предпочитаю фигурные скобки и `*`, но я также понимаю, что `*` может сбить с толку многих людей и сделать ваш код менее читабельным и удобным для новичков.

Решение

Вызывает множество чисел, таким образом возвращая множество с уникальными элементами из чисел.

```
def how_many_different_numbers (numbers):
    unique_numbers = set (numbers)
    return len (unique_numbers)
```

```
print (how_many_different_numbers
([1,2,3,1,2,3,4,1]))
```


95

Вы можете ознакомиться с кодом в Python Tutor по ссылке [qr95].

Скринкаст решения


96

Посмотрите короткое видео с объяснением решения: [qr96].

После выполнения упражнения

Всякий раз, когда я встречаю *уникальный* или *отличающийся* в спецификации проекта, я думаю о множествах, потому что они автоматически обеспечивают уникальность и работают с последовательностью значений. Поэтому если у вас есть последовательность имен пользователей, дат, IPадресов, адресов электронной почты или продуктов и вы хотите свести ее к последовательности, содержащей те же данные, но с каждым элементом, появляющимся только один раз, то множества могут быть чрезвычайно полезны.

Если вы хотите попрактиковаться в работе с множествами, можете попробовать выполнить следующие задания:

1. Просмотрите файл журнала сервера (например, `Apache` или `nginx`). Какие различные IP–адреса пытались получить доступ к вашему серверу?
2. Просматривая тот же журнал сервера, постарайтесь ответить, какие коды ответов были возвращены пользователям? Код 200 означает «OK», но есть также ошибки 403, 404 и 500. (Регулярные выражения здесь необязательны, но, возможно, помогут.)
3. Используйте `os.listdir`, чтобы получить имена файлов в текущем каталоге. Какие расширения файлов (т.е. суффиксы, следующие за конечным символом «`.`») находятся в этом каталоге? Вероятно, будет полезно использовать `os.path.splitext`.

Подводя итоги

Словари, без сомнения, являются самой универсальной и важной структурой данных в мире Python. Научиться использовать их эффективно и результативно — очень важно для того, чтобы чувствовать себя уверенно. В этой главе мы рассмотрели несколько способов их использования, включая отслеживание количества элементов и хранение данных, полученных от пользователя. Мы также увидели, что можно использовать `dict.get` для получения данных из словаря, не опасаясь, что ключ не существует.

При работе со словарями помните:

1. Ключи должны быть хэшируемыми, например, число или строка.
2. Значения могут быть любыми, включая другой словарь.
3. Ключи уникальны.
4. Можно выполнять итерации по ключам в цикле `for` или в генераторе.

5. Файлы

Файлы — неотъемлемая часть мира компьютеров, а значит, и программирования. Мы читаем данные из файлов и пишем в файлы. Даже когда что-то не является файлом — например, сетевое соединение — мы стараемся использовать интерфейс, похожий на файловый, потому что он нам хорошо знаком.

Для обычных пользователей существуют различные типы файлов — Word, Excel, Power Point, PDF и другие.

Для программистов все и проще, и сложнее. Проще в том смысле, что мы рассматриваем файлы как структуры данных, в которые можно записывать строки и из которых можно читать строки. Но работа с файлами также сложнее потому, что при считывании строки в память нам может понадобиться распарсить ее в структуру данных.

Работа с файлами — одна из самых простых и понятных вещей в Python. Это также одна из самых распространенных вещей, которые нам нужно делать, поскольку программы, не взаимодействующие с файловой системой, довольно скучны.

В этой главе мы будем практиковать работу с файлами: читать, записывать и обрабатывать данные, которые они содержат. Попутно вы познакомитесь с некоторыми концепциями, которые обычно используются при работе с файлами в Python, такие как итерация по содержимому файла и запись в файлы при помощи `with`.

В некоторых случаях мы будем работать с данными в формате CSV (значения, разделенные запятыми) или JSON (объектная нотация JavaScript) — два распространенных формата, с которыми работают модули стандартной библиотеки Python. Если вы за-

были основы работы с CSV или JSON, в этой главе приведены некоторые краткие напоминания.

После этой главы вам будет не только удобнее работать с файлами, но вы также будете лучше понимать, как можно переходить от структур данных в памяти (например, списков и словарей) к форматам данных на диске (например, CSV и JSON) и обратно. Таким образом, файлы позволяют сохранять структуры данных в неизменном виде — даже когда программа не работает или компьютер выключен — или же передавать такие структуры данных на другие компьютеры.

Таблица 5.1. Что вам нужно знать

Понятие	Что это?	Пример	Чтобы узнать подробнее
Файлы	Описание работы с файлами в Python.	`f = open ('/etc/ passwd')`	
`with`	Помещает объект в контекстный менеджер; обеспечивает очистку файла и закрывает его к концу блока.	`with open ('file. text') as f:`	
Контекстный менеджер	Позволяет вашим собственным объектам работать с оператором with.	`with MyObject () as m:`	
`set.update`	Добавляет элементы к множеству.	`s.update ([10, 20, 30])`	
`os.stat`	Извлекает информацию (размер, разрешения, тип) о файле.	`os.stat ('file. txt')`	

Окончание таблицы

Понятие	Что это?	Пример	Чтобы узнать подробнее
`os.listdir`	Возвращает список файлов каталога.	`os.listdir ('/etc/')`	
`glob.glob`	Возвращает список файлов, соответствующих шаблону.	`glob.glob ('/etc/*.conf')`	
Генератор словаря	Создает словарь на основе итератора.	`{word: len (word) for word in 'ab cde'. split ()}`	
`str.split`	Разбивает строки на части, возвращая список.	`# Возвращает ['ab', 'cd', 'ef'] 'ab cd ef'. split ()`	
`hashlib`	Модуль с криптографическими функциями.	`import hashlib`	
`csv`	Модуль для работы с файлами CSV.	`x = csv. reader (f)`	
`json`	Модуль для работы с JSON.	`json.loads (json_ string)`	

Упражнение 18. Последняя строка

Очень часто начинающие Python–программисты изучают, как можно перебирать строки файла, выводя по одной строке за раз. Но что, если меня не интересует каждая строка или даже большинство строк? Что, если меня интересует только одна строка файла — последняя?

Получение последней строки файла может показаться не слишком полезным действием. Но вспомните утилиты Unix `head` и `tail`, которые показывают первую и последнюю строки файла соответственно и которые я постоянно использую для просмотра файлов, особенно файлов журналов и конфигураций. Более того, умение читать определенные части файла, а не весь файл целиком, является полезным практическим навыком.

В этом упражнении вы напишите функцию (`get_final_line`), которая принимает в качестве аргумента имя файла. Функция должна печатать на экран последнюю строку этого файла.

Обсуждение

В коде решения используется ряд распространенных идиом Python, которые я сейчас объясню. По ходу дела вы увидите, как использование этих идиом приводит не только к более читабельному коду, но и к более эффективному выполнению.

В зависимости от того, какие аргументы вы используете при ее вызове, встроенная функция `open` может вернуть несколько различных объектов, таких как `TextIOWrapper` или `BufferedReader`. Все эти объекты реализуют один и тот же API для работы с файлами и поэтому описываются в мире Python как «файлоподобные объекты». Использование такого объекта позволяет нам забыть о множестве различных типов файловых систем и думать только в «файловых» терминах. Такой объект также позволяет нам воспользоваться преимуществами любых оптимизаций, таких как буферизация, которые может использовать операционная система.

Вот как обычно вызывается `open`:

```
f = open (filename)
```

В данном случае `filename` — это строка, представляющая собой корректное имя файла. Если мы вызываем `open` только с одним аргументом, этим аргументом обязательно будет имя файла. Второй, необязательный аргумент, — это строка, которая может включать несколько символов, указывающих, хотим ли мы читать из файла, записывать или добавлять в него (используя `r`, `w` или `a`), а также читать ли файл посимвольно (по умолчанию) или побайтно (параметр `b`, в этом случае мы будем использовать `rb`, `wb` или `ab`). (См. сноску про параметр b и чтение файла в байтовом, или двоичном, режиме.) Таким образом, я могу записать предыдущую строку кода в более полном виде как:

```
f = open (filename, 'r')
```

Поскольку мы читаем из файлов чаще, чем пишем в них, `r` является значением по умолчанию для второго аргумента. Обычно программы на Python не указывают `r` при чтении из файла.

Как вы могли заметить, мы поместили полученный объект в переменную `f`. А поскольку все файлоподобные объекты являются итерируемыми объектами, возвращая одну строку за итерацию, то:

```
for current_line in f:
print (current_line)
```

Но если вы планируете выполнить итерацию по `f` только один раз, то зачем вообще создавать ее как переменную? Мы можем обойтись без определения переменной и просто проитерировать файловый объект, который возвращает `open`:

```
for current_line in open (filename):
print (current_line)
```

При каждой итерации над файлоподобным объектом мы получаем следующую строку из файла до символа новой строки

`\n` включительно. Таким образом, в этом коде `line` всегда будет строкой, которая всегда содержит один символ `\n` в конце. Пустая строка в файле будет содержать только символ новой строки `\n`.

Теоретически файлы должны заканчиваться `\n`, так что вы никогда не закончите файл на середине строки. На практике я видел много файлов, которые не заканчиваются `\n`. Помните об этом, когда будете распечатывать содержимое файла: предположение, что файл всегда будет заканчиваться символом новой строки, может привести к проблемам.

А как насчет закрытия файла? Этот код будет работать и напечатает длину каждой строки в файле. Однако такой код не одобряется в мире Python, потому что он явно не закрывает файл. Когда речь идет о чтении из файлов, это не такая уж большая проблема, особенно если вы открываете только небольшое их количество за раз. Но если вы пишете в файлы или открываете много файлов одновременно, то вам следует закрывать файлы как для экономии ресурсов, так и для того, чтобы убедиться, что файл действительно закрыт.

Это можно сделать с помощью конструкции `with`. Я могу переписать предыдущий код следующим образом:

```
with open (filename) as f:
for one_line in f:
print (len (one_line))
```

Вместо того чтобы открывать файл и присваивать файловый объект непосредственно `f`, мы открыли его при помощи with, присвоили его `f` как часть оператора `with`, а затем открыли блок.

Более подробно об этом говорится в сноске про `with` и «контекстные менеджеры», но вы должны знать, что это стандартный питоновский способ открыть файл — в немалой степени потому, что он гарантирует, что файл будет закрыт к концу блока.

Бинарный режим с использованием b

Что произойдет, если открыть нетекстовый файл, например, PDF или JPEG, с помощью `open`, а затем попытаться выполнить итерации по одной строке за раз?

Во–первых, скорее всего, вы сразу же получите ошибку, потому что Python ожидает, что содержимое файла будет действительной строкой Unicode в формате UTF–8. Двоичные файлы по определению не используют Unicode. Когда Python попытается прочитать строку не в формате Unicode, он выдаст исключение, объясняя, что не может определить строку с таким содержимым.

Чтобы избежать этой проблемы, вы можете и должны открыть файл в *бинарном* или *байтовом* режиме, добавив `b` к `r`, `w` или `a` во втором аргументе к open, например:

```
for current_line in open (filename, 'rb'):
    print (current_line)
```

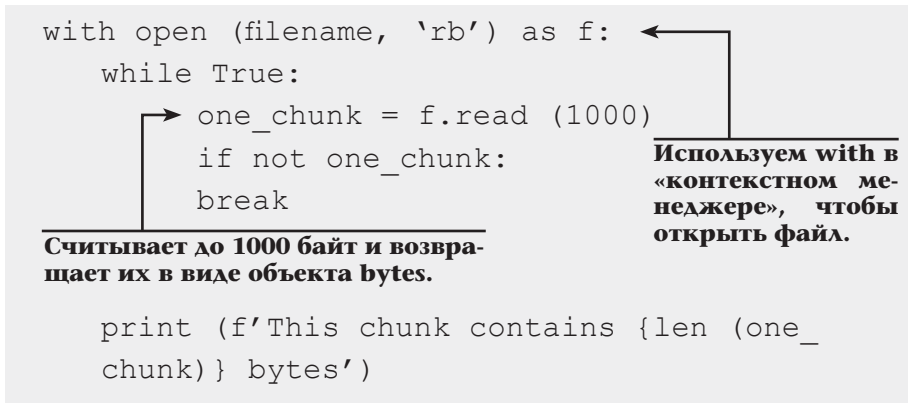
Открывает файл в режимах r (чтение) и b (двоичный).

Здесь тип current_line — байтовый, подобно строке, но без символов Unicode.

Теперь вас не будет сдерживать отсутствие символов Unicode.

Но подождите. Помните, что с каждой итерацией Python возвращает все до следующего символа `\n` включительно. В двоичном файле такой символ не будет появляться в конце каждой строки, потому что там нет строк, о которых можно было бы говорить. Без такого символа то, что вы получите в результате каждой итерации, скорее всего, будет бессмыслицей.

В итоге, если вы читаете из двоичного файла, то не забывайте использовать флаг `b`. Но, когда вы это сделаете, вы обнаружите, что не хотите читать файл по строкам. Вместо этого вы должны использовать метод `read` для получения фиксированного количества байт. Когда `read` возвращает 0 байт, вы будете знать, что находитесь в конце файла, например:



В этом конкретном упражнении вас попросили вывести последнюю строку файла. Один из способов:

```
for current_line in open (filename):
        pass
print (current_line)
```

Этот трюк работает, потому что мы перебираем строки файла и присваиваем `current_line` на каждой итерации, но на самом деле мы ничего не делаем в теле цикла `for`. Скорее, мы используем `pass`, который является способом сказать Python ничего не делать. (Python требует, чтобы у нас была хотя бы одна строка в блоке с отступом, например, тело цикла `for`.) Причина, по которой мы выполняем этот цикл, заключается в его побочном эффекте — а именно в том, что конечное значение, присвоенное `current_line`, остается на месте после выхода из цикла.

Однако перебор строк файла только для того, чтобы получить последнюю, кажется мне немного странным, даже если этот способ работает. Мое любимое решение, показанное на рисунке 5.1, заключается в итерационном просмотре каждой строки файла, получении текущей строк и немедленном присвоении ей значения `final_line`.

Когда мы выйдем из цикла, `final_line` будет содержать все, что было в самой последней строке. Таким образом, мы можем вывести ее на печать.

```
1  from io import StringIO
2
3  fakefile = StringIO('''
4  nobody:*:-2:-2::0:0:Unprivileged User:/var/empty:/usr/bin/false
5  root:*:0:0::0:0:System Administrator:/var/root:/bin/sh
6  daemon:*:1:1::0:0:System Services:/var/root:/usr/bin/false
7  ''')
8
9  def get_final_line(filename):
10     final_line = ''
11     for current_line in fakefile:
12         final_line = current_line
13     return final_line
14
15 print(get_final_line('/etc/passwd'))
```

Рисунок 5.1. Перед печатью последней строки.

Обычно `print` добавляет новую строку после печати чего-либо на экран. Однако, когда мы итерируем файл, каждая строка уже заканчивается символом новой строки. Это может привести к удвоению пробельных символов между выводимыми данными. Решение состоит в том, чтобы запретить `print` выводить что-либо на экран, переопределив значение по умолчанию `\n` в параметре `end`. Передавая `end=''`, мы указываем `print` добавить `''`, пустую строку, после печати `final_line`. С более подробной информацией об аргументах, которые можно передавать в `print`, ознакомтесь здесь:

http://mng.bz/RAAZ.

Решение

```
def get_final_line (filename):
    final_line = ''
    for current_line in open (filename):
        final_line = current_line
    return final_line

print (get_final_line ('/etc/
passwd'))
```

Итерируем каждую строку файла. Вам не нужно объявлять переменную: просто выполняйте итерацию непосредственно над результатом open.

109

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr109].

Симуляция файлов в Python Tutor

Сайт Python Tutor Филипа Гуо, который я использую для иллюстраций, а также для того, чтобы вы могли экспериментировать с решениями книги, не поддерживает работу с файлами. Это вполне объяснимо — свободную серверную систему, позволяющую людям запускать произвольный код, достаточно сложно создать и поддерживать. Разрешение людям работать с произвольными файлами добавило бы множество проблем с доставкой и безопасностью.

Однако существует решение — `StringIO`. Объекты `StringIO` — это то, что Python называет «файлоподобными объектами». Они реализуют тот же API, что и объекты `file`, позволяя нам читать из них и записывать в них так же, как в файлы. Однако, в отличие от файлов, объекты `StringIO` никогда не обращаются к файловой системе.

`StringIO` не был разработан для использования в Python Tutor, хотя он отлично подходит для решения проблем, связанных с его ограничениями. Чаще всего я вижу (и использую) `StringIO` в автоматизированных тестах. В конце концов, вы же не хотите, чтобы тест обращался к файловой системе — это сделает работу намного медленнее. Вместо этого вы можете использовать `StringIO` для имитации файла. Если вы занимаетесь тестированием программного обеспечения, вам стоит обратить серьезное внимание на `StringIO`, часть стандартной библиотеки Python. Вы можете загрузить его с помощью `from io import StringIO`.

Таким образом, версии кода в Python Tutor будут немного отличаться от версий в самой книге. Однако они должны работать одинаково, позволяя вам изучать код наглядно. К сожалению, отсутствует возможность добавить упражнения со списками каталогов и поэтому отсутствуют ссылки на Python Tutor.

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr110].

После выполнения упражнения

При работе с Python очень важно уметь итерировать файлы и понимать, как работать с их содержимым в процессе (и после) итерации. Также важно понимать, как превратить содержимое файла в структуру данных Python — то, что мы еще несколько раз рассмотрим в этой главе. Вот несколько идей того, что можно делать при таком итерационном просмотре файлов:

1. Итерация по строкам текстового файла. Найдите все слова (без пробелов в записи слова и не окруженные пробелами), которые содержат только целые числа, и росуммируйте их.
2. Создайте текстовый файл (с помощью редактора, не обязательно Python), содержащий два столбца, разделенных табуляцией, каждый из которых содержит число. Затем с помощью Python прочитайте созданный файл. Для каждой строки умножьте каждое первое число на второе, а затем просуммируйте результаты всех строк. Игнорируйте все строки, которые не содержат двух числовых столбцов.
3. Прочитайте текстовый файл, строка за строкой С помощью словаря подсчитайте, сколько раз каждая гласная (a, e, i, o и u) встречается в файле. Распечатайте полученную таблицу.

Упражнение 19. Создаем словарь из /etc/passwd

Принято (и полезно) рассматривать файлы как последовательности строк. В конце концов, когда вы итерируете файловый объект, вы получаете каждую из строк файла в виде строки, по одной за раз. Но часто имеет смысл превратить файл в более сложную структуру данных, такую как словарь.

В этом упражнении вы напишите функцию `passwd_to_dict`, которая считывает данные из «файла паролей» в стиле Unix, обычно хранящегося в файле `/etc/passwd`, и возвращает на его основе словарь. Если у вас нет доступа к такому файлу, вы можете скачать его по адресу [qr111].

Вот пример того, как выглядит файл:

```
nobody:*:—2:—2::0:0: Unprivileged User:/var/
empty:/usr/bin/false root:*:0:0::0:0: System
Administrator:/var/root:/bin/sh
daemon:*:1:1::0:0: System Services:/var/root:/
usr/bin/false
```

Каждая строка — это одна запись пользователя, состоящая из полей, разделенных двоеточием. Первое поле (индекс 0) — это имя пользователя, а третье поле (индекс 2) — уникальный идентификационный номер пользователя. (В системе, из которой я взял файл `/etc/passwd`, `nobody` имеет ID `-2`, `root` — ID `0`, а `daemon` — ID `1`.) Для наших целей вы можете игнорировать все поля, кроме этих двух.

Иногда файл содержит строки, которые не соответствуют этому формату. Например, мы обычно игнорируем строки, не содержащие ничего, кроме пробелов. Некоторые производители (например, Apple) включают комментарии в свои файлы `/etc/passwd`, в которых строка начинается с символа `#`.

Функция `passwd_to_dict` должна возвращать словарь на основе `/etc/passwd`, в котором ключами словаря являются имена пользователей, а значениями — идентификаторы пользователей.

Как строковые методы могут немного помочь

При проведении такого рода анализа и различных процедур полезны строковые методы `str.startswith`, `str.endswith` и `str.strip`.

Например, `str.startswith` возвращает `True` или `False`, в зависимости от того, начинается ли строка с определенной строки или нет:

```
s = 'abcd'
s.startswith ('a') → # Возвращает True
s.startswith ('abc') → # Возвращает True
s.startswith ('b') → # Возвращает False
```

Аналогично, `str.endswith` сообщает нам, заканчивается ли строка определенной строкой:

```
s = 'abcd'
s.endswith ('d')  → # Возвращает True
s.endswith ('cd') → # Возвращает True
s.endswith ('b')  → # Возвращает False
```

`str.strip` удаляет пробелы, а также \n, \r, \t и даже \v — с обеих сторон строки. Методы `str.lstrip` и `str.rstrip` удаляют пробелы только слева и справа соответственно.

Все эти методы возвращают строки:

```
s = ' \t\t\ta b c \t\t\n'
s.strip () # возвращает 'a b c'
s.lstrip () # возвращает 'a b c \t\t\n'
s.rstrip () # возвращает ' \t\t\ta b c'
```

Обсуждение

И снова мы открываем текстовый файл и итерируем его строки по одной за раз. Здесь мы предполагаем, что знаем формат файла и можем извлекать поля из каждой записи.

В данном случае мы разделяем каждую строку символом «:», используя метод `str.split`. `str.split` всегда возвращает список

строк, хотя длина этого списка зависит от того, сколько раз символ: встречается в строке. В случае с `/etc/passwd` мы будем считать, что любая строка, содержащая «:», является корректной записью пользователя и, следовательно, содержит все необходимые поля.

Однако файл может содержать строки комментариев, начинающиеся с `#`. Если бы мы вызвали str.split для этих строк, мы бы получили список, содержащий только один элемент, что приведет к исключению IndexError, если мы попытаемся получить `user_info` [2].

Поэтому важно игнорировать те строки, которые начинаются с `#`. К счастью, мы можем использовать метод `str.startswith`. В частности, я нахожу и стираю комментарии и пустые строки при помощи следующего кода: `if not line.startswith (('#', '\n')):`

При вызове функции `str.startswith` передается кортеж из двух строк. `str.startswith` вернет `True`, если одна из строк в этом кортеже находится в начале строки. Поскольку каждая строка содержит новую строку, включая пустые строки, можно сказать, что строка, начинающаяся с `\n`, является пустой строкой.

Предполагая, что запись о пользователе найдена, наша программа добавит новую пару ключ–значение в `users`. Ключ — `user_info [0]`, а значение — `user_info [2]`. Обратите внимание, пока значение переменной `user_info [0]` содержит строку, мы можем использовать ее в качестве ключа словаря.

Здесь я использую `with` для открытия файла, это гарантирует его закрытие после завершения блока.

(См. сноски о `with` и контекстных менеджерах.) [qr112]

Решение

```
def passwd_to_dict (filename):
    users = {}
    with open (filename) as passwd:
        for line in passwd:
            if not line.startswith (('#', '\n')):
```

Игнорирует комментарии и пустые строки.

```
            user_info = line.split (':')
            users [user_info [0]] =
            int (user_info [2])
    return users
```

Превращает строку в список строк.

```
print (passwd_to_dict ('/etc/passwd'))
```

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr113].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr114].

После выполнения упражнения

В определенный момент своей карьеры в Python вы перестанете воспринимать файлы как последовательности символов, хранящихся на диске, и начнете видеть в них сырой материал, который можно преобразовать в структуры данных Python. Наши программы обладают большей семантической силой при работе со структурированными данными (например, со словарями), чем со строками. Мы также можем делать больше и мыслить глубже, если читать файл как структуру данных, а не просто как строку.

Например, представьте себе CSV–файл, в котором каждая строка содержит название страны и ее население. Если читать этот файл как строку, то сравнить, например, население Франции и Таиланда будет возможно, но утомительно. Но если вы прочитаете этот файл как словарь, то такое сравнение будет довольно простым.

На самом деле, я большой поклонник чтения файлов в словари, в немалой степени потому, что многие форматы файлов поддаются такому переводу, но вы можете использовать и более сложные структуры данных. Вот несколько дополнительных упражнений, которые могут помочь вам увидеть эту связь и осуществить преобразование в вашем коде:

1. Прочитайте `/etc/passwd`, создав словарь, в котором входы в систему (последнее поле в каждой строке) являются ключами. Каждое значение будет представлять собой список пользователей, для которых эта оболочка определена как оболочка входа в систему.
2. Попросите пользователя ввести целые числа, разделенные пробелами. Из этого ввода создайте словарь, ключами которого являются коэффициенты для каждого числа, а значениями — списки, содержащие те целые числа пользователя, которые кратны этим коэффициентам.
3. Из `/etc/passwd` создайте словарь, в котором ключами будут имена пользователей (как в основном упражнении), а значениями — сами словари с ключами (и соответствующими значениями) для ID пользователя, домашнего каталога и оболочки.

`with` и контекстные менеджеры

Как мы уже видели, обычно файл открывается следующим образом:

```
with open ('myfile.txt', 'w') as f:
      f.write ('abc\n')
      f.write ('def\n')
```

Большинство людей правильно полагает, что использование `with` гарантирует, что файл `f` будет очищен и закрыт в конце блока. (Таким образом, вам не нужно явно вызывать `f.close ()`, чтобы обеспечить удаление содержимого.) Но поскольку with в подавляющем большинстве случаев используется с файлами, многие разработчики считают, что между `with` и файлами существует некая внутренняя связь. Правда в том, что `with` — это гораздо более общая конструкция Python, известная как контекстный менеджер. Основная идея заключается в следующем:

1. Вы используете `with` вместе с объектом и переменной, которой вы хотите присвоить объект.
2. Объект должен знать, как вести себя внутри контекстного менеджера.
3. Когда блок запускается, `with` обращается к объекту. Если для объекта определен метод `__enter__`, то он вызывается. В случае с файлами метод определен, но ничего не делает, кроме как возвращает сам файловый объект. Все, что возвращает этот метод, присваивается переменной `as` в конце строки `with`.
4. Когда блок заканчивается, `with` снова обращается к объекту, выполняя метод `__exit__`. Этот метод дает объекту возможность изменить или восстановить состояние, в котором он находился.

Итак, совершенно очевидно, каким образом `with` работает с файлами. Возможно, метод `__enter__` не важен и мало что делает, но метод `__exit__`, безусловно, важен и делает многое — в частности, выполняет сброс и закрывает файл. Если вы передаете в with двое или более объектов, методы `__enter__` и `__exit__` вызываются для каждого из них по очереди.

Другие объекты придерживаются протокола контекстного менеджера. Действительно, если вы хотите, вы можете написать свои собственные классы так, чтобы они знали, как вести себя внутри оператора `with`. (Подробности о том, как это сделать, приведены в таблице «Что нужно знать» в начале главы.)

Используются ли контекстные менеджеры только для работы с файлами? Нет, но это самый распространенный случай. Два других распространенных случая — (1) при обработке транзакций баз данных и (2) при блокировке определенных секций в многопоточном коде. В обеих ситуациях вы хотите иметь участок кода, который

выполняется в определенном контексте — и здесь на помощь приходит контекстное управление Python через `with`.

Если вы хотите узнать больше о контекстных менеджерах, вот хорошая статья на эту тему: [qr115].


115

Упражнение 20. Счетчик слов

Unix-системы содержат множество служебных функций. Одна из самых полезных для меня — `wc`, программа подсчета слов. Если запустить `wc` на текстовом файле, она подсчитает символы, слова и строки, которые содержит файл.

Задача этого упражнения — написать функцию `wordcount`, имитирующую команду `wc` Unix. Функция будет принимать на вход имя файла и печатать четыре строки вывода:

1. Количество символов (включая пробельные символы).
2. Количество слов (разделенных пробелами).
3. Количество линий.
4. Количество уникальных слов (с учетом регистра, поэтому NO будет отличаться от no).

Я разместил тестовый файл (wcfile.txt) на сайте [qr116]. Вы можете загрузить и использовать этот файл для тестирования вашей реализации `wc`. Подойдет любой файл, но, если вы используете этот, ваши результаты будут совпадать с моими. Содержимое этого файла будет выглядеть следующим образом:


116

```
Это тестовый файл.
Он содержит 28 слов и 20 различных слов.
Он также содержит 165 символов.
Он также содержит 11 строк.
```

```
Он также является самореферентной.
Вау!
```

Это упражнение, как и многие другие в этой главе, пытается помочь вам увидеть связь между текстовыми файлами и встроенными структурами данных Python. Очень часто Python используется для работы с файлами журналов и конфигурационными файлами, собирая и представляя эти данные в удобочитаемом формате.

Обсуждение

Эта программа демонстрирует ряд возможностей Python, которые многие программисты используют ежедневно. Прежде всего, многие новички в Python считают, что если им нужно получить информацию о четырех аспектах файла, то они должны прочитать его четыре раза. Это может означать, что нужно открыть файл один раз и прочитать его четыре раза, или даже открыть его четыре отдельных раза. Но в Python чаще всего файл просматривается один раз, итерация по каждой строке и накопление всех данных, которые программа может найти в этой строке.

Как мы будем хранить эти данные? Мы могли бы использовать отдельные переменные, и в этом нет ничего плохого. Но я предпочитаю использовать словарь (рисунок 5.2), поскольку подсчеты тесно связаны между собой, а также потому, что это сокращает объем кода, необходимого для создания отчета.

Итак, когда мы итерируем строки файла, как мы можем подсчитать число различных элементов? Подсчитывать строки проще всего: каждая итерация проходит через одну строку, поэтому мы можем просто добавить 1 к `counts ['lines']` в начале цикла.

Далее мы хотим подсчитать количество символов в файле. Поскольку мы уже выполняем итерацию по файлу, работы не так много. Мы получаем количество символов в текущей строке, вычисляя `len (one_line)`, а затем добавляем его к `counts ['characters']`.

Многие удивляются, что в это число входят пробелы и символы табуляции, а также символы новой строки. Да, даже «пустая» строка содержит один символ новой строки. Но если бы у нас не было символов новой строки, то для компьютера не было бы очевидно, когда он должен начинать новую строку. Поэтому такие символы необходимы, и они занимают некоторое место.

Далее мы хотим подсчитать количество слов. Для этого превратим `one_line` в список слов, вызвав `one_line.split`. Решение вызовет `split` без каких-либо аргументов, что в свою очередь вынудит его использовать все пробельные символы — пробелы, табуляцию и новые строки — в качестве разделителей. Результат помещаем в `counts ['words']`.

Последний пункт для подсчета — это уникальные слова. Теоретически мы могли бы использовать список для хранения новых слов. Но гораздо проще позволить Python сделать эту работу за нас, используя множество для гарантии уникальности. Так, мы создаем множество `unique_words` в начале программы, а затем используем `unique_words.update` для добавления всех слов в текущей строке в это множество (рисунок 5.3). Чтобы отчет работал с нашим словарем, мы добавим новую пару ключ-значение в `counts`, используя `len (unique_words)` для подсчета количества слов во множестве.

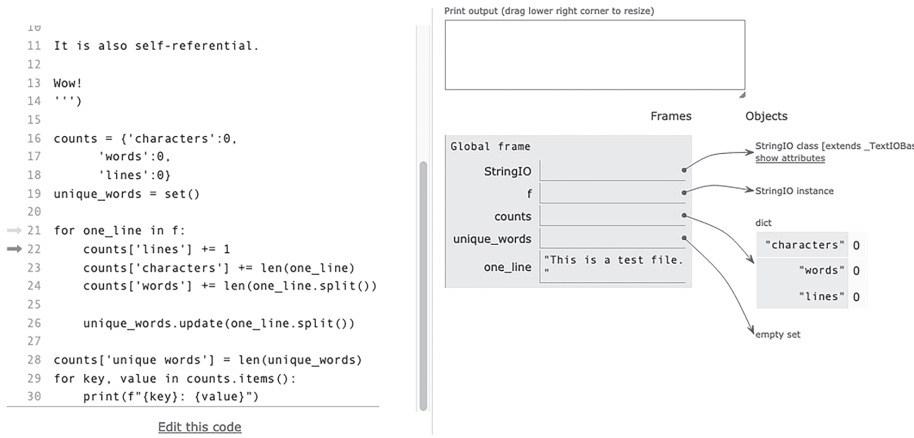


Рисунок 5.2. Инициализированные подсчеты в словаре.

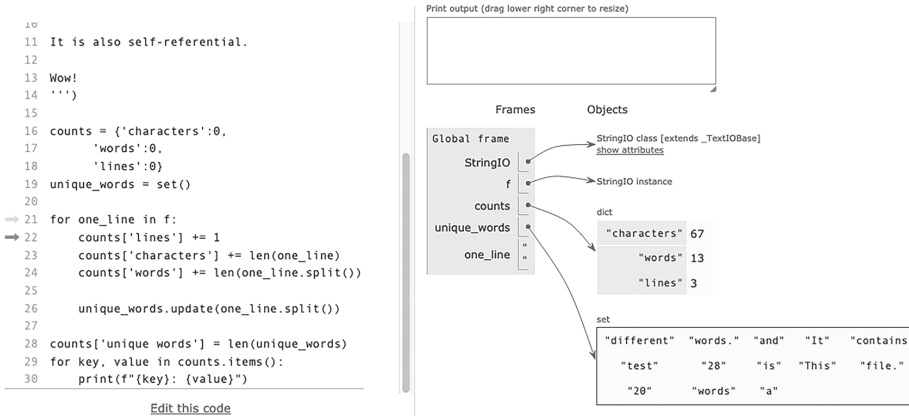


Рисунок 5.3. Структуры данных, включая уникальные слова, после нескольких строк.

Решение

```
def wordcount (filename):
    counts = {'characters': 0,
              'words': 0,
              'lines': 0}
    unique_words = set ()

    for one_line in open (filename):
          counts ['lines'] += 1
          counts ['characters'] += len (one_line)
          counts ['words'] += len (one_line.split ())

          unique_words.update (one_line.split ())

counts ['unique words'] = len (unique_words)
for key, value in counts.items ():
          print (f'{key}: {value}')

wordcount ('wcfile.txt')
```

Вы можете создавать множества с фигурными скобками, но только если они пустые! Используйте set() для создания нового пустого множества.

set.update добавляет все элементы итерируемого объекта во множество.

Подставляет длину множества в counts для комбинированного отчета.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr117].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr118].

После выполнения упражнения

Создание отчетов на основе файлов — распространенное применение Python, использование словарей для сбора информации из этих файлов также распространено. Вот некоторые дополнительные задания, которые вы можете попробовать выполнить:

1. Попросите пользователя ввести имя текстового файла, а затем (в одной строке, разделенной пробелами) слова, частота которых должна быть подсчитана в этом файле. Подсчитайте, сколько раз эти слова встречаются в словаре, используя введенные пользователем слова в качестве ключей, а подсчеты (counts) — в качестве значений.
2. Создайте словарь, в котором ключами будут имена файлов в вашей системе, а значениями — размеры этих файлов. Для вычисления размера можно использовать `os.stat`.
3. В заданном каталоге прочитайте каждый файл и подсчитайте частоту встречаемости каждой буквы. (Сделайте буквы строчными и игнорируйте небуквенные символы.) Используйте словарь для отслеживания частоты букв. Какие пять букв наиболее часто встречаются во всех этих файлах?

Упражнение 21. Самое длинное слово в файле

До сих пор мы работали с отдельными файлами. Однако многие задачи требуют анализа данных в нескольких файлах — например, всех файлов в словаре. Это упражнение поможет вам

попрактиковаться в работе с несколькими файлами, объединяя измерения для всех них.

В этом упражнении напишите две функции. `find_longest_word` принимает в качестве аргумента имя файла и возвращает самое длинное слово, найденное в файле. Вторая функция, `find_all_longest_words`, принимает имя каталога и возвращает словарь, в котором ключами являются имена файлов, а значениями — самые длинные слова из каждого файла.

Если у вас нет текстовых файлов, которые можно использовать для этого упражнения, вы можете загрузить и использовать zip–файл, который я создал из пяти самых популярных книг в Project Gutenberg qr119. Вы можете загрузить zip–файл с сайта qr120.

ПРИМЕЧАНИЕ Существует несколько способов решения данной проблемы. Если вы уже знаете, как использовать генераторы, а особенно генераторы словарей, то это, вероятно, самый питоновский подход. Но если вы еще не освоились с ними и не хотите пока что читать о них в главе 7, то не беспокойтесь — вы можете использовать классический цикл `for`, и все будет в порядке.

Обсуждение

В этом случае вас просят взять имя каталога, а затем найти самое длинное слово в каждом текстовом файле в этом каталоге. Как уже отмечалось, ваша функция должна возвращать словарь, ключами которого являются имена файлов, а значениями словаря — самые длинные слова в каждом файле.

Всякий раз, когда вы слышите, что вам нужно преобразовать коллекцию входных данных в коллекцию выходных, вы должны сразу же подумать о генераторах — чаще всего это генераторы списков, но также полезны генераторы множеств и генераторы словарей. В данном случае мы будем использовать генератор словарей — это означает, что мы создадим словарь на основе

итерации над исходным. В нашем случае источником будет список имен файлов. Имена файлов также будут ключами словаря, а значениями — результаты передачи имен файлов в функцию.

Другими словами, наш генератор словарей будет:

1. Итерировать список файлов в указанном каталоге, поместив имя файла в переменную `filename`.
2. Для каждого файла вызываем функцию `find_longest_word`, передав в качестве аргумента имя файла. Возвращаемым значением будет самая длинная строка в файле.
3. Каждая комбинация слов с самым длинным именем файла станет парой ключ–значение в создаваемом нами словаре.

Как мы можем реализовать `find_longest_word`? Мы можем считать все содержимое файла в строку, превратить эту строку в список, а затем найти самое длинное слово в списке с помощью `sorted`. Хотя это хорошо работает для коротких файлов, для файлов даже среднего размера будет использоваться много памяти.

Мое решение заключается в итерации по каждой строке файла, а затем по каждому слову в строке. Если мы находим слово, которое длиннее, чем текущее `longest_word`, мы заменяем старое слово новым. Когда мы закончим итерировать содержимое файла, мы возвращаем самое длинное слово, которое мы нашли.

Обратите внимание на `os.path.join`, который объединяет имя каталога с именем файла. Вы можете рассматривать `os.path.join` как специфическую версию `str.join` для имен файлов. У него есть и дополнительные преимущества, например, учет текущей операционной системы. В Windows `os.path.join` будет использовать обратные косые черты, в то время как в системах Mac и Unix/Linux он будет использовать прямую косую черту.

Решение

```
import os

def find_longest_word (filename): longest_word = ''
    for one_line in open (filename):
          for one_word in one_line.split ():
```

```
                if len (one_word) > len (longest_
                word):
                     longest_word = one_word
    return longest_word

def find_all_longest_words (dirname):
    return {filename:
         find_longest_word (os.path.join
         (dirname, filename))
         for filename in os.listdir (dirname)
         if os.path.isfile (os.path.join (dirname,
         filename))}

print (find_all_longest_words ('.'))
```

Получает имя файла и полный путь к нему.

Перебирает все файлы в каталоге dirname.

Нас интересуют только файлы, а не каталоги или специальные файлы.

Поскольку эти функции работают с каталогами, ссылка на Python Tutor отсутствует.

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr121].

После выполнения упражнения

Вы часто будете создавать отчеты о файлах и их содержимом с помощью словарей и других базовых структур данных в Python. Вот несколько дополнительных упражнений для дальнейшей отработки:

1. Используйте модуль `hashlib` стандартной библиотеки Python и его функцию `md5`, чтобы вычислить хэш MD5 для содержимого каждого файла в указанном пользователем каталоге. Затем выведите все имена файлов и их MD5-хэши.

2. Запросите у пользователя имя каталога. Покажите все файлы в каталоге, а также то, как давно каталог был изменен. Вы, вероятно, захотите выбрать легкий путь решения задачи и использовать комбинацию `os.stat` и пакета Arrow на PyPI [qr122].

3. Откройте файл журнала HTTP-сервера. (Если у вас его нет, то вы можете воспользоваться следующим файлом [qr123].) Просуммируйте число запросов, которые привели к числовым кодам ответа — 202, 304 и т.д.

Списки каталогов

Для языка, который утверждает, что «есть всего один способ реализации», в Python слишком много способов перечислить файлы в каталоге. Два наиболее распространенных — `os.listdir` и `glob.glob`, оба из которых я уже упоминал в этой главе. Третий способ — использовать `pathlib`, который предоставляет нам объектно-ориентированный API к файловой системе.

Самой простой и стандартной из них является `os.listdir`, функция из модуля `os`. Она возвращает список строк, имена файлов в каталоге: например,

```
filenames = os.listdir ('/etc/')
```

Хорошая новость заключается в том, что понять `os.listdir` и работать с ней очень просто. Плохая новость заключается в том, что она возвращает список имен файлов без имени каталога, что означает, что для открытия или работы с файлами вам придется добавить имя каталога в начале — в идеале с помощью `os.path.join`, который работает кроссплатформенно.

Другая проблема с `os.listdir` заключается в том, что вы не можете фильтровать имена файлов по шаблону. Вы получаете все, включая подкаталоги и скрытые файлы. Поэтому, если вам нужны только все файлы `.txt` в каталоге, `os.listdir` будет недостаточно.

Именно здесь на помощь приходит модуль `glob`. Он позволяет вам использовать шаблоны, иногда называемые `globbing`, для описания нужных вам файлов. Более того, он возвращает список строк, каждая из которых содержит полный путь к файлу. Например, я могу получить полные пути к файлам конфигурации в каталоге `/etc/` на моем компьютере с помощью модуля `glob`:

```
filenames = glob.glob ('/etc/*.conf')
```

В этом случае мне не нужно беспокоиться о других файлах или подкаталогах, что значительно облегчает работу. Долгое время функция `glob.glob` была моей палочкой-выручалочкой для поиска файлов.

Есть еще модуль `pathlib`, который входит в стандартную библиотеку Python и во многом облегчает работу. Вы начинаете с создания объекта `pathlib.Path`, который представляет собой файл или каталог:

```
import pathlib
p = pathlib.Path ('/etc/')
```

Когда у вас есть этот объект Path, вы можете делать с ним множество вещей, для которых раньше требовались отдельные функции, включая те, которые я только что описал. Например, вы можете получить итератор, возвращающий файлы в каталоге с помощью `iterdir`:

```
for one_filename in p.iterdir ():
print (one_filename)
```

В каждой итерации вы получаете не строку, а объект `Path` (точнее, на моем Mac я получаю объект `PosixPath`). Наличие полноценного объекта `Path`, а не строки, позволяет не только выводить имя файла, но и открывать и проверять его.

Если вы хотите получить список файлов, соответствующих шаблону, как я это делал с `glob.glob`, вы можете использовать метод glob:

```
for one_filename in p.glob ('*.conf'):
print (one_filename)
```

`pathlib` — отличное дополнение к последним версиям Python. Если у вас есть возможность использовать его, сделайте это: я обнаружил, что он проясняет и сокращает довольно много моего кода. Хорошее введение в pathlib можно найти здесь: [qr124].

Упражнение 22. Чтение и запись в CSV

В файле CSV каждая запись хранится в одной строке, а поля разделяются запятыми. CSV широко используется для обмена информацией, особенно (но не только) в мире науки о данных. Например, файл CSV может содержать информацию о различных овощах:

```
салат, зеленый, мягкая
морковь, апельсин, твердый
перец, зелень, твердый
баклажан, фиолетовый, мягкий
```

Каждая строка в этом CSV–файле содержит три поля, разделенные запятыми. Здесь нет заголовков, описывающих поля, хотя во многих CSV–файлах они есть.

Иногда запятая заменяется другим символом, чтобы избежать потенциальной двусмысленности. Мне лично больше всего нравится использовать символ TAB (`\t` в строках Python).

Python поставляется с модулем csv, который осуществляет запись в CSV–файлы и чтение из них. Например, вы можете записать в CSV–файл следующий код:

```
import csv

with open ('/tmp/stuff.csv', 'w') as f:
    o = csv.writer (f)
    o.writerow (range (5))
    o.writerow (['a', 'b', 'c', 'd', 'e'])
```

Создает объект csv.writer, обернув в него наш файлоподобный объект f.

Сохраняет этот список строк как запись в файл CSV, разделяя их запятыми.

Записывает в файл целые числа от 0 до 4, разделенные запятыми.

Не все файлы CSV обязательно выглядят как файлы CSV. Например, в стандартном файле Unix `/etc/passwd`, который содержит информацию о пользователях в системе (но не пароли пользователей, несмотря на имя), поля разделяются символами «:».

Для этого упражнения создайте функцию `passwd_to_csv`, которая принимает в качестве аргументов два имени файлов: первое — это файл в стиле `passwd` для чтения, а второе — имя файла, в который нужно записать вывод.

Содержимым нового файла является имя пользователя (индекс 0) и ID пользователя (индекс 2). Обратите внимание, что запись может содержать комментарий, в этом случае она не будет иметь ничего в индексе 2: вы должны принять это во внимание при записи файла. Выходной файл должен использовать символы TAB для разделения элементов. Таким образом, входные данные будут выглядеть следующим образом

```
root:*:0:0::0:0: System Administrator:/var/
root:/bin/sh
daemon:*:1:1::0:0: System Services:/var/root:/
usr/bin/false
```

```
# Это строка комментария
_ftp:*:98:—2::0:0: FTP Daemon:/var/empty:/usr/
 bin/false
```

и вывод будет выглядеть следующим образом:

```
root     0
daemon 1
_ftp     98
```

Обратите внимание, что строка комментария во входном файле не помещается в выходной файл. Можно предположить, что любая строка, содержащая, по крайней мере, два поля, разделенных двоеточием, является допустимой.

Как Python обрабатывает конец строки и новые строки на разных ОС

Различные операционные системы по-разному указывают на то, что мы достигли конца строки. Системы Unix, включая Mac, используют ASCII 10 (перевод строки, или LF). Системы Windows используют два символа, а именно ASCII 13 (возврат каретки, или CR) + ASCII 10. В старых компьютерах Mac используется только ASCII 13.

Python пытается устранить эти пробелы, проявляя гибкость здесь и делая некоторые хорошие подсказки при чтении файлов. Поэтому у меня редко возникали проблемы с использованием Python для чтения текстовых файлов, созданных с помощью Windows. Точно так же мои студенты (которые часто используют Windows) обычно не испытывают проблем с чтением файлов, которые я создал на Mac. Python сам определяет, какое окончание строки используется, поэтому нам не нужно давать никаких дополнительных подсказок. Внутри программы Python окончание строки обозначается символом `\n`.

Запись в файлы, напротив, немного сложнее. Python попытается использовать окончание строки, соответствующее операционной системе. Так, если вы пишете в файл под Windows, он будет использовать CR+LF (иногда показано как `\r\n`). Если вы пишете в файл на машине Unix, то будет использоваться LF.

Обычно это работает просто замечательно. Но иногда при чтении из файла вы можете увидеть слишком много или слишком мало новых строк. Это может означать, что Python угадал неправильно или что в файле используется несколько разных окончаний строк, что сбивает с толку алгоритм угадывания Python.

В таких случаях вы можете передать значение параметра `newline` в функцию `open`, используемую для открытия файлов. Вы можете попробовать явно использовать `newline='\n'` для принудительного перевода строки в стиле Unix или `newline='\r\n'` для принудительного перевода строки в стиле Windows. Если это не решит проблему, возможно, вам придется дополнительно изучить файл, чтобы увидеть, как он был определен.


125

Полное введение в работу с файлами CSV в Python смотрите на странице [qr125].

Обсуждение

Программа решения использует ряд аспектов Python, которые полезны при работе с файлами. Мы уже познакомились с `with` и обсудили его в этой главе. Здесь вы увидите, как можно использовать `with` для открытия двух отдельных файлов или вообще для определения любого количества объектов. Как только наш блок завершится, оба файла будут автоматически закрыты.

В операторе `with` мы определяем две переменные для двух файлов, с которыми мы будем работать. Файл `passwd` открывается для чтения из `/etc/passwd`. Файл output открыт для записи

и записывается в `/tmp/output.csv`. Наша программа будет действовать как промежуточное звено, переводя данные из входного файла и помещая переформатированное подмножество в выходной файл.

Для этого мы создадим один экземпляр `csv.reader`, который будет обернут `passwd`. Однако, поскольку `/etc/passwd` использует двоеточия (:) для разделения полей, мы должны сообщить это `csv.reader`. В противном случае он попытается использовать запятые, что, скорее всего, приведет к ошибке или, что еще хуже, не приведет, несмотря на некорректный парсинг файла. Аналогично мы определяем экземпляр csv.writer, оборачивая наш файл output и указывая, что мы хотим использовать `\t` в качестве разделителя.

Теперь, когда у нас есть объекты для чтения и записи данных CSV, мы можем просмотреть входной файл, записывая строку (линию) в выходной файл для каждого объекта входных данных. Мы берем имя пользователя (из индекса 0) и идентификатор пользователя (из индекса 2), создаем кортеж и передаем этот кортеж в `csv.writerow`. Наш объект `csv.writer` знает, как взять наши поля и вывести их в файл, разделив `\t`.

Возможно, самое сложное здесь — убедиться, что мы не пытаемся преобразовать строки, содержащие комментарии, то есть те, которые начинаются с символа хэша (#). Есть несколько способов сделать это, но здесь я просто проверил количество полей, которые мы получили для текущей строки ввода. Если есть только одно поле, то это должна быть строка комментария или, возможно, другой тип неправильно сформированной строки. В этом случае мы игнорируем строку полностью. Другим хорошим методом будет проверка наличия # в начале строки, возможно, с помощью `str.startswith`.

Решение

```
import csv
def passwd_to_csv (passwd_filename, csv_filename):
    with open (passwd_filename) as passwd,
```

```
? open (csv_filename, 'w') as output:
infile = csv.reader (passwd,
          delimiter=':')
     outfile = csv.writer (output,
          delimiter='\t')
for record in infile:
     if len (record) > 1:
          outfile.writerow ((record [0],
          record [2]))
```

Поля во входном файле разделяются двоеточиями («:»).

Поля в выходном файле разделяются табуляторами («\t»).

Поскольку мы не можем записывать в файлы в Python Tutor, для этого упражнения нет ссылки.

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr126].

После выполнения упражнения

Файлы CSV чрезвычайно полезны и распространены, и модуль csv, поставляемый вместе с Python, отлично с ними работает. Если вам нужно что–то более продвинутое, то вам стоит обратить внимание на `pandas`, который работает с большим числом вариаций CSV, а также со многими другими форматами.

Вот несколько дополнительных упражнений для улучшения навыков работы с файлами CSV:

1. Расширьте это упражнение, попросив пользователя ввести разделенный пробелами список целых чисел, указывающих, какие поля должны быть записаны в выходной CSV–файл. Также спросите пользователя, какой символ должен использоваться в качестве разделителя в выходном файле. Затем считайте данные из /etc/passwd, записывая выбранные пользователем поля, разделенные выбранным пользователем разделителем.

2. Напишите функцию, которая записывает словарь в CSV–файл. Каждая строка CSV–файла должна содержать три поля: (1) ключ, который мы будем считать строкой, (2) значение и (3) тип значения (например, str или int).
3. Создайте CSV–файл, в котором каждая строка содержит 10 случайных целых чисел от 10 до 100. Теперь считайте файл и выведите сумму и среднее значение чисел в каждой строке.

Упражнение 23. JSON

JSON — это популярный формат для обмена данными. В частности, многие веб–службы и API отправляют и получают данные с помощью JSON.

Данные в кодировке JSON могут быть прочитаны в очень большом количестве языков программирования, включая Python. В стандартную библиотеку Python входит модуль `json`, который можно использовать для преобразования строк в кодировке JSON в объекты Python и наоборот. Метод `json.load` считывает строку в кодировке JSON из файла и возвращает комбинацию объектов Python.

В этом упражнении вы будете анализировать данные тестов в средней школе. В файловой системе содержится каталог `scores`, содержащий несколько файлов в формате JSON. Каждый файл содержит оценки для одного класса. Напишите функцию `print_scores`, которая принимает в качестве аргумента имя каталога и печатает в кратком виде найденные оценки учащихся.

Если вы пытаетесь проанализировать оценки из класса 9a, они будут находиться в файле под названием `9a.json`, который выглядит следующим образом:

```
[{"математика": 90, "литература": 98, "естество-
знание": 97},
{"математика": 65, "литература": 79, "естество-
знание": 85},
```

```
{“математика”: 78, “литература”: 83, “ естество-
знание”: 75},
{“математика”: 92, “литература”: 78, “ естество-
знание”: 85},
{“математика”: 100, “литература”: 80, “ естество-
знание”: 90}]
```

В каталоге также могут содержаться файлы для 10–го класса (`10a.json`, `10b.json` и `10c.json`), и других классов, и классов средней школы. Каждый файл содержит JSON–эквивалент списка словарей, причем каждый словарь содержит баллы по нескольким различным школьным предметам.

ПРИМЕЧАНИЕ В правильном JSON используются двойные кавычки (“), а не одинарные кавычки (‘). Это может удивить и расстроить разработчиков Python.

Ваша функция должна вывести наивысшие, наименьшие и средние тестовые баллы по каждому предмету в каждом классе. Для файлов (`9a.json` и `9b.json`) в каталоге scores мы получим следующий результат:

```
scores/9a.json
естествознание: минимум 75, максимум 97, сред-
нее 86,4
литература: минимум 78, максимум 98, среднее 83,6
математика: минимум 65, максимум 100, среднее 85.0
scores/9b.json
естествознание: минимум 35, максимум 95, в сред-
нем 82,0
литература: минимум 38, максимум 98, среднее 72.0
математика: минимум 38, максимум 100, среднее 77,0
```

Вы можете скачать zip–файл с этими файлами JSON по ссылке [qr127].

Обсуждение

Во многих языках первым ответом на подобную проблему было бы: «Давайте создадим собственный класс!» Но в Python, хотя мы можем (и часто так и делаем) создавать собственные классы, зачастую проще и быстрее использовать встроенные структуры данных: списки, кортежи и словари.

В данном конкретном случае мы читаем из файла JSON. JSON — это представление данных, как и XML, сам по себе он не является типом данных. Таким образом, если мы хотим создать JSON, мы должны использовать модуль `json`, чтобы превратить наши данные Python в строки в формате JSON. А если мы хотим читать из файла JSON, мы должны прочитать содержимое файла в виде строк в нашей программе, а затем преобразовать его в структуры данных Python.

Однако в этом упражнении вас попросят поработать с несколькими файлами в одном каталоге. Мы знаем, что каталог называется `scores` и что все файлы имеют суффикс `.json`. Поэтому мы можем использовать `os.listdir` для каталога, фильтруя (возможно, с помощью генератора списка) все эти имена файлов так, чтобы работать только с теми, которые заканчиваются на `.json`.

Тем неменее, кажется более подходящим использовать `glob`, который принимает шаблон имени файла в стиле Unix, содержащий (среди прочих) символы * и? и возвращает список имен файлов, соответствующих шаблону. Таким образом, вызвав `glob.glob ('scores/*.json')`, мы получим все файлы, заканчивающиеся на `.json` в пределах каталога `scores`. Затем мы можем перебирать этот список, присваивая текущее имя файла (строку) `filename`.

Затем мы создадим новую запись в словаре `scores`, где будут храниться оценки. На самом деле это будет словарь словарей, в котором на первом уровне будет имя файла и, следовательно, класса, из которого мы будем считывать данные. Ключами второго уровня будут предметы, значениями словаря — список оценок, на основе которых мы сможем вычислить нужную нам ста-

тистику. Таким образом, как только мы определили `filename`, мы сразу же добавляем имя файла в качестве ключа к `scores`, а в качестве значения — новый пустой словарь.

Иногда вам нужно прочитать каждую строку файла в Python, а затем вызвать `json.loads`, чтобы превратить эту строку в данные. В нашем случае, однако, файл содержит один массив JSON. Поэтому мы должны использовать `json.load` для чтения из файлового объекта `infile`, который превращает содержимое файла в список словарей Python.

Поскольку `json.load` возвращает список словарей, мы сможем проитерировать его. Каждый результат теста помещается в переменную `result`, которая представляет собой словарь, где ключами являются предметы, а значениями — баллы. Наша цель — выявить некоторую статистику по каждому из предметов в классе, что означает, что в то время, как входной файл сообщает о баллах каждого ученика, наш отчет будет игнорировать учеников в пользу предметов.

Учитывая, что `result` является словарем, мы можем перебирать пары ключ–значение с помощью `result.items ()`, используя параллельное присваивание для итерации по ключу и значению (здесь они называются `subject` (предмет) и `score` (оценка)). Теперь мы не знаем заранее, какие предметы будут в нашем файле, и не знаем, сколько будет тестов. Исходя из этого, нам проще всего хранить баллы в списке. Это означает, что в нашем словаре `scores` будет один ключ верхнего уровня для каждого имени файла и один ключ второго уровня для каждого предмета. Значение второго уровня будет списком, который мы будем добавлять с каждой итерацией через распарсенный JSON–список.

Мы хотим добавить оценку в список:

```
scores [filename] [subject]
```

Прежде чем это сделать, нам нужно убедиться, что список существует. Один из простых способов сделать — это использовать `dict.setdefault`, который назначает пару ключ–значе–

ние в словарь, но только если ключ еще не существует. Другими словами, `d.setdefault (k, v)` — это то же самое, что и

```
if k not in d:
d [k] = v
```

Мы используем `dict.setdefault` для создания списка, если он еще не существует. В следующей строке мы добавляем оценку в список для этого предмета в этом классе.

Когда мы завершили наш начальный цикл `for`, у нас есть все оценки для каждого класса. Затем мы можем перебирать каждый класс, печатая имя класса.

Затем мы итерируем каждый предмет класса. Мы снова используем метод `dict.items`, чтобы вернуть пару ключ–значение — в данном случае мы называем их `subject` (для названия класса) и `subject_scores` (для списка оценок по этому предмету). Затем мы используем f–строку для получения вывода, используя встроенные `min` и `max`, а затем комбинируя функции `sum` и `len`, чтобы получить средний балл.

Хотя эта программа считывает данные из файла, содержащего JSON, и выводит их на экран пользователя, она может с таким же успехом считывать данные из сетевого соединения в формате JSON, и/или записывать их в файл или сокет в формате JSON. Пока мы используем встроенные и стандартные структуры данных Python, модуль `json` сможет принимать наши данные и превращать их в JSON.

Решение

```
import json
import glob

def print_scores (dirname):
   scores = {}
for filename in glob.glob (f'{dirname} /*.json'):
             scores [filename] = {}
```

Читает из файла infile и преобразовывает из JSON в объект Python.

```
with open (filename) as infile:
     for result in json.load (infile):
          for subject, score in result.items ():
               scores [filename].setdefault
               (subject, [])
               scores [filename] [subject].
               append (score)
```

Убедитесь, что subject существует как ключ в scores[filename].

```
for one_class in scores:
        print (one_class)
        for subject, subject_scores in scores
        [one_class].items ():
             min_score = min (subject_scores)
             max_score = max (subject_scores)
             average_score = (sum (subject_
             scores) /
                  len (subject_scores))

             print (subject)
             print (f'\tmin {min_score}')
             print (f'\tmax {max_score}')
             print (f'\taverage {average_
             score}')
```

Суммирует баллы.

Поскольку эти функции работают с каталогами, ссылка на Python Tutor отсутствует.

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr128].

После выполнения упражнения

Вот еще несколько задач, которые вы можете попробовать выполнить с использованием JSON:

1. Преобразовать файл `/etc/passwd` из формата CSV в JSON. Файл JSON будет содержать эквивалент списка кортежей Python, причем каждый кортеж будет представлять одну строку из файла.
2. Для решения несколько иной задачи превратите каждую строку в файле в словарь Python. Для этого потребуется идентифицировать каждое поле с уникальным именем столбца или ключа. Если вы не уверены, что делает каждое поле в `/etc/passwd`, вы можете выбрать ему произвольное имя.
3. Спросите у пользователя имя каталога. Переберите все файлы в этом каталоге (игнорируя подкаталоги), получите (через `os.stat`) размер файла и время его последнего изменения. Создайте на диске файл в формате JSON, содержащий имя каждого файла, его размер и временную метку модификации. Затем снова прочитайте файл и определите, какие файлы были изменены чаще всего и наименее недавно, а также какие файлы являются самыми большими и самыми маленькими в этом каталоге.

Упражнение 24. Переворачиваем строки

Во многих случаях мы хотим взять файл в одном формате и сохранить его в другом формате. В этой функции мы реализуем базовую версию этой идеи. Функция принимает два аргумента: имена входного файла (который будет считан из файла) и выходного файла (который будет создан). Например, если файл выглядит следующим образом:

```
abc def
ghi jkl
```

то выходной файл будет иметь вид

```
fed cba
lkj ihg
```

Обратите внимание, что новая строка остается в конце строки, а все остальные символы меняются местами.

Преобразование файлов из одного формата в другой и получение данных из одного файла и создание на их основе другого — обычные задачи. Например, вам может понадобиться перевести даты в другой формат, перевести временные метки из восточного летнего времени в среднее время по Гринвичу или преобразовать цены из евро в доллары. Также может потребоваться извлечь из входного файла только некоторые данные, например, для определенной даты или местоположения.

Обсуждение

Это решение зависит не только от того, что мы можем итерировать файл по одной строке за раз, но и от того, что мы можем работать с более чем одним объектом в операторе `with`. Помните, что оператор `with` принимает один или несколько объектов и позволяет присваивать им переменные. Мне особенно нравится, что, когда я хочу прочитать из одного файла и записать в другой, я могу просто использовать `with`, чтобы открыть один файл для чтения, второй — для записи, а затем сделать то, что я показал здесь.

Затем я читаю каждую строку входного файла. Далее переворачиваю строку, используя синтаксис срезов Python, — помните, что `s [::-1]` означает, что нам нужны все элементы `s`, от начала до конца, но я использую размер шага –1, что возвращает обратную версию строки.

Однако перед тем, как перевернуть строку, мы сначала хотим удалить символ новой строки, который является последним символом в строке. Поэтому мы сначала выполняем `str.rstrip()` на текущей строке, а затем переворачиваем ее. Затем мы записываем ее в выходной файл, добавляя символ новой строки, чтобы фактически спуститься на одну строку.

Использование `with` гарантирует, что оба файла будут закрыты по завершении блока. Когда мы закрываем файл, открытый для записи, он автоматически очищается, что означает, что

нам не нужно беспокоиться о том, были ли данные действительно сохранены на диск.

Следует отметить, что люди часто спрашивают меня, как читать из одного и того же файла и записывать в него. Python поддерживает это, используя режим `r+`. Но я считаю, что это открывает двери для многих потенциальных проблем из-за возможности перезаписать неправильный символ и тем самым испортить формат редактируемого файла. Я предлагаю использовать код по типу «читать из одного, записывать в другой», который дает примерно тот же эффект, но без потенциальной опасности испортить исходный файл.

Решение

```
def reverse_lines (infilename, outfilename):
    with open (infilename) as infile, open
    (outfilename, 'w') as outfile:
         for one_line in infile:
              outfile.write (f'{one_line.rstrip
              () [::-1]} \n')
```

str.rstrip удаляет все пробельные символы из правой части строки.

Поскольку эти функции работают с каталогами, ссылка на Python Tutor отсутствует.

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr129]

После выполнения упражнения

Вот еще несколько идей для упражнений по переводу файлов из одного формата в другой с использованием такого метода и `with`:

1. «Зашифруйте» текстовый файл, превратив все его символы в их числовые эквиваленты (с помощью встроенной функ-

ции `ord`) и записав этот файл на диск. Теперь «расшифруйте» файл (с помощью встроенной функции `chr`), превратив числа обратно в исходные символы.

2. На основе существующего текстового файла создайте два новых текстовых файла. Каждый из новых файлов будет содержать столько же строк, сколько и входной файл. В один выходной файл вы запишите все гласные (a, e, i, o и u) из входного файла. В другом — все согласные. (Вы можете игнорировать пунктуацию и пробелы.)
3. Последнее поле в `/etc/passwd` — это *shell*, командный интерпретатор Unix, который вызывается при входе пользователя в систему. Создайте файл, содержащий одну строку для каждой оболочки, в которой будет записано имя оболочки, а затем все имена пользователей, которые используют эту оболочку, например:

```
/bin/bash: root, jci, user, reuven, atara
/bin/sh: spamd, gitlab
```

Подводя итоги

Практически невозможно представить себе написание программ без использования файлов. И хотя существует множество различных типов файлов, Python особенно хорошо подходит для работы с текстовыми файлами — особенно, но не только, включая файлы журналов и конфигурационные файлы, а также файлы, отформатированные такими стандартными способами, как JSON и CSV.

При работе с файлами важно помнить несколько вещей:

1. Обычно вы открываете файлы для чтения или записи.
2. Вы можете (и должны) итерировать файлы по одной строке за раз, а не считывать в память сразу весь файл.
3. Использование `with` при открытии файла для записи гарантирует, что файл будет очищен и закрыт.
4. Модуль csv позволяет легко читать из файлов CSV и записывать в них.

5. Функции `dump` и `load` модуля `json` позволяют нам перемещаться между структурами данных Python и строками в формате JSON.
6. Чтение из файлов во встроенные типы данных Python — распространенная и мощная техника.

6. Функции

Функции являются одним из важнейших элементов программирования — но не потому, что в них есть техническая необходимость. Мы могли бы программировать и без функций, если бы это было действительно необходимо. Но функции дают ряд больших преимуществ.

Во-первых, они позволяют нам избежать повторений в нашем коде. Во многих программах есть инструкции, которые повторяются: например, запрос пользователя на вход в систему, чтение данных из определенного типа конфигурационного файла или вычисление длины MP3. Хотя компьютер не будет возражать (или даже жаловаться), если один и тот же код появится в нескольких местах, мы и люди, которым придется поддерживать код после того, как мы закончим работу с ним, будут страдать и, скорее всего, жаловаться. Такое повторение трудно запомнить и отследить. Более того, вы, скорее всего, обнаружите, что код нуждается в улучшении и обслуживании: если он встречается в вашей программе несколько раз, то вам придется находить и исправлять его каждый раз.

Как уже упоминалось в главе 2, при программировании хорошо помнить о фразе «не повторяйся» (DRY). А написание функций — это отличный способ применить правило «избавь код от повторов».

Второе преимущество функций заключается в том, что они позволяют нам (как разработчикам) мыслить на более высоком уровне абстракции. Как нельзя водить машину, если постоянно думать о том, что делают различные части вашего автомобиля,

так и нельзя программировать, если постоянно думать обо всех частях вашей программы и о том, что они делают. С семантической и когнитивной точки зрения мы оборачиваем функциональность в пакет с именем, а затем используем это имя для ссылки на него.

В естественном языке мы постоянно создаем новые глаголы, например, «программировать» и «писать». Конечно, мы могли бы описать эти действия с помощью гораздо большего количества слов и с гораздо большим количеством деталей, но это утомительно и отвлекает от работы.

Функции — это глаголы программирования. Они позволяют нам определять новые действия на основе старых и, таким образом, позволяют нам мыслить более сложными терминами.

По всем этим причинам функции являются полезным инструментом и доступны во всех языках программирования. Но функции Python являются особенными еще и потому, что они являются объектами, то есть с ними можно обращаться как с данными. Мы можем хранить функции в структурах данных и извлекать их оттуда. Использование функций таким образом кажется странным многим новичкам в Python, но это мощная техника, которая позволяет сократить объем кода и повысить гибкость.

Более того, Python не допускает многократного определения одной и той же функции. В некоторых языках функцию можно определить несколько раз, каждый раз с разной сигнатурой. Например, можно один раз определить функцию, принимающую в качестве аргумента строку, второй раз как принимающую в качестве аргумента — список, третий раз как принимающую — словарь, а четвертый раз как принимающую три аргумента типа `float`.

В Python такой функциональности не существует: когда вы определяете функцию, вы присваиваете переменную. И так же, как вы не ждете, что x будет одновременно содержать значения 5 и 7, вы точно так же не можете ожидать, что функция будет содержать несколько реализаций.

В Python эта проблема решается с помощью гибких параметров. Между значениями по умолчанию, переменным количе-

ством аргументов (`*args`) и аргументами с ключевыми словами (`**kwargs`) мы можем писать функции, которые справляются с различными ситуациями.

По мере изучения этой книги вы уже написали несколько функций, поэтому целью данной главы не является научить вас писать функции. Скорее, цель состоит в том, чтобы показать, как использовать различные техники, связанные с функциями. Это позволит вам не только написать код один раз и использовать его множество раз, но и создать иерархию новых глаголов, описывающих все более сложные и высокоуровневые задачи.

Таблица 6.1. Что вам нужно знать

Понятие	Что это?	Пример	Чтобы узнать подробнее
`def`	Ключевое слово для определения функций и методов.	`def double (x): return x * 2`	
`глобальное пространство`	В функции указывает на то, что переменная должна быть глобальной.	`global x`	
`nonlocal`	Во вложенной функции указывает, что переменная является локальной для внешней функции.	`nonlocal`	
модуль `operator`	Коллекция методов, реализующих встроенные операторы.	`operator. add (2,4)`	

Значения параметров по умолчанию

Допустим, я могу написать простую функцию, которая возвращает дружеское приветствие:

```
def hello (name):
   return f'Hello, {name}!'
```

Это будет работать нормально, если я предоставлю значение для name:

```
>>> hello ('world')
'Hello, world!'
```

Но что, если нет?

```
>>> hello ()
   Traceback (последний вызов):
        Файл "<stdin>", линия 1, в <module>
        TypeError: hello () отсутствует 1 обя-
        зательный позиционный аргумент: 'name'
```

Другими словами, Python знает, что функция принимает один аргумент. Поэтому если вы вызовете функцию с одним аргументом, все будет в порядке. Вызовите ее без аргументов (или с двумя аргументами, если на то пошло) и получите сообщение об ошибке.

Откуда Python знает, сколько аргументов должна принять функция? Он знает, потому что объект функции, который мы создали, когда определяли функцию с помощью `def`, отслеживает такие вещи. Вместо того чтобы вызывать функцию, мы можем заглянуть внутрь объекта функции. Атрибут `__code__` (см. рисунок 6.1) содержит ядро функции, включая байткоды, в которые была скомпилирована ваша функция. Внутри этого объекта находится ряд подсказок, которые Python хранит в себе, включая эту:

```
>>> hello.__code__.co_argcount
1
```

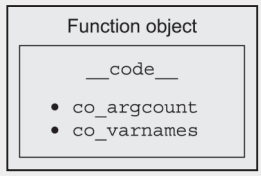


Рисунок 6.1. Объект функции вместе с `__code__`.

Другими словами, когда мы определяем нашу функцию с параметром, объект функции отслеживает его в `co_argcount`. И когда мы вызываем функцию, Python сравнивает количество аргументов с `co_argcount`. Если есть несоответствие, то мы получаем ошибку, как мы видели чуть раньше. Однако есть еще способ определить функцию так, чтобы аргумент был необязательным: мы можем добавить к параметру значение по умолчанию:

```
def hello (name='world'):
return f'Hello, {name}!'
```

Сейчас Python будет давать нам больше свободы действий. Если мы передаем аргумент, то это значение присваивается параметру `name`. Но если мы не передаем аргумент, тогда строке `world` присваивается значение name, как было по умолчанию (см. таблицу 6.2). Таким образом, мы можем вызывать нашу функцию либо без аргументов, либо с одним аргентом — при этом два аргумента недопустимы.

Таблица 6.2. Вызов hello

Вызываем	Значение name	Возвращаемое знаечние
`hello ()`	`world`, спасибо значению по умолчанию.	`Hello, world!`
`hello ('out there')`	`out there.`	`Hello, out there!`
`hello ('a', 'b')`	Ошибка: слишком много аргументов.	Нет возвращаемого значения.

ПРИМЕЧАНИЕ Параметры со значениями по умолчанию должны идти после параметров без значений по умолчанию.

ПРЕДУПРЕЖДЕНИЕ Никогда не используйте в качестве значения по умолчанию параметра изменяемое значение, например, список или словарь, поскольку значения по умолчанию сохраняются и повторно используются при каждом вызове функции. Это означает, что, если вы измените значение по умолчанию в одном вызове, это изменение будет и в следующем вызове. Большинство анализаторов кода и IDE предупредят вас, но об этом нужно помнить.

Упражнение 25. Генератор XML

Python часто используют не только для парсинга данных, но и для форматирования. В этом упражнении вы напишете функцию, которая использует комбинацию различных параметров и типов параметров для получения различных выходных данных.

Напишите функцию `myxml`, которая позволяет создавать простой вывод XML. Вывод функции всегда будет строкой. Функция может быть вызвана несколькими способами, как показано в таблице 6.3.

Таблица 6.3. Вызов myxml

Вызываем	Возвращаемое знаечние
`myxml ('foo')`	`<foo></foo>`
`myxml ('foo', 'bar')`	`<foo>bar</foo>`
`myxml ('foo', 'bar', a=1, b=2, c=3)`	`<foo a="1" b="2" c="3">bar</foo>`

Обратите внимание, что во всех случаях первым аргументом является имя тега. В двух последних случаях вторым аргументом является содержимое (текст), расположенное между открываю–

щим и закрывающим тегами. В третьем случае, пара имя–значения будет превращена в атрибуты внутри открывающего тега.

Обсуждение

Предположим, что наша функция принимает только один аргумент — имя тега. Это достаточно легко записать. Мы можем написать

```
def myxml (tagname): return
    f'< {tagname} ></ {tagname} >'
```

Если мы захотим передать второй (необязательный) аргумент, то потерпим поражение. Поэтому некоторые люди полагают, что наша функция должна принимать `*args`, то есть любое количество аргументов, помещенных в кортеж. Как правило, `*args` предназначен для ситуаций, в которых вы не знаете, сколько значений получите, и хотите принимать любое число.

Я обычно руководствуюсь следующим правилом: `*args` следует использовать тогда, когда вы собираетесь поместить значение в цикл `for`, и что если вы берете элементы из `*args` с числовыми индексами, то, вероятно, вы делаете что–то не так.

Другой вариант — использовать значение по умолчанию. Я выбрал последний вариант. Первый параметр является обязательным, а второй — необязательным. Если я сделаю второй параметр (который я здесь назвал `content`) пустой строкой, то я буду знать, что либо пользователь передает `content`, либо `content` пуст. В любом случае функция работает. Таким образом, я могу определить ее следующим образом:

```
def myxml (tagname, content=''): return
    f'< {tagname} > {content} </ {tagname} >'
```

Но как насчет пар ключ–значение, которые мы можем передать и которые затем помещаются в качестве атрибутов в открывающий тег?

Когда мы определяем функцию с `**kwargs`, мы говорим Python, что можем передать любую пару имя–значение в стиле `name=value`. Данные аргументы не передаются как обычно, а рассматриваются отдельно как *аргументы ключевого* слова. Они используются для создания словаря, который обычно обозначают как `kwargs`, ключами которого являются имена ключевых слов, а значениями — значения ключевых слов. Таким образом, мы можем написать:

```
def myxml (tagname, content='', **kwargs):
    attrs = ''.join ([f' {key} ="{value}"'
    for key, value in kwargs.items ()])
    return f'< {tagname} {attrs} > {content} </
    {tagname} >'
```

Как вы видите, я не просто беру пары ключ–значение из `**kwargs` и помещаю их в строку. Сначала я должен взять этот словарь и превратить его в пару имя–значение в формате XML. Я делаю это при помощи генератора списков, запущенного для словаря. Для каждой пары ключ–значение я создаю строку, следя за тем, чтобы первый символ в строке был пробелом, чтобы не «столкнуться» с `tagname` в открывающем теге.

В этом коде используется несколько общих парадигм Python. Поэтому давайте пройдемся по коду шаг за шагом, чтобы все стало понятнее:

1. `tagname` будет строкой (именем тега) в теле `myxml`, `content` будет строкой (чтобы ни находилось между тегами), а `kwargs` будет словарем (с парами имя–значение атрибутов).
2. И `content`, и `kwargs` могут быть пустыми, если пользователь не передал никаких значений для этих параметров.
3. Мы используем генератор списка для перебора элементов `kwargs.items ()`. Так мы сможем получить одну пару ключ–значение для каждой итерации.
4. Мы используем пару ключ–значение, присвоенную переменным `key` и `value`, для создания строки вида

key="value". Мы получаем одну такую строку для каждой пары ключ–значение, переданной пользователем.

5 Результатом генератора списка будет список строк. Мы объединяем эти строки с помощью str.join, при этом между элементами будет пустая строка.

6 Наконец, мы возвращаем комбинацию открывающего тега (с любыми атрибутами, которые мы могли получить), содержимого и закрывающего тега.

Решение

Функция содержит один обязательный параметр, один по умолчанию и **kwargs.

```
def myxml (tagname, content='', **kwargs):
    attrs = ''.join ([f' {key} ="{value}"'
          for key, value in kwargs.items ()])
    return f'< {tagname} {attrs} > {content} </
    {tagname} >'
```

Возвращаем строку в формате XML.

Используем генератор списка для создания строки из kwargs.

```
print (myxml ('tagname', 'hello', a=1, b=2, c=3))
```

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr134].

134

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr135].

135

После выполнения упражнения

Обучение работе с функциями и типами параметров, которые вы можете определять, занимает некоторое время, но оно того стоит. Вот несколько упражнений, которые помогут вам потренироваться работать с параметрами функций:

1. Напишите функцию `copyfile`, которая принимает один обязательный аргумент — имя входного файла — и любое количество дополнительных аргументов: имена файлов, в которые следует скопировать входные данные. Вызов `copyfile ('myfile.txt', 'copy1.txt', 'copy2.txt', 'copy3.txt')` создаст три копии `myfile.txt`: по одной в каждом файле `copy1.txt`, `copy2.txt` и `copy3.txt`.
2. Напишите функцию `factorial`, которая принимает любое количество числовых аргументов и возвращает результат умножения их всех друг на друга.
3. Напишите функцию `anyjoin`, которая работает аналогично `str.join`, за исключением того, что первым аргументом является последовательность любых типов (не только строк), а вторым аргументом по умолчанию будет " " (пробел), своеобразный «клей», который мы помещаем между элементами. Поэтому `anyjoin ([1,2,3])` вернет `1 2 3`, а `anyjoin ('abc', pass:'**')` вернет pass: `a**b**c`.

Область видимости переменных в Python

Область видимости переменных — это одна из тех тем, которые многие предпочитают игнорировать — сначала потому, что это скучно, а затем потому, что это очевидно. Дело в том, что определение области видимости переменных в Python сильно отличается от того, что я видел в других языках. Более того, это многое объясняет о том, как работает язык, и почему были приняты определенные решения.

Термин область видимости относится к переменным и всем именам внутри программы. Если я установлю значение переменной внутри функции, повлияю ли я на нее и за пределами функции? Что, если я устанавливаю значение переменной внутри цикла `for`? В Python есть четыре уровня видимости переменной:

1. Локальная область функции.

2. Область внешней функции.
3. Глобальное пространство.
4. Встроенная область видимости.

Они известны под аббревиатурой LEGB. Если вы находитесь в функции, то поиск будет осуществляться по всем четырем по порядку. Если вы находитесь вне функции, то поиск ведется только в двух последних (глобальном и встроенном). Как только идентификатор найден, Python прекращает поиск.

Это важное примечание, о котором следует помнить. Если вы не определили функцию, вы работаете на глобальном уровне. Отступы встречаются повсеместно в Python, но они совершенно не влияют на определение области видимости переменной.

Но что, если вы напишите `int ('s')`? Является ли `int` глобальной переменной? Нет, она находится в пространстве имен встроенных функций. В Python очень мало зарезервированных слов, многие из наиболее распространенных типов и функций, которые мы используем, не являются ни глобальными, ни зарезервированными ключевыми словами. Python выполняет поиск во встроенном пространстве имен после глобального, прежде чем сдаться и выдать исключение.

Что, если вы определите глобальное имя, которое идентично одному из встроенных? Тогда вы эффективно скроете это значение. Я постоянно вижу это на своих курсах, когда люди пишут что-то вроде:

```
sum = 0
for i in range (5):
   sum += i
print (sum)

print (sum ([10, 20, 30]))

TypeError: объект 'int' не является вызываемым
```

Почему мы получаем эту странную ошибку? Потому что в дополнение к функции sum, заданной во встроенном пространстве имен, мы также определяем глобальную переменную с именем `sum`. А поскольку глобальные переменные идут перед встроенными в пути поиска Python, Python находит, что `sum` является целым числом, и отказывается вызывать эту функцию.

Немного расстраивает тот факт, что язык не проверяет и не предупреждает вас о переделке имен во встроенных модулях. Однако существуют инструменты (например, `pylint`), которые подскажут вам, если вы случайно (или нет) создали несовместимое имя.

Локальные переменные

Если я определяю переменную внутри функции, то она считается *локальной* переменной. Локальные переменные существуют только до тех пор, пока существует функция: когда функция исчезает, исчезают и определенные ею локальные переменные, например:

```
x = 100

def foo ():
    x = 200
    print (x)

print (x)
foo ()
print (x)
```

Этот код выведет 100, 200 и затем еще раз 100. В коде мы определили две переменные: `x`, расположенная в глобальной области видимости и равная 100, и никогда не будет

меняться, и `x` в локальной области видимости, доступная только внутри функции `foo`, ее значение равно 200 и также не меняется. Тот факт, что обе переменные называются `x`, не вызовет противоречий в Python, поскольку внутри функции он будет видеть локальную `x` и полностью игнорировать глобальную.

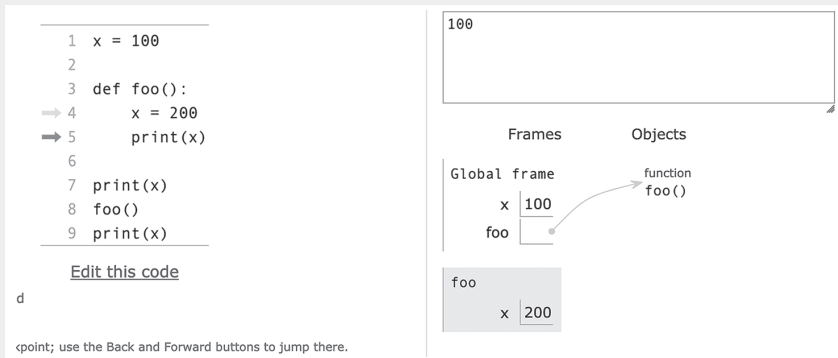


Рисунок 6.2 x внутри vs. вовне.

Ключевое слово `global`

Что, если внутри функции я хочу изменить глобальную переменную? Для этого необходимо использовать ключевое слово `global`, которое сообщает Python, что вы не заинтересованы в создании локальной переменной в этой функции. Скорее, любые извлечения или присваивания должны влиять на глобальную переменную, например:

```
x = 100

def foo ():
    global x
    x = 200
    print (x)

print (x)
```

```
foo ()
print (x)
```

Этот код выведет 100, 200, а затем 200, потому что есть только один `x` благодаря ключевому слову `global`.

Изменение глобальных переменных внутри функции — почти всегда плохая идея. Однако в редких случаях это необходимо. Например, вам может понадобиться обновить параметр конфигурации, который установлен как глобальная переменная.

Область внешней функции

Наконец, рассмотрим работу внутренних функций на примере следующего кода:

```
def foo (x):
    def bar (y):
        return x * y
    return bar

f = foo (10)
print (f (20))
```

Уже сейчас этот код кажется немного странным. Что происходит, когда мы определяем bar внутри `foo`? Данная внутренняя функция, иногда называемая замыканием, является функцией, которая определяется при выполнении `foo`. Действительно, каждый раз, когда мы вызываем `foo`, мы получаем обратно новую функцию с именем `bar`. Но, конечно, `bar` — это имя локальной переменной внутри `foo`: мы можем называть возвращаемую функцию как угодно.

Когда мы выполняем код, результат равен 200. Логично, что когда мы вызываем `f`, то мы выполняем `bar`, который

`foo` вернула. И мы можем понять, почему `bar` имеет доступ к `y`, поскольку это локальная переменная. Но что насчет `x`? Почему функция bar имеет доступ к `x`, локальной переменной в `foo`?

Конечно, благодаря LEGB:

1. Сначала Python ищет x локально, в локальной функции bar.
2. Далее Python ищет x во внешней функции foo.
3. Если бы x не было в foo, то Python продолжил бы поиск на глобальном уровне.
4. И если бы x не был глобальной переменной, то Python искал бы x во встроенном пространстве имен.

Что, если я хочу изменить значение `x`, локальной переменной, расположенной во внешней функции? Она не является глобальной, поэтому ключевое слово `global` не будет работать. В Python 3, однако, у нас есть ключевое слово nonlocal. Это ключевое слово сообщает Python: «Любое присваивание, которое мы делаем этой переменной, должно идти во внешнюю функцию, а не в (новую) локальную переменную», например:

```
def foo ():
    call_counter = 0
    def bar (y):
        nonlocal call_counter
        call_counter += 1
        return f'y = {y}, call_counter =
        {call_counter}'
    return bar
b = foo ()

for i in range (10, 100, 10):
        print (b (i))
```

Инициализирует call_counter как локальную переменную в foo.

Сообщает bar, что присваивание call_counter влияет на переменную в foo.

Увеличивает call_counter, значение которой сохраняется во всех прогонах bar.

Итерируем числа 10, 20, 30, ... 90. Вызываем b для каждого числа в данном диапазоне.

Вывод будет выглядеть следующим образом:

```
y = 10, call_counter = 1
y = 20, call_counter = 2
y = 30, call_counter = 3
y = 40, call_counter = 4
y = 50, call_counter = 5
y = 60, call_counter = 6
y = 70, call_counter = 7
y = 80, call_counter = 8
y = 90, call_counter = 9
```

Поэтому всякий раз, когда вы увидите, как Python обращается к переменной или устанавливает ее — а это бывает часто — вспомните о правиле определения области видимости LEGB и о том, что оно всегда, без исключения, используется для поиска всех идентификаторов, включая данные, функции, классы и модули.

Упражнение 26. Калькулятор с префиксной нотацией

В Python, как и в реальной жизни, мы обычно записываем математические выражения с использованием *инфиксной* нотации, то есть `2+3`. Но существует также так называемая префиксная нотация, в которой оператор предшествует аргументам. Префиксная нотация записывается как `+ 2 3`. Существует также *постфиксная* нотация, иногда известная как «обратная польская нотация» (или RPN, прим. пер. от reverse Polish notation), которая до сих пор используется на калькуляторах марки HP. Это будет выглядеть как `2 3 +`. И да, числа должны быть разделены пробелами.

Префиксная и постфиксная нотации полезны тем, что позволяют выполнять сложные операции без скобок. Например, если вы пишете `2 3 4 + *` в RPN, вы говорите системе сначала сло–

жить `3+4`, а затем умножить `2*7`. Именно поэтому на калькуляторах HP есть клавиша Enter, но нет клавиши =, что сильно путает новичков. В языке программирования Lisp префиксная нотация позволяет применять оператор ко многим числам (например, `(+ 1 2 3 4 5)`, а не путаться в большом количестве знаков `+`.

Для этого упражнения я хочу, чтобы вы написали функцию (`calc`), которая принимает один аргумент — строку, содержащую простое математическое выражение в префиксной нотации с оператором и двумя числами. Ваша программа будет анализировать входные данные и выдавать соответствующий вывод. Для наших целей достаточно будет шесть основных арифметических операций в Python: сложение, вычитание, умножение, деление (/), модуль (%) и экспоненция (**). Обычные математические правила Python должны работать, например, при делении всегда получается число с плавающей точкой. Для наших целей мы сделаем так, что аргумент будет содержать только один из шести операторов и два действительных числа.

Но подождите, здесь есть подвох — или подсказка, если хотите: вы должны реализовать каждую из операций как отдельную функцию, и вы не должны использовать оператор `if`, чтобы решить, какая функция должна быть запущена. Еще одна подсказка: посмотрите на модуль `operator`, функции которого реализуют многие операторы Python.

Обсуждение

В этом решении используется техника, известная как таблица диспетчеризации, вместе с модулем Python — `operator`. Это мое любимое решение данной проблемы, однако существуют еще решение — и, скорее всего, не то, о котором вы подумали в первую очередь.

Давайте начнем с самого простого решения и дойдем до того, которое я написал. Нам понадобится функция для каждого из операторов. Но затем нам нужно будет как'то преобразовать строку оператора (например, `+` или `**`) в функцию, которую мы хотим вызвать. Мы могли бы использовать оператор `if`,

но более распространенным способом в Python является использование словарей. В конце концов, нет ничего необычного в том, чтобы ключи содержали в себе строки, а поскольку в значении мы можем хранить что угодно, то это включает и функции.

ПРИМЕЧАНИЕ Многие мои студенты спрашивают меня, как создать оператор `switch-case` в Python. Они удивляются, хотя уже знают ответ, услышав, что в Python нет такого оператора и что вместо него мы используем `if`. Это часть философии Python — иметь один, и только один, способ сделать что-то. Это сокращает выбор у программистов, но делает код более понятным и простым в сопровождении.

Затем мы можем извлечь функцию из словаря и вызвать ее с помощью круглых скобок:

```
def add (a, b):
    return a + b

def sub (a, b):
    return a – b

def mul (a, b):
    return a * b

def div (a, b):
    return a / b

def pow (a, b):
    return a ** b

def mod (a, b):
    return a% b

def calc (to_solve):
    operations = {'+': add,
```

Ключи в операциях над словарями — это операторы строк, которые пользователь может ввести, а значения — это наши функции, связанные с этими строками.

```
                '-': sub,
                '*': mul,
                '/': div,
                '**': pow,
                '%': mod}

    op, first_s, second_s = to_solve.split ()
    first = int (first_s)
    second = int (second_s)

    return operations [op] (first, second)
```

Разбивает ввод пользователя на части.

Использует выбранный пользователем оператор как ключ в операциях, возвращая функцию, которую мы затем вызываем, передавая ей first и second в качестве аргументов.

Превращает каждый пользовательский ввод из строк в целые числа.

Возможно, моя любимая часть кода — это последняя строка. У нас есть словарь, в котором функции являются значениями. Таким образом, мы можем получить нужную нам функцию с помощью `operations [operator]`, где `operator` — это первая часть строки, которую мы разделили с помощью `str.split`. Получив функцию, мы можем вызвать ее с помощью круглых скобок, передав ей два операнда, `first` и `second`.

Но как мы получим `first` и `second`? Из строки ввода пользователя, в которой, как мы предполагаем, содержится три элемента. Мы используем `str.split`, чтобы разделить их на части, и сразу же распаковываем, чтобы присвоить их трем переменным.

Хеджирование ставок с помощью `maxsplit`

Если вам не нравится идея вызывать `str.split`, ожидая, что мы получим три результата, есть простой способ решить эту проблему. При вызове `str.split` передайте значение необязательному параметру `maxsplit`. Этот параметр указывает, сколько разбиений будет фактически вы-

полнено. Другими словами, это индекс последнего элемента в возвращаемом списке. Например, если я напишу

```
>>> s = 'a b c d e'
>>> s.split ()
['a', 'b', 'c', 'd', 'e']
```

Как видите, я получаю (как всегда) список строк. Поскольку я вызвал `str.split` без каких-либо аргументов, Python использовал пробельные символы в качестве разделителей. Но если я передам значение 3 в `maxsplit`, то я получу следующее:

```
>>> s = 'a b c d e'
>>> s.split (maxsplit=3)
['a', 'b', 'c', 'd e']
```

Обратите внимание, что возвращаемый список теперь состоит из четырех элементов. В документации Python говорится, что `maxsplit` указывает `str.split`, сколько должно быть частей. Я предпочитаю думать об этом значении как о наибольшем индексе в возвращаемом списке — поскольку возвращаемый список содержит четыре элемента, последний элемент будет иметь индекс 3. В любом случае `maxsplit` гарантирует, что при использовании распаковки результата мы не столкнемся с ошибкой.

Все это хорошо, но этот код не выглядит соответствующим принципу DRY. Тот факт, что мы должны определять каждую из наших функций, даже если они так похожи друг на друга и являются повторением существующей функциональности, немного расстраивает и не совсем соответствует Python.

К счастью, нам может помочь модуль `operator`, который поставляется вместе с Python. Импортировав `operator`, мы получим именно те функции, которые нам нужны: `add`, `sub`, `mul`,

`truediv/ floordiv`, `mod` и `pow`. Нам больше не нужно определять свои собственные функции, потому что мы можем использовать те, которые предоставляет модуль. Функция `add` в `operators` делает то, что мы обычно ожидаем от оператора +: она смотрит налево, определяет тип первого параметра и использует его, чтобы узнать, что вызывать. `operator.add`, как функция, не нуждается в том, чтобы смотреть налево: она проверяет тип своего первого аргумента и использует его, чтобы определить, какую версию `+` вызывать.

В этом конкретном упражнении мы ограничили вводимые пользователем данные целыми числами, поэтому мы не проводили никакой проверки типов. Но вы можете представить себе версию этого упражнения, в которой мы могли бы работать с различными типами, а не только с целыми числами. В этом случае различные функции из `operator` будут знать, что делать с любыми типами, которые мы им передадим.

Решение

Модуль оператора предоставляет функции, реализующие все встроенные операторы.

```
import operator

def calc (to_solve):
    operations = {'+': operator.add,
          '–': operator.sub,
          '*': operator.mul,
          '/': operator.truediv,
          '**': operator.pow,

    op, first_s, second_s = to_solve.split ()
    first = int (first_s)
    second = int (second_s) return

    operations [op] (first, second)

print (calc ('+ 23'))
```

Да, функции могут быть значениями в словаре!

Вы можете выбрать между truediv, который возвращает число типа *float*, как и оператор /, или *floordiv*, который возвращает целое число, как и оператор //.

Разделяет строку, присваивание через распаковку.

Вызывает функцию, полученную через оператор, передавая *first* и second в качестве аргументов.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor.

Скринкаст решения

Посмотрите короткое видео с объяснением решения.

После выполнения упражнения

Для многих новичков в Python достаточно странно воспринимать функции как данные и хранить их в структурах данных. Но это позволяет использовать методы, которые гораздо сложнее в других языках. Вот три упражнения, которые еще больше расширят эту идею:

1. Расширьте программу, которую вы написали, так, чтобы ввод пользователя мог содержать любое количество чисел (а не только два, как сейчас). Таким образом, программа будет обрабатывать `+ 357` или `/ 10055` и будет применять оператор слева направо, давая ответы `15` и `4` соответственно.
2. Напишите функцию `apply_to_each`, которая принимает два аргумента: функцию, принимающую один аргумент, и итерируемый объект. Верните список, значения которого являются результатом применения функции к каждому элементу итерируемого объекта. (Если это звучит знакомо, возможно, так и есть — это реализация классической функции `map`, все еще доступной в Python. Описание `map` можно найти в главе 7.)
3. Напишите функцию `transform_lines`, которая принимает три аргумента: функцию, принимающую один аргумент, имя входного файла и имя выходного файла. Вызов запустит функцию на каждой строке входного файла, а результаты будут записаны в выходной файл. (Подсказка: предыдущее и это упражнение тесно связаны.)

Упражнение 27. Генератор паролей

Даже сегодня многие люди используют один и тот же пароль на разных компьютерах. Это означает, что если кто–то узнает ваш пароль на системе A, то он сможет войти в системы B, C и D, где вы использовали тот же пароль. По этой причине многие люди (включая меня) используют программное обеспечение, которое создает (и затем запоминает) длинные пароли, сгенерированные рандомно. Если вы пользуетесь такой программой, то даже если система A будет взломана, ваши входы в системы B, C и D будут в безопасности.

В этом упражнении мы создадим функцию генерации паролей. Фактически мы создадим своеобразную фабрику для функций генерации паролей. Например, мне может понадобиться сгенерировать большое количество паролей, все из которых используют одно и то же множество символов. (Вы знаете, как это бывает. Некоторые приложения требуют сочетания заглавных и строчных букв, цифр и символов; другие требуют использования только букв; третьи разрешают использовать и буквы, и цифры.) Таким образом, вы будете вызывать функцию `create_password_generator` со строкой. Эта строка вернет функцию, которая сама принимает целочисленный аргумент. Вызов этой функции вернет пароль указанной длины, используя строку, из которой он был создан, например:

```
alpha_password = create_password_generator ('abcdef')
symbol_password = create_password_generator ('!@#$%')

print (alpha_password (5)) # efeaa
print (alpha_password (10)) # cacdacbada

print (symbol_password (5)) #%#@%@
print (symbol_password (10)) # @!%%$%$%%#
```

Для реализации этой функции полезно знать о модуле `random`, а точнее, о функции `random.choice` из этого модуля. Эта функция возвращает один (случайно выбранный) элемент из последовательности.

Смысл этого упражнения в том, чтобы понять, как работать с внутренними функциями: определять их, возвращать и использовать для создания множества подобных функций.

Обсуждение

Вот пример того, как можно использовать внутреннюю функцию, иногда называемую замыканием. Идея заключается в том, что мы вызываем функцию (`create_password_generator`), которая, в свою очередь, возвращает функцию (`create_password`). Возвращаемая внутренняя функция знает, что мы сделали при первоначальном вызове, но также обладает некоторыми собственными функциональными возможностями. В результате ее нужно определить как внутреннюю функцию, чтобы она могла получить доступ к переменным из первоначального (внешнего) вызова.

Определение внутренней функции происходит не тогда, когда Python впервые выполняет программу, а при выполнении внешней функции (`create_password_generator`). Действительно, мы создаем новую внутреннюю функцию один раз для каждого вызова `create_password_generator`.

Затем эта новая внутренняя функция возвращается вызывающей стороне. С точки зрения Python, здесь нет ничего особенного — мы можем вернуть любой объект Python из функции: список, словарь или даже функцию. Однако особенным здесь является то, что возвращаемая функция ссылается на переменную во внешней функции, где она была изначально определена.

В конце концов, мы хотим получить функцию, в которую можно передать целое число и из которой можно получить случайно сгенерированный пароль. Но пароль должен содержать определенные символы, а разные программы имеют разные ограничения на то, какие символы можно использовать для этих паролей. Так, нам может понадобиться пять буквенно-цифровых символов, или 10 цифр, или 15 символов, которые являются либо буквенно-цифровыми, либо пунктуационными.

Таким образом, мы определяем нашу внешнюю функцию так, чтобы она принимала единственный аргумент — строку, содер-

жащую символы, из которых мы хотим создать новый пароль. Результатом вызова этой функции, как было указано, является функция — динамически определяемая `create_password`. Эта внутренняя функция имеет доступ к исходной переменной `characters` во внешней функции благодаря правилу LEGB в Python для поиска переменных. (См. сноску «Область видимости переменных в Python».) Когда внутри `create_password` мы ищем переменную `characters`, которая будет находиться в области видимости внешней функции.

Если мы дважды вызовем `create_password_generator`, как показано на изображении из Python Tutor (рис. 6.3), каждый вызов будет возвращать отдельную версию `create_password` с отдельным значением символов.

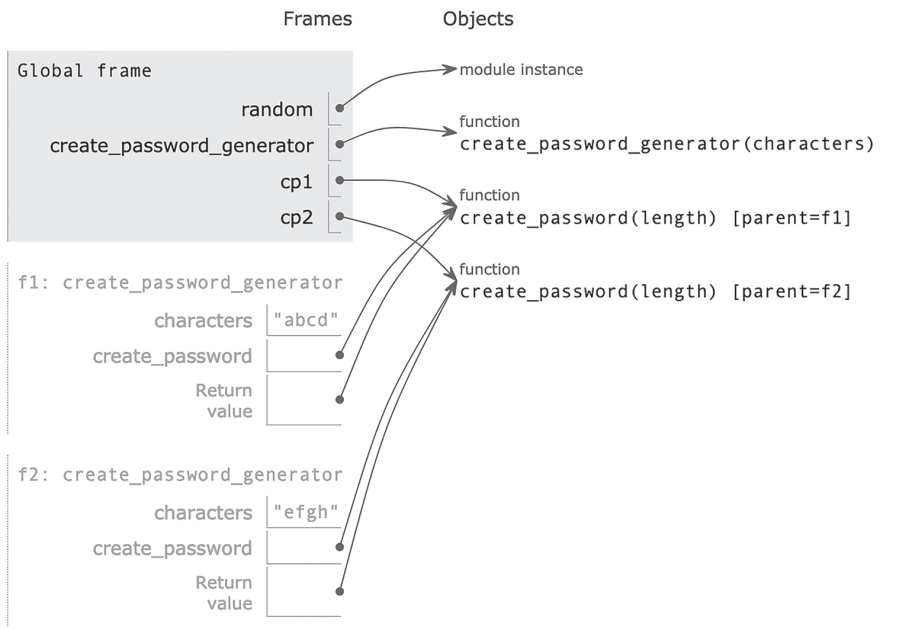


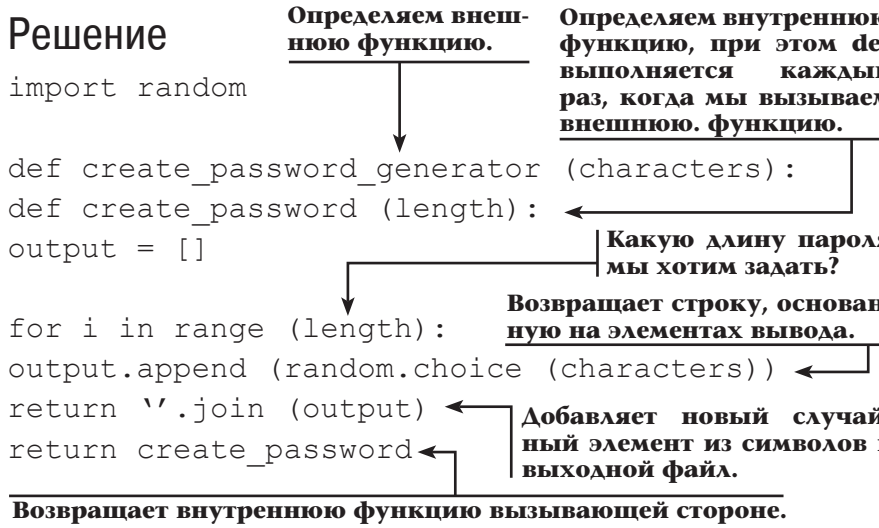
Рисунок 6.3. Визуализация (из Python Tutor) двух функций, генерирующих пароль.

Каждый вызов внешней функции возвращает новую функцию со своими локальными переменными. В то же время каждая из возвращаемых внутренних функций имеет доступ к локальным переменным своей внешней функции. Когда мы вызываем

одну из внутренних функций, мы получаем новый пароль, основанный на комбинации локальных переменных внутренней функции и локальных переменных внешней функции.

ПРИМЕЧАНИЕ Работа с внутренними функциями и замыканиями может поначалу удивить и сбить с толку. Это происходит в основном потому, что мы инстинктивно полагаем, что, когда функция возвращается, ее локальные переменные и состояние исчезают. Действительно, обычно это так, но помните, что в Python объект не освобождается и не удаляется сборщиком мусора, если на него имеется хотя бы одна ссылка. И если внутренняя функция все еще ссылается на фрейм стека, в котором она была определена, то внешняя функция будет существовать до тех пор, пока существует внутренняя функция.

Решение




```
alpha_password = create_password_generator ('abcdef')
symbol_password = create_password_generator ('!@#$%')

print (alpha_password (5))
print (alpha_password (10))

print (symbol_password (5))
print (symbol_password (10))
```

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr136].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr137].

После выполнения упражнения

Представление о функциях как о данных позволяет работать на еще более высоком уровне абстракции, чем обычные функции, и таким образом решать проблемы еще более высокого уровня, не заботясь о деталях низкого уровня. Тем не менее может потребоваться некоторое время, чтобы усвоить и понять, как передавать функции в качестве аргументов другим функциям или возвращать функции из других функций. Вот несколько дополнительных упражнений, которые вы можете попробовать для лучшего понимания и работы с ними:

1. Теперь, когда вы написали функцию для создания паролей, напишите функцию `create_pass-word_checker`, которая проверяет, соответствует ли данный пароль критериям приемлемости, установленным ИТ-сотрудниками. Другими словами, создайте функцию с четырьмя параметрами: `min_uppercase`, `min_lowercase`, `min_punctuation` и `min_digits`. Это минимальное количество заглавных букв, строчных букв, знаков препинания и цифр для приемлемого пароля. Выходным значением функции `create_password_ checker` является функция, которая принимает на вход потенциальный пароль (строку) и возвращает булево значение, указывающее, является ли эта строка приемлемым паролем.
2. Напишите функцию `getitem`, которая принимает один аргумент и возвращает функцию `f`. Возвращенная `f` может быть вызвана для любой структуры данных, элементы которой могут быть выбраны с помощью квадратных ско-

бок, и затем вернет этот элемент. Так, если я вызову `f = getitem ('a')` и если у меня есть словарь `d = {'a':1, 'b':2}`, то `f (d)` вернет `1`. (Это очень похоже на operator.itemgetter, очень полезную функцию для многих случаев.)

3. Напишите функцию `doboth`, которая принимает в качестве аргументов две функции (`f1` и `f2`) и возвращает одну функцию `g`. Вызов `g (x)` должен вернуть тот же результат, что и вызов `f2 (f1 (x))`.

Подводя итоги

Написать простые функции Python несложно. Но в чем функции Python действительно хороши — так это в гибкости, особенно когда речь идет об интерпретации параметров, и в том, что функции тоже являются данными. В этой главе мы рассмотрели все эти идеи, что должно натолкнуть вас на некоторые мысли о том, как использовать преимущества функций в ваших собственных программах.

Если вы когда-нибудь обнаружите, что пишете похожий код несколько раз, вам следует серьезно подумать о том, чтобы обобщить его в функцию, которую вы сможете вызывать из этих мест. Более того, если вы решили реализовать что-то, что вы, возможно, захотите использовать в будущем, реализуйте это как функцию. Кроме того, код, разбитый на функции, часто легче понять, поддерживать и тестировать, поэтому даже если вы не беспокоитесь о повторном использовании или более высоких уровнях абстракции, все равно может быть полезно писать код в виде функций.

7. Функциональное программирование с генераторами

Программисты всегда пытаются сделать больше, но с меньшим объемом кода, одновременно делая этот код более надежным и легким для отладки. И действительно, специалисты в области компьютерных наук разработали ряд методик, каждая из которых призвана приблизить нас к этой цели — короткому, надежному, удобному в обслуживании, мощному коду.

Нам будет интересен следующий набор методик — функциональное программирование. Его цель — сделать программы более надежными за счет коротких функций и неизменяемости данных. Я думаю, большинство разработчиков согласятся, что короткие функции — это хорошая идея, в немалой степени потому, что их легче понимать, тестировать и поддерживать.

Но каким образом можно реализовать короткие функции? Посредством неизменяемых данных. Если вы не можете изменять данные внутри функции, то функция (по моему опыту) в итоге будет короче, с меньшим количеством потенциальных способов для проверки. Таким образом, функциональные программы в конечном итоге содержат много коротких функций — в отличие от нефункциональных программ, которые часто имеют меньшее количество очень длинных функций. Функциональное программирование также предполагает, что функции можно передавать

в качестве аргументов другим функциям, что, как мы уже убедились, имеет место в Python.

Хорошая новость заключается в том, что функциональные методы способны сделать код коротким и элегантным. Плохая новость заключается в том, что для многих разработчиков функциональные техники непонятны. Отказ от модификации значений и отслеживание состояния могут быть отличными способами сделать ваше программное обеспечение более надежным, но они почти гарантированно запутают и разочаруют многих разработчиков.

Рассмотрим, например, что у вас есть объект `Person` в чисто функциональном языке. Если человек захочет изменить свое имя, вам не повезет, поскольку все данные неизменяемы. Вместо этого вам придется создать новый объект `Person` на основе старого, но с измененным именем. Само по себе это не страшно, но, учитывая, что реальный мир меняется и мы хотим, чтобы наши программы моделировали реальный мир, сохранение неизменности абсолютно всего может вызывать разочарование.

Но опять же, поскольку функциональные языки не могут изменять данные, они обычно предоставляют механизмы для получения последовательности входных данных, преобразования их каким-либо образом и получения последовательности выходных данных. Мы можем не иметь возможности изменить один объект `Person`, но мы можем написать функцию, которая принимает список объектов `Person`, применяет к каждому из них выражение Python, а затем получает обратно новый список объектов `Person`. В таком сценарии мы, возможно, не изменили исходные данные, но задачу выполнили. И код, необходимый для этого, обычно довольно короткий.

Итак, Python не является функциональным языком — у нас есть изменяемые типы данных и присваивание. Но некоторые функциональные техники пробрались в язык и считаются стандартными питоновскими способами решения некоторых проблем.

В частности, в Python предлагаются *генераторы* — современный подход к классическим функциям, зародившийся в Lisp, одном из первых изобретенных языков высокого уровня. Генераторы позволяют относительно легко создавать списки, множества

и словари на основе других структур данных. Тот факт, что функции Python являются объектами и могут передаваться в качестве аргументов или храниться в структурах данных, также пришел из функционального мира.

В некоторых решениях упражнений уже использовались генераторы, кое-какие упражнения были подготовкой для лучшего понимания. В этой главе мы сосредоточимся на том, как и когда использовать эти техники, и расширим способы их применения.

По моему опыту, обычно при первом знакомстве с функциональными техниками, и особенно с генераторами, к ним относятся равнодушно. Но со временем — да, на это могут уйти годы — разработчики все лучше начинают понимать, как, когда и зачем их применять. Так что даже если вы сможете решить проблемы в этой главе без использования функциональных методов, смысл здесь в том, чтобы потренироваться, попробовать решить задачу с их помощью и увидеть логику и элегантность, стоящую за этим способом решения задач. Преимущества могут быть не сразу очевидны, но со временем они окупятся.

Если все это звучит очень теоретически, и вы хотите увидеть несколько конкретных примеров генераторов в сравнении с традиционным процедурным программированием, то обратите внимание на сноску «Создание генераторов» в этой главе, где я более подробно рассматриваю различия.

Таблица 7. Что вам нужно знать

Понятие	Что это?	Пример	Чтобы узнать подробнее
Генератор списка	Создает список на основе элементов итерируемого объекта.	`[x*x for x in range (5)]`	
Генератор словаря	Создает словарь на основе элементов итерируемого объекта.	`{x: 2*x for x in range (5)}`	

Окончание таблицы

Понятие	Что это?	Пример	Чтобы узнать подробнее
Генератор множества	Создает множество на основе элементов итерируемого объекта.	`{x*x for x in range (5)}`	

Таблица 7.1. Чтобы узнать подробнее

Понятие	Что это?	Пример	Чтобы узнать подробнее
`input`	Предлагает пользователю ввести строку, затем возвращает строку.	`input ('Name: ')`	
`str.isdigit`	Возвращает `True` или `False`, если строка непустая и содержит только 0–9.	`# Возвращает True '5'. isdigit ()`	
`str.split`	Разбивает строки на части, возвращая список.	`# Возвращает ['ab', 'cd', 'ef'] 'ab cd ef'.split ()`	
`str.join`	Объединяет строки для создания новой строки.	`# Возвращает 'ab*cd*ef' '*'. join (['ab', 'cd', 'ef']) string.ascii`	
`_lowercase`	Все английские строчные буквы.	`string.ascii_ lowercase`	
`enumerate`	Возвращает итератор двухэлементных кортежей с индексом.	`enumerate ('abcd')`	

Упражнение 28. Объединение чисел

Люди часто спрашивают меня: «Когда я должен использовать генератор, а не классический цикл `for`?»

Мой ответ в основном таков: когда вы хотите преобразовать итерируемый объект в список, вам следует использовать генератор. Но если вы просто хотите выполнить что–то для каждого элемента итерируемого объекта, то лучше использовать цикл `for`.

Другими словами, является ли целью вашего цикла `for` создание нового списка? Если да, то используйте генератор. Но если ваша цель — выполнить что–то один раз для каждого элемента итерируемого объекта, отбрасывая или игнорируя любое возвращаемое значение, то предпочтительнее использовать цикл `for`. Например, я хочу получить длину слов в строке `s`. Я могу написать:

```
[len (one_word)
for one_word in s.split ()]
```

В этом примере мне важен список, который мы создаем, поэтому я использую генератор.

Но если моя строка `s` содержит список имен файлов, и я хочу создать новый файл для каждого из этих имен, то меня не интересует возвращаемое значение. Скорее, я хочу выполнить итерацию по именам файлов и создать файл, как показано ниже:

```
for one_filename in s.split ():
   with open (one_filename, 'w') as f:
        f.write (f'{one_filename} \n')
```

В этом примере я открываю (и тем самым создаю) каждый файл и записываю в него имя файла. Использование генератора в данном случае было бы неуместным, поскольку меня не интересует возвращаемое значение.

Преобразования — получение значений в списке, строке, словаре или другом итерируемом объекте и создание на их основе нового списка — часто встречаются в программировании. Вам

может понадобиться преобразовать имена файлов в объекты файлов, или слова в их длину, или имена пользователей в ID пользователей. Во всех этих случаях генератор является наиболее питоновским решением.

Это упражнение предназначено для того, чтобы вы начали понимать эту идею и применять ее. Оно может показаться простым, но в основе лежит глубокая и мощная идея, которая поможет вам увидеть дополнительные возможности для использования генераторов.

Для этого упражнения напишите функцию (`join_numbers`), которая принимает `диапазон (range)` целых чисел. Функция должна возвращать эти числа в виде строки с запятыми между числами. То есть, получив на вход `range (15)`, функция должна вернуть следующую строку:

```
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14
```

Подсказка: если вы думаете, что `str.join` — это хорошая идея, то вы, в общем, правы — но помните, что `str.join` не будет работать со списком целых чисел.

Обсуждение

В этом упражнении мы хотим использовать `str.join` для диапазона, который похож на список целых чисел. Если мы попытаемся сразу вызвать `str.join`, то получим ошибку:

```
>>> numbers = range (15)
>>> ‘,’.join (numbers)
Traceback (последний вызов):
     Файл “<stdin>”, линия 1, в <module>
TypeError: элемент последовательности 0: ожидался
экземпляр str, найден int
```

Это потому, что `str.join` работает только с последовательностью строк. Поэтому нам нужно преобразовать каждое из це-

лых чисел в нашем диапазоне (`numbers`) в строку. Затем, когда у нас будет список строк, основанный на нашем диапазоне целых чисел, мы можем вызвать `str.join`.

Решение заключается в использовании генератора списка для вызова функции str для каждого из чисел в диапазоне. В результате получится список строк, чего и ожидает `str.join`. Как?

Рассмотрим следующее: генератор списка говорит, что мы собираемся создать новый список. Все элементы нового списка основаны на элементах в исходном итераторе после применения выражения. Мы описываем новый список в терминах старого.

Вот несколько примеров, которые помогут вам понять, где и как использовать генераторы списков:

1. Я хочу узнать возраст каждого ученика в классе. Итак, мы начинаем со списка учеников и заканчиваем списком целых чисел. Вы можете написать функцию `student_age`, которая может быть применена для каждого студента для получения его возраста:

```
[student_age (one_student)
for one_student in all_students]
```

2. Я хочу узнать, сколько мм осадков выпало в каждый день предыдущего месяца. Итак, мы начинаем со списка дней и заканчиваем списком плавающих значений. Вы можете представить себе функцию `daily_rain`, применяемую для каждого дня:

```
[daily_rain (one_day)
for one_day in most_recent_month]
```

3. Я хочу узнать, сколько гласных было использовано в книге. Поэтому мы применим функцию `number_of_vowels` к каждому слову в книге, а затем выполним функцию `sum` для полученного списка:

```
[number_of_vowels (one_word)
for one_word in open (filename).read ().split ()]
```

Если эти три примера выглядят очень похожими, то это потому, что так оно и есть: часть силы генераторов списков заключается в простой формуле, которую мы повторяем. Каждый генератор списка содержит две части:

1. Исходный итерируемый объект.
2. Выражение, которое мы будем вызывать один раз для каждого элемента.

В случае нашего упражнения у нас был список целых чисел. Применив функцию `str` к каждому int в списке, мы получили список строк. `str.join` отлично работает со списками строк.

ПРИМЕЧАНИЕ Мы рассмотрим особенности протоколов итераторов в главе 10, которая посвящена этому вопросу. Вам не нужно разбираться в этих деталях, чтобы использовать генераторы. Однако если вам особенно интересно, что считается «итерируемым объектом», то прочтите первую часть этой главы, прежде чем продолжить.

Создаем генераторы

Генераторы обычно пишутся в одну строку:

```
[x*x for x in range (5)]
```

Я заметил, что для новичков в Python, как и иногда для опытных разработчиков, трудно понять, что происходит. Все становится сложнее, когда мы добавляем условие

```
[x*x for x in range (5) if x%2]
```

По этой причине я настоятельно рекомендую разработчикам Python разбивать генераторы списков на части. Python снисходителен к пробелам, если мы находимся внутри круглых скобок, что всегда (по определению) имеет место, когда мы находимся в генераторе. Мы можем разбить генератор следующим образом:

```
[x*x                    ←——— Выражение
for x in range (5)      ←——— Итерация
if x%2]                 ←——— Условие
```

Разделив выражение, итерацию и условие на разные строки, генератор становится более понятным. Кроме того, так легче экспериментировать с генераторами. Я буду писать большинство своих вычислений в этой книге, используя двухстрочный или трехстрочный формат, и я призываю вас делать то же самое.

Обратите внимание, что при использовании этой техники вложенные генераторы списков также становятся более понятными:

```
[(x, y)                 ←——— Выражение
for x in range (5)      ←——— Итерация #1, от 0 до 4.
if x%2                  ←——— Условие #1, игнорирование четных чисел.
for y in range (5)      ←——— Итерация #2, от 0 до 4.
if y%3]                 ←——— Условие #2, игнорирование чисел, кратных 3.
```

Другими словами, при таком генераторе списка получаются пары целых чисел, в которых первое число должно быть нечетным, а второе не может быть кратным 3. Вложенные генераторы могут быть трудны для понимания, но, когда каждая часть появляется в отдельной строке, понять происходящее становится проще.

Вложенные генераторы списков отлично подходят для работы со сложными структурами данных, такими как списки списков или списки кортежей. Например, предположим, что у меня есть словарь, описывающий страны и города, которые я посетил за последний год:

```
all_places = {'USA': ['Philadelphia', 'New York',
'Cleveland', 'San Jose', 'San Francisco'],
```

```
    'China':['Beijing', 'Shanghai', 'Guangzhou'],
    'UK': ['London'],
    'India': ['Hyderabad']}
```

Если мне нужен список городов, в которых я побывал, не обращая внимания на страны, я могу использовать вложенный генератор списка:

```
[one_city
for one_country, all_cities in all_places.items ()
for one_city in all_cities]
Я также могу создать список кортежей (city,
country):
[(one_city, one_country)
for one_country, all_cities in all_places.items ()
for one_city in all_cities]
```

И, конечно, я всегда могу отсортировать их с помощью sorted:

```
[(one_city, one_country)
for one_country, all_cities sorted (all_places.
items ())
for one_city in sorted (all_cities)]
```

Теперь, если вы имеете дело с большими объемами данных, то в результате работы генератора списка сразу же создается список, что может привести к использованию большого количества памяти. По этой причине многие разработчики Python утверждают, что лучше использовать генератор выражений.

Выражения-генераторы выглядят точно так же, как и генераторы списков, за исключением того, что вместо квадратных скобок используются обычные круглые скобки. Однако, как оказалось, это имеет большое значение: генератор списка должен создавать и возвращать свой выходной список одним махом, что

потенциально может занимать много памяти. Генератор выражений, напротив, возвращает вывод по одному фрагменту за раз. Например, рассмотрим

```
sum ([x*x for x in range (100000)])
```

В этом коде у `sum` один входной параметр — список целых чисел. Она выполняет итерации над списком целых чисел и суммирует их. Но учтите, что перед тем, как `sum` сможет запуститься, генератор должен завершить создание всего списка целых чисел. Потенциально этот список может быть очень большим и занимать много памяти.

Для сравнения рассмотрим следующий код:

```
sum ((x*x for x in range (100000)))
```

Здесь входные данные для `sum` — это не список, а генератор, который мы создали с помощью нашего выражения генератора. `sum` вернет точно такой же результат, как и раньше. Однако если в нашем первом примере был создан список, содержащий 100 000 элементов, то в данном случае используется гораздо меньше памяти. Генератор возвращает по одному элементу за раз, ожидая, пока `sum` запросит следующий по очереди элемент. Таким образом, за один раз мы потребляем память только на одно целое число, а не на огромный список целых чисел. В итоге можно сказать, что вы можете использовать выражения–генераторы практически везде, где вы можете использовать вычисления, но при этом вы будете использовать гораздо меньше памяти.

Оказывается, когда мы помещаем выражение–генератор в вызов функции, мы можем удалить внутренние круглые скобки: `sum (x*x for x in range (100000))`

Итак, вот синтаксис, который вы видели в решении этого упражнения, но с использованием выражения–генератора:

```
numbers = range (15)
print (',',.join (str (number)
for number in numbers))
```

Решение

```
def join_numbers (numbers):
    return ‘,’.join (str (number)
          for number in numbers)
print (join_numbers (range (15)))
```

Применяет str к каждому числу и помещает новую строку в список вывода.

Итерации над элементами чисел.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr147].

147

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr148].

148

После выполнения упражнения

Вот несколько заданий, с помощью которых вы можете выйти за рамки этого упражнения и познакомиться с новыми способами использования генераторов списков:

1. Как и в упражнении, возьмите список целых чисел и превратите их в строки. Однако вы хотите получить строки только для целых чисел от 0 до 10. Для этого вам потребуется понимание принципа работы оператора `if` в генераторах списков.
2. Учитывая список строк, содержащих шестнадцатеричные числа, просуммируйте эти числа.
3. Используйте генератор списка, чтобы изменить порядок слов в строках текстового файла. То есть, если первая строка — `abc def`, а вторая — `ghi jkl`, то вы должны вернуть список `[‘def abc’, ‘jkl ghi’]`.

`map`, `filter` и генераторы

Генераторы, по своей сути, делают две разные вещи. Во-первых, они преобразуют одну последовательность в другую, применяя выражение к каждому элементу входной

последовательности. Во–вторых, они отфильтровывают элементы из вывода.Приведем пример:

```
[x*x          ← x в квадрате.
for x in range (10) ← Для каждого числа от 0 до 9.
if x%2 == 0] ← Только если x четное.
```

В первой строке происходит преобразование, а в третьей — фильтрация. До генераторов Python эти функции традиционно реализовывались при помощи двух функций: `map` и `filter`. Действительно, эти функции продолжают существовать в Python, даже если они используются не так часто. `map` принимает два аргумента: функцию и итерируемый объект. Он применяет функцию к каждому элементу итерируемого объекта, возвращая новый итерируемый объект, например:

```
words = 'this is a bunch of words'.split ()   ← Создает список строк и присваивает его words.
x = map (len, words)   ← Применяет функцию len к каждому слову, в результате чего получается итерируемые объекты (целые числа).
print (sum (x))   ← Использует функцию суммы по x.
```

Обратите внимание, что `map` всегда возвращает итерируемый объект той же длины, что и входное значение. Это потому, что у него нет способа удалить элементы. Он применяет свою функцию ввода один раз для каждого элемента ввода. Таким образом, мы можем сказать, что `map` преобразует, но не фильтрует.

Функция, передаваемая в `map`, может быть любой функцией или методом, принимающим один аргумент. Вы можете использовать встроенные функции или написать свои собственные. Главное, что нужно помнить, это то, что вывод функции помещается в итерируемое выходное значение.

`filter` также принимает два аргумента — функцию и итерируемый объект — и использует функцию к каждому элементу. Но здесь выходное значение функции определяет, появится ли элемент в выводе — он вообще не преобразует элемент, например:

```
words = 'this is a bunch of words'.split ()

def is_a_long_word (one_word):
return len (one_word) > 4

x = filter (is_a_long_word, words)
print (' '.join (x))
```

Создает список строк и присваивает его words.

Определяет функцию, которая возвращает значение True или False в зависимости от переданного ей слова.

Применяет нашу функцию для каждого слова в words.

Показывает отфильтрованные слова.

Хотя функция, переданная в `filter`, не обязательно должна возвращать значение `True` или `False`, ее результат будет интерпретироваться как булево значение и использоваться для определения того, будет ли элемент помещен в выходную последовательность. Поэтому обычно целесообразно передавать функцию, возвращающую значение `True` или `False`.

Комбинация `map` и `filter` означает, что вы можете взять итерируемый объект, отфильтровать его элементы, а затем применить функцию к каждому из его элементов. Это оказывается чрезвычайно полезным и объясняет, почему `map` и `filter` существуют так долго — около 50-ти лет.

Тот факт, что функции можно передавать в качестве аргументов, является ключевым для выполнения `map` и `filter`. Это одна из причин, почему эти методы являются основной частью функционального программирования, поскольку они требуют, чтобы к функциям можно было относиться как к данным.

При этом современным способом выполнения подобных действий в Python считаются генераторы. В то время как мы передаем функции в `map` и `filter`, мы передаем выражения в генераторы.

Почему же тогда `map` и `filter` продолжают существовать в языке, если считается, что использование генераторов лучше? Отчасти по ностальгическим и историческим причинам, но также и потому, что иногда они могут делать то, что не так просто сделать с помощью генераторов. Например, map может принимать на вход несколько итерируемых объектов, а затем применять функции, которые будут работать с каждым из них:

```
import operator
letters = 'abcd'
numbers = range (1,5)

x = map (operator.mul, letters, numbers)
print (' '.join (x))
```

Мы будем использовать operator.mul в качестве функции map.

Устанавливает четырехэлементную строку.

Устанавливает диапазон целых чисел из четырех элементов.

Применяет operator.mul (умножение) к соответствующим элементам букв и цифр.

Соединяет строки вместе с пробелами и печатает результат.

В итоге вывод будет таким:

```
a bb ccc dddd
```

Используя генератор, мы могли бы переписать код как

```
import operator
letters = 'abcd'
numbers = range (1,5)

      print (' '.join (operator.mul (one_
      letter, one_number)
```

```
                for one_letter, one_number in zip
                (letters, numbers)))
```

Обратите внимание, что для одновременного перебора `букв` и `цифр` мне пришлось использовать здесь `zip`. Напротив, `map` может просто принимать дополнительные итерируемые аргументы.

Что такое выражение?

Выражение — это все в Python, что возвращает значение. Если это кажется вам немного абстрактным, то вы можете просто думать о выражении как о чем–либо, что вы можете присвоить переменной или вернуть из функции. Итак, `5` — это выражение, как и `5+3`, как и `len ('abcd')`.

Когда я говорю, что генераторы используют выражения, а не функции, я имею в виду, что мы не передаем функцию. Скорее, мы просто передаем то, что хотим, чтобы Python выполнил, подобно передаче тела функции без передачи формального определения функции.

Упражнение 29. Сложение чисел

В предыдущем упражнении мы взяли последовательность чисел и превратили ее в последовательность строк. На этот раз мы поступим наоборот — возьмем последовательность строк, превратим их в числа, а затем просуммируем. Но мы собираемся сделать это немного сложнее, потому что мы собираемся отфильтровать те строки, которые нельзя преобразовать в целые числа.

Наша функция (`sum_numbers`) будет принимать в качестве аргумента строку, например:

```
10 abc 20 de443055fg 40
```

Учитывая ввод, функция должна вернуть `100`. Это потому, что функция будет игнорировать любое слово, содержащее не цифры.

Попросите пользователя ввести целые числа, все сразу, используя `input`.

Обсуждение

В этом упражнении нам дана строка, которая, как мы предполагаем, содержит целые числа, разделенные пробелами. Мы хотим получить отдельные целые числа из строки и затем сложить их вместе. Самый простой способ сделать это — вызвать команду `str.split` для строки, которая возвращает список строк. Вызывая `str.split` без параметров, мы говорим Python, что в качестве разделителя следует использовать любую комбинацию пробелов.

Теперь у нас есть список строк, а не список целых чисел. Нам нужно перебрать все строки, превращая каждую из них в целое число путем вызова `int`. Самый простой способ превратить список строк в список целых чисел — это использовать генератор списка, как в коде решения. Теоретически мы можем вызвать встроенную функцию `sum` для списка целых чисел, и все будет готово.

Но есть одна загвоздка. Возможно, что пользователь вводит элементы, которые нельзя превратить в целые числа. Нам нужно избавиться от них: если мы попытаемся выполнить `int` на строке `abcd`, программа завершится с ошибкой.

К счастью, генератор списков может помочь нам и здесь. Мы можем использовать третью (фильтрующую) строку генератора, чтобы указать, что только те строки, которые могут быть преобразованы в числа, пройдут в первую строку. Для этого мы используем оператор `if`, применяя метод `str.isdigit`, чтобы выяснить, можем ли мы успешно превратить слово в целое число.

Затем мы вызываем `sum` для выражения-генератора, возвращая целое число. Наконец, мы выводим сумму с помощью f-строки.

Решение

```
def sum_numbers (numbers):
    return sum (int (number)
          for number in numbers.split ()
          if number.isdigit ())
print (sum_numbers ('1 2 3 a b c 4'))
```

Создает целое число на основе number.

Итерации по каждому из слов в numbers.

Игнорирует слова, которые не могут быть преобразованы в целые числа.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr149].

149

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr150].

150

После выполнения упражнения

Исходя из своего опыта, могу сказать, что одним из наиболее распространенных применений генераторов списков является комбинация преобразования и фильтрации. Вот несколько дополнительных упражнений, которые помогут вам убедиться в том, что вы не просто знаете синтаксис, но и умеете их применять:

1. Выведите строки текстового файла длиною более 20 символов, которые содержат хотя бы одну гласную.
2. В США телефонные номера состоят из 10 цифр — трехзначного кода региона, за которым следует семизначный номер. Несколько раз в моем детстве в районных кодах заканчивались телефонные номера, что вынуждало половину населения получать новый районный код. После такого разделения XXX–YYY–ZZZZ мог остаться XXX–YYY–ZZZZ, а мог стать NNN–YYY–ZZZZ, причем NNN был новым кодом города. Решение о том, какие номера должны остаться, а какие измениться, часто принималось на основе

последних семи цифр телефонных номеров. Используйте генератор списка для возврата нового списка строк, в котором код города любого телефонный номера, чей YYY начинается с цифр 0–5, будет изменен на XXX+1. Например, у нас есть список строк: `['123-456-7890', '123-333-4444', '123-777-8888']`, который мы хотим преобразовать в `['124-456-7890', '124-333-4444', '124-7778888']`.

3. Задайте список из пяти словарей. Каждый словарь будет содержать две пары ключ–значение, `name` и `age`, содержащие имя человека и его возраст (в годах). Используйте генератор списка для создания списка словарей, в котором каждый словарь содержит три пары ключ–значение: `name`, исходный `age` и третий ключ `age_in_months`, содержащий возраст человека в месяцах. Однако в итоговый результат не должны включаться любые исходные словари, представляющие лиц старше 20 лет.

Упражнение 30. Сглаживание списка

В Python довольно часто используются сложные структуры данных для хранения информации. Конечно, можно создать новый класс, но зачем это делать, если можно просто использовать комбинации списков, кортежей и словарей? Правда, это означает, что иногда вам придется превращать сложные структуры данных в более простые.

В этом упражнении мы потренируемся в этом. Напишите функцию, которая принимает список списков (глубиной всего в один элемент) и возвращает простую, одномерную версию списка. Таким образом, вызывая функцию

```
flatten ([[1,2], [3,4]])
```

мы получим

```
[1,2,3,4]
```

Обратите внимание, что существует несколько возможных решений этой задачи: я прошу вас решить ее с помощью генератора списков. Также не забудьте, что нам нужно сосредоточиться на сглаживании двухуровневого списка.

Обсуждение

Как мы уже заметили, генераторы списков позволяют нам оценивать выражение на каждом элементе итерируемого объекта. Но для обычного генератора списка вы не можете вернуть больше элементов, чем было во входном итерируемом объекте. Например, если входной итерируемый объект содержит 10 элементов, вы можете вернуть только 10, или меньше 10, если вы используете условие `if`.

Вложенные генераторы списков немного меняют эту ситуацию, поскольку результат может содержать столько элементов, сколько вложенных элементов у входного итерируемого объекта. Если задан список списков, первый `for` цикл будет повторяться для каждого элемента из `mylist`. Но второй цикл `for` будет перебирать элементы внутреннего списка. Мы можем получить один выходной элемент для каждого внутреннего входного элемента, что мы и делаем:

```
def flatten (mylist):
    return [one_element
            for one_sublist in mylist
            for one_element in one_sublist]
```

Решение

```
def flatten (mylist):
    return [one_element
            for one_sublist in mylist
            for one_element in one_sublist]

print (flatten ([[1,2], [3,4]]))
```

Итерация по каждому элементу списка mylist.

Итерация по каждому элементу списка one_sublist.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr151].

151

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr152].

152

После выполнения упражнения

Вложенные генераторы списков поначалу могут немного пугать, но они также весьма полезны во многих случаях. Вот несколько упражнений, которые вы можете попробовать решить, чтобы лучше понять, как их использовать:

1. Напишите версию упомянутой ранее функции `flatten` под названием `flatten_odd _ints`. Она будет делать то же самое, что и `flatten`, но на выходе будут только нечетные целые числа. Входные данные, которые не являются ни нечетными, ни целыми числами, должны быть исключены. Входные данные, содержащие строки, которые могут быть преобразованы в целые числа, должны быть преобразованы, а другие строки должны быть исключены.

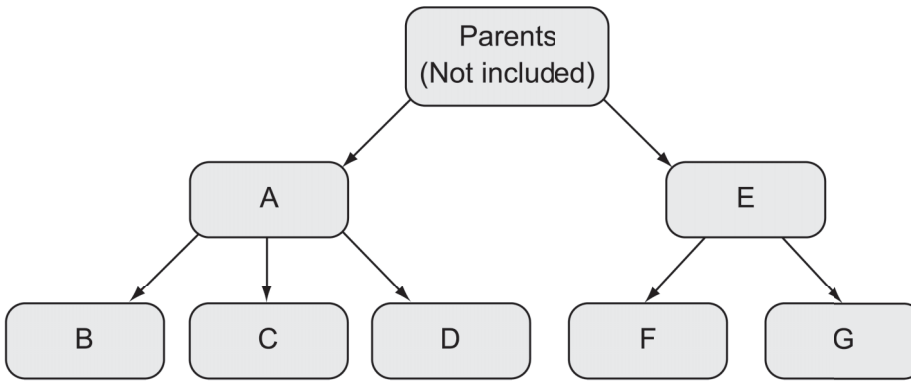


Рисунок 7.1. Схема семьи для вложенных генераторов списков.

2. Определите словарь, содержащий детей и внуков в семье. (Графическое представление см. на рисунке 7.1.) Каждый

ключ будет именем ребенка, а каждое значение — списком строк, представляющих его детей (т.е. внуков семьи). Таким образом, словарь `{'A': ['B', 'C', 'D'], 'E': ['F', 'G']} означает что A и E` — родные братья и сестры; B, C и D — дети A; F и G — дети E. Используйте генератор списка для создания списка имен внуков.

3. Повторите это упражнение, но замените имя каждого внука (сейчас это строка) словарем. Каждый словарь будет содержать две пары имя–значение, `name` и `age`. Создайте список имен внуков, отсортированный по возрасту, от старшего к младшему.

Упражнение 31. Перевод содержимого файла на поросячью латынь

Генераторы списков отлично подходят для преобразования списка. Но на самом деле они могут работать с любым итерируемым объектом, то есть с любым объектом Python, на котором можно запустить цикл `for`. Это означает, что исходными данными для генератора списка могут быть строка, список, кортеж, словарь, множество или даже файл.

В этом упражнении я хочу, чтобы вы написали функцию, которая принимает в качестве аргумента имя файла. Она возвращает строку с содержимым файла, но с переводом каждого слова на поросячью латынь, как в нашей функции `plword` в главе 2 «Строки». Возвращаемый перевод может игнорировать новые строки и не обязан обрабатывать прописные буквы и пунктуацию каким–либо особым образом.

Обсуждение

Мы видели, что вложенные генераторы списков можно использовать для итерации по сложным структурам данных. В данном случае мы итерируем содержимое файла. И действительно, мы могли бы итерировать каждую строку файла.

Но мы можем разбить задачу на части, используя вложенный генератор списка для итерации сначала каждой строки файла, а затем каждого слова в текущей строке. Тогда наша функция `plword` сможет работать с одним словом за раз.

Я знаю, что может быть очень трудно понять вложенные генераторы списков, по крайней мере, поначалу. Но со временем вы поймете, что они помогают элегантно разбить проблему на части.

В нашем решении есть небольшая проблема, хотя сначала она может показаться неочевидной. Как известно, генераторы списков создают списки. Это означает, что если мы переведем большой файл на поросячью латынь, то получим очень длинный список. Лучшим решением бы было вернуть итератор для экономии памяти, вычисляя только минимум, необходимый для каждой итерации.

Оказывается, сделать это довольно просто. Мы можем использовать выражение-генератор (как было предложено в первом упражнении этой главы), которое выглядит почти так же, как генератор списков, но с использованием круглых скобок, а не квадратных. Мы можем поместить выражение-генератор в вызов `str.join`, так же как и в генераторе списка, сэкономив при этом память.

Вот как будет выглядеть такой код:

```
def plfile (filename):
    return ' '.join ((plword (one_word)
          for one_line in open (filename)
          for one_word in one_line.split ()))
```

Но подождите — оказывается, если внутри вызова функции находится выражение-генератор, то оба набора круглых скобок не нужны. Можно оставить одну из них, что означает, что код будет выглядеть следующим образом:

```
def plfile (filename):
    return ' '.join (plword (one_word)
          for one_line in open (filename)
          for one_word in one_line.split ())
```

Теперь мы не только выполнили свою первоначальную задачу, но и использовали меньше памяти, чем требуется для генератора списка. Это может быть небольшой компромисс с точки зрения скорости, но обычно это оправданно, учитывая потенциальные проблемы, с которыми вы столкнетесь при считывании огромного файла в память за один раз.

Решение

```
def plword (word):
    if word [0] in 'aeiou':
         return word + 'way'
    return word [1:] + word [0] + 'ay'

def plfile (filename):
    return ' '.join (plword (one_word)
         for one_line in open (filename)
         for one_word in one_line.split ())
```

Итерации по каждой строке filename.

Итерация по каждому слову в текущей строке.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr153].


153

Обратите внимание, что поскольку Python Tutor не поддерживает работу с внешними файлами, я использовал экземпляр `StringIO` для симуляции файла.

Скринкаст решения


154

Посмотрите короткое видео с объяснением решения: [qr154].

После выполнения упражнения

При преобразовании и/или фильтрации сложных или вложенных структур данных, или (как в случае с файлом) того, что можно рассматривать как вложенную структуру данных, часто полезно использовать вложенный генератор списка:

1. В этом упражнении `plfile` применил функцию `plword` к каждому слову в файле. Напишите новую функцию `funcfile`, которая будет принимать два аргумента — имя файла и функцию. Выходом функции должна быть строка — результат вызова функции для каждого слова в текстовом файле. Вы можете рассматривать это как общую версию `plfile`, которая может возвращать любое строковое значение.
2. Используйте вложенный генератор списка для преобразования списка словарей в список двухэлементных кортежей (имя–значение), каждый из которых представляет одну из пар имя–значение в одном из словарей. Если более чем в одном кортеже есть одна и та же пара имя–значение, то кортеж должен появиться дважды.
3. Предположим, что у вас есть список словарей, в котором каждый словарь содержит две пары имя–значение: `name` и `hobbies`, где `name` — это имя человека, а `hobbies` — это множество строк с его увлечениями. Каковы три самых популярных хобби среди людей, перечисленных в словарях?

Упражнение 32. Переворачиваем словарь

Комбинация генераторов и словарей может быть довольно мощной. Вы можете захотеть изменить существующий словарь, удалив или изменив определенные элементы. Например, вы можете удалить всех пользователей, чей идентификационный номер меньше 500. Или вы можете захотеть найти ID всех пользователей, чьи имена начинаются с буквы «A».

Также нередко возникает необходимость перевернуть словарь, то есть поменять местами его ключи и значения. Представьте себе словарь, в котором ключами являются имена пользователей, а значениями — номера ID пользователей; может быть полезно перевернуть его, чтобы можно было искать по номеру ID.

Для этого упражнения сначала создайте словарь произвольного размера, в котором ключи и значения уникальны. (Ключ может быть значением, и наоборот). Пример:

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

Переверните словарь так, чтобы ключи и значения поменялись местами.

Обсуждение

Подобно тому, как генераторы списков обеспечивают простой способ создания списков на основе другого итерируемого объекта, генераторы словарей обеспечивают простой способ создания словаря на основе итерируемого объекта. Синтаксис выглядит следующим образом:

```
{KEY: VALUE
   for ITEM in ITERABLE}
```

Другими словами:

1. В основе генератора словаря лежит итерируемый объект — это, как правило, строка, список, кортеж, словарь, множество или файл.
2. Мы перебираем каждый такой элемент в цикле `for`.
3. Для каждого элемента мы выводим пару ключ–значение.

Обратите внимание, что двоеточие (`:`) отделяет ключ от значения. Это двоеточие является частью синтаксиса, что означает, что выражения по обе стороны от двоеточия оцениваются отдельно и не могут иметь общих данных.

В данном конкретном случае мы перебираем элементы словаря с именем `d`. Для этого мы используем метод `dict.items`, который при каждой итерации возвращает два значения — ключ и значение. Эти два значения передаются путем параллельного присвоения переменным `key` и `value`.

Другим способом решения этого упражнения является итерирование `d`, а не вывода `d.items ()`. Это дало бы нам ключи, требующие получения каждого значения:

```
{d [key]: key for key in d}
```

При помощи генератора я пытаюсь создать новый объект, основанный на старом. Все дело в значениях, возвращаемых выражением в начале генератора. Напротив, циклы `for` связаны с командами и выполнением этих команд.

Подумайте, какова ваша цель и что вам больше подходит: генератор или цикл `for`, например:

1. Для данной строки вам нужен список значений `ord` для каждого символа. Это должен быть генератор списка, потому что вы создаете список на основе строки, которая является итерируемой.
2. У вас есть список словарей, в каждом из которых содержатся имена и фамилии ваших друзей, и вы хотите внести эти данные в базу данных. В этом случае вы будете использовать обычный цикл `for`, потому что вас интересуют побочные эффекты, а не возвращаемое значение.

Решение

```
def flipped_dict (a_dict):
    return {value: key
            for key, value in a_dict.items ()}

print (flipped_dict ({'a':1, 'b':2, 'c':3}))
```

Все итерируемые объекты допустимы в генераторах, даже те, которые возвращают двухэлементные кортежи, такие как dict.items.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr155].

155

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr156].

156

После выполнения упражнения

Генераторы словарей предоставляет нам полезный способ создания новых словарей. Они обычно используются, когда вы хо–

тите создать словарь на основе итерируемого объекта, такого как список или файл. Мне особенно нравится использовать их, когда я хочу прочитать файл и преобразовать его содержимое в словарь. Вот несколько дополнительных идей, как попрактиковаться в использовании генераторов словарей:

1. Дана строка, содержащая несколько слов (разделенных пробелами), создайте словарь, в котором ключами будут слова, а значениями — количество гласных букв в каждом слове. Если строка имеет вид *this is an easy test*, то результирующий словарь будет иметь вид `{'this':1,'is':1, 'an':1, 'easy':2, 'test':1}`.
2. Создайте словарь, ключами которого являются имена файлов, а значениями — их размеры. В качестве входных данных может быть список файлов из `os.listdir` или `glob.glob`.
3. Найдите конфигурационный файл, в котором строки выглядят как «имя=значение». Используйте генератор словаря для чтения из файла, превращая каждую строку в пару «ключ–значение».

Упражнение 33. Преобразование переменных

Это упражнение должно продемонстрировать, как можно получить функцию в качестве аргумента функции и как генераторы могут помочь нам элегантно решать самые разные задачи.

Встроенная функция `map` принимает два аргумента: (a) функцию и (b) итерируемый объект. Она возвращает новую последовательность, которая является результатом применения функции к каждому элементу входного итерируемого объекта. Полное обсуждение `map` приведено в предыдущей сноске «map, filter и генераторы».

В этом упражнении мы создадим небольшую вариацию `map`, которая будет применять функцию к каждому из значений словаря. Результатом вызова этой функции, `transform_values`, является

новый словарь, ключи которого те же, что и у входного словаря, но значения которого были преобразованы функцией. (Название функции взято из Ruby on Rails, который предоставляет одноименную функцию). Функция, передаваемая в `transform_values`, должна принимать единственный аргумент — значение словаря.

Когда ваша функция transform_values заработает, вы сможете вызвать ее следующим образом:

```
d = {'a':1, 'b':2, 'c':3}
transform_values (lambda x: x*x, d)
```

Результатом этого вызова будет следующий словарь:

```
{'a': 1, 'b': 4, 'c': 9}
```

Обсуждение

Идея `transform_values` проста: вы хотите многократно вызывать функцию для значений словаря. Это означает, что вы должны перебирать пары ключ–значение. Для каждой пары вы хотите вызвать пользовательскую функцию для значения.

Мы знаем, что функции можно передавать в качестве аргументов, как и любые другие типы данных. В этом случае мы получаем функцию от пользователя, чтобы использовать ее. Мы используем круглые скобки, поэтому, если мы хотим вызвать функцию `func`, которую нам передал пользователь, мы просто пишем `func ()`. Или в данном случае, поскольку функция должна принимать один аргумент, мы напишем `func (value)`.

Мы можем перебирать пары ключ–значение в словаре с помощью `dict.items`, который возвращает итератор, предоставляющий одну за другой паруключ–значение в словаре. Но это не решает проблему, как взять эти пары ключ–значение и превратить их обратно в словарь.

Самый простой, быстрый и самый питоновский способ создания словаря на основе существующего итерируемого объекта — это использовать генератор словаря. Словарь, который мы возвращаем

из `transform_values`, будет иметь те же ключи, что и наш входной словарь. Но по мере итерации пар ключ–значение мы вызываем `func (value)`, применяя предоставленную пользователем функцию к каждому полученному значению и используя результат этого выражения в качестве нашего значения. Нам даже не нужно беспокоиться о том, какой тип значения вернет пользовательская функция, потому что значения словаря могут быть любого типа.

Решение

Применяет функцию, заданную пользователем, к каждому значению в словаре.

```
def transform_values (func, a_dict):
    return {key: func (value)
            for key, value in a_dict.items ()}

d = {'a':1, 'b':2, 'c':3}
print (transform_values (lambda x: x*x, d))
```

Итерируем каждую пару ключ-значение в словаре.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr157].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr158].

После выполнения упражнения

Генераторы словарей — это мощный инструмент в арсенале любого разработчика Python. Они позволяют нам создавать новые словари на основе существующих итерируемых объектов. Однако может потребоваться некоторое время, чтобы привыкнуть к ним и начать ими пользоваться. Вот несколько дополнительных упражнений для того, чтобы улучшить понимание и использование генераторов словарей:

1. Расширьте упражнение `transform_values`, пусть будет не один, а два аргумента функции. Первый аргумент

функции работает как и раньше, применяется к значению и выдает результат. Второй аргумент функции принимает два аргумента, ключ и значение, и определяет, будет ли вообще произведен вывод. То есть, вторая функция возвращает `True` или `False` и позволяет нам выборочно создать пару ключ–значение в конечном словаре.

2. Используйте генератор словарей для создания словаря, в котором ключами являются имена пользователей, а значениями — (целочисленные) ID пользователей, на основе файла `/etc/passwd` в стиле `Unix`. Подсказка: в типичном файле `/etc/passwd` имена пользователей являются первым полем в строке (т.е. индекс 0), а идентификаторы пользователей — третьим полем в строке (т.е. индекс 2). Если вам нужен пример файла `/etc/passwd`, вы можете скачать его по адресу [qr159].

3. Обратите внимание, что этот файл содержит комментарии, поэтому вам нужно будет удалить их при создании своего словаря.
4. Напишите функцию, которая принимает в качестве аргумента имя каталога (т.е. строку). Функция должна возвращать словарь, в котором ключами являются имена файлов в этом каталоге, а значениями — размеры файлов. Вы можете использовать `os.listdir` или `glob.glob` для получения файлов, но, поскольку размеры имеют только обычные файлы, вы захотите обработать результаты с помощью методов из `os.path`. Для определения размера файла вы можете использовать `os.stat` или (если хотите) просто проверить длину строки, полученной в результате чтения файла.

Упражнение 34.

Отчасти красота базовых структур данных Python заключается в том, что их можно использовать для решения самых разных задач. Но иногда, особенно поначалу, бывает непросто решить, ка–

кая из структур данных подходит и какой из их методов поможет вам легче всего решать задачи. Зачастую наилучшим способом оказывается комбинация методов.

В этом упражнении я хочу, чтобы вы написали функцию `get_sv`, которая возвращает множество всех «супервокальных» слов в словаре. Если вы никогда раньше не слышали термин супервокальный, вы не одиноки: я узнал о таких словах только несколько лет назад. Проще говоря, такие слова содержат все пять гласных в английском языке (a, e, i, o и u), каждая из которых появляется один раз и в алфавитном порядке.

Для целей данного упражнения я ослаблю определение, подойдет любое слово, содержащее все пять гласных в любом порядке и любое количество раз. Ваша функция должна найти все слова, которые соответствуют этому определению (т.е. содержат a, e, i, o и u), и вернуть множество этих слов.

Ваша функция должна принимать единственный аргумент: имя текстового файла, содержащего по одному слову в строке, как в словаре Unix/Linux. Если у вас нет такого файла words, вы можете загрузить его отсюда: [qr160].

Обсуждение

Прежде чем мы сможем создать множество супервокальных слов или считать их из файла, нам нужно найти способ определить, является ли слово супервокальным. (Опять же, это не точный термин). Как вариант, можно было бы использовать `in` пять раз, по одному разу для каждой гласной. Но это кажется немного экстремальным и неэффективным.

Вместо этого мы можем создать множество из нашего слова. В конце концов, строка — это последовательность, и мы всегда можем создать множество из любой последовательности благодаря встроенной функции `set`.

Хорошо, но как это нам поможет? Если у нас уже есть множество гласных, мы можем проверить, все ли они содержатся в слове, с помощью оператора `<`. Обычно `<` проверяет, меньше ли одна

точка данных, чем другая. Но в случае с множествами, он возвращает `True`, если элемент слева является подмножеством элемента справа.

Это означает, что для слова *superlogical* я могу сделать следующее:

```
vowels = {'a', 'e', 'i', 'o', 'u'}
word = 'superlogical'

if vowels < set (word):
   print ('Да, это супервокальное слово!')
else:
   print ('Нет, это обычное слово')
```

Это подойдет для одного слова. Но как сделать это для большого числа слов в файле? Вы могли бы использовать генератор списка. В конце концов, мы можем рассматривать наш файл как итератор, который возвращает строки. Если файл `words` содержит одно слово в строке, то итерация по строкам файла означает итерацию по разным строкам. Если множество гласных является множеством, созданным из текущего слова, то мы будем считать его супервокальным и включим текущее слово в выходной список.

Но нам нужен не список, а множество! К счастью, разница между созданием генератороа списка и генератора множества заключается в паре скобок. Мы используем квадратные скобки (`[]`) для генератора списка и фигурные скобки (`{}`) для генератора множества. Генератор с фигурными скобками и двоеточием — это генератор словаря, а без двоеточия — генератор множества. Подводя итог:

1. Мы перебираем строки файла.
2. Превращаем каждое слово в множество и проверяем, что гласные являются подмножеством букв нашего слова.
3. Если слово проходит эту проверку, мы включаем его (слово) в вывод.
4. Все выходные данные помещаются в множество благодаря генератору множества.

Использование множеств в качестве основы для текстовых сравнений может показаться неочевидным, по крайней мере, сначала. Но полезно научиться мыслить подобным образом, используя преимущества структур данных Python таким образом, о котором вы раньше не задумывались.

Решение

```
def get_sv (filename):
vowels = {'a', 'e', 'i','o', 'u'}

return {word.strip ()
for word in open (filename)
if vowels < set (word.lower ())}
```

Создает множество гласных.

Возвращает слово без пробелов по обеим сторонам.

Итерируем каждую строку в filename.

Содержит ли это слово все гласные?

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr161]. Обратите внимание, что поскольку Python Tutor не поддерживает работу с внешними файлами, то я использовал экземпляр `StringIO` для симуляции файла.

161

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr162].

162

После выполнения упражнения

Генераторы множеств отлично подходят в различных обстоятельствах, в том числе когда у вас есть входные данные, и вы хотите сократить их, чтобы получить только отдельные (уникальные) элементы.

Вот несколько дополнительных способов попрактиковаться с генератором множеств:

1. В файле `/etc/passwd`, который вы использовали ранее, какие различные оболочки (т.е. командные интерпрета–

торы, названные в последнем поле каждой строки) назначены пользователям? Используйте генератор множества, чтобы собрать их.
2. У нас есть текстовый файл, какова длина различных слов? Верните множество различных размеров слов в файле.
3. Создайте список, элементами которого являются строки, содержащие имена людей в вашей семье. Теперь используйте генератор множества (а еще лучше — вложенный генератор множества), чтобы найти, какие буквы используются в именах членов вашей семьи.

Упражнение 35a. Гематрия, часть 1

В этом упражнении мы снова попробуем что-то, что находится на пересечении строк и генераторов. На этот раз речь пойдет о генераторах словарей.

Когда вы были маленьким, вы могли создать или использовать «секретный» код, в котором `a` было 1, `b` было 2, `c` было 3 и так далее, до `z` (которое было 26). Этот тип кода является довольно древним и использовался несколькими различными группами более 2000 лет назад. «Гематрия», как она известна на иврите, — это способ, с помощью которого издавна нумеровались библейские стихи. И, конечно же, не стоит называть это секретным кодом, несмотря на то что вы могли думать так в детстве.

В этом упражнении, результат которого вы будете использовать в следующем, вам предлагается создать словарь, ключами которого являются (строчные) буквы английского алфавита, а значениями — числа от 1 до 26. И да, вы можете просто ввести `{'a':1, 'b':2, 'c':3}` и так далее, но я бы хотел, чтобы вы использовали генератор словаря.

Обсуждение

В решении вы будете использовать ряд различных аспектов Python, комбинируя их для создания словаря с минимальным количеством кода.

Во–первых, мы хотим создать словарь и поэтому обратимся к генератору словаря. Нашими ключами будут строчные буквы английского алфавита, а значениями — числа от 1 до 26.

Мы могли бы создать строку из строчных букв. Но вместо того чтобы делать это самостоятельно, мы можем положиться на модуль `string` и его атрибут `string.ascii_lowercase`, который очень полезен в таких ситуациях.

Но как мы можем пронумеровать буквы? Мы можем использовать встроенный итератор `enumerate`, который будет нумеровать наши символы по одному. Затем мы можем перехватить итерируемые кортежи с помощью распаковки, захватывая отдельно индекс и символ:

```
{char: index
for index, char in enumerate (string.ascii_lowercase)}
```

Единственная проблема в том, что `enumerate` начинает отсчет с 0, а мы хотим начать отсчет с 1. Конечно, мы могли бы просто добавить 1 к значению `index`. Однако мы можем сделать еще лучше, попросив `enumerate` начать считать с 1, и мы сделаем это, передав ему 1 в качестве второго аргумента:

```
{char: index
for index, char in enumerate (string.ascii_lowercase, 1)}
```

И, конечно, это дает нужный нам словарь. Мы будем использовать его в следующем упражнении.

Решение

```
import string

def gematria_dict ():
    return {char: index
            for index, char
                in enumerate (string.ascii_lowercase,1)}
print (gematria_dict ())
```

Возвращает пару ключ-значение, содержащую символ и целое число.

Итерация по строчным буквам с помощью enumerate.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr163].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr164].

После выполнения упражнения

Словари также известны как пары ключ–значение по той простой причине, что они содержат ключи и значения, а также потому, что ассоциации между двумя разными типами данных чрезвычайно распространены в программировании. Зачастую после помещения данных в словарь с ними становится легче работать и управлять ими. По этой причине важно знать, как добавлять информацию в словарь из различных форматов и источников. Вот несколько дополнительных упражнений, чтобы попрактиковаться в этом:

1. Функциональность многих программ изменяется с помощью конфигурационных файлов, которые часто задаются с помощью пар «имя–значение». То есть каждая строка файла содержит текст в виде `name=value`, где знак `=` отделяет имя от значения. Я подготовил пример файла конфигурации здесь [qr165]. Загрузите этот файл, а затем используйте генератор словаря, чтобы прочитать его содержимое с диска, превратив его в словарь, описывающий предпочтения пользователя. Обратите внимание, что все значения будут строками.

2. Создайте словарь на основе файла конфигурации, как в предыдущем упражнении, но на этот раз все значения должны быть целыми числами. Это означает, что вам нужно отфильтровать (и игнорировать) те значения, которые нельзя преобразовать в целые числа.
3. Иногда бывает полезно преобразовать данные из одного формата в другой. Загрузите список 1000 крупнейших го–

родов США в формате JSON по ссылке [qr166]. Используя генератор словаря, превратите его в словарь, в котором ключи — это названия городов, а значения — население этих городов. Почему в этом словаре всего 925 пар ключ–значение? Теперь создайте новый словарь, но установите для каждого ключа кортеж, содержащий штат и город. Гарантирует ли это, что будет 1000 пар ключ–значение?

Упражнение 35b. Гематрия, часть 2

В предыдущем упражнении вы создали словарь, который позволяет получить числовое значение из любой строчной буквы. Как вы понимаете, мы можем использовать этот словарь не только для нахождения числового значения одной буквы, но и для суммирования значений букв в слове, таким образом получая «значение» слова. Одна из игр, в которую любят играть еврейские мистики (хотя они, вероятно, пришли бы в ужас, если бы услышали, что я называю это игрой), состоит в том, чтобы найти слова с таким же значением гематрии. Если два слова имеют одинаковое значение гематрии, то они каким–то образом связаны.

В этом упражнении вы напишете две функции:

1. `gematria_for`, которая принимает одно слово (строку) в качестве аргумента и возвращает значение гематрии для этого слова.
2. `gematria_equal_words`, которая принимает одно слово и возвращает список тех слов из словаря, чьи значения гематрии совпадают со значениями текущего слова.

Например, если функция вызывается со словом `cat`, значение гематрии которого равно 24 (3 + 1 + 20), то функция вернет список строк, все значения гематрии которых также равны 24. (Это будет длинный список!) Любые нестрочные символы в вводе пользователя должны учитываться как 0 в нашей окончательной оценке слова. Источником слов для словаря слов будет файл

Unix, который вы использовали ранее в этой главе и который вы можете загрузить в генератор списков.

Обсуждение

Это решение сочетает в себе большое количество методов, которые мы уже обсуждали в этой книге и которые вы, вероятно, будете использовать в своей работе по программированию на Python. (Правда, я надеюсь, что вы не слишком много занимаетесь вычислением гематрии).

Во-первых, как мы рассчитываем гематрическую оценку для слова, учитывая наш gematria словарь? Мы хотим перебирать каждую букву в слове, беря значение из словаря.

И если буквы нет в словаре, мы присвоим ей значение 0.

Стандартный способ сделать это — использовать цикл for и dict.get:

```
total = 0
for one_letter in word:
   total += gematria.get (one_letter, 0)
```

И в этом нет ничего плохого как такового. Но генератор обычно является лучшим выбором, когда вы начинаете с одной итерации и пытаетесь создать другую итерацию. И в этом случае мы можем перебирать буквы в нашем слове в генераторе списка, вызывая sum для списка целых чисел, в результате получим:

```
def gematria_for (word):
   return sum (gematria.get (one_char,0)
        for one_char in word)
```

Как только мы сможем вычислить гематрию для одного слова, нам нужно найти все слова из словаря, которые ему эквивалентны. Мы снова можем сделать это с помощью генератора списка — на этот раз, используя условие if для фильтрации тех слов, чьи значения гематрии не равны:

```
def gematria_equal_words (word):
    our_score = gematria_for (input_word.lower ())
    return [one_word.strip ()
          for one_word in open ('/usr/share/dict/
          words')
          if gematria_for (one_word.lower ()) ==
          our_score]
```

Как видите, нам нужны слова в нижнем регистре. Но мы не изменяем и не преобразуем слово в первой строке нашего генератора. Скорее, мы просто фильтруем.

Тем временем мы итеративно перебираем все слова в файле словаря. Каждое слово в этом файле заканчивается новой строкой, что не влияет на наше значение гематрии, но это не то, что мы хотим вернуть пользователю в нашем генераторе списка.

Наконец, это упражнение демонстрирует, что, когда вы используете комплексный подход и ваше выходное выражение является сложным, часто хорошей идеей является создание отдельной функции, которую можно вызывать многократно.

Решение

```
import string

def gematria_dict ():
    return {char: index
          for index, char
          in enumerate (string.ascii_lowercase,1)}

GEMATRIA = gematria_dict ()
```

Получает значение для текущего символа, или 0, если символ не находится в словаре GEMATRIA.

```
def gematria_for (word):
return sum (GEMATRIA.get (one_char, 0)
          for one_char in word)
```

Перебирает символы в word.

```
def gematria_equal_words (input_word):
    our_score = gematria_for (input_word.
    lower ())
    return [one_word.strip ()
            for one_word in
            open ('/usr/share/dict/words')
    if gematria_for (one_word.lower ()) ==
            our_score]
```

Получает общую оценку для введенного слова.

Удаляет ведущие и последующие пробельные символы из one_word.

Итерации по каждому слову в англоязычном словаре.

Добавляет текущее слово в наш список только в том случае, если его гематрия совпадает с нашей.

Примечание: для этого упражнения нет ссылки на Python Tutor, поскольку в нем используется внешний файл.

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr167].

После выполнения упражнения

Когда у вас есть данные в словаре, вы часто можете использовать генератор для их преобразования различными способами. Вот несколько дополнительных упражнений, которые вы можете использовать, чтобы потренировать свои навыки работы со словарями и генераторами словарей:

1. Создайте словарь, ключи которого — названия городов, а значения — температуры в градусах Фаренгейта. Теперь используйте генератор словаря, чтобы преобразовать этот словарь в новый, сохранив старые ключи, но превратив значения в температуру в градусах Цельсия.
2. Создайте список кортежей, в котором каждый кортеж содержит три элемента: (1) имя и фамилию автора, (2) название книги и (3) цену книги в долларах США.
3. Используйте генератор словаря для превращения в словарь, ключами которого являются названия книг, значени-

ями — другие (под–) словари, с ключами — (a) имя автора, (b) фамилия автора и (c) цена книги в долларах США.

4. Создайте словарь, ключами которого будут названия валют, а значениями — значение этой валюты в долларах США. Напишите функцию, которая спрашивает пользователя, какую валюту он использует, а затем возвращает словарь из предыдущего упражнения, как и раньше, но с ценами, преобразованными в запрашиваемую валюту.

Подводя итоги

Генераторы, без сомнения, являются одной из самых сложных тем для изучения в Python. Синтаксис немного странный, и даже не совсем очевидно, где и когда использовать генераторы. В этой главе вы увидели множество примеров того, как и когда использовать генераторы, что, надеюсь, поможет вам не только использовать их, но и понимать, когда и где.

8. Модули и пакеты

Функциональное программирование, которое мы рассмотрели в предыдущей главе, — одна из самых запутанных тем, с которыми вы столкнетесь в мире программирования. Я рад сообщить вам, что эта глава о модулях Python станет разительным контрастом и будет одной из самых простых в этой книге. Модули важны, при этом их очень просто создавать и использовать. Так что, если вы обнаружите, что читаете эту главу и думаете: «Эй, это довольно очевидно», что ж, ничего страшного.

Что такое модули в Python и как они нам помогают? Я уже несколько раз упоминал в этой книге аббревиатуру DRY, сокращение от «не повторяйся». Как программисты, мы стремимся «высушить» (прим. перев. dry — сушить) наш код, беря одинаковые фрагменты кода и используя их несколько раз. Это упрощает понимание, управление и поддержку нашего кода. Тестирование такого кода тоже упрощается.

Когда у нас есть повторяющийся код в одной программе, мы можем избавить его от повторов, написав функцию, а затем повторно вызвав ее. Но что, если у нас есть повторяющийся код, который используется в нескольких программах? Мы можем создать библиотеку или, как говорят в мире Python, модуль.

Модули на самом деле выполняют две функции в Python. Во–первых, они позволяют нам повторно использовать код в разных программах, помогая нам улучшить возможность повторного использования и удобство сопровождения нашего кода. Таким образом, мы можем один раз определить функции и классы, вставить их в модуль и повторно использовать сколько

угодно раз. Это не только уменьшает объем работы, которую нам необходимо выполнить при внедрении новой системы, но и снижает нашу когнитивную нагрузку, поскольку нам не нужно беспокоиться о деталях реализации.

Например, предположим, что ваша компания разработала специальную формулу ценообразования, содержащую данные о погоде и индексы фондового рынка. Вы захотите использовать эту формулу ценообразования во многих частях вашего кода. Вместо того чтобы повторять код, вы можете определить функцию один раз, поместить ее в модуль, а затем использовать этот модуль везде в вашей программе, где вы хотите вычислять и отображать цены.

В модуле можно определить любой объект Python — от простых структур данных до функций и классов. Главный вопрос заключается в том, хотите ли вы, чтобы он был общим для нескольких программ, сейчас или в будущем.

Во-вторых, модули — это способ создания пространств имен в Python. Если два человека совместно работают над программным проектом, вы же не хотите беспокоиться о коллизии между выбранными ими именами переменных и функций? Каждый файл — то есть модуль — имеет свое собственное пространство имен, что гарантирует отсутствие конфликтов между ними.

Python содержит большее количество модулей, и даже самая маленькая нетривиальная программа Python будет содержать `import`, чтобы использовать один или несколько из них. В дополнение к стандартной библиотеке, как известно, программисты Python могут воспользоваться большим количеством модулей, доступных в Python Package Index. В этой главе мы рассмотрим использование и создание модулей, включая пакеты.

ПОДСКАЗКА Если вы ознакомитесь с PyPI по ссылке [qr168], вы обнаружите, что количество сторонних пакетов, предоставляемых сообществом, поразительно велико. На момент написания этой книги на PyPI насчитывалось более 200 000 пакетов, многие из которых содержали ошибки или не поддерживались. Как узнать,

какой из этих пакетов стоящий, а какой нет? Сайт Awesome Python [qr169] является попыткой исправить эту ситуацию и содержит отредактированные списки известных стабильных, поддерживаемых пакетов по различным темам. Советую проверять здесь пакеты, прежде чем обращаться к PyPI. Хотя это и не гарантирует, что пакет, который вы используете, будет отличным, это определенно повышает шансы на то, что он таким и будет.


169

Таблица 8.1. Что вам нужно знать

Понятие	Что это?	Пример	Чтобы узнать подробнее
`import`	Импортирование модулей.	`import os`	
`from X import Y`	Импортирует модуль X, но определяет только Y как глобальную переменную.	`from os import sep`	
`importlib.reload`	Повторно импортирует уже загруженный модуль, обычно для обновления определений в процессе разработки.	`importlib.reload (mymod)`	
`pip`	Программа командной строки для установки пакетов из PyPI.	`pip install packagename`	
`Decimal`	Класс, который точно обрабатывает числа с плавающей точкой.	`from decimal import Decimal`	

Импортирование модулей

Одна из крылатых фраз в мире Python — «батарейки в комплекте». Это относится к многочисленным телевизионным рекламным роликам, которые я видел в детстве и которые тратили первые 29,5 секунд на то, чтобы убедить нас купить их захватывающие, забавные, красивые игрушки… И только в последние полсекунды говорили: «Батарейки в комплект не входят» — это означало, что для того, чтобы наслаждаться продуктом, недостаточно купить его, нужно еще купить батарейки.

Под «батарейками в комплекте» подразумевается тот факт, что, когда вы скачиваете и устанавливаете Python, у вас есть все необходимое для работы. Это не совсем так, как было раньше, и PyPI (Python Package Index, описан отдельно в этой главе) предоставляет нам огромную коллекцию сторонних модулей Python, которые мы можем использовать для улучшения наших продуктов. Но факт остается фактом: *стандартная библиотека*, то есть то, что поставляется с Python при его установке, включает огромное количество модулей, которые мы можем использовать в наших программах.

Наиболее часто используемые вещи в стандартной библиотеке, такие как списки и словари, встроены в язык благодаря пространству имен, известному как `builtins` (или встроенное пространство имен). Вам не нужно беспокоиться об импорте вещей из модуля `builtins` благодаря правилу LEGB, о котором я рассказывал в главе 6. Но все остальное из стандартной библиотеки должно быть загружено в память, прежде чем его можно будет использовать.

Мы загружаем такой модуль с помощью `import`. Простейшая версия оператора `import` выглядит следующим образом:

```
import MODULENAME
```

Например, если я хочу использовать модуль `os`, то я пишу

```
import os
```

Обратите внимание на пару вещей:

Во–первых, это не функция, вы не говорите `import (os)`, скорее `import os`.

Во–вторых, мы не импортируем имя файла. Скорее, мы указываем переменную, которую хотим определить, а не файл, который должен быть загружен с диска. Поэтому не пытайтесь `импортировать os` или даже `импортировать os.py`. Как `def` определяет новую переменную, которая ссылается на функцию, так и `import` определяет новую переменную, которая ссылается на модуль.

Когда вы пишите `import os`, Python пытается найти файл, который соответствует имени переменной, которую вы определяете. Обычно он ищет `os.py` и `os.pyc`, где первый — это исходный код, а второй — скомпилированная версия. (Python использует временную метку файловой системы, чтобы определить, какая из них новее, и при необходимости создает новую байт–компилированную версию. Так что не беспокойтесь о компиляции!)

Python ищет подходящие файлы в ряде каталогов, видимых вам в `sys.path`. Это список строк, представляющих каталоги: Python будет перебирать имена каждому каталогу, пока не найдет подходящее имя модуля. Если в нескольких каталогах содержится модуль с одинаковым именем, то загружается первый попавшийся модуль, а все последующие модули полностью игнорируются. По моему опыту, это часто приводит к путанице и конфликтам, поэтому старайтесь выбирать нетривиальные, но понятные имена для своих модулей.

У `import` есть различные вариации, которые полезно знать и которые вы, вероятно, увидите в представленном коде, а также будете использовать в своем собственном.

Тем не менее конечная цель одна и та же: загрузить модуль и определить одно или несколько имен, связанных с модулем, в пространстве имен.

Если вас устраивает загрузка модуля и использование его имени в качестве переменной, то `import MODULENAME` — отличное решение. Но иногда имя слишком длинное. По этой причине вы захотите дать имени модуля псевдоним. Вы можете сделать это с помощью

```
import mymod as mm
```

Когда вы используете `as`, имя `mymod` не будет определено. Однако имя `mm` будет определено. Это излишне, если имя модуля короткое. Но если имя длинное или вы собираетесь часто ссылаться на него, то вы вполне можете захотеть дать ему более короткий псевдоним. Классическим примером является NumPy, который лежит в основе всех научных и численных вычислительных систем Python, включая науку о данных и машинное обучение. Этот модуль обычно импортируется с псевдонимом `np`:

```
import numpy as np
```

После импорта модуля все имена, которые были определены в глобальной области видимости файла, становятся доступны в качестве атрибутов через объект модуля. Например, модуль `os` определяет `sep`, который указывает, какая строка разделяет элементы пути к каталогу. Вы можете получить доступ к этому значению при помощи `os.sep`. Но если вы собираетесь использовать его часто, то постоянно писать `os.sep` будет немного утомительно. Не лучше ли просто называть его `sep`? Конечно, вы не можете этого сделать, потому что имя `sep` будет переменной, тогда как `os.sep` — это атрибут.

Однако вы можете устранить разрыв и получить загрузку атрибута, используя следующий синтаксис:

```
from os import sep
```

Обратите внимание, что это не определит переменную `os`, но определит переменную `sep`. Вы можете использовать `from.. import` для более чем одной переменной:

```
from os import sep, path
```

Теперь и `sep`, и `path` будут определены как переменные в глобальной области видимости.

Беспокоитесь о том, что один из этих импортированных атрибутов будет конфликтовать с существующей переменной, методом или именем модуля? Тогда вы можете использовать `from.. import.. as`:

```
from os import sep as s
```

Существует еще одна версия, которую я часто встречаю и которую я обычно советую людям не использовать. А именно:

```
from os import *
```

Это загрузит модуль `os` в память, но (что более важно) возьмет все атрибуты из `os` и определит их как глобальные переменные в текущем пространстве имен. Учитывая, что мы обычно хотим избегать глобальные переменные без необходимости, я вижу проблему, когда мы позволяем модулю решать, какие переменные должны быть определены.

ПРИМЕЧАНИЕ Не все имена из модуля будут импортированы с помощью `import *`. Имена, начинающиеся

с `_` (подчеркивания), будут игнорироваться. Более того, если модуль определяет список строк с именем `__all__`, то только имена, указанные в модуле, будут загружены с помощью `import *`. Однако `from X import Y` всегда будет работать, независимо от того, определено ли `__all__`.

В конце концов, `import` делает функции, классы и данные доступными для вас в вашем текущем пространстве имен. Учитывая огромное количество модулей, доступных как в стандартной библиотеке Python, так и на PyPI, это дает вам огромную потенциальную мощь и объясняет, почему так много программ на Python начинаются с нескольких строк импорта.

Упражнение 36. Налог с продаж

Модули позволяют нам сосредоточиться на высокоуровневом мышлении и не копаться в деталях реализации сложной функциональности. Таким образом, мы можем реализовать функцию один раз, поместить ее в модуль и использовать его много раз для реализации алгоритмов, о которых мы не хотим думать на ежедневной основе. Если бы, например, для создания веб-приложения вам пришлось разбираться и продираться через вычисления, связанные с интернет-безопасностью, вы бы никогда не закончили работу.

В этом упражнении вам предстоит реализовать в модуле несколько сложную (и причудливую) функцию для реализации налоговой политики в Республике Фридония. Идея заключается в том, что налоговая система настолько сложна, что правительство предоставит предприятиям модуль Python, выполняющий расчеты за них.

Налог с продаж на покупки во Фридонии зависит от того, где была совершена покупка, а также от времени покупки. Фридония состоит из четырех провинций, каждая из которых взимает свой процент налога:

1. Чико: 50%
2. Гроучо: 70%
3. Харпо: 50%
4. Зеппо: 40%

Да, в Фридонии довольно высокие налоги (настолько высокие, что говорят, что у них марксистское правительство). Однако эти налоги редко применяются в полном объеме. Это происходит потому, что размер налога зависит от часа, в который совершается покупка. Процент налога всегда умножается на час, в который была совершена покупка. В полночь (т.е. когда 24–часовые часы показывают 0) налог с продаж не взимается. С 12 часов дня до 13 часов дня применяется только 50% (12/24) налога. А с 11 часов вечера до полуночи применяется 95,8% (т.е. 23/24) налога.

Ваша задача — реализовать модуль Python `freedonia.py`. Он должен предоставлять функцию `calculate_tax`, принимающую три аргумента: сумму покупки, область, в которой была совершена покупка, и час (целое число от 0 до 24), в который это произошло. Функция `calculate_tax` должна возвращать конечную цену в виде числа типа `float`.

Таким образом, если я вызову функцию

```
calculate_tax (500, 'Harpo', 12
```

то покупка на 500 долларов в области Харпо (с 50% налогом) стоила бы 750 долларов. Однако, поскольку покупка была совершена в 12 часов дня, налог составляет только половину от максимального, или $125, что в сумме составляет $625. Если бы покупка была совершена в 9 часов вечера (то есть в 21:00 по 24–часовым часам), то налог составил бы 87,5% от его полной ставки, или 43,75%, при общей цене $718,75.

Более того, я хочу, чтобы вы написали это решение, используя два отдельных файла. Функция `calculate _tax`, а также все вспомогательные данные и функции должны находиться в файле `freedonia.py`, модуле Python.

Программа, вызывающая `calculate_tax`, должна находиться в файле `use_freedonia.py`, который затем использует `import` для загрузки функции.

Обсуждение

Модуль `freedonia` делает именно то, что должен делать модуль Python. А именно: он определяет структуры данных и функции, которые обеспечивают функциональность одной или нескольких других программ. Предоставляя этот уровень абстракции, он позволяет программисту сосредоточиться на том, что для него важно, например, на реализации интернет-магазина, не беспокоясь о тонкостях и деталях.

В то время как некоторые страны имеют чрезвычайно простые системы расчета налога с продаж, в других странах, например, в США, существует множество пересекающихся юрисдикций, каждая из которых применяет свой собственный налог с продаж, часто по разным ставкам и на разные типы товаров. Таким образом, хотя пример Фридонии несколько выдуман, приобретение или использование библиотек для расчета налогов не является чем-то необычным.

Наш модуль определяет словарь (`RATES`), в котором ключами являются области Фридонии, а значениями — налоговые ставки, которые должны там применяться. Таким образом, мы можем узнать ставку налогообложения в области Гроучо с помощью `RATES['Groucho']`. Или мы можем попросить пользователя ввести название области в переменную `province`, а затем получить `RATES[province]`. В любом случае это даст нам число с плавающей точкой, которое мы можем использовать для расчета налога.

При расчете фридонского налогообложения возникает проблема, связанная с тем, что с каждым днем налоги становятся все выше и выше. Чтобы облегчить этот расчет, я написал функцию `time_percentage`, которая просто берет час и возвращает его в процентах от 24 часов.

> **ПРИМЕЧАНИЕ** В Python 2 целочисленное деление всегда возвращает целое число, даже если это означает отбрасывание остатка. Если вы используете Python 2, убедитесь, что текущий час делится не на 24 (целое число), а на 24.0 (число с плавающей точкой).

Наконец, функция `calculate_tax` принимает три параметра: сумму продажи, название области, в которой произошла продажа, и час, в который произошла продажа, и возвращает число с плавающей точкой, указывающее фактическую ставку налога на текущий момент.

Десятичная версия

Если вы действительно занимаетесь вычислениями, связанными с серьезными деньгами, то почти наверняка вам не следует использовать числа типа `float`. Скорее, вам следует использовать целые числа или класс `Decimal`, оба из которых являются более точными. (Дополнительную информацию о неточности чисел с плавающей точкой см. в главе 1). Я хотел, чтобы это упражнение было посвящено созданию модуля, а не использованию класса `Decimal`, поэтому я не настаиваю на его использовании.

Вот как выглядело бы мое решение с использованием `Decimal`:

```
from decimal import Decimal

rates = {
    'Chico': Decimal ('0.5'),
    'Groucho': Decimal ('0.7'),
    'Harpo': Decimal ('0.5'),
    'Zeppo': Decimal ('0.4')
}

def time_percentage (hour):
   return hour / Decimal ('24.0')

def calculate_tax (amount, state, hour):
         return float (amount + (amount *rates
         [state] * time_percentage (hour)))
```

Обратите внимание, что этот код использует `Decimal` для строк, а не для плавающих чисел, чтобы обеспечить максимальную точность. В последний возможный момент мы возвращаем число с плавающей точкой. Также обратите внимание, что любое значение `Decimal`, умноженное или деленное на число, остается `Decimal`, поэтому нам нужно выполнить преобразование только в конце.

Вот программа, использующая наш модуль `freedonia`:

```
from freedonia import calculate_tax

tax_at_12noon = calculate_tax (100, 'Harpo', 12)
tax_at_9pm = calculate_tax (100, 'Harpo', 21)

print (f'You owe a total of: {tax_at_12noon}')
print (f'You owe a total of: {tax_at_9pm}')
```

Проверка ошибок питоновским способом

Поскольку модуль будет использоваться многими другими программами, важно, чтобы он был не только точным, но и имел достойную проверку ошибок. В нашем конкретном случае, например, мы хотим проверить, что час находится между 0 и 24.

Сейчас тот, кто передаст нашей функции недопустимый час, все равно получит ответ, хотя и бессмысленный. Лучшим решением было бы заставить функцию вызывать исключение, если входные данные недействительны. И хотя мы можем вызвать встроенное исключение Python (например, `ValueError`), обычно лучше создать свой собственный класс исключений и вызвать его, например:

```
class HourTooLowError (Exception): pass
class HourTooHighError (Exception): pass
```

```
def calculate_tax (amount, state, hour):

        if hour < 0:
           raise HourTooLowError (f'Hour of
           {hour} is < 0')

        if hour >= 24:
           raise HourTooHighError (f'Hour of
           {hour} is >= 24')

        return amount + (amount * rates [state]
        * time_percentage (hour))
```

Добавление таких исключений в ваш код считается очень питоничным и гарантирует, что любой, кто использует ваш модуль, не получит случайно плохой результат.

Решение

```
RATES = {
    'Chico': 0.5,
    'Groucho': 0.7,
    'Groucho': 0.7,
    'Zeppo': 0.4
    }

def time_percentage (hour):
        return hour / 24

def calculate_tax (amount, state, hour):
   return amount + (amount * rates [state] *
   time_percentage (hour))
print (calculate_tax (500, 'Harpo', 12))
```

Это означает, что мы получим 0% в полночь и чуть меньше 100% в 23:59.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr175].

175

Обратите внимание, что сайт Python Tutor не поддерживает модули, поэтому данное решение было помещено в один файл без использования `import`.

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr176].

После выполнения упражнения

Теперь, когда вы написали простую функцию, скрывающую более сложную функциональность, вот несколько других функций, которые вы можете написать в виде модулей:

1. Подоходный налог во многих странах представляет собой не фиксированный процент, а скорее, комбинацию различных «скобок». Таким образом, страна может не облагать налогом ваши первые 1000 долларов дохода, затем 10% со следующих 10 000 долларов, затем 20% со следующих 10 000 долларов, а затем 50% со всего, что выше этой суммы. Напишите функцию, которая берет чей-то доход и возвращает сумму налога, которую ему придется заплатить, суммируя проценты из разных скобок.
2. Напишите модуль, предоставляющий функцию, которая по заданной строке возвращает словарь, указывающий, сколько символов содержит результат `True` для каждой из следующих функций: `str.isdigit`, `str.isalpha` и `str.isspace`. Ключами должны быть `isdigit`, `isalpha` и `isspace`.
3. Метод `dict.fromkeys` упрощает создание нового словаря. Например, `dict.fromkeys ('abc')` создаст словарь `{'a': None, 'b': None, 'c': None}`. Вы также можете передать значение, которое будет присвоено каждому ключу, например, `dict.fromkeys ('abc', 5)`, в результате чего получится словарь `{'a':5, 'b':5, 'c':5}`. Реализуйте функцию, которая делает то же самое, что и `dict.keys`, но вторым аргументом которой яв-

ляется функция. Значение, связанное с ключом, будет результатом вызова `f (key)`.

Загрузка и перезагрузка модулей

Что происходит, когда вы используете import для загрузки модуля? Например, если вы напишите

```
import mymod
```

Python найдет `mymod.py` в нескольких каталогах, определенных в списке строк с именем `sys.path`. Если Python встречает файл в одном из этих каталогов, он загружает файл и прекращает поиск в любых других каталогах.

ПРИМЕЧАНИЕ Существует несколько способов изменить `sys.path`, в том числе путем установки переменной среды `PYTHONPATH` и создания файлов с расширением `.pth` в каталоге пакетов установки Python `site-packages`. Дополнительные сведения о настройке `sys.path` смотрите в документации Python или в этой полезной статье: [qr177].

177

Это означает, что `import` обычно делает две разные вещи: загружает модуль и определяет новую переменную. Но что произойдет, если ваша программа загрузит два модуля, каждый из которых, в свою очередь, загрузит еще модули? Например, предположим, что ваша программа импортирует как `pandas`, так и `scipy`, оба из которых загружают модуль `numpy`. В таком случае Python загрузит модуль в первый раз, но определит переменную только во второй раз. `import` загружает модуль только один раз, но всегда будет определять переменную, которую вы попросили создать.

Это делается с помощью словаря, определенного в `sys`, который называется `sys.modules`. Его ключи — это имена загруженных модулей, а его значения — фактические объекты модуля. Таким образом, когда мы пишим `import mymod`, Python сначала проверяет, находится ли `mymod` в `sys.modules`. Если это так, то он не ищет и не загружает модуль. Скорее, просто определяет имя.

Обычно это очень хорошо, поскольку нет необходимости перезагружать модуль после того, как программа начала работать. Но когда вы отлаживаете модуль в рамках интерактивного сеанса Python, вы хотите иметь возможность неоднократно перезагружать его, желательно без выхода из текущего сеанса Python.

В таких случаях вы можете использовать функцию `reload`, определенную в модуле `importlib`. Она принимает объект модуля в качестве аргумента, поэтому модуль уже должен быть определен и импортирован. И это то, что вы, вероятно, будете использовать все время в разработке и почти никогда в реальном производстве.

ПРИМЕЧАНИЕ В предыдущих версиях Python `reload` была встроенной функцией. Начиная с Python 3, она находится в модуле `importlib`, который вы должны импортировать, чтобы использовать ее.

УПРАЖНЕНИЕ 37. Меню

Если вы обнаружите, что пишете одну и ту же функцию несколько раз в разных программах или проектах, вы почти наверняка захотите превратить эту функцию в модуль. В этом упражнении вы напишете функцию, достаточно общего характера, чтобы ее можно было использовать в самых разных программах.

В частности, напишите новый модуль под названием menu (в файле `menu.py`). Модуль должен определять функцию, также

называемую `menu`. Функция принимает любое количество пар ключ–значение в качестве аргументов. Каждое значение должно быть *вызываемым* — с причудливым именем для функции или класса в Python.

Когда функция вызывается, пользователю предлагается ввести некоторые данные. Если пользователь вводит строку, которая соответствует одному из аргументов ключевого слова, будет вызвана функция, связанная с этим ключевым словом, и ее возвращаемое значение будет возвращено вызывающей стороне `menu`. Если пользователь вводит строку, которая не является одним из аргументов ключевого слова, ему будет выдано сообщение об ошибке и предложено повторить попытку.

Идея состоит в том, что вы сможете определить несколько функций, а затем указать, какой пользовательский ввод будет вызывать каждую функцию:

```
from menu import menu

def func_a ():
    return "A"

def func_b ():
    return "B"

return_value = menu (a=func_a, b=func_b)
print (f'Result is {return_value}')
```

В этом примере `return_value` будет содержать `A`, если пользователь выбирает `a`, или `B`, если пользователь выбирает `b`. Если пользователь вводит любую другую строку, ему предлагается повторить попытку. А затем мы напечатаем выбор пользователя, просто для подтверждения.

Обсуждение

Представленное здесь решение является еще одним примером таблицы диспетчеризации, которую мы видели ранее в книге в упражнении «Калькулятор префиксов». На этот раз мы исполь–

зуем параметр `**kwargs` для динамического создания таблицы диспетчеризации, а не с помощью «хардкодного» словаря (прим. пер. хардкод — написание значения непосредственно в коде, вместо того чтобы передавать его в качестве параметра).

В этом случае тот, кто вызывает функцию `menu`, предоставит ключевые слова, которые функционируют как опции меню, и функции, которые будут вызываться. Обратите внимание, что все эти функции не принимают аргументов, хотя вы можете представить сценарий, в котором пользователь мог бы предоставить больше входных данных.

Здесь мы используем `**`, который уже видели в упражнении по созданию XML. Вместо этого мы могли бы получить словарь в качестве одного аргумента, но нам кажется, что более простым способом создания словаря будет использование встроенного API Python для преобразования `**kwargs` в словарь.

Хотя я не просил вас об этом, мое решение предоставляет пользователю список допустимых пунктов меню. Я вызываю `str.join` для словаря, в следствие чего создаются строки из ключей с символами «/» между ними. Я также решил использовать `sorted`, чтобы представить их в алфавитном порядке.

Теперь мы можем запрашивать у пользователя ввод из любой функции с нулевым аргументом.

Почему мы проверяем `__name__`?

Одна из самых известных строк во всем Python выглядит так:

```
if __name__ == '__main__':
```

Что делает эта строка? Как она помогает? Эта строка является результатом нескольких разных вещей, происходящих при загрузке модуля:

1. Во-первых, при загрузке модуля его код выполняется с начала файла и до конца. Вы не просто определяете вещи: любой код в файле фактически выполняется.

Это означает, что вы можете (теоретически) вызывать print или использовать циклы `for`. В этом случае мы используем `if`, чтобы заставить некоторый код выполняться условно при его загрузке.

2. Во-вторых, переменная `__name__` либо определяется как `__main__`, что означает, что в настоящий момент все работает в начальном пространстве имен, по умолчанию и верхнем уровне, предоставленном Python, либо определяется как имя текущего модуля. Таким образом, `if` здесь проверяет, был ли модуль запущен напрямую или он был импортирован другим фрагментом кода Python.

Другими словами, строка кода говорит: «Выполняйте приведенный ниже код (т. е. внутри оператора `if`), только если выполняется программа верхнего уровня. Не обращайте внимания на то, что в `if`, когда мы `импортируем` этот модуль».

Вы можете использовать этот код несколькими способами:

3. Многие модули запускают свои собственные тесты при непосредственном вызове, а не при импорте.
4. Некоторые модули можно вызывать в интерактивном режиме, предоставляя пользователю функциональные возможности и интерфейс. Данный код позволяет это сделать, не вмешиваясь ни в какие определения функций.
5. В некоторых странных случаях, таких как модуль мультипроцессинга в Windows, код позволяет различать версии программы, которые загружаются и выполняются в отдельных процессах.

Хотя теоретически строк `if __name__ == '__main__'` в коде может быть сколько угодно, обычно эта строка появляется только один раз в конце файла модуля.

Несомненно, вы, столкнетесь с этим кодом и, возможно, даже писали его в прошлом. И теперь вы знаете, как это работает!

Решение

options — это словарь, заполненный аргументами ключевого слова.

Бесконечный цикл, из которого мы вырвемся, когда пользователь введет правильный ввод.

```
def menu (**options):
    while True:
        option_string = '/'.join (sorted (options))
        choice = input (
            f'Enter an option ({option_string}): ')
        if choice in options:
            return options [choice])
        print ('Not a valid option')

def func_a ():
    return "A"

def func_b ():
    return "B"

return_value = menu (a=func_a, b=func_b)
print (f'Result is {return_value}')
```

Создает строку отсортированных параметров, разделенных «/».

Если да, то верните результат выполнения функции.

Ввел ли пользователь ключ из **options?

В противном случае отругайте пользователя и попросите егоповторить попытку.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr178].


178

Обратите внимание, что сайт Python Tutor не поддерживает модули, поэтому данное решение было помещено в один файл без использования `import`.

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr179].


179

После выполнения упражнения

Теперь, когда вы написали и использовали два разных модуля Python, давайте пойдем дальше и поэкспериментируем с некоторыми более сложными методами и проблемами:

1. Напишите свою версию `menu.py`, которую можно импортировать (как в упражнении), но которая при вызове файла как отдельной программы из командной строки тестирует функцию. Если вы не знакомы с программным обеспечением для тестирования, таким как `pytest`, вы можете просто запустить программу и проверить результат.
2. Сделайте `menu.py` пакетом Python и загрузите его в PyPI. (Я предлагаю использовать ваше имя или инициалы, а затем «меню», чтобы избежать конфликтов имен.) Смотрите сноску о разнице между модулями и пакетами и о том, как вы можете участвовать в экосистеме PyPI со своими собственными проектами с открытым исходным кодом.
3. Определите модуль `stuff` с тремя переменными — `a`, `b` и `c` — и двумя функциями — `foo` и `bar`. Определите `__all__` так, чтобы `from stuff import *` вызывал импорт `a`, `c` и `bar`, но не `b` и `foo`.

Модули vs. пакеты

Эта глава посвящена модулям — как их создавать, импортировать и использовать. Но вы могли заметить, что мы часто используем другой термин, пакеты, при обсуждении кода Python. В чем разница между модулем и пакетом?

Модуль — это файл с расширением `.py`. Мы можем загрузить модуль с помощью `import`, как мы уже видели. Но что, если ваш проект достаточно большой, и было бы разумнее иметь несколько отдельных файлов? Как вы можете распространять эти файлы вместе?

Благодаря пакету, представляющему собой каталог с одним или более модулями Python. Например, предположим, у вас есть модули `first.py`, `second.py` и `third.py` и вы хотите сохранить их вместе. Вы можете поместить их все в каталог `mypackage`. Предполагая, что этот каталог находится в `sys.path`, вы можете написать

```
from mypackage import first
```

Python перейдет в каталог `mypackage`, найдет `first.py` и импортирует его. Вы можете получить доступ ко всем его атрибутам через `first.x`, `first.y` и так далее. В качестве альтернативы можно использовать

```
import mypackage.first
```

В этом случае Python все равно загрузит модуль `first`, но он будет доступен в вашей программе через длинное имя `mypackage.first`. Затем вы можете использовать `mypackage.first.x` и `mypackage.first.y`.

В качестве альтернативы можно использовать

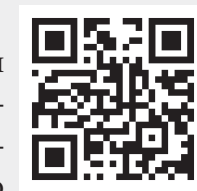
```
import mypackage
```

Но это будет полезно только в том случае, если в каталоге `mypackage` есть файл с именем `__init__.py`. В этом случае импорт `mypackage` фактически означает, что файл `__init__.py` загружен и, следовательно, выполняется. Внутри этого файла вы можете импортировать один или несколько модулей пакета.

А что, если вы хотите сделать свой пакет доступным другим? В таком случае вам следует создать пакет. Если это звучит странно, что вам нужен пакет для распространения вашего пакета, то это потому, что один и тот же термин, пакет, используется для двух разных понятий. Пакет PyPI, или дистрибутивный пакет, представляет собой оболочку пакета Python, содержащую информацию об авторе, совместимых версиях и лицензировании, а также автоматические тесты, зависимости и инструкции по установке.

Еще более запутанным, чем использование «пакета» для описания двух разных вещей, является тот факт, что и пакет

дистрибутива, и пакет Python являются каталогами и должны иметь одно и то же имя. Если ваш дистрибутивный пакет называется mypackage, у вас будет каталог с именем mypackage. Внутри этого каталога, среди прочего, будет подкаталог с именем mypackage, в который помещается пакет Python.



Если вы хотите распространять пакеты через PyPI, вам необходимо зарегистрироваться для получения имени пользователя и пароля на [qr180]. Как только вы это сделаете, следуйте следующим инструкциям, чтобы взять существующий пакет и загрузить его в PyPI с помощью Poetry, используя команды оболочки Unix:

Создание дистрибутивного пакета означает создание файла с именем `setup.py`, и я должен признать, что в течение многих лет я считал это настоящей рутиной. Оказывается, я был не одинок, и ряд разработчиков Python придумали способы относительно легкого создания дистрибутивных пакетов. Один из них, которым я пользуюсь некоторое время, называется Poetry и делает весь процесс простым и понятным.

Создает новый скелет пакета под названием mypackage.

```
$ poetry new mypackage
$ cd mypackage
$ cp –R ~/mypackage-code/* mypackage
$ poetry build
$ poetry publish
```

Переходит в каталог верхнего уровня.

Копирует содержимое пакета Python в его подкаталог.

Создает версии вашего пакета wheelfile и tar.gz в каталоге dist.

Публикует пакет в PyPI, для подтверждения вы вводите свое имя пользователя и пароль при появлении запроса.

Обратите внимание, что вы не можете загрузить конкретное имя `mypackage` в PyPI. Я рекомендую указывать

в имени пакета ваше имя пользователя или инициалы, если вы не собираетесь публиковать его для общественного использования.

Вы можете добавить множество других шагов к тем, которые я перечислил, например, вы можете (и должны) редактировать файл конфигурации `pyproject.toml`, в котором вы описываете версию вашего пакета, лицензию и зависимости. Но создание дистрибутивного пакета больше не является сложной задачей. Скорее, сложнее будет решить, каким кодом вы хотите поделиться с сообществом.

Подводя итоги

Модули и пакеты просты в написании и использовании, они помогают нам сделать наш код более коротким (согласно принципу DRY) и удобным для сопровождения. Особенно когда вы используете множество модулей и пакетов в стандартной библиотеке Python и на PyPI. Поэтому неудивительно, что многие программы Python начинаются с нескольких строк `import`. По мере того, как вы станете более свободно использовать Python, вы познакомитесь с большим число сторонних модулей, что позволит вам еще больше использовать их преимущества в своем коде.

9. Объекты

Объектно–ориентированное программирование стало основным или даже главным способом подхода к программированию. Идея проста: вместо того чтобы определять наши функции в одной части кода, а данные, над которыми работают эти функции, в отдельной части кода, мы определяем их вместе.

Или, если говорить языком, в *традиционном* процедурном программировании мы пишем список существительных (данные) и отдельный список глаголов (функции), оставляя на усмотрение программиста выяснение того, что с чем сочетается. В объектно–ориентированном программировании глаголы (функции) определяются вместе с существительными (данными), помогая нам понять, что с чем сочетается.

В мире объектно–ориентированного программирования каждое существительное — это *объект*. Мы говорим, что у каждого объекта есть тип, или класс, к которому он принадлежит. А глаголы (функции), которые мы можем вызывать на каждом объекте, называются методами.

Для примера традиционного процедурного программирования в сравнении с объектно–ориентированным программированием рассмотрим, как мы можем вычислить итоговую оценку студента, основанную на среднем значении его тестовых баллов. В процедурном программировании мы бы убедились, что оценки находятся в списке целых чисел, а затем написали бы функцию `average`, которая возвращала бы среднее арифметическое:

```
def average (numbers):
    return sum (numbers) / len (numbers)

scores = [85, 95, 98, 87, 80, 92]
print (f'The final score is {average (scores)}.')
```

Этот код работает. И работает надежно. Но вызывающая сторона отвечает за отслеживание чисел в виде списка… и за знание того, что мы должны вызвать метод `average`… и за их правильное объединение.

В объектно-ориентированном мире мы бы решили эту проблему, создав новый тип данных, который мы могли бы назвать `ScoreList`. Затем мы бы создали новый экземпляр `ScoreList`.

Даже если это одни и те же данные, `ScoreList` более явно и конкретно связан с нашей областью, чем общий список Python. Затем мы можем вызвать соответствующий метод для объекта `ScoreList`:

```
class ScoreList ():
    def __init__ (self, scores):
        self.scores = scores

    def average (self):
        return sum (self.scores) / len (self.scores)

scores = ScoreList ([85, 95, 98, 87, 80, 92])
print (f'The final score is {scores.average ()}.')
```

Как видите, нет никаких отличий от процедурного метода в том, что именно вычисляется, и даже в том, какую технику мы используем для этого. Но есть организационная и семантическая разница, которая позволяет нам мыслить по-другому.

Теперь мы мыслим на более высоком уровне абстракции и можем лучше рассуждать о нашем коде. Определение собственных типов также позволяет нам использовать сокращение при опи-

сании понятий. Подумайте о разнице между тем, чтобы сказать кому–то, что вы купили «книжную полку», и тем, чтобы описать «деревянные доски, скрепленные гвоздями и шурупами, хранящиеся вертикально и содержащие места для хранения книг». Первый вариант короче, менее двусмысленен и семантически сильнее, чем второй.

Еще одно преимущество заключается в том, что если мы решим вычислять среднее по–новому — например, некоторые учителя могут отбросить самый низкий балл, — то мы можем сохранить существующий интерфейс, изменив при этом базовую реализацию.

Итак, каковы основные причины для использования объектноориентированных методов?

1. Мы можем организовать наш код в виде отдельных объектов, каждый из которых обрабатывает различные аспекты нашего кода. Это облегчает планирование и сопровождение, а также позволяет разделить проект между несколькими людьми.
2. Мы можем создавать иерархии классов, каждый дочерний класс в иерархии наследует функциональность от своих родителей. Это сокращает объем кода и одновременно усиливает связи между схожими типами данных. Учитывая, что многие классы являются незначительными модификациями других классов, это экономит время.
3. Благодаря созданию типов данных, которые работают так же, как и встроенные типы Python, наш код ощущается как естественное расширение языка, а не как костыль. Более того, чтобы научиться использовать новый класс, требуется изучить лишь небольшую часть синтаксиса, поэтому вы можете сосредоточиться на основных идеях и функциональности.
4. Хотя Python не скрывает код и не делает его приватным, вы, скорее всего, услышите о разнице между реализацией объекта и его интерфейсом. Если я использую объект, то мне важен его интерфейс, то есть методы, которые я могу вызвать, и то, что они делают. То, как объект реализован вну–

три, не является для меня приоритетом и не влияет на мою повседневную работу. Таким образом, я могу сосредоточиться на кодировании, которое я хочу сделать, а не на внутреннем устройстве класса, который я использую, пользуясь абстракцией, созданную мной с помощью класса.

Объектно-ориентированное программирование не является панацеей — с годами мы обнаружили, что, как и все другие парадигмы, оно имеет как преимущества, так и недостатки. Например, легко создавать чудовищно большие объекты с огромным количеством методов, фактически создавая процедурную систему, замаскированную под объектно-ориентированную. Можно злоупотреблять наследованием, создавая иерархии, которые не имеют смысла. А если разбить систему на множество мелких частей, возникает проблема тестирования и интеграции этих частей с таким количеством возможных линий связи.

Тем не менее, объектная парадигма помогла многим программистам модифицировать свой код, сосредоточиться на конкретных аспектах программы, над которой они работают, и обмениваться данными с объектами, написанными другими людьми.

В Python мы любим говорить, что «все является объектом». По своей сути это означает, что язык является единообразным: типы (такие как `str` и `dict`), которые поставляются с языком, определены как классы с методами. Наши объекты работают так же, как и встроенные объекты, что сокращает кривую обучения как для тех, кто внедряет новые классы, так и для тех, кто их использует.

Подумайте, что, когда вы изучаете иностранный язык, вы обнаруживаете, что для существительных и глаголов есть всевозможные правила. Но затем вы неизбежно сталкиваетесь с несоответствиями и исключениями из этих правил. Благодаря единому набору правил для всех объектов Python устраняет эти трудности для тех, кто не является носителем языка, предоставляя нам, за неимением лучшего термина, эсперанто языков программирования. Выучив правило, вы можете применять его во всем языке.

ПРИМЕЧАНИЕ Одной из отличительных черт Python является его последовательность. Как только вы выучили правило, оно применяется ко всему языку без исключений. Если вы понимаете поиск переменных (LEGB, описанный в главе 6) и поиск атрибутов (ICPO, описанный далее в этой главе), вы будете знать правила, которые Python применяет постоянно, ко всем объектам без исключения — как к тем, которые вы создаете, так и к тем, которые встроены в язык.

В то же время Python не заставляет вас писать все в объектно-ориентированном стиле. Действительно, в программах на Python принято комбинировать парадигмы, используя смесь процедурного, функционального и объектно-ориентированного стилей. Какой стиль вы выберете и где, зависит только от вас. Но в конце концов, даже если вы пишете не в объектно-ориентированном стиле, вы все равно используете объекты Python.

Если вы собираетесь писать на Python, вы должны понимать объектную систему Python — как создаются объекты, как определяются классы и взаимодействуют с родительскими и как мы можем влиять на взаимодействие классов с остальным миром. Даже если вы пишете в процедурном стиле, вы все равно будете использовать классы, определенные другими людьми, и благодаря знанию того, как эти классы работают, ваше код станет более простым и понятным.

Эта глава содержит упражнения, направленные на то, чтобы помочь вам почувствовать себя более комфортно в работе с объектами Python. В ходе выполнения этих упражнений вы будете создавать классы и методы, создавать атрибуты на уровне объектов и классов, а также работать с такими понятиями, как композиция и наследование. По окончании вы будете готовы создавать объекты Python и работать с ними, а значит, писать и сопровождать код Python.

ПРИМЕЧАНИЕ Предыдущая глава, посвященная модулям, была короткой и простой. Эта глава, наоборот, длинная и содержит много важных идей, на усвоение которых может

потребоваться некоторое время. Эта глава потребует времени, но она стоит затраченных усилий. Понимание объектно–ориентированного программирования поможет вам не только в написании собственных классов, оно также поможет вам понять, как устроен сам Python и как работают встроенные типы.

Таблица 9.1. Что вам нужно знать

Понятие	Что это?	Пример	Чтобы узнать подробнее
`class`	Ключевое слово для создания классов Python.	`class Foo`	
`__init__`	Метод, вызываемый автоматически при создании нового экземпляра.	`def __init__ (self):`	
`__repr__`	Метод, возвращающий строку, содержащую печатное представление объекта.	`def __repr__ (self):`	
`super` built — in	Возвращает прокси–объект, на котором могут быть вызваны методы; обычно используется для вызова метода на родительском классе.	`super ().__ init__ ()`	
`dataclasses .dataclass`	Декоратор, упрощающий определение классов.	`@dataclass`	

Упражнение 38. Ложка для мороженого

Если вы собираетесь программировать с использованием объектов, то вы будете создавать классы —много классов. Каждый класс должен представлять один тип объекта и его поведение. Вы можете рассматривать класс как фабрику для создания объектов данного типа — так, класс `Car` будет создавать автомобили, также известные как «объекты автомобиля» или «экземпляры `Car`». Ваш побитый седан будет объектом автомобиля, как и новый шикарный внедорожник.

В этом упражнении вы определите класс `Scoop`, который представляет одну ложку с мороженым. Каждый шарик должен иметь единственный атрибут, flavor (вкус), строку, которую вы можете инициализировать при создании экземпляра `Scoop`.

После создания класса напишите функцию (`create_scoops`), которая создает три экземпляра класса `Scoop`, каждый из которых имеет свой `flavor` (рисунок 9.1). Поместите эти три экземпляра в список под названием `scoops` (рисунок 9.2). Наконец, проитерируйте список `scoops`, печатая `flavor` каждого шарика мороженого, который вы создали.

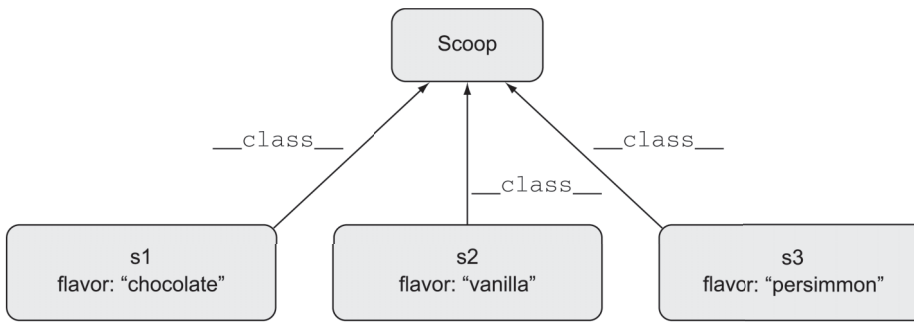


Рисунок 9.1. Три экземпляра `Scoop`, каждый из которых ссылается на свой класс.

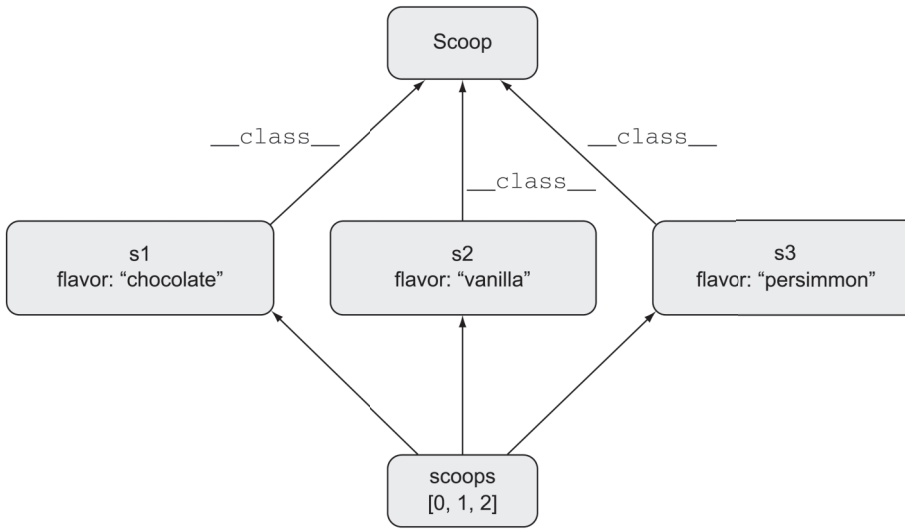


Рисунок 9.2. Наши три экземпляра `Scoop` в списке.

Обсуждение

Ключом к пониманию объектов в Python — и большей части языка Python — являются атрибуты. Каждый объект имеет тип и один или несколько атрибутов. Python сам определяет некоторые из этих атрибутов: вы можете определить их по символу `__` (в мире Python часто известному как дандер) в начале и конце имен атрибутов, например `__name__` или `__init__`.

Когда мы определяем новый класс, мы делаем это с помощью ключевого слова class. Затем мы называем класс (в данном случае `Scoop`) и указываем в круглых скобках класс или классы, от которых наследуется наш новый класс.

Наш метод `__init__` вызывается после создания нового экземпляра Scoop, но до того, как он будет возвращен тому, кто вызвал `Scoop ('flavor')`. Новый объект передается в `__init__` в `self` (т.е. первым параметром) вместе с аргументами, которые были переданы в `Scoop ()`. Таким образом, мы присваиваем `self.flavor = flavor`, создавая для нового экземпляра атрибут flavor со значением параметра flavor.

Поговорим о self

Первый параметр в каждом методе традиционно называется `self`. Однако `self` не является зарезервированным словом в Python: использование этого слова является условностью и пришло из языка Smalltalk, объектная система которого повлияла на структуру Python.

Во многих языках текущий объект известен как `this`. Более того, в таких языках `this` — это не параметр, а скорее специальное слово, обозначающее текущий объект. В Python нет такого специального слова: экземпляр, для которого был вызван метод, всегда будет известен как `self`, а `self` всегда будет первым параметром в каждом методе. Теоретически вы можете использовать любое имя для первого параметра, включая `this`. (Но, правда, какой уважающий себя язык стал бы так поступать?) Хотя ваша программа все равно будет работать, предполагается, что первый параметр, представляющий экземпляр, будет называться `self`, так что и вам следует поступать так же.

Как и в случае с обычными функциями Python, здесь нет принудительного применения типов. Предполагается, что `flavor` будет содержать значение `str`, поскольку документация указывает, что ожидается именно это.

ПРИМЕЧАНИЕ Если вы хотите более строгого контроля, то рассмотрите возможность использования аннотаций типов Python и Муру или аналогичного инструмента проверки типов. Более подробную информацию о Муру вы можете найти на сайте [qr186]. Также вы можете найти отличное введение в аннотации типов Python и их использование здесь [qr187].

Чтобы создать три `scoops`, я использую генератор списка, итерируя по `flavors` и создавая новые экземпляры `Scoop`. В результате получается список с тремя объектами `Scoop`, каждый из которых имеет свой `flavor`:

```
scoops = [Scoop (flavor)
for flavor in ('chocolate', 'vanilla', 'persimmon')]
```

Если вы привыкли работать с объектами на другом языке программирования, вам может быть интересно, где находятся методы `getter` и `setter` для получения и установки значения атрибута `flavor`. В Python, поскольку все является публичным, нет реальной необходимости в геттерах и сеттерах. И действительно, если у вас нет действительно веских причин для этого, вам, вероятно, следует избегать их использования.

ПРИМЕЧАНИЕ Если вам понадобится геттер или сеттер, вы можете рассмотреть Python property, которое скрывает вызов метода за API изменения или получения атрибута. Вы можете узнать больше о свойствах здесь: [qr188].

Я должен отметить, что даже наш простой класс `Scoop` демонстрирует несколько вещей, которые присущи почти всем классам Python. У нас есть метод `__init__`, параметры которого позволяют нам устанавливать атрибуты для вновь созданных экземпляров. Он хранит состояние внутри `self`, и таким образом можно хранить любой тип объектов Python — не только строки или числа, но и списки и словари, а также другие типы объектов.

ПРИМЕЧАНИЕ Не советую готовить мороженое с хурмой. Ваша семья никогда не позволит вам забыть об этом.

Решение

```
class Scoop ():
    def __init__ (self, flavor):
        self.flavor = flavor

def create_scoops ():
    scoops = [Scoop ('chocolate'),
              Scoop ('vanilla'),
              Scoop ('persimmon')]
for scoop in scoops:
        print (scoop.flavor)

create_scoops ()
```

Первым параметром каждого метода всегда будет self, представляющий текущий экземпляр.

Устанавливаем атрибут flavor как значение в параметре. flavor.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr189].


189

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr190].


190

После выполнения упражнения

Если вы программируете на Python, то, скорее всего, вам придется регулярно писать классы. И если вы это делаете, вы будете писать много методов `__init__`, которые добавляют атрибуты к объектам различного рода. Вот несколько дополнительных простых классов, которые вы можете написать, чтобы попрактиковаться в этом:

1. Напишите класс `Beverage`, экземплярами которого будут являться напитки. Каждый напиток должен иметь два атрибута: название (описывающее напиток) и температуру.

   Создайте несколько напитков и убедитесь, что их названия и температуры обрабатываются правильно.

2. Измените класс `Beverage`, чтобы можно было создать новый экземпляр, указав имя, а не температуру. Если вы это сделаете, то температура должна иметь значение по умолчанию 75 градусов Цельсия. Создайте несколько напитков и дважды проверьте, что температура имеет это значение по умолчанию, в случае если она не указана.
3. Создайте новый класс `LogFile`, который должен быть инициализирован именем файла. Внутри `__init__` откройте файл для записи и назначьте его атрибуту, `file`, который находится в экземпляре. Убедитесь, что запись в файл возможна через атрибут `file`.

Что делает __init__?

Простой класс в Python выглядит так:

```
class Foo ():
    def __init__ (self, x):
        self.x = x
```

И действительно, с классом `Foo` мы можем сказать, что

```
f = Foo (10) print (f.x)
```

Это заставляет многих людей, особенно тех, кто пришел из других языков, вызывать `__init__` конструктор, то есть метод, который фактически создает новый экземпляр `Foo`. Но это не совсем так.

Когда мы вызываем `Foo (10)`, Python сначала ищет идентификатор `Foo` так же, как он ищет любую другую переменную в языке, следуя правилу LEGB. Он находит `Foo` как глобально определенную переменную, ссылающуюся на класс. Классы можно вызывать, то есть их можно вызывать с помощью круглых скобок. И поэтому, когда мы

просим вызвать его и передать 10 в качестве аргумента, Python соглашается.

Но что на самом деле выполняется? Разумеется, метод конструктора, известный как `__new__`. Теперь вы почти никогда не должны реализовывать `__new__` самостоятельно: в некоторых случаях это может быть полезно, но в подавляющем большинстве случаев вы не хотите трогать или переопределять его. Это потому, что `__new__` создает новый объект, а мы не хотим иметь с этим дело.

Метод `__new__` также возвращает вновь созданный экземпляр Foo вызывающей стороне. Но перед этим он делает еще одну вещь: ищет, а затем вызывает метод `__init__`. Это означает, что `__init__` вызывается после создания объекта, но до его возврата.

А что делает `__init__`? Проще говоря, он добавляет к объекту новые атрибуты.

В то время как другие языки программирования говорят о «переменных экземпляра» и «переменных класса», у разработчиков Python есть только один инструмент, а именно атрибут. Когда в коде встречается `a.b`, можно сказать, что `b` является атрибутом `a`, что означает (в той или иной степени), что `b` относится к объекту, связанному с `a`. Атрибуты объекта можно рассматривать как его собственный приватный словарь.

Таким образом, задача `__init__` состоит в том, чтобы добавить один или несколько атрибутов нашему новому экземпляру. В отличие от таких языков, как C# и Java, в Python мы не просто объявляем атрибуты: мы должны фактически создавать и присваивать их во время выполнения, когда создается новый экземпляр.

Во всех методах Python параметр `self` относится к экземпляру. Любые атрибуты, которые мы добавляем к `self`, останутся после возвращения метода. Поэтому предпочтительнее присвоить кучу атрибутов `self` в `__init__`.

Давайте шаг за шагом разберем как это работает. Во–первых, давайте определим простой класс `Person`, который присваивает имя объекту:

```
class Person:
   def __init__ (self, name):
        self.name = name
```

Затем создадим новый экземпляр `Person`:

```
p = Person ('myname')
```

Что происходит внутри Python? Во–первых, метод `__new__`, который мы никогда не определяем, работает «за кулисами», создавая объект, как показано на рис. 9.3.

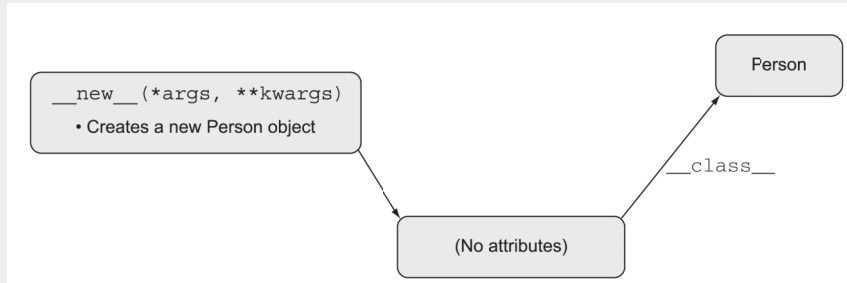


Рисунок 9.3. При создании объекта происходит вызов `__new__`.

Он создает новый экземпляр `Person` и сохраняет его в качестве локальной переменной. Но затем `__new__` вызывает `__init__`. Он передает только что созданный объект в качестве первого аргумента `__init__`, а затем передает все дополнительные аргументы, используя `*args` и `**kwargs`, как показано на рисунке 9.4.

Теперь `__init__` добавляет один или несколько атрибутов к новому объекту, как показано на рисунке 9.5, известному как `self`, локальной переменной.

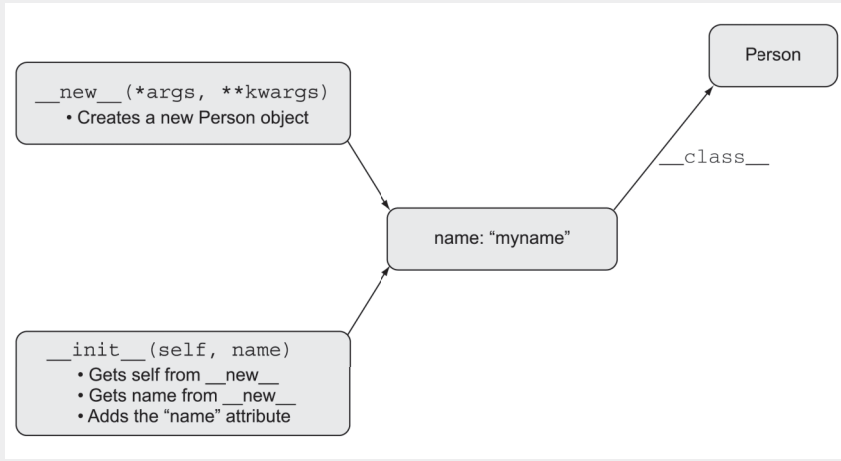


Рисунок 9.4. `__new__` вызывает `__init__`.

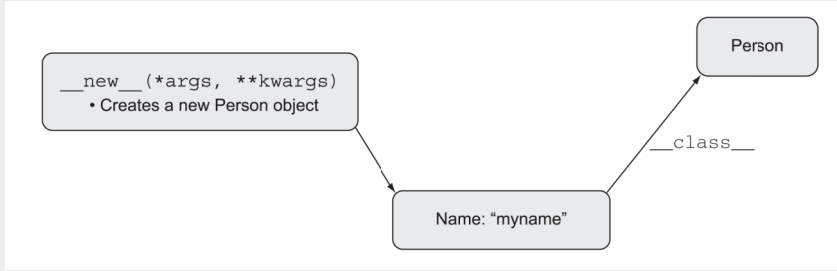


Рисунок 9.5. `__init__` добавляет атрибуты к объекту.

Наконец, `__new__` возвращает вновь созданный объект вызывающей стороне с добавленным атрибутом, как показано на рисунке 9.6.

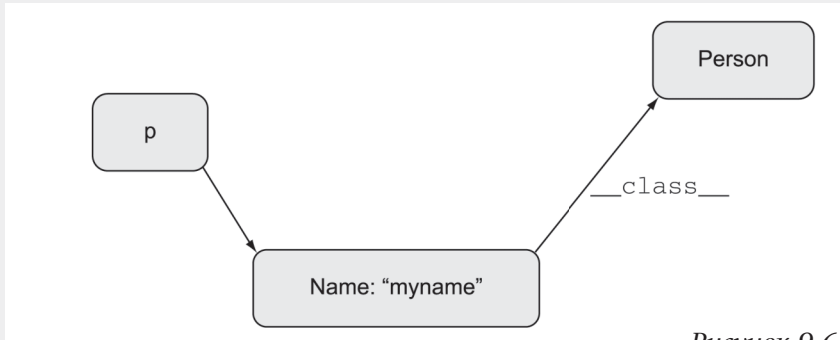


Рисунок 9.6.

Теперь можем ли мы добавить новые атрибуты к нашему экземпляру после выполнения `__init__`? Да, безусловно — никаких технических препятствий для этого нет. Но, как правило, вы хотите определить все свои атрибуты в `__init__`, чтобы ваш код был максимально читабельным и очевидным. Вы можете изменять значения позже, в других методах, но первоначальное определение должно находиться в `__init__`.

Обратите внимание, что `__init__` не использует ключевое слово `return`. Это потому, что его возвращаемое значение игнорируется и не имеет значения. Смысл `__init__` заключается в изменении нового экземпляра путем добавления атрибутов, а не в получении возвращаемого значения. После того как функция `__init__` завершила свою работу, `__new__` получает обновленный и измененный объект. После чего `__new__` возвращает этот новый объект вызывающему коду.

Упражнение 39. Чашка для мороженого

Каждый раз, когда я преподаю объектно-ориентированное программирование, я сталкиваюсь с людьми, которые изучали его раньше и убеждены, что самое важное в нем — это наследование. Наследование, конечно, важно, и мы рассмотрим его в ближайшее время, но более важной методикой является *композиция*, когда один объект содержит другой объект.

Называть это методикой в Python — немного перебор, поскольку все является объектом, и мы можем присваивать объектам атрибуты. Итак, наличие одного объекта, принадлежащего другому объекту, необходимо только для того, чтобы… связать объекты вместе.

Тем не менее, композиция также является важной методикой, потому что она позволяет нам создавать более крупные

объекты из более мелких. Я могу создать автомобиль из мотора, колес, шин, коробки передач, сидений и так далее. Я могу создать дом из стен, полов, дверей и тому подобное. Разделение проекта на более мелкие части, определение классов, описывающих эти части, а затем их объединение для создания более крупных объектов — вот как работает объектно-ориентированное программирование.

В этом упражнении мы увидим его уменьшенную версию. В предыдущем упражнении мы создали класс `Scoop`, представляющий собой один шарик мороженого. Однако, если мы действительно собираемся моделировать реальный мир, у нас должен быть еще один объект, в который мы можем поместить `scoops`. Поэтому я хочу, чтобы вы создали класс `Bowl`, представляющий миску, в которую мы можем положить наше мороженое (рис. 9.7), например:

```
s1 = Scoop ('chocolate')
s2 = Scoop ('vanilla')
s3 = Scoop ('persimmon')

b = Bowl ()
b.add_scoops (s1, s2)
b.add_scoops (s3)
print (b)
```

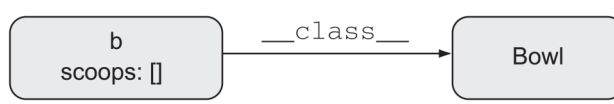


Рисунок 9.7. Новый экземпляр `Bowl` с пустым списком `scoops`.

Результатом выполнения `print (b)` должно стать отображение трех вкусов мороженого в нашей чаше (рисунок 9.8). Обратите внимание, что в миску можно добавить любое количество шариков с помощью функции `Bowl.add_scoops`.

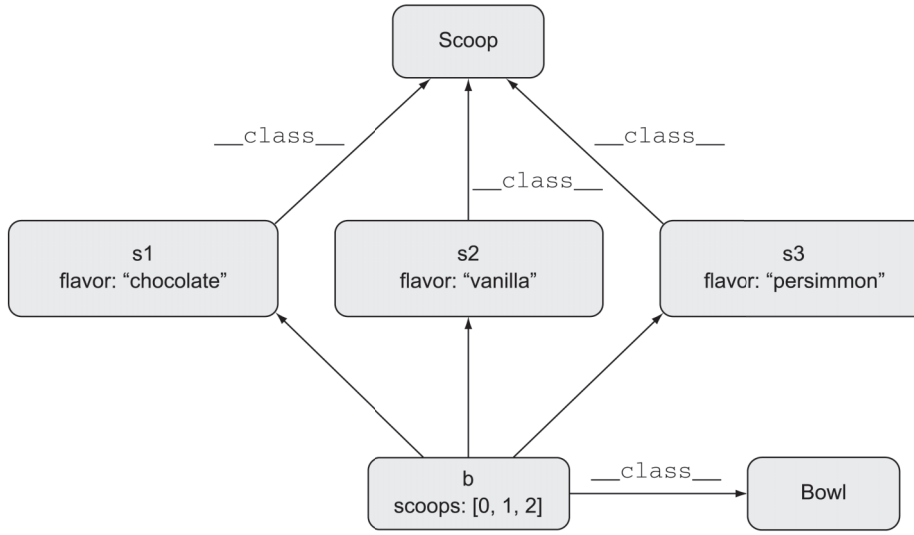


Рисунок 9.8. Три объекта `Scoop` в нашей миске.

Обсуждение

Решение не требует изменений в классе `Scoop`. Скорее, мы создаем нашу чашу таким образом, чтобы она могла содержать любое количество экземпляров `Scoop`.

Прежде всего мы определяем атрибут `self.scoops` нашего объекта как список. Теоретически мы могли бы использовать словарь или множество, но, учитывая, что нет никаких очевидных кандидатов на роль ключей и что мы, возможно, захотим сохранить порядок `scoops`, я считаю, что список — более логичный выбор.

Помните, что мы храним экземпляры `Scoop` в `self.scoops`. Мы не просто храним строку с описанием вкусов. У каждого экземпляра `Scoop` будет свой атрибут `flavor` — строка, содержащая вкус мороженого.

Мы создаем атрибут `self.scoops` в виде пустого списка в `__init__`.

Затем нам нужно определить `add_scoops`, который может принимать любое количество аргументов — которые, как мы предполагаем, являются экземплярами `Scoop` — и добавлять их

в миску. Это означает, почти по определению, что мы должны использовать оператор `splat (*)` при определении параметра `*new_scoops`.

`В результате` new_scoops будет кортежем, содержащим все аргументы, которые были переданы `add_scoop`s.

> **ПРИМЕЧАНИЕ** Существует большая разница между переменной `new_scoops` и атрибутом `self.scoops`. Первая является локальной переменной в функции и ссылается на кортеж объектов `Scoop`, которые пользователь передал в `add_scoops`. Вторая — это атрибут, присоединенный к локальной переменной `self`, который ссылается на экземпляр объекта, над которым мы сейчас работаем.

Затем мы можем перебирать каждый элемент `scoops`, добавляя его к атрибуту `self.scoops`. Мы делаем это в цикле `for`, вызывая `list.append` для каждого `scoop`.

Наконец, для печати `scoops` мы просто вызываем print `(b)`. Это приводит к вызову метода `__repr__` нашего объекта, предполагая, что он определен. Наш метод `__repr__` делает немного больше, чем вызывает `str.join` для строк, которые мы извлекаем из `flavors.repr` и `str`.

repr vs. str

Вы можете определить `__repr__` или `__str__` (или оба) для своих объектов. В теории `__repr__` создает строки, предназначенные для разработчиков и соответствующие синтаксису Python. В противоположность этому, `__str__` — это то, как ваш объект должен выглядеть для конечных пользователей.

На практике я обычно определяю `__repr__` и игнорирую `__str__`. Это потому, что `__repr__` охватывает оба случая, что просто прекрасно тогда, если я хочу, чтобы все представления строк были эквивалентны. Если я захочу

указать различие между выводом строк для разработчиков и для конечных пользователей, я всегда смогу добавить `__str__` позже.

В этой книге я буду использовать исключительно `__repr__`. Но если вы хотите использовать `__str__`, то это прекрасно, и, кроме того, это будет более официально правильный вариант.

Обратите внимание, однако, что мы не вызываем str.join для генератора списка, потому что здесь нет квадратных скобок. Скорее, мы вызываем его в выражении–генераторе, которое вы можете рассматривать как ленивую версию генератора списка. Правда, в таком случае выигрыша в производительности действительно нет. Я использовал его, чтобы продемонстрировать, что почти везде, где вы можете использовать генератор списка, вместо него можно использовать выражение–генератор.

is-a vs. has-a

Если у вас есть опыт объектно–ориентированного программирования, у вас может возникнуть соблазн написать здесь, что `Scoop` наследуется от `Bowl` или что `Bowl` наследуется от `Scoop`. Ни то, ни другое неверно, потому что наследование (которое мы рассмотрим позже в этой главе) описывает отношение, известное в компьютерных науках как «is–a». Мы можем сказать, что работник — это человек или что машина — это транспортное средство, что указывало бы на такое отношение.

В реальной жизни мы можем сказать, что миска содержит одну или несколько мерных ложек. В терминах программирования мы бы описали это так: Bowl has–a Scoop. Отношение «has–a» описывает не наследование, а композицию.

Я обнаружил, что относительные новички в объектно–ориентированном программировании часто убеждены, что

если задействованы два класса, то один из них, вероятно, должен наследоваться от другого. Указание на правило «is–a» для наследования по сравнению с правилом «has–a» для композиции помогает прояснить два разных отношения и то, когда уместно использовать наследование или композицию.

Решение

```
class Scoop ():
    def __init__ (self, flavor):
        self.flavor = flavor

class Bowl ():
    def __init__ (self):
        self.scoops = []

    def add_scoops (self, *new_scoops):
        for one_scoop in new_scoops:
            self.scoops.append (one_scoop)
def __repr__ (self):
    return '\n'.join (s.flavor for s in self.scoops)

s1 = Scoop ('chocolate')
s2 = Scoop ('vanilla')
s3 = Scoop ('persimmon')

b = Bowl ()
b.add_scoops (s1, s2)
b.add_scoops (s3)
print (b)
```

Инициализация self.scoops пустым списком.

***new_scoops — это то же самое, что и *args. Вы можете использовать любое название, какое захотите.**

Создает строку с помощью str.join и выражения-генератора.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr191].

191

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr192].

После выполнения упражнения

Теперь вы увидели, как создать явное отношение «has–a» между двумя классами. Вот еще несколько упражнений для знакомства с этим типом отношений:

1. Создайте класс `Book`, который позволит вам создавать книги с названием, автором и ценой. Затем создайте класс `Shelf`, на котором вы можете разместить одну или несколько книг с помощью метода `add_book`. Наконец, добавьте к классу `Shelf` метод `total_price`, который будет суммировать цены книг на полке.
2. Напишите метод `Shelf.has_book`, который принимает единственный строковый аргумент и возвращает `True` или `False` в зависимости от того, существует ли на полке книга с указанным названием.
3. Измените класс `Book` так, чтобы он добавил еще один атрибут, `width`. Затем добавьте атрибут `width` к каждому экземпляру `Shelf`. Когда `add_book` попытается добавить книги, чья суммарная ширина будет слишком велика для полки, вызовите исключение.

Сокращение избыточности с помощью `dataclass`

Чувствуете ли вы, что определения ваших классов повторяются? Если да, то вы не одиноки. Одна из самых распространенных жалоб, которые я слышу от людей по поводу классов Python, заключается в том, что метод `__init__` в основном делает одно и то же в каждом классе: принимает аргументы и присваивает их атрибутам `self`.

Начиная с Python 3.7, вы можете вырезать часть шаблонного кода для создания классов с помощью декоратора `dataclass`, сосредоточившись на коде, который вы действительно хотите написать. Например, вот как должен быть определен класс `Scoop`:

```
@dataclass class
Scoop ():
   Flavor: str
```

Обратите внимание: здесь нет метода `__init__`! Он здесь не нужен — используемый декоратор `@dataclass` пишет его за вас. Он также заботится о других вещах, таких как сравнения и улучшенная версия `__repr__`. В общем, вся суть классов данных заключается в том, чтобы уменьшить вашу рабочую нагрузку.

Обратите внимание, что мы использовали аннотацию типа (`str`), чтобы указать, что наш атрибут `flavor` должен принимать только строки. Аннотации типов обычно необязательны в Python, но если вы объявляете атрибуты в классе данных, то они обязательны. Python, как обычно, игнорирует эти аннотации типов: как упоминалось ранее в этой главе, проверка типов выполняется внешними программами, такими как Муру.

Также обратите внимание, что мы определяем `flavor` на уровне класса, даже если мы хотим, чтобы он был атрибутом наших экземпляров. Вы почти наверняка не захотите иметь один и тот же атрибут у экземпляров и классов, и это нормально: декоратор `dataclass` увидит атрибут вместе с его аннотацией типа и обработает все соответствующим образом. Как насчет нашего класса `Bowl`? Как мы можем определить его с помощью класса данных? Оказывается, нам нужно предоставить немного больше информации:

```
from typing import List
from dataclasses import dataclass, field

@dataclass class
Bowl ():
    scoops: List [Scoop] = field (default_factory=list)

    def add_scoops (self, *new_scoops):
          for one_scoop in new_scoops:
                self.scoops.append (one_scoop)

    def __repr__ (self):
          return '\n'.join (s.flavor for s in self.scoops)
```

Давайте проигнорируем методы `add_scoops` и `__repr__` и сосредоточимся на начале нашего класса. Во–первых, мы снова используем декоратор `@dataclass`. Но затем, когда мы определяем наш атрибут `scoops`, мы задаем не просто тип, а значение по умолчанию.

Обратите внимание, что тип, который мы предоставляем, `List [int]`, имеет заглавную букву L. Это означает, что он отличается от встроенного типа списка. Он берется из модуля `typing`, который поставляется вместе с Python и предоставляет нам объекты, предназначенные для использования в аннотациях типов. Тип `List`, когда он используется сам по себе, представляет собой список любого типа. Но в сочетании с квадратными скобками мы можем указать, что все элементы списка scoop будут объектами типа `Scoop`.

Обычно значения по умолчанию могут быть просто присвоены их атрибутам. Но поскольку `scoops` — это список, а значит, изменяем, наша задача немного сложнее. Когда мы создаем новый экземпляр `Bowl`, мы не хотим получить ссылку на существующий объект. Скорее, мы хотим вызвать

`list`, возвращая новый экземпляр `list` и присваивая его scoops. Для этого нам нужно использовать `default_factory`, который указывает dataclass, что он не должен повторно использовать существующие объекты, а должен создавать новые.

В этой книге используется классический, стандартный способ определения классов Python — отчасти для помощи людям, все еще использующим Python 3.6, и отчасти для того, чтобы вы могли понять, что происходит под капотом. Но я не удивлюсь, если `dataclass` со временем станет стандартным способом создания классов Python, и если вы захотите использовать его в своих решениях, то не стесняйтесь.

Как Python ищет атрибуты

В главе 6 я рассказал о том, как Python ищет переменные с помощью LEGB — сначала в локальной области видимости, затем в области внешней функции, затем в глобальной и, наконец, во встроенном пространстве имен. Python последовательно придерживается этого правила, и, зная его, становится легче изучать язык.

Python аналогичным образом ищет атрибуты по стандартному, четко определенному пути. Но этот путь сильно отличается от правила LEGB для переменных. Я называю это ICPO, что означает «instance (экземпляр), class (класс), parents (родители) и object (объект)». Я объясню, как это работает.

Когда вы запрашиваете у Python `a.b`, он сначала спрашивает у объекта `a`, есть ли у него атрибут с именем `b`. Если это так, то возвращается значение, связанное с a.b, и это конец процесса. Это I в ICPO — мы сначала проверяем экземпляр.

Но если у `a` нет атрибута `b`, то на этом Python не остановится. Он проверяет класс `a`, каким бы он ни был. То есть, если `a.b` не существует, мы ищем `type (a).b`. Если он

существует, то мы получаем значение обратно, и поиск заканчивается. Это и есть C в ICPO.

Этот механизм сразу же объясняет, почему и как методы определены в классах, и при этом могут быть вызваны через экземпляр. Рассмотрим следующий код:

```
s = 'abcd' print (s.upper ())
```

Здесь мы определяем `s` как строку. Затем мы вызываем `s.upper`. Python спрашивает `s`, есть ли у нее атрибут `upper`, и получает ответ «нет». Затем он спрашивает, есть ли у `str` атрибут `upper`, и получает «да». Объект метода извлекается из `str` и затем вызывается. В то же время мы можем говорить о методе как о `str.upper`, потому что он действительно определен в str и находится там.

Что происходит, если Python не может найти атрибут экземпляра или класса? Он начинает проверять родителей класса. До сих пор мы не сталкивались с этим; все наши классы автоматически и неявно были унаследованы от объекта. Но класс может наследоваться от любого другого класса — и часто это хорошая идея, поскольку подкласс может использовать функциональные возможности родительского класса.

Вот пример:

```
class Foo ():
   def __init__ (self, x):
        self.x = x
   def x2 (self):
        return self.x * 2
class Bar (Foo):
   def x3 (self):
   return self.x * 3
```

```
b = Bar (10)
print (b.x2 ())   ←——  Печатает 20
print (b.x3 ())   ←——  Печатает 30
```

В этом коде мы создаем экземпляр Bar — класса, наследуемого от Foo (рис. 9.9).

Когда мы создаем экземпляр Bar, Python ищет init. Где? Сначала в экземпляре, но его там нет. Потом в классе (Bar), но его там тоже нет. Затем он смотрит на родителя Bar — Foo, и находит там init. Этот метод запускается, устанавливая атрибут x, а затем возвращает, передавая нам b, экземпляр Bar со значением x, равным 10 (рис. 9.10).

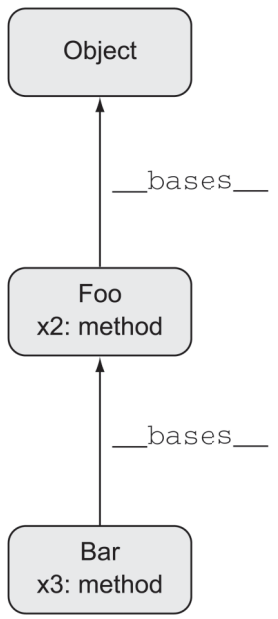


Рис. 9.9. `Bar` наследуется от `Foo`, который наследуется от объекта.

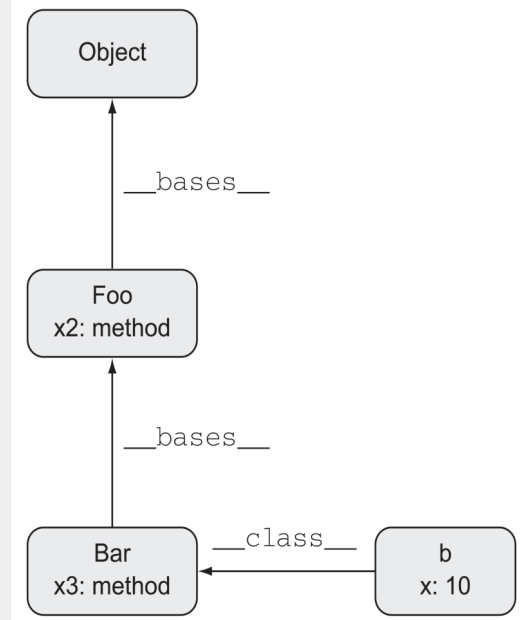


Рис. 9.10 b — экземпляр `Bar`.

То же самое происходит, когда мы вызываем `x2`. Мы ищем в `b` и не можем найти этот метод. Затем мы ищем в `type (b)`, или в `Bar`, и не можем найти этот метод. Но когда мы проверяем родителя `Bar`, `Foo`, мы находим его, и метод срабатывает. Если бы мы определили для `Bar` собственный метод с именем `x2`, то он бы выполнился вместо `Foo.x2`. Наконец, мы вызываем `x3`. Мы проверяем `b` и не находим его. Мы проверяем `Bar` и находим его, и метод, таким образом, срабатывает.

Что, если во время нашего поиска ICPO атрибут не существует в экземпляре, классе или родителе? Тогда мы обратимся к главному родителю во всем Python — `object`. Вы можете создать экземпляр `object`, но смысла в этом нет — он существует исключительно для того, чтобы другие классы могли наследоваться от него и таким образом получить доступ к его методам.

В результате, если вы не определите метод `__init__`, то будет запущен `object.__init__`. А если вы не определите `__repr__`, то будет запущен `object.__repr__` и так далее.

Последнее, что нужно запомнить при использовании пути поиска ICPO — это то, что побеждает первое совпадение. Это означает, что, если два атрибута в пути поиска имеют одинаковые имена, Python никогда не обнаружит второй атрибут. Как правило, это даже к лучшему, так как позволяет нам переопределять методы в подклассах. Но если вы не ожидаете, что это произойдет, то вы можете в конечном итоге быть удивлены.

Упражнение 40. Ограничения для чаши

В Python мы можем добавить атрибут практически к любому объекту. При написании классов типично и традиционно определять атрибуты данных для экземпляров и атрибуты методов для классов. Но нет причин, по которым мы не можем определять атрибуты данных и для классов.

В этом упражнении я хочу, чтобы вы определили атрибут класса, который будет функционировать как константа, гарантируя, что нам не придется «хардкодить» какие–либо значения в нашем классе.

В чем заключается задача здесь? Вы, наверное, заметили недостаток нашего класса `Bowl`, который, несомненно, нравится детям и ненавистен их родителям: вы можете положить в миску столько объектов `Scoop`, сколько захотите.

Давайте заставим детей грустить, а их родителей радоваться, ограничив количество черпаков в миске тремя. То есть вы можете добавлять столько ложек для каждого вызова `Bowl.add_scoops`, сколько захотите, и можете вызывать этот метод столько раз, сколько хотите, но только первые три ложечки будут действительно добавлены. Все остальные ложки будут проигнорированы.

Обсуждение

Для того чтобы это сработало, нам нужно внести всего два изменения в наш оригинальный класс `Bowl`.

Во–первых, нам нужно определить атрибут класса `Bowl`. Это проще всего сделать, задав атрибут в определении класса (рисунок 9.11). Установить `max_scoops = 3` в блоке класса — это то же самое, что написать впоследствии: `Bowl.max_scoops = 3`.

Рисунок 9.11 max_scoops находится в классе, поэтому даже пустой экземпляр имеет к нему доступ.

Но подождите, действительно ли нам нужно определять max_scoops в классе Bowl? Теоретически у нас есть два других варианта:

1. Определить максимум для экземпляра, а не для класса. Это сработает (например, добавьте `self.max_scoops = 3` в `__init__`), при этом будет означать, что у каждой миски будет разное максимальное количество ложек. Поместив атрибут на класс (рисунок 9.12), мы указываем, что все чашки будут иметь одинаковый максимум.
2. Мы могли бы также захардкодить значение 3 в нашем коде, а не использовать символическое имя, такое как max_scoops. Но это снизит нашу гибкость, особенно если мы захотим использовать наследование (как мы увидим позже). Более того, если мы решим изменить максимальное значение в дальнейшем, проще сделать это в одном месте с помощью назначения атрибута, а не в нескольких местах.

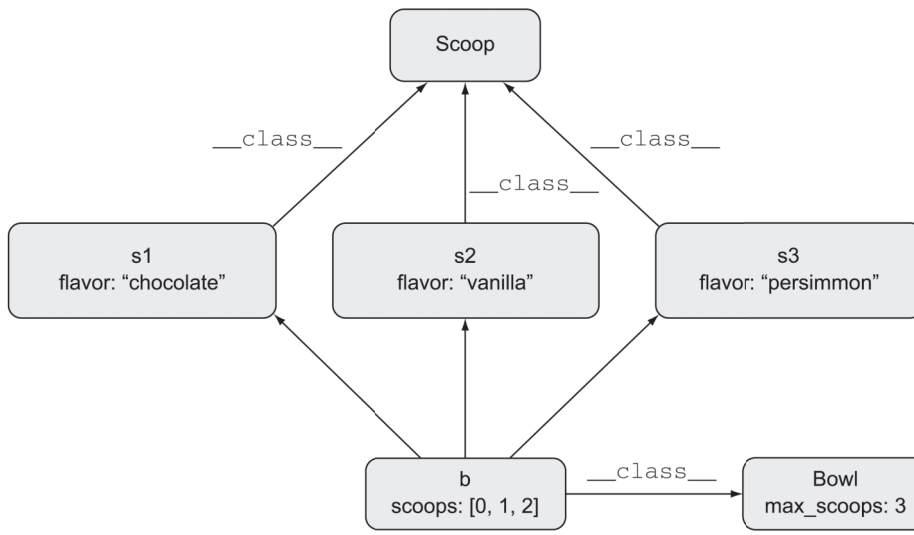


Рисунок 9.12. Экземпляр `Bowl`, содержащий scoop, с `max_scoops`, определенным в классе.

Во-вторых, нам нужно изменить `Bowl.add_scoops`, добавим оператор `if`, чтобы добавление новых scoops зависело от текущей длины `self.scoops` и значения `Bowl.max_scoops`.

Атрибуты класса — это просто статические переменные?

Если вы пришли из мира Java, C# или C++, то атрибуты класса очень похожи на статические переменные. Но они не являются статическими переменными, и вы не должны их так называть.

Вот несколько отличий атрибутов класса от статических переменных, хотя их использование может быть схожим:

Во-первых, атрибуты класса — это просто еще один пример атрибутов объекта Python. Это означает, что мы можем и должны рассуждать об атрибутах класса так же, как и обо всех остальных, используя правило поиска ICPO. Вы можете получить доступ к ним через класс (например, `ClassName.attrname`) или через экземпляр (например, `one_instance.attrname`). Первый вариант будет работать, потому что вы используете класс, а второй — потому что после проверки экземпляра Python проверяет его класс.

В решении для этого упражнения `Bowl.max_scoops` — это атрибут класса `Bowl`. Теоретически мы могли бы присвоить `max_scoops` каждому отдельному экземпляру Bowl, но логичнее сказать, что все объекты `Bowl` имеют одинаковое максимальное количество ложек.

Во-вторых, статические переменные совместно используются экземплярами и классом. Это означает, что присвоение переменной класса через экземпляр имеет тот же результат, что и присвоение ей через класс. В Python есть большая разница между присвоением переменной класса через экземпляр и присвоением через класс: в первом случае к экземпляру будет добавлен новый атрибут, что эффективно блокирует доступ к атрибуту класса.

То есть, если мы присваиваем `Bowl.max_scoops`, то мы изменяем максимальное количество ложек, которое мо-

жет быть у всех мисок. Но если мы присвоим `one_bowl.max_scoops`, мы установим новый атрибут для экземпляра `one_bowl`. Это поставит нас в ужасную ситуацию, когда для `Bowl.max_scoops` будет задано одно значение, а для `one_bowl.max_scoops` — другое. Более того, поиск `one_bowl.max_scoops` (согласно правилу ICPO) будет остановлен после нахождения атрибута в экземпляре и никогда не пойдет дальше искать в классе.

В-третьих, методы на самом деле тоже являются атрибутами класса. Но мы не думаем о них таким образом, потому что они определяются по-другому. В любом случае методы создаются с помощью `def` внутри определения класса.

Когда я вызываю `b.add_scoops`, Python ищет в `b` атрибут `add_scoops` и не находит его. Затем он ищет его в `Bowl` (т.е. в классе `b`) и находит его — и извлекает объект метода. Затем скобки вызывают метод. Это работает только в том случае, если метод действительно определен в классе, а это так и есть. Методы почти всегда определены в классе, и благодаря правилу ICPO Python будет искать их там.

Наконец, в Python нет констант, но мы можем имитировать их с помощью атрибутов класса. Как и в случае с `max_scoops`, я часто определяю атрибут класса, к которому затем можно получить доступ по имени как через класс, так и через экземпляры.

Например, атрибут класса max_scoops используется здесь как своего рода константа. Вместо того чтобы хранить закодированное число 3 везде, где мне нужно указать максимальное количество ложек, которые можно положить в миску, я могу обратиться к `Bowl.max_scoops`. Это добавляет ясности в мой код и позволяет мне в будущем изменить значение в одном месте.

Решение

```
class Scoop ():
    def __init__ (self, flavor):
        self.flavor = flavor

class Bowl ():
    max_scoops = 3

    def __init__ (self):
        self.scoops = []
    def add_scoops (self, *new_scoops):
        for one_scoop in new_scoops:
            if len (self.scoops) < Bowl.max_scoops:
                self.scoops.append (one_scoop)
    def __repr__ (self):
        return '\n'.join (s.flavor for s in self.scoops)

s1 = Scoop ('chocolate')
s2 = Scoop ('vanilla')
s3 = Scoop ('persimmon')
s4 = Scoop ('flavor 4')
s5 = Scoop ('flavor 5')

b = Bowl ()
b.add_scoops (s1, s2)
b.add_scoops (s3)
b.add_scoops (s4, s5)
print (b)
```

max_scoops не является переменной — это атрибут класса Bowl.

Использует Bowl.max_scoops для получения максимального количества для чаши, заданного в классе.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr193].

193

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr194].

После выполнения упражнения

Как я уже говорил, вы можете использовать атрибуты класса различными способами. Вот несколько дополнительных задач, которые помогут вам понять, как определять и использовать атрибуты класса:

1. Определите класс `Person` и атрибут класса `population`, значение которого увеличивается каждый раз, когда вы создаете новый экземпляр `Person`. Дважды проверьте, что после создания пяти экземпляров, названных `p1`… `p5`, `Person.population` и `p1.population` равны 5.
2. Python предоставляет метод `__del__`, который выполняется, когда объект собирается в мусор. (По моему опыту, удаление переменной или присвоение ее другому объекту довольно быстро вызовет `__del__`.) Измените класс `Person` так, чтобы при удалении экземпляра `Person` счетчик населения уменьшался на 1. Если вы плохо знакомы со сборщиком мусора или с тем, как это работает в Python, взгляните на эту статью: [qr195].

3. Определите класс `Transaction`, в котором каждый экземпляр представляет либо депозит, либо снятие средств с банковского счета. При создании нового экземпляра `Transaction` вам нужно будет указать сумму — положительную для депозита и отрицательную для снятия. Используйте атрибут класса для отслеживания текущего баланса, который должен быть равен значению всех экземпляров, созданных на сегодняшний день.

Наследование в Python

Пришло время использовать наследование — важную идею в объектно–ориентированном программировании. Основная идея отражает тот факт, что мы часто хотим создавать классы, которые очень похожи друг на друга. Таким образом, мы можем создать родительский класс, в котором мы определяем общее поведение. А затем мы можем создать один или несколько дочерних классов или подклассов, каждый из которых наследуется от родительского класса:

1. Если у меня уже есть класс `Person`, то я могу захотеть создать класс `Employee`, который идентичен Person, за исключением того, что у каждого сотрудника есть идентификационный номер, отдел и зарплата.
2. Если у меня уже есть класс `Vehicle`, то я могу создать класс `Car`, класс `Truck` и класс `Bicycle`.
3. Если у меня уже есть класс `Book`, то я могу создать класс `Textbook`, а также класс `Novel`.

Как вы видите, идея подкласса заключается в том, что он делает все то же самое, что и родительский класс, но при этом содержит дополнительную функциональность. Наследование позволяет нам применять принцип DRY к нашим классам и сохранять их упорядоченность.

Как работает наследование в Python? Определите второй класс (т.е. подкласс), поместив родительский класс в скобки:

```
class Person ():
   def __init__ (self, name):
        self.name = name

   def greet (self):
        return f'Hello, {self.name}'
```

```
class Employee (Person)
    def __init__ (self, name, id_number):
        self.name = name
        self.id_number = id_number
```

Так мы сообщаем Python, что Employee — это Person, то есть он наследуется от Person.

Вам это кажется забавным? Так и должно быть — скоро увидите.

С помощью этого кода мы можем создать экземпляр `Employee`, как обычно:

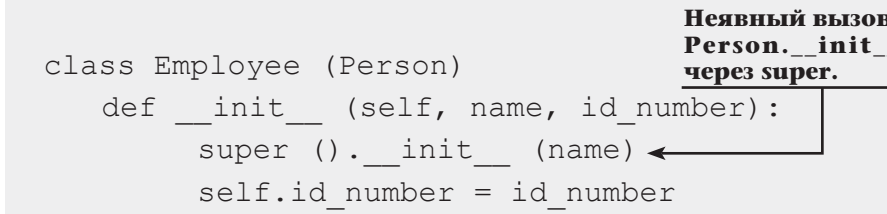
```
e = Employee ('empname', 1)
```

Но что произойдет, если мы вызовем `e.greet`? По правилу ICPO, Python сначала ищет атрибут `greet` в экземпляре `e`, но не находит его. Затем он ищет его в классе `Employee` и не находит. Затем Python обращается к родительскому классу `Person`, находит его, извлекает метод и вызывает его. Другими словами, наследование — это мощная технология, но в Python она является естественным следствием правила ICPO.

В моей реализации `Employee` есть одна странность, а именно то, что я установил `self.name` в `___init__`. Если вы работаете с таким языком, как Java, вам может быть интересно, почему я вообще должен его устанавливать, поскольку его уже устанавливает `Person.__init__`. Но в том-то и дело: в Python `__init__` действительно нужно выполнить, чтобы установить атрибут. Если бы мы удалили настройку `self.name` из `Employee.__init__`, атрибут никогда бы не был задан. По правилу ICPO всегда будет вызываться только один метод, и это будет тот, который находится ближе всего к экземпляру. Поскольку `Employee.__init__` ближе к экземпляру, чем `Person.__init__`, последний никогда не вызывается.

Хорошей новостью является то, что код, который я предоставил, работает. Но плохая новость в том, что это нарушает правило DRY, о котором я так часто упоминал.

Решение состоит в том, чтобы воспользоваться наследованием через `super`. Встроенная функция `super` позволяет нам вызывать метод для родительского объекта без явного указания имени этого родителя. Таким образом, в нашем коде мы могли бы переписать `Employee.__init__` следующим образом:



Упражнение 41. Чашка побольше

В то время как предыдущее упражнение, возможно, порадовало родителей и расстроило детей, наша работа как продавцов мороженого состоит в том, чтобы взволновать детей, а также забрать деньги у их родителей. Таким образом, наша компания начала предлагать продукт `BigBowl`, который может содержать до пяти мерных ложек.

Реализуйте `BigBowl` для этого упражнения таким образом, чтобы единственная разница между ним и классом `Bowl`, который мы создали ранее, заключалась в том, что он может иметь пять ложек, а не три. И да, это означает, что вы должны использовать наследование для достижения этой цели.

Вы можете изменить `Scoop` и `Bowl`, если необходимо, но такие изменения должны быть минимальными и оправданными.

ПРИМЕЧАНИЕ Как правило, целью наследования является добавление или изменение функциональности существующего класса без изменения родителя. Поэтому пуристам мо–

гут не понравиться эти инструкции, которые позволяют вносить изменения в родительский класс. Однако реальный мир не всегда идеален, и, если оба класса написаны одной и той же командой, вполне возможно, что автор дочернего класса сможет согласовать изменения в родительском классе.

Обсуждение

Это, надо признать, непростое дело, научит вас понимать, как работают атрибуты и особенно как они взаимодействуют между экземплярами, классами и родительскими классами. Если вы действительно понимаете правило ICPO, то решение будет рациональным.

В нашей предыдущей версии `Bowl.add_scoops` мы сказали, что хотим использовать `Bowl.max_scoops` для отслеживания максимально допустимого количества ложек. Все было нормально, пока все подклассы хотели использовать одно и то же значение.

Но здесь мы хотим использовать другое значение. То есть при вызове `add_scoops` для объекта `Bowl` максимальное значение должно быть `Bowl.max_scoops`. А при вызове `add_scoops` для объекта `BigBowl` максимальное значение должно быть `BigBowl.max_scoops`. И мы хотим избежать двойного написания `add_scoops`.

Самое простое решение — изменить нашу ссылку в `add_scoops` с `Bowl.max_scoops` на `self.max_scoops`. С этим изменением все будет работать следующим образом:

1. Если мы вызовем `Bowl.max_scoops`, мы получим 3.
2. Если мы вызовем `BigBowl.max_scoops`, мы получим 5.
3. Если мы вызовем `add_scoops` для экземпляра `Bowl`, то внутри метода мы запросим `self.max_scoops`. Согласно правилу поиска ICPO, Python сначала посмотрит на экземпляр, а затем на класс, которым в данном случае является `Bowl`, и вернет `Bowl.max_scoops` со значением 3.
4. Если мы вызовем `add_scoops` для экземпляра `BigBowl`, то внутри метода мы запросим `self.max_scoops`. Со-

гласно правилу поиска iCPO, Python сначала посмотрит на экземпляр, а затем на класс, которым в данном случае является `BigBowl`, и вернет `BigBowl.max_scoops` со значением 5.

Таким образом, мы воспользовались преимуществами наследования и гибкостью self, чтобы использовать один и тот же интерфейс для различных классов. Более того, мы смогли реализовать BigBowl с минимальным количеством кода, используя то, что мы уже написали для Bowl.

Решение

```
class Scoop ():
    def __init__ (self, flavor):
        self.flavor = flavor

    class Bowl ():
        max_scoops = 3

    def __init__ (self):
        self.scoops = []

    def add_scoops (self, *new_scoops):
        for one_scoop in new_scoops:
            if len (self.scoops) < self.max_scoops:
                self.scoops.append (one_scoop)

    def __repr__ (self):
        return '\n'.join (s.flavor for s in self.scoops)

class BigBowl (Bowl):
    max_scoops = 5

s1 = Scoop ('chocolate')
s2 = Scoop ('vanilla')
```

Bowl.max_scoops по-прежнему 3.

Использует self.max_scoops, а не Bowl.max_scoops, чтобы получить атрибут из правильного класса.

BigBowl.max_scoops теперь равен 5. 5.

```
s3 = Scoop ('persimmon')
s4 = Scoop ('flavor 4')
s5 = Scoop ('flavor 5')

bb = BigBowl ()
bb.add_scoops (s1, s2) bb.add_scoops (s3)
bb.add_scoops (s4, s5)
print (bb)
```

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr196].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr197].

После выполнения упражнения

Как я уже отмечал в этой главе, я думаю, что многие люди преувеличивают необходимость использования наследования в объектно–ориентированном коде. Но это не значит, что я считаю наследование ненужным или даже бесполезным. Правильно используемое, оно является мощным инструментом, который может уменьшить размер кода и улучшить его сопровождение.

Вот еще несколько способов попрактиковаться в использовании наследования:

1. Напишите класс `Envelope` с двумя атрибутами, `weight` (плавающая величина, измеряющая граммы) и `was_sent` (булево значение, по умолчанию `False`). Должно быть три метода: (1) `send`, который отправляет письмо и меняет значение `was_sent` на `True`, но только после того, как на конверте будет достаточно почтовых марок; (2) `add_postage`, который добавляет почтовые марки, равные значению его аргумента; и (3) `postage_needed`, который указывает, сколько почтовых марок требуется конверту в целом. Необходимое количество почтовых марок будет

равно весу конверта, умноженному на 10. Теперь напишите класс `BigEnvelope`, который работает так же, как `Envelope`, за исключением того, что почтовые расходы будут в 15 раз больше веса, а не в 10.

2. Создайте класс `Phone`, представляющий мобильный телефон. (Стационарные телефоны еще существуют?) Телефон должен реализовать метод `dial`, который набирает номер телефона (или имитирует это). Реализуйте подкласс `SmartPhone`, который использует метод `Phone.dial`, но при этом реализует свой собственный метод `run_app`. Теперь реализуйте подкласс `iPhone`, который реализует не только метод `run_app`, но и свой собственный метод `dial`, который вызывает метод `dial` родительского класса, но вывод которого в знак его крутости написан в нижнем регистре.
3. Определите класс `Bread`, представляющий буханку хлеба. Мы должны иметь возможность вызвать метод `get_nutrition` для объекта, передав целое число, представляющее количество ломтиков, которые мы хотим съесть. В ответ мы получим словарь, пары ключ–значение которой будут представлять калории, углеводы, натрий, сахар и жир, указывая статистику питания для данного количества ломтиков.

Теперь реализуйте два новых класса, которые наследуются от `Bread`, а именно `WholeWheatBread` и `RyeBread`. Каждый класс должен реализовать один и тот же метод `get_nutrition`, но с разной информацией о питании, где это необходимо.

Упражнение 42. FlexibleDict

Я уже говорил, что основной смысл наследования заключается в использовании преимуществ существующей функциональности. Есть несколько способов и причин сделать это, и одна из них — создать новое поведение, которое похоже на существу–

ющий класс, но отличается от него. Например, Python поставляется не только с `dict`, но и с `Counter` и `defaultdict`. Наследуясь от `dict`, эти два класса могут реализовать только те методы, которые отличаются от `dict`, полагаясь на исходный класс для большинства функций.

В этом упражнении мы также реализуем подкласс `dict`, который я назову `FlexibleDict`. Ключи словаря — это объекты Python, и как таковые они идентифицируются с типом. Поэтому если вы используете ключ 1 (целое число) для хранения значения, то вы не можете использовать ключ '1' (строка) для извлечения этого значения. Однако `FlexibleDict` позволяет это сделать. Если он не найдет ключ пользователя, он попытается преобразовать ключ в `str` и `int`, прежде чем сдаться, например:

```
fd = FlexibleDict ()

fd ['a'] = 100
print (fd ['a'])

fd [5] = 500
print (fd [5])

fd [1] = 100
print (fd ['1'])

fd ['1'] = 100 print (fd [1])
```

Печатает 100, как обычный словарь.

Печатает 500, как обычный словарь.

int ключ.

Печатает 100, хотя мы передали строку str.

str key.

Печатает 100, хотя мы передали int.

Обсуждение

Класс в этом упражнении, `FlexibleDict`, является примером того, почему вы можете захотеть наследовать от встроенного типа. Это довольно редко, но, как вы можете заметить, это позволяет нам создать альтернативный тип словаря.

Спецификация `FlexibleDict` указывает, что все должно работать как обычный словарь, за исключением поиска. Таким

образом, нам нужно переопределить только один метод — `__getitem__`, который всегда связан с квадратными скобками в Python. В самом деле, если вы когда–нибудь задавались вопросом, почему строки, списки, кортежи и словари определяются по–разному, но все используют квадратные скобки, причина в этом методе.

Поскольку все должно быть таким же, как `dict`, за исключением этого единственного метода, мы можем наследоваться от словаря, написать один метод и готово.

Этот метод получает ключевой аргумент. Если ключа нет в словаре, то пытаемся превратить его в строку и целое число. Поскольку мы можем столкнуться с ошибкой `ValueError`, пытающейся преобразовать ключ в целое число, мы по пути перехватываем эту ошибку. На каждом шагу мы проверяем, может ли на самом деле работать версия ключа с другим типом, и если да, то мы переназначаем значение ключа.

В конце метода мы вызываем наш родительский метод `__getitem__`. Почему бы нам просто не использовать квадратные скобки? Потому что это приведет к бесконечному циклу, поскольку квадратные скобки определены для вызова `__getitem__`. Другими словами, `a [b]` превращается в `a.__getitem__ (b)`. Если мы затем включим `self [b]` в определение `__getitem__`, мы получим вызов самого метода. Таким образом, нам нужно явно вызвать родительский метод, который в любом случае вернет связанное значение.

ПРИМЕЧАНИЕ Хотя `FlexibleDict` (и некоторые из задач «Помимо упражнений») могут быть отличными для обучения навыкам работы с Python, встраивание такой гибкости в Python совершенно не соответствует Python и не рекомендуется. Одна из ключевых идей в Python — код должен быть однозначным, Python лучше получить ошибку, чем предполагать.

Решение

```
class FlexibleDict (dict):
    def __getitem__ (self, key):
        try:
            if key in self:
                pass
            elif str (key) in self:
                key = str (key)
            elif int (key) in self:
                key = int (key)
        except ValueError:
            pass
        return dict.__getitem__ (self, key)

fd = FlexibleDict ()

fd ['a'] = 100
print (fd ['a'])

fd [5] = 500
print (fd [5])

fd [1] = 100
print (fd ['1'])

fd ['1'] = 100
print (fd [1])
```

__getitem__ — это то, что вызывают квадратные скобки [ ].

Есть ли у нас запрашиваемый ключ?

Если нет, то попробуем преобразовать его в строку.

Если нет, то пытаемся превратить его в целое число.

Если мы не можем превратить его в целое число, то игнорируем его.

Попробуйте использовать обычный словарь __getitem__, либо с оригинальным ключом, либо с модифицированным.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr198].

198

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr199].

199

После выполнения упражнения

Теперь мы увидели, как расширить встроенный класс с помощью наследования. Вот еще несколько упражнений, в которых вы также поэкспериментируете с расширением некоторых встроенных классов:

1. С `FlexibleDict` мы разрешили пользователю использовать любой ключ, но при этом были гибкими при извлечении. Реализуйте `StringKeyDict`, который преобразует свои ключи в строки как часть назначения. Таким образом, сразу после того, как вы напишите `skd [1] = 10`, вы сможете написать `skd ['1']` и получить возвращенное значение 10. Это может пригодиться, если вы будете считывать ключи из файла и не сможете отличить строки от целых чисел.
2. Класс `RecentDict` работает так же, как `dict`, за исключением того, что он содержит заданное пользователем количество пар ключ–значение, которое определяется при создании экземпляра. В `RecentDict (5)` сохраняются только пять последних пар ключ–значение: если пар больше пяти, то самый старый ключ удаляется вместе со своим значением. Примечание: ваша реализация может учитывать тот факт, что современные словари хранят свои пары ключ–значение в хронологическом порядке.
3. Класс `FlatList` наследуется от списка и переопределяет метод `append`. Если `append` передается итерируемый объект, то он должен добавлять каждый элемент итерируемого объекта отдельно. Это означает, что `fl.append ([10, 20, 30])` не добавит список `[10, 20, 30]` к `fl`, а добавит отдельные целые числа 10, 20 и 30. Вы можете использовать встроенную функция `iter`, чтобы определить, действительно ли переданный аргумент является итерируемым.

Упражнение 43. Животные

Для последних трех упражнений в этой главе мы собираемся создать набор классов, объединяющих все идеи, рассмотренные в этой главе: классы, методы, атрибуты, композицию и наследование. Одно дело изучать и использовать их по отдельности, но, когда вы объединяете эти техники вместе, вы видите их возможности и понимаете организационные и семантические преимущества, которые они дают.

Для целей этих упражнений вы — директор по информационным технологиям в зоопарке. В зоопарке содержится несколько различных видов животных, и по бюджетным соображениям некоторые из них должны быть размещены вместе с другими животными.

Мы будем представлять животных в виде объектов Python, причем каждый вид определяется как отдельный класс. Все объекты определенного класса будут иметь одинаковый вид и количество лап, но цвет будет отличаться у разных экземпляров. Таким образом, мы можем создать белую овцу:

```
s = Sheep ('white')
```

Аналогичным образом я могу получить информацию о животном из объекта, извлекая его атрибуты:

```
print (s.species)      ← Напечатает sheep.
print (s.color)        ← Напечатает white.
print (s.number_of_legs) ← Напечатает «4».
```

Если я преобразую `animal` в строку (используя `str` или `print`), я получу строку, объединяющую все эти детали:

```
print (s)  ← Напечатает White sheep, 4 legs.
```

Мы будем считать, что в нашем зоопарке содержатся четыре разных типа животных: овцы, волки, змеи и попугаи (зоопарк

испытывает некоторые бюджетные трудности, поэтому коллекция животных у нас небольшая и необычная). Создайте классы для каждого из этих типов, чтобы мы могли распечатать каждый из них и получить отчет о цвете, виде и количестве лап.

Обсуждение

Конечная цель очевидна: мы хотим создать четыре различных класса (Wolf, Sheep, Snake и Parrot), каждый из которых принимает единственный аргумент (цвет). Результатом вызова каждого из этих классов будет новый экземпляр с тремя атрибутами: `species`, `color` и `number_of_legs`.

Наивная реализация просто создаст каждый из этих четырех классов. Но, конечно, часть смысла здесь заключается в использовании наследования, и тот факт, что поведение в каждом классе в основном идентично, означает, что мы действительно можем воспользоваться этим. Но что войдет в класс `Animal`, от которого все наследуют, и что войдет в каждый из отдельных подклассов?

Поскольку все классы животных будут содержать одинаковые атрибуты, мы можем определить `__repr__` в классе `Animal`, от которого они все наследуют. В моей версии используется f-строка и атрибуты берутся из self. Обратите внимание, что `self` в этом случае будет экземпляром не `Animal`, а одного из классов, которые наследуются от `Animal`.

Итак, что еще должно быть в `Animal` и что должно быть в подклассах? Здесь нет жесткого и однозначного правила, но в данном конкретном случае я решил, что `Animal.__init__` будет местом назначения, а метод `__init__` в каждом подклассе будет вызывать `Animal.__init__` с жестко заданным количеством лап, а также цветом, указанным пользователем (рисунок 9.13).

Теоретически, `__init__` в подклассе может вызывать `Animal.__init__` напрямую и по имени. Но у нас также есть доступ к super, который возвращает объект, для которого наш метод должен быть вызван.

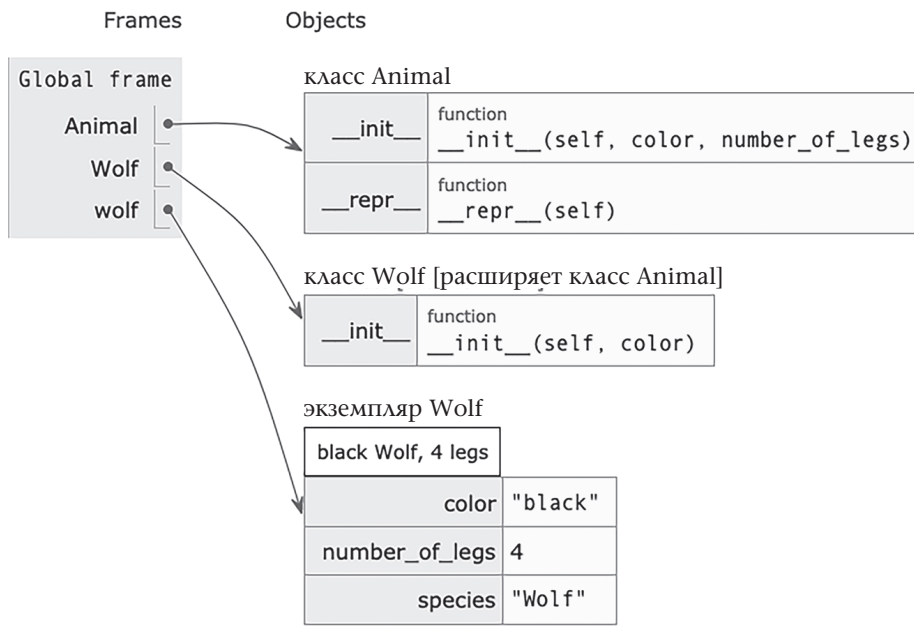


Рисунок 9.13. `Wolf` наследуется от `Animal`. Обратите внимание, какие методы где определены.

Другими словами, вызывая `super().__init__`, мы знаем, что нужный метод будет вызван для нужного нам объекта, и мы можем просто передать аргументы `color` и `number_of_legs`.

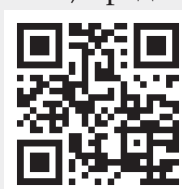
Но подождите, а как же атрибут `species`? Как мы можем задать его без участия пользователя?

Мое решение этой проблемы состояло в том, чтобы воспользоваться тем фактом, что классы Python очень похожи на модули со схожим поведением. Точно так же, как модуль имеет атрибут `__name__`, отражающий, какой модуль был загружен, так и классы содержат атрибут `__name_`, представляющий собой строку, содержащую имя текущего класса. Таким образом, если я вызываю `self.__class__` для объекта, я получаю его класс, а если я вызываю `self.__class__.__name__`, я получаю строковое представление класса.

Абстрактные базовые классы

Здесь класс `Animal` — это то, что другие языки могли бы назвать абстрактным базовым классом, что означает, что экземпляр данного класса на самом деле не будет создан, но от которого будут наследоваться другие классы. В Python вам не нужно объявлять такой класс абстрактным, но вы также не получите принудительного исполнения, как в других языках. Если вы действительно хотите, вы можете импортировать ABC–Meta из модуля `abc` (абстрактный базовый класс). Следуя его инструкциям, вы сможете объявить определенные методы абстрактными, что означает, что они должны быть переопределены в дочернем элементе.

Я не большой поклонник абстрактных базовых классов: я думаю, что достаточно задокументировать класс как абстрактный, без накладных расходов или языкового принуждения. Является ли это разумным подходом, зависит от нескольких факторов, в том числе от характера и размера проекта, над которым вы работаете, и от того, имеете ли вы опыт работы с динамическими языками. Большой проект с участием большого количества разработчиков, вероятно, выиграет от дополнительных мер безопасности, предоставляемых абстрактным базовым классом.


200

Если вы хотите узнать больше об абстрактных базовых классах в Python, вы можете прочитать об ABCMeta здесь: [qr200].

Решение

Наш базовый класс Animal принимает цвет и количество лап.

Превращает текущий объект класса в строку.

```
class Animal ():
   def __init__ (self, color, number_of_legs):
        self.species = self.__class__.__name__
        self.color = color
        self.number_of_legs = number_of_legs
```

```
    def __repr__ (self):
        return f'{self.color} {self.species},
        → {self.number_of_legs} legs'

class Wolf (Animal):
    def __init__ (self, color):
        super ().__init__ (color, 4)

class Sheep (Animal):
    def __init__ (self, color):
        super ().__init__ (color, 4)

class Snake (Animal):
    def __init__ (self, color):
        super ().__init__ (color, 0)

class Parrot (Animal):
    def __init__ (self, color):
        super ().__init__ (color, 2)

wolf = Wolf ('black')
sheep = Sheep ('white')
snake = Snake ('brown')
parrot = Parrot ('green')

print (wolf)
print (sheep)
print (snake)
print (parrot)
```

Использует f-строку для получения соответствующего вывода.

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr201].

201

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr202].

202

После выполнения упражнения

В этом упражнении мы ввели несколько классов как часть иерархии. Вот некоторые дополнительные способы работы с наследованием и размышления о последствиях принимаемых нами проектных решений. Я должен отметить, что эти вопросы, а также вопросы, приведенные в этой главе, будут сочетать практическую практику с некоторыми более глубокими философскими вопросами о «правильном» способе работы с объектно-ориентированными системами:

1. Вместо того чтобы каждый класс животных наследовался напрямую от `Animal`, определите несколько новых классов `ZeroLeggedAnimal`, `TwoLeggedAnimal` и `FourLeggedAnimal`, каждый из которых наследуется от `Animal`, и определите количество лап для каждого экземпляра.

   Теперь измените `Wolf`, `Sheep`, `Snake` и `Parrot` так, чтобы каждый класс наследовался от одного из этих новых классов, а не непосредственно от `Animal`. Как это влияет на определения ваших методов?
2. Вместо того чтобы писать метод `__init__` в каждом подклассе, мы могли бы также использовать атрибут класса, `number_of_legs`, в каждом подклассе — аналогично тому, что мы делали ранее с `Bowl` и `BigBowl`. Реализуйте иерархию таким образом. Вам вообще нужен метод `__init__` в каждом подклассе или будет достаточно `Animal.__init__`?
3. Предположим, что метод `__repr__` каждого класса должен печатать звук животного, а также стандартную строку, которую мы реализовали ранее. Другими словами, `str(sheep)` будет `Baa — белая овца, 4 ноги`. Как бы вы использовали наследование для максимального повторного использования кода?

Упражнение 44. Клетки

Теперь, когда мы создали несколько животных, пришло время поместить их в клетки. Для этого упражнения создайте класс `Cage`, в который вы можете поместить одно или несколько животных, следующим образом:

```
c1 = Cage (1)
c1.add_animals (wolf, sheep)

c2 = Cage (2)
c2.add_animals (snake, parrot)
```

Когда вы создаете новую клетку (`Cage`), вы присваиваете ей уникальный идентификационный номер. (Уникальность не обязательна, но это поможет нам различать клетки.) Затем вы сможете вызывать add_animals для новой клетки, передавая любое количество животных, которые будут помещены в клетку. Я также хочу, чтобы вы определили метод `__repr__`, чтобы при печати клетки печатался не только идентификатор клетки, но и каждое из содержащихся в ней животных.

Обсуждение

Определение класса `Cage` в решении в чем-то похоже на класс Bowl, который мы определили ранее в этой главе.

Когда мы создаем новую клетку, метод `__init__` инициализирует `self.animals` пустым списком, что позволяет нам добавлять (и даже удалять) животных по мере необходимости. Мы также храним переданный нам идентификационный номер в параметре `id_number`.

Затем мы реализуем `Cage.add_animals`, в котором используются методы, аналогичные тем, что мы использовали в `Bowl.add_scoops`. И снова мы используем оператор splat `(*)`, чтобы получить все аргументы в одном кортеже (`animals`). Хотя мы могли бы использовать `list.extend` для добавления всех новых животных в `list.animals`, я все равно буду использовать здесь цикл for, чтобы добавлять их по одному. Вы можете увидеть, как Python Tutor изображает двух животных в клетке на рисунке 9.14.

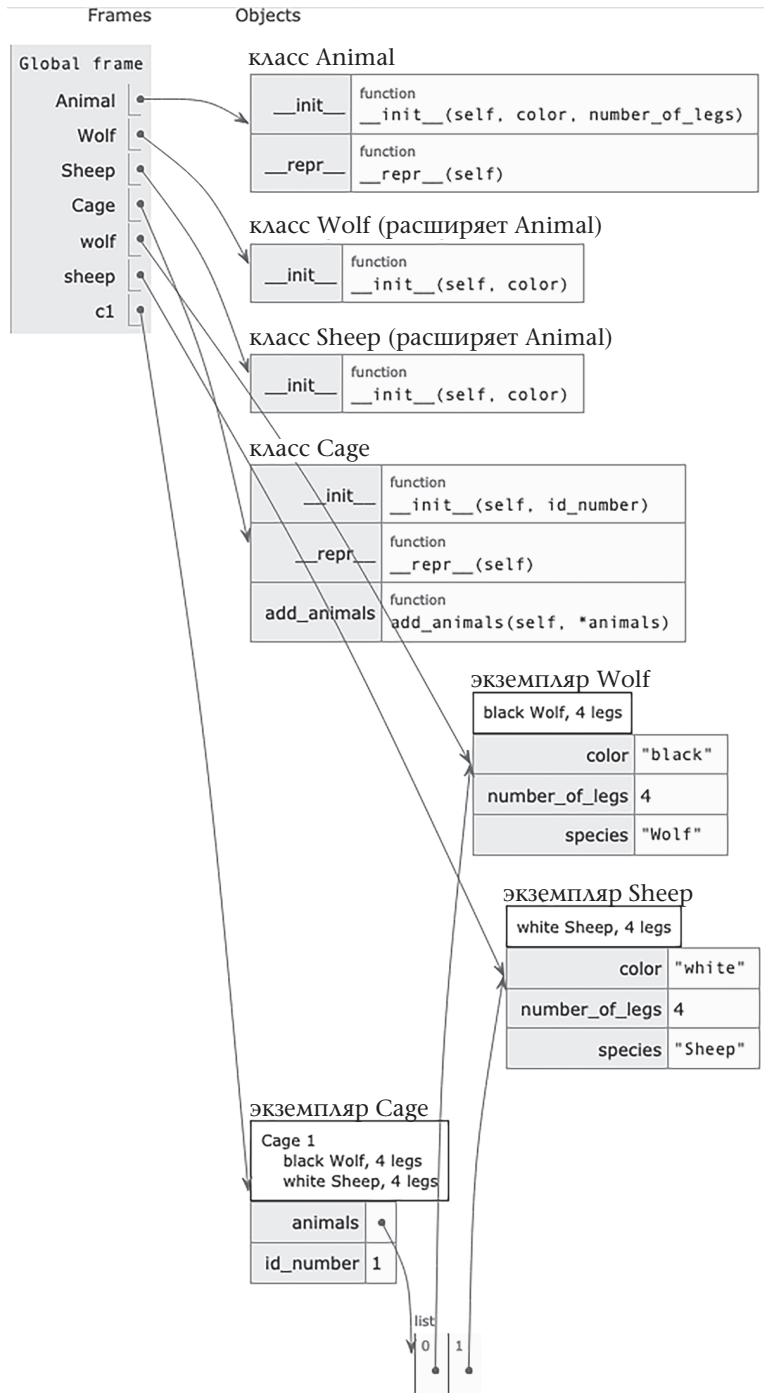


Рисунок 9.14. Экземпляр Cage, содержащий одного волка и одну овцу.

На мой взгляд, самая интересная часть нашего определения `Cage` — это использование `__repr__` для создания отчета. Для клетки `c1` команда `print (c1)` напечатает идентификатор клетки, а затем всех животных в клетке, используя их наглядные представления. Мы делаем это, сначала печатая базовый заголовок, что не так уж и сложно. Но затем мы берем каждое животное в `self.animals` и используем выражение-генератор (то есть ленивую форму генератора списка), чтобы вернуть последовательность строк. Каждая строка в этой последовательности будет состоять из табуляции и описания животного. Затем мы передаем результат нашего выражения-генератора в `str.join`, который помещает символы новой строки между каждым животным.

Решение

```
class Animal ():
   def __init__ (self, color, number_of_legs):
   self.species = self.__class__.__name__
   self.color = color self.number_of_legs =
   number_of_legs

def __repr__ (self):
   return f'{self.color} {self.species}, {self.
   number_of_legs} legs'

class Wolf (Animal):
   def __init__ (self, color):
        super ().__init__ (color, 4)

class Sheep (Animal):
   def __init__ (self, color):
        super ().__init__ (color, 4)

class Snake (Animal):
   def __init__ (self, color):
        super ().__init__ (color, 0)
```

```
class Parrot (Animal):
    def __init__ (self, color):
        super ().__init__ (color, 2)

class Cage ():
    def __init__ (self, id_number):
        self.id_number = id_number
        self.animals = []

    def add_animals (self, *animals):
        for one_animal in animals:
            self.animals.append (one_animal)

def __repr__ (self):
    output = f'Cage {self.id_number} \n'
    output += '\n'.join ('\t' + str (animal)
        for animal in self.animals)

        return output

wolf = Wolf ('black')
sheep = Sheep ('white')
snake = Snake ('brown')
parrot = Parrot ('green')

c1 = Cage (1)
c1.add_animals (wolf, sheep)
c2 = Cage (2)
c2.add_animals (snake, parrot)

print (c1)
print (c2)
```

Устанавливает идентификационный номер для каждой клетки, чтобы мы могли различать их.

Устанавливает пустой список, в который мы будем помещать. животных.

Строка для каждой клетки будет в основном состоять из строки, основанной на выражении-генераторе.


203

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr203].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr204].

После выполнения упражнения

Мы снова видим потребность в композиции в наших классах — создании объектов, которые являются контейнерами для других объектов. Вот несколько возможных расширений этого кода, все они основаны на идеях, которые мы уже видели в этой главе и которые вы увидите повторяющимися почти в каждой объектно-ориентированной системе, которую вы будете создавать и с которой столкнетесь:

1. Как видите, нет никаких ограничений на количество животных, которые потенциально могут быть помещены в клетку. Точно так же, как мы установили ограничение в три ложки в `Bowl` и пять в `BigBowl`, вы должны аналогичным образом создать классы `Cage` и `BigCage`, которые ограничивают количество помещенных животных.
2. Не очень реалистично говорить, что мы будем ограничивать количество животных в клетке. Скорее, имеет смысл описать, сколько места необходимо каждому животному, и убедиться, что общее количество места, необходимое для каждого животного, не превышает пространство в каждой клетке.

   Таким образом, вам следует изменить каждый из подклассов `Animal`, чтобы включить атрибут `space_required`. Затем измените классы `Cage` и `BigCage`, чтобы отразить, сколько места есть в каждом из них. Добавление большего количества животных, чем может вместить клетка, должно вызвать исключение.
3. Наши смотрители зоопарка обладают мрачным чувством юмора, когда дело доходит до размещения животных вместе: они помещают волков и овец в первую клетку, а змей и птиц — в другую. (Хорошая новость заключается в том, что при такой конфигурации зоопарк сможет сэкономить

на корме для половины животных.) Определите словарь, описывающий, какие животные могут находиться вместе с другими. Ключами в словаре будут классы, а значениями — списки классов, которые могут быть размещены вместе с ключами. Затем, при добавлении новых животных в текущую клетку, вы будете проверять их совместимость. Попытка добавить животное в клетку, в которой уже содержится несовместимое животное, вызовет исключение.

УПРАЖНЕНИЕ 45. Зоопарк

Наконец, пришло время создать наш объект `Zoo`. Он будет содержать объекты вольеров, а они, в свою очередь, будут содержать животных. Наш класс `Zoo` должен поддерживать следующие операции:

1. Мы должны иметь возможность распечатать все клетки зоопарка `z` (с их идентификационными номерами) и животных, находящихся в них, просто вызвав `print (z)`.
2. Мы должны иметь возможность получить животных определенного цвета, вызвав метод `z.animals_by_color`. Например, мы можем получить всех черных животных, вызвав метод `z.animals_by_color ('black')`. Результатом должен быть список объектов `Animal`.
3. Мы должны иметь возможность получить животных с определенным количеством ног, вызвав метод `z.animals_by_legs`. Например, мы можем получить всех четвероногих животных, вызвав метод `z.animals_by_legs (4)`. Результатом должен быть список объектов `Animal`.
4. Наконец, у нас есть потенциальный спонсор нашего зоопарка, который хочет предоставить носки для всех животных. Таким образом, нам нужно иметь возможность вызвать `z.number_of_legs ()` и получить подсчет общего количества ног (или лап) у всех животных в нашем зоопарке.

Следовательно, задача состоит в том, чтобы создать класс `Zoo`, для которого мы можем вызвать следующие функции:

```
z = Zoo ()
z.add_cages (c1, c2)

print (z)
print (z.animals_by_color ('white'))
print (z.animals_by_legs (4))
print (z.number_of_legs ())
```

Обсуждение

В некотором смысле наш класс `Zoo` очень похож на наш класс `Cage`. У него есть список атрибутов, `self.cages`, в котором мы будем хранить клетки. У него есть метод `add_cages`, который принимает `*args` и, таким образом, принимает любое количество входов. Даже метод `__repr__` похож на то, что мы делали с `Cage.__repr__`. Мы просто используем `str.join` для вывода, полученного в результате выполнения str для каждой клетки, точно так же, как клетки запускают str для каждого животного. Аналогично мы будем использовать выражение–генератор, которое будет немного эффективнее, чем генератор списка.

Но затем, когда дело дойдет до трех методов, которые нам нужно было создать, мы немного поменяем направление. В методах `animals_by_color` и `animals_by_legs` мы хотим получить животных определенного цвета или с определенным количеством лап. Здесь мы воспользуемся тем, что зоопарк содержит список клеток, а каждая клетка содержит список животных. Тем самым мы можем использовать вложенный генератор списка, получив список всех животных.

Но, конечно, нам не нужны все животные, поэтому у нас есть оператор `if`, который отсеивает тех, кто нам не нужен. В случае с `animals_by_color` мы включаем только тех животных, которые имеют нужный цвет, а что касается `animals_by_legs` — мы сохраняем только тех животных, у которых есть требуемое количество лап.

Но у нас также есть `number_of_legs`, который работает немного по–другому. Здесь мы хотим получить целое число, отражающее количество лап во всем зоопарке. Здесь мы можем воспользоваться встроенным методом `sum`, передав ему выражение–генератор, осуществив проход через каждую клетку и получив количество лап у каждого животного. Таким образом, метод вернет целое число.

Хотя лагеря объектно–ориентированного и функционального программирования десятилетиями спорят о том, какой подход лучше, я думаю, что методы класса `Zoo` показывают нам, что у каждого из них есть свои сильные стороны и что наш код может быть коротким, элегантным и точным, если мы объединим эти методы. Тем не менее, я часто встречаю сопротивление от студентов, которые видят этот код и говорят, что это нарушение объектно–ориентированного принципа инкапсуляции, который гарантирует, что мы не можем (или не должны) напрямую обращаться к данным других объектов.

Правильно это или нет, но такие нарушения довольно часто встречаются и в мире Python.

Поскольку все данные являются общедоступными (т.е. нет ни `private`, ни `protected`), считается хорошим и разумным просто «вычерпывать» данные из объектов. Однако это также означает, что тот, кто пишет класс, обязан документировать его и поддерживать API — или документировать элементы, которые в будущем могут устареть или которые могут удалить.

Решение

Это самое длинное и сложное определение класса в этой главе — и тем не менее, каждый из методов использует приемы, которые мы обсуждали как в этой главе, так и в этой книге:

```
class Zoo ():
    def __init__ (self):
        self.cages = []
```

Устанавливает атрибут self.cages, список, в котором мы будем хранить клетки.

```
    def add_cages (self, *cages):
        for one_cage in cages:
            self.cages.append (one_cage)

    def __repr__ (self):
        return '\n'.join (str (one_cage)
            for one_cage in self.cages)

    def animals_by_color (self, color):
        return [one_animal
            for one_cage in self.cages
            for one_animal in one_cage.animals
            if one_animal.color == color]
```

Определяет метод, который будет возвращать объекты животных, соответствующие выбранному цвету.

```
    def animals_by_legs (self, number_of_legs):
        return [one_animal
            for one_cage in self.cages
            for one_animal in one_cage.animals
            if one_animal.number_of_legs ==
                number_of_legs]
```

Определяет метод, который будет возвращать объекты животных, соответствующие количеству лап.

Возвращает количество лап.

```
    def number_of_legs (self):
        return sum (one_animal.number_of_legs
            for one_cage in self.cages
            for one_animal in one_cage.animals)

wolf = Wolf ('black')
sheep = Sheep ('white')
snake = Snake ('brown')
parrot = Parrot ('green')

print (wolf)
print (sheep)
```

```
print (snake)
print (parrot)

c1 = Cage (1)
c1.add_animals (wolf, sheep)

c2 = Cage (2)
c2.add_animals (snake, parrot)

z = Zoo ()
z.add_cages (c1, c2)

print (z)
print (z.animals_by_color ('white'))
print (z.animals_by_legs (4))
print (z.number_of_legs ())
```

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr205].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr206].

После выполнения упражнения

Теперь, когда вы увидели, как все эти элементы сочетаются друг с другом в нашем классе `Zoo`, вот несколько дополнительных упражнений, которые вы, возможно, захотите попробовать, чтобы расширить то, что мы уже сделали, и лучше понять объектно-ориентированное программирование на Python:

1. Измените множество `animals_by_color` так, чтобы оно принимал любое количество цветов. Животные любого из перечисленных окрасов должны быть возвращены. Метод должен вызвать исключение, если цвета не переданы.
2. В настоящее время мы обращаемся с нашим классом `Zoo`

так, будто это одноэлементный объект, то есть класс, который имеет только один экземпляр. Каким грустным был бы мир, если бы в нем был только один зоопарк! Предположим, что у нас есть два экземпляра `Zoo`, представляющие два разных зоопарка, и мы хотим перенести животное из одного в другой. Реализуйте метод `Zoo.transfer_animal`, который принимает `target_zoo` и подкласс `Animal` в качестве аргументов. Первое животное указанного типа удаляется из зоопарка, для которого мы вызвали метод, и помещается в первую клетку нужного зоопарка.

3. Объедините методы `animals_by_color` и `animals_by_legs` в один метод `get_animals`, который использует `kwargs` для получения имен и значений. Единственными допустимыми именами были бы `color` и `legs`. Затем метод будет использовать одно или оба этих ключевых слова для создания запроса, который возвращает тех животных, которые соответствуют переданным критериям.

Подводя итоги

Объектно-ориентированное программирование — это набор методов, но это также и образ мышления. Во многих языках вам навязывают объектно-ориентированное программирование, так что вы постоянно пытаетесь программировать согласно его синтаксису и структуре. Python пытается найти баланс, предлагая все объектно-ориентированные функции, которые мы, вероятно, захотим или используем, но простым, бесконфликтным способом. Таким образом, объекты Python предоставляют нам структуру и организацию, которые могут облегчить написание, чтение и (самое главное) сопровождение нашего кода.

10. Итераторы и генераторы

Вы когда–нибудь замечали, что многие объекты Python знают, как вести себя внутри цикла `for`? Это не случайно. Итерация настолько полезна и настолько распространена, что Python позволяет легко сделать объект итерируемым. Для этого достаточно реализовать несколько моделей поведения, известных как протокол итератора.

В этой главе мы рассмотрим этот протокол и то, как его можно использовать для создания итерируемых объектов. Мы сделаем это тремя способами:

1 Мы создадим собственные итераторы с помощью классов Python, непосредственно реализуя протокол.

2 Мы создадим генераторы, объекты, реализующие протокол, на основе чего–то, что очень похоже на функцию. Неудивительно, что они известны как функции–генераторы.

3 Мы также будем создавать генераторы с помощью выражения–генераторов, которые очень похожи на списки.

Даже новички в Python знают, что если вы хотите перебрать символы в строке, то вы можете написать:

```
for i in 'abcd':
print (i)
```

Напечатает каждое из a, b, c и d, в отдельной строке.

Это кажется само собой разумеющимся, и в этом весь смысл. А что, если вам нужно просто выполнить фрагмент кода пять раз?

Можете ли вы выполнить итерацию над целым числом 5? Многие новички в Python полагают, что ответ «да», и пишут следующее:

```
for i in 5:
print (i)
```

Это не сработает.

Этот код выдает ошибку:

```
TypeError: объект 'int' не является итерируемым
```

Из этого мы видим, что в то время как строки, списки и словари являются итерируемыми, целые числа таковыми не являются. Они не являются таковыми, потому что не реализуют протокол итератора, который состоит из трех частей:

1. Метод `__iter__`, который возвращает итератор.
2. Метод `__next__`, который должен быть определен для итератора.
3. Исключение `StopIteration`, которое итератор выдает при окончании итерации.

Последовательности (строки, списки и кортежи) являются наиболее распространенной формой итерируемых объектов, но большое количество других объектов, таких как файлы и словари, также являются итераторами. Лучше всего то, что при определении собственных классов вы можете сделать их итерируемыми. Все, что вам нужно сделать, это убедиться, что протокол итератора применен к объекту.

С учетом данных трех пунктов мы можем теперь понять, что на самом деле делает цикл `for`:

1. Он спрашивает объект, является ли он итерируемым, используя встроенную функцию `iter`. Эта функция вызывает метод `__iter__` для нужного объекта. То, что возвращает `__iter__`, называется итератором.
2. Если объект является итерируемым, то цикл `for` вызывает следующую встроенную функцию у возвращенного итератора. Эта функция вызывает `__next__` у итератора.
3. Если `__next__` вызывает исключение `Stopiteration`, то цикл завершается.

Этот протокол объясняет пару вещей, которые обычно озадачивают новичков в Python:

1 Почему нам не нужны индексы? В языках, подобных C, нам нужен числовой индекс для наших итераций. Это нужно для того, чтобы цикл мог пройтись по каждому элементу коллекции по очереди. В этих случаях цикл отвечает за отслеживание текущего местоположения. В Python за создание следующего элемента отвечает сам объект. Цикл `for` не знает, находимся ли мы на первом элементе или на последнем. Но он знает, когда мы достигли конца.

2 Как получается, что разные объекты ведут себя по-разному в циклах `for`? Ведь строки возвращают символы, словари возвращают ключи, а файлы возвращают строки. Ответ заключается в том, что объект-итератор может возвращать все, что захочет. Поэтому строковые итераторы возвращают символы, словари-итераторы возвращают ключи, а файловые итераторы возвращают строки в файле.

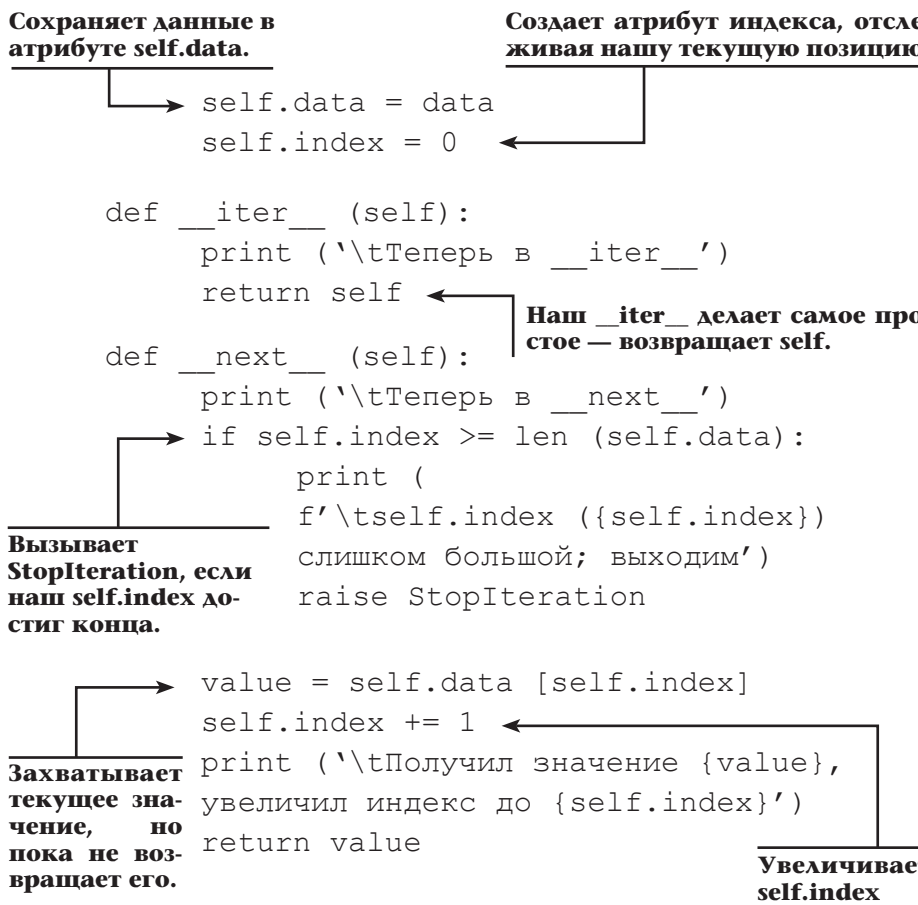
Если вы определяете новый класс, вы можете сделать его итерируемым следующим образом:

1. Определите метод `__iter__`, который принимает в качестве аргумента только `self` и возвращает `self`. Другими словами, когда Python спросит ваш объект: «Ты итерируемый?» Ответ будет: «Да, и я сам себе итератор».
2. Определите метод `__next__`, который принимает в качестве аргумента только `self`. Этот метод должен либо возвращать значение, либо исключение `StopIteration`. Если он никогда не возвращает `StopIteration`, то любой цикл `for` на этом объекте никогда не завершится.

Есть и более сложные способы, включая возврат другого объекта из `__iter__`. Я покажу и объясню это позже.

Вот простой класс, который реализует протокол, оборачивая себя вокруг итерируемого объекта, указывая, когда он проходит через каждый этап итерации:

```
class LoudIterator ():
        def __init__ (self, data):
        print ('\tТеперь в __init__')
```




```
for one_item in LoudIterator ('abc'):
    print (one_item)
```

Если мы выполним этот код, то получим следующий результат:

```
Теперь в __init__
        Теперь в __iter__
        Теперь в __next__
        Получил значение a, увеличил индекс до 1
a
        Теперь в __next__
        Получил значение b, увеличил индекс до 2
b
        Теперь в __next__
        Получил значение c, увеличил индекс до 3
```

```
c
        Теперь в __next__
        self.index (3) слишком большой; выходим
```

Этот вывод показываем нам процесс итерации, который мы уже видели ранее, начиная с вызова `__iter__` и затем повторяющихся вызовов `__next__`. Цикл завершается, когда итератор вызывает StopIteration.

Добавление таких методов в класс работает, когда вы создаете свои собственные новые типы. В Python есть еще два способа создания итераторов:

1 Можно использовать выражение–генератор, которое мы уже видели и использовали. Как вы, возможно, помните, выражения–генераторы выглядят и работают так же, как и генераторы списков, за исключением того, что вы используете круглые скобки, а не квадратные. Но в отличие от генераторов списков, которые возвращают списки, занимающие много памяти, выражения–генераторы возвращают по одному элементу за раз.

2 Вы можете использовать функцию–генератор — нечто, что выглядит как функция, но при выполнении действует как итератор, например:

```
def foo ():
yield 1
yield 2
yield 3
```

Когда мы вызываем `foo`, тело функции не выполняется. Вместо этого мы получаем обратно объект–генератор, то, что реализует протокол итератора. Таким образом, мы можем поместить его в цикл `for`:

```
g = foo ()
for one_item in g:
print (one_item)
```

Этот цикл выведет 1, 2 и 3. Почему? Потому что с каждой итерацией (т.е. каждый раз, когда мы вызываем `next` на `g`)

функция выполняет следующий оператор yield, возвращает значение, полученное из `yield`, и затем «засыпает», ожидая следующей итерации. Когда функция-генератор выходит из цикла, она автоматически вызывает `StopIteration`, тем самым завершая цикл.

Итераторы широко распространены в Python, потому что они очень удобны — и во многом они стали удобными именно потому, что широко распространены. В этой главе вы попрактикуетесь в написании всех этих типов итераторов и поймете, когда следует использовать каждый из этих методов.

Итерируемый объект vs. итератор

Термины «итерируемый объект» и «итератор» очень похожи, но имеют разное значение:

1. Итерируемый объект может быть помещен в цикл `for` или в генератор списка. Чтобы объект был итерируемым, он должен реализовать метод `__iter__`. Этот метод должен возвращать итератор.
2. Итератор — это объект, реализующий метод `__next__`.

Во многих случаях итерируемый объект является своим собственным итератором. Например, файловые объекты являются собственными итераторами. Но во многих других случаях, например, строки и списки, итерируемый объект возвращает в качестве итератора отдельный, другой объект.

Таблица 10.1. Что вам нужно знать

Понятие	Что это?	Пример	Чтобы узнать подробнее
iter	Встроенная функция, которая возвращает итератор объекта.	iter (‘abcd’)	

Продолжение таблицы

Понятие	Что это?	Пример	Чтобы узнать подробнее
`next`	Встроенная функция, которая запрашивает следующий объект из итератора.	`next (i)`	
`Stop Iteration`	Исключение, возникающее при завершении цикла.	`raise Stop-Iteration`	
`enumerate`	Помогает нам пронумеровать итерируемые объекты.	`for i, c in enumerate ('ab'): print (f'{i}: {c}')`	
Итерируемый объект	Категория данных в Python.	Итерируемые объекты можно помещать в циклы for или передавать многим функциям.	
`itertools`	Модуль с множеством классов для реализации итерируемого объекта.	`import itertools`	
`range`	Возвращает итерируемую последовательность целых чисел.	`# каждое третье целое число, от 10 # до (не включая) 50 range (10, 50, 3)`	
`os.listdir`	Возвращает список файлов каталога.	`os.listdir ('/etc/')`	

Окончание таблицы

Понятие	Что это?	Пример	Чтобы узнать подробнее
`os.walk`	Итерации по файлам в каталоге.	`os.walk ('/etc/')`	
`yield`	Временно возвращает управление циклу, по желанию возвращая значение.	`yield 5`	
`os.path.join`	Возвращает строку, основанную на компонентах пути.	`os.path.join ('etc', 'passwd')`	
`time.perf_counter`	Возвращает количество секунд, прошедших с момента запуска программы (в виде плавающей величины).	`time.perf_counter ()`	
`zip`	Принимает n итерируемых объектов в качестве аргументов и возвращает итератор кортежей длины n.	`# возвращает [('a', 10), # ('b', 20), ('c', 30)] zip ('abc', [10, 20, 30])`	

Упражнение 46. MyEnumerate

Встроенная функция `enumerate` позволяет нам получить не только элементы последовательности, но и индекс каждого элемента, как в примере:

```
for index, letter in enumerate ('abc'):
print (f'{index}: {letter}')
```

Создайте свой собственный класс `MyEnumerate`, чтобы кто-то мог использовать его вместо `enumerate`. Он должен будет возвращать кортеж при каждой итерации, причем первый элемент кортежа будет индексом (начиная с 0), а второй — текущим элементом из базовой структуры данных. Попытка использовать `MyEnumerate` с неитерируемым аргументом приведет к ошибке.

Обсуждение

В этом упражнении наш класс `MyEnumerate` будет принимать один итерируемый объект. При каждой итерации мы будем получать обратно не один из элементов аргумента, а кортеж из двух элементов.

Это означает, что в конце концов нам понадобится метод `__next__`, который будет возвращать кортеж. Более того, он должен будет отслеживать текущий индекс. Поскольку `__next__`, как и все методы и функции, теряет свою локальную область видимости между вызовами, нам нужно будет хранить текущий индекс в другом месте. Где? В самом объекте, в качестве атрибута.

Следовательно, наш метод `__init__` инициализирует два атрибута: `self.data`, где хранится объект, который мы итерируем, и `self.index`, который будет начинаться с 0 и увеличиваться с каждым вызовом `__next__`. Наша реализация `__iter__` будет стандартной, которую мы видели до сих пор, а именно `return self`.

Наконец, `__next__` проверяет, не превысил ли `self.index` длину `self.data`. Если да, то мы вызываем StopIteration, что приводит к завершению цикла `for`.

Многоклассовые итераторы

До сих пор мы видели, что наш метод `__iter__` должен состоять из строки `return self` и не более. Часто это вполне приемлемый способ. Но вы можете попасть в беду. Например, что произойдет, если я использую наш класс `MyEnumerate` следующим образом?

```
e = MyEnumerate ('abc')

print ('** A **')
for index, one_item in e:
   print (f'{index}: {one_item}')

print ('** B **')
for index, one_item in e:
   print (f'{index}: {one_item}')
```

Мы увидим следующее:

```
** A **
0: a
1: b
2: c
** B **
```

Почему мы не получили во втором раунде `a`, `b` и `c`? Потому что каждый раз мы используем один и тот же объект-итератор. В первый раз `self.index` проходит через 0, 1 и 2, а затем останавливается. Во второй раз `self.index` уже равен 2, что больше `len (self.data)`, и поэтому он немедленно выходит из цикла.

Наше решение return `self` для `__iter__` подходит, если вы хотите именно такого поведения. Но во многих случаях нам нужно что-то более сложное. Самое простое решение — использовать второй класс — класс-помощник, который, если хотите, будет итератором для нашего класса. Многие встроенные классы Python уже делают это, включая строки, списки, кортежи и словари. В этом случае мы реализуем `__iter__` в основном классе, но его задача — вернуть новый экземпляр вспомогательного класса:

```
# в MyEnumerate
def __iter__ (self):
    return MyEnumerateIterator (self.data)
```

Затем мы определяем `MyEnumerateIterator`, новый и отдельный класс, чей `__init__` похож на тот, который мы уже определили для `MyIterator`, а `__next__` берется непосредственно из `MyIterator`.

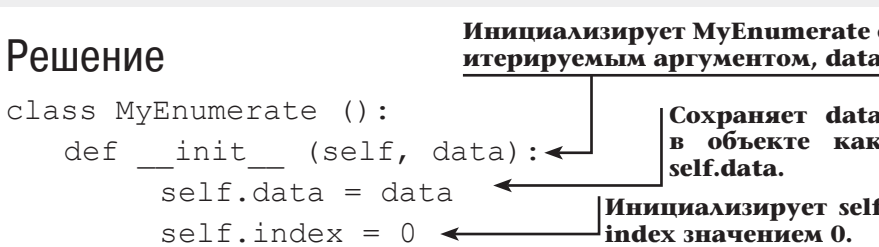
У такой конструкции есть два преимущества:

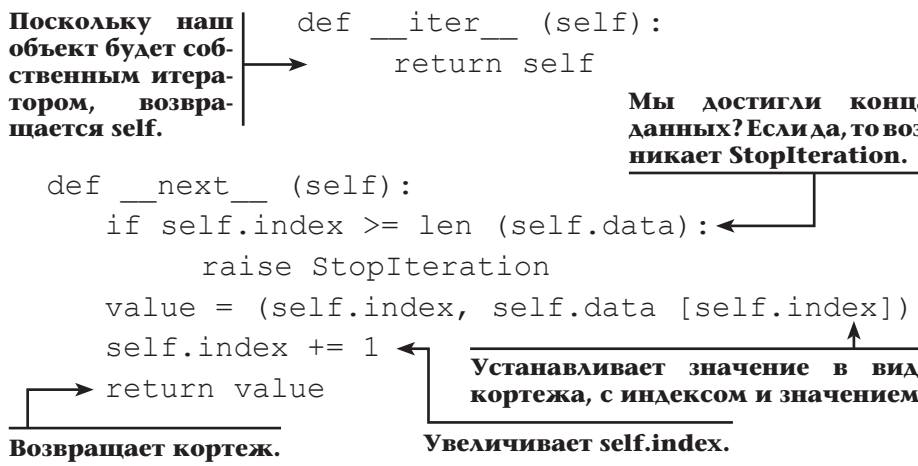
1 Как мы уже видели, отделив итерируемый объект от итератора, мы можем поместить наш итерируемый объект в любое количество циклов `for`, не беспокоясь о потери итераций.

2 Второе преимущество — организационное. Если мы хотим сделать класс итерируемым, то итерации — это небольшая часть функциональности. Таким образом, действительно ли мы хотим загромождать класс `__next__`, а также атрибутами, используемыми только при итерации? Передавая такие проблемы вспомогательному классу итератора, мы отделяем аспекты итерируемого объекта и позволяем каждому классу сосредоточиться на своей роли.

Многие считают, что проблему можно решить более простым способом, просто сбрасывая `self.index` к 0 при каждом вызове `__iter__`. Но это тоже имеет свои недостатки. Это означает, что если мы захотим использовать одну и ту же итерабельную переменную в двух разных циклах одновременно, то они будут мешать друг другу. С вспомогательным классом таких проблем не возникнет.

Решение






```
for index, letter in MyEnumerate ('abc'):
print (f'{index}: {letter}')
```

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr220].

220

Обратите внимание, что Python Tutor иногда выводит сообщение об ошибке, когда возникает ошибка StopIteration.

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr221].

221

После выполнения упражнения

Теперь, когда вы создали простой класс итератора, давайте копнем немного глубже:

Перепишите `MyEnumerate` так, чтобы он использовал вспомогательный класс (`MyEnumerateIterator`), как описано в разделе «Обсуждение». В итоге у `MyEnumerate` будет метод `__iter__`, возвращающий новый экземпляр `MyEnumerateIterator`, а вспомогательный класс будет реализовывать `__next__`.

Он должен работать так же, но будет выдавать результаты, если мы будем итерировать по нему два раза подряд.

1. Встроенный метод `enumerate` принимает второй необязательный аргумент — целое число, представляющее первый индекс, который должен быть использован. (Это особенно удобно при нумерации вещей для нетехнарей, которые считают, что нумерация начинается с 1, а не с 0.)
2. Переопределите `MyEnumerate` как функцию-генератор, а не как класс.

Упражнение 47. Круг

Из рассмотренных нами примеров может показаться, что в случае с итерируемым объектом просто перебираются элементы любых данных, которые он хранит, а затем происходит выход. Но итератор может делать все что угодно и возвращать любые данные, вплоть до того момента, когда он вызовет `StopIteration`. В этом упражнении мы увидим, как это работает.

Определите класс `Circle`, который при определении принимает два аргумента: последовательность и число. Идея заключается в том, что объект будет возвращать элементы определенное количество раз. Если число больше, чем количество элементов, то последовательность повторяется по мере необходимости. Вы должны определить класс так, чтобы он использовал помощника (который я называю `CircleIterator`). Например:

```
c = Circle ('abc', 5)
print (list (c))
```

Печатает a, b, c, a, b.

Обсуждение

Во многих отношениях наш класс `Circle` — это простой итератор, перебирающий все свои значения. Но нам может потребоваться предоставить больше выходов, чем входов, пройдя по кругу к началу один или несколько раз.

Хитрость здесь заключается в использовании оператора `modulus` (%), который возвращает целочисленный остаток от операции деления. `Modulus` часто используется в програм-

мах, чтобы гарантировать, что мы можем выполнить цикл столько раз, сколько нам нужно.

В данном случае мы получаем данные из `self.data`, как обычно. Но элементом будет не `self.data [self.index]`, а `self.data [self.index% len (self.data)]`.

Поскольку `self.index`, скорее всего, окажется больше, чем `len (self.data)`, мы больше не сможем использовать его в качестве теста на то, следует ли нам выдавать StopIteration. Скорее, нам понадобится отдельный атрибут self.max_times, который скажет нам, сколько итераций мы должны выполнить.

Как только мы все это установили, реализация становится довольно простой. Наш класс Circle остается только с `__init__` и `__iter__`, последний из которых возвращает новый экземпляр `CircleIterator`. Обратите внимание, что мы должны передать `self.data` и `self.max_times` в `CircleIterator`, и поэтому нам нужно сохранить их как атрибуты в нашем экземпляре `Circle`.

Затем наш итератор использует логику, которую мы описали в методе `__next__`, чтобы возвращать по одному элементу за раз, пока у нас не будет элементов `self.max_times`.

Другое решение

Оливер Хах и Рейк Торманн, которые читали предыдущее издание этой книги, поделились со мной элегантным решением:

```
class Circle ():

    def __init__ (self, data, max_times):
         self.data = data
         self.max_times = max_times

    def __iter__ (self):
         n = len (self.data)
         return (self.data [x% n] for x in range
         (self.max_times))
```

В этой версии `Circle` мы используем тот факт, что итерируемый класс может возвращать любой итератор, а не только `self` и не только экземпляр вспомогательного класса. В этом случае возвращается выражение–генератор, которое по всем стандартам является итератором.

Выражение–генератор выполняет итерации определенное количество раз, определяемое `self.max_times`, и передает его в `range`. Затем мы можем выполнить итерацию по `range`, возвращая соответствующий элемент `self.data` с каждой итерацией.

Таким образом, мы видим, что существует множество способов ответить на вопрос «Что должен возвращать `__iter__`?».

Пока он возвращает объект итератора, неважно, будет ли это итерируемый `self`, экземпляр вспомогательного класса или генератор.

Решение

```
class CircleIterator ():
    def __init__ (self, data, max_times):
          self.data = data
          self.max_times = max_times self.index = 0

    def __next__ (self):
          if self.index >= self.max_times:
                raise StopIteration
          value = self.data [self.index% len (self.data)]
          self.index += 1
          return value

class Circle ():
    def __init__ (self, data, max_times):
          self.data = data
          self.max_times = max_times
```

```
    def __iter__ (self):
        return CircleIterator (self.data,
                               self.max_times)

c = Circle ('abc', 5)
print (list (c))
```


222

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr222].

Скринкаст решения


223

Посмотрите короткое видео с объяснением решения: [qr223].

После выполнения упражнения

Надеюсь, вы начинаете понимать потенциал итераторов и то, что их можно писать по-разному. Вот несколько дополнительных упражнений, которые заставят вас задуматься о том, какими могут быть эти способы:

1. Вместо того чтобы писать помощника, вы можете определить возможности итерации в классе, а затем наследоваться от него. Реализуйте `Circle` как класс, наследующий от `CircleIterator`, который реализует `__init__` и `__next__`. Конечно, родительский класс должен знать, что возвращать на каждой итерации — добавьте новый атрибут в `Circle`, `self.returns`, список имен атрибутов, которые должны быть возвращены.
2. Реализуйте `Circle` как функцию-генератор, а не как класс.
3. Реализуйте класс `MyRange`, возвращающий итератор, который работает так же, как `range`, по крайней мере, в циклах `for`. (Современные объекты range обладают множеством других возможностей, например, возможностью подписки. Не думайте об этом.) Класс, как и `range`, должен принимать один, два или три целочисленных аргумента.

Упражнение 48. Все строки, все файлы

Файловые объекты, как мы видели, являются итераторами: когда мы помещаем их в цикл `for`, каждый итератор возвращает следующую строку из файла. Но что, если мы хотим прочитать несколько файлов? Было бы неплохо иметь итератор, который проходит через каждый из них.

В этом упражнении я хочу, чтобы вы создали именно такой итератор, используя функцию-генератор. То есть эта функция-генератор будет принимать в качестве аргумента имя каталога. При каждой итерации генератор должен возвращать одну строку из одного файла в этом каталоге. Таким образом, если каталог содержит пять файлов и каждый файл содержит 10 строк, то генератор вернет в общей сложности 50 строк — каждую строку из файла 0, затем каждую строку из файла 1, затем каждую строку из файла 2, пока не переберет все строки из файла 4.

Если вы столкнулись с файлом, который не может быть открыт — потому что это каталог, потому что у вас нет разрешения на чтение из него и так далее — вам следует просто игнорировать проблему.

Обсуждение

Начнем обсуждение с того, что если вы действительно хотите сделать все правильно, то вам следует использовать функцию `os.walk`, которая проходит через каждый файл в каталоге, а затем спускается в его подкаталоги. Но мы проигнорируем это и будем работать над пониманием функции-генератора `all_lines`, которую я создал здесь.

Сначала мы вызываем `os.listdir` в `path`. Будет возвращен список строк. Важно помнить, что `os.listdir` возвращает только имена файлов, а не полный путь к ним. Это означает, что мы не можем просто открыть имя файла: нам нужно объединить путь с именем файла.

Мы могли бы использовать `str.join`, или даже просто `+`, или f-строку. Но есть лучший подход, а именно `os.path.join`, который принимает любое количество параметров

(благодаря `*args`) и затем соединяет их вместе со значением `os.sep`, символа разделения каталогов для текущей операционной системы. Таким образом, нам не нужно думать о том, где мы находимся — в системе Unix или Windows — Python может сделать эту работу за нас.

Что, если возникнет проблема с чтением из файла? Мы отлавливаем это с помощью исключения `OSError`, в котором у нас нет ничего, кроме `pass`. Ключевое слово `pass` означает, что Python не должен ничего делать: оно необходимо из-за структуры синтаксиса Python, который требует, чтобы после двоеточия следовал отступ. Но мы не хотим ничего делать в случае возникновения ошибки, поэтому используем `pass`.

А если проблемы нет? Тогда мы просто возвращаем текущую строку с помощью `yield`. Сразу после `yield` функция переходит в спящий режим, ожидая следующего раза, когда цикл `for` вызовет `next`.

ПРИМЕЧАНИЕ Использование except без указания того, какое исключение вы можете получить, вообще не одобряется, тем более, если вы используете его в паре с pass. Если вы сделаете это в производственном коде, вы, несомненно, столкнетесь с проблемами в какой-то момент, и поскольку вы не отловили конкретные исключения и не записали ошибки в журнал, у вас возникнут проблемы с решением проблемы. Хорошее (хотя и немного устаревшее) введение в исключения Python и то, как их следует использовать, смотрите на сайте: qr224.

Решение

Получает список файлов в path.

Использует os.path.join для создания полного имени файла, который мы будем открывать.

```
import os
   def all_lines (path):
        for filename in os.listdir (path):
             full_filename = os.path.join (path,
                                           filename)
```

```
try:
    for line in open (full_filename):
        yield line

except OSError:
    pass
```

Открывает и итерирует каждую строку в full_filename.

Возвращает строку с использованием yield, необходимого в итераторах.

Игнорирует проблемы, связанные с файлами.

Сайт Python Tutor не работает с файлами, поэтому ссылки на него нет. Но вы можете просмотреть все строки из всех файлов в каталоге /etc/ на вашем компьютере с помощью команды

```
for one_line in all_lines ('/etc/'):
print (one_line)
```

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr225].

После выполнения упражнения

Если что–то, что вы хотите сделать в качестве итератора, не соответствует существующему классу, но может быть определено как функция, то использование функции–генератора, вероятно, будет хорошим решением. Функции–генераторы особенно полезны при получении потенциально больших объемов данных, их разбивке на части и возврате выходных данных в темпе, который не перегружает систему.

Вот некоторые другие проблемы, которые можно решить с помощью функций–генераторов:

1. Модифицируйте `all_lines` таким образом, чтобы она возвращала не строку при каждой итерации, а кортеж. Кортеж должен содержать четыре элемента: имя файла, текущий номер файла (из всех возвращенных `os.listdir`), номер строки в текущем файле и текущую строку.

2. Текущая версия функции `all_lines` возвращает все строки из первого файла, затем все строки из второго файла и так далее. Измените функцию так, чтобы она возвращала первую строку из каждого файла, затем вторую строку из каждого файла, пока не будут возвращены все строки из всех файлов. Когда вы закончите печатать строки из более коротких файлов, игнорируйте эти файлы, продолжая выводить строки из более длинных файлов.
3. Измените `all_lines` так, чтобы она принимала два аргумента — имя каталога и строку. Возвращаются только те строки, которые содержат строку (т.е. для которых можно написать `s in line`). Если вы умеете работать с регулярными выражениями и модулем `re` в Python, то можете воспользоваться ими.

ПРИМЕЧАНИЕ В функциях-генераторах нам не нужно явно вызывать `StopIteration`. Это происходит автоматически, когда генератор достигает конца функции. Действительно, вызывать `StopIteration` внутри генератора не следует. Если вы хотите преждевременно выйти из функции, лучше использовать оператор `return`. Использование `return` со значением (например, return 5) в функции-генераторе не является ошибкой, но значение будет проигнорировано. Таким образом, в функции-генераторе `yield` означает, что вы хотите продолжить работу генератора и вернуть значение для текущей итерации, а `return` означает, что вы хотите полностью завершить работу.

Упражнение 49. Сколько времени прошло

Иногда смысл итератора не в том, чтобы изменить существующие данные, а в том, чтобы предоставить данные в дополнение к тем, которые мы получили ранее. Более того, генератор не обязательно предоставляет все свои значения сразу: можно запро-

сить тогда, когда нам понадобится дополнительное значение. Действительно, тот факт, что генераторы сохраняют свое состояние во время сна между итерациями, означает, что они могут зависать, «слоняться» без дела в ожидании, пока не понадобится следующее значение.

В этом упражнении напишите функцию–генератор, аргумент которой должен быть итерируемым. При каждой итерации генератор будет возвращать двухэлементный кортеж. Первым элементом кортежа будет целое число, показывающее, сколько секунд прошло с момента предыдущей итерации. Вторым элементом кортежа будет следующий элемент из переданного аргумента.

Обратите внимание, что время должно относиться к предыдущей итерации, а не к моменту, когда генератор был впервые создан или вызван. Следовательно, значение тайминга в первой итерации будет равно 0.

Вы можете использовать `time.perf_counter`, который возвращает количество секунд с момента запуска программы. Можно использовать `time.time`, но `perf_counter` считается более надежным для таких целей.

Обсуждение

Функция генератора решения принимает один фрагмент данных и выполняет итерацию по нему. Однако для каждого элемента она возвращает двухэлементный кортеж, в котором первый элемент — это время, прошедшее с момента выполнения предыдущей итерации.

Чтобы это работало, мы должны всегда знать, когда была выполнена предыдущая итерация. Таким образом, мы всегда вычисляем и устанавливаем `last_time`, перед тем как выдать текущие значения `delta` и `item`.

При этом нам нужно иметь значение для `delta` в первый раз, когда мы получаем результат. Оно должно быть равно `0`. Чтобы обойти это, мы установим `last_time` в `None` в верхней части функции. Затем, на каждой итерации, мы вычисляем `delta` как разницу между `current_time` и `last_time` или `current_time`.

Если значение `last_time` равно `None`, то мы получим значение `current_time`. Это должно произойти только один раз — после первой итерации значение `last_time` никогда не будет равно нулю.

Обычно вызов функции несколько раз означает, что локальные переменные сбрасываются при каждом вызове. Однако функция–генератор работает по–другому: она вызывается только один раз и поэтому имеет один стековый кадр. Это означает, что локальные переменные, включая параметры, сохраняют свои значения при каждом вызове. Таким образом, мы можем установить такие значения, как `last_time`, и использовать их в последующих итерациях.

Решение

```
import time

def elapsed_since (data):
    last_time = None
    for item in data:
        current_time = time.perf_counter ()
        delta = current_time - (last_time
        or current_time)
        last_time = time.perf_counter ()

        yield (delta, item)

for t in elapsed_since ('abcd'):
    print (t)
    time.sleep (2)
```

Инициализирует last_time значением None.

Получает соответствующее значение времени.

Возвращает двухэлементный кортеж.

Вычисляет дельту между прошлым временем и настоящим.


226

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr226].


227

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr227].

После выполнения упражнения

В этом упражнении мы увидели, как мы можем комбинировать предоставленные пользователем данные с дополнительной информацией из системы. Вот еще несколько упражнений для практики написания таких функций генератора:

1. Существующая функция `elapsed_since` сообщала, сколько времени прошло между итерациями. Теперь напишите функцию–генератор, которая принимает два аргумента — часть данных и минимальное количество времени, которое должно пройти между итерациями. Если следующий элемент запрашивается через протокол итератора (т.е. `next`), а время, прошедшее с момента предыдущей итерации, превышает установленный пользователем минимум, то значение возвращается. Если нет, то генератор использует `time.sleep` для ожидания, пока не истечет соответствующее количество времени.
2. Напишите функцию генератора, `file_usage_timing`, которая принимает в качестве аргумента одно имя каталога. С каждой итерацией мы получаем кортеж, содержащий не только текущее имя файла, но и три отчета о последнем использовании файла: время доступа (`atime`), время модификации (`mtime`) и время создания (`ctime`). Подсказка: все они доступны через функцию `os.stat`.
3. Напишите функцию–генератор, которая принимает два элемента: итерируемый объект и функцию. На каждой итерации вызывается функция для текущего элемента. Если результат `True`, то элемент возвращается в исходном виде. В противном случае, пока функция не вернет значение `True`, будет проверяться следующий элемент. Альтернатива: реализовать как обычную функцию, которая возвращает выражение–генератор.

Упражнение 50. MyChain

Как вы можете себе представить, паттерны итераторов имеют тенденцию повторяться. По этой причине Python поставляется с модулем `itertools`, который упрощает создание множества типов итераторов. Классы в `itertools` были оптимизированы и отлажены во многих проектах и часто включают функции, которые вы, возможно, не рассматривали. Определенно стоит помнить об этом модуле для своих собственных проектов.

Один из моих любимых объектов в `itertools` называется `chain`. Он принимает любое количество итерируемых объектов в качестве аргументов, а затем возвращает каждый из их элементов по одному, как если бы все они были частью одного итерируемого объекта, например:

```
from itertools import chain

for one_item in chain ('abc', [1,2,3], {'a':1, 'b':2}):
    print (one_item)
```

Этот код напечатает:

```
a
b
c
1
2
3
a
b
```

Финальные `'a'` и `'b'` исходят из переданного нами словаря, поскольку в результате итерирования по словарю его ключи возвращаются.

Хотя `itertools.chain` удобен и продуман, его не так уж сложно реализовать. В данном упражнении именно это вы

и должны сделать: реализовать функцию–генератор с именем `mychain`, которая принимает любое количество аргументов, каждый из которых является итерируемым. С каждой итерацией он должен возвращать следующий элемент из текущего итерируемого объекта или первый элемент из последующего итерируемого объекта — если только вы не находитесь в конце, и в этом случае он должен завершиться.

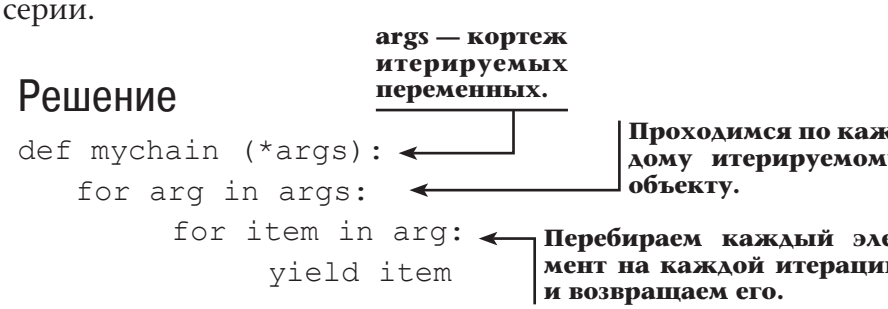
Обсуждение

Это правда, что вы могли бы создать для этого класс Python, реализующий протокол итератора, с `__iter__` и `__call__`. Но, как видите, код намного проще, легче для понимания и элегантнее, когда мы используем функцию–генератор.

Наша функция принимает `*args` в качестве параметра, что означает, что `args` будет кортежем при выполнении нашей функции. Поскольку это кортеж, мы можем перебирать его элементы, сколько бы их ни было.

Мы указали, что каждый аргумент, передаваемый в `mychain`, должен быть итерируемым, что означает, что мы также должны иметь возможность итерировать эти аргументы. Затем во внутреннем цикле `for` мы просто возвращаем значение текущей строки. Это возвращает вызывающему объекту текущее значение, но также сохраняет текущее место в функции–генераторе. Таким образом, в следующий раз, когда мы вызовем `__next__` для нашего объекта итерации, мы получим следующий элемент в серии.

Решение




```
def mychain (*args):
    for arg in args:
        for item in arg:
            yield item

for one_item in mychain ('abc', [1,2,3], {'a':1, 'b':2}):
    print (one_item)
```

Вы можете ознакомиться с одной из версий этого кода в Python Tutor [qr228].

Скринкаст решения

Посмотрите короткое видео с объяснением решения: [qr229].

После выполнения упражнения

В этом упражнении мы разобрали некоторые встроенные функции, чтобы повторно реализовать их самостоятельно. В частности, мы увидели, как можно создать собственную версию `itertools.chain` в качестве функции-генератора. Вот некоторые дополнительные задачи, которые вы можете решить с помощью функций-генераторов:

1. Встроенная функция `zip` возвращает итератор, который, учитывая итерируемые аргументы, возвращает кортежи, взятые из элементов этих аргументов. Первая итерация вернет кортеж из аргумента с индексом 0, вторая итерация вернет кортеж из аргумента с индексом 1 и т. д., остановившись, когда закончится самый короткий из аргументов. Таким образом, `zip ('abc', [10, 20, 30])` вернет итератор, эквивалентный `[('a', 10), ('b', 20), ('c', 30)]`. Напишите функцию-генератор, которая переопределяет `zip` таким образом.
2. Повторно реализуйте функцию `all_lines` из упражнения 49, используя `mychain`.
3. В разделе «После выполнения упражнения» для упражнения 48 вы реализовали класс `MyRange`, который имитирует встроенный класс `range`. Теперь сделайте то же самое, но используя выражение-генератор.

Подводя итоги

В этой главе мы рассмотрели протокол итератора и то, как мы можем реализовать и использовать его различными способами. Хотя нам нравится говорить, что в Python есть только один

способ делать что–либо, вы можете наблюдать, что существует как минимум три разных способа создания итератора:

1. Добавьте в класс соответствующие методы.
2. Напишите функцию–генератор.
3. Используйте выражение–генератор.

Протокол итератора распространен и удобен в Python. К настоящему моменту ситуация напоминает проблему курицы и яйца: стоит ли добавлять протокол итератора к вашим объектам, потому что большое количество программ будет ожидать, что объекты будут его поддерживать? Или программы используют протокол итератора, потому что его поддерживает очень много программ? Ответ может быть не совсем понятен, но выводы ясны. Если у вас есть коллекция данных или что–то, что можно интерпретировать как коллекцию, то стоит добавить в класс соответствующие методы. И если вы не создаете новый класс, вы все равно можете воспользоваться преимуществами итерируемых объектов с функциями–генараторами и выражениями–генераторами.

Я надеюсь, что упражнения из этой главы помогут вам понять:

1. Как добавить протокол итератора в ваш класс.
2. Как добавить протокол итератора в класс через вспомогательный класс итератора.
3. Как писать функции–генераторы, которые фильтруют, изменяют и добавляют к итераторам, которые вы иным образом создали или использовали
4. Как использовать более эффективные выражения–генераторы, чем генераторы списков.

Заключение

Поздравляем! Вы дошли до конца книги, а это (если вы не заглядываете наперед) означает, что вы выполнили большое количество упражнений по Python. В результате ваши навыки Python улучшились.

Во–первых, теперь вы лучше знакомы с синтаксисом и методами Python. Подобно изучению иностранного языка: у вас

раньше мог быть определенный словарный запас и знание некоторых грамматических структур, но теперь вы можете говорить свободнее. Вам не нужно долго думать, чтобы решить, какое слово выбрать. Вы не будете использовать конструкции, которые работают, но считаются непитоновскими.

Во-вторых, вы видели достаточное количество различных проблем и использовали Python для их решения, чтобы знать, что делать при возникновении новых трудностей. Вы будете знать, какие вопросы задавать, как разбивать проблемы на составляющие и какие конструкции Python лучше всего соответствуют вашим решениям. Вы сможете сравнить преимущества различных вариантов, а затем внедрить лучшие из них в свой код.

В-третьих, вы теперь ближе знакомы с тем, как работает Python, и со словарным запасом, который используется языком для решения задач. Это означает, что вам легче будет понять документацию Python, а также экосистему сообщества, состоящую из блогов, учебников, статей и видео. Описания будут иметь больше смысла, а примеры будут более эффективными.

Таким образом, более свободное владение Python поможет вам писать более качественный код за меньшее время, сохраняя его читабельность и питоничность. Это также означает, что вы сможете учиться дальше на своем пути разработчика.

Я желаю вам успехов в вашей карьере разработчика Python и надеюсь, что вы продолжите находить все новые способы, чтобы и дальше практиковать Python.

Содержание

Заметки

Заметки

Заметки

Заметки

Заметки

16+

Серия «Программирование для всех»
Издание для досуга
Демалысқа арналған баспа

Реувен Л.
PYTHON-ИНТЕНСИВ: 50 БЫСТРЫХ УПРАЖНЕНИЙ

Заведующая редакций *О. Максимова*
Ответственный редактор *А. Семенова*
Менеджер проекта *В. Живина*
Дизайнер обложки *А. Шмулий*
Технический редактор *Н. Чернышева*
Верстальщик *О. Недосекина*

Подписано в печать 01.12.2023
Формат 70x100/16 Усл. Печ. л. 28.6
Печать офсетная. Гарнитура SonetSerif. Бумага офсетная.
Тираж экз. Заказ №

Произведено в Российской Федерации
Изготовлено в 2024 г.
Оригинал–макет подготовлен редакцией «Времена», импринт «Альфа»
Изготовитель: ООО «Издательство АСТ»
129085, Российская Федерация, г. Москва, Звездный бульвар, д. 21, стр. 1,
комн. 705, пом. I, этаж 7
Наш электронный адрес: WWW.AST.RU
E–mail: ask@ast.ru
Общероссийский классификатор продукции ОК–034–2014 (КПЕС 2008);
58.11.1 – книги, брошюры печатные

«Баспа Аста» деген ООО
129085, г. Мәскеу, Жулдызды гүлзар, д. 21, 1 құрылым, 705 бөлме, пом. 1, 7-қабат
Біздің электрондық мекенжаймыз : www.ast.ru
E-mail: ask@ast.ru
Интернет-магазин: www.book24.kz Интернет-дүкен: www.book24.kz
Импортер в Республику Казахстан и Представитель по приему претензий
в Республике Казахстан — ТОО РДЦ Алматы, г. Алматы.
Қазақстан Республикасына импорттаушы және Қазақстан Республикасында наразылықтарды
қабылдау бойынша өкіл -«РДЦ-Алматы» ЖШС, Алматы
к.,Домбровский көш., 3«а», Б литері офис 1. Тел.: 8(727) 2 51 59 90,91 ,
факс: 8 (727) 251 59 92 ішкі 107; E-mail: RDC-Almaty@eksmo.kz , www.book24.kz
Тауар белгісі: «АСТ» Өндірілген жылы: 2024
Өнімнін жарамдылық; мерзімі шектелмеген.
Сертификация қарастырылмаған

Издадим вашу книгу! Рукопись присылать на litagent@ast.ru
Мы в социальных сетях. Присоединяйтесь!
vk.com/ast_nonfiction

Автор, **Лернер Реувен** преподает Python и data sc1ence компаниям по всему миру.

«Python-интенсив: 50 **быстрых упражнений»** - пособие по программированию для продолжающих, тех, кто владеет теоретической базой языка Python.

Книга отлично подойдет всем, кто хочет применить свои знания на практике. Перед каждым упражнением вы найдете **теоретическую выжимку, необходимую для успешного выполнения заданий.** Также пособие содержит ссылки на разбор упражнений и полезные материалы.

С помощью этой книги вы освоите такие базовые понятия языка Python, как:

- **основные структуры данных;**
- **функции;**
- **генераторы;**
- **объектно ориентированное программирование;**
- **итераторы.**

рекомендовано Библиотекой программиста

MANNING

ISBN 978-5-17-155721-8

9 785171 557218 >