O'REILLY®

2-е издание

Дополнено для Python 3

Основы Python

НАУЧИТЕСЬ ДУМАТЬ КАК ПРОГРАММИСТ

Аллен Б. Дауни

Эту книгу хорошо дополняют:

Python для детей
Джейсон Бриггс

Программирование на Python
Кэрол Вордерман

Аналитическая культура
Карл Андерсон

Основы глубокого обучения
Нихиль Будума, Николас Локашо

Как работают технологии
Dorling Kindersley (DK)

Allen B. Downey

Think Python

SECOND EDITION

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

Аллен Б. Дауни

Основы Python

НАУЧИТЕСЬ ДУМАТЬ КАК ПРОГРАММИСТ

Перевод с английского
Сергея Черникова

Москва
«Манн, Иванов и Фербер»
2021

УДК 004.43
ББК 32.973
Д 21

Научный редактор Андрей Родионов

Издано с разрешения O'Reilly Media, Inc.

На русском языке публикуется впервые

Дауни Аллен

Д 21 Основы Python. Научитесь думать как программист / Аллен Б. Дауни ; пер. с англ. С. Черникова ; [науч. ред. А. Родионов]. — Москва : Манн, Иванов и Фербер, 2021. — 304 с.

ISBN 978-5-00146-798-4

Это практическое руководство последовательно раскрывает основы программирования на языке Python. Вы будете продвигаться от самых простых тем к сложным и получите полное представление об одном из самых популярных языков программирования.

А еще вы поймете, как думают программисты, и сможете применять этот подход к решению даже повседневных задач.

УДК 004.43
ББК 32.973

ISBN 978-5-00146-798-4

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

СТРАННАЯ ИСТОРИЯ ЭТОЙ КНИГИ

В январе 1999 года я готовился преподавать вводный курс программирования на языке Java. Я уже делал это трижды и теперь находился в замешательстве. Слишком много студентов не тянули курс, и даже среди преуспевающих общий уровень оставлял желать лучшего.

Как я обратил внимание, одной из причин были книги. Огромные, с излишним количеством ненужных подробностей о языке Java и явным недостатком простых уроков по программированию. И все студенты попадали в одну и ту же ловушку: бодрый старт, плавный прогресс, а вблизи пятой главы ловушка захлопывалась. Студенты получали слишком много нового материала и слишком быстро, и остаток семестра приходилось по кусочкам собирать знания воедино.

За две недели до начала занятий я решил написать собственную книгу. Вот что я хотел:

— сделать ее лаконичной. Лучше студенты прочтут десять страниц, чем не прочтут пятьдесят;
— быть осторожным с терминами. Как можно меньше использовать профессиональный жаргон и давать определение каждому термину при первом вхождении;
— увеличивать сложность постепенно. Чтобы избежать «ловушек», я взял самые трудные темы и разбил их на серии маленьких шагов;
— я сфокусировался на практике, а не на теории программирования. Я описал необходимый минимум знаний о языке Java и опустил все остальное.

Требовалось привлекательное название, поэтому по воле случая моя книга была названа «Думай как компьютерный ученый».

Первая версия книги получилась далеко не шедевром, но стала эффективной. Студенты читали и, главное, понимали, так что я мог уделять

время сложным темам, интересному материалу и (самое главное) позволить студентам практиковаться.

Я выпустил книгу под лицензией GNU Free Documentation License, которая позволяет читателям бесплатно копировать, изменять и распространять материалы из книги.

То, что произошло дальше, — удивительная история. Джефф Элкнер, учитель одной из школ в Вирджинии, взял мою книгу и адаптировал ее под язык программирования Python. Он прислал мне копию своего детища, и я приобрел необычный опыт: я изучал Python в процессе чтения собственной книги. Я опубликовал первое издание адаптации под Python в 2001 году в издательстве Green Tea Press.

В 2003 году я начал преподавать в колледже имени Франклина У. Олина, и темой первых моих уроков стал язык Python. Контраст с Java был поразительным. Студенты меньше страдали, больше учились, работали над более интересными проектами и в целом получали гораздо больше удовольствия.

С тех пор я продолжал развивать книгу, исправляя ошибки, улучшая некоторые примеры и добавляя материал, в частности упражнения.

Результатом стала эта книга, теперь с менее пафосным названием — «Основы Python». Перечислю некоторые из изменений.

— Я добавил раздел об отладке в конец каждой главы. В этих разделах представлены общие методы поиска и предотвращения ошибок, а также предупреждения о подводных камнях Python.
— Я добавил дополнительные упражнения, начиная с коротких задач для закрепления материала и заканчивая несколькими крупными проектами. Большинство упражнений содержат ссылки на мои варианты решений.
— Я добавил серию случаев из практики — объемные примеры с упражнениями, решениями и обсуждением.
— Я подробнее поговорил о способе разработки программ и основных шаблонов проектирования.
— Я добавил приложения об отладке и анализе алгоритмов.

Второе издание книги содержит следующие изменения.

— Текст книги и код всех примеров были обновлены до версии Python 3.
— Я добавил несколько новых разделов и разместил больше информации в интернете, чтобы помочь новичкам запустить Python в браузере, поэтому устанавливать Python не придется, пока вы сами не захотите.

— Для модуля turtle из главы 4 я переключился с собственного графического «черепашьего» пакета Swampy на стандартный модуль Python, turtle, простой в установке и более мощный.
— Я добавил новую главу под названием «Синтаксический сахар». В ней описаны дополнительные возможности Python, не строго необходимые, но иногда очень полезные.

Надеюсь, вам понравится работать с этой книгой, и она поможет вам научиться программировать и мыслить как программист, хотя бы немного.

Аллен Б. Дауни
Инженерно-технический колледж
имени Франклина У. Олина

УСЛОВНЫЕ ОБОЗНАЧЕНИЯ

В этой книге используются следующие обозначения.

Курсивный шрифт
Им оформлены новые термины, имена и расширения файлов.

Полужирный шрифт
Указывает на термины, определенные в словаре терминов, а также полужирным шрифтом выделены URL-адреса и адреса электронной почты.

`Моноширинный шрифт`
Используется для оформления листингов программ, а также для выделения в основном тексте фрагментов кода программ, таких как имена переменных или функций, базы данных, типы данных, переменных сред, инструкции и зарезервированные слова.

`Полужирный моноширинный шрифт`
Обозначает команды или текст, который должен быть введен пользователем.

`Курсивный моноширинный шрифт`
Так оформлен текст, который читатель должен заменить собственными значениями или значениями, определенными контекстом.

ИСПОЛЬЗОВАНИЕ ПРИМЕРОВ КОДА

Дополнительный материал (примеры кода, упражнения и так далее) доступны для загрузки по адресу **http://thinkpython2.com/code**.

БЛАГОДАРНОСТИ

Большое спасибо Джеффу Элкнеру, который адаптировал мою книгу про язык Java под Python, запустил этот проект и познакомил меня с ныне любимым моим языком.

Также спасибо Крису Мейерсу за то, что написал несколько разделов в книгу «Основы Python».

Спасибо организации Free Software Foundation за разработку лицензии GNU Free Documentation License, которая сделала возможной мое сотрудничество с Джеффом и Крисом, и Creative Commons за лицензию, которую я использую сейчас.

Спасибо редакторам из издательства Lulu, которые работали над книгой How to Think Like a Computer Scientist «Основы Python».

Спасибо редакторам из издательства O'Reilly Media, которые трудились над книгой Think Python «Основы Python».

Спасибо всем студентам, которые работали с предыдущими изданиями этой книги, и всем читателям (перечисленным ниже), которые прислали исправления и предложения.

СПИСОК УЧАСТНИКОВ ПРОЕКТА

За последние несколько лет свыше сотни проницательных и вдумчивых читателей прислали свои предложения и исправления. Их вклад и энтузиазм очень способствовали развитию проекта.

Предложения или исправления отправляйте на адрес **feedback@thinkpython.com**. Если ваши отзывы помогут, я добавлю вас в список участников (если, конечно, вы не попросите об обратном).

Если вы напишете хотя бы фрагмент предложения, в котором обнаружили ошибку, вы упростите мне поиск. Номера страниц и разделов тоже можно указывать, но это не так удобно. Спасибо!

— Ллойд Хью Аллен прислал исправление в раздел 8.4.
— Ивон Булянн прислал исправление семантической ошибки в главе 5.

— Фред Бреммер представил исправление в раздел 2.1.
— Джона Коэн написал Perl-скрипт для преобразования исходного кода LaTeX этой книги в красивый HTML-код.
— Майкл Конлон предложил исправить грамматическую ошибку в главе 2 и помог со стилистикой главы 1, а также инициировал обсуждение технических аспектов работы интерпретаторов.
— Бенуа Жирар исправил смешную ошибку в разделе 5.6.
— Кортни Глисон и Кэтрин Смит написали скрипт *horsebet.py*, который использовался в качестве примера в предыдущем издании книги. Их программа теперь опубликована на сайте книги.
— Ли Харр представил больше исправлений, чем я могу тут перечислить, посему его можно указать в качестве одного из главных научных редакторов.
— Джеймс Кейлин — очень внимательный студент. Он предоставил многочисленные исправления.
— Дэвид Кершоу исправил нерабочую функцию `catTwice()` в разделе 3.10.
— Эдди Лам прислал многочисленные исправления в главы 1, 2 и 3. Он также исправил код `make`-файла, чтобы тот создавал индекс при первом запуске, и помог нам настроить систему управления версиями.
— Ман Ён Ли прислал правку кода в примере в разделе 2.4.
— Дэвид Майо отметил, что слово «неосознанно» в главе 1 необходимо изменить на «подсознательно».
— Крис Макалун прислал несколько исправлений в разделы 3.9 и 3.10.
— Мэтью Дж. Моэлтер, опытный редактор, прислал многочисленные исправления и улучшения книги.
— Саймон Дикон Монтфорд сообщил об отсутствии определения функции и нескольких опечатках в главе 3. Он также нашел ошибки в функции инкремента в главе 13.
— Джон Оузтс исправил определение «возвращаемого значения» в главе 3.
— Кевин Паркс прислал ценные комментарии и маркетинговое предложение о продвижении книги.
— Дэвид Пул сообщил об опечатке в словаре терминов главы 1, а также написал добрые слова о проекте.
— Майкл Шмитт отправил исправление к главе о файлах и исключениях.
— Робин Шоу указал на ошибку в разделе 13.1, где функция `printTime()` использовалась в примере до определения.

— Пол Слай обнаружил ошибки в главе 7 и Perl-скрипте Джоны Коэна, который генерирует HTML из LaTeX.
— Крейг Т. Снидал протестировал книгу в Университете Дрю. Он внес несколько ценных предложений и исправлений.
— Ян Томас и его ученики используют книгу на курсах по программированию. Они первыми проверили главы второй половины книги и внесли многочисленные исправления и предложения.
— Кит Верхейден прислал изменения для главы 3.
— Питер Уинстанли сообщил нам о назойливой ошибке в главе 3.
— Крис Вробель исправил код в главе, посвященной вводу-выводу и исключениям.
— Моше Задка внес неоценимый вклад. Он не только написал черновик главы о словарях, но и взял на себя руководство проектом на ранних этапах.
— Кристоф Цвершке прислал несколько исправлений и педагогических предложений и объяснил разницу между немецкими словами *gleich* и *selbe*.
— Джеймс Майер заметил множество орфографических и типографских ошибок, в том числе две в этом списке.
— Хайден Макафи обнаружил потенциально противоречивое несоответствие между двумя примерами.
— Анхель Арнал из международной команды переводчиков, работающих над испанской версией книги, нашел несколько ошибок в английской версии.
— Таухидул Хок и Лекс Бережный создали иллюстрации к главе 1 и помогли усовершенствовать многие другие рисунки.
— Д-р Микеле Альзетта обнаружил ошибку в главе 8 и прислал несколько интересных педагогических комментариев и предложений для алгоритмов Фибоначчи и карточной игры Old Maid.
— Энди Митчелл обнаружил опечатку в главе 1 и проблему с примером из главы 2.
— Калин Харви предложил разъяснение к главе 7 и нашел несколько опечаток.
— Кристофер П. Смит нашел несколько опечаток и помог нам обновить книгу под Python версии 2.2.
— Дэвид Хатчинс нашел опечатку в предисловии.

— Грегор Лингл преподает Python в средней школе в Вене, Австрия. Он работал над переводом книги на немецкий язык и обнаружил пару ошибок в главе 5.
— Джули Питерс нашла опечатку в предисловии.
— Флорин Оприна предложил улучшение метода `makeTime()`, исправление метода `printTime()` и нашел забавную опечатку.
— Д. Ж. Верб предложил разъяснение к главе 3.
— Кен обнаружил несколько ошибок в главах 8, 9 и 11.
— Иво Вевер нашел опечатку в главе 5 и предложил разъяснение к главе 3.
— Кертис Янко предложил разъяснение к главе 2.
— Бен Логан прислал информацию о многих опечатках и проблемах при переводе книги в формат HTML.
— Джейсон Армстронг увидел, что пропущено слово в главе 2.
— Луи Кордье заметил в главе 16 место, где код не соответствовал тексту.
— Брайан Кейн предложил несколько пояснений к главам 2 и 3.
— Роб Блэк прислал множество исправлений, включая некоторые изменения для Python 2.2.
— Жан-Филипп Рей, сотрудник компании Ecole Centrale Paris, прислал несколько исправлений, включая некоторые обновления для Python 2.2, и другие значимые улучшения.
— Джейсон Мэдер из Университета Джорджа Вашингтона внес ряд полезных предложений и исправлений.
— Ян Гундтофте-Брюн напомнил нам, какой артикль нужно использовать со словом ошибка (*an* error).
— Абель Давид и Алексис Динно напомнили нам, что множественное число от matrix — это matrices, а не matrixes. Эта ошибка была в книге годами, и два читателя с одинаковыми инициалами сообщили об этом в один и тот же день. Удивительно.
— Чарльз Тейер призвал нас избавиться от точек с запятой, которые мы ставили в конце некоторых строк кода, и помог разобраться с путаницей в использовании терминов «аргумент» и «параметр».
— Роджер Сперберг указал на извращенную логику в главе 3.
— Сэм Булл указал на запутанный абзац в главе 2.
— Эндрю Ченг указал на два случая использования переменных до их определения.

— С. Кори Капел обнаружил пропущенное слово и опечатку в главе 4.
— Алессандра помогла разобраться с «черепашкой».
— Вим Шампань нашел смысловую ошибку в примере со словарями.
— Дуглас Райт указал на проблему с целочисленным делением в функции `arc()`.
— Джаред Спиндор нашел ненужный текст.
— Лин Пэйхэн прислал несколько очень полезных предложений.
— Рэй Хагтведт прислал информацию о двух ошибках и одной не совсем ошибке.
— Торстен Хюбш указал на несоответствие в модуле Swampy.
— Инга Петухова исправила код примера в главе 14.
— Арне Бабенхаузерхайде прислал несколько полезных исправлений.
— Марк Э. Касида здорово помог избавиться от повторов.
— Скотт Тайлер доработал некоторые упущения. А еще прислал кучу исправлений.
— Гордон Шепард прислал несколько исправлений, каждое в отдельном письме.
— Эндрю Тернер заметил ошибку в главе 8.
— Адам Хобарт исправил проблему с целочисленным делением в функции `arc()`.
— Дэрил Хаммонд и Сара Циммерман указали, что я слишком рано начал объяснять модуль math.pi. А еще Сара заметила опечатку.
— Джордж Сасс обнаружил ошибку в разделе «Отладка».
— Брайан Бингхэм предложил упражнение 11.5.
— Лиа Энгельберт-Фентон обнаружила, что я использовал слово `tuple` в качестве имени переменной вопреки моему собственному совету. А также нашла кучу опечаток и случаев обращения ранее определения.
— Джо Фанке заметил опечатку.
— Чао Чао Чен обнаружил несоответствие в примере с Фибоначчи.
— Джефф Пейн разъяснил разницу между словами *space* и *spam*.
— Либо Пенти нашел опечатку.
— Грегг Линд и Эбигейл Хейтхофф предложили упражнение 14.3.
— Макс Хайлперин прислал ряд исправлений и предложений. Макс — один из авторов экстраординарной книги Concrete Abstractions (Course Technology, 1998), которую вы можете прочитать, когда закроете эту.
— Чотипат Порнавалай обнаружил ошибку в сообщении об ошибке.

— Станислав Антол прислал список очень полезных предложений.
— Эрик Пашман прислал ряд исправлений для глав 4–11.
— Мигель Азеведо нашел несколько опечаток.
— Цзяньхуа Лю прислал длиннющий список исправлений.
— Ник Кинг нашел пропущенное слово.
— Мартин Зютер прислал длинный список предложений.
— Адам Циммерман обнаружил несоответствие в моем экземпляре «экземпляра» и несколько других ошибок.
— Ратнакар Тивари добавил сноску, объясняющую, что такое вырожденные треугольники.
— Анураг Гоэль предложил другое решение для функции `is_abecedarian()` и некоторые дополнительные исправления. И он знает, как правильно пишется имя Джейн Остин!
— Келли Кратцер заметил одну опечатку.
— Марк Гриффитс указал на запутанный пример в главе 3.
— Ройдан Онги обнаружил ошибку в моей реализации метода Ньютона.
— Патрик Воловец помог мне с проблемой в HTML-версии.
— Марк Чонофски рассказал мне о новом ключевом слове в Python 3.
— Рассел Коулман помог мне с геометрией.
— Вэй Хуан заметил несколько опечаток.
— Карен Барбер обнаружила самую древнюю опечатку в книге.
— Нам Нгуен нашел опечатку и указал, что я использовал шаблон Decorator, но не упомянул об этом.
— Стефан Морен прислал несколько исправлений и предложений.
— Пол Ступ исправил опечатку в функции `uses_only()`.
— Эрик Броннер указал на путаницу в обсуждении порядка операций.
— Александрос Гезерлис своими приложениями установил новый стандарт их количества и качества. Мы очень признательны!
— Серый Томас знает, где право, а где лево.
— Джованни Эскобар Соса прислал длинный список исправлений и предложений.
— Аликс Этьен исправила один из URL-адресов.
— Куанг Он нашел опечатку.
— Даниэль Нилсон исправил ошибку в порядке операций.
— Уилл Макгиннис отметил, что функция `polyline()` была определена по-разному в двух местах.
— Сваруп Саху заметил пропущенную точку с запятой.

— Фрэнк Хекер указал на неясности в упражнении и некоторые неработающие ссылки.
— Анимеш Б помог мне сделать запутанный пример более понятным.
— Мартин Касперсен обнаружил две ошибки округления.
— Грегор Ульм прислал несколько исправлений и предложений.
— Димитриос Циригкас предложил мне уточнить упражнение.
— Карлос Тафур прислал список исправлений и предложений.
— Мартин Нордслеттен обнаружил ошибку в решении.
— Ларс О.Д. Кристенсен нашел неработающую ссылку.
— Виктор Симеоне нашел опечатку.
— Свен Хоукстер отметил, что имя переменной `input` совпадает с именем встроенной функции.
— Вьет Ле нашел опечатку.
— Стивен Грегори указал на проблему с функцией `cmp()` в Python 3.
— Мэтью Шульц дал мне знать о неработающей ссылке.
— Локеш Кумар Макани сообщил о неработающих ссылках и некоторых изменениях в сообщениях об ошибках.
— Ишвар Бхат исправил мое утверждение о последней теореме Ферма.
— Брайан Макги предложил разъяснение.
— Андреа Занелла перевела книгу на итальянский язык и попутно внесла ряд исправлений.
— Огромное спасибо Мелиссе Льюис и Лучано Рамальо за прекрасные комментарии и предложения по поводу второго издания.
— Спасибо Гарри Персивалю из компании PythonAnywhere за его помощь, позволившую людям запускать Python в браузере.
— Ксавье Ван Обель внес несколько полезных исправлений во второе издание.

ГЛАВА 1
ПУТЬ РАЗРАБОТКИ

Цель этой книги — научить вас мыслить как настоящий программист. Этот способ сочетает в себе особенности мышления математика, инженера и ученого. Как математики компьютерные специалисты используют формальные языки для выражения идей (в частности, вычислений). Как инженеры они что-то проектируют, собирают отдельные компоненты в системы и оценивают компромиссы между альтернативами. Как ученые они наблюдают за поведением сложных систем, формируют гипотезы и тестируют прогнозы.

Единственный самый важный навык для разработчика — умение **находить решение задачи**. Для этого он должен сформулировать задачу, подойти творчески к поиску решения, а затем точно и ясно его реализовать. Как видите, обучение программированию — это прекрасная возможность попрактиковаться в решении задач. Вот почему эта глава называется «Путь разработки».

С одной стороны, вы будете учиться программировать, что само по себе полезный навык. С другой — вы будете использовать программирование как средство для достижения цели. По мере того как мы будем продвигаться дальше, вы поймете, о чем я.

ЧТО ТАКОЕ ПРОГРАММА?

Программа — это последовательность инструкций, в которых указано, как выполнять вычисления. Вычисления могут быть математическими, такими как решение системы уравнений или поиск корней многочлена, но это также могут быть символические вычисления, например поиск и замена текста в документе, или что-то графическое, например обработка изображения или воспроизведение видеоролика.

Детали реализации выглядят по-разному на разных языках, но несколько основных инструкций универсальны для любого языка:

— *ввод данных (input):*
  Получение данных с клавиатуры, из файла, по сети или с другого устройства.
— *вывод данных (output):*
  Отображение данных на экране, сохранение их в файл, отправка по сети и так далее.
— *математические операции (math):*
  Выполнение основных математических операций, таких как сложение и умножение.
— *условное выполнение (conditional execution):*
  Проверка определенных условий и выполнение соответствующего кода.
— *повторение (repetition):*
  Выполнение некоторого действия несколько раз, часто с некоторыми изменениями.

Верьте или нет, но это все, что нужно знать. Каждая программа, которую вы когда-либо использовали, независимо от ее сложности, состоит из таких инструкций. Таким образом, вы можете представить программирование как процесс разбиения большой и сложной задачи на всё более мелкие подзадачи, пока подзадачи не станут достаточно простыми, чтобы их можно было сформулировать с помощью одной из этих инструкций.

ЗАПУСК PYTHON

Работа с Python начинается с установки Python и связанного программного обеспечения на компьютер. Если вы знакомы с вашей операционной системой и особенно если вы знакомы с интерфейсом командной строки, у вас не должно возникнуть проблем. Но новичкам сложновато изучать системное администрирование и программирование одновременно.

Чтобы облегчить задачу, я рекомендую запустить Python в браузере. Позже, когда вы освоитесь, я предложу вам установить Python на компьютер.

Существует несколько веб-сайтов для запуска Python. Если у вас уже есть любимый, можете смело использовать его. В противном случае я рекомендую

PythonAnywhere. Подробные инструкции по началу работы приведены на странице **http://tinyurl.com/thinkpython2e**.

Существует две версии языка, Python 2 и Python 3. Они очень похожи, поэтому, если вы изучите одну из них, то легко сможете использовать и другую. На самом деле есть только несколько отличий, с которыми вы столкнетесь как новичок. Эта книга написана под Python 3, но я добавил несколько примечаний, касающихся и Python 2.

Интерпретатор (interpreter) Python — это программа, которая анализирует, обрабатывает и выполняет код Python. В зависимости от установленной операционной системы вы можете запустить интерпретатор, щелкнув мышью по значку или набрав слово python в командной строке. В случае успешного запуска вы должны увидеть примерно следующий результат:

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Первые три строки содержат информацию об интерпретаторе и операционной системе, в которой он запущен, поэтому у вас сведения могут отличаться. Но вы должны проверить, что номер версии (в этом примере — `3.4.0`) начинается с 3, что указывает на то, что вы используете Python 3. Если номер версии начинается с 2, вы работаете (как вы уже догадались) с Python 2.

Последняя строка представляет собой **приглашение (prompt)**, которое указывает, что интерпретатор ожидает ввод команды от пользователя. Если вы наберете в строке 1 + 1 и нажмете клавишу **Enter**, интерпретатор отобразит результат:

```
>>> 1 + 1
2
```

Теперь вы готовы начать учиться. С этого момента я предполагаю, что вы знаете, как использовать интерпретатор Python.

ПЕРВАЯ ПРОГРАММА

Традиционно первая программа, которую пишут на любом новом языке программирования называется Hello, World! Все, что она делает, это отображает слова Hello, World! (то есть «Привет, мир!»). На языке Python программа выглядит так:

```
>>> print('Привет, мир!')
```

Это пример **инструкции печати**, хотя на самом деле она ничего не печатает на бумаге. Она отображает результат на экране. В этом случае результатом будут следующие слова:

```
Привет, мир!
```

Кавычки в программе отмечают начало и конец отображаемого текста; они не видны в выводе.

Скобки `()` указывают, что `print` — это функция. Мы рассмотрим функции в главе 3.

В Python 2 инструкция печати немного отличается; это не функция, поэтому скобки не используются.

```
>>> print 'Привет, мир!'
```

Это различие будет иметь смысл далее, но для начала достаточно просто об этом помнить.

АРИФМЕТИЧЕСКИЕ ОПЕРАТОРЫ

После простенькой программы Hello, World наш следующий шаг — арифметика. В языке Python есть **операторы (operators)**, которые выглядят как специальные символы, представляющие вычисления, такие как сложение и умножение.

Операторы `+`, `-` и `*` выполняют сложение, вычитание и умножение, соответственно, как показано в следующих примерах:

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

Оператор `/` выполняет деление:

```
>>> 84 / 2
42.0
```

Вы можете спросить, почему результат `42.0`, а не `42`. Я объясню это в следующем разделе.

Наконец, оператор ** выполняет возведение в степень, то есть умножает число на это же число указанное количество раз:

```
>>> 6 ** 2 + 6
42
```

В некоторых других языках для возведения в степень используется символ ^, но в Python это побитовый оператор, называемый XOR (исключающее «ИЛИ»). Если вы не знакомы с побитовыми операторами, результат вас удивит:

```
>>> 6 ^ 2
4
```

Я не буду рассматривать побитовые операторы в этой книге, но вы можете прочитать о них по адресу **http://wiki.python.org/moin/BitwiseOperators**.

ЗНАЧЕНИЯ И ТИПЫ

Значение (value) — это одно из основных понятий, с которым работает программа, например буква или цифра. Мы уже видели некоторые значения: `2`, `42.0` и `'Привет, мир!'`

Эти значения принадлежат разному **типу**: `2` — это **целое число (int)**, `42.0` — **число с плавающей точкой*** **(floating-point number)**, а `'Привет, мир!'` — **строка (str)**.

Если вы не знаете, какого типа указанное значение, интерпретатор может подсказать вам:

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Привет, мир!')
<class 'str'>
```

Слово `class` здесь используется для обозначения типа.

Неудивительно, что целые числа принадлежат к целочисленному типу `int`, строки относятся к `str`, а числа с плавающей точкой — это `float`.

---

* Числа с плавающей точкой — вещественные числа, такие как 1.45, 0.00453 или –3.789. В России также может использоваться термин «плавающая запятая», но так как в большинстве языков программирования для отделения дробной части от целой используется именно точка, далее будет использовано международное обозначение. *Прим. перев.*

А как насчет таких значений, как `'2'` и `'42.0'`? Они выглядят как числа, но указаны в кавычках, как строки:

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

Интерпретатор определяет их как строки.

При вводе большого целого числа иногда хочется использовать запятые как разделители между группами цифр, например так: `1,000,000`. В этом случае Python не опознает *целое число*:

```
>>> 1,000,000
(1, 0, 0)
```

Это совсем не то, что мы ожидали! Python интерпретирует `1,000,000` как последовательность целых чисел через запятую. Мы узнаем больше об этом виде последовательности позже.

ФОРМАЛЬНЫЕ И ЕСТЕСТВЕННЫЕ ЯЗЫКИ

На **естественных языках** говорят люди; это, например, английский, испанский и французский языки. Они не были спроектированы людьми (хотя люди и пытаются соблюдать в них некий порядок); они развивались естественно.

Формальные языки разрабатываются людьми для определенных целей. Например, математические символы представляют собой формальный язык, который особенно хорош для обозначения отношений между числами и символами. Химики используют формальный язык для представления химической структуры молекул. И самое важное:

> **Языки программирования — это формальные языки, предназначенные для выражения вычислений.**

Формальные языки, как правило, имеют строгие **синтаксические** правила, которым подчиняется структура кода. Например, в математике утверждение 3 + 3 = 6 имеет правильный синтаксис, а 3 + = 3$6 — нет. В химии H_2O — синтаксически правильная формула, а *2Zz* — нет.

Синтаксические правила бывают двух видов: относящиеся к **токенам** и к структуре. Токены — основные элементы языка, такие как слова, числа

и химические элементы. Одна из проблем с 3 + = 3$6 заключается в том, что это недопустимый в математике токен (по крайней мере, по моим сведениям). Аналогично, *2Zz* недопустимо, так как нет элемента с сокращением *Zz*.

Второй тип правил синтаксиса относится к способу объединения токенов. Операция 3 + = 3 недопустима, поскольку хотя символы + и = и являются допустимыми токенами, их нельзя использовать один сразу за другим. Точно так же в химической формуле нижний индекс указывается после имени элемента, а не перед.

Это x0рошо структуриров@нное пр3длож3ние с недопустимым т*кенами. Это токены предложение допустимые содержит, но недопустимую структуру все.

Когда вы читаете предложение на естественном языке или утверждение на формальном языке, вы должны понять структуру (хотя на естественном языке вы делаете это подсознательно). Этот процесс называется **синтаксическим разбором**, или **парсингом**.

Хотя у формальных и естественных языков много общих черт — токены, структура и синтаксис, — есть некоторые различия:

— *двусмысленность:*
  Естественные языки полны неоднозначности, с которой люди справляются с помощью контекстных подсказок и другой информации. Формальные языки спроектированы быть максимально однозначными, что означает, что любое утверждение имеет ровно одно значение независимо от контекста.
— *избыточность:*
  Чтобы компенсировать неоднозначность и уменьшить недопонимание, в естественных языках много избыточности. В результате они часто многословны. Формальные языки менее избыточны и более лаконичны.
— *буквальность:*
  Естественные языки полны идиом и метафор. Если я скажу: «Белая ворона», то, вероятно, я имею в виду не белую ворону или другую птицу, а «человека не такого, как все». Утверждения в формальных языках означают именно то, что они означают.

Поскольку все мы растем в среде естественного языка, иногда трудно приспособиться к формальным языкам. Различие между формальным

и естественным языком похоже на разницу между поэзией и прозой, более того:

— *Поэзия:*
  Слова используются как ради их звучания, так и ради их значения, и стихотворение в целом создает определенный эффект или вызывает эмоциональный отклик. Неоднозначность не только распространена, но и часто намеренна.
— *Проза:*
  Буквальное значение слов наиболее важно, а структура вносит больший смысл. Проза легче поддается анализу, чем поэзия, но все же часто неоднозначна.
— *Программы:*
  Значение компьютерной программы однозначно и буквально, и его можно полностью понять, проанализировав токены и структуру.

Формальные языки более насыщенные, чем естественные, поэтому их чтение занимает больше времени. Кроме того, структура важна, поэтому не всегда лучше читать сверху вниз, слева направо. Вместо этого научитесь анализировать программу в своей голове, выявляя токены и интерпретируя структуру. Наконец, детали имеют значение. Небольшие ошибки в написании и пунктуации, некритичные в естественных языках, важны в формальном языке.

ОТЛАДКА

Программисты делают ошибки. По интересной случайности* ошибки программирования называются багами (в пер. с англ. — жуками), а процесс их отслеживания называется **отладкой** (debugging).

Программирование и особенно отладка иногда вызывает сильные эмоции. Если вы долго боретесь с трудной ошибкой, то можете начать злиться или впадать в уныние.

Зачастую люди реагируют на компьютеры, как если бы те тоже были людьми. Когда они работают хорошо, мы считаем их коллегами, а когда они упрямы или грубы, мы реагируем на них соответствующим образом

* По самой распространенной версии, в 1946 году разработка компьютера Mark II была приостановлена из-за сбоя, которые был вызван попаданием мотылька между контактами (от англ. bug — жук, насекомое). *Прим. ред.*

(см. книгу Ривза и Насса, The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places).

Вот что поможет подготовиться к этим эмоциям. Один из подходов состоит в том, чтобы воспринимать компьютер как работника — со своими сильными сторонами, такими как скорость и точность, и с недостатками, такими как отсутствие сопереживания и неспособность понимать общую картину.

Ваша работа — стать хорошим управленцем: найти способы использовать сильные и слабые стороны. И найти способы использовать свои эмоции для решения проблемы, не позволяя им снижать эффективность работы.

Учиться отладке нелегко, но это ценный навык, полезный не только для программирования. В конце каждой главы есть раздел с моими предложениями по отладке. Надеюсь, что они помогут!

СЛОВАРЬ ТЕРМИНОВ

Решение задачи (problem solving):

Процесс формулирования задачи, поиска решения и его реализации.

Высокоуровневый язык (high-level language):

Язык программирования, разработанный, чтобы программистам было легко и удобно его использовать. Python — это высокоуровневый язык.

Низкоуровневый язык (low-level language):

Язык программирования, приближенный к инструкциям, которые понимает компьютер, но сложный для человека; также называется «машинным языком», или «языком ассемблера».

Портативность (portability):

Способность программы функционировать на компьютерах разного типа.

Интерпретатор (interpreter):

Программа, которая анализирует, обрабатывает, считывает и выполняет исходный код.

Приглашение (prompt):

Символы, отображаемые интерпретатором для обозначения ожидания ввода пользователем.

Программа (program):

Набор инструкций, определяющих вычисления.

Инструкция печати (print statement):
Инструкция, которая заставляет интерпретатор Python отображать значение на экране.

Оператор (operator):
Специальный символ (или символы), позволяющий выполнить простые вычисления, такие как сложение, умножение или соединение строк.

Значение (value):
Одна из основных единиц данных, таких как число или строка, которыми манипулирует программа.

Тип (type):
Тип — это категория значения. Основные типы переменных: целые числа (int), числа с плавающей точкой (float), строки (str).

Целое число (int):
Тип, который представляет целые числа.

Число с плавающей точкой (floating-point):
Тип, представляющий числа с дробной частью.

Строка (str):
Тип, который представляет собой последовательности символов.

Естественный язык (natural language):
Язык, на котором говорят люди и который развивался естественным путем.

Формальный язык (formal language):
Язык, который люди разработали для определенных целей, таких как представление математических концепций или компьютерных программ; все языки программирования — формальные языки.

Токен (token):
Один из основных элементов синтаксической структуры программы, аналог «слова» в естественном языке.

Синтаксис (syntax):
Правила, определяющие структуру исходного кода программы.

Парсинг (parsing):
Разбор кода программы и анализ синтаксической структуры.

Ошибка (bug):
Ошибка в программе.

Отладка (debugging):

Поиск и исправление ошибок.

УПРАЖНЕНИЯ

Упражнение 1.1

Рекомендуется прочитать эту книгу, сидя за компьютером, чтобы вы могли попробовать выполнить примеры самостоятельно по мере необходимости.

Каждый раз, когда экспериментируете с новой функцией, вы должны попытаться сделать ошибку. Например, что произойдет с программой "Hello, world!", если вы пропустите одну из кавычек? А если обе? А если вы напишете `print` с ошибкой?

Такие эксперименты помогут вам не только запомнить прочитанное, но и поспособствуют эффективному программированию, так как познакомят вас с основными сообщениями об ошибках.

Лучше ошибаться сейчас и нарочно, чем позже и случайно.

1. Что произойдет с инструкцией печати, если вы пропустите одну из скобок? Обе?
2. Тот же вопрос, но если вы пропустите одну из кавычек? Обе?
3. Вы можете использовать знак минус, чтобы указать отрицательное число, например `-2`. Что произойдет, если вы укажете знак плюс перед числом? К чему приведет код `2++2`?
4. В математике нули в начале — это абсолютно нормально, например, так: `02`. Что произойдет, если вы попробуете это сделать в Python?
5. Что произойдет, если указать два значения без оператора между ними?

Упражнение 1.2

Запустите интерпретатор Python и используйте его в качестве калькулятора.

1. Сколько секунд в 42 минутах и 42 секундах?
2. Сколько миль в 10 километрах? Подсказка: одна миля равна 1,61 км.
3. Если вы пробежали 10 километров за 42 минуты 42 секунды, каков ваш средний темп бега (время, затраченное на преодоление мили, в минутах и секундах)? Какова ваша средняя скорость в милях в час?

ГЛАВА 2
ПЕРЕМЕННЫЕ, ВЫРАЖЕНИЯ И ИНСТРУКЦИИ

Одна из самых мощных функций языка программирования — возможность манипулировать **переменными (variables)**. Переменная — это имя, которое ссылается на значение.

ИНСТРУКЦИИ ПРИСВАИВАНИЯ

Инструкция присваивания (assignment statement) создает новую переменную и присваивает ей значение:

```
>>> message = 'Сообщение на совершенно другую тему'
>>> n = 17
>>> pi = 3.141592653589793
```

В этом примере три инструкции присваивания. Первая присваивает строку новой переменной с именем `message`; вторая задает переменной `n` целочисленное значение 17; третья присваивает приближенное значение π переменной `pi`.

Обычный способ представления переменных на бумаге — это написать имя со стрелкой, указывающей на его значение. Такое обозначение называется **диаграммой состояний**, потому что оно отражает, в каком состоянии находится каждая из переменных (воспринимайте ее как душевное состояние переменной). На рис. 2.1 показано представление переменных из данного примера.

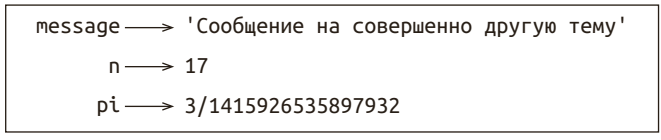


Рис. 2.1. Диаграмма состояний

ИМЕНА ПЕРЕМЕННЫХ

Хорошей практикой будет привычка выбирать имена переменных, которые явно указывают на свое предназначение.

Имена переменных могут иметь любую длину. Они могут содержать как латинские буквы, так и цифры, но не могут начинаться с цифры. Можно использовать прописные буквы, но среди программистов для имен переменных принято использовать только строчные буквы.

Символ подчеркивания «_» также может указываться в имени. Он часто используется в именах, состоящих из словосочетаний, таких как `your_name` или `airspeed_of_unladen_swallow`.

Если вы присвоите переменной недопустимое имя, то при выполнении программы увидите синтаксическую ошибку:

```
>>> 76trombones = 'большой концерт'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Технология бродильных производств'
SyntaxError: invalid syntax
```

`76trombones` — недопустимое имя, потому что начинается с цифры. Имя `more@` недопустимо, поскольку содержит недопустимый символ @. Но что не так с `class`?

Оказывается, `class` — одно из **зарезервированных слов (keywords)*** в языке Python. Интерпретатор использует такие слова для распознавания структуры программы, и их нельзя использовать в качестве имен переменных.

В языке Python 3 используют следующие зарезервированные слова:

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

* Зарезервированные слова иногда называют ключевыми. *Прим. ред.*

Вам не нужно запоминать этот список. В большинстве сред разработки зарезервированные слова отображаются другим цветом; если вы попытаетесь использовать их в качестве имен переменных, вы сразу это заметите.

ВЫРАЖЕНИЯ И ИНСТРУКЦИИ

Выражение (expression) — это комбинация значений, переменных и операторов. Само по себе значение или переменная также считаются выражениями, так что все приведенные ниже выражения допустимы:

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

Когда вы вводите выражение в командной строке, интерпретатор **вычисляет** его и возвращает значение. В этом примере переменная `n` имеет значение `17`, а выражение `n + 25` возвращает значение `42`.

Инструкция (statement) — это блок кода, который что-то делает, например создает переменную или выводит значение на экран.

```
>>> n = 17
>>> print(n)
```

Первая строка — это инструкция присваивания, которая присваивает значение переменной `n`. Вторая строка — это инструкция печати, которая выводит значение `n` на экран.

ВЫПОЛНЕНИЕ СКРИПТОВ

Когда вы вводите инструкцию, интерпретатор выполняет ее и возвращает результат.

До сих пор мы запускали Python в **интерактивном режиме**, что означает, что вы взаимодействовали непосредственно с интерпретатором. Интерактивный режим — прекрасный способ научиться программировать, но, если вы разрабатываете настоящую программу, он будет неудобным.

Альтернатива заключается в сохранении исходного кода в файл, называемый **скриптом (script)**, и последующем запуске интерпретатора для его

выполнения. По соглашению, файлы скриптов Python имеют имена, которые заканчиваются на *.py*.

Если вы знаете, как создать и запустить файл скрипта на своем компьютере, мы можем продолжить. В противном случае я вновь рекомендую использовать PythonAnywhere. Я опубликовал инструкции по работе с файлами скриптов на странице **http://tinyurl.com/thinkpython2e**.

Поскольку среда разработки Python поддерживает оба режима, вы можете проверить части кода в интерактивном режиме, прежде чем помещать их в скрипт. Но между интерактивным режимом и запуском скриптов есть различия, которые могут сбивать с толку.

Например, если вы используете Python в качестве калькулятора, вы можете ввести:

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

На первой строке переменной `miles` присваивается значение, но без наглядного эффекта. На второй строке интерпретатор вычисляет выражение и возвращает его результат. Оказывается, марафонская дистанция составляет около 42 километров.

Но если вы сохраните этот же код в файле скрипта и запустите его, вы вообще ничего не увидите. При выполнении скриптов выражения сами по себе не имеют наглядного эффекта. Python фактически вычисляет выражение, но не отображает значение, если вы не попросите это сделать явно:

```
miles = 26.2
print(miles * 1.61)
```

Такой результат может сначала сбить с толку.

Скрипт обычно содержит последовательность инструкций. Если их указано более одной, результаты отображаются по одному при выполнении инструкций.

Например, скрипт:

```
print(1)
x = 2
print(x)
```

выводит следующее:

```
1
2
```

Инструкция присваивания не возвращает значение.

Чтобы убедиться, что вы всё поняли, введите следующие инструкции в интерпретаторе Python и посмотрите, что произойдет:

```
5
x = 5
x + 1
```

Теперь поместите те же инструкции в скрипт и запустите его. Каков результат? Теперь измените скрипт так, чтобы результат каждого выражения отображался на экране.

ПРИОРИТЕТ ОПЕРАЦИЙ

Когда выражение содержит более одной операции, порядок вычисления зависит от **приоритета операций (order of operations)**. С математическими операторами Python следует математическому соглашению. Аббревиатура PEMDAS является простым способом запоминать правила.

- **Скобки (Parentheses)** имеют наивысший приоритет и могут использоваться для принудительного вычисления выражения в нужном вам порядке. Поскольку выражения в скобках вычисляются первыми, `2 * (3 - 1)` равно 4, а `(1 + 1) ** (5 - 2)` равно 8. Вы также можете использовать круглые скобки, чтобы упростить чтение выражения, например так: `(minute * 100) / 60`, даже если результат в этом случае не изменится.
- **Возведение в степень (exponentiation)** имеет следующий наивысший приоритет, поэтому `1 + 2 ** 3` равно 9, а не 27, а `2 * 3 ** 2` результирует в 18, а не 36.
- **Умножение (multiplication)** и **деление (division)** имеют более высокий приоритет, чем **сложение (addition)** и **вычитание (subtraction)**. Таким образом, `2 * 3 - 1` равно 5, а не 4, а `6 + 4 / 2` равно 8, а не 5.
- Операции с одинаковым приоритетом вычисляются слева направо (кроме возведения в степень). Таким образом, в выражении `degrees / 2 * pi` сначала происходит деление, а результат умножается на значение переменной `pi`. Чтобы разделить `degrees` на 2π, вы можете использовать скобки или написать выражение `degrees / 2 / pi`.

Я не стремлюсь запомнить приоритет операций. Если я не могу с ходу определить приоритеты, глядя на выражение, я добавляю скобки, чтобы сделать их очевидными.

ОПЕРАЦИИ СО СТРОКАМИ

Как правило, вы не можете выполнять математические операции над строками, даже если строки выглядят как числа, поэтому следующие действия недопустимы:

```
'2' - '1' 'шоколадка' / 'легко' 'третий' * 'это чары'
```

Но есть два исключения: + и *.

Операция + выполняет **конкатенацию строк**, что означает, что строки соединяются, связываются между собой. Например:

```
>>> first = 'пятко'
>>> second = 'дер'
>>> first + second
пяткодер
```

Операция * также работает со строками; она выполняет повторение. Например, результат выражения `'Спам' * 3` будет равен `СпамСпамСпам`. Если одно из значений строковое, другое должно быть целым числом.

Такое использование операций + и * имеет смысл по аналогии со сложением и умножением. Точно так же, как `4 * 3` эквивалентно `4 + 4 + 4`, ожидается, что `'Спам' * 3` будет аналогично выражению `'Спам' + 'Спам' + 'Спам'`, и это так. С другой стороны, есть существенное отличие конкатенации и повторения строк от целочисленного сложения и умножения. Сможете определить свойство, которое есть у сложения, но нет у конкатенации?

КОММЕНТАРИИ

По мере того как код программы усложняется, его становится все труднее читать. Формальные языки — насыщенные, и часто с одного взгляда трудно понять, что код делает или почему.

По этой причине рекомендуется сопровождать свои программы заметками и объяснять на естественном языке, что делает программа. Такие заметки называются **комментариями**, и начинаются они с символа #:

```
# Вычисляем процент прошедшего часа
percentage = (minute * 100) / 60
```

В этом примере комментарий расположен в отдельной строке. Вы также можете поместить комментарии в конце строки с выражением:

```
percentage = (minute * 100) / 60 # Процент часа
```

Весь текст от символа # до конца строки игнорируется — он не влияет на выполнение программы.

Комментарии наиболее полезны, когда документируют неочевидные инструкции кода. Разумно предположить, что разработчик сам сможет выяснить, что делает код, поэтому полезнее объяснить почему.

Этот комментарий к коду лишний и бесполезный:

```
v = 5 # Присваиваем значение 5 переменной v
```

А этот комментарий содержит полезную информацию, которой нет в коде:

```
v = 5 # Задаем скорость в метрах в секунду.
```

Умело подобранные имена переменных уменьшают потребность в комментариях, но длинные имена могут затруднить чтение сложных выражений, поэтому надо искать компромисс.

ОТЛАДКА

В программе могут возникать три вида ошибок: синтаксические ошибки, ошибки в процессе выполнения и семантические ошибки. Полезно различать их, чтобы находить и исправлять быстрее.

1. *Синтаксическая ошибка (syntax error).*
   Синтаксис определяет структуру и правила написания выражений и инструкций. Например, круглые скобки должны использоваться в паре, поэтому выражение `(1 + 2)` допустимо, а выражение `8)` приведет к **синтаксической ошибке**.

   Если в вашей программе обнаружится синтаксическая ошибка, Python отобразит сообщение об ошибке и завершит выполнение скрипта. В первые недели карьеры программиста вы можете потратить много времени на поиск синтаксических ошибок. С опытом вы будете ошибаться реже, а находить ошибки — быстрее.

2. *Ошибка в процессе выполнения (runtime error).*
   Второй тип ошибки — это **ошибка в процессе выполнения**, которая называется так, потому что проявляется только после запуска программы. Еще эти ошибки называются **исключениями**, поскольку обычно указывают на то, что произошло что-то исключительное (и плохое).

Ошибки в процессе выполнения редко встречаются в простейших программах, которые вы увидите в первых главах этой книги, поэтому пройдет некоторое время, прежде чем вы столкнетесь с ними.

3. *Семантическая ошибка (semantic error).*
   Третий тип ошибок — семантический, что означает «связанный со смыслом». Если в вашей программе есть семантическая ошибка, программа будет работать без вывода сообщений об ошибках, но неправильно. Она будет делать не то, что вы хотели, а то, что сказали.

   Находить семантические ошибки нелегко, потому что для этого нужно проанализировать исходный код программы и попытаться понять, что она делает.

СЛОВАРЬ ТЕРМИНОВ

Переменная (variable):
Имя, которое ссылается на значение.

Присваивание (assignment):
Назначение значения переменной.

Диаграмма состояний (state diagram):
Графическое представление набора переменных и значений, к которым они относятся.

Зарезервированное или ключевое слово (keyword):
Слово, которое используется в процессе анализа (парсинга) исходного кода программы интерпретатором. Такие слова, как `if`, `def` и `while`, нельзя использовать при определении имен переменных.

Операнд (operand):
Одно из значений, которыми оперирует оператор*.

Выражение (expression):
Комбинация значений, переменных и операторов, которая возвращает результат.

Вычисление (evaluate):
Упрощение выражения и преобразование операндов для получения конечного результата.

---

* В выражении 3+3 оператором является +, а операндов два: первая и последняя тройка.

Инструкция (statement):

Часть кода, представляющая команду или действие. Вы уже знакомы с инструкциями присвоения и печати.

Выполнение (execute):

Выполнение инструкции интерпретатором.

Интерактивный режим (interactive mode):

Способ использования интерпретатора Python путем ввода кода в командной строке.

Выполнение скрипта (script mode):

Способ использования интерпретатора Python для чтения исходного кода из файла и его выполнения.

Скрипт (script):

Исходный код программы, записанный в файл.

Приоритет операций (order of operations):

Правила, регулирующие порядок, в котором вычисляются выражения, включающие несколько операторов и операндов.

Конкатенация (concatenate):

Объединение двух операндов.

Комментарий (comment):

Информация, поясняющая исходный код. Не влияет на ход выполнения программы.

Синтаксическая ошибка (syntax error):

Ошибка в программе, которая делает невозможным синтаксический анализ и, следовательно, не позволяет интерпретировать программу.

Исключение (exception):

Ошибка, обнаруженная во время выполнения программы.

Семантика (semantics):

Смысловое значение, предназначение программы.

Семантическая ошибка (semantic error):

Ошибка, которая заставляет программу работать не так, как предполагал разработчик.

УПРАЖНЕНИЯ

Упражнение 2.1

Повторю мой совет из предыдущей главы: всякий раз, когда вы изучаете новые возможности языка Python, вы должны опробовать ее в интерактивном режиме и специально сделать ошибки, чтобы увидеть, что идет не так.

— Вы видели, что выражение `n = 42` допустимо. А как насчет `42 = n`?
— Как насчет выражения `x = y = 1`?
— В некоторых языках каждая инструкция заканчивается точкой с запятой – `;`. Что произойдет, если вы введете точку с запятой в конце инструкции Python?
— А если в конце инструкции вы введете точку?
— В математике вы можете перемножить *x* и *y* следующим образом: *xy*. Что произойдет, если вы попробуете выполнить такое выражение в Python?

Упражнение 2.2

Попрактикуйтесь использовать интерпретатор Python в качестве калькулятора.

1. Объем сферы с радиусом *r* составляет $\frac{4}{3} \times \pi r^3$. Каков объем сферы с радиусом 5?
2. Предположим, что книга стоит 249 рублей 50 копеек, при этом книжный магазин предоставляет скидку в 40%. Стоимость доставки составляет 100 рублей за первый экземпляр и 49 рублей 50 копеек за каждый дополнительный. В какую сумму обойдется закупка 60 экземпляров?
3. Если я вышел из дома в 6:52 утра и пробежал 1 км в легком темпе (1 км за 8 мин. 15 сек.), потом 3 км в среднем темпе (1 км за 7 мин. 12 сек.) и 1 км в легком темпе снова, то во сколько я вернусь домой позавтракать?

ГЛАВА 3
ФУНКЦИИ

В контексте программирования **функция (function)** — это блок программы (или подпрограмма), к которому можно обращаться из любого другого места программы. Чтобы создать функцию, нужно определить ее имя, принимаемые аргументы и последовательность инструкций, которые она должна выполнять. Впоследствии вы можете «вызвать» функцию по ее имени.

ВЫЗОВ ФУНКЦИИ

Мы уже видели один пример **вызова функции (function call)**:

```
>>> type(42)
<class 'int'>
```

Имя функции — `type`. Выражение в скобках называется **аргументом (argument)** функции. Если необходимо передать несколько аргументов — они указываются через запятую. Результат выполнения функции `type()` — вывод типа аргумента.

Обычно говорят, что функция «принимает» аргументы и «возвращает» значение или результат.

Python поддерживает функции, которые преобразуют значения из одного типа в другой. Функция `int()` принимает любое значение и, если может, преобразует его в целое число или «жалуется» на те, которые не может преобразовать:

```
>>> int('32')
32
>>> int('Привет')
ValueError: invalid literal for int(): 'Привет'
```

Функция `int()` может преобразовывать значения с плавающей точкой в целые числа, но не округляет, а просто отбрасывает дробную часть:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

Функция `float()` преобразует целые числа и строки в числа с плавающей точкой:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Наконец, `str` преобразует принимаемый аргумент в строку:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

МАТЕМАТИЧЕСКИЕ ФУНКЦИИ

В Python входит математический модуль, который предоставляет доступ к большинству известных математических функций. **Модуль (module)** — это файл, содержащий подборку связанных функций.

Прежде чем мы сможем использовать функции в модуле, мы должны импортировать его с помощью **инструкции импорта (import statement)**:

```
>>> import math
```

Эта инструкция создает **объект модуля (module object)** `math`. Если вы вызовете объект модуля в интерактивном режиме, вы получите некоторую информацию о нем:

```
>>> math
<module 'math' (built-in)>
```

Объект модуля содержит функции, переменные и константы, определенные в модуле. Чтобы получить доступ к одной из функций, вы должны указать имя модуля и имя функции, разделенные точкой. Такой формат называется **точечной нотацией (dot notation)**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
```

```
>>> radians = 0.7
>>> height = math.sin(radians)
```

В первом примере используется функция `math.log10()` для вычисления отношения сигнал/шум в децибелах (при условии, что определены переменные `signal_power` и `noise_power`). Математический модуль также предоставляет функцию `log()`, которая вычисляет логарифм по основанию *e*.

Второй пример находит синус для значения переменной `radians`. Имя переменной служит подсказкой, что `sin()` и другие тригонометрические функции (`cos()`, `tan()` и так далее) принимают аргументы в радианах. Чтобы преобразовать градусы в радианы, разделите значение на 180 и умножьте на π:

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

Выражение `math.pi` возвращает константу `pi` из математического модуля. Ее значение является приближенным к числу π с точностью до 15 знаков.

Если вы знаете тригонометрию, вы можете проверить предыдущий результат, сравнив его с квадратным корнем из 2, деленным на 2:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

КОМПОЗИЦИИ

До сих пор мы рассматривали элементы программы — переменные, выражения и инструкции — изолированно, не говоря о том, как их объединять.

Одна из наиболее полезных возможностей языков программирования заключается в их способности брать небольшие строительные блоки и составлять из них **композицию**. Например, аргумент функции может быть выражением любого типа, включая арифметические операции:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

И даже вызовом других функций:

```
x = math.exp(math.log(x+1))
```

Практически везде, где вы можете поместить значение, можно поместить произвольное выражение, с одним исключением: левая часть инструкции

присваивания должна быть именем переменной. Любое другое выражение слева будет считаться синтаксической ошибкой (мы увидим исключения из этого правила позже).

```
>>> minutes = hours * 60                # правильно
>>> hours * 60 = minutes                # неправильно!
SyntaxError: can't assign to operator
```

ДОБАВЛЕНИЕ НОВЫХ ФУНКЦИЙ

До сих пор мы использовали только функции, которые изначально присутствуют в языке Python, но можно создавать и новые функции. Для **определения новой функции (function definition)** необходимо указать ее имя и последовательность инструкций, которые выполняются при ее вызове.

Ниже представлен пример*:

```
def print_lyrics():
    print('Я дровосек, и со мной все в порядке.')
    print('Я работал весь день, и теперь я в достатке.')
```

Слово `def` — ключевое; оно указывает, что это определение функции. Имя функции — `print_lyrics`. Правила для имен функций те же, что и для имен переменных: латинские буквы, цифры и подчеркивание допустимы, но первый символ не может быть цифрой. Вы не можете использовать зарезервированные (ключевые) слова в качестве имен функций, и вам следует избегать использования переменных и функций с одинаковыми именами.

Пустые скобки после имени указывают, что эта функция не принимает никаких аргументов.

Первая строка в функции называется **заголовком (header)**, весь остальной ее код называется **телом (body)**. Заголовок должен заканчиваться двоеточием, а тело должно иметь отступ. По правилам, отступ всегда равен четырем пробелам. Тело может содержать любое количество инструкций.

Строки в инструкциях печати заключены в кавычки. Можно использовать как одинарные, так и двойные кавычки, главное — в паре; большинство программистов используют одинарные кавычки, за исключением случаев, когда в строке используется апостроф (по сути, та же одинарная кавычка).

---

* Здесь и далее в качестве примеров использованы слова из скетчей комик-группы «Монти Пайтон». *Прим. ред.*

Все кавычки (одинарные и двойные) должны быть «прямыми», их набирают клавишей, расположенной рядом с **Enter** на клавиатуре. «Косые» кавычки и кавычки-елочки (**“”,** «») нельзя использовать в коде на языке Python.

Если вы определяете функции в интерактивном режиме, интерпретатор отображает многоточие (**…**), чтобы вы знали, что определение еще не завершено:

```
>>> def print_lyrics():
... print('Я дровосек, и со мной все в порядке.')
... print('Я работал весь день, и теперь я в достатке.')
...
```

Чтобы завершить определение функции, вы должны два раза нажать клавишу Enter (то есть ввести пустую строку).

Определение функции создает **объект функции (function object)**, который имеет тип `function`:

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
<class 'function'>
```

Синтаксис для вызова новой функции такой же, как и для встроенных функций:

```
>>> print_lyrics()
Я дровосек, и со мной все в порядке.
Я работал весь день, и теперь я в достатке.
```

Как только вы определили функцию, вы можете использовать ее внутри другой функции. Например, чтобы повторить предыдущий припев, мы могли бы написать функцию с именем `repeat_lyrics()`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

А затем вызовем `repeat_lyrics`:

```
>>> repeat_lyrics()
Я дровосек, и со мной все в порядке.
Я работал весь день, и теперь я в достатке.
Я дровосек, и со мной все в порядке.
Я работал весь день, и теперь я в достатке.
```

Хотя на самом деле песня заканчивается по-другому.

ОПРЕДЕЛЕНИЕ И ИСПОЛЬЗОВАНИЕ

Собрав воедино фрагменты кода из предыдущего раздела, мы получим такую программу:

```
def print_lyrics():
    print('Я дровосек, и со мной все в порядке.')
    print('Я работал весь день, и теперь я в достатке.')

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

Эта программа определяет две функции: `print_lyrics()` и `repeat_lyrics()`. Определения функций выполняются интерпретатором так же, как и другие инструкции, но в результате создаются объекты функции, а инструкции внутри функций не выполняются до тех пор, пока эти функции не будут вызваны.

Очевидно, вы должны создать функцию, прежде чем сможете ее выполнить. Другими словами, определение функции должно быть выполнено *до* вызова функции.

В качестве упражнения переместите последнюю строку этой программы вверх, чтобы перед определениями появился вызов функции. Запустите программу и посмотрите, какое сообщение об ошибке появится.

Теперь переместите вызов функции назад, а определение функции `print_lyrics()` поставьте после определения `repeat_lyrics()`. Что происходит при запуске этой программы?

ПОРЯДОК ВЫПОЛНЕНИЯ

Чтобы убедиться, что функция определена перед ее первым использованием, нужно знать **порядок выполнения инструкций (flow of execution)***.

Выполнение программы всегда начинается с первой инструкции. Далее инструкции выполняются по одной, сверху вниз.

Определения функций в коде программы только создают объект функции. Инструкции, находящиеся в теле функции, не выполняются, пока она не будет вызвана.

---

* Мы используем термин «порядок выполнения» как более однозначный, хотя распространен и «поток выполнения». Но у потока есть и другое значение (см. ru.wikipedia.org/wiki/Поток_выполнения). *Прим. науч. ред.*

Вызов функции похож на небольшую остановку. Вместо перехода к следующей инструкции интерпретатор переходит к телу функции, выполняет инструкции там, а затем возвращается, чтобы продолжить с того места, где остановился.

Это звучит просто, пока вы не узнаете, что одна функция может вызывать другую. В середине одной функции программе может потребоваться выполнить инструкции другой функции. Затем при запуске этой новой функции программе может потребоваться запустить еще одну функцию!

К счастью, Python хорошо отслеживает, где находится интерпретатор, поэтому каждый раз, когда функция завершает работу, интерпретатор возвращается туда, где она остановилась в вызывающей ее функции. Когда он доходит до конца программы, она завершает работу.

Таким образом, при изучении кода программы необходимо следить за порядком выполнения инструкций, так как он не всегда происходит сверху вниз.

ПАРАМЕТРЫ И АРГУМЕНТЫ

Некоторые из функций, которые мы видели, требуют передачи аргументов. Например, когда вы вызываете функцию `math.sin()`, вы передаете число в качестве аргумента. Некоторые функции принимают более одного аргумента: функция `math.pow()` принимает два: основание и экспоненту.

Внутри функции аргументы присваиваются переменным, называемым **параметрами (parameters)**. Ниже представлено определение функции, которая принимает аргумент:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

Эта функция назначает аргумент параметру `bruce`. Когда функция вызывается, она выводит на экран значение параметра (каким бы оно ни было) дважды.

Эта функция работает с любым значением, которое может быть выведено на экран:

```
>>> print_twice('Спам')
Спам
Спам
>>> print_twice(42)
42
42
```

```
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

Правила композиции для встроенных функций применяют и к новым, определяемым программистом, функциям. Поэтому мы можем использовать любые выражения в качестве аргумента для функции `print_twice()`:

```
>>> print_twice('Спам '*4)
Спам Спам Спам Спам
Спам Спам Спам Спам
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

Аргумент вычисляется до вызова функции, поэтому в примерах выражения `'Спам' * 4` и `math.cos(math.pi)` вычисляются только один раз.

Вы также можете использовать переменную в качестве аргумента:

```
>>> michael = 'Эрик, полпчелы.'
>>> print_twice(michael)
Эрик, полпчелы.
Эрик, полпчелы.
```

Имя переменной, которую мы передаем в качестве аргумента (`michael`), не имеет ничего общего с именем параметра (`bruce`). Неважно, какое имя переменной было у значения в вызывающей функции; сейчас в функции `print_twice()` мы будем обращаться к нему по имени `bruce`.

ПЕРЕМЕННЫЕ И ПАРАМЕТРЫ ВНУТРИ ФУНКЦИЙ — ЛОКАЛЬНЫ

Когда вы создаете переменную внутри функции, она является **локальной (local)**, что означает, что она существует только внутри функции. Например:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

Эта функция принимает два аргумента, объединяет их и выводит результат дважды. Ниже представлен пример, где она используется:

```
>>> line1 = 'Тили-тили'
>>> line2 = 'трали-вали.'
>>> cat_twice(line1, line2)
```

```
Тили-тили трали-вали.
Тили-тили трали-вали.
```

Когда выполнение функции `cat_twice()` завершается, переменная `cat` уничтожается. Если мы попытаемся вывести ее, то получим исключение:

```
>>> print(cat)
NameError: name 'cat' is not defined
```

Параметры также локальны. Например, за пределами функции `print_twice()` нет ничего, что относится к `bruce`.

СТЕКОВЫЕ ДИАГРАММЫ

Чтобы отследить, какие переменные и где можно использовать, полезно нарисовать **стековую диаграмму**. Как и диаграммы состояний, стековые диаграммы отображают значение каждой переменной, но они также показывают функцию, которой принадлежит каждая переменная.

Каждая функция представлена отдельным **фреймом**. Фрейм — это блок с именем функции рядом с ним, а также параметрами и переменными функции внутри него. Диаграмма стека для предыдущего примера показана на рис. 3.1.

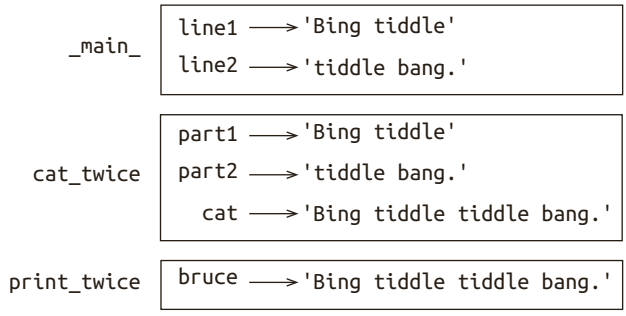


Рис. 3.1. Стековая диаграмма

Фреймы расположены в стеке, который указывает, какая функция вызвала какую, и так далее. В этом примере функция `print_twice()` была вызвана функцией `cat_twice()`, а функция `cat_twice()` вызвана `__main__` — это имя самого верхнего фрейма. Когда вы создаете переменную вне какой-либо функции, она принадлежит `__main__`.

Каждый параметр ссылается на то же значение, что и соответствующий ему аргумент. Так, `part1` имеет то же значение, что и `line1`; `part2` имеет то же значение, что и `line2`; а `bruce` имеет то же значение, что и `cat`.

Если во время вызова функции возникает ошибка, Python выводит имя функции, имя функции, вызвавшей ее, и имя функции, вызвавшей ту, вплоть до `__main__`.

Например, если вы попытаетесь получить доступ к `cat` из `print_twice`, вы получите следующую ошибку `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print(cat)
NameError: name 'cat' is not defined
```

Этот список функций называется **трассировкой (traceback)**. Он сообщает вам, в каком программном файле произошла ошибка и какая строка какой функции выполнялась в это время. Он также показывает строку кода, которая вызвала ошибку.

Порядок функций в трассировке совпадает с порядком фреймов в стековой диаграмме. Функция, в которой произошла ошибка, находится внизу.

РЕЗУЛЬТАТИВНЫЕ ФУНКЦИИ И VOID-ФУНКЦИИ

Некоторые из функций, которые мы использовали, например математические, возвращают результат. За неимением другого решения я называю их **результативные (fruitful) функции**. Другие функции, вроде `print_twice()`, выполняют действие, но не возвращают значение. Они называются **void-функциями** (или иногда — **функциями, не возвращающими результат**).

Чаще всего при вызове функции, возвращающей значение, вы планируете что-то сделать с результатом; например, вы можете присвоить его переменной или использовать в составе выражения:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

Если вызвать функцию в интерактивном режиме, Python выведет результат:

```
>>> math.sqrt(5)
2.2360679774997898
```

Но если в скрипте вызвать функцию, возвращающую значение, и не сохранить возвращаемое значение в переменную — оно потеряется навсегда.

```
math.sqrt(5)
```

Этот скрипт вычисляет квадратный корень из 5, но, поскольку он не хранит и не выводит результат, он не очень полезен.

Void-функции могут выводить что-то на экран или иметь какой-то другой эффект, но они не возвращают никакого значения. Если вы попробуете присвоить такую функцию переменной, вы получите специальное значение None:

```
>>> result = print_twice('Бинго')
Бинго
Бинго
>>> print(result)
None
```

Значение None — это не то же самое, что строка 'None'. Это специальное значение, которое имеет собственный тип:

```
>>> print(type(None))
<class 'NoneType'>
```

Все функции, которые мы написали до сих пор, не возвращали никакой результат. В следующих главах мы начнем писать функции, которые возвращают значения.

ЗАЧЕМ НУЖНЫ ФУНКЦИИ?

Поясним, почему стоит тратить силы и делить код программы на функции.

- Создание новой функции позволяет присвоить имя группе инструкций, что упрощает чтение и отладку программы.
- Функции уменьшают объем кода программы за счет исключения повторяющегося кода. А если вы захотите что-то изменить, нужно будет сделать это только в одном месте.
- Разделение кода большой программы на функции упрощает отладку, так как можно тестировать каждую из функций индивидуально.

— Хорошо продуманные функции зачастую полезны во многих программах. Как только вы напишете и отладите одну, вы сможете использовать ее снова и снова.

ОТЛАДКА

Один из важнейших навыков разработчика — отладка. Иногда она раздражает, но все равно остается едва ли не самой интеллектуально насыщенной, сложной и интересной частью программирования.

Отладка в чем-то похожа на детективную работу. Вы сталкиваетесь с подсказками и должны воссоздать процессы и события, которые привели к полученным результатам.

Отладка похожа на экспериментальную науку. Как только вы поймете, что что-то идет не так, вы измените свою программу и попробуете снова. Если ваша гипотеза оказалась верной, вы можете предсказать результат модификации и на шаг приблизиться к работающей программе. Если ваша гипотеза была неверна, вы должны придумать новую. Как заметил Шерлок Холмс: «Когда вы устранили невозможное, все, что остается, каким бы невероятным оно ни было, должно быть правдой» (Артур Конан Дойл, «Знак Четырех»).

Для некоторых людей программирование и отладка — одно и то же. То есть программирование — это процесс постепенной отладки программы, пока она не выполнит то, что вы хотите. Идея в том, что вы должны начать с работающей программы и вносить небольшие изменения, отлаживая их по мере продвижения.

Например, Linux — это операционная система, содержащая миллионы строк кода. Но началась она как простая программа, которую Линус Торвальдс использовал для исследования чипа Intel 80386. По словам Ларри Гринфилда, «одним из ранних проектов Линуса была программа, которая переключалась между печатью строк “AAAA” и “BBBB”. Позже это превратилось в Linux» (Руководство пользователя Linux, Бета-версия 1).

СЛОВАРЬ ТЕРМИНОВ

Функция:

Именованная последовательность инструкций, выполняющая некоторые полезные вычисления. Функции не обязательно принимают аргументы и возвращают значения.

Определение функции:

Инструкция, которая создает новую функцию с указанием ее имени, параметров и инструкций, которые она выполняет.

Объект функции:

Значение, созданное определением функции. Имя функции — это переменная, которая ссылается на объект функции.

Заголовок:

Первая строка определения функции.

Тело:

Последовательность инструкций внутри определения функции.

Параметр:

Имя, используемое внутри функции для ссылки на значение, переданное в качестве аргумента.

Вызов функции:

Инструкция, которая запускает функцию. Она состоит из имени функции, за которым следует список аргументов в скобках.

Аргумент:

Значение, предоставляемое функции при ее вызове. Это значение присваивается соответствующему параметру в функции.

Локальная переменная:

Переменная, определенная внутри функции. Локальная переменная может использоваться только внутри ее функции.

Возвращаемое значение:

Результат функции. Если вызов функции используется в качестве выражения, возвращаемое значение является значением выражения.

Функция, возвращающая значение/результат:

Функция, которая возвращает значение/результат.

Void-функция (функция, не возвращающая результат):

Функция, которая всегда возвращает `None`.

`None`:

Специальное значение, возвращаемое функциями, которые не возвращают результат.

Модуль:
: Файл, содержащий набор связанных функций и других определений.

Инструкция импорта:
: Инструкция, которая считывает файл модуля и создает объект модуля.

Объект модуля:
: Значение, созданное инструкцией `import`, через которое можно получить доступ к модулю и его функциям.

Точечная нотация:
: Синтаксис для вызова функции модуля, для чего нужно указать имя модуля, а за ним — точку и имя функции.

Композиция:
: Использование выражения в составе большего выражения или инструкции в части другой, более крупной инструкции.

Порядок выполнения:
: Порядок, в котором выполняются инструкции.

Стековая диаграмма:
: Графическое представление стека функций, их переменных и значений, на которые они ссылаются.

Фрейм:
: Блок в диаграмме стека, представляющий вызов функции. Он содержит локальные переменные и параметры функции.

Трассировка:
: Список функций, которые выполняются, распечатывается при возникновении ошибки (исключения).

УПРАЖНЕНИЯ

Упражнение 3.1

Напишите функцию `right_justify()`, которая принимает строку `s` в качестве параметра и печатает строку с достаточным количеством пробелов в начале строки, так чтобы последняя буква строки находилась в столбце 70 на экране:

```
>>> right_justify('monty')
                                        monty
```

Совет. Используйте конкатенацию и повторение строк. Кроме того, Python содержит встроенную функцию `len()`, которая возвращает длину строки, так что значение `len('monty')` равно `5`.

Упражнение 3.2

Объект функции — это значение, которое вы можете присвоить переменной или передать в качестве аргумента. Например, `do_twice()` — это функция, которая принимает объект функции в качестве аргумента и вызывает его дважды:

```
def do_twice (f):
    f()
    f()
```

Ниже представлен пример, в котором функция `do_twice()` используется для вызова функции `print_spam()` дважды:

```
def print_spam():
    print('спам')

do_twice(print_spam)
```

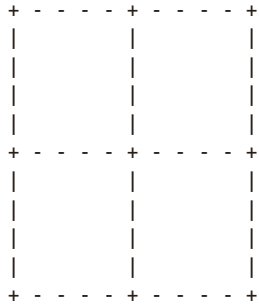
1. Запишите этот пример в файл скрипта и протестируйте его.
2. Измените код функции `do_twice()` так, чтобы она принимала два аргумента, объект функции и значение и дважды вызывала функцию, передавая значение в качестве аргумента.
3. Запишите определение функции `print_twice()`, приведенное ранее в этой главе, в свой скрипт.
4. Используйте модифицированную версию функции `do_twice()`, чтобы дважды вызвать функцию `print_twice()`, передав в качестве аргумента значение `'спам'`.
5. Определите новую функцию `do_four()`, которая принимает объект функции и значение и вызывает функцию четыре раза, передавая значение в качестве параметра. В теле этой функции должно быть только две инструкции, а не четыре.

Решение: **thinkpython2.com/code/do_four.py**.

Упражнение 3.3

Примечание. Используйте только те инструкции и функции, которые мы уже изучили.

1. Напишите функцию, которая рисует сетку следующим образом:



Совет. Чтобы вывести более одного значения в строке, вы можете напечатать последовательность значений через запятую:

```
print('+', '-')
```

По умолчанию функция `print()` переходит на следующую строку, но вы можете переопределить это поведение и указать пробел в конце, например так:

```
print('+', end=' ')
print('-')
```

Результатом выполнения этих инструкций будет `'+ -'`.

Функция `print` без аргумента завершает текущую строку и переходит на следующую.

2. Напишите функцию, которая рисует похожую сетку с четырьмя строками и четырьмя столбцами.

Решение: **thinkpython2.com/code/grid.py**. Примечание: это упражнение основано на задаче, впервые опубликованной в книге Стива Олайна, Practical C Programming, Third Edition, вышедшей в издательстве O'Reilly Media в 1997 году.

ГЛАВА 4
ПРАКТИЧЕСКИЙ ПРИМЕР: РАЗРАБОТКА ИНТЕРФЕЙСА

В этой главе разобран подход к процессу разработки функций, которые используются вместе.

Тут представлен модуль `turtle`, который создает графические изображения с помощью черепашьей графики. Модуль `turtle` входит в большинство дистрибутивов Python, но в приложении PythonAnywhere он не работает (по крайней мере, так было, когда я писал книгу).

Если вы установили Python на свой компьютер, то модуль `turtle` уже доступен и вы сможете запустить примеры. Если еще не установили — сейчас самое подходящее время. Я опубликовал инструкции по ссылке **tinyurl.com/thinkpython2e**.

Примеры кода из этой главы доступны по адресу **thinkpython2.com/code/polygon.py**.

МОДУЛЬ TURTLE

Чтобы проверить, установлен ли у вас модуль `turtle`, откройте интерпретатор Python и введите:

```
>>> import turtle
>>> bob = turtle.Turtle()
```

Когда вы запустите этот код, должно появиться новое окно с маленькой стрелкой, похожей на черепашку. Закройте окно.

Создайте файл *mypolygon.py* и введите следующий код:

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

Модуль `turtle` (со строчной буквы *t*) предоставляет функцию `Turtle` (с прописной буквы *T*), создающую объект типа `Turtle`, который мы назначаем переменной `bob`. Вывод переменной `bob` отображает что-то вроде:

```
<turtle.Turtle object at 0xb7bfbf4c>
```

Строка означает, что `bob` относится к объекту с типом `Turtle`, как определено в модуле `turtle`.

Метод `mainloop()` ожидает, пока пользователь что-нибудь сделает, хотя в данном случае пользователю ничего не остается, кроме как закрыть окно.

Создав черепашку, вы можете вызвать **метод**, чтобы переместить ее в окне. Метод похож на функцию, но с несколько иным синтаксисом. Например, чтобы переместить черепашку вперед, используется следующий код:

```
bob.fd(100)
```

Метод `fd()` связан с объектом черепашки, который мы назвали `bob`. Вызов метода похож на выполнение запроса: вы просите черепашку по имени `bob` двигаться вперед.

Аргумент метода `fd()` — это расстояние в пикселях, поэтому фактический размер пройденного расстояния зависит от разрешения вашего экрана.

Другие методы, которые вы можете вызвать: `bk()` — для перемещения назад, `lt()` — для поворота влево и `rt()` — для поворота вправо. В качестве аргумента методов `lt()` и `rt()` указывается угол в градусах.

Кроме того, каждая черепашка держит перо, которое находится либо внизу, либо вверху. Если перо опущено, черепашка оставляет след при движении. Методы `pu()` и `pd()` означают «перо вверх — pen up» и «перо вниз — pen down», соответственно.

Чтобы нарисовать прямой угол, добавьте следующие строки в программу (после создания `bob` и перед вызовом `mainloop()`):

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

Когда вы запустите эту программу, вы увидите, что черепашка `bob` двигается на восток, а затем на север, оставляя за собой тонкую линию.

Теперь измените программу так, чтобы нарисовать квадрат. Не переходите к следующему разделу, пока у вас не получится!

ПРОСТОЕ ПОВТОРЕНИЕ

Скорее всего, вы написали что-то вроде этого:

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

Мы можем сделать то же самое, но написав меньше кода, если воспользуемся инструкцией `for`. Добавьте показанный ниже код в файл *mypolygon.py* и запустите программу снова:

```
for i in range(4):
    print('Hello!')
```

Вы должны увидеть следующее:

```
Hello!
Hello!
Hello!
Hello!
```

Это простейший пример использования инструкции `for`, позже мы рассмотрим более сложные. Но его должно быть достаточно, чтобы вы смогли переписать свою программу для рисования квадратов. Не продолжайте чтение, пока не выполните задание.

Ниже показана инструкция `for`, которая рисует квадрат:

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

Синтаксис инструкции `for` похож на синтаксис определения функции. У него есть заголовок, который заканчивается двоеточием, и отступ для тела. Тело может содержать любое количество инструкций.

Инструкция `for` также называется **циклом (loop)**, потому что порядок выполнения проходит через тело цикла, а затем возвращается к началу. В данном случае инструкции в теле цикла выполняются четыре раза.

Этот вариант на самом деле немного отличается от предыдущего кода для рисования квадрата, потому что черепашка делает еще один поворот после отрисовки последней стороны квадрата. Дополнительный ход занимает больше времени, но упрощает код, если мы делаем в цикле одно и то же несколько раз. Эта версия также приводит черепашку в исходное положение, и она смотрит в начальном направлении.

УПРАЖНЕНИЯ

Ниже приводится серия упражнений с использованием TurtleWorld. Несмотря на то что они забавны, у них еще есть и смысл. Выполняя их, подумайте, в чем суть.

Далее вы найдете решения этих задачек, но не подглядывайте, пока не закончите (или, по крайней мере, не попробуете).

1. Напишите функцию `square()`, которая принимает параметр `t` (черепашку). Эта функция должна рисовать квадрат с помощью черепашки.

   Напишите вызов функции, который передает `bob` в качестве аргумента функции `square()`, а затем снова запустите программу.
2. Добавьте еще один параметр `length` в функцию `square()`. Измените тело функции так, чтобы длина сторон была равна `length`, а затем измените вызов функции, чтобы передать ей второй аргумент. Запустите программу еще раз. Протестируйте программу, указывая различные значения `length`.
3. Скопируйте код функции `square()` и измените ее имя на `polygon()`. Добавьте параметр `n` и измените тело функции так, чтобы она рисовала *n*-сторонний правильный многоугольник.

   Подсказка: внешние углы *n*-стороннего правильного многоугольника равны 360/*n* градусов.
4. Напишите функцию `circle()`, которая принимает черепашку `t` и радиус `r` в качестве параметров и рисует приблизительный круг, вызывая функцию `polygon()` с соответствующей длиной и количеством сторон. Проверьте функцию, указывая различные значения `r`.

   Подсказка: определите длину окружности круга (`circumference`) и убедитесь, что `length * n = circumference`.
5. Создайте более обобщенную версию функции `circle()` с именем `arc()`, которая принимает дополнительный параметр `angle`, определяющий,

какую долю круга нарисовать. Значение параметра `angle` измеряется в градусах, поэтому, когда оно равно `360`, функция `arc()` должна рисовать замкнутый круг.

ИНКАПСУЛЯЦИЯ

В первом упражнении требовалось поместить код для рисования квадрата в определение функции, а затем вызвать эту функцию, передав черепашку в качестве параметра. Ниже представлено решение.

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

square(bob)
```

Внутренние инструкции `fd` и `lt` имеют двойной отступ, показывая, что они находятся внутри цикла `for`, а тот, в свою очередь, находится внутри определения функции. Следующая строка, `square(bob)`, не имеет отступов слева, что указывает на конец как цикла `for`, так и определения функции.

Внутри функции `t` относится к той же черепашке `bob`, поэтому `t.lt(90)` имеет тот же результат, что и `bob.lt(90)`. В таком случае, почему бы не вызвать параметр `bob`? Суть в том, что `t` может быть любой черепашкой, а не только `bob`, поэтому вы можете создать вторую черепашку и передать ее в качестве аргумента функции `square()`:

```
alice = Turtle()
square(alice)
```

Заключение фрагмента кода в функцию называется **инкапсуляцией (encapsulation)**. Одно из преимуществ инкапсуляции в том, что она позволяет присвоить фрагменту кода имя, которое помогает документировать код. Другое преимущество состоит в том, что, если вы повторно используете код, более лаконично вызвать функцию два раза, а не копировать/вставлять тело!

ОБОБЩЕНИЕ

На следующем шаге в функцию `square()` добавляется параметр `length`. Ниже показано решение:

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)

square(bob, 100)
```

Добавление параметра в функцию называется **обобщением**, или — иногда — **универсализацией**, поскольку в результате функция становится более универсальной: в предыдущей версии квадрат всегда был одинакового размера; в этой версии размер может быть любой.

Следующий шаг также является обобщением. Вместо того чтобы рисовать квадраты, функция `polygon()` рисует обычные многоугольники с любым числом сторон. Ниже показано решение:

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)

polygon(bob, 7, 70)
```

В этом примере рисуется семисторонний многоугольник с длиной стороны 70.

Если вы используете версию Python 2, значение угла может быть округлено из-за целочисленного деления. Простое решение заключается в вычислении угла как `angle = 360.0 / n`. Поскольку числитель представлен числом с плавающей точкой, то и результатом будет число с плавающей точкой.

Если функция имеет несколько числовых аргументов, легко забыть, что они собой представляют или в каком порядке должны перечисляться. В этом случае рекомендуется в список аргументов включать имена параметров:

```
polygon(bob, n=7, length=70)
```

Это так называемые **ключевые аргументы (keyword arguments)**, потому что они включают в себя имена параметров в качестве «ключевых слов» (не путать с зарезервированными ключевыми словами Python, такими как `while` и `def`).

Такой синтаксис упрощает понимание кода программы. Это также напоминание о том, как работают аргументы и параметры: когда вы вызываете функцию, значения аргументов присваиваются параметрам.

РАЗРАБОТКА ИНТЕРФЕЙСА

На следующем шаге нужно написать функцию `circle()`, которая принимает радиус `r` в качестве параметра. Ниже показано простое решение, где используется функция `polygon()` для рисования многоугольника с 50 сторонами:

```
import math

def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

Первая строка вычисляет длину окружности круга с радиусом `r` по формуле $2\pi r$. Так как мы используем свойство `math.pi`, мы должны импортировать модуль `math`. По соглашению инструкции импорта обычно указываются в начале скрипта.

Число `n` — это количество отрезков в нашей окружности, а `length` — длина каждого отрезка. Таким образом, функция `polygon()` рисует 50-сторонний многоугольник, который приближен по форме к кругу с радиусом `r`.

Одно из ограничений этого решения заключается в том, что `n` — это константа, а значит, для очень больших кругов отрезки линии слишком длинные, а для маленьких кругов мы тратим время на рисование очень маленьких отрезков. Одно из решений — обобщить функцию, задав `n` в качестве параметра. Это дало бы пользователю (кто бы ни вызывал функцию `circle`) больше контроля, но интерфейс был бы менее понятен.

Интерфейс (interface) функции представляет собой краткое изложение того, как ее использовать: какие параметры? Что делает функция? Что представляет собой возвращаемое значение? Интерфейс «понятен», если он позволяет программисту вызывать функцию так, как он хочет, не касаясь ненужных деталей.

В этом примере `r` принадлежит интерфейсу, потому что определяет круг, который следует нарисовать. Значение `n` менее важно, потому что оно относится к деталям того, *как* круг должен быть отображен.

Вместо того чтобы загромождать интерфейс, лучше выбрать подходящее значение `n` в зависимости от размера окружности:

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 1
```

```
    length = circumference / n
    polygon(t, n, length)
```

Теперь число сегментов — это целое число, приблизительно равное circumference/3, поэтому длина каждого сегмента составляет приблизительно 3, что достаточно мало, чтобы круги выглядели хорошо, но достаточно велико, чтобы функция была эффективна и приемлема для круга любого размера.

РЕФАКТОРИНГ

Когда я писал функцию circle(), я мог повторно использовать функцию polygon(), потому что многогранный многоугольник приблизительно похож на круг. Но функция arc() не так дружелюбна; мы не можем использовать функцию polygon() или circle(), чтобы нарисовать дугу.

Один из вариантов — начать с дублирования функции polygon() и преобразовать ее в arc(). Результат будет выглядеть так:

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = angle / n

    for i in range(n):
        t.fd(step_length)
        t.lt(step_angle)
```

Вторая часть функции выше выглядит как функция polygon(), но мы не можем повторно использовать функцию polygon() без изменения интерфейса. Мы можем обобщить, то есть универсализировать, функцию polygon(), передав угол в качестве третьего аргумента, но тогда имя polygon() больше не годится! Вместо этого давайте назовем ее общим именем polyline():

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

Теперь мы можем переписать функции polygon() и arc(), чтобы они использовали функцию polyline():

```
def polygon(t, n, length):
    ngle = 360.0 / n
    polyline(t, n, length, angle)
```

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

Наконец, мы можем переписать функцию `circle()`, чтобы она использовала функцию `arc()`:

```
def circle(t, r):
    arc(t, r, 360)
```

Этот процесс — изменения структуры программы для улучшения интерфейсов и упрощения повторного использования кода — называется **рефакторингом (refactoring)**. В данном случае мы заметили, что в функциях `arc()` и `polygon()` используется похожий код, поэтому мы «вынесли его» в функцию `polyline()`.

Если бы мы планировали заранее, мы могли бы сначала написать функцию `polyline()` и избежать рефакторинга, но в начале проекта недостаточно информации для проработки всех интерфейсов. А уже в процессе разработки вы начинаете лучше понимать проблему. Иногда рефакторинг — признак того, что вы чему-то научились.

СПОСОБ РАЗРАБОТКИ

Способ разработки (development plan) — это подход или метод в процессе написания программ. Метод, который мы использовали в этом примере: «инкапсуляция и обобщение». Шаги такие.

1. Начните с написания небольшой программы без определения функций.
2. Как только вы запустите программу и она будет работать, найдите связанную часть кода и инкапсулируйте ее в функцию, присвойте ей имя.
3. Обобщите функцию, добавив подходящие параметры.
4. Повторяйте шаги 1–3, пока не получите набор рабочих функций. Копируйте и вставляйте работающий код, чтобы избежать повторного ввода (и повторной отладки).
5. Ищите возможности улучшить программу путем рефакторинга. Например, если у вас есть похожий код в нескольких местах, рассмотрите возможность его преобразования в универсальную функцию.

Описанный процесс не лишен недостатков — и мы увидим альтернативы позже, — но он может быть полезен, если вы не знаете заранее, как разделить программу на функции. Такой подход позволяет вам проектировать по мере разработки.

СТРОКИ ДОКУМЕНТАЦИИ

Строка документации (docstring) — это строка в начале функции, которая объясняет интерфейс ("doc" — это сокращение от "documentation"). Ниже представлен пример:

```
def polyline(t, n, length, angle):
    """ Рисует n отрезков с заданной длиной length и углами angle
    (в градусах) между ними. t — это черепашка.
    """
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

По соглашению все строки документации заключаются в тройные кавычки, также они называются многострочными, поскольку тройные кавычки позволяют тексту занимать более одной строки.

Описание должно быть кратким, но понятным, со всей важной информацией, которая понадобится для использования этой функции. Здесь кратко объясняется, что делает функция (без деталей, как именно она это делает), как каждый параметр влияет на поведение функции и какой тип должен иметь каждый параметр (если это не очевидно).

Написание такого рода документации — важная часть разработки интерфейсов. Хорошо разработанный интерфейс должен быть простым для объяснения; если вам сложно объяснить хотя бы одну из ваших функций, возможно, интерфейс следует улучшить.

ОТЛАДКА

Интерфейс похож на соглашение между функцией и вызывающей стороной. Вызывающая сторона соглашается предоставить определенные параметры, а функция соглашается выполнить определенную работу.

Например, функции `polyline()` требуется четыре аргумента: `t` должен иметь тип `Turtle`; `n` должен быть целым числом; `length` должен быть положительным числом; и `angle` должен быть числом, представленным в градусах.

Эти требования называются **входными условиями** (preconditions), потому что они должны быть выполнены *до* того, как функция начнет выполняться. И наоборот, условия в конце функции называются **выходными условиями** (postconditions). Выходные условия включают предполагаемый эффект функции (например, рисование отрезков линий) и любые побочные эффекты (например, перемещение черепашки или внесение других изменений).

Входные условия лежат в зоне ответственности вызывающей части программы. Если вызывающая часть нарушает (правильно задокументированные!) предварительные условия и в результате функция работает неправильно, ошибка происходит при вызове, а не в самой функции.

Если предварительные условия выполнены, а постусловия — нет, ошибка внутри функции. Если ваши входные и выходные условия понятны, они могут помочь при отладке.

СЛОВАРЬ ТЕРМИНОВ

Метод:
: Функция, которая связана с объектом и вызывается с использованием точечной нотации.

Цикл:
: Часть программы, которая может запускаться повторно.

Инкапсуляция:
: Процесс преобразования последовательности инструкций в определение функции.

Обобщение:
: Процесс замены излишне специфичного кода (например, числа) на более общий (например, на переменную или параметр).

Ключевой аргумент:
: Аргумент, который включает имя параметра в качестве ключевого слова.

Интерфейс:
: Описание того, как использовать функцию, включая имя, описание аргументов и возвращаемого значения.

Рефакторинг:
: Процесс модификации рабочей программы для улучшения функциональных интерфейсов и качества кода в целом.

Способ разработки:
: Подход к процессу написания программ.

Строка документации:

Строка, которая находится в верхней части определения функции для документирования интерфейса функции.

Входное условие:

Требование, которое должно быть выполнено вызывающей стороной до запуска функции.

Выходное условие:

Требование, которое должно быть выполнено функцией до ее завершения.

УПРАЖНЕНИЯ

Упражнение 4.1

Скачайте код этой главы по адресу **thinkpython2.com/code/polygon.py**.

1. Нарисуйте стековую диаграмму, которая показывает состояние программы при выполнении функции `circle(bob, radius)`. Вы можете вычислить значения вручную или добавить инструкцию печати в код.
2. Версия функции `arc()` в разделе «Рефакторинг» не очень точна, поскольку линейное приближение круга всегда находится за пределами истинного круга. В результате черепашка оказывается в нескольких пикселях от позиции, где должна быть. Мое решение демонстрирует, как уменьшить влияние этой ошибки. Прочтите код и попробуйте в нем разобраться. Если вы нарисуете диаграмму, вы сможете точнее увидеть, как он работает.

Упражнение 4.2

Напишите общий набор функций, которые могут рисовать цветы, показанные на рис. 4.1.

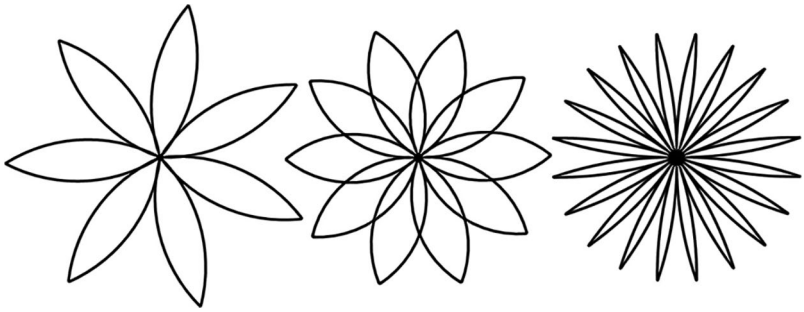


Рис. 4.1. Цветы, нарисованные черепашкой

Решение: **thinkpython2.com/code/flower.py**, также потребуется файл **thinkpython2.com/code/polygon.py**.

Упражнение 4.3
Напишите общий набор функций, которые могут рисовать фигуры, показанные на рис. 4.2.

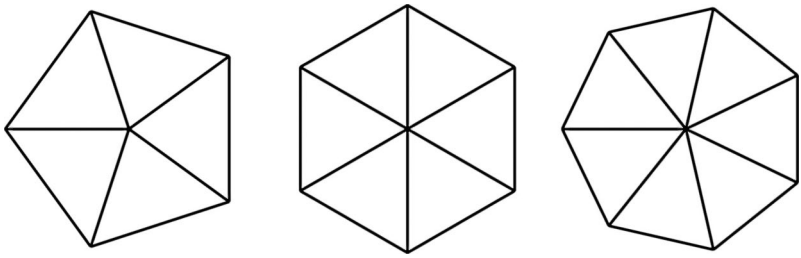


Рис. 4.2. Черепашьи пироги

Решение: **thinkpython2.com/code/pie.py**.

Упражнение 4.4
Буквы алфавита могут быть собраны из относительно небольшого числа основных элементов, таких как вертикальные и горизонтальные линии, и несколько кривых.

Разработайте алфавит, который можно нарисовать с минимальным количеством основных элементов, а затем напишите функции, которые рисуют буквы.

Вы должны написать по одной функции для каждой буквы с именами `draw_a()`, `draw_b()` и так далее, а затем поместить ваши функции в файл *letters.py*. Вы можете скачать программу «черепашья печатная машинка» по адресу **thinkpython2.com/code/typewriter.py**, если понадобится протестировать ваш код.

Готовое решение доступно по адресу **thinkpython2.com/code/letters.py**; также потребуется файл **thinkpython2.com/code/polygon.py**.

Упражнение 4.5
Почитайте о спиралях на вики-странице **en.wikipedia.org/wiki/Spiral**; затем напишите программу, которая рисует спираль Архимеда (или любую другую).

Решение: **thinkpython2.com/code/spiral.py**.

ГЛАВА 5
УСЛОВИЯ И РЕКУРСИЯ

Основная тема этой главы — инструкция `if`, которая выполняет различный код в зависимости от состояния программы. Но сначала я хочу представить два новых оператора: целочисленное деление и деление по модулю.

ЦЕЛОЧИСЛЕННОЕ ДЕЛЕНИЕ И ДЕЛЕНИЕ ПО МОДУЛЮ

Оператор **целочисленного деления (floor division)**, `//`, делит одно число на другое и округляет результат вниз до целого. Например, предположим, что продолжительность фильма составляет 105 минут. Допустим, надо вычислить, сколько это в часах. Обычное деление возвращает число с плавающей точкой:

```
>>> minutes = 105
>>> minutes / 60
1.75
```

Но мы обычно не пишем часы в таком формате. Целочисленное деление возвращает целое число часов, отбрасывая дробную часть:

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

Чтобы получить остаток, вы можете вычесть один час в минутах:

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

Альтернатива — использование оператора **деления по модулю (modulus operator)**, `%`, который делит два числа и возвращает остаток:

```
>>> remainder = minutes % 60
>>> remainder
45
```

Оператор деления по модулю более полезен, чем кажется на первый взгляд. Например, вы можете проверить, делится ли одно число на другое: если `x % y` равно нулю, то `x` делится на `y`.

Кроме того, вы можете извлечь самую правую цифру или все цифры числа. Например, `x % 10` возвращает крайнюю правую цифру `x` (в десятичной системе счисления). Точно так же `x % 100` возвращает последние две цифры.

Если вы используете Python 2, деление выполняется иначе. Оператор деления `/` выполняет целочисленное деление, если оба операнда представлены целыми числами, и деление с плавающей точкой, если хотя бы один из операндов представлен числом с плавающей точкой.

ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ

Логическим (boolean expression) (или булевым) называется выражение, которое может быть истинным или ложным. В следующих примерах используется оператор `==`, который сравнивает два операнда и возвращает `True`, если они равны, и `False` — если нет:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` и `False` — это специальные значения, принадлежащие типу `bool`; это не строки:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

`==` — один из **операторов сравнения (relational operators)**. Другие показаны ниже:

```
x != y          # x не равно y
x > y           # x больше, чем y
x < y           # x меньше, чем y
x >= y          # x больше или равно y
x <= y          # x меньше или равно y
```

Хотя эти операции, вероятно, вам знакомы, символы в языке Python отличаются от математических. Распространенная ошибка заключается в использовании одного знака равенства (`=`) вместо двойного (`==`). Запомните, что `=` — оператор присваивания, а `==` — оператор сравнения. Кроме того, в Python нет таких обозначений, как `=<` или `=>`.

ЛОГИЧЕСКИЕ ОПЕРАТОРЫ

Существует три **логических оператора (logical operators)**: `and`, `or` и `not`. Семантика (значение) этих операторов аналогична значению этих слов в английском языке. Например, `x > 0 and x < 10` истинно, только если `x` больше `0` и меньше `10`.

`n % 2 == 0 or n % 3 == 0` истинно, если выполняется *одно или оба* условия, то есть если число делится на 2 или на 3 без остатка.

Наконец, оператор `not` отрицает логическое выражение, поэтому `not(x > y)` истинно, если `x > y` ложно, то есть если `x` меньше или равно `y`.

Строго говоря, операнды логических операторов должны быть логическими выражениями, но Python не очень строг в этом плане. Любое ненулевое число интерпретируется как `True`:

```
>>> 42 and True
True
```

Такая гибкость может быть полезной, но есть некоторые тонкости, которые могут ввести в заблуждение. Поэтому старайтесь не использовать это свойство (по крайней мере, пока не будете уверены в том, что делаете).

УСЛОВНОЕ ВЫПОЛНЕНИЕ

Для написания полезных программ почти всегда нужна возможность проверять условия и соответственно изменять поведение программы. Условные инструкции (conditional statements) дают нам эту возможность. Самая простая форма — это инструкция `if`:

```
if x > 0:
    print('x — положительное число')
```

Логическое выражение после `if` называется **условием (condition)**. Если оно истинно, инструкции с отступом выполняются. Если нет, то ничего не происходит.

Инструкции `if` имеют ту же структуру, что и определения функций: заголовок, за которым следует тело с отступом. Такие инструкции называются **составными (compound statements)**.

Количество инструкций, которые могут быть указаны в теле, не ограничено, но должна быть хотя бы одна. Иногда приходится создавать тело без инструкций (как правило, чтобы зарезервировать место для кода, который

вы напишете в будущем). В этом случае вы можете использовать инструкцию `pass`, которая ничего не делает.

```
if x < 0:
    pass            # нужно обработать отрицательные значения!
```

АЛЬТЕРНАТИВНОЕ ВЫПОЛНЕНИЕ

Вторая форма инструкции `if` — «альтернативное выполнение», в котором есть две возможности, и условие определяет, какая из них выполняется. Синтаксис выглядит так:

```
if x % 2 == 0:
    print('x — четное число')
else:
    print('x — нечетное число')
```

Если остаток от деления `x` на `2` равен `0`, то мы знаем, что `x` — четное число, и программа отображает соответствующее сообщение. Если условие ложно, выполняется второй набор инструкций. Поскольку условие может быть либо истинным, либо ложным, будет выполнена ровно одна из альтернатив. Альтернативы называются **ветвями (branches)**, потому что это и есть ветви в потоке выполнения.

СВЯЗАННЫЕ УСЛОВИЯ

Иногда существует более двух возможностей и требуется более двух ветвей. Один из способов выразить подобное вычисление — использовать **связанные условия (chained conditional)**:

```
if x < y:
    print('x меньше, чем y')
elif x > y:
    print('x больше, чем y')
else:
    print('x и y равны')
```

`elif` — это аббревиатура для “else if”. Напомню, выполняться будет только одна ветвь. Ограничений на количество инструкций `elif` нет. Инструкции `else` должны быть указаны в конце, сколько бы их ни было.

```
if choice == 'a':
    draw_a()
```

```
elif choice == 'б':
    draw_b()
elif choice == 'c':
    draw_c()
```

Каждое условие проверяется по порядку. Если первое ложно, проверяется следующее, и так далее. Если одно из них истинно, выполняется соответствующая ветвь, и инструкция заканчивает свою работу. Даже если истинно более чем одно условие, выполняется только *первая* истинная ветвь.

ВЛОЖЕННЫЕ УСЛОВИЯ

Одно условие также может быть вложено в другое. Мы могли бы переписать пример из предыдущего раздела следующим образом:

```
if x == y:
    print('x и y равны')
else:
    if x < y:
        print('x меньше, чем y')
    else:
        print('x больше, чем y')
```

Внешняя условная инструкция содержит две ветви. Первая ветвь содержит обычную инструкцию. Вторая ветвь содержит еще одну инструкцию `if`, у которой две собственные ветви. Эти две ветви — простые инструкции. Разумеется, они могли бы быть так же и условными операторами.

Несмотря на то что отступы инструкций позволяют разглядеть структуру, **вложенные условные инструкции (nested conditionals)** очень трудно прочитать с ходу. Поэтому рекомендуется их избегать.

Логические операторы часто предоставляют способ упростить набор вложенных условных инструкций. Например, мы можем переписать предыдущий код, используя всего одно условие:

```
if 0 < x:
    if x < 10:
        print('x — положительное однозначное число.')
```

Инструкция `print` запустится только в том случае, если выполнены оба условия, поэтому мы можем получить тот же эффект с помощью оператора `and`:

```
if 0 < x and x < 10:
    print('x — положительное однозначное число.')
```

Для такого рода условий Python предоставляет более краткий вариант записи:

```
if 0 < x < 10:
    print('x — положительное однозначное число.')
```

РЕКУРСИЯ

Функция может вызывать другую внутри себя; также функция может вызывать и саму себя. Хотя на первый взгляд выгода неочевидна, на деле это одна из самых волшебных вещей, которые может сделать программа. Например, взгляните на следующую функцию:

```
def countdown(n):
    if n <= 0:
        print('Готово!')
    else:
        print(n)
        countdown(n-1)
```

Если значение переменной `n` равно 0 или отрицательно, выводится слово «Готово!». В противном случае выводится значение переменной `n`, а затем вызывается функция с именем `countdown()` — функция вызывает саму себя, — передавая в качестве аргумента `n-1`.

Что произойдет, если мы вызовем эту функцию показанным образом?

```
>>> countdown(3)
```

Выполнение `countdown()` начинается с проверки выражения `n = 3`, и, поскольку значение переменной `n` больше 0, выводится значение `3`, а затем функция вызывает саму себя...

Выполнение `countdown()` начинается с проверки выражения `n = 2`, и, поскольку значение переменной `n` больше 0, выводится значение `2`, а затем функция вызывает саму себя…

Выполнение `countdown()` начинается с проверки выражения `n = 1`, и поскольку значение переменной `n` больше 0, выводится значение 1, а затем функция вызывает саму себя…

Выполнение `countdown()` начинается с проверки выражения `n = 0`, и, поскольку значение переменной `n` не превышает `0` (в данном случае равно `0`), выводится слово «Готово!», а затем функция завершается.

Функция `countdown()`, получившая на вход `n=1`, завершается.

Функция `countdown()`, получившая на вход `n=2`, завершается.

Функция `countdown()`, получившая на вход `n=3`, завершается.

И теперь вы снова в `__main__`. Итак, общий вывод выглядит так:

```
3
2
1
Готово!
```

Функция, которая вызывает сама себя, называется **рекурсивной (recursive)**, а процесс ее выполнения называется **рекурсией (recursion)**.

В качестве другого примера мы можем написать функцию, которая печатает строку `n` раз:

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

Если условие `n <= 0` истинно, **инструкция** `return` завершает выполнение функции. Порядок выполнения немедленно возвращается вызывающей стороне, а остальные строки функции не запускаются.

Остальная часть функции похожа на функцию `countdown()`: она выводит значение переменной `s`, а затем вызывает себя, чтобы вывести `s` еще $n - 1$ раз. Таким образом, количество выведенных строк равно `1 + (n - 1)`, что в сумме составляет `n`.

Для простых примеров, подобных этому, вероятно, проще использовать цикл `for`. Но позже мы увидим примеры, которые сложно написать с помощью цикла `for` и легко реализовать с помощью рекурсии, так что сейчас лучшее время, чтобы изучить этот момент.

СТЕКОВЫЕ ДИАГРАММЫ ДЛЯ РЕКУРСИВНЫХ ФУНКЦИЙ

В разделе «Стековые диаграммы» главы 3 мы использовали стековую диаграмму для представления состояния программы во время вызова функции. Аналогичная диаграмма поможет интерпретировать и рекурсивную функцию.

Каждый раз, когда вызывается функция, Python создает фрейм, содержащий локальные переменные и параметры функции. Для рекурсивной функции в стеке может находиться более одного фрейма одновременно.

Рис. 5.1 показывает диаграмму стека для функции `countdown()`, вызванной с аргументом `n = 3`.

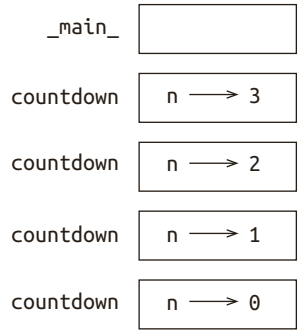


Рис. 5.1. Стековая диаграмма

Как и в остальных случаях, вершина стека — фрейм для `__main__`. Он пуст, потому что мы не создали никаких переменных в `__main__` и не передали туда никаких аргументов.

Четыре фрейма `countdown` имеют разные значения для параметра `n`. Дно стека, где `n = 0`, называется **базовым случаем (base case)**. При таком значении не происходит рекурсивного вызова, так что фреймы больше не создаются.

В качестве упражнения нарисуйте диаграмму стека для функции `print_n()`, вызываемой с параметрами `s = 'Hello'` и `n = 2`. Затем напишите функцию `do_n()`, которая принимает объект функции и число `n` в качестве аргументов и которая вызывает данную функцию `n` раз.

БЕСКОНЕЧНАЯ РЕКУРСИЯ

Если рекурсия никогда не достигает базового случая, она продолжает делать рекурсивные вызовы бесконечно, и программа никогда не завершается. Это так называемая **бесконечная рекурсия (infinite recursion)**, и, как правило, не очень хорошая идея. Ниже представлена маленькая программа с бесконечной рекурсией:

```
def recurse():
    recurse()
```

В большинстве языков программирования программа с бесконечной рекурсией на самом деле не работает вечно. Python сообщает об ошибке при достижении максимальной глубины рекурсии:

```
    File "<stdin>", line 2, in recurse
    File "<stdin>", line 2, in recurse
    File "<stdin>", line 2, in recurse
                    .
                    .
                    .
    File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

Эта трассировка немного больше той, что мы видели в предыдущей главе. Когда возникает ошибка, в стеке содержится 1000 рекурсивных фреймов!

Если вы создали бесконечную рекурсию случайно, проверьте свою функцию, чтобы убедиться, что существует базовый случай, при котором рекурсивный вызов не выполняется. И если есть базовый случай, проверьте, гарантированно ли вы его достигаете.

ВВОД С КЛАВИАТУРЫ

Программы, которые мы писали до сих пор, не принимают никакой информации от пользователя. Они просто делают одно и то же при каждом запуске.

Python предоставляет встроенную функцию `input()`, которая останавливает программу и ждет ввода данных от пользователя. Когда пользователь нажимает клавишу **Return/Enter**, программа возобновляет работу и функция `input()` возвращает ввод пользователя в виде строки. В Python 2 эта функция называется `raw_input()`.

```
>>> text = input()
Чего вы ждете?
>>> text
Чего вы ждете?
```

Прежде чем запрашивать данные от пользователя, рекомендуется напечатать подсказку, указывающую пользователю, что вводить. Функция `input()` принимает подсказку в качестве аргумента:

```
>>> name = input('Как... вас зовут?\n')
Как... вас зовут?
Артур, король бриттов!
>>> name
Артур, король бриттов!
```

Последовательность `\n` в конце подсказки — это специальный символ **конца строки (newline)**, вызывающий переход на новую строку. Вот почему ввод пользователя отображается на следующей строке после подсказки.

Если пользователь должен ввести целое число, то вы можете попробовать преобразовать возвращаемое значение в тип `int`:

```
>>> prompt = 'Какова... скорость ласточки без груза?\n'
>>> speed = input(prompt)
Какова... скорость ласточки без груза?
42
>>> int(speed)
42
```

Но если пользователь напечатает что-то отличное от последовательности цифр, то вы получите ошибку:

```
>>> speed = input(prompt)
Какова... скорость ласточки без груза?
Имеется в виду африканская или европейская ласточка?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

ОТЛАДКА

При возникновении синтаксической ошибки или ошибки выполнения сообщение содержит много информации, иногда это сбивает с толку. К наиболее полезным сведениям можно отнести:

— тип ошибки;
— место возникновения.

Синтаксические ошибки обычно легко найти, но есть несколько подводных камней. Ошибки, связанные с отступами, может быть сложно обнаружить, потому что пробелы и табуляции на глаз не всегда различимы и могут быть пропущены.

```
>>> x = 5
>>> y = 6
    File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

В этом примере проблема в том, что во второй строке есть отступ в один пробел. Но сообщение об ошибке указывает на само имя переменной `y`, что вводит в заблуждение. Обычно сообщения об ошибках указывают на место,

где проблема обнаружена, но фактическая ошибка может возникнуть и раньше в коде, иногда даже в предыдущей строке.

То же самое относится и к ошибкам во время выполнения. Предположим, вы пытаетесь вычислить отношение «сигнал/шум» в децибелах. Формула имеет вид $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$. В Python вы можете написать что-то вроде этого:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

Когда вы запустите эту программу, то увидите исключение:

```
Traceback (most recent call last):
    File "snr.py", line 5, in ?
        decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

Сообщение об ошибке указывает на строку 5, но в этой строке нет ошибки. Чтобы найти реальную ошибку, нужно вывести значение переменной `ratio`, которое оказывается равным `0`. Проблема в строке 4, в которой используется целочисленное деление вместо деления с плавающей точкой.

Вы должны внимательно читать сообщения об ошибках, но не думайте, что все, что там сказано, — абсолютно точная правда.

СЛОВАРЬ ТЕРМИНОВ

Целочисленное деление:

Оператор `//`, который делит два числа и округляет вниз (в сторону нуля) до целого числа.

Оператор деления по модулю:

Оператор, обозначенный знаком `%`, который вычисляет остаток от (после) деления числа и возвращает остаток после деления одного числа на другое.

Логическое выражение:

Выражение, результирующее значение которого — истина или ложь (`True` или `False`).

Оператор сравнения:

Тип оператора, который сравнивает свои операнды: `==`, `!=`, `>`, `<`, `>=` и `<=`.

Логический оператор:

Оператор, который объединяет логические выражения `and`, `or` и `not`.

Условная инструкция

Инструкция, которая управляет порядком выполнения в зависимости от истинности/ложности условия.

Условие:

Логическое выражение в условной инструкции, которое определяет, какая ветвь должна быть выполнена.

Составная инструкция:

Инструкция, которая состоит из заголовка и тела. Заголовок заканчивается двоеточием (`:`). Тело имеет отступ относительно заголовка.

Ветвь:

Одна из альтернативных последовательностей инструкций в коде условной инструкции.

Связанные условия:

Условная инструкция с несколькими альтернативными ветвями.

Вложенное условие:

Условная инструкция, которая расположена в одной из ветвей другой условной инструкции.

Инструкция `return`*:*

Инструкция, осуществляющая немедленное завершение функции и возвращение управления вызывающей стороне.

Рекурсия:

Процесс вызова функцией самой себя.

Базовый случай:

Условная ветвь в рекурсивной функции, которая не выполняет рекурсивный вызов.

Бесконечная рекурсия:

Рекурсия, которая не имеет базового случая или никогда не достигает его. В конце концов, бесконечная рекурсия вызывает ошибку выполнения (runtime error).

УПРАЖНЕНИЯ

Упражнение 5.1

Модуль `time` предоставляет функцию, также называемую `time()`, которая возвращает текущее время по Гринвичу в формате «эпохи» — отсчета от произвольно выбранного момента. В системах UNIX эпоха началась 1 января 1970 года.

```
>>> import time
>>> time.time()
1437746094.5735958
```

Напишите скрипт, который считывает текущее время и преобразует его в суточное время в часах, минутах и секундах, а также отображает количество дней, прошедших с начала эпохи.

Упражнение 5.2

Великая теорема Ферма гласит, что для любого натурального числа $n > 2$ уравнение:

$$a^n + b^n = c^n$$

не имеет решений в целых ненулевых числах a, b, c.

1. Напишите функцию `check_fermat()`, которая принимает четыре параметра — `a`, `b`, `c` и `n` — и проверяет, выполняется ли теорема Ферма. Если `n` больше `2` и

   $$a^n + b^n = c^n,$$

   в программе должно быть напечатано: «Не может быть, Ферма ошибся!» В противном случае программа должна вывести: «Нет, это не работает».
2. Напишите функцию, которая предлагает пользователю ввести значения для переменных a, b, c и n, преобразует их в целые числа и использует функцию `check_fermat()`, чтобы проверить, нарушают ли они теорему Ферма.

Упражнение 5.3

Если у вас три отрезка, из них, возможно, получится составить треугольник. Например, если один из отрезков имеет длину 12 см, а два других — по 1 см,

вы не сможете сделать так, чтобы короткие отрезки пересеклись. Для любых трех значений длины есть простой тест, позволяющий проверить, возможно ли построить треугольник:

> Если какая-либо из трех длин больше, чем сумма двух других, вы не сможете образовать треугольник. В противном случае сможете. (Если сумма двух длин равна третьей, они образуют так называемый «вырожденный» треугольник.)

1. Напишите функцию `is_triangle()`, которая принимает три целых числа в качестве аргументов и выводит либо «Да», либо «Нет» в зависимости от того, возможно ли образовать треугольник из отрезков с заданной длиной.
2. Напишите функцию, которая предлагает пользователю ввести три длины отрезка, преобразует их в целые числа и использует функцию `is_triangle()`, чтобы проверить, могут ли отрезки заданной длины образовать треугольник.

Упражнение 5.4

Каков результат выполнения следующей программы? Нарисуйте стековую диаграмму, которая показывает состояние программы на момент, когда она выводит результат.

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```

1. Что произойдет, если вызвать эту функцию следующим образом: `recurse(-1, 0)`?
2. Напишите строку документации, которая объясняет, что нужно знать для того, чтобы использовать эту функцию (и ничего лишнего).

В следующих упражнениях используется модуль черепашки, описанный в главе 4.

Упражнение 5.5

Прочитайте код показанной ниже функции и попробуйте выяснить, что она делает (см. примеры из главы 4). Затем запустите ее и посмотрите, верны ли ваши предположения.

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

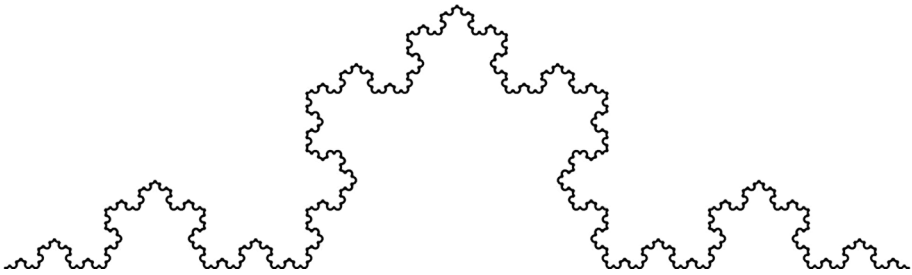


Рис. 5.2. Кривая Коха

Упражнение 5.6

Кривая Коха — это фрактал, который выглядит примерно так, как показано на рис. 5.2. Чтобы нарисовать кривую Коха с длиной *x*, все, что вам нужно сделать, это:

1. Нарисовать кривую Коха длиной *x*/3.
2. Повернуть налево на 60 градусов.
3. Нарисовать кривую Коха длиной *x*/3.
4. Повернуть направо на 120 градусов.
5. Нарисовать кривую Коха длиной *x*/3.
6. Повернуть налево на 60 градусов.
7. Нарисовать кривую Коха длиной *x*/3.

Исключение составляет случай, когда x меньше 3: в этом случае вы можете просто нарисовать прямую линию длиной *x*.

1. Напишите функцию `Koch()`, которая принимает в качестве параметров черепашку и значение длины и использует черепашку для построения кривой Коха с заданной длиной.

2. Напишите функцию `snowflake()`, которая рисует три кривые Коха, создавая контур снежинки.
   Решение: **thinkpython2.com/code/koch.py**.
3. Кривая Коха может быть обобщена несколькими способами. По адресу **ru.wikipedia.org/wiki/Кривая_Коха** вы найдете больше примеров и сможете реализовать наиболее понравившийся.

ГЛАВА 6
ФУНКЦИИ, ВОЗВРАЩАЮЩИЕ ЗНАЧЕНИЕ

Многие из функций Python, которые мы использовали, например математические, возвращают значения. Но все функции, которые мы писали до сих пор, были пустыми: они возвращают результат, такой как печать значения или перемещение черепашки, но у них нет возвращаемого значения. В этой главе вы научитесь писать **функции, возвращающие значение**.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

Вызов функции генерирует возвращаемое значение, которое мы обычно присваиваем переменной или используем как часть выражения.

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

Все функции, которые мы написали до этого, были пустыми. Проще говоря, они не имеют возвращаемого значения; а еще точнее, их возвращаемое значение — `None`.

В этой главе мы (наконец) будем писать функции, возвращающие значение. Первый пример — это функция `area()`, которая возвращает площадь круга с заданным радиусом:

```
def area(radius):
    a = math.pi * radius**2
    return a
```

Мы уже встречались с `return`, но в функции, возвращающей значение, инструкция `return` содержит выражение. Этот код означает: «Немедленно вернись из этой функции и используй следующее выражение в качестве возвращаемого значения». Выражение может быть сколь угодно сложным, поэтому мы могли бы записать эту функцию более кратко:

```
def area(radius):
    return math.pi * radius**2
```

С другой стороны, **временные переменные (temporary variables)** наподобие `a` могут облегчить отладку.

Иногда полезно иметь несколько инструкций `return`, по одному в каждой ветви условной конструкции:

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

Поскольку эти инструкции `return` находятся в альтернативных условных ветвях, выполняется только одна из них.

Как только `return` выполнена, функция завершается и не выполняет никакие последующие инструкции. Код, который находится после инструкции `return` или любой другой позиции, куда порядок выполнения никогда не сможет привести, называется **мертвым кодом (dead code)**.

При разработке функции, возвращающей значение, полезно убедиться, что каждый возможный вариант развития событий заканчивается инструкцией `return`. Например:

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

Эта функция некорректна, так как если переменной `x` присвоено значение `0`, ни одно из условий не выполнится, и функция завершится без вызова инструкции `return`. Если поток выполнения достигает конца функции, возвращаемое значение — `None`, что не является абсолютным значением нуля:

```
>>> absolute_value(0)
None
```

Кстати, Python предоставляет встроенную функцию `abs()`, которая вычисляет абсолютные значения.

В качестве упражнения напишите функцию сравнения, которая принимает два значения, `x` и `y`, и возвращает `1`, если `x > y`, `0`, если `x == y`, и `-1`, если `x < y`.

ПОШАГОВАЯ РАЗРАБОТКА

Если вы пишете сложные функции, отладка отнимает уйму сил и времени.

Чтобы справляться с усложняющейся программой, вы можете попробовать процесс **пошаговой разработки (incremental development)**. Цель пошаговой разработки — избежать длительных сеансов отладки, добавляя и тестируя только небольшое количество кода за раз.

В качестве примера предположим, что вы хотите найти расстояние между двумя точками, заданными координатами (`x`$_1$, `y`$_1$) и (`x`$_2$, `y`$_2$). По теореме Пифагора расстояние составляет:

$$\text{Расстояние} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Первый шаг — рассмотреть, как должна выглядеть функция расстояния (`distance()`) в Python. Другими словами, каковы входные данные (параметры) и каковы выходные данные (возвращаемое значение)?

В этом случае входные данные — две точки, которые вы можете представить, используя четыре числа. Возвращаемое значение — это расстояние, представленное значением с плавающей точкой.

Сразу же вы можете написать каркас функции:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Очевидно, что эта версия не рассчитывает расстояние; функция всегда возвращает ноль. Но она синтаксически верна и она работает, а это значит, что вы можете проверить ее, прежде чем сделать что-то более сложное.

Чтобы проверить новую функцию, вызовите ее с примерами аргументов:

```
>>> distance(1, 2, 4, 6)
0.0
```

Я выбрал эти значения так, чтобы горизонтальное расстояние было 3, а вертикальное — 4. Таким образом, результат будет равен 5, гипотенуза треугольника со сторонами 3–4–5.

При тестировании функции полезно знать правильный ответ.

На данный момент мы подтвердили, что функция синтаксически верна и мы можем добавлять код в тело. Следующий разумный шаг — определить разницу между $x_2 - x_1$ и $y_2 - y_1$. Следующая версия функции сохраняет эти значения во временных переменных и печатает их:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('значение dx равно:', dx)
    print('значение dy равно:', dy)
    return 0.0
```

Если функция работает, она должна отображать значение переменной `dx`, равное `3`, и значение переменной `dy`, равное `4`. Если это так, мы знаем, что функция получает правильные аргументы и выполняет первые вычисления правильно. Если нет, то надо проверить всего несколько строк.

Далее мы вычисляем сумму квадратов значений переменных `dx` и `dy`:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('сумма квадратов равна: ', dsquared)
    return 0.0
```

Опять же, нужно запустить программу на этом этапе и проверить вывод (который должен быть представлен числом `25`). Наконец, вы можете использовать функцию `math.sqrt()`, чтобы вычислить окончательное значение и вернуть его:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

Если все работает, то мы выполнили поставленную задачу. В противном случае вы можете напечатать значение результата перед инструкцией `return`.

Окончательная версия функции ничего не отображает при запуске; она только возвращает значение. Написанные нами инструкции печати полезны для отладки, но как только вы задействуете функцию в проекте, нужно удалить их. Такой код называется **отладочным (scaffolding)**, потому что он полезен при разработке программы, но не является частью конечного продукта.

Начиная программировать, вы должны добавлять по одной или две строки кода за раз. По мере накопления опыта вы можете писать и отлаживать

бóльшие фрагменты кода. В любом случае пошаговая разработка поможет сэкономить много времени на отладке.

Ключевые аспекты процесса:

1. Начинайте с рабочей версии программы и вносите небольшие изменения. В любой момент, если есть ошибка, нужно четко представлять, где она возникает.
2. Используйте переменные для хранения промежуточных значений, чтобы вы могли отображать и проверять их.
3. Когда программа заработает, вы можете удалить некоторые отладочные фрагменты кода или объединить несколько инструкций в составные выражения, но только если это не затруднит чтение кода программы.

В качестве упражнения в пошаговом режиме напишите функцию `hypotenuse()`, которая возвращает длину гипотенузы прямоугольного треугольника с учетом длины двух катетов, которые передаются в качестве аргументов. Фиксируйте каждый шаг процесса разработки по мере продвижения.

КОМПОЗИЦИЯ

Как вы уже знаете, одну функцию можно вызывать из другой. В качестве примера мы напишем функцию, которая берет две точки, центр круга и точку по периметру, после чего вычисляет площадь круга.

Предположим, что центральная точка хранится в качестве значений переменных `xc` и `yc`, а точка периметра — в переменных `xp` и `yp`. Первый шаг — найти радиус круга, то есть расстояние между двумя точками. И мы только что написали функцию, `distance()`, которая делает это:

```
radius = distance(xc, yc, xp, yp)
```

Следующий шаг — найти площадь круга с этим радиусом; мы только что написали и это тоже:

```
result = area(radius)
```

Инкапсулируя эти шаги в функцию, получаем:

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

Временные переменные `result` и `radius` полезны для разработки и отладки, но как только программа заработает, мы можем сократить ее код, объединив вызовы функций:

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

ЛОГИЧЕСКИЕ ФУНКЦИИ

Функции могут возвращать логические значения, что позволяет без лишних усилий скрывать сложные тесты внутри функций. Например:

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

Логическим функциям принято присваивать имена, которые звучат как вопросы да/нет; так, функция `is_divisible()` возвращает `True` или `False`, показывая, делится ли значение переменной `x` на `y`.

Ниже представлен пример:

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

Результат оператора `==` — логическое значение, поэтому мы можем оформить функцию более лаконично, возвращая напрямую это значение:

```
def is_divisible(x, y):
    return x % y == 0
```

Логические функции часто используются в условных инструкциях:

```
if is_divisible(x, y):
    print('x делится на y')
```

Заманчиво написать что-то вроде:

```
if is_divisible(x, y) == True:
    print('x делится на y')
```

Но дополнительное сравнение не обязательно.

В качестве упражнения напишите функцию `is_between(x, y, z)`, которая возвращает `True`, если истинно выражение $x \leq y \leq z$, и `False` в противном случае.

БОЛЬШЕ РЕКУРСИИ

Мы изучили Python лишь немного, но, представьте себе, это уже *полноценный* курс языка программирования, то есть полученных знаний хватит, чтобы произвести все вычисления. Любая когда-либо написанная программа может быть переписана с помощью только тех функций, которые вы уже изучили (на самом деле вам понадобится еще несколько команд для управления такими устройствами, как мышь, носители информации и тому подобное, но на этом всё).

Доказательство этого — нетривиальное упражнение, впервые выполненное Аланом Тьюрингом, одним из первых ИТ-специалистов (некоторые утверждают, что он был математиком, но многие первые ученые-информатики начинали как математики). Соответственно, оно известно как тезис Тьюринга. Для более полного (и точного) обсуждения тезиса Тьюринга я рекомендую прочитать книгу Майкла Сипсера Introduction to the Theory of Computation (Course Technology, 2012).

Чтобы разобраться с возможностями изученных инструментов, мы оценим несколько рекурсивно определенных математических функций. Рекурсивное определение аналогично тавтологии в том смысле, что определение содержит ссылку на определяемую вещь. Действительно тавтологическое определение не очень полезно:

*стрижающий**:

Прилагательное используется для описания чего-то стрижающего.

Увидев такое определение в словаре, вы вероятно пришли бы в замешательство. С другой стороны, если вы посмотрите на определение функции факториала, обозначенной символом *!*, вы увидите следующее:

$0! = 1$
$n! = n\ (n - 1)!$

Это определение говорит, что факториал 0 равен 1, а факториал любого другого значения, *n*, равен произведению *n* и факториала *n*–1.

* Неологизм, придуманный Льюисом Кэрроллом. В оригинале: vorpal (sword или blade) — стрижающий (или вострый) меч. *Прим. пер.*

Итак, *3!* — это 3 раза *2!*, что в 2 раза *1!*, что в 1 раз *0!*. Собираем всё вместе: *3!* равно 3 умножить на 2 умножить на 1 умножить на 1, что составляет 6.

Если вы можете написать рекурсивное определение чего-либо, вы можете написать программу на Python для его вычисления. Первый шаг — решить, какими должны быть параметры. В данном случае очевидно, что функция `factorial()` принимает целое число:

```
def factorial(n):
```

Если аргумент равен `0`, всё, что нужно сделать, — это вернуть `1`:

```
def factorial(n):
    if n == 0:
        return 1
```

В противном случае, и это интересная часть, мы должны сделать рекурсивный вызов, чтобы найти факториал *n*–1, а затем умножить его на n:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

Порядок выполнения этой программы аналогичен порядку обратного отсчета в разделе «Рекурсия» главы 5. Если мы вызовем функцию `factorial()` со значением `3`:

Поскольку `3` не равно `0`, мы берем вторую ветвь и вычисляем факториал `n-1`...
  Поскольку `2` не равно `0`, мы берем вторую ветвь и вычисляем факториал `n-1`...
    Поскольку `1` не равно `0`, мы берем вторую ветвь и вычисляем факториал `n-1`...
      Поскольку `0` равно `0`, мы берем первую ветвь и возвращаем `1`, не делая более рекурсивных вызовов.
    Возвращаемое значение `1` умножается на `n`, равное `1`, и результат возвращается.
  Возвращаемое значение `1` умножается на `n`, равное `2`, и результат возвращается.
Возвращаемое значение (`2`) умножается на `n`, которое равно `3`, а результат `6` становится возвращаемым значением вызова функции, которая запустила весь процесс.

На рис. 6.1 показано, как выглядит стековая диаграмма для этой последовательности вызовов функций.

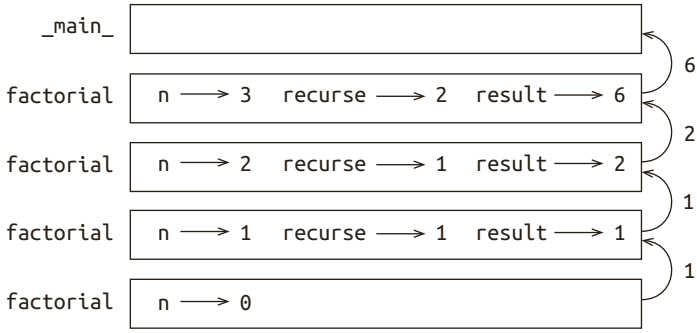


Рис. 6.1. Стековая диаграмма

Возвращаемые значения показаны как переданные обратно в стек. В каждом фрейме возвращаемое значение — это значение `result`, которое является произведением `n` и `recurse`.

В последнем фрейме локальные переменные `recurse` и `result` не существуют, потому что ветвь, в которой они создаются, не выполняется.

СЛЕПАЯ ВЕРА

Следование порядку выполнения — один из способов чтения программ, но он может быстро надоесть. Альтернатива ему в том, что я называю «слепая вера». Когда вы доходите до вызова функции, вместо того чтобы следовать порядку выполнения, вы *предполагаете*, что функция работает правильно и возвращает правильный результат.

На самом деле вы уже практикуете слепую веру, когда используете встроенные функции. Когда вы вызываете функции `math.cos()` или `math.exp()`, вы не проверяете тело этих функций. Вы просто предполагаете, что они работают, потому что люди, которые написали встроенные функции, были хорошими программистами.

То же самое происходит, когда вы вызываете одну из собственных функций. Например, в разделе «Логические функции» в этой главе мы написали функцию `is_divisible()`, которая определяет, делится ли одно число на другое. Как только мы убедимся в том, что эта функция верна — изучив код и протестировав, — мы сможем использовать эту функцию, не глядя вновь на ее тело.

То же самое относится и к рекурсивным программам. Когда вы добираетесь до рекурсивного вызова, вместо того чтобы следовать порядку выполнения, вы должны предположить, что рекурсивный вызов работает (возвращает правильный результат), а затем спросить себя: «Предполагая, что я могу найти факториал *n*–1, могу ли я вычислить факториал *n*?» Очевидно, что можете, просто умножив на *n*.

Конечно, немного странно предполагать, что функция работает правильно, когда вы еще не написали ее, но именно поэтому это называется слепой верой!

ЕЩЕ ОДИН ПРИМЕР

После факториала наиболее распространенным примером рекурсивно определенной математической функции могут служить числа Фибоначчи, которые имеют следующее определение (см. **https://ru.wikipedia.org/wiki/Числа_Фибоначчи**[*]):

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(*n*) = fibonacci(*n* − 1) + fibonacci(*n* − 2)

В переводе на Python это выглядит так:

```
def fibonacci (n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Если вы попытаетесь проследить здесь порядок выполнения, даже для небольших значений *n*, ваша голова взорвется. Но согласно принципу слепой веры, если вы предполагаете, что два рекурсивных вызова работают правильно, тогда становится ясно, что вы получите правильный результат, сложив их вместе.

[*] Последовательность Фибоначчи начинается с 0 и 1, а далее в ней каждое следующее число является результатом сложения двух предыдущих. *Прим. ред.*

ПРОВЕРКА ТИПОВ

Что произойдет, если мы вызовем функцию `factorial()` и передадим ей значение `1.5` в качестве аргумента?

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

Это похоже на бесконечную рекурсию. Как такое может быть?! Функция имеет базовый случай `n == 0`. Но, если `n` — не целое число, мы можем *пропустить* базовый случай, и рекурсия станет бесконечной.

В первом рекурсивном вызове значение `n` равно `0.5`. В следующем — `-0.5`. Дальше оно становится меньше (дальше от нуля), но никогда не будет равно `0`.

У нас есть два выхода. Мы можем попытаться обобщить функцию `factorial()` для работы с числами с плавающей точкой или мы можем выполнить проверку типов аргументов этой функции. Первый вариант называется гамма-функцией и чуточку выходит за рамки этой книги. Так что мы пойдем вторым путем.

Мы можем использовать встроенную функцию `isinstance()`, чтобы проверить тип аргумента. Мы также можем убедиться, что аргумент — положительное число:

```
def factorial (n):
    if not isinstance(n, int):
        print('Факториал определяется только для целых чисел.')
        return None
    elif n < 0:
        print('Факториал не определяется для отрицательных целых чисел.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Первый случай обрабатывает нецелые числа; второй обрабатывает отрицательные целые числа. В обоих случаях программа выводит сообщение об ошибке и возвращает значение `None`, чтобы указать, что что-то пошло не так:

```
>>> factorial('fred')
Факториал определяется только для целых чисел.
None
>>> factorial(-2)
Факториал не определяется для отрицательных целых чисел.
None
```

Если мы пройдем обе проверки, то узнаем, что значение переменной `n` положительно или равно нулю, поэтому можем доказать, что рекурсия завершится.

Эта программа демонстрирует шаблон, который иногда называют **защитником (guardian)**. Первые два условия действуют как защитники и охраняют код, который зависит от значений, способных вызвать ошибку. Защитники позволяют убедиться, что код корректный.

В разделе «Обратный поиск» главы 11 мы увидим более гибкую альтернативу вывода сообщения об ошибке: вызов исключения.

ОТЛАДКА

Разбиение большой программы на более мелкие функции создает естественные контрольные точки для отладки. Если функция не работает, то существует три варианта.

— Что-то не так с аргументами, которые функция получает; нарушение предварительного условия.
— Что-то не так с самой функцией; нарушено выходное условие.
— Что-то не так с возвращаемым значением или способом его использования.

Чтобы исключить первый случай, вы можете добавить инструкцию `print` в начале функции и отобразить значения параметров (и, возможно, их типы). Или вы можете написать код, который явно проверяет выполнение предварительных условий.

Если параметры допустимы, добавьте инструкцию `print` перед каждой инструкцией `return` и отобразите возвращаемое значение. Если возможно, проверьте результат вручную.

Попробуйте вызвать функцию со значениями, которые упрощают проверку результата (см. раздел «Пошаговая разработка» в начале этой главы).

Если функция работает, взгляните на ее вызов, чтобы убедиться, что возвращаемое значение используется правильно (и используется ли вообще!).

Добавление инструкций `print` в начале и конце функции может помочь отследить порядок выполнения. Например, ниже представлена версия функции `factorial()` с инструкциями `print`:

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'факториал', n)
```

```
    if n == 0:
        print(space, 'возвращает 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'возвращает', result)
        return result
```

Переменная space — это строка из пробельных символов, которая управляет отступом при выводе данных. Ниже показан результат вызова функции `factorial(4)`:

```
            факториал 4
         факториал 3
      факториал 2
   факториал 1
факториал 0
возвращает 1
   возвращает 1
      возвращает 2
         возвращает 6
            возвращает 24
```

Если вы запутались в порядке выполнения, такой вид вывода может сильно помочь. Он отнимает дополнительное время при разработке, но ускоряет отладку.

СЛОВАРЬ ТЕРМИНОВ

Временная переменная:
Переменная, используемая для хранения промежуточного значения в сложном вычислении.

Мертвый код:
Часть программы, которая никогда не выполняется, часто потому, что она указана после инструкции `return`.

Пошаговая разработка:
Способ разработки программы, минимизирующий стадии отладки путем добавления и тестирования небольшого объема кода раз за разом.

Отладочный код:
Код, который используется при разработке программы, но не является частью окончательной версии.

Защитник:

Шаблон проектирования, где есть условная инструкция для проверки и обработки условий, которые могут привести к ошибке.

УПРАЖНЕНИЯ

Упражнение 6.1

Нарисуйте стековую диаграмму для следующей программы. Что выводит программа?

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z  square = b(total)**2
    return square

x = 1
y = x + 1
print(c(x, y+3, x+y))
```

Упражнение 6.2

Функция Аккермана, $A(m, n)$, определена следующим образом:

$$A(m,\ n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \quad n = 0 \\ A\big(m-1,\ A(m,\ n-1,1)\big) & \text{if } m > 0 \quad n > 0 \end{cases}$$

О функции читайте: **https://ru.wikipedia.org/wiki/Функция_Аккермана**. Напишите функцию `ack()`, которая вычисляет функцию Аккермана. Используйте функцию для вычисления `ack(3, 4)`, результат должен быть равен `125`. Что происходит при бóльших значениях `m` и `n`?

Решение: **thinkpython2.com/code/ackermann.py**.

Упражнение 6.3

Палиндром — слово, которое пишется одинаково справа налево и слева направо, например «топот» и «шалаш». Рекурсивное определение: слово —

это палиндром, если первая и последняя буквы совпадают, а между ними — палиндром.

Ниже приведены функции, которые принимают строковый аргумент и возвращают первую и последнюю буквы, а также буквы между ними:

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

Мы разберем, как они работают, в главе 8.

1. Сохраните эти функции в файл с именем *palindrome.py* и протестируйте их. Что произойдет, если вызвать функцию `middle()` со строковым значением из двух букв? Одной буквы? Как насчет пустой строки, которая определяется как `''` и не содержит букв?
2. Напишите функцию `is_palindrome()`, которая принимает строковый аргумент и возвращает `True`, если это палиндром, и `False` в противном случае. Помните, что вы можете использовать встроенную функцию `len()`, чтобы проверить длину строки.

Решение: **thinkpython2.com/code/palindrome_soln.py**.

Упражнение 6.4

Число *a* является степенью *b*, если оно делится на *b* и *a*/*b* является степенью *b*. Напишите функцию `is_power()`, которая принимает параметры *a* и *b* и возвращает `True`, если *a* является степенью *b*. Примечание: вам придется подумать над базовым случаем.

Упражнение 6.5

Наибольший общий делитель (greatest common divisor, gcd) *a* и *b* — это наибольшее число, на которое оба числа делятся без остатка.

Один способ найти наибольший общий делитель двух чисел основан на наблюдении, что если *r* является остатком при делении *a* на *b*, то $gcd(a, b) = gcd(b, r)$. В качестве базового случая мы можем использовать $gcd(a, 0) = a$.

Напишите функцию `gcd()`, которая принимает параметры `a` и `b` и возвращает их наибольший общий делитель.

Примечание. Это упражнение основано на примере из книги Structure and Interpretation of Computer Programs Абельсона и Суссмана (MIT Press, 1996).

ГЛАВА 7
ИТЕРАЦИИ

Эта глава посвящена итерациям, то есть многократному выполнению блока инструкций. Вы видели своего рода итерацию с использованием рекурсии в разделе «Рекурсии» главы 5. Вы видели и другой вариант, с использованием цикла `for`, в разделе «Простое повторение» главы 4. В этой главе вы увидите третий вариант, с использованием инструкции `while`. Но сначала я хочу рассказать чуть больше о присваивании значений переменным.

ПЕРЕНАЗНАЧЕНИЕ

Как вы, возможно, заметили, одной переменной можно несколько раз присваивать значения. После присваивания переменная перестает ссылаться на предыдущее значение и ссылается на новое.

```
>>> x = 5
>>> x 5
>>> x = 7
>>> x 7
```

Когда мы в первый раз печатаем значение переменной `x`, оно равно `5`; во второй раз ее значение равно 7 (так как мы переназначили его операцией `x = 7`).

На рис. 7.1 показано, как выглядит переназначение на диаграмме состояний.

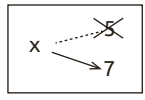


Рис. 7.1. Диаграмма состояний

Сейчас я хочу остановиться на одном моменте, вызывающем большую путаницу. В языке Python знак равенства (=) используется для присваивания,

поэтому заманчиво интерпретировать оператор типа `a = b` как математическое утверждение равенства; то есть утверждение, что `a` и `b` равны. Но это категорически неправильно.

Во-первых, равенство — это симметричные отношения, а присвоение — нет. Например, в математике, если $a = 7$, то $7 = a$. Но в Python утверждение `a = 7` допустимо, а `7 = a` — нет.

Кроме того, в математике утверждение о равенстве либо истинно, либо ложно — и это неизменно. Если $a = b$ сейчас, то a всегда будет равно b. В Python инструкция присваивания может сделать две переменные равными, но они не обязаны оставаться такими:

```
>>> a = 5
>>> b = a # a и b сейчас равны
>>> a = 3 # a и b больше не равны
>>> b 5
```

Третья строка изменяет значение переменной `a`, но не меняет значение переменной `b`, поэтому они больше не равны.

Переназначение переменных полезно, но будьте осторожны. Если значения переменных часто меняются, может усложниться чтение и отладка кода.

ОБНОВЛЕНИЕ ПЕРЕМЕННЫХ

Распространенный способ переназначения переменных — обновление, при котором новое значение переменной зависит от старого.

```
>>> x = x + 1
```

Данный код означает «получить текущее значение переменной `x`, добавить единицу и затем присвоить переменной `x` новое значение».

Если вы попытаетесь обновить переменную, которая не существует, вы получите ошибку, потому что Python вычисляет правую сторону, прежде чем он присваивает значение переменной `x`:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Прежде чем вы сможете обновить переменную, вы должны **инициализировать (initialize)** ее, обычно путем простого присваивания значения:

```
>>> x = 0
>>> x = x + 1
```

Обновление переменной путем увеличения на 1 называется **инкрементированием (increment)**; вычитание 1 называется **декрементированием (decrement)**.

ИНСТРУКЦИЯ WHILE

Компьютеры часто используют для автоматизации повторяющихся задач. Выполнять одинаковые или похожие задачи без ошибок — это то, что компьютеры выполняют хорошо, а люди — плохо. В компьютерной программе повторение также называется **итерацией (iteration)**.

Вы уже видели две функции, `countdown()` и `print_n()`, которые выполняют итерацию с использованием рекурсии. Поскольку итерации очень распространены, Python предоставляет встроенные конструкции, чтобы упростить их. Одна из них — инструкция `for`, которую вы видели в разделе «Простое повторение» главы 4. Мы к ней еще вернемся.

Еще одна инструкция — `while`. Ниже представлена версия функции `countdown()`, в которой используется инструкция `while`:

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Готово!')
```

Вы можете прочитать инструкцию `while` вот так: «пока значение переменной `n` больше `0`, выводить значение переменной `n` и затем уменьшать его на `1`. Когда значение переменной станет равно `0`, вывести `Готово!`»

Более формально ниже представлен порядок выполнения инструкции `while`.

1. Определить, истинно или ложно условие.
2. Если ложно, выйти из инструкции `while` и продолжить выполнение программы со следующей строки кода.
3. Если условие истинно, выполнить тело инструкции и вернуться к шагу 1.

Такой тип порядка выполнения называется **циклом**, потому что третий шаг возвращает к началу.

Тело цикла должно изменять значение одной или нескольких переменных, чтобы в итоге условие стало ложным и цикл завершился. В противном случае

цикл будет повторяться вечно, формируя так называемый **бесконечный цикл (infinite loop)**.

С точки зрения программистов, инструкция на шампуне «намылить, смыть, повторить» — самый что ни на есть бесконечный цикл.

В случае обратного отсчета мы можем доказать, что цикл завершается: если значение переменной `n` равно нулю или отрицательному числу, то цикл никогда не запустится. В противном случае значение переменной `n` уменьшается на каждом шаге, поэтому в итоге мы должны получить `0`.

Для некоторых других циклов все не так просто. Например:

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:    # n = четное значение
            n = n / 2
        else:             # n = нечетное значение
            n = n*3 + 1
```

Условием для этого цикла является код `n != 1`, поэтому цикл будет выполняться до тех пор, пока значение переменной `n` не станет равным `1`, то есть условие не станет ложным.

Каждый раз в цикле программа выводит значение переменной `n`, а затем проверяет, четное оно или нечетное. Если оно четное, значение переменной `n` делится на `2`. Если нечетное — значение переменной `n` заменяется на `n * 3 + 1`. Например, если аргумент, переданный в функцию `sequence()`, равен `3`, результирующие значения переменной `n` равны `3`, `10`, `5`, `16`, `8`, `4`, `2`, `1`.

Поскольку значение переменной `n` иногда увеличивается, а иногда уменьшается, нет очевидных доказательств, что оно когда-либо достигнет `1` или что программа завершится. Для некоторых конкретных значений переменной `n` мы можем доказать завершение. Например, если начальное значение представлено степенью двойки, значение переменной `n` будет каждый раз проходить через цикл, пока не достигнет `1`. Предыдущий пример заканчивается такой последовательностью, начиная с `16`.

Сложнее вопрос, можем ли мы доказать, что эта программа завершается для всех положительных значений переменной `n`. До сих пор никто не смог доказать или опровергнуть это! (Больше подробностей: **ru.wikipedia.org/wiki/Гипотеза_Коллатца**.)

В качестве упражнения перепишите функцию `print_n()` из раздела «Рекурсия» главы 5, используя итерации вместо рекурсии.

ИНСТРУКЦИЯ BREAK

Иногда вы не знаете, что пора заканчивать цикл, пока не пройдете половину тела. Тогда инструкция `break` поможет выйти из цикла.

Например, предположим, что вы хотите принимать данные от пользователя, пока он не введет «`готово`». Вы могли бы реализовать это так:

```
while True:
    line = input('> ')
    if line == 'готово':
        break
    print(line)

print('Готово!')
```

Условие цикла — `True`, что всегда истинно, поэтому цикл выполняется до тех пор, пока не будет достигнута инструкция `break`.

Во время каждой итерации пользователю выводится приглашение к вводу в виде угловой скобки. Если пользователь вводит слово `готово`, инструкция `break` завершает цикл. В противном случае программа повторяет все, что пользователь вводит, и возвращается к началу цикла. Испытаем:

```
> не готово
не готово
> готово
Готово!
```

Использовать `while` – распространенный способ разработки циклов, потому что вы можете проверить условие в любом месте цикла (не только сверху) и выразить условие остановки в утвердительной форме («остановите, когда это произойдет»), а не в отрицательной («продолжайте до тех пор, пока это не произойдет»).

КВАДРАТНЫЕ КОРНИ

Циклы часто используют в программах для получения числовых результатов и начинают с приблизительного ответа, итеративно улучшая его.

Например, один из способов вычисления квадратных корней — метод Ньютона. Предположим, что вы хотите вычислить квадратный корень из a. Если вы начнете практически с любого приблизительного значения x, вы можете вычислить наилучшее приблизительное значение по следующей формуле:

$$y = \frac{x + a / x}{2}.$$

Например, если *a* равно 4, а *x* равно 3:

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y 2.16666666667
```

Результат приближен к правильному ответу ($\sqrt{4} = 2$). Если мы повторим процесс с новым значением, результат станет еще ближе:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

После нескольких итераций результат практически точен:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

В общем, нельзя сказать, сколько шагов нужно, чтобы получить правильный ответ, но мы узнаем его, когда достигнем, потому что приблизительное значение перестает меняться:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y

2.0
```

Когда `y == x`, мы можем остановиться. Ниже представлен цикл, который стартует с начального значения переменной `x` и улучшает результат до тех пор, пока он не перестанет меняться:

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

Для большинства значений данный прием работает нормально, но сравнение чисел с плавающей точкой иногда дает неверный результат. Значения с плавающей точкой только приблизительные: большинство рациональных чисел, таких как ⅓, и иррациональных чисел, таких как $\sqrt{2}$, не могут быть точно представлены значением с плавающей точкой.

Вместо того чтобы проверять точное равенство значений переменных `x` и `y`, надежнее использовать встроенную функцию `abs()` для вычисления абсолютного значения разности между значениями:

```
if abs(y-x) < epsilon:
    break
```

где `epsilon` имеет значение примерно `0.0000001`, которое определяет, достаточно ли приближено значение.

АЛГОРИТМЫ

Метод Ньютона — это **алгоритм**: способ решения определенной категории задач (в данном случае — вычисления квадратных корней).

Чтобы понять, что такое алгоритм, можно начать с чего-то, не являющегося алгоритмом. Когда вы учились умножать однозначные числа, вы, вероятно, зубрили таблицу умножения. По сути, вы запомнили 100 конкретных решений. Такие знания не являются алгоритмическими.

Но если бы вы были ленивыми, вы могли бы выучить несколько трюков. Например, чтобы найти произведение n и 9, вы можете написать n–1 в качестве первой цифры и 10–n в качестве второй цифры. Этот трюк является общим решением для умножения любого однозначного числа на 9. Это алгоритм!

Точно так же методы, которые вы изучили для сложения с переносом, вычитанием с заимствованием и длинным делением, — это алгоритмы. Отличительная черта алгоритмов в том, что для их выполнения не требуется никакого интеллекта. Это механические процессы, где каждый следующий шаг следует из предыдущего в соответствии с простым набором правил.

Выполнение алгоритмов скучно, но их разработка интересна, интеллектуально сложна и является центральной частью информатики.

Некоторые вещи, которые даются людям естественно и легко, сложнее всего выразить с помощью алгоритма. Понимание естественного языка — отличный пример. Мы все делаем это, но до сих пор никто не смог объяснить как, по крайней мере не в форме алгоритма.

ОТЛАДКА

Когда вы начнете писать большие программы, отладка будет занимать все больше времени. Чем больше кода, тем выше вероятность ошибиться и больше места для багов.

Один из способов сократить время отладки — «отладка делением пополам». Например, если в вашей программе 100 строк и вы проверяете их по одной, это займет 100 шагов.

Вместо этого попробуйте разбить задачу пополам. В середине программы или где-то рядом с ней, найдите промежуточное значение, которое вы можете проверить. Добавьте инструкцию `print` (или что-то еще, что можно проверить) и запустите программу.

Если проверка средней точки неверна, проблема должна быть в первой половине программы. Если результат верный — проблема во второй половине.

Каждый раз, когда вы выполняете такую проверку, вы вдвое сокращаете количество строк, среди которых стоит искать ошибку. После шести шагов (что меньше 100) у вас останется одна или две строки кода теоретически.

На практике не всегда понятно, что такое «середина программы», и не всегда возможно проверить именно там. Не имеет смысла считать строки и искать точную середину. Вместо этого подумайте о моментах в программе, где вероятны ошибки, и местах, где легко сделать проверку. Затем выберите место, где, по вашему мнению, шансы на ошибку до и после примерно одинаковы.

СЛОВАРЬ ТЕРМИНОВ

Переназначение:

Присвоение нового значения переменной, которая уже существует.

Обновление:

Присвоение, при котором новое значение переменной зависит от старого.

Инициализация:

Присвоение, которое задает начальное значение переменной, которое позже может быть обновлено.

Инкремент:

Обновление, которое увеличивает значение переменной (часто на единицу).

Декремент:

Обновление, которое уменьшает значение переменной.

Итерация:

Многократное выполнение набора инструкций с использованием либо рекурсивного вызова функции, либо цикла.

Бесконечный цикл:

Цикл, в котором условие завершения никогда не выполняется.

Алгоритм:

Универсальный процесс решения определенной категории задач.

УПРАЖНЕНИЯ

Упражнение 7.1

Скопируйте цикл из раздела «Квадратные корни» этой главы и инкапсулируйте его в функцию `mysqrt()`, которая принимает `a` в качестве параметра, выбирает разумное значение `x` и возвращает результат вычисления квадратного корня из `a`.

Чтобы проверить ее, напишите функцию `test_square_root()`, которая печатает таблицу следующим образом:

```
a    mysqrt(a)      math.sqrt(a)    diff
-    ---------      ------------    ----
1.0  1.0            1.0             0.0
2.0  1.41421356237  1.41421356237   2.22044604925e-16
3.0  1.73205080757  1.73205080757   0.0
4.0  2.0            2.0             0.0
5.0  2.2360679775   2.2360679775    0.0
6.0  2.44948974278  2.44948974278   0.0
7.0  2.64575131106  2.64575131106   0.0
8.0  2.82842712475  2.82842712475   4.4408920985e-16
9.0  3.0            3.0             0.0
```

Первый столбец — это число *a*; второй столбец — это квадратный корень, вычисленный с помощью функции `mysqrt()`; третий столбец — это квадратный корень, вычисленный с помощью функции `math.sqrt()`; четвертый столбец — абсолютная величина разницы между двумя результатами.

Упражнение 7.2

Встроенная функция `eval()` принимает строку и вычисляет ее с помощью интерпретатора Python.

Например:

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

Напишите функцию `eval_loop()`, которая итеративно запрашивает пользовательский ввод, принимает выражение, вычисляет его с помощью функции `eval()` и печатает результат.

Это должно продолжаться до тех пор, пока пользователь не введет слово `готово`, после чего функция возвращает значение последнего выражения, которое она вычислила.

Упражнение 7.3

Математик Сриниваса Рамануджан нашел бесконечный ряд, который можно использовать для вычисления приближенного значения $1/\pi$:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}.$$

Напишите функцию `estimate_pi()`, которая использует эту формулу для вычисления и возврата приближенного значения π. Вы должны использовать цикл `while` для вычисления слагаемых, пока последнее слагаемое не будет меньше, чем `1e-15` (так в Python обозначается степень 10^{-15}). Вы можете проверить результат, сравнив его со значением константы `math.pi`.

Решение: **thinkpython2.com/code/pi.py**.

ГЛАВА 8
СТРОКИ

Строки не похожи на целые числа, числа с плавающей точкой и логические значения. Строка — это **последовательность**, то есть упорядоченная коллекция значений. В этой главе вы узнаете, как получить доступ к символам, составляющим строку, и узнаете о некоторых методах работы со строками.

СТРОКА — ЭТО ПОСЛЕДОВАТЕЛЬНОСТЬ

Строка — это последовательность символов. Вы можете получить доступ к любому символу, используя квадратные скобки:

```
>>> fruit = 'банан'
>>> letter = fruit[1]
```

Вторая инструкция выбирает символ под номером 1 из значения переменной `fruit` и присваивает его переменной `letter`.

Выражение в квадратных скобках называется **индексом (index)**. Индекс указывает, какой символ в последовательности вы хотите вернуть (отсюда и название индекс = указатель).

Но результат вас удивит:

```
>>> letter
'а'
```

Для большинства людей первая буква в слове «банан» — это `б`, а не `а`. Но программисты знают, что индекс является смещением от начала строки, а смещение первой буквы равно нулю.

```
>>> letter = fruit[0]
>>> letter
'б'
```

Таким образом, б — это нулевая буква слова «банан», а — это первая буква, а н — это вторая буква и так далее.

В качестве индекса вы можете использовать выражение, которое содержит переменные и операторы:

```
>>> i = 1
>>> fruit[i]
'а'
>>> fruit[i+1]
'н'
```

Но значение индекса должно быть целым числом. В противном случае вы получите ошибку:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

ФУНКЦИЯ LEN()

Встроенная функция `len()` возвращает количество символов в строке:

```
>>> fruit = 'банан'
>>> len(fruit)
5
```

Чтобы получить последнюю букву строки, вы, возможно, захотите попробовать что-то вроде этого:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

Причина ошибки `IndexError` в том, что в слове `банан` нет буквы с индексом 5. Поскольку мы начали считать с нуля, то пять букв пронумерованы от 0 до 4. Чтобы получить последний символ, вы должны вычесть 1 из значения переменной `length`:

```
>>> last = fruit[length-1]
>>> last
'н'
```

Так же вы можете использовать отрицательные индексы, отсчет начинается с конца строки. Выражение `fruit[-1]` возвращает последнюю букву, `fruit[-2]` — вторую с конца и так далее.

ОБХОД ЭЛЕМЕНТОВ С ПОМОЩЬЮ ЦИКЛА FOR

Многие задачи требуют посимвольной обработки строки. Часто обработка начинается с начала, берется каждый символ по очереди, что-то с ним происходит, и так продолжается, пока не будет достигнут конец строки. Эта схема обработки называется **обходом (traversal)**. Один из способов написать обход — использовать цикл `while`:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

Этот цикл обходит строку и выводит каждую букву в отдельной строке. Условие цикла — `index < len(fruit)`, поэтому, когда значение переменной `index` равно длине строки, условие ложно, и тело цикла не выполняется. Последний символ, к которому осуществлялся доступ, имеет индекс `len(fruit)-1`, он и является последним символом в строке.

В качестве упражнения напишите функцию, которая принимает строку в качестве аргумента и отображает буквы в обратном порядке, по одной на строку.

Другой способ реализовать обход — с помощью цикла `for`:

```
for letter in fruit:
    print(letter)
```

В цикле каждый следующий символ в строке присваивается переменной `letter`. Цикл продолжается до тех пор, пока не останется ни одного символа.

В следующем примере показано, как использовать конкатенацию (сложение строк) и цикл `for` для вывода результата в алфавитном порядке. В книге Роберта Макклоски «Дорогу утятам» нескольких утят зовут так: Бряк, Вряк, Квяк, Кряк, Ляк, Мяк, Няк и Шмяк. Этот цикл выводит эти имена в следующем порядке:

```
prefixes = 'БВККЛМНШ'
suffix = 'ряк'

for letter in prefixes:
    print(letter + suffix)
```

Результат следующий:

```
Бряк
Вряк
Кряк
Кряк
Лряк
Мряк
Нряк
Шряк
```

Конечно, это не совсем верно, потому что в списке оказались два Кряка, а Ляк, Мяк и Няк написаны с ошибками. В качестве упражнения измените программу, чтобы исправить эти ошибки.

СРЕЗЫ СТРОК

Сегмент строки называется **срезом (slice)**. Выбор среза аналогичен выбору символа:

```
>>> s = 'Монти Пайтон'
>>> s[0:5]
'Монти'
>>> s[6:12]
'Пайтон'
```

Инструкция `[n:m]` возвращает часть строки начиная с *n*-го символа и заканчивая *m*-ным, первый включается в результат, а последний — нет. Это поведение не соответствует интуитивному пониманию, но может помочь представить, что индексы указывают на промежутки *между* символами, показанные на рис. 8.1.

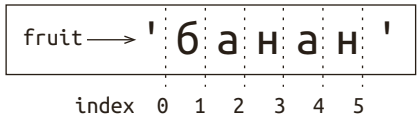


Рис. 8.1. Срезы индексов

Если вы опустите первый индекс (перед двоеточием), срез начнется с начала строки. Если вы опустите второй индекс, срез продолжится до конца строки:

```
>>> fruit = 'банан'
>>> fruit[:3]
```

```
'бан'
>>> fruit[3:]
'ан'
```

Если первый индекс больше или равен второму, результатом будет пустая строка, окруженная двумя кавычками:

```
>>> fruit = 'банан'
>>> fruit[3:3]
''
```

Пустая строка не содержит символов и имеет длину 0, но в остальном это такая же строка, как и любая другая.

Продолжая этот пример, как вы думаете, к какому результату приведет запрос `fruit[:]`? Попробуйте и изучите.

СТРОКИ — НЕИЗМЕНЯЕМЫЙ ТИП ДАННЫХ

Соблазнительно использовать оператор `[]` в левой части строки присваивания, чтобы изменить символ в строке. Например, так:

```
>>> greeting = 'Добрый день!'
>>> greeting[0] = 'Б'
TypeError: 'str' object does not support item assignment
```

В данном случае объект — это строка, а элемент — символ, который вы пытались назначить. Пока будем считать, что объект — это то же самое, что и значение, но мы уточним это определение позже (в разделе «Объекты и значения» главы 10).

Причина ошибки заключается в том, что строки — это неизменяемый тип данных, то есть вы не можете изменить существующую строку. Лучшее, что вы можете сделать, это создать новую строку — вариацию оригинала:

```
>>> greeting = 'Добрый день!'
>>> new_greeting = 'Б' + greeting[1:]
>>> new_greeting
'Бобрый день!'
```

Этот пример объединяет новую первую букву со срезом строки `greeting`. Исходная строка (переменная `greeting`) остается без изменений.

ПОИСК

Для чего предназначена эта функция?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
         index = index + 1
    return -1
```

В некотором смысле функция `find()` обратна инструкции `[]`. Вместо того чтобы брать индекс и извлекать соответствующий символ, он берет символ и находит индекс, где находится этот символ. Если символ не найден, функция возвращает `-1`.

Это первый пример инструкции `return` внутри цикла, который вы увидели. Если `word[index] == letter`, то функция выходит из цикла и немедленно возвращает управление.

Если символ не обнаружен в строке, программа выходит из цикла и возвращает `-1`.

Этот алгоритм — обход последовательности и возврат, когда мы находим то, что ищем, — называется **поиском (search)**.

В качестве упражнения измените функцию `find()` так, чтобы она имела третий параметр: индекс в слове (`word`), с которого она должен начать поиск.

ЦИКЛЫ И СЧЕТЧИКИ

Следующая программа подсчитывает, сколько раз буква `a` встречается в строке:

```
word = 'банан'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

Эта программа демонстрирует другой шаблон вычислений, называемый **счетчиком (counter)**. Переменная `count` инициализируется со значением `0`, а затем оно увеличивается каждый раз, когда обнаруживается буква `a`. Когда

цикл завершается, переменная count содержит результат — общее число букв a в строке.

В качестве упражнения инкапсулируйте этот код в функцию count() и обобщите ее так, чтобы она принимала строку и букву в качестве аргументов.

Затем перепишите функцию так, чтобы вместо обхода строки она использовала версию функции find() с тремя параметрами из предыдущего раздела.

СТРОКОВЫЕ МЕТОДЫ

Строки предоставляют методы для множества полезных операций. Метод похож на функцию — он принимает аргументы и возвращает значение, но его синтаксис отличается. Например, метод upper() принимает строку и возвращает новую строку со всеми прописными буквами.

Вместо синтаксиса функции upper(word) используется синтаксис метода word.upper():

```
>>> word = 'банан'
>>> new_word = word.upper()
>>> new_word
'БАНАН'
```

Такая форма точечной нотации требует имя метода, upper, и имя строки, к которой применяется метод, word. Пустые скобки указывают, что этот метод не принимает аргументов.

В этом случае, то есть в инструкции word.upper(), мы **вызываем** метод upper() объекта word.

Существует строковый метод find(), который удивительно похож на функцию, которую мы написали:

```
>>> word = 'банан'
>>> index = word.find('а')
>>> index
1
```

В этом примере мы вызываем поиск по слову и передаем искомую букву в качестве параметра.

На самом деле, метод find() более общий, чем наша функция; он может найти подстроки, а не только символы:

```
>>> word.find('на')
2
```

По умолчанию поиск начинается с начала строки, но данный метод может принимать второй аргумент, индекс, с которого поиск должен начинаться:

```
>>> word.find('ан', 3)
3
```

Это пример **необязательного аргумента (optional argument)**. Метод `find()` также может принимать и третий аргумент, индекс, на котором поиск должен остановиться:

```
>>> name = 'боб'
>>> name.find('б', 1, 2)
-1
```

Этот поиск завершается неудачно, потому что символ `б` не встречается в диапазоне индексов от 1 до 2, не включая 2. Поиск до второго индекса, но не включая его, делает метод `find()` согласующимся с операцией среза.

ОПЕРАТОР `IN`

Слово `in` — это логический оператор, который принимает две строки и возвращает `True`, если первая является подстрокой для второй:

```
>>> 'а' in 'банан'
True
>>> 'семя' in 'банан'
False
```

Например, показанная ниже функция печатает все буквы из аргумента `word1`, которые также есть в аргументе `word2`:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

Вы можете прочитать этот цикл так: «для (каждой) буквы в (первом) слове, если (эта) буква (встречается) во (втором) слове, напечатать (эту) букву».

Вот что вы получите, если сравнить яблоки и апельсины:

```
>>> in_both('апельсин', 'абрикос')
а
с
и
```

СРАВНЕНИЕ СТРОК

Операторы сравнения работают и со строками. Чтобы увидеть, равны ли две строки, нужно сделать следующее:

```
if word == 'банан':
    print('Шикарно, бананы.')
```

Другие операторы сравнения полезны для сортировки слов в алфавитном порядке:

```
if word < 'банан':
    print('Ваше слово, ' + word + ', располагается до слова банан.')
elif word > 'банан':
    print('Ваше слово, ' + word + ', располагается после слова банан.')
else:
    print('Шикарно, бананы.')
```

Python не воспринимает прописные и строчные буквы так, как это делают люди. В нем все прописные буквы предшествуют всем строчным, поэтому:

```
Ваше слово, Дыня, располагается до слова банан.
```

Распространенный способ решить эту проблему — преобразовать строки в стандартный формат, например во все строчные буквы, перед выполнением сравнения. Имейте это в виду, если придется иметь дело с человеком с дынями.

ОТЛАДКА

Когда вы используете индексы для обхода значений в последовательности, очень сложно получить корректные начальное и конечное значения индексов. Ниже показана функция, которая сравнивает два слова и возвращает значение `True`, если одно из слов обратно по отношению к другому, но она содержит две ошибки:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
```

```
        i = i+1
        j = j-1

    return True
```

Первая инструкция `if` проверяет, одинаковы ли слова по длине. Если нет, можно немедленно вернуть `False`. В противном случае можно утверждать, что слова одинаковы по длине, а значит, можно продолжить выполнение. Это пример использования шаблона защитника из раздела «Проверка типов» главы 6.

Переменные `i` и `j` — индексы: `i` проходит `word1` от начала до конца, а `j` проходит `word2` от конца до начала. Если найдутся две буквы, которые не совпадают, можно немедленно вернуть `False`. Если по завершении цикла все буквы совпали, возвращается `True`.

Если мы протестируем эту функцию на словах «порт» и «троп», ожидается, что вернется значение `True`, но появляется ошибка `IndexError`:

```
>>> is_reverse('порт', 'троп')
...
    File “reverse.py”, line 15, in is_reverse
        if word1[i] != word2[j]:
IndexError: string index out of range
```

Для отладки подобных ошибок первым делом я должен вывести значения индексов непосредственно перед строкой, где появляется ошибка.

```
        while j > 0:
            print(i, j)   # выводим индексы

        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
```

Теперь, когда я снова запускаю программу, я получаю больше информации:

```
>>> is_reverse('порт', 'троп')
0 4
...
IndexError: string index out of range
```

При первой итерации значение `j` равно 4, что выходит за пределы диапазона индексов строки `'порт'`. Индекс последнего символа равен 3, поэтому начальное значение для `j` должно быть `len(word2)-1`.

Если я исправлю эту ошибку и снова запущу программу, вот что я получу:

```
>>> is_reverse('порт', 'троп')
0 3
1 2
2 1
True
```

На этот раз мы получили правильный ответ, но похоже, что цикл выполнялся только трижды, что немного подозрительно. Чтобы лучше понять, что происходит, полезно нарисовать диаграмму состояний. Во время первой итерации фрейм для функции `is_reverse()` показан на рис. 8.2.

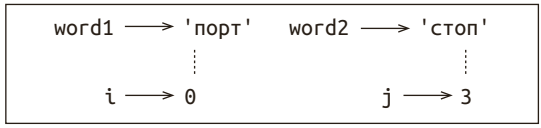


Рис. 8.2. Диаграмма состояний

Я расположил переменные во фрейме иначе и добавил пунктирные линии, чтобы показать, что значения `i` и `j` указывают на символы в `word1` и `word2`.

Начиная с этой диаграммы, выполните программу на бумаге, меняя значения `i` и `j` во время каждой итерации. Найдите и исправьте вторую ошибку в этой функции.

СЛОВАРЬ ТЕРМИНОВ

Объект:

Что-то, на что может ссылаться переменная. На данный момент вы можете использовать понятия «объект» и «значение» взаимозаменяемо.

Последовательность:

Упорядоченная коллекция значений, где каждое значение идентифицируется целочисленным индексом.

Элемент:

Одно из значений в последовательности.

Индекс:

Целочисленное значение, используемое для доступа к элементу в последовательности, например символа в строке. В Python индексы начинаются с 0.

Срез:
: Часть строки, определенная диапазоном индексов.

Пустая строка:
: Строка без символов и длиной, равной 0, представлена двумя кавычками.

Неизменяемый тип:
: Свойство последовательности, элементы которой нельзя изменить.

Обход:
: Проход элементов в последовательности, выполняя аналогичную операцию для каждого.

Поиск:
: Алгоритм обхода, который останавливается, когда находит то, что ищет.

Счетчик:
: Переменная, используемая для подсчета чего-либо; обычно инициализируется нулем, а затем увеличивается.

Вызов метода:
: Процесс вызова метода.

Необязательный аргумент:
: Аргумент функции или метода, который использовать необязательно.

УПРАЖНЕНИЯ

Упражнение 8.1
Прочитайте документацию по строковым методам по адресу **docs.python.org/3/library/stdtypes.html#string-method**. Возможно, вы захотите поэкспериментировать с некоторыми из них, чтобы понять, как они работают. Методы `strip()` и `replace()` особенно полезны.

В документации используется синтаксис, который может сбивать с толку. Например, в методе `find(sub[, start[, end]])` квадратные скобки окружают необязательные аргументы. Так что аргумент `sub` обязателен, а `start` — нет, причем если вы используете аргумент `start`, то можете по желанию добавить и `end`.

Упражнение 8.2
Существует строковый метод `count()`, похожий на функцию из раздела «Циклы и счетчики» в этой главе. Прочитайте документацию по этому методу и напишите его вызов, который подсчитывает число букв *a* в слове «банан».

Упражнение 8.3

Срез строки может принимать третий индекс, который определяет «размер шага», то есть количество символов, которое надо пропустить. Размер шага 2 означает: надо учитывать каждый второй символ; 3 — каждый третий, и так далее.

```
>>> fruit = 'банан'
>>> fruit[0:5:2]
'бнн'
```

При размере шага -1 слово обрабатывается с конца, поэтому срез [::-1] выводит строку в обратном порядке.

Используйте это для написания однострочной версии функции `is_palindrome()` из упражнения 6.3.

Упражнение 8.4

Показанные ниже функции предназначены для проверки наличия в строке строчных букв, но по крайней мере некоторые из них содержат ошибки. Для каждой функции опишите, что на самом деле делает функция (предполагая, что параметр является строкой).

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
            flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag
```

```
def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Упражнение 8.5

Шифр Цезаря — несложный метод шифрования, который включает в себя ротации каждой буквы на фиксированное количество позиций. Ротация буквы означает сдвиг ее по алфавиту, при необходимости переход на начало алфавита. Например, А со сдвигом на 3 — это Г, а Я на 1 — это А.

Чтобы совершить ротацию слова, сдвиньте каждую букву на одинаковую величину. Например, слово «пайтон», сдвинутое на 7, становится «цжрщхф», а «ъшцйч» на –10 — «роман». В фильме 2001 года «Космическая одиссея» корабль называется HAL, то есть IBM со сдвигом на –1.

Напишите функцию `rotate_word()`, которая принимает строку и целое число в качестве параметров и возвращает новую строку, содержащую буквы из исходной строки, смещенные на заданную величину.

Возможно, вы захотите использовать встроенную функцию `ord()`, которая преобразует символ в числовой код, и функцию `chr()`, которая преобразует числовые коды в символы. Буквы алфавита кодируются в алфавитном порядке, например:

```
>>> ord('в') - ord('а')
2
```

так как «*в*» имеет в алфавите индекс 2. Но будьте осторожны: числовые коды для прописных букв отличаются.

Потенциально оскорбительные шутки в интернете иногда закодированы в ROT13, то есть шифром Цезаря с ротацией 13. Если вас нелегко обидеть, найдите и расшифруйте некоторые из них.

Решение: **thinkpython2.com/code/rotate.py**.

ГЛАВА 9
ПРАКТИЧЕСКИЙ ПРИМЕР: ИГРА СЛОВ

В этой главе представлен второй практический пример: решение головоломки с поиском слов с определенными свойствами. Мы найдем самые длинные палиндромы в английском языке и слова, буквы которых располагаются в алфавитном порядке. И я представлю другой подход к разработке программы: сведение к ранее решенной задаче.

ЧТЕНИЕ СПИСКА СЛОВ

Для выполнения упражнений в этой главе нам нужен список английских слов. В интернете доступно множество списков слов, но наиболее подходящим для наших целей будет собранный и переданный в общественное достояние Грейди Уордом в рамках проекта Moby lexicon (см. **https://wikipedia.org/wiki/Moby_Project**). Этот список составлен из 113 809 официальных кроссвордов, то есть слов, которые считаются пригодными для использования в кроссвордах и других играх со словами. В коллекции Moby имя файла, который нам нужен, — *113809of.fic*; вы можете скачать копию в виде файла *words.txt* с сайта **thinkpython2.com/code/words.txt**.

Это текстовый файл, поэтому вы можете открыть его в любом текстовом редакторе, но вы также можете прочитать его непосредственно в среде Python. Встроенная функция `open()` принимает имя файла в качестве параметра и возвращает объект файла, который вы можете использовать для чтения файла*.

```
>>> fin = open('words.txt')
```

* Если будете работать таким образом, не забывайте сохранять файлы. Подробнее об этом можно почитать тут: https://stackoverflow.com/questions/40445910/what-is-the-most-pythonic-way-to-open-a-file. *Прим. науч. ред.*

Здесь fin — это частое название для объекта файла, использующегося для ввода (от file input). Объект файла предоставляет несколько методов для чтения, в том числе `readline()`, который считывает символы из файла до тех пор, пока он не встретит символ перехода на новую строку, и возвращает результат в виде строки:

```
>>> fin.readline()
'aa\r\n'
```

Первое английское слово в этом списке — «aa» — разновидность лавы. Последовательность `\r\n` представляет два непечатаемых символа: возврат каретки и переход на новую строку, которые отделяют это слово от следующего.

Объект файла отслеживает, где он находится в файле, поэтому, если вы снова вызовете метод `readline()`, вы увидите следующее слово:

```
>>> fin.readline()
'aah\r\n'
```

Следующее слово — «aah», тоже вполне словарное. Если вам мешают непечатаемые символы, мы можем избавиться от них с помощью строкового метода `strip()`:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

Вы также можете использовать объект файла в составе цикла `for`. Программа ниже считывает файл *words.txt* и печатает каждое слово, по одному в строке:

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

УПРАЖНЕНИЯ

Решения этих упражнений показаны в следующем разделе. Но стоит, по крайней мере, попытаться выполнить каждое из упражнений, прежде чем читать решения.

Упражнение 9.1
Напишите программу, которая считывает файл *words.txt* и печатает только слова длиной более 20 символов (не считая символы окончания строки).

Упражнение 9.2
В 1939 году Эрнест Винсент Райт опубликовал роман длиной 50 000 слов под названием «Гэдсби», в котором не встречается буква «e». Так как буква «e» в английском языке самая распространенная, написать такую книгу было нелегко.

На самом деле, трудно выразить мысль без наиболее распространенного символа. Поначалу это сложно, но внимательность и часы тренировок постепенно улучшат вашу производительность.

Напишите функцию `has_no_e()`, которая возвращает `True`, если в данном слове нет буквы «e».

Измените функцию из предыдущего упражнения, чтобы печатать только те слова, которые не содержат букву «e», и вычислите количество (в процентах) слов без этой буквы в списке.

Упражнение 9.3
Напишите функцию `avoids()`, которая принимает слово и строку запрещенных букв и которая возвращает `True`, если в слове нет запрещенных букв.

Измените вашу программу, чтобы предложить пользователю ввести строку запрещенных букв, а затем печатает количество слов, которые не содержат ни одного из них. Можете ли вы найти сочетание из пяти запрещенных букв, которое исключает наименьшее количество слов?

Упражнение 9.4
Напишите функцию `uses_only()`, которая принимает слово и последовательность букв и возвращает `True`, если слово содержит только буквы из списка. Можете ли вы составить предложение, используя только буквы `acefhlo`?

Упражнение 9.5
Напишите функцию `uses_all()` которая принимает слово и последовательность обязательных букв и возвращает `True`, если слово содержит все обязательные буквы из списка. Вычислите количество слов, которые используют все гласные — `aeiou`? А как насчет `aeiouy`?

Упражнение 9.6
Напишите функцию `is_abecedarian()`, которая возвращает `True`, если буквы в слове располагаются в алфавитном порядке (двойные буквы допустимы). Сколько таких слов в списке?

ПОИСК

У всех упражнений в предыдущем разделе есть кое-что общее; их можно решить с помощью алгоритма поиска из раздела «Поиск» в предыдущей главе. Простейший пример выглядит так:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

Цикл `for` перебирает символы в слове (`word`). Если буква `e` обнаружена, немедленно возвращается значение `False`; в противном случае выполняется переход к следующей букве. Если выход из цикла выполнен нормально, это означает, что буква `e` не обнаружена, и поэтому возвращается значение `True`.

Вы могли бы написать такую функцию более кратко, используя оператор `in`, но я начал с этой версии, потому что она демонстрирует логику алгоритма поиска.

Функция `avoids()` — это обобщенная версия функции `has_no_e()`, но имеет ту же структуру:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

Значение `False` возвращается, как только обнаружена запрещенная буква; если же достигнут конец цикла, возвращается значение `True`.

Функция `use_only()` похожа, но у нее обратный смысл условия:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

Вместо списка запрещенных букв используется список доступных букв. Если в слове обнаружена буква, которой нет в списке, возвращается значение `False`.

Функция `uses_all()` аналогична, за исключением того, что мы меняем местами роли слова и строки символов:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

Вместо того чтобы перебирать буквы в слове, цикл проходит по обязательным буквам. Если в слове нет нужных букв, возвращается значение `False`.

Если бы вы реально думали как программисты, то поняли, что `uses_all()` — это частный случай ранее решенной задачи, и написали бы:

```
def uses_all(word, required):
    return uses_only(required, word)
```

Это пример плана разработки программы, который называется **сведением к ранее решенной задаче**, то есть вы рассматриваете новую задачу как частный случай уже решенной и применяете существующее решение.

ЦИКЛЫ С ИНДЕКСАМИ

Я писал функции в предыдущем разделе с циклами `for`, потому что мне нужны были только символы в строках; мне не нужно было ничего делать с индексами.

В функции `is_abecedarian()` мы должны сравнить соседние буквы, что немного сложно организовать с помощью цикла `for`:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

В качестве альтернативы можно использовать рекурсию

```
def is_abecedarian(word):
    if len(word) <= 1:
```

```
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

Другой вариант — использовать цикл `while`:

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

Цикл начинается с `i = 0` и заканчивается, когда переменная `i` становится равна `len(word)-1`. Каждый раз в цикле сравнивается *i*-й символ (который можно рассматривать как текущий) с (*i*+1)-м символом (который можно рассматривать как следующий).

Если следующий символ меньше (в алфавите размещен раньше) текущего, то определяется нарушение алфавитного порядка и возвращается значение `False`.

Если достигается конец цикла без обнаружения ошибки, то слово проходит тест. Чтобы убедиться, что цикл заканчивается правильно, рассмотрите пример со словом flossy. Длина слова равна 6, поэтому последний раз цикл запускается, когда значение переменной `i` равно 4 — индексу второго с конца символа. На последней итерации второй с конца символ сравнивается с последним, что нам и нужно.

Ниже представлена версия `is_palindrome()` (см. упражнение 6.3), в которой использованы два индекса: один с начала и идет вперед по слову; другой идет с конца слова к началу.

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1

    return True
```

Или мы можем свести код к ранее решенной задаче и написать:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

Здесь используется функция `is_reverse()`, изображенная на рис. 8.2.

ОТЛАДКА

Тестировать программы сложно. Функции в этой главе относительно легко протестировать, поскольку вы можете проверить результаты самостоятельно. Но составить набор слов, который мог бы выявить все возможные ошибки, крайне тяжело, если не невозможно.

Взяв функцию `has_no_e()` в качестве примера, нужно проверить два очевидных случая: слова, которые содержат латинскую букву «e», должны возвращать `False`, а слова, которые не содержат ее, — `True`. Оба случая довольно просты.

Но в обоих есть менее очевидные ситуации. Среди слов с буквой «e», вы должны проверить слова, где «e» стоит в начале, конце и посередине. Вы должны проверить длинные слова, короткие слова и очень короткие слова, такие как пустая строка. Пустая строка — пример особого неочевидного случая, где часто встречаются ошибки.

В дополнение к созданным вами тестовым примерам вы можете протестировать собственную программу с помощью списка слов, например *words.txt*. Проверяя выходные данные, вы будете находить ошибки, но будьте внимательны: вы можете определить один вид ошибок (слова, которые не должны быть включены, но есть), но не сможете другой (слова, которые должны быть включены, но не включены).

В целом тестирование помогает обнаружить ошибки, но создать хороший набор тестовых данных непросто, и даже если вы это сделаете, нет никакой уверенности, что ваша программа абсолютно корректна. Как сказал один легендарный программист:

> Тестирование программы может выявить ошибку, но не может доказать, что ошибок нет!
>
> Эдсгер В. Дейкстра

СЛОВАРЬ ТЕРМИНОВ

Объект файла:

Значение, которое представляет открытый файл.

Сведение к ранее решенной задаче:

Способ решения задачи, выражая ее как частный случай ранее решенной.

Особый случай:

Нетипичный или неочевидный тестовый случай (который, вероятно, не будет обработан правильно).

УПРАЖНЕНИЯ

Упражнение 9.7

Это задание основано на загадке из радиопередачи Car Talk (**www.cartalk.com/content/puzzlers**).

Найдите слово с тремя последовательными удвоенными буквами. Я подскажу пару слов, которые почти удовлетворяют условию, но не совсем. Например, слово *c-o-m-m-i-t-t-e-e*. Было бы замечательно, если бы не коварно закравшаяся буква "i". *Или M-i-s-s-i-s-s-i-p-p-i*. Если бы вы могли убрать буквы "i", оно подошло бы идеально. Но есть слово, которое состоит из трех последовательных пар букв, и, насколько мне известно, это единственное такое слово. Может быть, есть еще пять сотен таких, но я могу вспомнить только одно. Что это за слово?

Напишите программу, чтобы найти его.

Решение: **thinkpython2.com/code/cartalk1.py**.

Упражнение 9.8

Вот еще одно задание из радиопередачи Car Talk (**www.cartalk.com/content/puzzlers**).

«На днях я ехал по шоссе и случайно бросил взгляд на одометр. Как и большинство одометров, он показывает шесть цифр, только целые километры. Так, например, если бы у моей машины был пробег 300 000 километров, одометр показал бы 3-0-0-0-0-0.

То, что я увидел в тот день, было очень интересно. Я заметил, что последние четыре цифры были палиндромом; то есть они читаются одинаково справа налево и слева направо. Например, 5-4-4-5 — палиндром, поэтому мой одометр мог бы показывать 3-1-5-4-4-5.

Через километр последние пять чисел стали палиндромом. Например, могло быть значение 3-6-5-4-5-6. Еще через один километр средние четыре числа из шести были палиндромом. Готовы? Еще спустя километр все шесть были палиндромом!

Вопрос: какой пробег был на одометре, когда я впервые посмотрел?»

Напишите программу на языке Python, которая проверяет все шестизначные числа и печатает любые числа, которые удовлетворяют этим требованиям.

Решение: **thinkpython2.com/code/cartalk2.py**.

Упражнение 9.9

Вот еще одно задание из радиопередачи Car Talk, которое вы можете решить с помощью поиска (**www.cartalk.com/content/puzzlers**).

«Недавно я навещал маму, и мы поняли, что две цифры моего возраста в обратном порядке равны ее возрасту. Например, если ей 73 года, мне 37. Мы задались вопросом, как часто это происходило на протяжении жизни, но отвлеклись на другие темы и не нашли ответа.

Когда я вернулся домой, то понял, что такое уже случалось шесть раз. Еще я понял, что, если нам повезет, совпадение случится через несколько лет, а если нам очень повезет, то и еще раз после этого. Другими словами, такие совпадения произошли бы восемь раз за все время. Итак, вопрос в том, сколько мне сейчас лет?»

Напишите программу на Python, способную решить эту задачу. Подсказка: вам может пригодиться строковый метод `zfill()`.

Решение: **thinkpython2.com/code/cartalk3.py**.

ГЛАВА 10
СПИСКИ

В этой главе представлен один из самых полезных встроенных типов данных языка Python — список (list). Вы также узнаете больше об объектах и о том, что может произойти, если в коде программы используется несколько имен для одного и того же объекта.

СПИСОК — ЭТО ПОСЛЕДОВАТЕЛЬНОСТЬ

Как и строка, **список (list)** представляет собой последовательность значений. Значения в строке — это символы, а в списке они могут быть любого типа. Значения в списке называются **элементами (elements, items)**.

Есть несколько способов создать новый список; самый простой — заключить элементы в квадратные скобки [и]:

```
[10, 20, 30, 40]
['квакающая лягушка', 'блеющий барашек', 'поющий жаворонок']
```

Первый пример — список из четырех целых чисел. Второй — список трех строк. Элементы списка не обязательно должны быть одного типа. Список ниже содержит строку, число с плавающей точкой, целое число и (внимание!) другой список:

```
['спам', 2.0, 5, [10, 20]]
```

Список внутри другого списка называется **вложенным (nested)**.

Список, который не содержит элементов, называется **пустым**. Вы можете создать такой с помощью пустых скобок [].

Как и следовало ожидать, вы можете присваивать переменным значения типа список:

```
>>> cheeses = ['Чеддер', 'Эдам', 'Гауда']
>>> numbers = [42, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Чеддер', 'Эдам', 'Гауда'] [42, 123] []
```

СПИСКИ — ИЗМЕНЯЕМЫЙ ТИП ДАННЫХ

Синтаксис для доступа к элементам списка такой же, что и для доступа к символам строки — инструкция []. В скобках указывается индекс. Не забудьте, что индексы начинаются с 0:

```
>>> cheeses[0]
'Чеддер'
```

В отличие от строк, списки можно изменять. Когда инструкция [] находится в левой части инструкции присваивания, она идентифицирует элемент списка, который будет назначен:

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

Первый элемент списка `numbers`, раньше был `123`, теперь равен `5`.

На рис. 10.1 показана диаграмма состояния списков `cheeses`, `numbers` и `empty`.

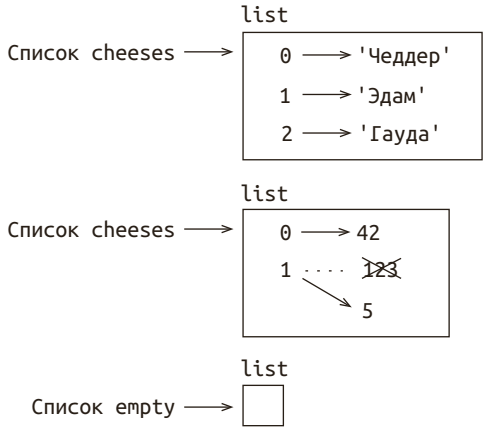


Рис. 10.1. Диаграмма состояния

Списки можно представить в виде слова «список» снаружи и элементами списка внутри. Список `cheeses` содержит три элемента с индексами `0`, `1` и `2`. Список `numbers` содержит два элемента. На диаграмме показано, что значение второго элемента было изменено со `123` на `5`. Список `empty` не содержит элементов.

Индексы списка работают так же, как индексы строк:

— любое целочисленное выражение может быть использовано в качестве индекса;
— если вы попытаетесь прочитать или записать элемент, который не существует, вы увидите ошибку `IndexError`;
— если индекс имеет отрицательное значение, он считывается в обратном направлении, начиная с конца списка*.

Оператор `in` также применим и к спискам:

```
>>> cheeses = ['Чеддер', 'Эдам', 'Гауда']
>>> 'Эдам' in cheeses
True
>>> 'Бри' in cheeses
False
```

ОБХОД СПИСКА

Самый распространенный способ обхода элементов списка — цикл `for`. Синтаксис такой же, как и для строк:

```
for cheese in cheeses:
    print(cheese)
```

Этот способ подойдет, если вам нужно только прочитать элементы списка. Но если вы хотите добавить или обновить элементы, индексы необходимы. Распространенный способ сделать это — объединить встроенные функции `range()` и `len()`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

Этот цикл обходит список и обновляет каждый элемент. Функция `len()` возвращает количество элементов в списке. А функция `range()` возвращает список индексов от 0 до n–1, где n — длина списка. При каждой итерации цикла переменной `i` присваивается индекс следующего элемента. Инструкция присваивания в теле использует переменную `i` для чтения старого значения элемента и присвоения нового.

---

* Индекс –1 будет означать последний элемент, –2 — предпоследний, –3 — предпредпоследний и так далее. *Прим. науч. ред.*

Тело цикла for для пустого списка не выполняется:

```
for x in []:
    print('Так не бывает.')
```

Хотя в списке может быть другой список, вложенный все равно считается одним элементом.

Длина этого списка равна четырем:

```
['спам', 1, ['Бри', 'Рокфор', 'Пармезан'], [1, 2, 3]]
```

ОПЕРАЦИИ СО СПИСКАМИ

Оператор + объединяет списки:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b >>> c
[1, 2, 3, 4, 5, 6]
```

Оператор * повторяет список заданное количество раз:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Первый пример повторяет [0] четыре раза. Второй пример повторяет список [1, 2, 3] три раза.

СРЕЗЫ СПИСКОВ

Оператор среза применим и к спискам:

```
>>> t = ['а', 'б', 'в', 'г', 'д', 'е']
>>> t[1:3]
['б', 'в']
>>> t[:4]
['а', 'б', 'в', 'г']
>>> t[3:]
['г', 'д', 'е']
```

Если вы опустите первый индекс, срез начнется с нулевого элемента. Если вы опустите второй индекс, срез продолжится до конца списка. Поэтому, если вы опустите оба индекса, срез будет копией всего списка:

```
>>> t[:]
['а', 'б', 'в', 'г', 'д', 'е']
```

Поскольку списки можно изменять, рекомендуется создавать копии списков перед изменением.

Оператор среза в левой части операции присваивания позволяет обновлять сразу несколько элементов:

```
>>> t = ['а', 'б', 'в', 'г', 'д', 'е']
>>> t[1:3] = ['й', 'ч']
>>> t
['а', 'й', 'ч', 'г', 'д', 'е']
```

МЕТОДЫ СПИСКОВ

В Python для работы со списками есть встроенные методы. Метод `append()` добавляет новый элемент в конец списка:

```
>>> t = ['а', 'б', 'в']
>>> t.append('г')
>>> t
['а', 'б', 'в', 'г']
```

Метод `extend()` принимает список в качестве аргумента и добавляет все элементы нового списка к старому:

```
>>> t1 = ['а', 'б', 'в']
>>> t2 = ['г', 'д']
>>> t1.extend(t2)
>>> t1
['а', 'б', 'в', 'г', 'д']
```

В этом примере значение переменной `t2` остается неизменным.

Метод `sort()` упорядочивает элементы списка по возрастанию:

```
>>> t = ['г', 'в', 'д', 'б', 'а']
>>> t.sort()
>>> t
['а', 'б', 'в', 'г', 'д']
```

Большинство методов списка не возвращают результат, то есть они изменяют список и возвращают `None`. Если вы случайно напишете `t = t.sort()`, вы будете разочарованы.

СОПОСТАВЛЕНИЕ, ФИЛЬТРАЦИЯ И СОКРАЩЕНИЕ

Чтобы сложить все числа в списке, вы можете использовать такой цикл:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

Переменная `total` инициализируется со значением `0`. Каждый раз в цикле переменная `x` получает один элемент из списка. Оператор `+=` предоставляет краткий способ обновления переменной. Показанный ниже оператор **комбинированного присваивания (augmented assignment statement)**:

```
total += x
```

эквивалентен записи

```
total = total + x
```

По мере выполнения цикла в переменной `sum` накапливается сумма элементов; переменная, используемая таким образом, иногда называется **счетчиком (accumulator)**.

Складывать элементы списка требуется так часто, что Python предоставляет для этого встроенную функцию `sum()`:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

Операцию, которая объединяет последовательность элементов в одно значение, называют **сокращением (reduce)**.

Иногда нужно обойти один список для создания другого. Например, функция ниже берет список строк и возвращает новый список, содержащий прописные буквы:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

Переменная `res` инициализируется как пустой список; каждый раз в цикле мы добавляем следующий элемент. Так что переменная `res` — это еще один вид счетчика.

Операция, подобная функции `capitalize_all()`, называется **сопоставлением** или **маппингом (map)**, потому что она «сопоставляет» функцию (в данном случае метод `capitalize()`) с каждым из элементов последовательности.

Другая распространенная операция — выбор некоторых элементов из множества элементов списка и возврат подмножества. В качестве примера показанная ниже функция берет список строк и возвращает новый список, содержащий только прописные буквы:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

Строковый метод `isupper` возвращает `True`, если строка содержит только прописные буквы.

Операция, подобная функции `only_upper()`, называется **фильтром (filter)**, потому что она выбирает некоторые элементы и отфильтровывает другие.

Большинство частых операций со списком можно выразить как комбинацию сопоставления, фильтрации и сокращения.

УДАЛЕНИЕ ЭЛЕМЕНТОВ

Существует несколько способов удалить элементы из списка. Если вы знаете индекс нужного элемента, подойдет метод `pop()`:

```
>>> t = ['а', 'б', 'в']
>>> x = t.pop(1)
>>> t
['а', 'в']
>>> x
'б'
```

Метод `pop()` изменяет список и возвращает удаленный элемент. Если вы не предоставите индекс, он удалит и вернет последний элемент.

Если вам не нужно удаленное значение, вы можете использовать инструкцию `del`:

```
>>> t = ['а', 'б', 'в']
>>> del t[1]
>>> t
['а', 'в']
```

Если известен элемент, который вы хотите удалить (но не индекс), можно использовать команду `remove`:

```
>>> t = ['а', 'б', 'в']
>>> t.remove('б')
>>> t
['а', 'в']
```

Возвращаемое значение `remove` — `None`.

Чтобы удалить более одного элемента, вы можете использовать команду `del` и индекс среза:

```
>>> t = ['а', 'б', 'в', 'г', 'д', 'е']
>>> del t[1:5]
>>> t
['а', 'е']
```

Как обычно, срез выделяет все элементы вплоть до второго индекса, не включая его.

СПИСКИ И СТРОКИ

Строка — это последовательность символов, а список — это последовательность значений, но список символов не то же самое, что и строка. Чтобы преобразовать строку в список символов, вы можете использовать функцию `list()`:

```
>>> s = 'спам'
>>> t = list(s)
>>> t
['с', 'п', 'а', 'м']
```

Поскольку `list()` — это имя встроенной функции, не следует использовать его в качестве имени переменной. Я также избегаю имени `l` (латинская буква L), потому что оно очень похоже на `1` (один). Вот почему я использую имя `t`, а не `l`.

Функция `list()` разбивает строку на отдельные буквы. Если вы хотите разбить строку на слова, вы можете использовать метод `split()`:

```
>>> s = 'тоска по фьордам'
>>> t = s.split()
>>> t
['тоска', 'по', 'фьордам']
```

Необязательный аргумент, называемый **разделителем (delimiter)**, указывает, какие символы использовать в качестве границ слов. В следующем примере в качестве разделителя используется дефис:

```
>>> s = 'спам-спам-спам'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['спам', 'спам', 'спам']
```

Метод `join()` — это противоположность методу `split()`. Он принимает список строк и конкатенирует, то есть объединяет, элементы. Метод `join()` — строковый, поэтому вы должны вызвать его для разделителя и передать список в качестве параметра:

```
>>> t = ['тоска', 'по', 'фьордам']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'тоска по фьордам'
```

В этом случае в роли разделителя — пробел, поэтому метод `join()` ставит пробел между словами. Чтобы объединить строки без пробелов, вы можете использовать пустую строку, указав символы `''` в качестве разделителя.

ОБЪЕКТЫ И ЗНАЧЕНИЯ

Допустим, мы выполним следующие операции присваивания:

```
a = 'банан'
b = 'банан'
```

Мы знаем, что обе переменные, `a` и `b`, ссылаются на строку, но мы не знаем, ссылаются ли они на одну и ту же строку. Существует два возможных состояния, показанных на рис. 10.2.

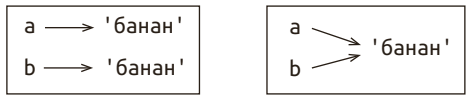


Рис. 10.2. Диаграмма состояний

В одном случае переменные `a` и `b` относятся к двум разным объектам, имеющим одинаковое значение. Во втором случае они ссылаются на один и тот же объект.

Чтобы проверить, ссылаются ли две переменные на один и тот же объект, вы можете использовать оператор `is`:

```
>>> a = 'банан'
>>> b = 'банан'
>>> a is b
True
```

В этом примере Python создал только один строковый объект, и переменные `a` и `b` ссылаются на него.

Но когда вы создаете два списка, вы получаете два объекта:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

Таким образом, диаграмма состояния показана на рис. 10.3.

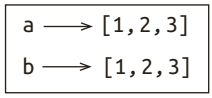


Рис. 10.3. Диаграмма состояния

В этом случае мы говорим, что два списка **эквивалентны (equivalent)**, потому что они имеют одинаковые элементы, но не **идентичны (identical)**, потому что они не являются одним и тем же объектом. Если два объекта идентичны, они также эквивалентны, но, если они эквивалентны, они не обязательно идентичны.

До сих пор мы использовали понятия «объект» и «значение» взаимозаменяемо, но правильнее сказать, что объект имеет значение. Если вы выполните `[1, 2, 3]`, вы получите объект списка, значение которого представляет собой последовательность целых чисел. Если другой список имеет те же элементы, мы говорим, что он имеет то же значение, но не является тем же объектом.

ПСЕВДОНИМЫ

Если переменная `a` ссылается на объект и вы назначаете `b = a`, тогда обе переменные ссылаются на один и тот же объект:

```
>>> a = [1, 2, 3]
>>> b = a
```

```
>>> b is a
True
```

Таким образом, диаграмма состояния показана на рис. 10.4.

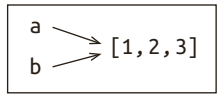


Рис. 10.4. Диаграмма состояния

Связь переменной с объектом называется **ссылкой (reference)**. В этом примере есть две ссылки на один и тот же объект.

Объект с более чем одной ссылкой имеет более одного имени, поэтому мы говорим, что у объекта есть **псевдонимы** или **синонимы (aliases)**.

Если объект с псевдонимом изменяемый, изменения одного псевдонима (синонима) влияют на другой:

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

И хотя это свойство бывает полезным, из-за него возникает много ошибок. Как правило, безопаснее не использовать псевдонимы при работе с изменяемыми объектами.

Для неизменяемых объектов, таких как строки, создание псевдонимов не представляет большой проблемы. Как в этом примере:

```
a = 'банан'
b = 'банан'
```

Почти никогда не имеет значения, относятся переменные `a` и `b` к одной и той же строке или нет.

АРГУМЕНТЫ СПИСКА

Когда вы передаете список в функцию, функция получает ссылку на список. Если функция изменяет список, вызывающая сторона видит изменение. Например, функция `delete_head()` удаляет первый элемент из списка:

```
def delete_head(t):
    del t[0]
```

Ниже продемонстрирован пример:

```
>>> letters = ['а', 'б', 'в']
>>> delete_head(letters)
>>> letters
['б', 'в']
```

Параметр `t` и переменная `letters` — псевдонимы для одного и того же объекта. Диаграмма стека показана на рис. 10.5.

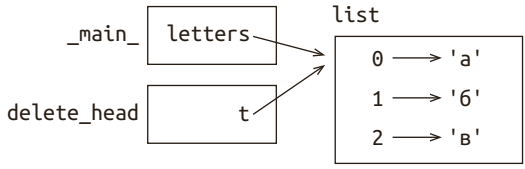


Рис. 10.5. Стековая диаграмма

Поскольку список используется в двух фреймах, я нарисовал его между ними.

Важно различать операции, которые изменяют списки, и операции, которые создают новые списки. Например, метод `append()` изменяет список, а оператор + создает новый список:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

Метод `append()` изменяет список и возвращает `None`:

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
>>> t1
```

Оператор + создает новый список и оставляет исходный список без изменений.

Это различие важно, когда вы пишете функции, которые должны изменять списки. Например, показанная ниже функция не удаляет первый символ списка:

```
def bad_delete_head(t):
    t = t[1:]                # НЕПРАВИЛЬНО!
```

Оператор среза создает новый список, и присваивание заставляет переменную `t` ссылаться на него, но это не влияет на список, переданный в функцию.

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

Перед выполнением функции `bad_delete_head()`, переменные `t` и `t4` ссылаются на один и тот же список. В конце переменная `t` ссылается на новый список, а переменная `t4` по-прежнему ссылается на оригинальный, неизменный список.

Альтернатива — написание функции, которая создает и возвращает новый список. Например, функция `tail()` возвращает все, кроме первого элемента списка:

```
def tail(t):
    return t[1:]
```

Эта функция оставляет исходный список без изменений. Вот как это работает:

```
>>> letters = ['а', 'б', 'в']
>>> rest = tail(letters)
>>> rest
['б', 'в']
```

ОТЛАДКА

Если использовать списки и другие изменяемые объекты неосторожно, за отладкой, возможно, придется провести много часов. Ниже перечислены некоторые распространенные ошибки и способы их избежать.

1. Большинство методов списка изменяют аргумент и возвращают None. Это противоположно строковым методам, которые возвращают новую строку и оставляют оригинал в покое.

Если вы привыкли писать код для строк следующим образом:

```
word = word.strip()
```

Соблазнительно написать код списка так:

```
t = t.sort()      # НЕПРАВИЛЬНО!
```

Поскольку функция `sort()` возвращает `None`, следующая операция, которую вы попытаетесь выполнить с переменной t, скорее всего, закончится ошибкой.

Прежде чем использовать методы и операторы списка, вы должны внимательно прочитать документацию, а затем протестировать их в интерактивном режиме.

2. Выберите что-то одно и придерживайтесь этого.

Часть проблемы со списками в том, что способов решить задачу слишком много. Например, чтобы удалить элемент из списка, можно использовать `pop`, `remove`, `del` или даже присваивание среза.

Чтобы добавить элемент, можно использовать метод `append()` или оператор +. Если переменная `t` является списком, а `x` — элементом списка, верно следующее:

```
t.append(x)
t = t + [x]
t += [x]
```

а это — неверно:

```
t.append([x])       # НЕПРАВИЛЬНО!
t = t.append(x)     # НЕПРАВИЛЬНО!
t + [x]             # НЕПРАВИЛЬНО!
t = t + x           # НЕПРАВИЛЬНО!
```

Выполните каждый из этих примеров в интерактивном режиме, чтобы как следует разобраться, что они делают. Обратите внимание, что только последний вызывает ошибку во время выполнения; остальные три законны, но они работают не так.

3. Делайте копии, чтобы избежать псевдонимов (синонимов).

Если вы хотите использовать метод вроде `sort()`, который изменяет аргумент, но вам необходимо сохранить исходный список, можно сделать копию:

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

В этом примере можно использовать встроенную функцию `sorted()`, которая возвращает новый отсортированный список и оставляет оригинал в покое:

```
>>> t2 = sorted(t)
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

СЛОВАРЬ ТЕРМИНОВ

Список:
Последовательность значений.

Элемент:
Одно из значений в списке (или другой последовательности).

Вложенный список:
Список, который является элементом другого списка.

Счетчик:
Переменная, используемая в цикле для суммирования или накопления результата.

Комбинированное присваивание:
Код, который обновляет значение переменной, используя оператор, такой как +=.

Сокращение:
Алгоритм обработки, который проходит список и накапливает примененные к элементам вычисления (операции) в одном результате.

Сопоставление (маппирование):
Алгоритм обработки, который проходит список и выполняет операцию над каждым элементом.

Фильтр:
Алгоритм обработки, который проходит список и выбирает элементы, которые удовлетворяют некоторому критерию.

Объект:
Что-то, на что может ссылаться переменная. Объект имеет тип и значение.

Эквивалентный:
Имеющий то же значение.

Идентичный:
Являющийся тем же объектом (подразумевает эквивалентность).

Ссылка:
Ассоциация между переменной и ее значением.

Псевдонимы:
Обстоятельство, при котором две или более переменных ссылаются на один и тот же объект.

Разделитель:
Символ или строка, используемые, чтобы указать, где строка должна быть разделена.

УПРАЖНЕНИЯ

Вы можете скачать решения этих упражнений по адресу **thinkpython2.com/code/list_exercises.py**.

Упражнение 10.1
Напишите функцию `nested_sum()`, которая берет список списков целых чисел и складывает элементы из всех вложенных списков. Например:

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

Упражнение 10.2
Напишите функцию `cumsum()`, которая берет список чисел и возвращает кумулятивную сумму; то есть новый список, где *i*-й элемент — это сумма первых элементов *i* +1 из исходного списка. Например:

```
>>> t = [1, 2, 3]
>>> cumsum(t)
[1, 3, 6]
```

Упражнение 10.3
Напишите функцию `middle()`, которая принимает список и возвращает новый список — со всеми элементами, кроме первого и последнего. Например:

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

Упражнение 10.4

Напишите функцию `chop()`, которая принимает список, модифицирует его, удаляя первый и последний элементы, и возвращает `None`. Например:

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

Упражнение 10.5

Напишите функцию `is_sorted()`, которая принимает список в качестве параметра и возвращает `True`, если список отсортирован в порядке возрастания, и `False` в противном случае. Например:

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['б', 'а'])
False
```

Упражнение 10.6

Два слова называются анаграммами, если одно слово можно получить, переставив местами буквы другого. Напишите функцию `is_anagram()`, которая принимает две строки и возвращает `True`, если они являются анаграммами.

Упражнение 10.7

Напишите функцию `has_duplicates()`, которая принимает список и возвращает `True`, если есть какой-либо элемент, который появляется более одного раза. Она не должна изменять исходный список.

Упражнение 10.8

Это упражнение относится к так называемому парадоксу дней рождения, о котором вы можете прочитать по адресу **ru.wikipedia.org/wiki/Парадокс_дней_рождения**.

Если в вашем классе 23 ученика, каковы шансы, что у вас двоих день рождения в один день? Вы можете оценить эту вероятность, генерируя случайные выборки из 23 дней рождения и проверив их на совпадение.

Подсказка: вы можете генерировать случайные дни рождения с помощью функции `randint()` из модуля `random`.

Вы можете скачать мое решение по адресу **thinkpython2.com/code/birthday.py**.

Упражнение 10.9

Напишите функцию, которая читает файл *words.txt* и создает список с одним элементом на слово. Напишите две версии этой функции, одну с помощью метода `append()`, а другую — с помощью кода `t = t + [x]`. Какой из них выполняется дольше? Почему?

Решение: **thinkpython2.com/code/wordlist.py**.

Упражнение 10.10

Проверить, есть ли слово в списке слов, можно оператором `in`. Но он ищет слова по порядку, так что это медленный способ.

Поскольку слова расположены в алфавитном порядке, мы можем ускорить процесс с помощью дихотомического поиска (также известного как двоичный, или бинарный, поиск), который делает примерно то же, что и вы, когда ищете слово в словаре. Вы начинаете с середины и проверяете, стоит ли слово, которое вы ищете, перед словом в середине списка. Если это так, вы ищете в первой половине списка таким же образом. В противном случае переходите ко второй половине.

В любом случае вы сокращаете оставшееся пространство поиска в два раза. Если список слов содержит 113 809 слов, потребуется около 17 шагов, чтобы найти слово или сделать вывод, что его там нет.

Напишите функцию `in_bisect()`, которая принимает отсортированный список и целевое значение и возвращает индекс значения в списке, если оно там встречается, или `None`, если это не так.

Или вы можете прочитать документацию по применению модуля `bisect` и использовать его!

Решение: **thinkpython2.com/code/inlist.py**.

Упражнение 10.11

Два слова являются «обратной парой», если порядок букв каждого из них обратен порядку в другом. Напишите программу, которая находит все обратные пары в списке слов.

Решение: **thinkpython2.com/code/reverse_pair.py**.

Упражнение 10.12

Два слова образуют «взаимозамкнутость», если брать поочередно буквы от каждого, и они образуют новое слово. Например, слова shoe и cold таким образом образуют schooled.

Решение: **thinkpython2.com/code/interlock.py**. Примечание: на это упражнение меня вдохновил пример на сайте **puzzlers.org**.

1. Напишите программу, которая находит все «взаимнозамкнутые» пары слов. Подсказка: не перебирайте все пары!
2. Можете ли вы найти три слова, образующих взаимозамкнутость; то есть каждая третья буква первого, второго или третьего слова образует новое слово.

ГЛАВА 11
СЛОВАРИ

В этой главе представлен еще один встроенный тип данных, который называется **словарем**. Словари — одна из лучших функций Python; это строительные блоки многих эффективных и элегантных алгоритмов.

СЛОВАРЬ — ЭТО ПОСЛЕДОВАТЕЛЬНОСТЬ СОПОСТАВЛЕНИЙ

Словарь похож на список, но более универсальный. В списке индексы должны быть целыми числами; в словаре они могут быть (почти) любого типа.

Словарь содержит набор индексов, которые называются **ключами (keys)**, и набор значений. Каждый ключ связан с одним значением. Связь ключа и значения называется **парой «ключ — значение» (key-value pair)** или иногда **элементом (item)**.

На математическом языке словарь — это коллекция сопоставлений ключей со значениями, поэтому вы также можете сказать, что каждый ключ «соответствует» значению. В качестве примера мы создадим словарь, который сопоставляет русские и испанские слова, поэтому все ключи и значения представлены строками.

Функция `dict()` создает новый словарь без элементов. Поскольку `dict()` — имя встроенной функции, не следует использовать его в качестве имени переменной.

```
>>> rus2sp = dict()
>>> rus2sp
{}
```

Фигурные скобки `{}` представляют пустой словарь. Чтобы добавить элемент в словарь, вы можете использовать квадратные скобки:

```
>>> rus2sp['один'] = 'uno'
```

Эта строка создает элемент, который сопоставляет ключ 'один' со значением 'uno'. Если мы снова выведем словарь, мы увидим пару «ключ — значение» с двоеточием между ключом и значением:

```
>>> rus2sp
{'один': 'uno'}
```

Такой формат вывода также используется и при вводе. Например, вы можете создать новый словарь из трех элементов:

```
>>> rus2sp = {'один': 'uno', 'два': 'dos', 'три': 'tres'}
```

Но если вы выведете словарь `rus2sp`, то результат вас может удивить:

```
>>> rus2sp
{'один': 'uno', 'три': 'tres', 'два': 'dos'}
```

Порядок пар «ключ — значение» может не совпадать. Если вы выполните этот же пример на своем компьютере, вы можете получить другой результат. В общем, порядок элементов в словаре непредсказуем.

Но это не проблема, потому что элементы словаря никогда не индексируются с помощью целочисленных индексов. Вместо этого применяются ключи для поиска соответствующих значений:

```
>>> rus2sp['два']
'dos'
```

Ключ 'два' всегда соответствует значению 'dos', поэтому порядок элементов не имеет значения.

Если ключ отсутствует в словаре, вы получите исключение:

```
>>> rus2sp['четыре']
KeyError: 'четыре'
```

К словарям применима функция `len()`; она возвращает количество пар «ключ — значение»:

```
>>> len(rus2sp)
3
```

Оператор `in` также работает со словарями; он сообщает, существует ли такой *ключ* в словаре (но не значение).

```
>>> 'один' in rus2sp
True
>>> 'uno' in rus2sp
False
```

Чтобы проверить, существует ли какое-либо значение в словаре, можно использовать метод `values()`, который возвращает коллекцию значений, а затем применить оператор `in`:

```
>>> vals = rus2sp.values()
>>> 'uno' in vals
True
```

Оператор `in` задействует разные алгоритмы для списков и словарей. Для списков выполняется поиск элементов списка по порядку, как в разделе «Поиск» главы 8. По мере того как список становится длиннее, время поиска увеличивается прямо пропорционально.

Для словарей Python использует алгоритм, называемый хеш-таблицей, с одним замечательным свойством: выполнение оператора `in` занимает примерно одинаковое количество времени, независимо от того, сколько элементов в словаре. Я объясняю, как это возможно, в разделе «Хеш-таблицы» главы 21, но пока вы не прочитаете еще несколько глав, объяснение может не иметь смысла.

СЛОВАРЬ КАК НАБОР СЧЕТЧИКОВ

Предположим, вам дана строка и вы хотите посчитать, сколько раз встречается каждая буква. Есть несколько способов сделать это.

1. Вы можете создать 33 переменные, по одной для каждой буквы алфавита. Затем вы можете пройти строку и для каждого символа увеличить значение соответствующего счетчика, возможно, используя цепочку условных выражений.
2. Вы можете создать список из 33 элементов. Затем вы можете преобразовать каждый символ в число (используя встроенную функцию `ord()`), использовать число в качестве индекса в списке и увеличивать соответствующий счетчик.
3. Вы можете создать словарь с символами в качестве ключей и счетчиками в качестве соответствующих значений. Когда вы видите символ в первый раз, вы добавляете элемент в словарь. После этого вы увеличиваете значение уже существующего элемента.

Каждый из этих способов выполняет одно и то же вычисление, но по-своему.

Реализация (implementation) — это способ выполнения вычислений; некоторые реализации лучше, чем другие. Например, преимущество

реализации с использованием словаря состоит в том, что нам не нужно заранее знать, какие буквы встречаются в строке, и нам нужно выделить место только лишь для букв, которые есть в строке.

Вот как будет выглядеть код:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

Имя функции — `histogram()`, в переводе с английского «гистограмма» — статистический термин для обозначения набора счетчиков (или частотностей).

Первая строка функции создает пустой словарь. Цикл `for` обходит строку посимвольно. На каждой итерации цикла, если символа `c` нет в словаре, создается новый элемент с ключом `c` и начальным значением 1 (поскольку мы видели эту букву первый раз). Если `c` уже есть в словаре, мы увеличиваем `d[c]`.

Вот как это работает:

```
>>> h = histogram('бронтозавр')
>>> h
{'б': 1, 'р': 2, 'о': 2, 'н': 1, 'т': 1, 'з': 1, 'а': 1, 'в': 1}
```

Гистограмма показывает, что буква «б» встречается один раз, «р» и «о» встречаются дважды, и так далее.

В словарях реализован метод `get()`, который принимает ключ и значение по умолчанию. Если ключ есть в словаре, метод `get()` возвращает соответствующее значение; в противном случае он возвращает значение по умолчанию. Например:

```
>>> h = histogram('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('б', 0)
0
```

В качестве упражнения используйте метод `get()`, чтобы сократить код функции `histogram()`. В результате должен быть удалена инструкция `if`.

ЦИКЛЫ И СЛОВАРИ

Если вы используете словарь с инструкцией `for`, он обходит ключи словаря. Например, функция `print_hist()` печатает каждый ключ и соответствующее значение:

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

Вот так выглядит результат:

```
>>> h = histogram('попугай')
>>> print_hist(h)
п 2
о 1
у 1
г 1
а 1
й 1
```

Опять же, ключи выводятся в произвольном порядке. Чтобы обойти ключи в определенном порядке, вы можете использовать встроенную функцию `sorted()`:

```
>>> for key in sorted(h):
...     print(key, h[key])
а 1
г 1
й 1
о 1
п 2
у 1
```

ОБРАТНЫЙ ПОИСК

Если у вас есть словарь `d` и ключ `k`, легко найти соответствующее значение `v = d[k]`.

Эта операция называется **поиском (lookup)**.

А как быть, если у вас есть `v` и вы хотите найти `k`? Тут две проблемы: во-первых, может быть несколько ключей, сопоставляемых со значением `v`. В зависимости от задачи вы можете выбрать один или вам потребуется составить список, содержащий все ключи. Во-вторых, нет простого синтаксиса

для **обратного поиска (reverse lookup)** — придется искать решение нетривиальной задачи.

Ниже показана функция, которая принимает значение и возвращает первый ключ, сопоставляемый с этим значением:

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

Эта функция — еще один пример алгоритма поиска, но тут используется нечто новое: инструкция `raise`. **Инструкция raise** вызывает исключение, в данном случае — `LookupError` — встроенное исключение, которое указывает на неудачное завершение операции поиска.

Если достигнут конец цикла, то значения `v` нет в словаре, поэтому вызывается исключение.

Ниже показан пример успешного обратного поиска:

```
>>> h = histogram('попугай')
>>> k = reverse_lookup(h, 2)
>>> k
'п'
```

И неудачного:

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 5, in reverse_lookup
LookupError
```

Результат, когда вы вызываете исключение, такой же, как и когда Python вызывает его: он печатает трейсбэк (трассировку) и сообщение об ошибке.

Инструкция `raise` может принять подробное сообщение об ошибке в качестве необязательного аргумента. Например:

```
>>> raise LookupError('value does not appear in the dictionary')
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

Обратный поиск намного медленнее, чем прямой; если вам придется выполнять его часто, или если словарь станет достаточно большим, производительность вашей программы снизится.

СЛОВАРИ И СПИСКИ

Списки могут быть значениями в словаре. Например, если есть словарь, который сопоставляет буквы с их частотностью, попробуйте инвертировать его, то есть создать словарь, который сопоставляет частотность с буквами. Поскольку может быть несколько букв с одной и той же частотностью, каждое значение в инвертированном словаре должно быть списком букв.

Ниже показана функция, которая инвертирует словарь:

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

На каждой итерации цикла переменная `key` получает ключ из словаря `d`, а `val` получает соответствующее значение. Если значения `val` нет в `inverse`, это означает, что мы не встречали его раньше, поэтому мы создаем новый элемент и инициализируем его с помощью **синглтона (singletone)** — списка, который содержит один элемент. В противном случае мы уже встречали это значение раньше, поэтому добавляем соответствующий ключ в список.

Ниже представлен пример:

```
>>> hist = histogram('попугай')
>>> hist
{'п': 2, 'о': 1, 'у': 1, 'г': 1, 'а': 1, 'й': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{2: ['п'], 1: ['о', 'у', 'г', 'а', 'й']}
```

На рис. 11.1 показана диаграмма состояния словарей `hist` и `inverse`. Словарь представлен в виде прямоугольника с надписью `dict` над ним и парами «ключ — значение» внутри. Если значения целочисленные, числа с плавающей точкой или строки, я рисую их внутри прямоугольника, а списки я обычно рисую снаружи, просто чтобы не усложнять диаграмму.

Списки могут быть значениями в словаре, как показано в этом примере, но они не могут быть ключами.

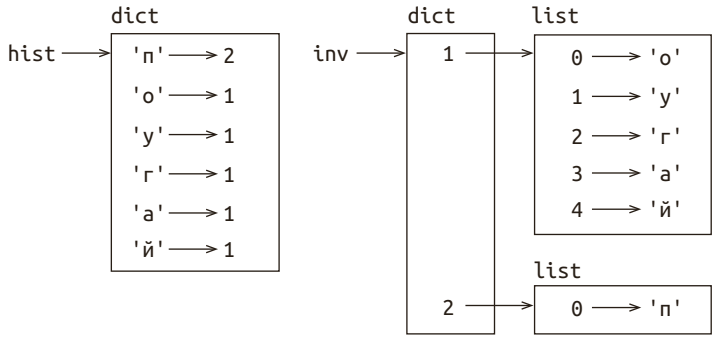


Рис. 11.1. Диаграмма состояния

Вот что произойдет, если вы попытаетесь сделать ключом список:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'ой-ёй'
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

Ранее я упоминал, что словарь реализован с использованием хеш-таблицы, а это означает, что ключи **хешируемые**.

Хеш (hash) — это функция, которая принимает значение любого типа и возвращает целое число. Словари используют эти целые числа, называемые хеш-значениями, для хранения и поиска пар «ключ — значение».

Эта система работает нормально, когда ключи неизменны. Но если ключи будут относиться к изменяемому типу данных, такому как списки, может произойти что-то плохое. Например, когда вы создаете пару «ключ — значение», Python хеширует ключ и сохраняет его в соответствующем месте. Если вы измените ключ, а затем снова его хешируете, он помещается в другое место. В этом случае у вас окажется две записи для одного и того же ключа или вы не сможете найти ключ вообще. В любом случае словарь не будет работать правильно.

Вот почему ключи должны быть хешируемыми, а изменяемые типы, такие как списки, не могут быть ключами. Простейший способ обойти это ограничение — использовать кортежи, которые мы рассмотрим в следующей главе.

Поскольку словари изменяемы, их нельзя использовать в качестве ключей, но можно — в качестве значений.

ЗНАЧЕНИЯ MEMO

Если вы экспериментировали с функцией fibonacci() из раздела «Еще один пример» главы 6, вы могли заметить, что чем больше передаваемый аргумент, тем дольше выполняется функция. Более того, время выполнения быстро увеличивается.

Чтобы понять почему, рассмотрим рис. 11.2, на котором показан **граф вызова (call graph)** для функции fibonacci() с n = 4.

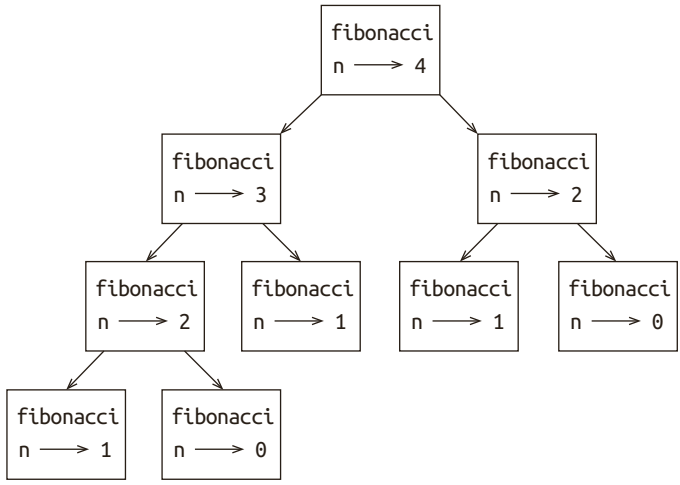


Рис. 11.2. Граф вызова

На графе вызова показаны фреймы функций со стрелками, соединяющими каждый фрейм с фреймами вызываемых функций. На вершине графа функция fibonacci() для n = 4, которая вызывает fibonacci() для n = 3 и n = 2. В свою очередь, функция fibonacci() для n = 3 вызывает fibonacci() для n = 2 и n = 1. И так далее.

Посчитайте, сколько раз вызываются функции fibonacci(0) и fibonacci(1). Это неэффективное решение проблемы, и все становится еще хуже по мере роста значения аргумента.

Одно из решений — отслеживать уже вычисленные значения путем сохранения их в словаре. Ранее вычисленное значение, которое сохраняется для последующего использования, называется **memo (от англ. memory)**. Ниже показана версия функции fibonacci() с memo:

```
known = {0:0, 1:1}
```

```
def fibonacci(n):
    if n in known:
        return known[n]

    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

Словарь known отслеживает числа Фибоначчи, которые мы уже знаем. Он начинается с двух пунктов: 0 к 0 и 1 к 1.

Всякий раз при вызове функции fibonacci() она проверяет словарь known. Если результат уже есть, его можно вернуть немедленно. В противном случае функция должна вычислить новое значение, добавить его в словарь, а затем вернуть его.

Если вы запустите эту версию функции fibonacci() и сравните ее с оригинальной, то обнаружите, что она намного быстрее.

ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

В предыдущем примере словарь known создается вне функции, поэтому он принадлежит специальному фрейму `__main__`. Переменные в `__main__` называют **глобальными (global)**, потому что к ним можно получить доступ из любой функции. В отличие от локальных переменных, которые стираются, когда функция завершает выполнение, глобальные переменные сохраняются между вызовами функции.

Обычно глобальные переменные используются для **флагов (flags)** — логических переменных, которые указывают, истинно ли условие. Некоторые программы используют флаг verbose для управления уровнем детализации в выводе:

```
verbose = True

def example1():
    if verbose:
        print('Выполнение example1')
```

Если вы попытаетесь переназначить глобальную переменную, то получите неожиданный эффект. В следующем примере должно отслеживаться, была ли вызвана функция:

```
been_called = False
```

```
def example2():
    been_called = True        # НЕПРАВИЛЬНО!
```

Но если запустите его, вы увидите, что значение переменной `been_called` не меняется. Проблема в том, что `example2` создает новую локальную переменную `been_called`. Локальная переменная исчезает после завершения функции и не влияет на глобальную переменную.

Чтобы переназначить глобальную переменную внутри функции, перед ее использованием вы должны объявить, что эта переменная глобальная:

```
been_called = False

def example2():
    global been_called
    been_called = True
```

Инструкция `global` сообщает интерпретатору что-то вроде: «В этой функции под `been_called` я подразумеваю глобальную переменную, а не локальную».

Ниже показан пример кода с попыткой обновить значение глобальной переменной:

```
count = 0

def example3():
    count = count + 1         # НЕПРАВИЛЬНО
```

Если вы попытаетесь его запустить, то увидите:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python предполагает, что переменная `count` локальна, а согласно этому предположению вы считываете значение переменной, прежде чем задать его. Решение, опять же, заключается в объявлении переменной `count` глобальной:

```
def example3():
    global count
    count += 1
```

Если глобальная переменная ссылается на изменяемое значение, вы можете изменить значение, не объявляя переменную:

```
known = {0:0, 1:1}

def example4():
    known[2] = 1
```

Таким образом, можно добавлять, удалять и заменять элементы глобального списка или словаря, но, если вы хотите переназначить переменную, вы должны объявить ее:

```
def example5():
    global known
    known = dict()
```

Глобальные переменные полезны, но, если их много и вы часто их модифицируете, отладка программ усложнится.

ОТЛАДКА

При работе с большими наборами данных печать и проверка выходных данных вручную — непосильная задача. Вот несколько советов по отладке больших объемов данных.

Уменьшите объем входных данных

Если возможно, уменьшите размер набора данных. Например, если программа считывает текстовый файл, начните с первых 10 строк или с наименьшего примера, какой найдете. Вы можете либо отредактировать сами файлы, либо (и это лучше) изменить программу, чтобы она читала только первые *n* строк.

Если есть ошибка, вы можете изменить значение *n* на наименьшее из тех, что вызывает ошибку, а затем постепенно увеличивать его по мере обнаружения и исправления ошибок.

Проверьте сводные данные и типы

Вместо того чтобы печатать и проверять весь набор данных, рассмотрите возможность печати сводных данных: например, количества элементов в словаре или суммы набора чисел.

Распространенная причина ошибок во время выполнения — значения неправильного типа. Для поиска такого рода ошибок часто достаточно вывести тип значения.

Напишите тесты для самопроверки

Попробуйте написать код для автоматического поиска ошибок. Например, если вы вычисляете среднее значение для набора чисел, вы можете проверить, что результат не больше самого большого элемента в списке или не меньше самого маленького. Это называется «проверкой адекватности», поскольку она обнаруживает «неадекватные» результаты.

Другой вид проверки — сравнить результаты двух разных вычислений, чтобы увидеть, согласуются ли они. Это называется «проверка согласованности».

Отформатируйте вывод

Форматирование результатов отладки упростит обнаружение ошибки. Вы видели пример в разделе «Отладка» главы 6. Модуль `pprint()` предоставляет одноименную функцию, которая отображает встроенные типы в более удобном формате. Добавим, что `pprint` — это сокращение от pretty print.

Опять же, время, которое вы тратите на написание отладочного кода, может сократить время, которое вы тратите на саму отладку.

СЛОВАРЬ ТЕРМИНОВ

Сопоставление (маппинг):

Отношение, в котором каждый элемент одного набора соответствует элементу другого набора.

Словарь:

Сопоставление ключей с соответствующими значениями.

Пара «ключ — значение»:

Представление сопоставления ключа со значением.

Элемент:

В отношении словарей — другое название пары «ключ — значение».

Ключ:

Объект, указанный в словаре как первая часть пары «ключ — значение».

Значение:

Объект, указанный в словаре как вторая часть пары «ключ — значение». Это более конкретно, чем наше предыдущее использование слова «значение».

Реализация:

Способ выполнения вычислений.

Хеш-таблица:

Структура данных, используемая для реализации словарей Python.

Хеш-функция:

Функция, используемая хеш-таблицей для вычисления местоположения ключа.

Хешируемый:

Тип, к которому можно применить хеш-функцию. Неизменяемые типы, такие как целые числа, числа с плавающей точкой и строки, — хешируемые; а изменяемые типы, такие как списки и словари, не являются таковыми.

Поиск:

Метод, операция над словарем, которая на основе ключа находит соответствующее значение.

Обратный поиск:

Алгоритм словаря, которая принимает значение и находит один или несколько ключей, сопоставленных с ним.

Инструкция `raise`*:*

Инструкция, которая (намеренно) вызывает исключение.

Синглтон:

Список или другая последовательность с одним элементом.

Граф вызовов:

Диаграмма, показывающая каждый фрейм, созданный во время выполнения программы, со стрелкой от каждого вызывающего к каждому вызываемому.

Мето:

Вычисленное значение, сохраняемое во избежание ненужных вычислений в будущем.

Глобальная переменная:

Переменная, определенная за пределами функции. Глобальные переменные доступны в любой функции.

Инструкция `global`*:*

Инструкция, который объявляет, что используется глобальная переменная.

Флаг:

Логическая переменная, используемая для индикации истинности условия.

Объявление:

Инструкция наподобие `global`, которая сообщает интерпретатору сведения о переменной.

УПРАЖНЕНИЯ

Упражнение 11.1

Напишите функцию, которая читает слова из файла *words.txt* и сохраняет в виде ключей в словаре. Неважно, какие будут значения. Затем с помощью оператора `in` можно быстро проверить, есть ли строка в словаре.

Если вы выполнили упражнение 10.10, вы можете сравнить скорость этой реализации с оператором `in` в списке и с поиском делением пополам (бинарным поиском).

Упражнение 11.2

Прочитайте документацию по методу словаря `setdefault()` и используйте его для написания более краткой версии функции `invert_dict()`.

Решение: **thinkpython2.com/code/invert_dict.py**.

Упражнение 11.3

Вспомните функцию Аккермана из упражнения 6.2 и посмотрите, позволяет ли memo ускорить вычисление функцию с большими аргументами. Подсказка: нет.

Решение: **thinkpython2.com/code/ackermann_memo.py**.

Упражнение 11.4

Если вы выполнили упражнение 10.7, у вас уже есть функция `has_duplicates()`, которая принимает список в качестве параметра и возвращает `True`, если какой-либо объект встречается в списке более одного раза.

Используйте словарь, чтобы написать более быструю и простую версию функции `has_duplicates()`.

Решение: **thinkpython2.com/code/has_duplicates.py**.

Упражнение 11.5

Два слова являются «парами ротации», если вы можете повернуть одно из них и получить другое (функция `rotate_word()` из упражнения 8.5).

Напишите программу, которая читает список слов и находит все пары ротации.

Решение: **thinkpython2.com/code/rotate_pairs.py**.

Упражнение 11.6

Вот еще одно задание с сайта Car Talk (**www.cartalk.com/content/puzzlers**).

> Это упражнение мне отправил парень по имени Дэн О'Лири. Недавно он натолкнулся на обычное пятибуквенное слово из одного слога, обладающее следующим уникальным свойством. Если убрать первую букву, оставшиеся образуют омофон исходного слова, то есть слово, которое звучит точно так же. Если заменить первую букву, то есть вернуть ее обратно и удалить вторую букву, то в результате получится еще один омофон исходного слова. И вопрос в том, что это за слово.
>
> Сейчас я приведу пример, который не работает. Давайте посмотрим слово из пяти букв wrack. WRACK обычно используется в словосочетании wrack with pain («биться от боли»). Если я уберу первую букву, у меня останется четырехбуквенное слово RACK. Его можно перевести как «стойка». Например: "Holy cow, did you see the rack on that buck! It must have been a nine-pointer!" Это идеальный омофон. Если вы вернете w и удалите r, то вы получите слово wack, которое является настоящим словом, это просто не омофон двух предыдущих слов.

Но есть, по крайней мере, одно слово, известное и Дэну, и нам, которое даст два омофона, если вы удалите одну из первых двух букв, и образует два новых четырехбуквенных слова. И вопрос в том, что это за слово.

Вы можете использовать словарь из упражнения 11.1, чтобы проверить, есть ли строка в списке слов.

Чтобы проверить, являются ли два слова омофонами, вы можете использовать словарь произношения CMU. Вы можете скачать его с сайта **www.speech.cs.cmu.edu/cgi-bin/cmudict** или **thinkpython2.com/code/c06d**. Еще можно загрузить файл **thinkpython2.com/code/pronounce.py** с функцией `read_dictionary()`. Она читает словарь произношения и возвращает словарь Python, который сопоставляет каждое слово со строкой, описывающей его произношение.

Напишите программу, в которой перечислены все слова — решения этой головоломки.

Решение: **thinkpython2.com/code/homophone.py**.

ГЛАВА 12
КОРТЕЖИ

В этой главе представлен еще один встроенный тип — кортеж, а затем показано, как списки, словари и кортежи работают вместе. Я также продемонстрирую одну хитрость при работе с функциями, которые могут принимать переменное число аргументов: операторы сборки и разбивки.

В устной речи кортежи часто называют «тапл» или «тупл» от английского tuple. Забавно, что даже среди носителей языка нет единого мнения о правильном произношении. В контексте программирования чаще используется первый вариант.

КОРТЕЖИ — НЕИЗМЕНЯЕМЫЙ ТИП ДАННЫХ

Кортеж — это последовательность значений. Значения могут быть любого типа, а их индексы — целыми числами, и в этом отношении кортежи похожи на списки. Важное отличие заключается в том, что кортежи неизменяемы.

Синтаксически кортеж — это список значений, разделенных запятой:

```
>>> t = 'а', 'б', 'в', 'г', 'д'
```

Хотя в этом нет необходимости, кортежи обычно помещают в скобки:

```
>>> t = ('а', 'б', 'в', 'г', 'д')
```

Чтобы создать кортеж с одним элементом, необходимо добавить заключительную запятую:

```
>>> t1 = 'а',
>>> type(t1)
<class 'tuple'>
```

Простое помещение одного значения в скобки не делает его кортежем:

```
>>> t2 = ('а')
>>> type(t2)
<class 'str'>
```

Еще один способ создания кортежа — это встроенная функция `tuple()`. Без аргументов она создает пустой кортеж:

```
>>> t = tuple()
>>> t
()
```

Если аргумент является последовательностью (строка, список или кортеж), результатом будет кортеж с элементами последовательности:

```
>>> t = tuple('люпины')
>>> t
('л', 'ю', 'п', 'и', 'н', 'ы')
```

Поскольку `tuple()` — это имя встроенной функции, вам не следует использовать его в качестве имени переменной.

Большинство операторов списков также работают и с кортежами. В квадратных скобках указывается индекс элемента:

```
>>> t = ('а', 'б', 'в', 'г', 'д')
>>> t[0]
'а'
```

И оператор среза возвращает диапазон элементов:

```
>>> t[1:3]
('б', 'в')
```

Но если вы попытаетесь изменить один из элементов кортежа, вы получите ошибку:

```
>>> t[0] = 'А'
TypeError: object doesn't support item assignment
```

Изменять кортежи нельзя, но можно заменить один кортеж другим:

```
>>> t = ('А',) + t[1:]
>>> t
('А', 'б', 'в', 'г', 'д')
```

Этот код создает новый кортеж, а затем присваивает переменной `t` ссылку на него.

Операторы сравнения работают с кортежами так же, как и с другими последовательностями; Python начинает сравнение с первого элемента каждой последовательности. Если они равны, он переходит к следующим элементам и так далее, пока не найдет отличающиеся элементы. Последующие элементы не учитываются (даже если они большие).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

ПРИСВАИВАНИЕ ЗНАЧЕНИЯ КОРТЕЖА

Иногда нужно поменять местами значения двух переменных. При обычном присваивании вы должны использовать временную переменную. Например, чтобы поменять местами значения переменных `a` и `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Это громоздкое решение; **присваивание значения кортежа (tuple assignment)** более элегантно:

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

Слева — кортеж переменных; справа — кортеж выражений. Каждое значение присваивается соответствующей переменной. Все выражения с правой стороны вычисляются перед тем, как происходит присваивание.

Количество переменных слева и количество значений справа должно быть одинаковым:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

Обобщая, правая сторона может быть любой последовательностью (строка, список или кортеж). Например, чтобы разделить адрес электронной почты на имя пользователя и домен, вы можете написать:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

Возвращаемое функцией `split()` значение представляет собой список с двумя элементами; первый элемент назначается переменной `uname`, второй — переменной `domain`:

```
>>> uname
'monty'
>>> domain
'python.org'
```

КОРТЕЖИ КАК ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

Строго говоря, функция может возвращать только одно значение. Но если значение является кортежем, эффект такой, будто возвращается несколько значений. Например, если вы хотите разделить два целых числа и вычислить частное и остаток, вычислить `x/y` и затем `x%y` неэффективно. Лучше вычислять оба значения одновременно.

Встроенная функция `divmod()` принимает два аргумента и возвращает кортеж из двух значений: частное и остаток. Вы можете сохранить результат как кортеж:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

Или присваивайте значение кортежа для хранения элементов отдельно:

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

Ниже показан пример функции, которая возвращает кортеж:

```
def min_max(t):
    return min(t), max(t)
```

Функции `max()` и `min()` — встроенные, они находят самый большой и самый маленький элементы последовательности. Функция `min_max()` вычисляет оба и возвращает кортеж из двух значений.

КОРТЕЖИ С ПЕРЕМЕННЫМ ЧИСЛОМ АРГУМЕНТОВ

Функции могут принимать разное количество аргументов. Имя параметра, начинающееся с `*`, **оператора сборки**, собирает аргументы в кортеж. Например, функция `printall()` принимает любое количество аргументов и печатает их:

```
def printall(*args):
    print(args)
```

Параметр сборки может называться как вам нравится, но обычно используют `args`. Вот как работает эта функция:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

Обратным процессом сборки является **разбивка (scatter)**. Если у вас есть последовательность значений и вы хотите передать ее функции в виде нескольких аргументов, используйте оператор *. Например, функция `divmod()` принимает ровно два аргумента; но не работает с кортежем:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

Но если вы разобьете кортеж, это сработает:

```
>>> divmod(*t)
(2, 1)
```

Многие встроенные функции используют кортежи аргументов переменной длины. Например, функции `max()` и `min()` могут принимать любое количество аргументов:

```
>>> max(1, 2, 3)
3
```

А `sum()` — нет:

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

В качестве упражнения напишите функцию `sumall()`, которая принимает любое количество аргументов и возвращает их сумму.

СПИСКИ И КОРТЕЖИ

`zip()` — это встроенная функция, которая принимает две или более последовательности и возвращает список кортежей, где каждый кортеж содержит по одному элементу из каждой последовательности. Имя функции переводится как застежка-молния, которая как бы соединяет и чередует два ряда зубцов.

В примере ниже «застегивается» строка и список:

```
>>> s = 'абв'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x039A53C8>
```

Результатом будет **zip-объект**, способный перебирать пары. Наиболее часто функция `zip()` применяется в цикле `for`:

```
>>> for pair in zip(s, t):
...     print(pair)
...
('а', 0)
('б', 1)
('в', 2)
```

Zip-объект — это своего рода **итератор (iterator)**, который представляет собой любой объект, обходящий последовательность. Итераторы по-своему похожи на списки, но, в отличие от списков, вы не можете использовать индекс для выбора элемента из итератора.

Если вы хотите использовать операторы и методы списков, вы можете использовать zip-объект для создания списка:

```
>>> list(zip(s, t))
[('а', 0), ('б', 1), ('в', 2)]
```

Результатом будет список кортежей; в этом примере каждый кортеж содержит символ из строки и соответствующий элемент из списка.

Если последовательности разной длины, результат имеет длину более короткой из них:

```
>>> list(zip('Анна', 'Кларк'))
[('А', 'К'), ('н', 'л'), ('н', 'а'), ('а', 'р')]
```

Вы можете использовать кортежи в цикле `for` для обхода списка кортежей:

```
t = [('а', 0), ('б', 1), ('в', 2)]
for letter, number in t:
    print(number, letter)
```

Каждый раз в цикле Python выбирает следующий кортеж в списке и присваивает элементы переменным `letter` и `number`. Вывод этого цикла:

```
0 a
1 b
2 c
```

Если вы скомбинируете функцию `zip()`, цикл `for` и присваивание кортежей, вы получите полезную идиому для прохождения двух (или более) последовательностей одновременно. Например, функция `has_match()` принимает

две последовательности, t1 и t2, и возвращает True, если существует индекс i такой, что t1[i] == t2[i]:

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

Если вам нужно просмотреть элементы последовательности и их индексы, вы можете использовать встроенную функцию enumerate():

```
for index, element in enumerate('абв'):
    print(index, element)
```

Результатом выполнения функции enumerate() будет объект перечисления, который повторяет последовательность пар; каждая пара содержит индекс (начиная с 0) и элемент из данной последовательности. Для этого примера вывод будет такой:

```
0 а
1 б
2 в
```

И снова.

СЛОВАРИ И КОРТЕЖИ

К словарям применим метод items(), который возвращает последовательность кортежей, где каждый кортеж представлен парой «ключ — значение»:

```
>>> d = {'а':0, 'б':1, 'в':2}
>>> t = d.items()
>>> t dict_items([('а', 0), ('б', 1), ('в', 2)])
```

Результатом будет объект dict_items, который служит итератором для пар «ключ — значение».

Вы можете использовать его в цикле for:

```
>>> for key, value in d.items():
...     print(key, value)
а 0
б 1
в 2
```

Или, наоборот, вы можете использовать список кортежей для инициализации нового словаря:

```
>>> t = [('а', 0), ('в', 2), ('б', 1)]
>>> d = dict(t)
>>> d
{'а': 0, 'в': 2, 'б': 1}
```

Сочетание функций `dict()` и `zip()` дает краткий способ создания словаря:

```
>>> d = dict(zip('абв', range(3)))
>>> d
{'а': 0, 'б': 1, 'в': 2}
```

Словарный метод `update()` также принимает список кортежей и добавляет их в виде пар «ключ — значение» в существующий словарь.

Обычно в словарях в качестве ключей используются кортежи (в основном потому, что вы не можете использовать списки). Например, телефонный справочник может связывать пару фамилии и имени с телефонным номером. Предполагая, что мы определили переменные `last`, `first` и `number`, мы могли бы написать:

```
directory[last, first] = number
```

Выражение в скобках — кортеж. Мы могли бы использовать присвоение кортежей, чтобы обойти этот словарь:

```
for last, first in directory:
    print(first, last, directory[last,first])
```

Этот цикл перебирает ключи в словаре, которые представлены кортежами. Он присваивает элементы каждого кортежа переменным `last` и `first`, а затем печатает имя и соответствующий номер телефона.

Существует два способа представления кортежей на диаграмме состояний. Более подробная версия показывает индексы и элементы в том виде, в котором они отображаются в списке. Например, кортеж (`'Иванов', 'Иван'`) будет выглядеть так, как показано на рис. 12.1.

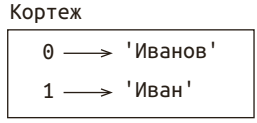


Рис. 12.1. Диаграмма состояния

На более крупной диаграмме вы можете не указывать детали. Например, диаграмма телефонного справочника может выглядеть так, как показано на рис. 12.2.

Рис. 12.2. Диаграмма состояний

Здесь кортежи показаны с использованием синтаксиса Python в качестве графического сокращения. Номер телефона на диаграмме — горячая линия жалоб BBC, пожалуйста, не звоните туда!

ПОСЛЕДОВАТЕЛЬНОСТИ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

Я сосредоточился на списках кортежей, но почти все примеры в этой главе выполнимы и со списками списков, кортежами кортежей и кортежами списков. Чтобы избежать перечисления возможных комбинаций, иногда проще говорить о последовательностях последовательностей.

Во многих контекстах различные виды последовательностей (строки, списки и кортежи) взаимозаменяемы. Как же выбрать нужный?

Начнем с очевидного: строки более ограничены, чем другие последовательности, потому что элементы должны быть символами. А еще строки неизменяемы. Если вам нужно изменять символы в строке (а не создавать новую строку), лучше вместо строк использовать список символов.

Списками пользуются чаще, чем кортежами, в основном потому, что они изменяемые. Но иногда стоит предпочесть кортежи.

1. В некоторых контекстах, например в инструкции `return`, синтаксически проще создать кортеж, чем список.
2. Если вы хотите использовать последовательность в качестве ключа словаря, вы должны использовать неизменяемый тип, такой как кортеж или строка.

3. Если вы передаете последовательность в качестве аргумента функции, использование кортежей снижает вероятность непредвиденного поведения из-за псевдонимов.

Поскольку кортежи неизменяемы, они не предоставляют такие методы, как `sort()` и `reverse()`, которые изменяют существующие списки. Но Python предоставляет встроенную функцию `sorted()`, принимающую любую последовательность и возвращающую список с теми же элементами в отсортированном порядке, и функцию `reversed()`, которая принимает последовательность и возвращает итератор, с теми же элементами, но в обратном порядке.

ОТЛАДКА

Списки, словари и кортежи — примеры **структур данных (data structures)**. В этой главе вы встретили *составные* структуры данных, такие как списки кортежей и словари, где использованы кортежи в качестве ключей и списки в качестве значений. Составные структуры данных удобны, но в них легко допустить ошибки, которые я называю **ошибками формы (shape errors)**, то есть ошибки, вызванные неправильным типом, размером или структурой структуры данных. Например, если вы ожидаете список с одним целым числом, а я дам вам целое число (не в списке), функция не будет работать.

Чтобы помочь отладить подобные ошибки, я написал модуль `structshape`, который предоставляет одноименную функцию, принимающую любой тип структуры данных в качестве аргумента и возвращающую строку с информацией о форме структуры данных. Вы можете скачать его с **thinkpython2.com/code/structshape.py**.

Ниже показан результат работы этой функции для простого списка:

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
'list of 3 int'
```

Можно было бы ее улучшить, чтобы она могла вывести `'list of 3 ints'`, но проще не иметь дело со множественным числом. Ниже показан список списков:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

Если элементы списка не одного типа, функция `structshape()` группирует их по порядку типов:

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

Вот список кортежей:

```
>>> s = 'абв'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

А вот словарь с тремя элементами, который сопоставляет целые числа со строками:

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

Если у вас возникли проблемы с отслеживанием типов ваших структур данных, функция `structshape()` может помочь.

СЛОВАРЬ ТЕРМИНОВ

Кортеж:
Неизменяемая последовательность элементов.

Присваивание значения кортежа:
Присваивание с последовательностью справа и кортежем переменных слева. Правая сторона вычисляется, а затем ее элементы присваиваются переменным слева.

Сборка:
Операция сборки кортежа как аргумента переменной длины.

Разбивка:
Операция разделения последовательности на список отдельных аргументов.

Zip-объект:
Результат вызова встроенной функции `zip()`; объект, который проходит через последовательность кортежей.

Итератор:

Объект, который может обходить последовательность, но не предоставляет инструкций и методов списка.

Структура данных:

Коллекция связанных значений, часто организованная в списки, словари, кортежи и тому подобное.

Ошибка формы:

Ошибка, вызванная тем, что значение имеет неправильную форму, то есть неправильный тип или размер.

УПРАЖНЕНИЯ

Упражнение 12.1

Напишите функцию `most_frequent()`, которая принимает строку и печатает буквы в порядке убывания их частотности. Найдите образцы текста на разных языках и посмотрите, как различается частотность букв в разных языках. Сравните ваши результаты с таблицами на странице **ru.wikipedia.org/wiki/Частотность**.

Решение: **thinkpython2.com/code/most_frequent.py**.

Упражнение 12.2

Больше анаграмм!

1. Напишите программу, которая считывает список слов из файла (см. раздел «Чтение списка слов» главы 9) и печатает все наборы слов-анаграмм.

Ниже показан пример результата:

```
['товар', 'тавро', 'автор', 'отвар', 'рвота']
['австралопитек', 'ватерполистка']
['покраснение', 'пенсионерка']
['декор', 'докер', 'кредо']
```

Подсказка: вы можете создать словарь, который будет сопоставлен (маппирован) из набора букв в список слов, которые можно записать этими буквами. Вопрос в том, как вы можете представить коллекцию букв так, чтобы их можно было использовать в качестве ключа.

2. Измените предыдущую программу, чтобы она сначала печатала самый длинный список анаграмм, затем второй по длине и так далее.

3. В игре «Скрэббл» ситуация «бинго» — это когда вы из всех семи фишек на руках вместе с буквой на доске составляете восьмибуквенное слово. Какие наборы из восьми букв наиболее вероятно приведут к «бинго»? Подсказка: их семь.

Решение: **thinkpython2.com/code/anagram_sets.py**.

Упражнение 12.3

Два слова образуют «парную метаграмму», если вы можете превратить одно в другое, поменяв две буквы, например «тесто» и «месть». Напишите программу, которая находит все парные метаграммы в словаре. Подсказка: не проверяйте все пары слов и не проверяйте все возможные перестановки.

Решение: **thinkpython2.com/code/metathesis.py**. Примечание: на это упражнение меня вдохновил пример на сайте **puzzlers.org**.

Упражнение 12.4

Вот еще одно задание с сайта Car Talk (**www.cartalk.com/content/puzzlers**).

> Определите самое длинное английское слово, которое остается словарным английским словом, если удалять из него буквы по одной.
>
> При этом буквы могут быть удалены с любого конца или из середины, но их нельзя переставлять местами. Каждый раз, когда вы удаляете букву, должно получаться другое английские слово. Если вы сделаете всё правильно, в итоге получите одну букву, и это тоже будет английское слово, которое встречается в словаре. Я хочу знать: какое такое слово самое длинное и сколько в нем букв?
>
> Я приведу небольшой скромный пример: *Sprite*. Хорошо? Вы начинаете со *sprite*, удаляете букву *r* из середины слова, остается *spite*, затем удаляем *e* с конца, остается *spit*. Удаляем *s* — получаем *pit*, затем *it* и *i*.

Напишите программу, позволяющую найти все слова, которые можно уменьшить таким образом, а затем найдите самое длинное.

Это упражнение немного сложнее прочих, поэтому вот несколько советов.

1. Возможно, вы захотите написать функцию, которая берет слово и вычисляет список всех слов, которые могут быть сформированы путем удаления одной буквы. Получатся «потомки» этого слова.
2. Рекурсивно слово подходит, если любой из его потомков удовлетворяет условию. В качестве базового случая вы можете взять пустую строку.

3. Предоставленный мной список слов в файле *words.txt* не содержит однобуквенных слов. Поэтому вы можете добавить слова “I”, “a” и пустую строку.
4. Чтобы повысить производительность вашей программы, вы можете запомнить слова, которые, как уже выяснилось, подходят.

Решение: **thinkpython2.com/code/reducible.py**.

ГЛАВА 13
ПРАКТИЧЕСКИЙ ПРИМЕР: ВЫБОР СТРУКТУРЫ ДАННЫХ

Вы уже познакомились с основными структурами данных Python и увидели некоторые алгоритмы, где их можно использовать. Если вы хотите больше узнать об алгоритмах, самое время прочитать главу 21. Хотя вы и без нее поймете материал этой главы. Так что читайте, когда захотите.

В этой главе представлены примеры упражнений, в которых нужно самим выбрать структуры данных, а еще они позволят вам практиковаться в использовании этих структур.

ЧАСТОТНЫЙ АНАЛИЗ СЛОВ

Как обычно, вы должны по крайней мере попытаться решить каждое из упражнений, прежде чем читать решения.

Упражнение 13.1
Напишите программу, которая считывает содержимое файла, разбивает каждую строку на слова, удаляет пробелы и знаки препинания из слов и преобразует их в строчные.

Подсказка. Модуль `string()` предоставляет строку `whitespace`, которая содержит непечатаемые символы: пробел, отступ, символ новой строки и тому подобное, и строку `punctuation`, в которой содержатся знаки пунктуации. Давайте посмотрим, сможем ли мы заставить Python выругаться:

```
>>> import string
>>> string.punctuation
'!”#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

Кроме того, можно использовать строковые методы `strip()`, `replace()` и `translate()`.

Упражнение 13.2
Посетите сайт проекта Gutenberg (**gutenberg.org**) с книгами, не защищенными авторским правом, и скачайте вашу любимую книгу в текстовом формате.

Измените программу из предыдущего упражнения так, чтобы она читала загруженную книгу, пропускала заголовок в начале файла и обрабатывала оставшиеся слова, как и в предыдущем упражнении.

Затем измените программу, чтобы подсчитать общее количество слов в книге и количество вхождений каждого слова.

Выведите количество разных слов, использованных в книге. Сравните разные книги разных авторов, написанные в разные эпохи. У кого из авторов самый большой словарный запас?

Упражнение 13.3
Измените программу из предыдущего упражнения, чтобы напечатать 20 наиболее часто используемых в книге слов.

Упражнение 13.4
Измените предыдущую программу, чтобы прочитать список слов (см. раздел «Чтение списка слов» главы 9), а затем выведите все слова книги, которых нет в этом списке. Сколько из них являются опечатками? Сколько из них распространенные слова, которые должны быть в списке, а сколько действительно непонятные?

СЛУЧАЙНЫЕ ЧИСЛА

При одних и тех же входных данных большинство компьютерных программ каждый раз генерируют одни и те же выходные данные, которые называют **детерминированными (deterministic)**. Детерминированность, как правило, хорошая вещь, так как логично, что одно и то же вычисление дает всегда один и тот же результат. Однако для некоторых приложений необходимо, чтобы компьютер был непредсказуемым. Очевидный, но не единственный пример — игры.

Сделать программу по-настоящему недетерминированной сложно, но существуют способы сделать ее почти таковой. Один из способов — использование алгоритмов, генерирующих **псевдослучайные (pseudorandom)** числа. Псевдослучайные числа на самом деле не истинно случайны, так как генерируются посредством детерминированных вычислений, но отличить их на глазок от случайных практически невозможно.

Модуль `random` предоставляет функции, которые генерируют псевдослучайные числа (далее я буду называть их «случайными»).

Функция `random()` возвращает случайное значение с плавающей точкой в диапазоне от `0.0` до `1.0` (включая `0.0` и исключая `1.0`). Каждый раз, когда вы вызываете функцию `random()`, вы получаете следующее число из длинной последовательности. Чтобы попрактиковаться, выполните показанный ниже цикл:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Функция `randint()` принимает параметры `low` и `high` и возвращает целое число в диапазоне от `low` до `high` (включая оба):

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Чтобы выбрать случайный элемент из последовательности, вы можете использовать функцию `choice()`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

Модуль `random` также предоставляет функции для генерации случайных значений из непрерывных распределений, включая распределение по Гауссу, экспоненциальное, гамма-распределение и некоторые другие.

Упражнение 13.5

Напишите функцию `choose_from_hist()`, которая принимает гистограмму, определенную в разделе «Словарь как набор счетчиков» главы 11, и возвращает случайное значение из гистограммы, вероятность выбора которого пропорциональна частотности. Например, для этой гистограммы:

```
>>> t = ['a', 'a', 'б']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'б': 1}
```

Ваша функция должна возвращать букву «а» с вероятностью ⅔ и «б» с вероятностью ⅓.

ГИСТОГРАММА СЛОВ

Вы должны выполнить предыдущие упражнения, прежде чем переходить к этому разделу. Вы можете скачать мое решение по адресу **thinkpython2.com/code/analyze_book1.py**. Вам также понадобится файл **thinkpython2.com/code/emma.txt**.

Ниже показан код программы, которая читает файл и строит гистограмму слов для него:

```
import string

def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist

def process_line(line, hist):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

Эта программа читает файл *emma.txt* с текстом романа «Эмма» Джейн Остин.

Функция `process_file()` проходит по строкам файла, передавая их по одной функции `process_line()`. Гистограмма `hist` используется как счетчик.

В функции `process_line()` используется строковый метод `replace()`, позволяющий заменить дефисы пробелами, прежде чем использовать метод `split()`, разбивающий строку на список. Программа также обходит список слов и использует методы `strip()` и `lower()`, чтобы удалить знаки препинания и преобразовать все буквы в строчные. (Это значит, что строки «конвертируются»; помните, что строки неизменяемы, поэтому такие методы, как `strip()` и `lower()`, возвращают новые строки.)

Наконец, функция process_line() обновляет гистограмму, создавая новый элемент или увеличивая значение существующего.

Чтобы подсчитать общее количество слов в файле, мы можем сложить частотности в гистограмме:

```
def total_words(hist):
    return sum(hist.values())
```

Количество разных слов — это просто количество элементов в словаре:

```
def different_words(hist):
    return len(hist)
```

Ниже показан код, который выводит результат:

```
print('Итоговое количество слов:', total_words(hist))
print('Количество разных слов:', different_words(hist))
```

И сам результат:

```
Итоговое количество слов: 161080
Количество разных слов: 7214
```

САМЫЕ РАСПРОСТРАНЕННЫЕ СЛОВА

Чтобы найти наиболее распространенные слова, мы можем составить список кортежей, где каждый кортеж содержит слово и его частотность, и отсортировать его.

Показанная ниже функция принимает гистограмму и возвращает список кортежей частотности слов:

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

В каждом кортеже частотность указывается первой, поэтому результирующий список сортируется по частотности.

Ниже показан цикл, который печатает 10 самых распространенных слов:

```
t = most_common(hist)
print('Самые распространенные слова:')
for freq, word in t[:10]:
    print(word, freq, sep='\t')
```

Я использую ключевой аргумент `sep`, чтобы указать функции `print()` использовать в качестве «разделителя» символ отступа, а не пробел, поэтому второй столбец выстроен в ровный ряд. Ниже показаны результаты для романа «Эмма»:

```
Самые распространенные слова:
to    5242
the   5205
and   4897
of    4295
i     3191
a     3130
it    2529
her   2483
was   2400
she   2364
```

Этот код можно упростить с помощью параметра `key` функции `sort()`. Если вам интересно, как это работает, обратитесь к странице **wiki.python.org/moin/HowTo/Sorting**.

НЕОБЯЗАТЕЛЬНЫЕ ПАРАМЕТРЫ

Вы видели встроенные функции и методы, которые принимают необязательные аргументы. Можно также писать собственные функции с необязательными аргументами. В качестве примера ниже показана функция, которая печатает наиболее распространенные слова в виде гистограммы:

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print('Самые распространенные слова:')
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

Первый параметр обязателен; второй — нет. **По умолчанию (default value)** переменной `num` присвоено значение `10`.

Если вы передадите только один аргумент:

```
print_most_common(hist)
```

переменной `num` будет присвоено значение по умолчанию. Если вы передадите два аргумента:

```
print_most_common(hist, 20)
```

то переменная num получает значение второго аргумента. Другими словами, необязательный аргумент изменяет значение по умолчанию.

Если функция имеет как обязательные, так и необязательные параметры, все обязательные параметры должны быть указаны первыми, а необязательные — вторыми.

ВЫЧИТАНИЕ СЛОВАРЕЙ

Поиск в книге слов, отсутствующих в файле *words.txt*, — проблема, которую можно определить как **вычитание множеств**. То есть мы хотим найти все слова из одного набора (слова в книге), которых нет в другом (слова в списке).

Функция subtract() принимает словари d1 и d2 в качестве параметров и возвращает новый словарь, который содержит все ключи из d1, которых нет в d2. Так как значения нам совершенно не важны, мы установим их все в None:

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

Чтобы найти в книге слова, которых нет в файле *words.txt*, мы можем использовать функцию process_file() для построения гистограммы для файла *words.txt*, а затем вычесть одну из другой:

```
words = process_file('words.txt')
diff = subtract(hist, words)

print("Слова из книги, которых нет в списке слов:")
for word in diff:
    print(word, end=' ')
```

Ниже показан фрагмент результата для романа «Эмма»:

```
Слова из книги, которых нет в списке слов: rencontre jane's blanche woodhouses
disingenuousness friend's venice apartment ...
```

Некоторые из этих слов — имена и слова в притяжательном падеже. Другие, такие как rencontre, устарели. Но есть несколько распространенных слов, которые действительно должны быть в списке!

Упражнение 13.6

В Python есть структура данных `set` (множество), которая предоставляет распространенные операции над множествами. О них можно прочитать в разделе «Множества» главы 19 или в документации по адресу **docs.python.org/3/library/stdtypes.html#types-set**.

Напишите программу, которая использует вычитание множеств, чтобы найти слова в книге, которых нет в списке слов.

Решение: **thinkpython2.com/code/analyze_book2.py**.

СЛУЧАЙНЫЕ СЛОВА

Чтобы выбрать случайное слово из гистограммы, самый простой алгоритм — создать список с несколькими копиями каждого слова в соответствии с его частотностью, а затем выбрать из этого списка:

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

Выражение `[word] * freq` создает список с копиями слов, где число копий соответствует значению `freq`. Метод `extends()` похож на `append()`, за исключением того, что аргумент здесь представлен последовательностью.

Этот алгоритм приемлем, но не очень эффективен; каждый раз, когда вы выбираете случайное слово, он перестраивает список, равный размеру оригинальной книги. Самый очевидный способ улучшить алгоритм — создать список один раз, а затем сделать несколько выборок, но список все равно будет очень большой.

Есть такая альтернатива.

1. Используйте `keys`, чтобы получить список слов в книге.
2. Составьте список, который содержит совокупную сумму частотности слов (см. упражнение 10.2). Последний элемент в этом списке — общее количество слов в книге, *n*.
3. Выберите случайное число в диапазоне от 1 до *n*. Используйте дихотомический поиск (см. упражнение 10.10), чтобы найти индекс, где случайное число находится в накопленной сумме.
4. Используйте индекс, чтобы найти соответствующее слово в списке слов.

Упражнение 13.7
Напишите программу, которая использует этот алгоритм для выбора случайного слова из книги.

Решение: **thinkpython2.com/code/analyze_book3.py**.

ЦЕПИ МАРКОВА*

Если выбрать слова из книги случайным образом, то можно получить нечто вроде словаря, но вряд ли — предложение:

```
this the small regard harriet which knightley's it most things
```

Ряд случайных слов редко имеет смысл, потому что между словами нет никакой связи. Например, в реальном предложении вы ожидаете, что за артиклем the следует прилагательное или существительное, а не глагол или наречие.

Один из способов измерять такого рода отношения — анализ цепи Маркова, который для заданной последовательности слов оценивает вероятность появления определенного следующего слова. Например, песня Eric, the Half a Bee начинается так:

Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D'you see?

But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee
Due to some ancient injury?

В этом тексте после фразы half the всегда следует слово bee, но после фразы bee может следовать слово has или is.

Результат анализа цепи Маркова — сопоставление каждого префикса (например, half the и bee) со всеми возможными суффиксами (например, has и is).

* Цепь Маркова — последовательность случайных событий, в которой каждое последующее событие зависит от предыдущего. *Прим. ред.*

Учитывая это сопоставление, можно генерировать произвольный текст, начиная с любого префикса и выбирая случайным образом из возможных суффиксов. А потом объедините конец префикса и новый суффикс, чтобы сформировать следующий префикс, и повторите.

Например, если вы начинаете с префикса Half a, то следующим словом должно быть bee, потому что этот префикс появляется в тексте только один раз. Следующий префикс — a bee, поэтому следующий суффикс может быть philosophically, be или due.

В этом примере длина префикса всегда равна двум, но вы можете выполнить анализ цепи Маркова для префикса любой длины.

Упражнение 13.8

Цепь Маркова

1. Напишите программу, которая читает текст из файла и выполняет анализ цепи Маркова. Результатом должен быть словарь, сопоставляющий (маппирующий) префиксы в набор возможных суффиксов. Набор может быть списком, кортежем или словарем — по вашему выбору. Вы можете протестировать вашу программу с префиксом длины 2, но вы должны написать программу так, чтобы было легко попробовать другие длины.
2. Добавьте функцию в предыдущую программу для генерации случайного текста на основе анализа цепи Маркова. Ниже показан пример для романа «Эмма» с длиной префикса 2:

   > "He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.

   В этом примере я оставил пунктуацию в словах. Синтаксически результат почти правильный, но не совсем. Семантически отрывок почти имеет смысл, но не совсем.

   Что произойдет, если увеличить длину префикса? Случайный текст приобретет больше смысла?
3. Когда программа заработает, попробуйте смешать тесты. Если скомбинировать две или несколько книг, то результатом будет случайный текст с интересной смесью словарей и фраз из этих источников.

Примечание: этот пример взят из книги Кернигана и Пайка «Практика программирования»*.

Вы должны попробовать выполнить это упражнение, прежде чем продолжить; потом вы можете скачать мое решение с **thinkpython2.com/code/markov.py**. Вам также понадобится файл **thinkpython2.com/code/emma.txt**.

СТРУКТУРЫ ДАННЫХ

Генерировать случайные тексты с помощью анализа цепи Маркова — забавно, но в этом упражнении есть и полезный смысл: выбор структуры данных. В своем решении предыдущих упражнений вы должны были выбрать:

— как представлять префиксы;
— как представить коллекцию возможных суффиксов;
— как представить сопоставление каждого префикса с набором возможных суффиксов.

Последнее легко: словарь — очевидный выбор для сопоставления ключей с соответствующими значениями.

Для префиксов наиболее очевидными вариантами могут быть строка, список строк или кортеж строк.

Для суффиксов один из вариантов — список; другой — гистограмма (словарь).

Что же выбрать? Первый шаг — подумать об операциях, которые нужно выполнить для каждой структуры данных. Для префиксов нам нужно удалять слова из начала и добавлять в конец. Например, если текущий префикс Half a, а следующее слово bee, вы должны быть в состоянии сформировать следующий префикс a bee.

Сначала может показаться, что список — идеальный выбор, поскольку элементы легко добавлять и удалять, но нам также необходима возможность использовать префиксы в качестве ключей в словаре, то есть списки не подойдут. С кортежами нельзя добавлять или удалять, но можно использовать оператор сложения для формирования нового кортежа:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

* Керниган Брайан, Пайк Роб. Практика программирования. М.: Вильямс, 2019.

Функция `shift()` принимает кортеж слов, префикс и строку `word` и формирует новый кортеж, в котором содержатся все слова из префикса, кроме первого, а `word` добавляются в конец.

Для набора суффиксов необходимо выполнить следующие операции: добавить новый суффикс (или увеличить частотность существующего) и выбрать случайный суффикс.

Добавить новый суффикс одинаково легко с помощью списка или гистограммы. Выбрать случайный элемент из списка нетрудно; труднее сделать выбор из гистограммы (см. упражнение 13.7).

До сих пор мы говорили в основном о простоте реализации, но есть и другие факторы, которые следует учитывать при выборе структур данных. Один из них — время выполнения. Иногда есть теоретическое обоснование, почему одна структура данных будет быстрее, чем другая. К примеру, я упомянул, что оператор `in` быстрее работает для словарей, чем для списков, по крайней мере в тех случаях, когда количество элементов велико.

Но не всегда заранее известно, какая реализация будет быстрее. Один из вариантов — реализовать обе и посмотреть, что лучше. Такой подход называется **сравнительным анализом (benchmarking)**. Практическая альтернатива: выбрать структуру данных, которую проще всего реализовать, а затем посмотреть, достаточно ли она быстра для предполагаемого приложения. Если да, то нет необходимости что-то менять. Если же скорости не хватает, есть инструменты, такие как модуль `profile`, которые определяют самые «медлительные» места в программе.

Другой фактор, который следует учитывать, — используемая память. Так, применение гистограммы для набора суффиксов может занять меньшее количество, поскольку вам нужно хранить каждое слово только один раз, независимо от того, сколько раз оно встречается в тексте. Иногда экономия памяти ускоряет выполнение программы, и в самом плохом случае программа может вообще не работать, если на устройстве не хватает памяти. Но для многих приложений объем памяти — второстепенный фактор после времени выполнения.

В заключение: в этом рассуждении я подразумевал, что мы должны использовать одну структуру данных и для анализа, и для генерации. Но поскольку это отдельные фазы, можно также использовать одну структуру для анализа, а затем преобразовать ее в другую уже для генерации. Это будет чистый выигрыш, если время, сэкономленное на генерации, превысит время, потраченное на конверсию.

ОТЛАДКА

Когда вы отлаживаете программу, особенно если исправляете серьезную ошибку, на помощь придут следующие пять пунктов.

Чтение:

Изучите свой код, прочитайте его снова и убедитесь, что он делает именно то, что вы хотели.

Выполнение:

Экспериментируйте, внося изменения и запуская разные версии программы. Часто, если вы выводите диагностическую информацию в нужном месте программы, проблема становится очевидной, а иногда приходится писать отладочный код.

Размышление:

Потратьте время, чтобы подумать! Что это за ошибка: синтаксическая, в процессе выполнения или семантическая? Какую информацию вы можете получить из сообщений об ошибках или из выходных данных программы? Какая ошибка может вызвать проблему, которую вы видите? Что вы меняли в последний раз, до появления проблемы?

Метод утенка:

Если вы объясняете проблему кому-то еще, вы иногда находите ответ, прежде чем закончите задавать вопрос. Часто вам не нужен собеседник, достаточно поговорить с резиновой уточкой. Такой прием лег в основу известной стратегии **«метод утенка» (rubber duck debugging)**: **ru.wikipedia.org/wiki/Метод_утёнка**.

Откат:

В какой-то момент лучше всего отступить назад и отменить последние изменения, пока вы не вернетесь к работоспособному варианту программы, который понимаете. И тогда можно продолжить программировать.

Начинающие программисты иногда зацикливаются на одном из этих действий и забывают о других. Но у каждого свои недостатки.

Например, чтение кода может помочь, если проблема связана с опечаткой, но не с ошибкой в идее реализации. Если вы не понимаете, что делает ваша программа, вы можете прочитать ее сто раз и не увидеть ошибку, потому что ошибка у вас в голове.

Эксперименты могут помочь, особенно если вы запускаете небольшие простые тесты. Но если вы проводите эксперименты, не думая и не читая

свой код, вы рискуете попасть в ловушку, которую я называю «программирование случайным блужданием». Вас затянет процесс внесения случайных изменений до тех пор, пока программа не начнет выдавать нужный результат. Излишне говорить, что программирование случайным блужданием отнимает много времени.

Пора остановиться и подумать. Отладка похожа на экспериментальную науку. У вас должна быть хотя бы одна гипотеза о том, в чем проблема. Если есть две или более, попробуйте придумать тест, который бы исключил одну из них.

Но даже лучшие методы отладки потерпят неудачу, если будет слишком много ошибок или если код, который вы пытаетесь исправить, чересчур велик и сложен. Иногда лучший вариант — отступить, упростив программу, пока вы не вернетесь к тому, что работает и что вы понимаете.

Начинающие программисты не очень любят отступать, потому что не могут удалить строку кода (даже если она неправильная). Если вам от этого легче, скопируйте вашу программу в другой файл, прежде чем удалять строки. Затем вы можете копировать кусочки обратно по одному.

Чтобы найти серьезную ошибку, нужно читать, выполнять, размышлять, а иногда и отступать. Если вам не помог один способ, попробуйте другие.

СЛОВАРЬ ТЕРМИНОВ

Детерминированный:
: Относится к программе, которая делает то же самое каждый раз, когда запускается, с теми же входными данными.

Псевдослучайный:
: Относится к последовательности чисел, которая выглядит случайной, но генерируется детерминированной программой.

Значение по умолчанию:
: Значение, присваиваемое необязательному параметру, если аргумент не задан.

Переопределить:
: Заменить значение по умолчанию на аргумент.

Сравнительный анализ:
: Процесс выбора между структурами данных путем реализации альтернатив и тестирования их на выборке возможных входных данных.

Отладка методом утенка:

Отладка путем объяснения вашей проблемы неодушевленному предмету, например резиновой уточке. Постановка проблемы может помочь вам решить ее, даже если резиновая уточка не знает Python.

УПРАЖНЕНИЯ

Упражнение 13.9

«Ранг» слова — это его позиция в списке слов, отсортированных по частотности: наиболее распространенное слово имеет ранг 1, второе по частотности — ранг 2 и так далее.

Закон Ципфа описывает связь между рангами и частотностями слов в естественных языках (**ru.wikipedia.org/wiki/Закон_Ципфа**). В частности, он гласит, что частотность f слова с рангом r равна:

$$f = cr^{-s},$$

где s и c — параметры, которые зависят от языка и текста. Если вы возьмете логарифм обеих сторон этого уравнения, вы получите:

$$\log f = \log c - s\log r.$$

Поэтому, если вы построите $\log f$ по отношению к $\log r$, вы должны получить прямую линию с наклоном $-s$ и свободным членом $\log c$.

Напишите программу, которая считывает текст из файла, считает частотности слов и печатает каждое слово в отдельной строке в порядке убывания частоты с помощью $\log f$ и $\log r$. Используйте любую графическую программу на ваш выбор, чтобы наглядно представить результаты и проверить, образуют ли они прямую линию. Можете ли вы оценить значение s?

Решение: **thinkpython2.com/code/zipf.py**. Для запуска моего решения вам понадобится модуль для построения графиков `matplotlib`. Если вы установили дистрибутив Anaconda, у вас уже есть библиотека `matplot`; в противном случае вам придется установить ее.

ГЛАВА 14
ФАЙЛЫ

В этой главе представлена идея «персистентных» программ, которые могут хранить данные в постоянном хранилище, и показано, как использовать разные постоянные хранилища, такие как файлы и базы данных.

УСТОЙЧИВОСТЬ (ПЕРСИСТЕНТНОСТЬ)

Большинство программ, с которыми мы встречались до сих пор, временны в том смысле, что они работают в течение короткого времени и дают некоторый результат, но, когда они заканчивают выполнение, их данные исчезают. Если вы запустите программу снова, она начнет с нуля.

Другие программы **персистентны (persistent)**: они работают в течение длительного времени (или постоянно); хранят хотя бы часть своих данных в постоянном хранилище (например, на жестком диске); и если их закрыть или перезапустить, они начнут выполнение с того места, где остановились.

Примерами постоянных программ могут служить операционные системы, которые работают все время, пока компьютер включен, и веб-серверы, которые постоянно ожидают запросов в сети.

Один из самых простых способов сохранять данные в программах — это чтение и запись текстовых файлов. Мы уже видели программы, которые читают текстовые файлы. В этой главе мы увидим программы, которые их пишут.

Альтернатива — сохранение состояния программы в базе данных. В этой главе я представлю простую базу данных и модуль `pickle`, который упрощает хранение данных программы.

ЧТЕНИЕ И ЗАПИСЬ

Текстовый файл — это последовательность символов, хранящаяся на постоянном носителе, таком как жесткий диск, flash-накопитель или оптический диск. Вы уже видели, как открыть и прочитать файл в разделе «Чтение списка слов» главы 9.

Чтобы изменить файл, вы должны открыть его в режиме 'w', который передается в качестве второго параметра:

```
>>> fout = open('output.txt', 'w')
```

Если файл уже существует, его открытие в режиме записи удаляет старые данные, поэтому будьте осторожны! Если файл не существует, создается новый.

Функция `open()` возвращает объект файла, который предоставляет методы для работы с файлом. Метод `write()` сохраняет данные в файл:

```
>>> line1 = "Я ждал это время, и вот это время пришло,\n"
>>> fout.write(line1)
42
```

Возвращаемое значение — это количество символов, которые были записаны в файл. Файловый объект отслеживает, где он находится, поэтому, если вы снова вызовете метод `write()`, он добавляет новые данные в конец файла:

```
>>> line2 = “Те, кто молчал, перестали молчать.\n”
>>> fout.write(line2)
35
```

Когда вы закончите запись, вы должны закрыть файл:

```
>>> fout.close()
```

Если вы не закроете файл, он закроется автоматически при завершении программы.

ОПЕРАТОР ФОРМАТИРОВАНИЯ

Аргумент функции `write()` должен быть строкой, поэтому, если мы хотим поместить другие значения в файл, нужно преобразовать их в строки. Самый простой способ сделать это — с помощью функции `str()`:

```
>>> x = 52
>>> fout.write(str(x))
```

Альтернатива — использование **оператора форматирования (format operator)**, `%`. Если мы работаем с целыми числами, то `%` — оператор деления по модулю. Но когда первый операнд — строка, символ `%` представляет собой оператор форматирования.

Первый операнд — это **форматирующая строка (format sequences)**, которая содержит одну или несколько последовательностей форматирования, которые определяют, как будет отформатирован второй операнд. Результатом будет строка.

Например, последовательность `'%d'` означает, что второй операнд должен быть отформатирован как десятичное целое число:

```
>>> camels = 45
>>> '%d' % camels
'45'
```

В результате получается строка `'45'`, которую не следует путать с целочисленным значением `45`.

Форматирующая последовательность может указываться в любом месте строки, поэтому вы можете вставить значение в предложение:

```
>>> 'У меня есть %d верблюдов.' % camels
'У меня есть 45 верблюдов.'
```

Если в строке указано более одной последовательности форматирования, второй аргумент должен быть кортежем. Каждая последовательность форматирования соответствует элементу кортежа по порядку.

В следующем примере последовательность `'%d'` используется для форматирования целого числа, `'%g'` — для форматирования числа с плавающей точкой, а `'%s'` — для форматирования строки:

```
>>> 'За %g года я купил %d %s.' % (3.5, 7, 'верблюдов')
'За 3.5 года я купил 7 верблюдов.'
```

Количество элементов в кортеже должно соответствовать количеству последовательностей форматирования в строке. Кроме того, типы элементов должны соответствовать этим последовательностям:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'рублей'
TypeError: %d format: a number is required, not str
```

В первом примере недостаточно элементов; во втором у элемента неправильный тип.

Для получения дополнительной информации об операторе форматирования см. **docs.python.org/3/library/stdtypes.html#printf-style-string-formatting**. Более мощная альтернатива — метод `str.format()`, о котором вы можете прочитать по адресу **docs.python.org/3/library/stdtypes.html#str.format**.

ИМЕНА ФАЙЛОВ И ПУТИ

Файлы организованы по **каталогам (directories)** (их еще называют папками или директориями). Каждая запущенная программа имеет «текущий каталог», который используется по умолчанию для большинства операций. Например, когда вы открываете файл для чтения, Python ищет его в текущем каталоге.

Модуль `os` предоставляет функции для работы с файлами и каталогами (`os` означает `operating system` — операционная система). Функция `os.getcwd()` возвращает имя текущего каталога:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

Переменная `cwd` означает current working directory — «текущий рабочий каталог». Результат функции — `/home/dinsdale` — домашний каталог пользователя `dinsdale`.

Строка типа `'/home/dinsdale'`, которая идентифицирует файл или каталог, называется **путем (path)**.

Простое имя файла, например *memo.txt*, также считается путем, но это **относительный путь (relative path)**, поскольку он относителен текущего каталога. Если текущий каталог — */home/dinsdale*, имя файла *memo.txt* будет ссылаться на расположение */home/dinsdale/memo.txt*.

Путь, который начинается с /, не зависит от текущего каталога; он называется **абсолютным путем (absolute path)**. Чтобы выяснить абсолютный путь к файлу, можно использовать функцию `os.path.abspath()`:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

Также модуль `os.path` предоставляет другие функции для работы с файлами и путями. Функция `os.path.exists()` проверяет, существует ли файл или каталог:

```
>>> os.path.exists('memo.txt')
True
```

Если он существует, функция `os.path.isdir()` проверяет, каталог ли это:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

Аналогично, функция `os.path.isfile()` проверяет, является ли объект файлом.

Функция `os.listdir()` возвращает список файлов (и других каталогов) в текущем каталоге:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

Чтобы продемонстрировать эти функции на практике, я приведу пример: показанная ниже программа «проходит» по каталогу, печатает имена всех файлов и рекурсивно вызывает себя во всех каталогах:

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

Функция `os.path.join()` берет имена каталога и файла и объединяет их в полный путь.

Модуль `os` предоставляет похожую функцию `walk()`, которая более универсальна. В качестве упражнения прочитайте документацию и используйте ее для вывода имен файлов в текущем каталоге и его подкаталогах. Вы можете скачать мое решение по адресу **thinkpython2.com/code/walk.py**.

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Во время работы с файлами часто что-то идет не так. Если вы попытаетесь открыть несуществующий файл, появится ошибка `IOError`:

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

Если у вас нет разрешения на доступ к файлу:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

И если вы попытаетесь открыть каталог для чтения, то получите следующую ошибку:

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

Чтобы избежать этих ошибок, можно использовать такие функции, как `os.path.exists()` и `os.path.isfile()`, но проверка всех вариантов потребовала бы много времени и кода (значение `Errno 21` указывает, что может возникнуть не менее 21 причины ошибки).

Лучше двигаться вперед и работать над проблемами, если они случаются, — именно это и делает инструкция `try`. Синтаксис похож на инструкцию `if ... else`:

```
try:
    fin = open('bad_file')
except:
    print('Что-то пошло не так.')
```

Python начинает с выполнения ветви `try`. Если все идет хорошо, он пропускает блок `except` и продолжает. Но если возникает исключение, Python выходит из блока `try` и выполняет блок `except`.

Обработка исключения с помощью оператора `try` называется **перехватом (catch)** исключения. В этом примере блок `except` печатает сообщение об ошибке, что не очень полезно. В общем случае перехват исключения дает шанс решить проблему, попытаться еще раз или, по крайней мере, завершить программу аккуратно.

БАЗЫ ДАННЫХ

База данных — это файл для организованного хранения данных. Многие базы данных организованы как словари в том смысле, что они сопоставляют ключи и значения. Самое большое различие между базой данных и словарем в том, что база данных находится на диске (или другом постоянном хранилище), поэтому она сохраняется после завершения программы.

Модуль `dbm` предоставляет интерфейс для создания и обновления файлов баз данных. В качестве примера я создам базу данных, которая содержит подписи для изображений.

Открытие базы данных аналогично открытию любых других файлов:

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

Режим `'c'` означает, что база данных должна быть создана, если она еще не существует.

Результатом будет объект базы данных, который можно использовать (для большинства операций) как словарь.

Когда вы создаете новый элемент, модуль dbm обновляет файл базы данных:

```
>>> db['cleese.png'] = 'Фотография Ивана Клизина.'
```

Когда вы запрашиваете доступ к одному из элементов, модуль dbm читает файл:

```
>>> db['cleese.png']
b'Фотография Ивана Клизина.'
```

Результатом будет **байтовый объект (bytes object)**, поэтому он начинается с b. Он во многом похож на строку. Когда вы изучите Python глубже, разница станет важной, но пока ее можно игнорировать.

Если вы присваиваете существующему ключу новое значение, модуль dbm заменит старое значение:

```
>>> db['cleese.png'] = 'Фотография Ивана Клизина на прогулке.'
>>> db['cleese.png']
b'Фотография Ивана Клизина на прогулке.'
```

Некоторые словарные методы, такие как keys() или items(), не работают с объектами баз данных. Но итерация с помощью цикла for допустима:

```
for key in db:
    print(key, db[key])
```

Как и в случае с другими файлами, вы должны закрыть базу данных, когда закончите:

```
>>> db.close()
```

СЕРИАЛИЗАЦИЯ

Ограничение модуля dbm в том, что ключи и значения должны быть строками или байтами. Если вы попытаетесь использовать любой другой тип, вы получите ошибку.

Тут приходит на помощь модуль pickle. Он **сериализует** — переводит практически любой тип объекта в строку, пригодную для хранения в базе данных, а также **десериализует** — переводит строки обратно в объекты.

Функция pickle.dumps() принимает объект в качестве параметра и возвращает строковое представление:

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

Такой формат непонятен человеку, но модуль `pickle` может легко его интерпретировать. Функция `pickle.loads()` восстанавливает объект:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

Хотя новый объект имеет то же значение, что и старый, он не является (вообще) тем же объектом:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

Другими словами, сериализация, а затем десериализация имеет тот же эффект, что и копирование объекта.

Вы можете использовать модуль `pickle` для хранения не строковых данных в базах данных. Фактически эта комбинация настолько распространена, что она была вынесена в отдельный модуль `shelve`.

КОНВЕЙЕР

Большинство операционных систем предоставляют интерфейс командной строки, также известный как **оболочка (shell)**. Оболочки обычно предоставляют команды для навигации по файловой системе и запуска приложений. Например, в системе Unix вы можете перемещаться по каталогам с помощью команды `cd`, отображать содержимое каталогов с помощью команды `ls` и запускать веб-браузеры, набрав, к примеру, `firefox`.

Любую программу, которую можно запустить из оболочки, можно запустить из Python с использованием **объекта конвейера (pipe object)**, который представляет собой работающую программу.

Например, Unix-команда `ls -l` обычно отображает содержимое текущего каталога в длинном формате. Вы можете запустить команду `ls` с помощью метода `os.popen()`*:

* Сегодня функция popen() считается устаревшей, то есть мы должны прекратить ее использование и начать использовать модуль subprocess. Но для простых случаев я нахожу модуль subprocess сложнее, чем нужно. Так что я продолжу использовать функцию popen(), пока она не будет удалена. *Прим. авт.*

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

Аргумент представляет собой строку, содержащую команду оболочки. Возвращаемое значение — это объект, который ведет себя как открытый файл. Вы можете прочитать вывод команды `ls` построчно с помощью `readline` или получить всё сразу с помощью `read`:

```
>>> res = fp.read()
```

Когда вы закончите работу, вы закрываете конвейер как файл:

```
>>> stat = fp.close()
>>> print(stat)
None
```

Возвращаемое значение является окончательным статусом процесса `ls`; `None` означает, что он закончился нормально (без ошибок).

Большинство Unix-подобных систем предоставляют команду `md5sum`, которая считывает содержимое файла и вычисляет «контрольную сумму». Вы можете прочитать об алгоритме MD5 по адресу **ru.wikipedia.org/wiki/MD5**. Эта команда предоставляет эффективный способ проверить, имеют ли два файла одинаковое содержимое. Вероятность того, что разное содержимое позволит вычислить одну и ту же контрольную сумму, крайне мала (то есть это вряд ли произойдет за время существования Вселенной).

Вы можете использовать конвейер для запуска команды `md5sum` из Python и получить результат:

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0 book.tex
>>> print(stat)
None
```

СОЗДАНИЕ СОБСТВЕННЫХ МОДУЛЕЙ

Любой файл, содержащий код Python, может быть импортирован как модуль. Предположим, у вас есть файл *wc.py* со следующим кодом:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

Если вы запустите эту программу, она сама прочитает и напечатает количество строк в файле, которое равно 7. Вы также можете импортировать файл следующим способом:

```
>>> import wc
7
```

Теперь у вас есть объект модуля `wc`:

```
>>> wc
<module 'wc' from 'wc.py'>
```

Объект модуля предоставляет функцию `linecount()`:

```
>>> wc.linecount('wc.py')
7
```

Таким образом создаются модули на Python.

Единственная проблема в этом примере в том, что при импорте модуля выполняется последующий тестовый код. Хотелось бы, чтобы при импорте модуля новые функции определялись, но не запускались.

Программы, которые будут импортированы как модули, часто используют следующую конструкцию:

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

где `__name__` — это встроенная переменная, которая настраивается при запуске программы. Если программа работает как скрипт, `__name__` имеет значение `'__main__'`; в этом случае тестовый код выполняется. В противном случае, если модуль импортируется, тестовый код пропускается.

В качестве упражнения добавьте эти строки в файл *wc.py* и запустите его как скрипт. Затем запустите интерпретатор Python и импортируйте модуль `wc`. Какое значение получила переменная `__name__` при импорте модуля?

Предупреждение: если вы повторно импортируете модуль, который уже был импортирован, Python пропустит инструкцию импорта. Он не загружает заново файл, даже если тот изменился.

Чтобы перезагрузить модуль, попробуйте воспользоваться встроенной функцией `reload()`. Правда, с ней бывает сложно управиться, поэтому самое простое, что можно сделать, — это перезапустить интерпретатор и затем снова импортировать модуль.

ОТЛАДКА

Когда вы читаете и записываете файлы, у вас могут появиться проблемы с непечатаемыми символами. Эти ошибки бывает трудно отловить, потому что символы пробелов, отступов и перевода строки обычно невидимы:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2 3
4
```

Здесь поможет встроенная функция `repr()`. Она принимает любой объект в качестве аргумента и возвращает строковое представление объекта. В строковом представлении непечатаемые символы представлены в виде последовательности с обратной косой чертой:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

Это способ пригодится в процессе отладки.

Другая вероятная проблема заключается в том, что разные системы используют разные символы для обозначения конца строки: в одних это символ новой строки, `\n`, а в других — символ возврата каретки, `\r`. А часть систем вообще использует оба варианта. Если вы перемещаете файлы между различными системами, несоответствия могут вызвать проблемы.

У большинства систем существуют приложения для преобразования из одного формата в другой. Найти их и прочитать больше об этой проблеме можно тут: **ru.wikipedia.org/wiki/Перевод_строки**. Разумеется, всегда можно попробовать самому написать такое приложение.

СЛОВАРЬ ТЕРМИНОВ

Устойчивость (персистентность):

Относится к программе, которая работает неопределенно долго и хранит хотя бы часть своих данных в постоянном хранилище.

Оператор форматирования:

Оператор `%`, который принимает строку формата и кортеж и генерирует строку, которая включает элементы кортежа, отформатированные в соответствии со строкой форматирования.

Форматирующая строка:

Строка, используемая с оператором форматирования, которая содержит последовательности форматирования.

Последовательность форматирования:

Последовательность символов в строке формата, например `%d`, которая указывает, как следует форматировать значение.

Текстовый файл:

Последовательность символов, находящаяся в постоянном хранилище, например на жестком диске.

Каталог:

Именованная коллекция файлов, также называемая папкой.

Путь:

Строка, которая идентифицирует файл.

Относительный путь:

Путь, который начинается с текущего каталога.

Абсолютный путь:

Путь, который начинается с самого верхнего (корневого) каталога в файловой системе.

Перехват исключения:

Для обработки исключения программы и ее правильного завершения используются инструкции `try` и `except`.

База данных:

Файл, содержимое которого организовано как словарь с ключами и соответствующими значениями.

Байтовый объект:

Объект, похожий на строку.

Оболочка:

Программа, которая позволяет пользователям вводить команды, а затем выполнять их, запуская другие программы.

Объект конвейера:

Объект, представляющий работающую программу, позволяющую ей запускать команды и получать результаты их выполнения.

УПРАЖНЕНИЯ

Упражнение 14.1

Напишите функцию `sed()`, которая принимает в качестве аргументов строку шаблона, строку замены и два имени файлов; программа должна считать содержимое первого файла и записать его во второй файл (создавая его при необходимости). Если строка шаблона встречается где-либо в файле, ее следует заменить строкой замены.

Если при открытии, чтении, записи или закрытии файлов возникает ошибка, программа должна перехватить исключение, напечатать сообщение об ошибке и завершить работу.

Решение: **thinkpython2.com/code/sed.py**.

Упражнение 14.2

Если вы скачаете мое решение для упражнения 12.2 по адресу **thinkpython2.com/code/anagram_sets.py**, то увидите, что моя программа создает словарь, который сопоставляет отсортированную строку букв со списком слов, которые могут быть составлены из этих букв. Например, `'opst'` сопоставляется со списком `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Напишите модуль, который импортирует модуль `anagram_sets` и предоставляет две новые функции: `store_anagrams()` должна сохранять словарь анаграмм на «полке», `read_anagrams()` должна находить слово и возвращать список его анаграмм.

Решение: **thinkpython2.com/code/anagram_db.py**.

Упражнение 14.3

В большой коллекции MP3-файлов может храниться несколько копий одной и той же песни в разных каталогах или с разными именами файлов. Цель этого упражнения — найти дубликаты.

1. Напишите программу, которая рекурсивно просматривает каталог и все его подкаталоги и возвращает список полных путей для всех файлов с заданным суффиксом (например, `.mp3`). Подсказка: модуль

`os.path` предоставляет несколько полезных функций для управления файлами и путями.

2. Чтобы распознать дубликаты, вы можете использовать команду `md5sum` для вычисления «контрольной суммы» каждого файла. Если два файла имеют одинаковую контрольную сумму, вероятно, имеют одинаковое содержимое.
3. Чтобы перепроверить себя, вы можете использовать Unix-команду `diff`.

Решение: **thinkpython2.com/code/find_duplicates.py**.

ГЛАВА 15
КЛАССЫ И ОБЪЕКТЫ

На данный момент вы знаете, как использовать функции для организации кода и встроенные типы для организации данных. Следующий шаг — изучение «объектно-ориентированного программирования», в котором используют пользовательские типы, определенные программистом для организации и кода, и данных. Объектно-ориентированное программирование — емкая тема; нам понадобится несколько глав, чтобы в ней разобраться.

Примеры кода из этой главы доступны по адресу **thinkpython2.com/code/Point1.py**, а решения упражнений — в файле **thinkpython2.com/code/Point1_soln.py**.

ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ

Мы использовали многие из встроенных типов Python, а сейчас определим новый тип. В качестве примера создадим тип с именем `Point`, который представляет точку в двумерном пространстве.

В математических обозначениях точки координат часто пишутся в скобках и разделяются запятой. Например, (0,0) представляет точку начала координат, а (*x*, *y*) представляет точку на *x* единиц правее и на *y* единиц выше от точки начала координат.

Есть несколько способов представить точки в Python:

— мы могли бы хранить координаты отдельно в двух переменных, `x` и `y`,
— мы могли бы сохранить координаты как элементы в списке или кортеже,
— мы могли бы создать новый тип для представления точек как объектов.

Создать новый тип сложнее, чем другие варианты, но у этого способа есть преимущества, которые скоро станут очевидны.

Заданный программистом тип называется **классом (class)**. Определение класса выглядит так:

```
class Point:
    """Представление точки в двумерном пространстве."""
```

Заголовок указывает, что новый класс называется `Point`. Тело — это строка документации, объясняющая, для чего предназначен класс. Вы можете определить переменные и методы внутри определения класса, но об этом поговорим позже.

Определение класса `Point` создает **объект класса (class object)**:

```
>>> Point
<class '__main__.Point'>
```

Поскольку класс `Point` определен на верхнем уровне, его «полное имя» — `__main__.Point`.

Объект класса похож на фабрику для создания объектов. Чтобы создать объект `Point`, вы можете вызвать `Point`, как если бы это была функция:

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

Возвращаемое значение является ссылкой на объект `Point`, который мы присваиваем переменной `blank`.

Создание нового объекта называется **созданием экземпляра (instantiation)**, а сам объект — это **экземпляр (instance)** класса.

Когда вы выводите экземпляр, Python сообщает вам, к какому классу он принадлежит и где он хранится в памяти (префикс `0x` означает, что следующее число шестнадцатеричное).

Каждый объект — это экземпляр некоторого класса, поэтому понятия «объект» и «экземпляр» взаимозаменяемы. Но в этой главе я использую термин «экземпляр», чтобы обозначить пользовательские типы — то есть созданные разработчиком программы.

АТРИБУТЫ

Вы можете присвоить значения экземпляру, используя точечную нотацию:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

Похоже на синтаксис для выбора переменной из модуля, например `math.pi` или `string.whitespace`. Однако в этом случае мы присваиваем значения именованным элементам объекта. Эти элементы называются **атрибутами (attributes)**.

Следующая диаграмма отражает результат этих присваиваний. Диаграмма состояния, которая показывает объект и его атрибуты, называется **диаграммой объекта (object diagram)**; см. рис. 15.1.

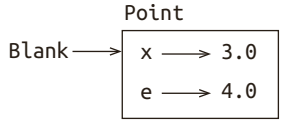


Рис. 15.1. Диаграмма объекта

Переменная `blank` ссылается на объект `Point`, который содержит два атрибута. Каждый атрибут ссылается на число с плавающей точкой.

Вы можете прочитать значение атрибута, используя тот же синтаксис:

```
>>> blank.y
4.0
>>> x = blank.x
>>> x 3.0
```

Выражение `blank.x` означает: «Перейти к объекту `blank` и извлечь значение атрибута `x`». В нашем примере мы присваиваем это значение переменной `x`. Между переменной `x` и атрибутом `x` конфликта не возникает.

Можно использовать точечную нотацию как часть другого выражения. Например:

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

Можно передать экземпляр в качестве аргумента обычным способом. Например:

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

Функция `print_point()` принимает точку координат в качестве аргумента и отображает ее в математической форме.

Чтобы вызвать ее, вы можете передать `blank` в качестве аргумента:

```
>>> print_point(blank)
(3.0, 4.0)
```

Внутри функции `p` — это синоним объекта `blank`, поэтому, если функция изменяет `p`, изменяется и `blank`.

В качестве упражнения напишите функцию `distance_between_points()`, которая принимает два объекта `Point` в качестве аргументов и возвращает расстояние между ними.

ПРЯМОУГОЛЬНИКИ

Иногда сразу очевидно, какими должны быть атрибуты объекта, а иногда приходится подумать. Например, представьте, что вы разрабатываете класс для представления прямоугольников. Какие атрибуты вы бы использовали, чтобы указать расположение и размер прямоугольника? Вы можете игнорировать наклон — в целях упрощения задачи. Предположим, что прямоугольник ориентирован строго по вертикали или горизонтали.

Доступны как минимум два варианта:

— указать расположение одного угла прямоугольника (или центра), ширину и высоту;
— указать расположение двух противоположных углов.

Трудно сказать, какой способ лучше, поэтому мы реализуем первый, просто в качестве примера.

Ниже представлено определение класса:

```
class Rectangle:
    """Определяет прямоугольник.
    атрибуты: width, height, corner.
    """
```

Строка документации перечисляет атрибуты: ширина (`width`) и высота (`height`) — это числа; угол (`corner`) — это объект `Point`, который определяет левый нижний угол.

Чтобы представить прямоугольник, вы должны создать экземпляр объекта `Rectangle` и присвоить значения атрибутам:

```
box = Rectangle()
box.width = 100.0
```

```
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

Выражение `box.corner.x` означает: «Перейдите к объекту, на который ссылается переменная `box`, и выберите атрибут `corner`, затем перейдите к этому объекту и выберите атрибут `x`».

На рис. 15.2 показана диаграмма этого объекта. Объект, который является атрибутом другого объекта, называется **вложенным (embedded)**.

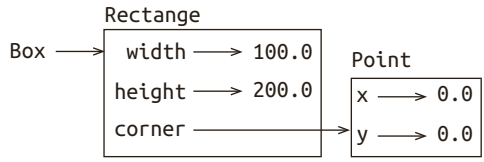


Рис. 15.2. Диаграмма объекта

ВОЗВРАЩЕНИЕ ЭКЗЕМПЛЯРОВ

Функции могут возвращать экземпляры. Например, функция `find_center()` принимает `Rectangle` в качестве аргумента и возвращает точку, которая содержит координаты центра прямоугольника:

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

Ниже показан пример, который передает `box` в качестве аргумента и присваивает результирующую точку переменной `center`:

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

ОБЪЕКТЫ ИЗМЕНЯЕМЫ

Состояние объекта можно изменять, присвоив новое значение одному из его атрибутов. Например, чтобы изменить размер прямоугольника без изменения его положения, вы можете изменить значения переменных `width` и `height`:

```
box.width = box.width + 50
box.height = box.height + 100
```

Вы также можете написать функции, которые изменяют объекты. Например, функция `grow_rectangle()` принимает объект `Rectangle` и два числа, `dwidth` и `dheight`, и прибавляет эти числа к ширине и высоте прямоугольника:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Вот что получается в результате:

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

В функции `rect` — это псевдоним для `box`, поэтому, когда функция изменяет прямоугольник, `box` тоже меняется.

В качестве упражнения напишите функцию `move_rectangle()`, которая принимает объект `Rectangle` и два числа с именами `dx` и `dy`. Следует изменить местоположение прямоугольника, добавив `dx` к координате `x` угла и `dy` к координате `y` угла.

КОПИРОВАНИЕ

Псевдонимы затрудняют понимание программы, поскольку изменения в одном месте способны вызывать неожиданные последствия в другом. Трудно отслеживать все переменные, которые могут ссылаться на данный объект.

Копирование объекта — частая альтернатива псевдонимам. Модуль `copy` предоставляет функцию `copy()`, которая позволяет дублировать любой объект:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

Объекты `p1` и `p2` содержат одинаковые данные, но они не являются одним и тем же объектом `Point`:

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

Оператор `is` указывает, что `p1` и `p2` не являются одним и тем же объектом, чего мы и ожидали. Однако вы могли подумать, что оператор `==` вернет значение `True`, потому что эти точки содержат одинаковые данные. Вы будете разочарованы, узнав, что для экземпляров поведение по умолчанию оператора `==` такое же, как и для оператора `is`; он проверяет идентичность объекта, а не эквивалентность. Такая ситуация возникает потому, что при обработке пользовательских типов интерпретатор Python не знает, что следует считать эквивалентным. По крайней мере, не сейчас.

Если вы используете метод `copy.copy()` для дублирования прямоугольника, вы обнаружите, что он копирует объект `Rectangle`, но не вложенный объект `Point`:

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Рис. 15.3 показывает, как выглядит диаграмма объекта. Эта операция называется **поверхностным копированием (shallow copy)**, потому что она копирует объект и любые содержащиеся в нем ссылки, но не вложенные объекты.

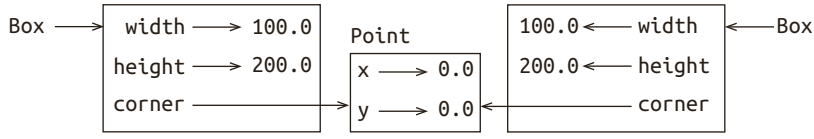


Рис. 15.3. Диаграмма объектов

Для большинства приложений это не тот результат, который вы хотите получить. В нашем примере вызов функции `grow_rectangle()` для одного из прямоугольников не повлияет на другой, а вызов функции `move_rectangle()`

для любого из них повлияет на оба! Такое поведение сбивает с толку и ведет к ошибкам.

К счастью, модуль copy предоставляет метод deepcopy(), который копирует не только объект, но и объекты, на которые он ссылается, любой степени вложенности. Неудивительно, что эта операция называется **глубоким копированием (deep copy)**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

Теперь box3 и box — совершенно разные объекты.

В качестве упражнения напишите версию функции move_rectangle(), которая создает и возвращает новый прямоугольник вместо изменения старого.

ОТЛАДКА

Когда вы начинаете работать с объектами, то сталкиваетесь с некоторыми новыми исключениями. Если вы попытаетесь получить доступ к атрибуту, который не существует, вы увидите ошибку AttributeError:

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

Если вы не уверены, к какому типу принадлежит объект, вы можете уточнить:

```
>>> type(p)
<class '__main__.Point'>
```

Вы также можете использовать функцию isinstance(), чтобы проверить, является ли объект экземпляром класса:

```
>>> isinstance(p, Point)
True
```

Если вы не уверены, имеет ли объект определенный атрибут, вы можете использовать встроенную функцию hasattr():

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

Первый аргумент может быть любым объектом. Второй аргумент — это строка, содержащая имя атрибута.

Еще вы можете использовать инструкцию `try`, чтобы увидеть, есть ли у объекта нужные вам атрибуты:

```
try:
    x = p.x
except AttributeError:
    x = 0
```

Такой подход упростит написание функций с различными типами; подробнее об этом читайте в разделе «Полиморфизм» главы 17.

СЛОВАРЬ ТЕРМИНОВ

Класс:
Тип, определенный программистом. Определение класса создает новый объект класса.

Объект класса:
Объект, который содержит информацию о типе, определенном программистом. Объект класса может использоваться для создания экземпляров типа.

Экземпляр:
Объект, который принадлежит классу.

Создание экземпляра:
Создание нового объекта.

Атрибут:
Одно из именованных значений, связанных с объектом.

Вложенный объект:
Объект, являющийся атрибутом другого объекта.

Поверхностная копия:
Копия содержимого объекта, включая любые ссылки на вложенные объекты; реализуется функцией `copy()` модуля `copy`.

Глубокая копия:

Копия содержимого объекта, а также объектов любой степени вложенности; реализуется функцией `deepcopy()` модуля `copy`.

Диаграмма объекта:

Диаграмма, отображающая объекты, их атрибуты и значения атрибутов.

УПРАЖНЕНИЯ

Упражнение 15.1

Напишите определение класса `Circle` с атрибутами `center` и `radius`, где `center` — это объект `Point`, а `radius` — числовое значение.

Создайте объект `Circle`, представляющий собой круг с центром в точке с координатами (150, 100) и радиусом, равным 75.

Напишите функцию `point_in_circle()`, которая принимает в качестве аргументов объекты `Circle` и `Point` и возвращает `True`, если `Point` лежит внутри круга или на его границе.

Напишите функцию `rect_in_circle()`, которая принимает в качестве аргументов объекты `Circle` и `Rectangle` и возвращает `True`, если `Rectangle` полностью лежит в кругу или на его границе.

Напишите функцию `rect_circle_overlap()`, которая принимает в качестве аргументов объекты `Circle` и `Rectangle` и возвращает `True`, если любой из углов прямоугольника оказывается внутри круга. Или, чтобы усложнить задачу, верните `True`, если какая-либо часть прямоугольника попадает внутрь круга.

Решение: **thinkpython2.com/code/Circle.py**.

Упражнение 15.2

Напишите функцию `draw_rect()`, которая принимает в качестве аргументов объекты `Turtle` и `Rectangle` и использует `Turtle` для рисования `Rectangle`. В главе 4 рассказывается, как использовать объект `Turtle`.

Напишите функцию `draw_circle()`, которая принимает в качестве аргументов объекты `Turtle` и `Circle` и рисует круг.

Решение: **thinkpython2.com/code/draw.py**.

ГЛАВА 16
КЛАССЫ И ФУНКЦИИ

Теперь, когда вы знаете, как создавать новые типы, можно приступить к написанию функций, которые могут принимать их как параметры и возвращать как результат.

В этой главе я также представляю «функциональный стиль программирования» и два новых подхода к разработке программ.

Примеры кода из этой главы доступны по адресу **thinkpython2.com/code/Time1.py**.

Решения упражнений — в файле **thinkpython2.com/code/Time1_soln.py**.

КЛАСС `Time`

В качестве еще одного примера пользовательского типа мы определим класс `Time`, который хранит значение времени дня. Определение класса выглядит так:

```
class Time:
    """Определяет время суток.

    атрибуты: hour, minute, second
    """
```

Мы можем создать новый объект `Time` и присвоить значения атрибутам `hour`, `minute` и `second`:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

Диаграмма состояний объекта `Time` показана на рис. 16.1.

В качестве упражнения напишите функцию `print_time()`, которая принимает объект `Time` и печатает его в форме `hour:minute:second`. Подсказка:

шаблон форматирования `'%.2d'` печатает целое число, используя не менее двух цифр, при необходимости добавляя ноль слева.

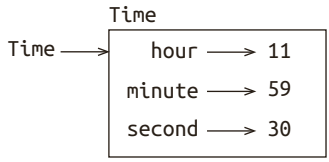


Рис. 16.1. Диаграмма объекта

Напишите логическую функцию `is_after()`, которая принимает два объекта `Time`, `t1` и `t2` и возвращает `True`, если `t1` хронологически следует за `t2`, и `False` в противном случае. Дополнительное задание: не используйте инструкцию `if`.

ЧИСТЫЕ ФУНКЦИИ

В следующих нескольких разделах мы напишем две функции, которые складывают значения времени. Они демонстрируют два вида функций: чистые функции и модификаторы. Они также демонстрируют подход к разработке, который я называю **«прототип и доработка» (prototype and patch)**. Это способ решения сложной задачи: начиная с простого, постепенно увеличивать сложность.

Ниже показан простой прототип функции `add_time()`:

```
def add_time(t1, t2):
sum = Time()
sum.hour = t1.hour + t2.hour
sum.minute = t1.minute + t2.minute
sum.second = t1.second + t2.second
return sum
```

Функция создает новый объект `Time`, инициализирует его атрибуты и возвращает ссылку на новый объект. Это так называемая **чистая функция (pure function)**: она не изменяет ни один из объектов, переданных ей в качестве аргументов, и не оказывает никакого эффекта, то есть не отображает значение или не получает пользовательский ввод, а только возвращает значение.

Чтобы протестировать эту функцию, я создам два объекта `Time`: `start` содержит время начала фильма, к примеру «Монти Пайтон и Священный Грааль», а `duration` — продолжительность фильма, 1 час 35 минут.

Функция `add_time()` выясняет, во сколько фильм закончится:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0
>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

Результат `10:80:00` весьма неожиданный. Проблема в том, что эта функция не обрабатывает случаи, когда количество секунд или минут превышает шестьдесят. В таких случаях мы должны «перенести» дополнительные секунды в столбец минут или дополнительные минуты в столбец часов.

Ниже показана доработанная версия:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

 return sum
```

Хотя эта функция корректна, очень уж она велика. Мы рассмотрим ее лаконичный вариант позже.

МОДИФИКАТОРЫ

Иногда функции нужно изменять объекты, которые она получает в качестве параметров. В этом случае изменения видны вызывающей функции. Функции, которые работают таким образом, называются **модификаторами (modifiers)**.

Функция `increment()`, которая добавляет указанное количество секунд к объекту `Time`, может быть оформлена как модификатор. Вот грубый набросок:

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

Первая строка выполняет основную операцию; остальное касается особых случаев, которые мы обсудили раньше.

Эта функция корректна? Что произойдет, если секунд намного больше, чем 60?

В этом случае недостаточно перенести единицу однократно; мы должны делать это, пока значение переменной `time.second` не станет меньше `60`. Одно из решений — замена инструкций `if` на инструкции `while`. Так функция будет правильной, но не очень эффективной. В качестве упражнения напишите правильную версию функции `increment()`, не используя циклы.

Все, что можно сделать с помощью модификаторов, реализуемо и с помощью чистых функций. Фактически некоторые языки программирования допускают использование только чистых функций. Есть некоторые наблюдения, что программы, использующие только чистые функции, быстрее разрабатывать и они менее подвержены ошибкам, чем программы, использующие модификаторы. Но и модификаторы иногда удобны, а функциональные программы, как правило, менее эффективны.

В общем, я рекомендую писать чистые функции всякий раз, когда это целесообразно, и прибегать к модификаторам, только если это решение заметно выигрывает по сравнению с первым. Такой подход можно назвать **функциональным стилем программирования (functional programming style)**.

В качестве упражнения напишите «чистую» версию функции `increment()`, которая создает и возвращает новый объект `Time`, а не изменяет параметр.

ПРОТОТИП ИЛИ ПЛАНИРОВАНИЕ

Показанный подход к разработке называется «прототип и доработка». Для каждой функции я написал прототип, который выполнял базовые вычисления, а затем по ходу дела проверял их, исправляя ошибки и дорабатывая.

Этот подход особенно эффективен, если вы еще не разобрались в проблеме как следует. Но постепенные исправления могут генерировать код, который окажется излишне сложен (поскольку нужно обрабатывать множество особых случаев) и ненадежен (так как трудно узнать, нашли ли вы все ошибки).

Альтернативой является **спроектированная разработка (designed development)**, в которой глубокое понимание проблемы значительно облегчает написание кода. В данном случае мы понимаем, что объект `Time` на самом деле представлен трехзначным числом в шестидесятеричной системе счисления (см. **ru.wikipedia.org/wiki/Шестидесятеричная_система_счисления**). Атрибут `second` — это «столбец единиц», `minute` — «столбец шестидесяти», а `hour` — «столбец тридцати шести сотен».

Когда мы писали функции `add_time()` и `increment()`, мы, по сути, производили сложение по основанию 60, поэтому нам пришлось переходить от одного столбца к другому.

Это наблюдение предполагает другой подход ко всей проблеме — мы можем преобразовать объекты `Time` в целые числа и учесть тот факт, что компьютер знает, как выполнять целочисленные вычисления.

Ниже показана функция, которая преобразует объекты `Time` в целые числа:

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

А вот функция, которая преобразует целое число в `Time` (напомню, что функция `divmod()` делит первый аргумент на второй и возвращает частное и остаток как кортеж):

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

Возможно, вам придется немного обдумать это решение и выполнить несколько тестов, чтобы убедиться, что функции работают правильно. Один из способов проверить их — проверить, что `time_to_int (int_to_time (x)) == x` для многих значений `x`. Это пример проверки согласованности.

Как только вы убедитесь, что они верны, вы можете с их помощью переписать функцию `add_time()`:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Эта версия короче оригинала и проще в смысле проверки. В качестве упражнения перепишите функцию `increment()`, используя функции `time_to_int()` и `int_to_time()`.

В некотором смысле выполнить преобразование из основания 60 в основание 10 и обратно сложнее, чем просто обработать значение времени. Преобразование из одной системы счисления в другую более абстрактно; наше понимание значений времени намного проще.

Но если мы будем рассматривать время как числа с основанием 60 и вкладывать силы в разработку функций преобразования (`time_to_int()` и `int_to_time()`), мы получим программу, которая будет короче, проще для чтения и отладки и более надежна.

А еще в нее легче добавить функции. Например, представьте, что надо вычесть значение одного объекта `Time` из другого, чтобы найти продолжительность чего-либо. Наивный подход заключается в реализации вычитания с заимствованием. Но будет проще и, скорее всего, правильнее использовать функции преобразования.

По иронии судьбы, иногда усложнение задачи (или обобщение) упрощает ее (потому что меньше особых случаев и меньше возможностей для ошибок).

ОТЛАДКА

Объект `Time` сформирован правильно, если значения минут и секунд находятся в диапазоне от 0 до 60 (включая 0, но не 60) и если значение часа положительное. Часы и минуты должны быть целыми значениями, но секунды могут иметь дробную часть.

Подобные требования называются **инвариантами (invariants)**, потому что они всегда должны быть верными. Иными словами, если они не соответствуют действительности, что-то пошло не так.

Написание кода для проверки инвариантов помогает обнаружить ошибки и их причины. Например, у вас может быть функция типа `valid_time()`, которая принимает объект `Time` и возвращает `False`, если она нарушает инвариант:

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
```

```
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

В начале каждой функции вы можете проверить аргументы, чтобы убедиться, что они допустимы:

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Или вы можете использовать **инструкцию** `assert`, которая проверяет заданный инвариант и вызывает исключение в случае сбоя:

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Инструкция `assert` удобна, потому что различает основной код и код проверки на ошибки.

СЛОВАРЬ ТЕРМИНОВ

Прототип и доработка:
: Подход к разработке, предусматривающий написание чернового варианта программы, тестирование и исправление ошибок по мере их обнаружения.

Спроектированная разработка:
: Подход к разработке, который включает в себя понимание проблемы на высоком уровне и больше планирования, чем поэтапная разработка или разработка прототипов.

Чистая функция:
: Функция, не изменяющая ни один из объектов, которые она получает в качестве аргументов. Большинство чистых функций возвращают значения.

Модификатор:
: Функция, которая изменяет один или несколько объектов, которые она получает в качестве аргументов. Большинство модификаторов являются пустыми функциями; то есть возвращают `None`.

Функциональный стиль программирования:
: Стиль разработки программы, в котором большинство функций чистые.

Инвариант:
: Условие, которое всегда должно быть истинным во время выполнения программы.

Инструкция `assert`*:*
: Инструкция, которая проверяет условие и вызывает исключение в случае, если оно не выполняется.

УПРАЖНЕНИЯ

Примеры кода из этой главы доступны по адресу **thinkpython2.com/code/Time1.py**; решения упражнений — по адресу **thinkpython2.com/code/Time1_soln.py**.

Упражнение 16.1
Напишите функцию `mul_time()`, которая принимает объект `Time` и число и возвращает новый объект `Time`, который содержит произведение исходного времени и числа.

Затем на основе `mul_time()` напишите функцию, которая принимает объект `Time`, представляющий время окончания гонки, и число, представляющее расстояние, и возвращает объект `Time`, представляющий среднюю скорость (минут на один километр или милю).

Упражнение 16.2
Модуль `datetime` предоставляет объекты `time`, похожие на объекты `Time` в этой главе, но с более богатым набором методов и инструкций. Прочитайте документацию по адресу **docs.python.org/3/library/datetime.html**.

1. Используйте модуль `datetime`, чтобы написать программу, которая получает текущую дату и выводит день недели.
2. Напишите программу, которая принимает в качестве ввода день рождения и выводит возраст пользователя и количество дней, часов, минут и секунд до следующего дня рождения.
3. Для двух людей, родившихся в разные дни, есть день, когда один в два раза старше другого. Это так называемый *двойной день*. Напишите

программу, которая принимает два дня рождения и вычисляет двойной день именинников.

4. Чтобы усложнить задание, напишите обобщенную версию, которая вычисляет день, когда один человек в *n* раз старше другого.

Решение: **thinkpython2.com/code/double.py**.

ГЛАВА 17
КЛАССЫ И МЕТОДЫ

Хотя мы используем некоторые объектно-ориентированные возможности Python, программы из двух последних глав на самом деле нельзя назвать таковыми. Дело в том, что они не представляют отношения между пользовательскими типами и функциями, которые с ними работают. Следующий шаг — преобразование этих функций в методы, явно обозначающие эти отношения.

Примеры кода из этой главы доступны по адресу **thinkpython2.com/code/Time2.py**, а решения упражнений — по адресу **thinkpython2.com/code/Point2_soln.py**.

ПРИЗНАКИ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Python — это **объектно-ориентированный язык программирования (object-oriented programming language)**; то есть он предоставляет возможности, поддерживающие объектно-ориентированное программирование (ООП), которые имеют следующие определяющие характеристики.

— Программы состоят из определений классов и методов.
— Большинство вычислений выражается в виде операций над объектами.
— Объекты часто описывают вещи в реальном мире, а методы часто соответствуют способам взаимодействия с ними.

Например, класс `Time` из главы 16 определяет, как люди записывают время суток, а функции, которые мы определили, соответствуют тому, что люди делают со временем. Точно так же классы `Point` и `Rectangle` из главы 15 соответствуют математическим понятиям точки координат и прямоугольника.

До сих пор мы не использовали возможности Python для поддержки ООП. Эти функции не строго необходимы; большинство из них предоставляют

альтернативный синтаксис для операций, которые мы уже выполняли. Но во многих случаях альтернатива более лаконична и точнее передает структуру программы.

Например, в файле *Time1.py* нет очевидной связи между определением класса и определениями функций, которые следуют за ним. При некотором рассмотрении становится очевидным, что каждая функция принимает в качестве аргумента хотя бы один объект `Time`.

Это наблюдение мотивирует на создание **методов (methods)**. Метод — это функция, связанная с определенным классом. Мы уже видели методы для строк, списков, словарей и кортежей. В этой главе мы определим методы для пользовательских типов.

Методы семантически совпадают с функциями, но есть два синтаксических различия.

— Методы определены внутри определения класса, чтобы создать явную связь между классом и методом.
— Синтаксис вызова метода отличается от синтаксиса вызова функции.

В следующих разделах мы возьмем функции из двух предыдущих глав и преобразуем их в методы. Это преобразование чисто механическое, достаточно просто выполнить последовательность шагов. Если вам хорошо удаются переходы из одной формы в другую, вы сможете подбирать самые подходящие формы под конкретные задачи.

ПЕЧАТЬ ОБЪЕКТОВ

В главе 16 мы определили класс `Time` и в разделе «Класс `Time`» той же главы написали функцию `print_time()`:

```
class Time:
    """Определяет время суток."""

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Чтобы вызвать эту функцию, вы должны передать объект `Time` в качестве аргумента:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
```

```
>>> print_time(start)
09:45:00
```

Чтобы превратить функцию `print_time()` в метод, все, что нам нужно сделать, — это переместить определение функции в определение класса. Обратите внимание на изменение отступа.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Теперь есть два способа вызвать метод `print_time()`. Первый (и менее распространенный) способ — использовать синтаксис функции:

```
>>> Time.print_time(start)
09:45:00
```

В этом варианте точечной нотации `Time` — это имя класса, а `print_time` — это имя метода; `start` передается как параметр.

Второй (и более краткий) способ — использовать синтаксис метода:

```
>>> start.print_time()
09:45:00
```

В этом варианте точечной нотации `print_time` — это имя метода (опять же), а `start` — это объект, к которому вызывается метод, он называется **субъектом (subject)**. Так же, как субъект предложения (подлежащее) — то, о чем предложение, субъект вызова метода — то, о чем метод.

Внутри метода значение субъекта присваивается первому параметру, поэтому в этом случае `start` присваивается переменной `time`.

По соглашению первый параметр метода называется `self`, поэтому обычно метод наподобие `print_time()` определяют следующим образом:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

Причина этого соглашения — в неявной метафоре.

— Синтаксис для вызова функции `print_time(start)` предполагает, что функция является активным агентом. Он говорит что-то вроде: «Эй, `print_time()`! Вот объект для тебя, напечатай его».
— В объектно-ориентированном программировании объекты являются активными агентами. Вызов метода, такого как `start.print_time()`, говорит: «Эй, `start`! Пожалуйста, напечатай себя».

Такое изменение в перспективе выглядит более изящным, но его полезность не очевидна. Как в примерах, которые мы видели до сих пор, например. Но иногда смещение ответственности с функций на объекты позволяет писать более универсальные функции (или методы), а также упрощает поддержку и повторное использование кода.

В качестве упражнения перепишите функцию `time_to_int()` (из раздела «Прототип или планирование» главы 16) как метод. У вас может возникнуть соблазн переписать и функцию `int_to_time()` как метод, но это не имеет смысла, потому что у нас нет объекта, чтобы его вызвать.

ЕЩЕ ПРИМЕР

Ниже показана версия функции `increment()` (из раздела «Модификаторы» главы 16), переписанная как метод:

```
# внутри класса Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

Эта версия предполагает, что функция `time_to_int()` реализована как метод. Также обратите внимание, что это чистая функция, а не модификатор.

Вот как бы вы вызывали функцию `increment()`:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

Субъект `start` присваивается первому параметру, `self`. Аргумент `1337` присваивается второму параметру, `seconds`.

Этот механизм может сбивать с толку, вы часто будете совершать ошибки. Например, если вы вызываете функцию `increment()` с двумя аргументами, вы получите:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

Сообщение об ошибке изначально сбивает с толку, потому что в скобках есть только два аргумента. Но субъект также считается аргументом, так что их всего три.

Кстати, **позиционным (positional argument)** называется аргумент, который не имеет имени параметра (ключевого слова). В вызове этой функции

```
sketch(parrot, cage, dead=True)
```

аргументы `parrot` и `cage` позиционные, а `dead` — это ключевой аргумент.

БОЛЕЕ СЛОЖНЫЙ ПРИМЕР

Переписать функцию `is_after()` (из раздела «Класс `Time`» предыдущей главы) немного сложнее, потому что эта функция принимает два объекта `Time` в качестве параметров. В этом случае принято называть первый параметр `self`, а второй — `other`:

```
# внутри класса Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

Чтобы использовать этот метод, вы должны вызвать его для одного объекта и передать другой в качестве аргумента:

```
>>> end.is_after(start)
True
```

Особенность этого синтаксиса в том, что он в переводе с английского он читается так: «окончание после запуска».

МЕТОД `INIT`

`init` (сокращение от “initialization” — «инициализация») — это специальный метод, который вызывается при создании экземпляра объекта. Его полное имя `__init__` (два символа подчеркивания, затем `init`, а затем еще два подчеркивания). Метод инициализации для класса `Time` может выглядеть следующим образом:

```
# внутри класса Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

Обычно параметры `__init__` называют теми же именами, что и атрибуты.

Операция

```
self.hour = hour
```

сохраняет значение параметра `hour` в качестве атрибута `self`.

Параметры необязательны, поэтому, если вы вызовете `Time` без аргументов, то получите значения по умолчанию:

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

Если вы предоставите один аргумент, он переопределит часы:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

Если вы предоставите два аргумента, они переопределят часы и минуты:

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

И если вы предоставите три аргумента, они переопределят все три значения по умолчанию.

Потренируйтесь и напишите метод инициализации для класса `Point`, который принимает `x` и `y` в качестве необязательных параметров и назначает их соответствующим атрибутам.

МЕТОД __STR__

`__str__` — это специальный метод, такой же как `__init__`, который должен возвращать строковое представление объекта.

В качестве примера ниже показан метод `str` для объектов `Time`:

```
# внутри класса Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

Когда вы передаете объект функции `print()`, Python вызывает метод `str`:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

Когда я создаю новый класс, я почти всегда начинаю с `__init__`, чтобы упростить создание экземпляров объектов, и `__str__`, что полезно для отладки.

В качестве упражнения напишите метод `str` для класса `Point`. Создайте объект `Point` и напечатайте его.

ПЕРЕГРУЗКА ОПЕРАТОРОВ

Определив другие специальные методы, вы можете настроить поведение операторов с пользовательскими типами. Например, если вы определяете метод `__add__` для класса `Time`, то можете использовать оператор + для объектов `Time`.

Вот как может выглядеть определение:

```
# внутри класса Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

И вот как вы можете использовать этот метод:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

Когда вы применяете оператор + к объектам `Time`, Python вызывает метод `__add__`. Когда вы печатаете результат, Python вызывает метод `__str__`. Так что за кадром происходит много всего!

Изменение поведения оператора таким образом, чтобы он работал с пользовательскими типами, называется **перегрузкой оператора (operator overloading)**. Для каждого оператора в Python есть соответствующий метод, вроде `__add__`. Для получения дополнительной информации см. страницу **docs.python.org/3/reference/datamodel.html#specialnames**.

В качестве упражнения напишите метод `add` для класса `Point`.

ДИСПЕТЧЕРИЗАЦИЯ НА ОСНОВЕ ТИПОВ

В предыдущем разделе мы складывали два объекта `Time`, но вы также можете прибавить целое число к объекту `Time`. Ниже приведена версия метода

__add__, который проверяет тип other и вызывает функции add_time() или increment():

```
# внутри класса Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

Встроенная функция isinstance() принимает значение и объект класса и возвращает True, если значение является экземпляром класса.

Если other является объектом Time, метод __add__ вызывает функцию add_time(). В противном случае предполагается, что параметр числовой и вызывается функция increment(). Эта операция называется **диспетчеризацией на основе типа (type-based dispatch)**, поскольку отправляет вычисления различными методами в зависимости от типа аргументов.

Ниже показаны примеры, которые используют оператор + с разными типами:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

К сожалению, эта реализация сложения не позволяет переставлять местами слагаемые. Если целое число представлено первым операндом, вы получите исключение:

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

Проблема в том, что вместо того, чтобы просить объект Time добавить целое число, Python просит целое число добавить объект Time, не зная, как это

сделать. Но есть умное решение этой проблемы: специальный метод `__radd__`, который расшифровывается “right-side add” — «сложение с правой стороны». Этот метод вызывается, когда объект `Time` появляется справа от оператора +. Ниже приведено определение:

```
# внутри класса Time:

    def __radd__(self, other):
        return self.__add__(other)
```

А здесь показан пример использования:

```
>>> print(1337 + start)
10:07:17
```

В качестве упражнения напишите метод `add` для класса `Point`, который работает с объектом `Point` или с кортежем.

— Если второй операнд — объект `Point`, метод должен вернуть новую точку, координата `x` которой является суммой координат `x` операндов, и так же для координат `y`.
— Если второй операнд — кортеж, метод должен добавить первый элемент кортежа к координате `x`, а второй элемент — к координате `y` и вернуть новую точку.

ПОЛИМОРФИЗМ

Диспетчеризация на основе типов полезна в случае необходимости, но (к счастью) эти случаи не так часты. Обойтись без нее можно, написав функции, которые корректно работают с аргументами разных типов.

Многие функции, которые мы написали для строковых значений, также применимы и к другим типам последовательностей. К примеру, в разделе «Словарь как набор счетчиков» главы 11 мы использовали гистограмму, чтобы подсчитать, сколько раз каждая буква встречается в слове:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

Эта функция также работает со списками, кортежами и даже словарями: если элементы `s` хешируются, то их можно использовать в качестве ключей в `d`:

```
>>> t = ['спам', 'яичница', 'спам', 'спам', 'бекон', 'спам']
>>> histogram(t)
{'спам': 4, 'яичница': 1, 'бекон': 1}
```

Функции, которые поддерживают работу с несколькими типами, называются **полиморфными (polymorphic)**. Полиморфизм упрощает повторное использование кода. Например, встроенная функция `sum()`, которая складывает элементы последовательности, работает до тех пор, пока элементы последовательности поддерживают сложение.

Поскольку объекты `Time` предоставляют метод `add()`, функция `sum()` будет с ними работать:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

В общем, если все операции внутри функции поддерживают заданный тип, функция работает с этим типом.

Лучший вид полиморфизма — непреднамеренный, когда вы обнаруживаете, что уже написанная функция применима к типу, который вы никогда не планировали с ней использовать.

ИНТЕРФЕЙС И РЕАЛИЗАЦИЯ

Одна из задач объектно-ориентированного проектирования — сделать сопровождение и обновление программного обеспечения более удобными. Это означает, что программа будет работоспособна при изменении ее частей и при появлении новых требований.

Принцип проектирования, который помогает решать эту задачу, состоит в том, чтобы отделить интерфейсы от реализаций. Для объектов это означает, что методы, предоставляемые классом, не должны зависеть от того, как представлены его атрибуты.

Например, в этой главе мы разработали класс, который представляет время суток. Методы, предоставляемые этим классом, включают `time_to_int()`, `is_after()` и `add_time()`.

Мы могли бы реализовать эти методы несколькими способами. Детали реализации зависят от того, как мы представляем время. В этой главе атрибуты объекта `Time` — это `hour`, `minute` и `second`.

В качестве альтернативы мы могли бы заменить эти атрибуты одним целым числом, представляющим количество секунд, прошедших с полуночи. Эта реализация упрощает написание одних методов, таких как `is_after()`, но усложняет другие.

После создания класса, возможно, вы придумаете лучший способ реализации. Если другие части программы используют ваш класс, изменение интерфейса может занять много времени и там могут появиться ошибки.

Но если вы тщательно спроектировали интерфейс, то можете изменить реализацию, не меняя интерфейс, что означает, что другие части программы не нужно будет трогать.

ОТЛАДКА

Вполне можно добавлять атрибуты к объектам в любой момент выполнения программы, но, если у вас есть объекты одного типа, которые не имеют одинаковых атрибутов, легко допустить ошибки. Поэтому рекомендуется инициализация всех атрибутов объекта в методе `__init__`.

Если вы не уверены, имеет ли объект определенный атрибут, используйте встроенную функцию `hasattr()` (см. раздел «Отладка» главы 15).

Другой способ доступа к атрибутам — встроенная функция `vars()`. Она принимает объект и возвращает словарь, который сопоставляет имена атрибутов (в виде строк) с их значениями:

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

В целях отладки может пригодиться и показанная ниже функция:

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

Функция `print_attributes()` обходит словарь и печатает имя каждого атрибута и соответствующее ему значение.

Встроенная функция `getattr()` принимает объект и имя атрибута (в виде строки) и возвращает значение атрибута.

СЛОВАРЬ ТЕРМИНОВ

Объектно-ориентированный язык:
Язык, который предоставляет функции, такие как пользовательские типы и методы, которые облегчают объектно-ориентированное программирование.

Объектно-ориентированное программирование (ООП):
Стиль программирования, в котором данные и операции над ними организованы в классы и методы.

Метод:
Функция, которая определена внутри класса и вызывается для экземпляров этого класса.

Субъект:
Объект, на который вызывается метод.

Позиционный аргумент:
Аргумент, который не включает имя параметра, и поэтому не является ключевым аргументом.

Перегрузка операторов:
Изменение поведения оператора таким образом, чтобы он работал с пользовательскими типами.

Диспетчеризация на основе типов:
Шаблон программирования, который проверяет тип операнда и вызывает разные функции для разных типов.

Полиморфный:
Относящийся к функции, которая может работать более чем с одним типом.

Сокрытие информации:
Принцип, согласно которому интерфейс, предоставляемый объектом, не должен зависеть от его реализации, в частности, от представления его атрибутов.

УПРАЖНЕНИЯ

Упражнение 17.1
Скачайте файл по адресу **thinkpython2.com/code/Time2.py**. Замените атрибуты `Time` на одно целое число, представляющее секунды, прошедшие

с полуночи. Затем измените методы (и функцию `int_to_time()`) для поддержки новой реализации. Вам не нужно изменять тестовый код в методе `main()`. Когда вы закончите, вывод должен быть таким же, как и раньше.

Решение: **thinkpython2.com/code/Time2_soln.py**.

Упражнение 17.2

Это упражнение — скорее предостережение об одной из самых распространенных и трудно уловимых ошибок в Python. Напишите определение для класса `Kangaroo` с помощью следующих методов:

1. Метод `__init__`, который инициализирует атрибут `pouch_contents` в пустой список.
2. Метод `put_in_pouch()`, который берет объект любого типа и добавляет его в `pouch_contents`.
3. Метод `__str__`, который возвращает строковое представление объекта `Kangaroo` и содержимое его сумки (`pouch`).

Протестируйте свой код, создав два объекта `Kangaroo`, назначив их переменным с именами `kanga` и `roo`, а затем добавив `roo` к содержимому сумки `kanga`.

Скачать файл можно по адресу **thinkpython2.com/code/BadKangaroo.py**. В нем показано решение предыдущей задачи с одной большой и неприятной ошибкой. Найдите и исправьте ошибку.

Если вы зашли в тупик, то можете скачать файл **thinkpython2.com/code/GoodKangaroo.py**, где объясняется проблема и демонстрируется решение.

ГЛАВА 18
НАСЛЕДОВАНИЕ

Наследование (inheritance) — особенность языка, которая чаще всего ассоциируется с объектно-ориентированным программированием. Наследование дает возможность определить новый класс, являющийся модифицированной версией уже существующего класса. В этой главе я продемонстрирую наследование, используя классы, которые представляют игральные карты, карточные колоды и покерные комбинации.

Если вы не играете в покер, то можете подробнее прочитать о нем по адресу **ru.wikipedia.org/wiki/Покер**, но это необязательно; я расскажу все, что нужно знать, чтобы выполнить упражнения.

Примеры кода из этой главы доступны по адресу **thinkpython2.com/code/Card.py**.

ОБЪЕКТЫ КАРТ

В колоде 52 карты, каждая из которых принадлежит одной из четырех мастей и одному из тринадцати рангов. Масти — пики, червы, бубны и трефы (приведены в порядке убывания при игре в бридж). Ранги — Туз, 2, 3, 4, 5, 6, 7, 8, 9, 10, Валет, Дама и Король. В зависимости от разновидности покера, в которую вы играете, туз может быть выше короля или ниже 2.

Если мы хотим определить новый объект для представления игральной карты, очевидно, какими должны быть атрибуты: ранг и масть — `rank` и `suit`. Не совсем понятно, какого типа должны быть атрибуты. Одна из идей — использовать строковые значения со словами вроде `Spade` (пики) для мастей и `Queen` (дама) для рангов. Но тогда будет нелегко сравнивать карты, чтобы определить, какая имеет более высокий ранг или масть.

Альтернатива — использование целых чисел для **кодирования (encode)** рангов и мастей. В этом контексте «кодировать» означает, что мы собираемся определить соответствие между числами и мастями или между числами и рангами. Этот вид кодирования не секретный (иначе мы бы назвали его «шифрование»).

Например, эта таблица показывает масти и соответствующие им целочисленные коды:

```
Пики  → 3
Червы → 2
Бубны → 1
Трефы → 0
```

Такие коды упрощают сравнение карт; поскольку более высокие масти соответствуют бóльшим числам, мы можем сравнивать масти, сравнивая их коды.

Сопоставление рангов также теперь ясно; каждый из числовых рангов сопоставляется с соответствующим целым числом, а для фигур так:

```
Туз    → 1
Валет  → 11
Дама   → 12
Король → 13
```

Я использую символ →, чтобы было ясно, что эти сопоставления не являются частью программы Python. Это часть проектирования программы, и в коде эти сопоставления явно не появляются.

Определение класса `Card` выглядит так:

```
class Card:
    """Определяет обычную игральную карту."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

Как обычно, метод инициализации принимает необязательный параметр для каждого из атрибутов. Карта по умолчанию — это двойка треф.

Чтобы создать карту, вы вызываете класс `Card` с мастью и рангом нужной вам карты в качестве аргументов:

```
queen_of_diamonds = Card(1, 12)
```

АТРИБУТЫ КЛАССА

Чтобы вывести объекты `Card` в более привычной для нас форме, нам необходимо сопоставить целочисленные коды с соответствующими рангами и мастями. Очевидный способ сделать это — с помощью списков строк. Мы присваиваем эти списки **атрибутам класса (class attributes)**:

```
# внутри класса Card:

    suit_names = ['Трефы', 'Бубны', 'Червы', 'Пики']
    rank_names = [None, 'Туз', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'Валет', 'Дама', 'Король']

    def __str__(self):
        return '%s масти %s' % (Card.rank_names[self.rank],
                                Card.suit_names[self.suit])
```

Переменные, такие как `suit_names` и `rank_names`, которые определены внутри класса, но вне любого метода, называются атрибутами класса, потому что они связаны с объектом класса `Card`.

Этот термин отличает их от таких переменных, как `suit` и `rank`, которые называются **атрибутами экземпляра (instance attributes)**, поскольку связаны с конкретным экземпляром.

Оба вида атрибутов доступны с использованием точечной нотации. Например, в методе `__str__`, `self` — это объект `Card`, а `self.rank` — его ранг. Аналогично `Card` является объектом класса, а `Card.rank_names` — списком строк, связанных с классом.

У каждой карты есть собственные масть и ранг, но есть только одна копия переменных `suit_names` и `rank_names` для всех.

То есть выражение `Card.rank_names[self.rank]` означает «использовать атрибут `rank` объекта `self` в качестве индекса в списке `rank_names` класса `Card` и выбрать соответствующую строку».

Первый элемент в списке `rank_names` — `None`, потому что нет карты с нулевым рангом. Используя `None` в качестве заполнителя, мы получаем точное сопоставление индексов с рангами, а именно индекс `2` сопоставляется со строкой `'2'`, и так далее. Хотя мы могли бы использовать и словарь вместо списка.

Используя методы, которые у нас есть, мы можем создавать и печатать игральные карты:

```
>>> card1 = Card(2, 11)
>>> print(card1)
Валет масти Червы
```

На рис. 18.1 показана схема объекта класса `Card` и одного экземпляра `Card`. `Card` — это объект типа `class`; поэтому он имеет тип `type`. А `card1` — экземпляр `Card`, поэтому его тип — `Card`. Чтобы сэкономить место, я не рисовал содержимое переменных `suit_names` и `rank_names`.

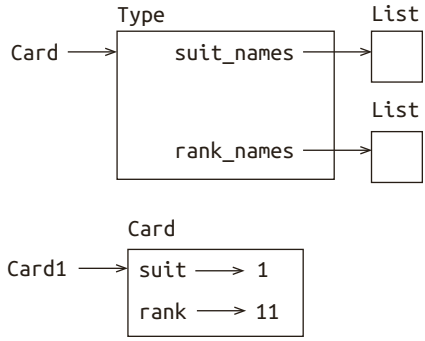


Рис. 18.1. Диаграмма объекта

СРАВНЕНИЕ КАРТ

Для встроенных типов существуют операторы сравнения (<, >, == и т. д.), которые сравнивают значения и определяют, какой операнд больше, а какой меньше или они равны.

Для пользовательских типов мы можем переопределить поведение встроенных операторов, используя метод `__lt__`, который означает «меньше, чем (less than)».

Метод `__lt__` принимает два параметра, `self` и `other`, и возвращает `True`, если `self` строго меньше, чем `other`.

Правильный порядок возрастания значимости карт не очевиден. Например, что лучше — тройка треф или двойка бубен? Одна карта имеет более высокий ранг, а другая — более высокую масть. Чтобы сравнить карты, вы должны решить, что важнее: ранг или масть.

Ответ может зависеть от того, в какую версию покера вы играете, но, чтобы не усложнять ситуацию, мы решим, что масть более важна, поэтому все пики превосходят все бубны и так далее.

После этого мы можем написать `__lt__`:

```
# внутри класса Card:

    def __lt__(self, other):
        # проверка масти
        if self.suit < other.suit: return True
        if self.suit > other.suit: return False

        # Масти одинаковые, проверить ранги
        return self.rank < other.rank
```

Вы можете написать код лаконичнее, сравнивая кортежи:

```
# внутри класса Card:

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

В качестве упражнения напишите метод `__lt__` для объектов `Time`. Вы можете сравнивать не только кортежи, но и целые числа.

КОЛОДЫ

Теперь, когда у нас есть карты, следующий шаг — определить колоды. Поскольку колода состоит из карт, для каждой колоды естественно указывать список карт в качестве атрибута.

Ниже приведено определение класса `Deck`. Метод инициализации создает атрибут `cards` и генерирует стандартный набор из 52 карт:

```
class Deck:

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

Самый простой способ заполнить колоду — это вложенный цикл. Внешний цикл перечисляет масти от 0 до 3. Внутренний цикл перечисляет ранги от 1 до 13. Каждая итерация создает новую карту с текущей мастью и рангом и добавляет ее к переменной `self.cards`.

ПЕЧАТЬ КОЛОДЫ

Ниже показан метод `__str__` класса `Deck`:

```
# внутри класса Deck:

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

Этот метод демонстрирует эффективный способ накопления большой строки: создание списка строк и затем использование строкового метода `join()`. Встроенная функция `str()` вызывает метод `__str__` для каждой карты и возвращает ее строковое представление.

Поскольку мы вызываем метод `join()` для символа новой строки (`\n`), карты указываются по одной на каждой строке.

Вот как выглядит результат (сокращенный):

```
>>> deck = Deck()
>>> print(deck)
Туз масти Трефы
2 масти Трефы
3 масти Трефы
...
10 масти Пики
Валет масти Пики
Дама масти Пики
Король масти Пики
```

Технически это одна строка, но пользователь видит результат на 52 строках из-за того, что непечатаемые символы переводят «каретку» на новую строку.

ДОБАВЛЕНИЕ, УДАЛЕНИЕ, ТАСОВАНИЕ И СОРТИРОВКА

Для раздачи карт нам нужен метод, который удаляет карту из колоды и возвращает ее. Для этого есть удобный метод `pop()` списка:

```
# внутри класса Deck:

    def pop_card(self):
        return self.cards.pop()
```

Поскольку метод `pop()` удаляет последнюю карту в списке, мы берем карты снизу колоды.

Чтобы добавить карту, мы можем использовать метод `append()` списка:

```
# внутри класса Deck:

    def add_card(self, card):
        self.cards.append(card)
```

Подобный метод, который использует другой метод, не выполняя дополнительной работы, иногда называют **декорирующим (veneer)**. Метафора

происходит из области деревообработки, где тонкий слой дерева хорошего качества клеят поверх более дешевой древесины и тем самым улучшают внешний вид.

В нашем случае `add_card()` — это «тонкий» метод, который выражает операцию списка в терминах, подходящих для колод. Что также улучшает внешний вид или интерфейс реализации.

Другим примером может служить метод `shuffle()` класса `Deck`, который использует одноименную функцию из модуля `random`:

```
# внутри класса Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

Не забудьте импортировать модуль `random`.

В качестве упражнения напишите метод `sort()` класса `Deck`, который использует метод списка `sort()` для сортировки карт в колоде. Метод `sort()` будет задействовать метод `__lt__`, который мы определили, чтобы указать порядок.

НАСЛЕДОВАНИЕ

Наследование (inheritance) позволяет определить новый класс — модифицированную версию уже существующего класса. Например, мы хотим, чтобы класс представлял «руку» (hand), то есть набор карт, которые держит один игрок. Рука похожа на колоду: обе составлены из набора карт и требуют операций, таких как добавление и удаление карт.

При этом рука отличается от колоды; есть операции, которые мы хотим определить для рук, но которые не имеют смысла для колоды. Например, в покере мы можем сравнить комбинации двух игроков, чтобы увидеть, какая выигрывает. В бридже — посчитать очки у одного игрока, чтобы знать, какую сделать ставку.

Эти отношения между классами — похожие, но разные — поддаются наследованию. Чтобы определить новый класс, который наследуется от существующего, поместите имя существующего класса в круглые скобки:

```
class Hand(Deck):
    """Определяет игральные карты в руке."""
```

Это определение указывает, что класс `Hand` наследуется от класса `Deck`; это означает, что мы можем использовать такие методы, как `pop_card()` и `add_card()` для класса `Hand`, так же как и для класса `Deck`.

Когда новый класс наследуется от существующего, существующий называется **родительским (parent)**, а новый класс — **дочерним (child)**.

В этом примере класс `Hand` наследует метод `__init__` от класса `Deck`, но на самом деле делает не то, что нам нужно: вместо заполнения руки 52 новыми картами метод инициализации класса `Hand` должен инициализировать переменную `cards` как пустой список.

Если мы предоставляем метод инициализации в классе `Hand`, он переопределяет тот же метод класса `Deck`:

```
# внутри класса Hand:
# inside class Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

При создании класса `Hand` Python вызывает данный метод инициализации, а не метод, определенный в классе `Deck`.

```
>>> hand = Hand('новая рука')
>>> hand.cards
[]
>>> hand.label
'новая рука'
```

Другие методы унаследованы от класса `Deck`, поэтому мы можем использовать методы `pop_card()` и `add_card()` для раздачи карты:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

Дальше будет естественной инкапсуляция этого кода в метод `move_cards()`:

```
# внутри класса Deck:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

Метод `move_cards()` принимает два аргумента: объект `Hand` и количество карт для раздачи. Он изменяет как `self`, так и `hand`, и возвращает `None`.

В некоторых играх карты перемещаются из одной руки в другую или обратно в колоду. Вы можете использовать метод `move_cards()` для любой

из этих операций: `self` может быть и колодой, и рукой, а `hand`, несмотря на название, также может быть `Deck`.

Наследование — полезная функция. С ним можно писать более элегантные и лаконичные программы. Наследование упрощает повторное использование кода, так как можно настроить поведение родительских классов, а не менять их. В некоторых случаях структура наследования отражает естественную структуру проблемы, что облегчает понимание проекта.

С другой стороны, наследование затрудняет чтение кода программ. При вызове метода иногда неясно, где найти его определение. Соответствующий код может быть распределен по нескольким модулям. Кроме того, многие задачи, которые выполняются с помощью наследования, можно решить без него точно так же или даже лучше.

ДИАГРАММЫ КЛАССОВ

До сих пор мы видели стековые диаграммы, которые отражают состояние программы, и диаграммы объектов, которые показывают атрибуты объекта и их значения. Эти диаграммы представляют собой моментальный снимок при выполнении программы, поэтому они меняются по мере работы программы.

Они очень подробны; для некоторых целей даже чрезмерно. Диаграмма классов — более абстрактное представление структуры программы. Вместо того чтобы показывать отдельные объекты, она отражает классы и отношения между ними.

Существует несколько типов отношений между классами.

— Объекты в одном классе могут содержать ссылки на объекты в другом классе. Например, каждый прямоугольник Rectangle содержит ссылку на точку Point, а каждая колода Deck содержит ссылки на множество карт Card. Этот тип отношений называется **`HAS-A`** (a Rectangle has a Point).

— Один класс может наследоваться от другого. Например, рука Hand в каком-то смысле является колодой Deck. Этот тип отношений называется **`IS-A`** (a hand is a kind of a Deck).

— Один класс может зависеть от другого в том смысле, что объекты в одном классе принимают объекты во втором классе в качестве параметров или используют объекты во втором классе как часть

вычисления. Этот вид отношений называется **зависимостью (dependency)**.

Диаграмма классов — это графическое представление этих отношений. В качестве примера на рис. 18.2 показаны отношения между классами `Card`, `Deck` и `Hand`.

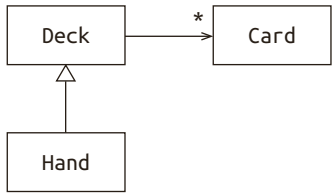


Рис. 18.2. Диаграмма классов

Стрелка между классами `Hand` и `Deck` представляет отношение IS-A; в этом случае указывает на то, что класс `Hand` наследуется от класса `Deck`.

Стрелка между классами `Deck` и `Card` представляет отношение HAS-A; в этом случае класс `Deck` содержит ссылки на объекты `Card`.

Звездочка (*) возле стрелки — это **множественность (multiplicity)**; она указывает, сколько карт содержит колода. Множественность может быть простым числом, таким как `52`, диапазоном типа `5...7`, или звездочкой, которая указывает, что в колоде может быть сколько угодно карт.

На этой диаграмме нет зависимостей. Они обычно отображаются пунктирной стрелкой. Или, если зависимостей много, их иногда опускают.

Более подробная диаграмма может показать, что колода фактически содержит список карт, но встроенные типы, такие как `list` и `dict`, обычно не включаются в диаграммы классов.

ИНКАПСУЛЯЦИЯ ДАННЫХ

Предыдущие главы демонстрируют подход, который мы могли бы назвать «объектно-ориентированной разработкой». Мы определили нужные нам объекты, такие как `Point`, `Rectangle` и `Time`, и классы для их представления. В каждом случае есть очевидное соответствие между объектом и некоторой сущностью в реальном мире (или, по крайней мере, в математическом).

Не всегда очевидно, какие объекты вам нужны и как они должны взаимодействовать. В этом случае вам требуется другой подход к разработке. Точно

так же, как мы открыли интерфейсы функций путем инкапсуляции и обобщения, мы можем обнаружить интерфейсы классов путем **инкапсуляции данных (data encapsulation)**.

Хороший пример — анализ цепи Маркова из одноименного раздела главы 13. Если вы загрузите мой код по ссылке **thinkpython2.com/code/markov.py**, то увидите, что там используются две глобальные переменные — `suffix_map` и `prefix`, которые считываются и записываются в нескольких функциях.

```
suffix_map = {}
prefix = ()
```

Поскольку это глобальные переменные, мы можем выполнять только один анализ за раз. Если мы прочитаем два текста, их префиксы и суффиксы будут добавлены к одним и тем же структурам данных (что создаст интересный сгенерированный текст).

Чтобы выполнить несколько анализов и сохранить их отдельно, мы можем инкапсулировать состояние каждого анализа в объекте. Вот как это выглядит:

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

Далее мы трансформируем функции в методы. Например, вот в такой метод `process_word()`:

```
def process_word(self, word, order=2):
    if len(self.prefix) < order:
        self.prefix += (word,)
        return

    try:
        self.suffix_map[self.prefix].append(word)
    except KeyError:
        # если для этого префикса нет записи, создать ее
        self.suffix_map[self.prefix] = [word]

    self.prefix = shift(self.prefix, word)
```

Преобразование подобной программы — изменение структуры без изменения поведения — еще один пример рефакторинга (см. раздел «Рефакторинг» главы 4).

Этот пример предлагает способ разработки для проектирования объектов и методов.

1. Начните с разработки функций, которые считывают глобальные переменные и записывают в них значения (когда это необходимо).
2. Когда программа заработает, поищите связи между глобальными переменными и функциями, которые их используют.
3. Инкапсулируйте связанные переменные как атрибуты объекта.
4. Преобразуйте связанные функции в методы нового класса.

В качестве упражнения загрузите мой код по адресу **thinkpython2.com/code/markov.py** и выполните описанные выше шаги, чтобы инкапсулировать глобальные переменные в качестве атрибутов нового класса `Markov`.

Решение: **thinkpython2.com/code/Markov.py** (обратите внимание на прописную букву M в имени файла).

ОТЛАДКА

Наследование может затруднить отладку, потому что при вызове метода для объекта иногда трудно понять, какой метод будет вызван.

Предположим, вы пишете функцию, взаимодействующую с объектами `Hand`. Вам бы хотелось работать со всеми типами «рук», такими как `PokerHands`, `BridgeHands` и так далее. Если вы вызываете метод, подобный `shuffle()`, вы можете получить метод, определенный в классе `Deck`, но если какой-либо из подклассов переопределит этот метод, вы получите его версию вместо версии класса `Deck`. Такое поведение, как правило, полезно, но может сбивать с толку.

Каждый раз, когда вы не уверены в последовательности выполнения вашей программы, самое простое решение — добавить инструкции `print` в начале соответствующих методов. Если `Deck.shuffle` печатает сообщение, в котором говорится что-то вроде `Running Deck.shuffle`, то при запуске программы вы сможете легко отследить порядок выполнения.

В качестве альтернативы попробуйте показанную ниже функцию, которая принимает объект и имя метода (в виде строки) и возвращает класс, предоставляющий определение метода:

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

Пример:

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

Таким образом, метод `shuffle()` для данной руки — тот, что в `Deck`.

Функция `find_defining_class()` использует метод `mro()` для получения списка объектов классов (типов), в которых будет выполняться поиск методов. "MRO" означает «порядок разрешения метода (method resolution order)», который представляет собой последовательность классов, которые Python ищет для «разрешения» имени метода.

Дам совет по проектированию: когда вы переопределяете метод, интерфейс нового метода должен быть таким же, как и старый. Он должен принимать те же параметры, возвращать один и тот же тип и подчиняться тем же входным и выходным условиям. И тогда любая функция, предназначенная для работы с экземпляром родительского класса, например `Deck`, также будет работать с экземплярами дочерних классов, такими как `Hand` и `PokerHand`.

Если вы нарушите это правило, которое еще называют «принцип подстановки Лискова», ваш код рухнет, как (простите!) карточный домик.

СЛОВАРЬ ТЕРМИНОВ

Кодировать:
: Представить один набор значений, используя другой набор значений, определив сопоставление между ними.

Атрибуты класса:
: Атрибут, связанный с объектом класса. Атрибуты класса определены внутри определения класса, но вне любого метода.

Атрибут экземпляра:
: Атрибут, связанный с экземпляром класса.

Декор:
: Метод или функция, которая предоставляет другой интерфейс для другой функции без дополнительных вычислений.

Наследование:
: Возможность определить новый класс — модифицированную версию ранее определенного класса.

Родительский класс:
: Класс, от которого наследуется дочерний класс.

Дочерний класс:
: Новый класс, созданный путем наследования от существующего класса, также называется подклассом.

Отношения IS-A:
: Связь между дочерним классом и его родительским классом.

Отношения HAS-A:
: Связь между двумя классами, когда экземпляры одного класса содержат ссылки на экземпляры другого.

Зависимость:
: Связь между двумя классами, когда экземпляры одного класса используют экземпляры другого класса, но не хранят их как атрибуты.

Диаграммы классов:
: Диаграмма, которая отражает классы в программе и отношения между ними.

Множественность:
: Обозначение на диаграмме классов, которое отражает для отношения HAS-A, сколько ссылок существует на экземпляры другого класса.

Инкапсуляция данных:
: Подход к разработке программы, включающий прототип с использованием глобальных переменных и окончательную версию, которая превращает глобальные переменные в атрибуты экземпляра.

УПРАЖНЕНИЯ

Упражнение 18.1

Для показанной ниже программы нарисуйте UML-диаграмму классов, которая отражает эти классы и отношения между ними.

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong
```

```
class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

Упражнение 18.2

Напишите метод для класса `Deck`, который называется `deal_hands()` и принимает два параметра: количество игроков и количество карт для каждого. Он должен создать соответствующее количество объектов `Hand`, сдать соответствующее количество карт на одну руку и вернуть список `Hands`.

Упражнение 18.3

Ниже приведены возможные комбинации в покере в порядке увеличения значения и уменьшения вероятности:

Пара:
: Две карты с одинаковым рангом.

Две пары:
: Две пары карт с одинаковым рангом.

Тройка:
: Три карты с одинаковым рангом.

Стрит:
: Пять карт любой масти с рангами по порядку (тузы могут иметь как наивысший, так и низший ранг, поэтому Туз–2–3–4–5 — стрит, так же как и 10–Валет–Дама–Король–Туз, однако Дама–Король–Туз–2–3 не подходит).

Флеш:
: Пять карт одной масти.

Фул-хаус:
: Три карты с одним рангом, две карты с другим.

Каре:
: Четыре карты одного ранга.

Стрит-флеш:

Пять карт в последовательности (как в комбинации стрит, только одной масти).

Цель этих упражнений — оценка вероятности появления комбинаций в руке.

1. Загрузите следующие файлы по адресу **thinkpython2.com/code**:

   Card.py:

   Полная версия классов `Card`, `Deck` и `Hand` для этой главы.

   PokerHand.py:

   Неполная реализация класса, представляющего покерную комбинацию, и некоторый код, который ее тестирует.
2. Если вы запустите файл PokerHand.py, он раздаст семь 7-карточных покерных комбинаций и проверит, нет ли в них флеша. Внимательно прочитайте этот код, прежде чем продолжить.
3. Добавьте в файл PokerHand.py методы `has_pair()`, `has_twopair()` и так далее. Методы должны возвращать значение `True` или `False` в зависимости от того, соответствует ли рука (раздача) соответствующим критериям. Ваш код должен корректно работать для рук с любым количеством карт (хотя 5 и 7 — наиболее распространены).
4. Напишите метод `classify()`, который определяет комбинацию наивысшего значения для раздачи и устанавливает соответствующий атрибут `label`. Например, комбинация из 7 карт может содержать флеш и пару; она должна быть помечена как «Флеш».
5. Когда методы поиска комбинаций будут полностью работоспособны, оцените вероятность появления различных комбинаций. Напишите в файле PokerHand.py функцию, которая перетасовывает колоду карт, снова раздает ее, находит комбинации и подсчитывает, сколько раз встречается каждая из них.
6. Напечатайте таблицу комбинаций с их вероятностями. Запускайте вашу программу снова и снова, увеличивая количество раздач, пока выходные значения не приблизятся к разумной степени точности. Сравните ваши результаты со значениями по адресу **en.wikipedia.org/wiki/Hand_rankings**.

Решение: **thinkpython2.com/code/PokerHandSoln.py**.

ГЛАВА 19
СИНТАКСИЧЕСКИЙ САХАР

Одна из целей этой книги — рассказать о Python не больше, чем необходимо. Когда существовало два способа решить задачу, я выбирал один и ничего не говорил о другом. Иногда я показывал второй в качестве упражнения.

Теперь я хочу вернуться к некоторым полезным моментам. У языка Python есть ряд необязательных особенностей — можно написать хороший код и без них, — но иногда они помогают написать более лаконичный, или читаемый, или эффективный код. А в некоторых случаях получается сделать лучше все характеристики кода.

УСЛОВНЫЕ ВЫРАЖЕНИЯ

Условные инструкции встречались нам в разделе «Условное выполнение» главы 5. Их часто используют для выбора одного из двух значений. Например:

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

Эта инструкция проверяет, положительно ли значение переменной `x`. Если это так, программа запускает метод `math.log()`. Если это не так, `math.log()` вызовет ошибку `ValueError`. Чтобы избежать завершения работы программы, мы генерируем “NaN” — специальное значение с плавающей точкой, которое представляет «Не число (Not a Number)».

Мы можем сократить этот код, используя условное выражение:

```
y = math.log(x) if x > 0 else float('nan')
```

Эту строку можно прочитать так: «переменная `y` получает значение `log-x`, если `x` больше `0`; в противном случае ей присваивается значение `NaN`».

Рекурсивные функции иногда можно переписать с использованием условных выражений. Ниже показана рекурсивная версия функции `factorial()`:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Мы можем переписать ее так:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

Еще одна область для применения условных выражений — обработка необязательных аргументов. В качестве примера ниже показан метод инициализации из файла *GoodKangaroo.py* (см. упражнение 17.2):

```
def __init__(self, name, contents=None):
    self.name = name
    if contents == None:
        contents = []
    elf.pouch_contents = contents
```

Мы можем переписать его следующим образом:

```
def __init__(self, name, contents=None):
    self.name = name
    self.pouch_contents = [] if contents == None else contents
```

В целом вы можете заменить условную инструкцию условным выражением, если обе ветви содержат простые выражения, которые либо возвращаются, либо присваиваются одной и той же переменной.

ГЕНЕРАТОРЫ СПИСКОВ

В разделе «Сопоставление, фильтрация и сокращение» главы 10 вы увидели шаблоны сопоставления (маппирования) и фильтрации. Функция, показанная ниже, берет список строк, применяет строковый метод `capitalize()` ко всем элементам и возвращает новый список строк:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

Мы можем сократить ее код, используя **генератор списка (list comprehension)**:

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

Скобки указывают, что мы создаем новый список. Выражение внутри скобок определяет элементы списка, а код с `for` указывает, какую последовательность мы обходим.

Синтаксис генератора списка немного неудобен, потому что переменная `s` цикла в этом примере появляется в выражении прежде, чем мы переходим к определению.

Генераторы списков также можно использовать для фильтрации. Например, показанная ниже функция выбирает только элементы `t` в верхнем регистре и возвращает новый список:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

Попробуем переписать код, используя генератор списков:

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

Генератор списков краток и легко читается, по крайней мере для простых выражений. И как правило, он выполняется быстрее, чем эквивалент с циклом, иногда даже гораздо быстрее. Так что, если вы злитесь на меня за то, что я не упомянул их раньше, я вас пойму.

Но в свою защиту скажу, что генератор списков сложнее отладить, потому что нельзя поместить в него инструкцию `print`. Я предлагаю вам использовать генераторы списков только в случае вычислений настолько простых, что вы, вероятно, выполните их правильно с первого раза. А новичкам с генераторами списков стоит подождать.

ВЫРАЖЕНИЯ-ГЕНЕРАТОРЫ

Выражения-генераторы (generator expressions) похожи на генераторы списков, но с круглыми скобками, а не с квадратными:

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

Результатом будет объект генератора, который способен перебирать последовательность значений. Но, в отличие от генератора списка, он не вычисляет значения сразу. Он ожидает запрос. Встроенная функция `next()` получает следующее значение от генератора:

```
>>> next(g)
0
>>> next(g)
1
```

Когда вы добираетесь до конца последовательности, функция `next()` вызывает исключение `StopIteration`.

Вы также можете использовать цикл `for` для перебора значений:

```
>>> for val in g:
...     print(val)
4
9
16
```

Объект генератора отслеживает, где он находится в последовательности, поэтому цикл `for` начинает с того места, где остановился предыдущий `next`. Если генератор продолжит выполнение, он вызовет `StopException`:

```
>>> next(g)
StopIteration
```

Выражения-генераторы часто используют с такими функциями, как `sum()`, `max()` и `min()`:

```
>>> sum(x**2 for x in range(5))
30
```

ФУНКЦИИ ANY() И ALL()

Python предоставляет встроенную функцию `any()`, которая принимает последовательность логических значений и возвращает `True`, если *любое* из значений имеет значение `True`. Работает со списками:

```
>>> any([False, False, True])
True
```

Но она часто используется с выражениями-генераторами:

```
>>> any(letter == 'т' for letter in 'монти')
True
```

Этот пример не показателен, потому что он делает то же самое, что и оператор `in`. Но мы могли бы использовать функцию `any()`, чтобы переписать некоторые функции поиска из раздела «Поиск» главы 9. Например, мы могли бы написать функцию `avoids()`:

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

Функция читается так: «слово `word` не запрещается (`forbidden`), если в слове `word` нет запрещенных букв».

Использование функции `any()` с выражением-генератором эффективно, потому что оно немедленно останавливается, если находит значение `True`, то есть не нужно проверять всю последовательность.

Python предоставляет еще одну встроенную функцию — `all()`, которая возвращает `True`, если *все* элементы последовательности имеют значение `True`. В качестве упражнения используйте ее, чтобы переписать функцию `use_all()` из раздела «Поиск» главы 9.

МНОЖЕСТВА

В разделе «Вычитание словарей» главы 13 я использовал словари, чтобы найти слова, которые встречаются в документе, но не в списке слов. Функция, которую я написал, принимает аргумент `d1`, который содержит слова из документа в качестве ключей, и аргумент `d2`, который содержит список слов. Она возвращает словарь, содержащий ключи из `d1`, которых нет в `d2`:

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

Во всех этих словарях значения равны `None`, потому что они нам не нужны. В результате мы тратим часть памяти впустую.

Python предоставляет другой встроенный тип, называемый **множеством** (`set`), который ведет себя как набор ключей словаря без значений. Добавление

элементов в множество происходит быстро, и так же быстро проверяется наличие элемента. А еще множества предоставляют методы и операторы для вычисления общих операций множества.

Например, вычитание множеств доступно как метод `difference()`, или как оператор `-`. Таким образом, мы можем переписать код функции `subtract()` так:

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

Результатом будет множество, а не словарь, но в случае таких операций, как итерация, поведение абсолютно такое же.

Некоторые из упражнений в этой книге можно выполнить проще и эффективнее с помощью множеств. В качестве примера ниже показано решение задачи с функцией `has_duplicates()` из упражнения 10.7, в котором используется словарь:

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

Когда элемент встречается впервые, он добавляется в словарь. Если один и тот же элемент встречается дважды, функция возвращает `True`.

Используя наборы, мы можем изменить функцию вот так:

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

Элемент может встречаться в множестве только один раз, поэтому, если элемент в `t` появляется более одного раза, множество будет меньше `t`. Если дубликатов нет, множество будет того же размера, что и `t`.

Мы также можем использовать множества для выполнения некоторых упражнений из главы 9. В качестве примера ниже показана версия функции `uses_only()` с циклом:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

Функция `uses_only()` проверяет, все ли буквы в слове `word` входят в `available`. Мы можем переписать ее так:

```
def uses_only(word, available):
    return set(word) <= set(available)
```

Оператор `<=` проверяет, является ли один набор подмножеством другого, включая возможность того, что они равны. Последнее верно, если все буквы в слове содержатся в `available`.

В качестве упражнения перепишите функцию `avoids()`, используя множества.

СЧЕТЧИКИ

Счетчик похож на множество, за исключением того, что, если элемент появляется более одного раза, счетчик отслеживает, сколько раз он встречается. Если вы знакомы с математической идеей **мультимножества (multiset)**, функция `Counter()` — естественный способ представления мультимножества.

Функция `Counter()` определяется в стандартном модуле `collections`, поэтому его необходимо импортировать. Вы можете инициализировать функцию `Counter()` для строки, списка и других типов, поддерживающих итерацию:

```
>>> from collections import Counter
>>> count = Counter('попугай')
>>> count
Counter({'п': 2, 'о': 1, 'у': 1, 'г': 1, 'а': 1, 'й': 1})
```

Счетчики ведут себя как словари во многих отношениях; они сопоставляют каждый ключ с количеством раз, которое он встречается. Как и в словарях, ключи должны быть хешируемыми.

В отличие от словарей, счетчики не создают исключение, если вы обращаетесь к несуществующему элементу. Вместо этого они возвращают 0:

```
>>> count['е']
0
```

Мы можем переписать функцию `is_anagram()` из упражнения 10.6 с помощью счетчиков:

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

Если два слова — анаграммы, они содержат одинаковые буквы в одинаковом количестве, поэтому их счетчики эквивалентны.

Счетчики предоставляют методы и операторы для выполнения операций, подобных множеству, включая сложение, вычитание, объединение и пересечение. И они предоставляют зачастую полезный метод `most_common()`, который возвращает список пар «значение — частотность», отсортированных от наиболее распространенного к наименее:

```
>>> count = Counter('попугай')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
п 2
о 1
у 1
```

ТИП DEFAULTDICT

Модуль `collections` также предоставляет тип `defaultdict`, который похож на словарь, за исключением того, что при обращении к несуществующему ключу он генерирует новое значение на лету.

Когда вы создаете `defaultdict`, вы предоставляете функцию для создания новых значений. Функция, используемая для создания объектов, иногда называется **фабрикой (factory)**. Встроенные функции, которые создают списки, множества и другие типы, могут использоваться в качестве фабрик:

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

Обратите внимание, что аргумент — это `list` — объект класса, а не `list()` — новый список. Предоставляемая вами функция не вызывается, пока вы не обратитесь к ключу, которого не существует:

```
>>> t = d['новый ключ']
>>> t
[]
```

Новый список `t` также добавляется в словарь. Поэтому, если мы изменим `t`, изменение отразится и на `d`:

```
>>> t.append('новое значение')
>>> d
defaultdict(<class 'list'>, {'новый ключ': ['новое значение']})
```

Если вы создаете словарь списков, вы можете упростить код, используя тип `defaultdict`. В моем решении упражнения 12.2 в файле **thinkpython2.com/code/anagram_sets.py** я создаю словарь, сопоставляющий отсортированную строку букв со списком слов, которые можно составить из этих букв. Например:

`'opst'` маппирует в список `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Ниже показан исходный код:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

Код можно упростить с помощью метода `setdefault()`, который вы встречали в упражнении 11.2:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

Недостаток такого решения в том, что новый список создается каждый раз, независимо от того, требуется ли он. Это не проблема для списков, но может стать таковой, если фабричная функция сложна.

Мы можем избежать ее и упростить код, используя `defaultdict`:

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

Мое решение для упражнения 18.3, которое вы можете скачать по адресу **thinkpython2.com/code/PokerHandSoln.py**, включает метод `setdefault()`

в функции `has_straightflush()`. Недостаток этого решения — создание объекта `Hand` каждый раз в цикле, независимо от того, нужен он или нет. Перепишите этот код, используя `defaultdict`.

ИМЕНОВАННЫЕ КОРТЕЖИ

Многие простые объекты — это, в принципе, коллекции связанных значений. Например, объект `Point`, определенный в главе 15, содержит два числа: `x` и `y`. Когда определяют такой класс, обычно начинают с методов `__init__` и `__str__`:

```
class Point:

    def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

Получается приличный фрагмент кода для передачи небольшого количества информации. Python предоставляет более лаконичный способ сказать то же самое:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

Первый аргумент — это имя класса, который вы хотите создать. Второй — список атрибутов, которые объекты `Point` должны иметь, в виде строк. Возвращаемое значение `namedtuple` — объект этого класса:

```
>>> Point
<class '__main__.Point'>
```

Класс `Point` автоматически предоставляет такие методы, как `__init__` и `__str__`, поэтому вам не нужно их писать.

Чтобы создать объект `Point`, вы используете класс `Point` как функцию:

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

Метод `__init__` назначает аргументы атрибутам, используя предоставленные вами имена.

Метод `__str__` печатает представление объекта `Point` и его атрибутов.

Вы можете получить доступ к элементам именованного кортежа по имени:

```
>>> p.x, p.y
(1, 2)
```

Но вы также можете рассматривать именованный кортеж как кортеж:

```
>>> p[0], p[1]
(1, 2)
```

```
>>> x, y = p
>>> x, y
(1, 2)
```

Именованные кортежи обеспечивают быстрый способ определения простых классов. Недостаток заключается в том, что простые классы не всегда остаются таковыми. Если позже вам понадобится добавить методы в именованный кортеж, можно определить новый класс, который наследуется от именованного кортежа:

```
class Pointier(Point):
    # сюда добавляются методы
```

Или можно переключиться на обычное определение класса.

СБОР ИМЕНОВАННЫХ АРГУМЕНТОВ

В разделе «Кортежи с переменным числом аргументов» главы 12 вы увидели, как написать функцию, которая собирает свои аргументы в кортеж:

```
def printall(*args):
    print(args)
```

Вы можете вызывать эту функцию с любым количеством позиционных аргументов (то есть аргументов, у которых нет ключевых слов):

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

Но оператор * не собирает именованные аргументы:

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

Чтобы собрать именованные аргументы, подойдет оператор **:

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

Вы можете назвать параметр сбора именованных слов как хотите, но обычно используют имя `kwargs`. В результате получается словарь, который сопоставляет ключевые слова и значения:

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

Если у вас есть словарь ключевых слов и значений, вы можете использовать оператор разбивки **, чтобы вызвать функцию:

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

Без этого оператора функция будет рассматривать переменную `d` как единый позиционный аргумент, поэтому она присвоит значение `d` переменной `x` и выдаст ошибку, потому что нечего присваивать переменной `y`:

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
    File “<stdin>”, line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

При работе с функциями, имеющими большое количество параметров, в ряде случаев полезно создавать и передавать словари, в которых указаны часто используемые параметры.

СЛОВАРЬ ТЕРМИНОВ

Условные выражения:
: Выражение, которое принимает одно из двух значений, в зависимости от условия.

Генератор списков:
: Выражение с циклом `for` в квадратных скобках, которое возвращает новый список.

Выражение-генератор:
: Выражение с циклом `for` в скобках, которое возвращает объект генератора.

Мультимножество:

Математическая сущность, которая представляет собой сопоставление между элементами набора и числом их появления.

Фабрика:

Функция, обычно передаваемая в качестве параметра, используется для создания объектов.

УПРАЖНЕНИЯ

Упражнение 19.1

Ниже приведен код функции, которая вычисляет биномиальный коэффициент рекурсивно:

```
def binomial_coeff(n, k):
    """Вычисление биномиального коэффициента.

    n: количество попыток
    k: количество успешных результатов

    returns: int
    """
    if k == 0:
        return 1
    if n == 0:
        return 0

    res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
    return res
```

Перепишите тело функции, используя вложенные условные выражения.

Примечание: эта функция не очень эффективна, потому что она снова и снова вычисляет одни и те же значения. Ее можно сделать эффективнее с помощью мемоизации (см. раздел «Значения Memo» главы 11). Однако мемоизировать сложнее, если вы пишете с помощью условных выражений.

ГЛАВА 20
ОТЛАДКА

При отладке кода необходимо различать типы ошибок, чтобы быстрее их находить и исправлять.

— **Синтаксические ошибки (Syntax errors)** обнаруживаются интерпретатором, когда он переводит исходный код в байт-код. Они указывают, на проблемы со структурой программы. Пример: пропуск двоеточия в конце инструкции `def` создает несколько избыточное сообщение `SyntaxError: invalid syntax`.
— **Ошибки во время выполнения (Runtime errors)** генерируются интерпретатором, если что-то идет не так во время работы программы. Большинство сообщений об ошибках во время выполнения содержат информацию о том, где произошла ошибка и какие функции выполнялись. Пример: бесконечная рекурсия в итоге приводит к превышению максимальной глубины рекурсии (`maximum recursion depth exceeded`) во время выполнения.
— **Семантические ошибки (Semantic errors)** — это проблемы с программой, которая не выдает сообщений об ошибках, но работает неправильно. Пример: выражение может вычисляться не в ожидаемом порядке, что приводит к неверному результату.

Первый шаг в отладке — выяснить, с какой ошибкой вы имеете дело. Хотя следующие разделы организованы по типу ошибки, некоторые методы применимы в разных ситуациях.

СИНТАКСИЧЕСКИЕ ОШИБКИ

Синтаксические ошибки, как правило, легко исправить, если выяснить, где они возникают. К сожалению, сообщения об ошибках часто бесполезны. Наиболее распространены сообщения типа `SyntaxError: invalid syntax`

и `SyntaxError: invalid token`, и ни одно из них не является достаточно информативным.

Из сообщения не всегда понятно, где в программе возникла проблема. Сообщение говорит о том, где интерпретатор Python столкнулся с проблемой, но вовсе не обязательно о том, где она реально допущена. Например, после выполнения строки появляется сообщение об ошибке, но сама она находится в предыдущей строке.

Если вы разрабатываете программу поэтапно, то будет проще сообразить, где находится ошибка. Скорее всего, она в последней добавленной вами строке кода.

Если вы копируете листинги из этой книги, начните с тщательного сравнения своего кода с листингами в книге или моими файлами примеров. Проверьте каждый символ. В то же время помните, что я сам мог ошибиться, поэтому, если вы встретили синтаксическую ошибку, возможно, она есть и у меня.

Вот несколько способов избежать наиболее распространенных синтаксических ошибок.

1. Убедитесь, что вы не используете зарезервированное слово языка Python в качестве имени переменной.
2. Убедитесь, что в конце заголовка каждой составной инструкции есть двоеточие, в том числе у инструкций `while`, `if` и `def`.
3. Проверьте парность кавычек для строковых переменных (каждой открывающей соответствует одна закрывающая). Убедитесь, что все кавычки прямые (`''` или `"`), а не косые (`` `´ `` или `‟″`) и уже тем более не елочки (`«»`).
4. Если у вас есть многострочные строки в тройных кавычках (одинарных или двойных), убедитесь, что вы правильно завершили строку. Некорректно завершенная строка может вызвать ошибку `invalid token` в конце программы, или следующая часть программы может обрабатываться как строка, пока не встретится следующая. Во втором случае вообще может не быть сообщения об ошибке!
5. Незакрытый открывающий оператор — `(`, `{` или `[` — приведет к тому, что Python продолжит обрабатывать следующую строку как часть текущего оператора. Как правило, ошибка возникает почти сразу же в следующей строке.
6. Проверьте, что в условных операторах вы использовали символы `==`, а не `=`.

7. Проверьте отступы, чтобы убедиться, что код выровнен так, как это должно быть. Python может обрабатывать и пробелы, и отступы (их создают клавишей **Tab**), но если вы будете путаться с ними, возникнут проблемы. Лучший способ избежать этой проблемы — использовать текстовый редактор, который знает синтаксис Python и генерирует последовательные отступы автоматически.
8. Если в коде есть символы, не входящие в таблицу кодировки ASCII (включая строки и комментарии), могут появиться проблемы, хотя Python 3 обычно корректно обрабатывает такие символы. Будьте осторожны, если вставляете текст с веб-страницы или из другого источника.

Если ничего не сработало, переходите к следующему разделу*...

Я ПРОДОЛЖАЮ ВНОСИТЬ ИЗМЕНЕНИЯ, И ЭТО НЕ ПОМОГАЕТ

Если интерпретатор оповещает об ошибке, а вы ее не видите, причина в том, что вы и интерпретатор работаете с разным кодом. Проверьте свою среду программирования и убедитесь, что программа, которую вы редактируете, совпадает с той, которую пытается запустить Python.

Если вы не уверены, попробуйте поместить очевидную и преднамеренную синтаксическую ошибку в начало программы. Теперь запустите программу снова. Если интерпретатор не находит новую ошибку, вы запускаете не тот файл.

Есть несколько вероятных причин.

— Вы отредактировали файл и забыли сохранить изменения перед повторным запуском. Некоторые среды программирования делают это автоматически, но в большинстве сохранять изменения нужно вручную.
— Вы сохранили файл под новым именем, а в интерпретаторе все еще используете файл со старым именем.
— Неверны настройки вашей среды разработки.
— Если вы пишете модуль и используете инструкцию `import`, убедитесь, что вы не назвали свой модуль тем же именем, что и один из стандартных модулей Python.

---

* А еще убедитесь, что для имен переменных, методов, функций и прочих элементов языка Python вы не используете кириллицу и символы из других языков, отличных от латиницы. *Прим. ред.*

— Если вы изменили импортированный модуль, помните, что вам нужно перезапустить интерпретатор или использовать функцию `reload()`, чтобы интерпретатор прочитал измененный файл. Если вы повторно импортируете модуль в уже запущенном файле, ничего не произойдет.

Если вы застряли и не можете понять, что происходит, один из подходов состоит в том, чтобы начать заново с простой программы, такой как Hello, World! и убедиться, что вы можете успешно запустить заведомо исправную программу. Затем постепенно добавляйте фрагменты кода своей программы в новую.

ОШИБКИ ВО ВРЕМЯ ВЫПОЛНЕНИЯ

Если ваша программа синтаксически верна, Python может прочитать ее и по крайней мере запустить. Что еще могло пойти не так?

МОЯ ПРОГРАММА АБСОЛЮТНО НИЧЕГО НЕ ДЕЛАЕТ

Эта самая частая проблема, если программа состоит из функций и классов, но фактически не вызывает ни одной функции и не начинает выполнение. Возможно, вы так и хотели, если планируете импортировать ее как модуль только для предоставления классов и функций.

Но если это не так, убедитесь, что в программе есть вызов функции и что порядок выполнения достигает ее (см. раздел «Порядок выполнения» ниже).

ПРОГРАММА ЗАВИСАЕТ

Если программа останавливается и, кажется, ничего не делает, говорят, что она «зависла». Часто это значит, что программа застряла в бесконечном цикле или бесконечной рекурсии.

— Если вы подозреваете определенный цикл, добавьте инструкцию `print` непосредственно перед ним с текстом «вход в цикл», а другую сразу после — с текстом «выход из цикла».
— Запустите программу. Если вы получаете первое сообщение, а второе — нет, то вы попали в бесконечный цикл. Обратитесь к разделу «Бесконечный цикл» далее.

— В большинстве случаев бесконечная рекурсия заставляет программу работать некоторое время, а затем появляется сообщение «`RuntimeError: Maximum recursion depth exceeded`». Если это происходит, обратитесь к разделу «Бесконечная рекурсия» ниже.
— Если подобная ошибка не возникает, но вы подозреваете, что есть проблема с рекурсивным методом или функцией, вы все равно можете использовать методы решения проблем из раздела «Бесконечная рекурсия».
— Если ни один из этих шагов не работает, начните тестировать другие циклы и рекурсивные функции/методы.
— Если и это не сработает, то возможно, что вы не понимаете ход выполнения вашей программы. Обратитесь к разделу «Порядок выполнения» ниже.

Бесконечный цикл

Если вы подозреваете, что у вас в программе возник бесконечный цикл, и знаете, какой цикл вызывает проблему, добавьте в конец этого цикла инструкцию `print`, которая печатает значения переменных, используемых в условии, и значение условия.

Например:

```
while x > 0 and y < 0 :
    # что-то происходит с x
    # что-то происходит с y

    print('x: ', x)
    print('y: ', y)
    print("condition: ", (x > 0 and y < 0))
```

Теперь, если вы запустите программу, вы увидите три строки вывода для каждой итерации цикла. На последней итерации условие должно быть `False`. Если цикл продолжится, вы сможете увидеть значения переменных `x` и `y` и выяснить, почему они не обновляются корректно.

Бесконечная рекурсия

В большинстве случаев бесконечная рекурсия заставляет программу работать некоторое время, а затем выдает ошибку «`Maximum recursion depth exceeded`» (превышение максимальной глубины рекурсии).

Если вы подозреваете, что функция вызывает бесконечную рекурсию, убедитесь, что существует базовый случай. Должно быть какое-то условие, которое заставляет функцию возвращаться из рекурсивного вызова. Если нет, вам нужно еще раз продумать алгоритм и определить этот базовый случай.

Если есть базовый случай, но программа, кажется, не достигает его, добавьте в начало функции инструкцию `print`, которая печатает параметры. И при запуске программы вы будете видеть несколько строк вывода каждый раз, когда вызывается функция, и увидите значения параметров. Если параметры не меняются в сторону базового случая, вы получите представление о том, почему этого не происходит.

Порядок выполнения

Если вы не уверены в том, как порядок выполнения проходит через вашу программу, добавьте инструкции `print` в начало каждой функции с сообщением типа «вход в функцию foo», где `foo` — имя вашей функции.

Теперь, когда вы запустите программу, инструкция `print` выведет трассировку каждой функции при ее вызове.

КОГДА Я ЗАПУСКАЮ ПРОГРАММУ, Я ПОЛУЧАЮ ИСКЛЮЧЕНИЕ

Если что-то идет не так во время выполнения, Python печатает сообщение с именем исключения, строкой программы, в которой возникла проблема, и трассировкой.

Трассировка идентифицирует функцию, которая выполняется в данный момент, а затем функцию, которая ее вызвала, и функцию, которая ее вызвала, и так далее. Другими словами, он отслеживает последовательность вызовов функций, которая привела вас туда, где вы находитесь, включая номер строки в файле, где произошел каждый вызов.

Первый шаг — изучить место в программе, где произошла ошибка, и посмотреть, сможете ли вы выяснить, что произошло. Ниже приведены некоторые наиболее распространенные ошибки во время выполнения.

NameError:

Вы пытаетесь использовать переменную, которая не существует в текущей среде. Проверьте, правильно ли написано имя. И помните, что на локальные переменные нельзя ссылаться извне функции, в которой они определены.

TypeError:

Причин может быть несколько.

— Вы пытаетесь использовать значение ненадлежащим образом. Пример: в качестве индекса строки, списка или кортежа использован тип, отличный от целого числа.

— Существует несоответствие между элементами в строке формата и элементами, переданными для преобразования. Так бывает или когда число элементов не совпадает, или когда вызвано неверное преобразование.

— Вы передаете неправильное количество аргументов в функцию. При использовании метода посмотрите на его определение и убедитесь, что первый параметр — это `self`. Затем посмотрите на вызов метода; убедитесь, что вы вызываете метод для объекта с правильным типом и правильно предоставляете другие аргументы.

KeyError:

Вы пытаетесь получить доступ к элементу словаря с помощью ключа, которого нет в словаре. Если ключи являются строками, помните, что регистр имеет значение.

AttributeError:

Вы пытаетесь получить доступ к несуществующему атрибуту или методу. Проверьте орфографию! Вы можете использовать встроенную функцию `vars()` для перечисления существующих атрибутов.

Если ошибка `AttributeError` сообщает, что объект имеет тип `NoneType`, это означает, что его значение равно `None`. Так что проблема не в имени атрибута, а в самом объекте.

Причина отсутствия объекта может быть в том, что вы забыли вернуть значение из функции; если вы дошли до конца функции и не достигли инструкции `return`, то функция возвращает `None`. Другая распространенная причина — использование результата метода списка, такого как `sort()`, который возвращает `None`.

IndexError:

Индекс, который вы используете для доступа к списку, строке или кортежу, больше его длины минус один. Непосредственно перед местом возникновения ошибки добавьте инструкцию `print`, чтобы вывести значение индекса и длину массива. Правильный ли размер массива? Правильное ли значение индекса?

Отладчик Python (pdb) полезен для отслеживания исключений, поскольку позволяет вам проверить состояние программы непосредственно перед ошибкой. Вы можете прочитать о программе pdb по ссылке **docs.python.org/3/library/pdb.html**.

Я ДОБАВИЛ ТАК МНОГО ИНСТРУКЦИЙ ПЕЧАТИ, ЧТО ЗАВАЛЕН ВЫВОДОМ

Одна из проблем в отладке при помощи инструкций `print` заключается в том, что вы можете утонуть в выводе. Существует два способа: сократить вывод или упростить программу.

Чтобы сократить вывод, можно удалить или закомментировать бесполезные инструкции `print`, либо объединить их, или отформатировать вывод, чтобы его было легче понять.

Есть несколько способов упростить программу. Сначала сократите задачу, над которой работает программа. Например, если вы ищете список, ищите небольшой список. Если программа принимает данные от пользователя, передайте ей самый простой ввод, который вызывает проблему.

Во-вторых, очистите программу. Удалите бесполезный код и реорганизуйте программу, чтобы сделать ее максимально удобной для чтения. Например, если вы подозреваете, что проблема в глубоко вложенной части программы, попробуйте упростить структуру этой части. Если вы подозреваете в ошибках большую функцию, попробуйте разбить ее на более мелкие и протестировать их по отдельности.

Часто поиск минимального контрольного примера позволяет найти ошибку. Если окажется, что в одной ситуации программа работает, а в другой — нет, это даст вам представление о том, что происходит.

Точно так же переписывание фрагмента кода помогает найти трудноуловимые ошибки. Если вы вносите изменения, которые, по вашему мнению, не должны влиять на программу, но все-таки влияют, это может служить подсказкой.

СЕМАНТИЧЕСКИЕ ОШИБКИ

В некотором смысле семантические ошибки труднее всего отлаживать, потому что интерпретатор не предоставляет информации о том, что происходит не так. Только вы знаете, что должна делать программа.

Первый шаг — установить связь между кодом программы и поведением, которое вы видите. Вам нужно выдвинуть гипотезу о том, что на самом деле

делает программа. И это сложно — во многом потому, что компьютеры слишком быстрые!

Было бы неплохо замедлить программу до человеческой скорости, и некоторые отладчики позволяют это сделать. Но часто на то, чтобы вставить несколько инструкций `print` в подходящие места, нужно меньше времени, чем на настройку отладчика, вставку и удаление точек останова* и «пошаговое выполнение» программы до места возникновения ошибки.

МОЯ ПРОГРАММА НЕ РАБОТАЕТ

Вы должны задать себе следующие вопросы.

— Программа должна что-то сделать, но не делает? Найдите раздел кода, который отвечает за эту функциональность, и убедитесь, что он выполняется когда нужно.
— Программа делает что-то не то? Найдите в своей программе код, который отвечает за эту функциональность, и посмотрите, выполняется ли он, когда этого не следует делать.
— Выполнение фрагмента кода приводит к неправильному результату? Убедитесь, что вы понимаете рассматриваемый код, особенно если он включает функции или методы из других модулей Python. Прочитайте документацию по функциям, которые вызываете. Протестируйте их, написав простые контрольные тесты и проверив результаты.

Чтобы программировать, нужно понимать, как работают программы. Если вы пишете программу, которая не выполняет то, что вы ожидаете, зачастую проблема не в программе — она в вашей ментальной модели.

Лучший способ исправить вашу ментальную модель — разбить программу на компоненты (обычно это функции и методы) и протестировать каждый компонент по отдельности. Как только вы обнаружите несоответствие между вашей моделью и реальностью, вы сможете решить проблему.

Разумеется, нужно создавать и тестировать компоненты программы по мере разработки. И если вы столкнетесь с проблемой, то у вас будет лишь небольшой фрагмент нового кода, в котором могла возникнуть ошибка.

---

* Точка останова (англ. breakpoint) — маркер, на котором отладчик останавливает выполнение программы. *Прим. ред.*

МОЕ ГРОМОЗДКОЕ СЛОЖНОЕ ВЫРАЖЕНИЕ ДЕЛАЕТ НЕ ТО, ЧТО Я ХОЧУ

Если вы умеете писать читабельные сложные выражения — это хорошо, но их трудно отладить — а это уже плохо. Будет полезно разбить сложное выражение на группу операций присваивания временным переменным.

Например, код

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

можно переписать следующим образом:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

Вторая версия легче читается, ведь имена переменных дают дополнительные сведения о программе и ее легче отлаживать, потому что можно проверять типы промежуточных переменных и выводить их значения.

Другая проблема со сложными выражениями заключается в том, что порядок вычислений может отличаться от ожидаемого. Например, если вы переводите выражение $\frac{x}{2\pi}$ в Python, вы можете написать:

```
y = x/2 * math.pi
```

Это неправильно, потому что умножение и деление имеют одинаковый приоритет и вычисляются слева направо. Так что это выражение вычисляется как $\frac{x}{2} \times \pi$.

Хороший способ отладки выражений — добавить круглые скобки, чтобы сделать порядок вычислений явным:

```
y = x/(2 * math.pi)
```

Если вы не уверены в порядке действий, используйте скобки. Мало того что программа будет правильной (в смысле выполнения того, что вы хотели), она также будет более наглядной для других людей, которые не помнят приоритет операций.

У МЕНЯ ЕСТЬ ФУНКЦИЯ, КОТОРАЯ ВОЗВРАЩАЕТ НЕ ТО, ЧТО Я ОЖИДАЮ

Если вы работаете с инструкцией `return` со сложным выражением, у вас нет возможности напечатать результат перед его выполнением. Опять же, попробуйте использовать временную переменную. Например, вместо:

```
return self.hands[i].removeMatches()
```

вы могли бы написать:

```
count = self.hands[i].removeMatches()
return count
```

Теперь у вас есть возможность отобразить значение переменной `count` перед возвратом.

Я ДЕЙСТВИТЕЛЬНО В ТУПИКЕ, И МНЕ НУЖНА ПОМОЩЬ

Сначала попробуйте отойти от компьютера на несколько минут. Компьютеры излучают волны и воздействуют на мозг, вызывая следующие симптомы:

— подавленность и гнев;
— суеверные убеждения («компьютер ненавидит меня») и магическое мышление («программа работает только тогда, когда я надеваю шляпу задом наперед»);
— программирование случайным блужданием (попытка написать все возможные решения и выбрать то, которое работает).

Если вы обнаружите у себя один или несколько симптомов, встаньте из-за компьютера и сходите на прогулку. Когда вы успокоитесь, подумайте о программе. Что она делает? Каковы возможные причины такого поведения? Когда в последний раз у вас была рабочая программа и что вы сделали потом?

Иногда нужно время, чтобы найти баг. Я часто понимаю, где была ошибка, когда нахожусь вдали от компьютера и позволяю себе отвлечься. Хорошо обдумывать баги в транспорте, ду́ше и в постели перед тем, как заснуть.

НЕТ, МНЕ РЕАЛЬНО НУЖНА ПОМОЩЬ

Так бывает. Даже лучшие программисты, бывает, заходят в тупик. Иногда вы работаете над программой так долго, что не видите ошибки. Тогда вам поможет свежий взгляд.

Прежде чем позвать кого-то на помощь, убедитесь, что вы готовы к этому. Ваша программа должна быть максимально простой, и вы должны работать с минимальным количеством входных данных, которые вызывают ошибку. Вы должны расставить инструкции `print` в соответствующих местах

программы (и вывод, который они производят, должен быть понятным). Вы должны понимать проблему достаточно хорошо, чтобы описать ее кратко.

Когда вы приглашаете кого-то помочь, обязательно предоставьте необходимую информацию.

— Если появляется сообщение об ошибке, какое оно и на какую часть программы оно указывает?
— Что вы сделали прямо перед тем, как произошла эта ошибка? Какими были последние строки кода, которые вы написали, или какой новый контрольный тест провален?
— Как вы уже пробовали решить проблему и что вы выяснили?

Когда найдете ошибку, подумайте, как можно было ускорить ее поиск. Тогда в следующий раз, когда увидите нечто подобное, вы сможете справиться с ним быстрее.

Помните, что цель не просто в том, чтобы заставить программу работать. Важно еще понимать, как именно это делать.

ГЛАВА 21
АНАЛИЗ АЛГОРИТМОВ

Это приложение — краткая выжимка из книги Think Complexity Аллена Б. Дауни, опубликованной издательством O'Reilly Media. Вы можете взяться за нее, когда закончите читать текущую книгу.

Анализ алгоритмов (Analysis of algorithms) — это отрасль компьютерных наук, изучающих производительность алгоритмов, особенно их требования к времени выполнения и памяти. См. **en.wikipedia.org/wiki/Analysis_of_algorithms**.

Практическая цель анализа алгоритмов состоит в том, чтобы предсказать производительность различных алгоритмов и эффективно управлять проектными решениями.

Во время президентской кампании 2008 года в США кандидата Барака Обаму попросили провести импровизированный анализ во время визита в офис Google. Генеральный директор Эрик Шмидт в шутку спросил его о «наиболее эффективном способе сортировки миллиона 32-разрядных целых чисел». Обама, очевидно, был предупрежден, потому что быстро ответил: «Я думаю, что пузырьковая сортировка была бы неправильным решением». См. **bit.ly/1MpIwTf**.

Это правда: пузырьковая сортировка концептуально проста, но работает медленно для больших наборов данных. Ответ, который Шмидт, вероятно, искал, — это «поразрядная сортировка» (**ru.wikipedia.org/wiki/Поразрядная_сортировка**)*.

---

* Но если вам задают такой вопрос на собеседовании, я думаю, что лучший ответ: «Самый быстрый способ сортировки миллиона целых чисел — взять любую функцию сортировки, предоставляемую языком, который я использую. Его производительность достаточно хороша для подавляющего большинства приложений, но если бы оказалось, что мое приложение слишком медленное, я бы использовал профилировщик, чтобы увидеть, на что тратится время. Если бы казалось, что более быстрый алгоритм сортировки окажет значительное влияние на производительность, то я бы искал хорошую реализацию поразрядной сортировки». *Прим. авт.*

Цель анализа алгоритмов — проведение значимых сравнений между алгоритмами, но есть некоторые проблемы.

— Относительная производительность алгоритмов может зависеть от характеристик аппаратного обеспечения, поэтому один алгоритм может работать быстрее на компьютере А, другой — на компьютере Б. Общее решение этой проблемы — определить **машинную модель (machine model)** и проанализировать количество шагов или операций, которые требуются алгоритму при реализации данной модели.
— Относительная производительность может зависеть от конкретного набора данных. Например, некоторые алгоритмы сортировки работают быстрее, если данные уже частично отсортированы, а другие, наоборот, — медленнее. Распространенный способ избежать этой проблемы — анализ **наихудшего случая (worst case)**. Иногда полезно проанализировать производительность среднего случая, но это обычно сложнее, и не всегда очевидно, какой набор случаев является средним.
— Относительная производительность также зависит от масштаба задачи. Алгоритм сортировки, быстрый для небольших списков, может быть медленным для длинных списков. Обычное решение этой проблемы состоит в выражении времени выполнения (или количества операций) как функции размера задачи и группировке функций по категориям в зависимости от того, насколько быстро они растут с увеличением масштаба задачи.

В таком сравнении хорошо то, что оно приводит к простой классификации алгоритмов. Например, если я знаю, что время выполнения алгоритма А имеет тенденцию быть пропорциональным размеру входных данных, а алгоритм Б имеет тенденцию быть пропорциональным n^2, то я ожидаю, что А будет быстрее Б, по крайней мере для больших значений n.

Этот вид анализа сопровождается некоторыми оговорками, но мы вернемся к ним позже.

ПОРЯДОК РОСТА

Предположим, вы проанализировали два алгоритма и выразили их время выполнения в терминах размера входных данных: алгоритм А предпринимает $100n + 1$ шагов для решения проблемы с размером n; алгоритм Б состоит из $n^2 + n + 1$ шагов.

В следующей таблице показано время выполнения этих алгоритмов для задач разных размеров:

Размер ввода	Время выполнения алгоритма А	Время выполнения алгоритма Б
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

При $n = 10$ алгоритм А выглядит довольно плохо; он работает почти в 10 раз дольше алгоритма Б. Но для $n = 100$ время выполнения примерно одинаково, а для больших значений алгоритм А значительно эффективнее.

Основная причина заключается в том, что при больших значениях n любая функция, содержащая член n^2, будет расти быстрее, чем функция, чей ведущий член равен n. **Ведущий член (leading term)** — член с наибольшим показателем степени.

Для алгоритма А ведущий член имеет большой коэффициент, 100, поэтому Б работает лучше А при малых n. Но независимо от коэффициентов, всегда будут некоторые значения n, где $an^2 > bn$, для любых значений a и b.

Тот же аргумент применим к неведущим членам. Даже если бы время выполнения алгоритма А было n +1000000, оно все равно было бы лучше, чем алгоритм Б для достаточно большого n.

В целом ожидается, что алгоритм с меньшим ведущим членом будет лучшим алгоритмом для больших задач, но для небольших задач может существовать **точка пересечения (crossover point)**, где другой алгоритм окажется лучше. Расположение точки пересечения зависит от деталей реализации алгоритмов, входных данных и аппаратного обеспечения, поэтому ее обычно игнорируют в целях алгоритмического анализа. Но это не значит, что вы можете забыть о ней.

Если два алгоритма имеют один и тот же порядок ведущего члена, трудно сказать, какой из них лучше; опять же, ответ зависит от деталей. Таким образом, с точки зрения алгоритмического анализа функции с одним и тем же порядком ведущего члена считаются эквивалентными, даже если они имеют разные коэффициенты.

Порядок роста (order of growths) — это набор функций, асимптотический прирост которых считается эквивалентным. Например, $2n$, $100n$ и $n + 1$

принадлежат к одному и тому же порядку роста, который записывается как O(*n*) в **обозначениях большого «О» (Big-Oh notation)** и часто называется **линейным (linear)**, потому что каждая функция в этом наборе линейно зависит от *n*.

Все функции с ведущим членом n^2 имеют производительность $O(n^2)$; они называются **квадратичными (quadratic)**.

В следующей таблице показаны некоторые из порядков роста, которые чаще всего встречаются в алгоритмическом анализе, в порядке возрастания сложности.

Порядок роста	Название
$O(1)$	константный
$O(\log b^n)$	логарифмический (для любого b)
$O(n)$	линейный
$O(n \log b^n)$	линейно-логарифмический
$O(n^2)$	квадратичный
$O(n^3)$	кубический
$O(c^n)$	экспоненциальный (для любого c)

Для логарифмических членов основание логарифма не имеет значения; изменение основания эквивалентно умножению на константу, которая не меняет порядок роста. Точно так же все экспоненциальные функции принадлежат к одному и тому же порядку роста независимо от основания показателя. Экспоненциальные функции возрастают очень быстро, поэтому экспоненциальные алгоритмы могут быть полезны только для небольших задач.

Упражнение 21.1

Прочитайте страницу «Википедии» о нотации большого «О» по адресу **ru.wikipedia.org/wiki/«O»_большое_и_«o»_малое** и ответьте на следующие вопросы:

1. Каков порядок роста $n^3 + n^2$? А как насчет 1000000 n^3+n^2? А насчет $n^3+1000000n^2$?
2. Каков порядок роста $(n^2+n)*(n+1)$? Прежде чем начать умножение, помните, что вам нужен только ведущий член.
3. Если *f* принадлежит O(*g*), для некоторой неопределенной функции *g*, что мы можем сказать об $af + b$?

4. Если f_1 и f_2 принадлежат O(g), что мы можем сказать об $f_1 + f_2$?
5. Если f_1 принадлежит O(g) и f_2 принадлежит O(h), что мы можем сказать об $f_1 + f_2$?
6. Если f_1 принадлежит O(g) и f_2 принадлежит O(h), что мы можем сказать об $f_1 * f_2$?

Программисты, которым важна производительность, часто испытывают трудности с анализом такого рода. И на то есть причина: иногда коэффициенты и неведущие члены могут влиять на результат. Иногда детали аппаратного обеспечения, язык программирования и характеристики входных данных имеют большое значение. А для небольших задач асимптотическое поведение не имеет значения.

Но если учитывать эти предостережения, алгоритмический анализ — полезный инструмент. По крайней мере для больших задач «лучшие» алгоритмы обычно лучше, а иногда — и значительно лучше. Разница между двумя алгоритмами с одинаковым порядком роста обычно является постоянным фактором, но разница между хорошим алгоритмом и плохим алгоритмом не ограничена!

АНАЛИЗ ОСНОВНЫХ ОПЕРАЦИЙ PYTHON

В Python большинство арифметических операций занимают постоянное время. Умножение обычно требует больше времени, чем сложение и вычитание, а деление занимает еще больше времени, но время выполнения не зависит от величины операндов. Очень большие целые числа являются исключением; в этом случае время выполнения увеличивается с количеством цифр.

Операции с использованием индексов — чтение или запись элементов в последовательности или словаре — также занимают постоянное время независимо от размера структуры данных.

Цикл `for`, который обходит последовательность или словарь, обычно линейный, если все операции в теле цикла выполняются за постоянное время. Например, сложение элементов списка линейно:

```
total = 0
for x in t:
    total += x
```

Встроенная функция `sum()` также линейна, потому что делает то же самое, но зачастую она быстрее, потому что это более эффективная

реализация; на языке алгоритмического анализа она имеет меньший ведущий коэффициент.

Как правило, если тело цикла находится в $O(n^a)$, то весь цикл находится в $O(n^{a+1})$. Исключение составляют случаи, когда вы можете доказать, что цикл завершается после постоянного числа итераций. Если цикл выполняется k раз независимо от n, то цикл находится в $O(na)$, даже для больших k.

Умножение на k не меняет порядок роста, так же как и деление. Таким образом, если тело цикла находится в $O(n^a)$ и выполняется n/k раз, цикл находится в $O(n^{a+1})$ даже для больших k.

Большинство операций со строками и кортежами линейны, за исключением работы с индексами и функции `len()`, которые постоянны по времени. Встроенные функции `min()` и `max()` линейны. Время выполнения операции среза пропорционально длине вывода, но не зависит от размера ввода.

Конкатенация строк линейна, время выполнения зависит от суммы длин операндов.

Все строковые методы линейны, но, если длины строк ограничены константой — например, операциями над одиночными символами, — они считаются постоянными по времени. Строковый метод `join()` линейный; время выполнения зависит от общей длины строк.

Большинство методов списка линейны, но есть некоторые исключения.

- Добавление элемента в конец списка занимает в среднем постоянное время; когда списку не хватает места, он иногда перемещается в другое место, но общее время для n операций составляет $O(n)$, поэтому среднее время для каждой операции составляет $O(1)$.
- Удаление элемента из конца списка всегда занимает постоянное время.
- Сортировка — это $O(n \log n)$.

Большинство операций и методов словаря занимают постоянное время, но есть некоторые исключения.

- Время выполнения обновления пропорционально размеру словаря, передаваемого в качестве параметра, а не словаря, который обновляется.
- Функции `keys()`, `values()` и `items()` занимают постоянное время, потому что они возвращают итераторы. Но если вы перебираете итераторы, цикл будет линейным.

Производительность словарей — одно из маленьких чудес информатики. Мы изучим, как они работают, в разделе «Хеш-таблицы» далее в этой главе.

Упражнение 21.2
Прочтите страницу «Википедии» об алгоритмах сортировки по адресу **ru.wikipedia.org/wiki/Алгоритм_сортировки** и ответьте на следующие вопросы.

1. Что такое сортировка сравнением? Каков наилучший наихудший случай порядка роста для сортировки сравнением? Каков наилучший наихудший случай порядка роста для любого алгоритма сортировки?
2. Каков порядок роста сортировки пузырьком, и почему Барак Обама считает, что это «неправильный путь»?
3. Каков порядок роста поразрядной сортировки? Какие предварительные условия мы должны выполнить, чтобы ее использовать?
4. Что такое устойчивая сортировка и почему она может иметь значение на практике?
5. Какой алгоритм сортировки наихудший?
6. Какой алгоритм сортировки использует библиотека языка C? Какой алгоритм сортировки использует Python? Эти алгоритмы стабильны?
7. Многие из алгоритмов, основанных не на сравнении, линейны, так почему же Python использует сравнительную сортировку O(n log n)?

Возможно, вам придется поискать ответы в Google.

АНАЛИЗ АЛГОРИТМОВ ПОИСКА

Поиск (search) — это алгоритм, который берет набор значений и целевой элемент и определяет, находится ли цель в наборе, часто возвращая индекс цели.

Простейшим алгоритмом поиска можно назвать линейный поиск, который проходит элементы коллекции по порядку, останавливаясь, если находит цель. В худшем случае он должен пройти через всю коллекцию, поэтому время выполнения линейно.

Оператор `in` для последовательностей использует линейный поиск, как и строковые методы, такие как `find()` и `count()`.

Если элементы последовательности расположены по порядку, вы можете использовать **поиск делением пополам (bisection search)**, который имеет сложность O(log n). Поиск делением пополам похож на алгоритм, который вы можете использовать для поиска слова в словаре (я имею в виду бумажный словарь, а не структуру данных). Вместо того чтобы начинать с начала

и проверять каждый элемент по порядку, вы начинаете с элемента в центре и проверяете, находится ли искомое слово до или после. Если оно встречается раньше, то вы ищете в первой половине последовательности. В противном случае вы ищете во второй половине. В любом случае вы сокращаете количество оставшихся элементов вдвое.

Если последовательность содержит 1 млн элементов, чтобы найти слово или сделать вывод, что его там нет, потребуется всего 20 шагов. Это примерно в 50 тысяч раз быстрее, чем линейный поиск.

Поиск делением пополам намного быстрее, чем линейный поиск, но он требует упорядоченной последовательности, а она, в свою очередь, может потребовать дополнительной работы.

Существует еще одна структура данных, называемая **хеш-таблицей (hashtable)**, которая работает еще быстрее: она может выполнять поиск за постоянное время — и не требует сортировки элементов. Словари Python реализованы с использованием хеш-таблиц, поэтому большинство операций со словарем, включая оператор `in`, занимают постоянное время.

ХЕШ-ТАБЛИЦЫ

Чтобы объяснить, как работают хеш-таблицы и почему они так хороши, я начну с простой реализации сопоставления (map) и постепенно буду ее улучшать, пока не приду к хеш-таблице.

Я использую Python для демонстрации, но в реальной жизни вы не писали бы этот код на Python, вы бы просто использовали словарь! Таким образом, представьте в этой главе, что словарей не существует и что вы хотите реализовать структуру данных, которая сопоставляет ключи и значения. Операции, которые вы должны реализовать:

`add(k, v)`:

Добавляет новый элемент, который сопоставляет ключ `k` и значение `v`. В Python для словаря `d` эта операция записывается как `d[k] = v`.

`get(k)`:

Найти и вернуть значение, соответствующее ключу `k`. В Python для того же словаря `d` эта операция записывается как `d[k]` или `d.get(k)`.

Сейчас я предполагаю, что каждый ключ встречается только один раз. Простейшая реализация этого интерфейса использует список кортежей, где каждый кортеж представлен парой «ключ — значение»:

```
class LinearMap:

    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
    raise KeyError
```

Метод `add()` добавляет кортеж «ключ — значение» в список элементов, это постоянная по времени операция.

Метод `get()` использует цикл `for` для поиска в списке: если он находит целевой ключ, то возвращает соответствующее значение; в противном случае вызывает исключение `KeyError`. Эта операция линейна.

Альтернатива — хранение отсортированного по ключам списка. Тогда метод `get()` может использовать поиск делением пополам, с производительностью O(log *n*). Но вставка нового элемента в середину списка по сложности линейна, так что это не лучший вариант. Существуют и другие структуры данных, которые могут реализовывать методы `add()` и `get()` за логарифмическое время, но это все же не так хорошо, как постоянное время, поэтому давайте двигаться дальше.

Один из способов улучшить класс `LinearMap` — разбить список пар «ключ — значение» на меньшие списки. Ниже показана реализация класса `BetterMap`, представляющая собой список из 100 `LinearMaps`. Как мы увидим через секунду, порядок роста для метода `get()` все еще линейный, но класс `BetterMap` — это шаг на пути к хеш-таблицам:

```
class BetterMap:

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)
```

```
    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

Метод `__init__` составляет список из *n* `LinearMaps`.

Метод `find_map()` используется методами `add()` и `get()`, чтобы выяснить, куда поместить новый элемент или где искать.

Также метод `find_map()` использует встроенную функцию `hash()`, которая принимает практически любой объект Python и возвращает целое число. У этой реализации есть ограничение: она работает только с хешируемыми ключами. Изменяемые типы, такие как списки и словари, нельзя хешировать.

Хешируемые объекты, которые считаются эквивалентными, возвращают одно и то же хеш-значение, но это не всегда работает в обратную сторону: два объекта с разными значениями могут возвращать одно и то же хеш-значение.

Еще метод `find_map()` использует оператор деления по модулю для переноса значений хеша в диапазон от `0` до `len(self.maps)`, поэтому результатом является допустимый индекс в списке. Конечно, это означает, что множество различных значений хеш-функции будут перенесены в один и тот же индекс. Но если хеш-функция распределяет значения довольно равномерно (именно для этого и предназначены хеш-функции), то мы можем ожидать, что в `LinearMap` будет *n*/100 элементов.

Поскольку время выполнения метода `LinearMap.get` пропорционально количеству элементов, мы ожидаем, что `BetterMap` будет примерно в 100 раз быстрее, чем `LinearMap`. Порядок роста все еще линейный, но ведущий коэффициент меньше. Это хорошо, но все же не так хорошо, как хеш-таблица.

Вот (наконец) ключевая идея, которая делает хеш-таблицы такими быстрыми: если вы можете ограничить максимальную длину `LinearMap`, `LinearMap.get` будет работать за постоянное время. Все, что вам нужно, — это отслеживать количество элементов, и, когда количество элементов в `LinearMap` превышает пороговое значение, измените размер хеш-таблицы, добавив дополнительные `LinearMap`.

Ниже показана реализация хеш-таблицы:

```
class HashMap:

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)
```

```
def add(self, k, v):
    if self.num == len(self.maps.maps):
        self.resize()

    self.maps.add(k, v)
    self.num += 1

def resize(self):
    new_maps = BetterMap(self.num * 2)
    for m in self.maps.maps:
        for k, v in m.items:
            new_maps.add(k, v)
    self.maps = new_maps
```

Каждый `HashMap` содержит `BetterMap`; метод `__init__` начинается всего с двух `LinearMap` и инициализирует переменную `num`, которая отслеживает количество элементов.

Метод `get()` просто перенаправляет в `BetterMap`. Реальная работа происходит в методе `add()`, который проверяет количество элементов и размер `BetterMap`: если они равны, среднее количество элементов в `LinearMap` равно 1, поэтому он вызывает метод `resize()`.

Тот создает новый экземпляр `BetterMap`, вдвое больший по сравнению с предыдущим, а затем перехеширует элементы со старого сопоставления в новое.

Перехеширование необходимо, потому что изменение числа `LinearMap` меняет знаменатель оператора модуля в методе `find_map()`. Это означает, что некоторые объекты, которые раньше хешировали в один и тот же `LinearMap`, будут разделены (чего мы и хотели, верно?).

Перехеширование — линейная операция, поэтому изменение размера также линейно, и это выглядит не так уж здорово, ведь я обещал, что операция добавления будет происходить за константное (постоянное) время. Но помните, что нам не нужно каждый раз менять размер, поэтому добавление обычно происходит за постоянное время и только иногда за линейное. Общий объем работ при вызове метода `add()` n раз пропорционален n, поэтому среднее время каждого добавления постоянно!

Чтобы увидеть, как это работает, представьте, что начинаете с пустой таблицы `HashTable` и постепенно добавляете последовательность элементов. Мы начинаем с двух `LinearMap`, поэтому первые два добавления выполняются быстро (изменение размера не требуется). Допустим, каждый из них занимает одну единицу работы. Следующее добавление требует изменения размера, поэтому мы должны перехешировать первые два элемента (скажем, еще

две единицы работы), а затем добавить третий элемент (еще одна единица). Добавление следующего элемента стоит одну единицу, поэтому на данный момент — всего четыре единицы работы.

Следующее дополнение стоит пять единиц, но следующие три — по одной, таким образом, общее количество составляет 14 единиц для первых восьми добавлений.

Следующее добавление стоит уже девять единиц, но затем мы можем добавить еще семь до следующего изменения размера, так что общее количество составляет 30 единиц для первых 16 добавлений.

После 32 добавлений общая стоимость составляет 62 единицы, и я надеюсь, что вы начинаете видеть закономерность. После n добавлений, где n — степень двойки, общая стоимость составляет $2n - 2$ единицы, таким образом, средняя работа на добавление составляет немного меньше, чем 2 единицы. Если n является степенью двойки, это лучший случай. Для других значений n средняя работа немного выше, но это не важно. Важно то, что это $O(1)$.

На рис. 21.1 графически показано, как это работает. Каждый блок представляет собой единицу работы. Столбцы показывают общую работу для каждого добавления в порядке слева направо: первые два добавления стоят одну единицу, третье стоит три единицы и так далее.

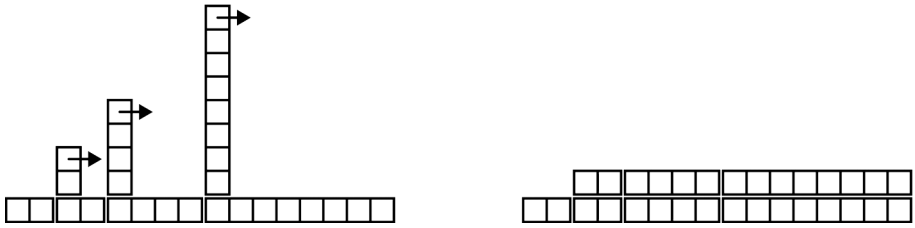


Рис. 21.1. Стоимость добавления в хеш-таблицу

Дополнительная работа на перехеширование выглядит как последовательность все более высоких башен с увеличивающимся пространством между ними. Теперь, если вы опрокинете башни, распределяя стоимость изменения размера по всем добавлениям, вы можете увидеть, что общая стоимость после n добавлений составляет $2n - 2$.

Важная особенность этого алгоритма в том, что, когда мы изменяем размер `HashTable`, он растет в геометрической прогрессии; то есть мы умножаем размер на константу. Если вы будете увеличивать размер арифметически, добавляя фиксированное число каждый раз, среднее время каждого выполнения метода `add()` будет линейным.

Вы можете скачать мою реализацию `HashMap` по адресу **thinkpython2.com/code/Map.py**, но помните, что нет причин применять ее; если вам нужно сопоставление, просто используйте словарь Python.

СЛОВАРЬ ТЕРМИНОВ

Анализ алгоритмов:

Способ сравнения алгоритмов с точки зрения их требований по времени выполнения и/или памяти.

Машинная модель:

Упрощенное представление компьютера, используемого для описания алгоритмов.

Наихудший вариант:

Ввод, который заставляет данный алгоритм работать медленнее всего (или требует больше всего памяти).

Ведущий член:

В многочлене — член с наибольшим показателем степени.

Точка пересечения:

Размер задачи, когда два алгоритма требуют одинакового времени выполнения или одинаковое количество памяти.

Порядок роста:

Набор функций, которые возрастают так, что считаются эквивалентными с точки зрения анализа алгоритмов. Например, все функции, которые возрастают линейно, принадлежат к одному и тому же порядку роста.

Нотация большого «О»:

Нотация для представления порядка роста; например, O(n) представляет набор функций, которые возрастают линейно.

Линейный:

Алгоритм, время выполнения которого пропорционально размеру задачи, по крайней мере для задач большого размера.

Квадратичный:

Алгоритм, время выполнения которого пропорционально n^2, где n — это мера размера задачи.

Поиск:

Задача нахождения элемента последовательности (например, списка или словаря) или определения его отсутствия.

Хеш-таблица:

Структура данных, которая представляет собой набор пар «ключ — значение» и выполняет поиск за постоянное время.

ОБ АВТОРЕ

Аллен Дауни — профессор компьютерных наук в Инженерно-техническом колледже имени Франклина У. Олина. Он преподавал в колледже Уэллсли, колледже Колби и Калифорнийском университете в Беркли. Он имеет докторскую степень в области компьютерных наук, которую получил в Калифорнийском университете в Беркли, а также степень магистра и бакалавра в Массачусетском технологическом институте.

ИЗОБРАЖЕНИЕ НА ОБЛОЖКЕ

Животное на обложке этой книги — каролинский попугай, также известный как *Conuropsis carolinensis*. Этот попугай обитал на юго-востоке Соединенных Штатов и был единственным попугаем на континенте к северу от Мексики. Когда-то этот вид был распространен далеко на север, до Нью-Йорка и Великих озер, хотя чаще встречался на территориях от Флориды до Каролины.

Каролинский попугай был почти полностью зеленым с желтой головой и оранжевыми перышками, которые появлялись на лбу и щеках в зрелом возрасте. Средний размер птицы — 31–33 см. У каролинского попугая был громкий, буйный голос, и он постоянно болтал во время кормежки. Обитал в дуплах возле болот и берегов рек. Каролинский попугай был очень общительным животным, особи формировали небольшие стаи, численность которых могла вырастать до нескольких сотен попугаев при наличии пропитания.

К сожалению, они часто кормились зерном на полях, и фермеры отстреливали птиц, чтобы уберечь урожай. Попугаи летели на помощь раненым особям, поэтому фермерам удавалось уничтожать целые стаи. Кроме того, их перья использовались для украшения женских шляп, а некоторых попугаев продавали в зоомагазинах. Сочетание этих факторов привело к тому, что к концу 1800-х годов каролинский попугай стал редким видом, и, вероятно, передавшиеся от домашней птицы заболевания способствовали сокращению популяции. В 1939 году вид был провозглашен вымершим.

Сегодня в музеях мира выставлено более 700 чучел каролинских попугаев.

Многие из животных на обложках книг издательства O'Reilly находятся под угрозой исчезновения; все они важны для планеты.

Чтобы узнать больше о том, как вы можете помочь в предотвращении вымирания видов, посетите сайт **animals.oreilly.com**.

Изображение на обложке взято из каталога Johnson's Natural History.

ПРОДАЖИ

МЕНЕДЖМЕНТ

ИСТОРИИ УСПЕХА

УПРАВЛЕНИЕ ПРОЕКТАМИ

ПЕРЕГОВОРЫ

HR

Научно-популярное издание

Дауни Аллен

Основы Python

Научитесь думать как программист

Шеф-редактор *Ренат Шагабутдинов*
Ответственный редактор *Ольга Копыт*
Арт-директор *Алексей Богомолов*
Верстка обложки *Наталия Майкова*
Верстка *Вячеслав Лукьяненко*
Корректоры *Лев Зелексон, Олег Пономарев*

Изготовитель: ООО «Манн, Иванов и Фербер»
123104, Россия, г. Москва, Б. Козихинский пер.,
д. 7, стр. 2

mann-ivanov-ferber.ru
facebook.com/mifbooks
vk.com/mifbooks
instagram.com/mifbooks

O'REILLY®

Если вы хотите научиться программировать, стоит начать с языка Python. Сегодня это едва ли не самый популярный язык программирования, а освоить его сравнительно просто.

Практическое руководство у вас в руках последовательно раскрывает все важные темы от простых к более сложным. Сочетание теории, примеров и заданий поможет разобраться в программировании на Python. Для их выполнения вам не потребуются специальные навыки или серьезные математические знания. Книга подходит новичкам и тем, кто уже владеет языком Python, но хотел бы структурировать свои знания или иметь под рукой справочник.

Но что еще важнее, эта книга меняет мышление. Вы не только приобретете сугубо технические навыки и знания, но и научитесь смотреть на многие проблемы взглядом разработчика.

***

Аллен Дауни — профессор компьютерных наук, преподаватель, автор почти полутора десятков книг по программированию. Имеет степень Университета Беркли и МИТ.

ISBN 978-5-00146-798-4

9 785001 467984 >

Максимально полезные книги на сайте **mann-ivanov-ferber.ru**

издательство
МАНН, ИВАНОВ И ФЕРБЕР

Like facebook.com/**mifbooks**

vk.com/**mifbooks**

instagram.com/**mifbooks**