3-Е ИЗДАНИЕ

ИЗУЧАЕМ PYTHON

ПРОГРАММИРОВАНИЕ ИГР,
ВИЗУАЛИЗАЦИЯ ДАННЫХ, ВЕБ-ПРИЛОЖЕНИЯ

ЭРИК МЭТИЗ

ПРОДАНО
СВЫШЕ 500 000
ЭКЗЕМПЛЯРОВ!

no starch
press

by Eric Matthes

PYTHON CRASH COURSE

2ND EDITION

A Hands-On, Project-Based Introduction to Programming

no starch press
San Francisco

ЭРИК МЭТИЗ

ИЗУЧАЕМ PYTHON

ПРОГРАММИРОВАНИЕ ИГР, ВИЗУАЛИЗАЦИЯ ДАННЫХ, ВЕБ-ПРИЛОЖЕНИЯ

3-Е ИЗДАНИЕ

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск
2021

ББК 32.973.2-018.1
УДК 004.43
М97

Мэтиз Эрик

М97 Изучаем Python: программирование игр, визуализация данных, веб-приложения. 3-е изд. — СПб.: Питер, 2021. — 512 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1528-0

«Изучаем Python» — это самое популярное в мире руководство по языку Python. Вы сможете максимально быстро освоить Python, научитесь писать программы, устранять ошибки и создавать работающие приложения.

В первой части книги вы познакомитесь с основными концепциями программирования, такими как переменные, списки, классы и циклы, а простые упражнения познакомят вас с шаблонами чистого кода. Вы узнаете, как делать программы интерактивными и как протестировать код, прежде чем добавлять в проект. Во второй части вы примените новые знания на практике и создадите три проекта: аркадную игру в стиле Space Invaders, визуализацию данных с удобными библиотеками Python и простое веб-приложение, которое можно быстро развернуть онлайн.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1593279288 англ.

ISBN 978-5-4461-1528-0

Оглавление

Отзывы о книге

Интересно наблюдать за тем, как No Starch Press создает будущую классику, которая по праву может занять место рядом с более традиционными книгами по программированию. Эта книга — одна из них.

Грег Лейден (Greg Laden), ScienceBlogs

Автор рассматривает довольно сложные проекты и излагает материал в последовательной, логичной и приятной манере, которая привлекает читателя к теме.

Full Circle Magazine

Хорошая подача материала с доступными объяснениями фрагментов кода. Книга ведет читателя вперед шаг за шагом, строя все более сложный код и объясняя все происходящее на этом пути.

FlickThrough Reviews

Изучение Python с этой книгой оставило в высшей степени положительные впечатления! Отличный вариант, если вы только начинаете изучать Python.

Mikke Goes Coding

Об авторе

Эрик Мэтиз, преподаватель физики и математики, живет на Аляске и ведет курс Python начального уровня. Эрик пишет программы с пяти лет, а в настоящее время занимается разработкой продуктов, которые исправляют недочеты в системе образования и помогают использовать возможности программных продуктов с открытым кодом в системе образования. В свободное время занимается альпинизмом и проводит время с семьей.

О научном редакторе

Кеннет Лав — преподаватель и программист Python с многолетним стажем. Он выступал с докладами и лекциями на конференциях, занимался профессиональной подготовкой, работал внештатным программистом Python и Django, а в настоящее время работает программистом в O'Reilly Media. Кеннет является одним из создателей пакета `django-braces`, предоставляющего удобные примеси (mixins) для представлений на базе классов Django. Желающие могут читать его сообщения в Твиттере (@kennethlove).

Моему отцу, который никогда не жалел времени, чтобы ответить на мои вопросы по программированию, и Эверу, который только начинает задавать мне свои вопросы

Предисловие к третьему изданию

Предыдущее издание «Изучаем Python: программирование игр, визуализация данных, веб-приложения» получило множество положительных отзывов. Тираж книги составил более 500 тысяч экземпляров, включая переводы на 8 языков. Я получал письма и сообщения как от читателей, которым было всего десять лет, так и от пенсионеров, желающих освоить программирование для собственного удовольствия. Материал книги используется для обучения в школах и колледжах. Студенты, которым приходится иметь дело с более сложными учебниками, используют мою книгу как дополнительный источник для своих занятий; по их мнению, книга является хорошим подспорьем в учебе. Разные люди используют ее для повышения своей квалификации и для работы над параллельными проектами. Короче говоря, книга используется для самых разнообразных целей — как я и надеялся.

Я был очень рад тому, что мне представилась возможность поработать над следующим изданием. Python — достаточно зрелый язык, но он продолжает развиваться, как и все языки. При переработке материала книги я старался сделать его более доступным и компактным. Необходимости в изучении Python 2 больше нет, поэтому новое издание полностью сосредоточено на Python 3. Установка многих пакетов Python упростилась, поэтому упростились и инструкции по установке и настройке. Я добавил несколько тем, которые, на мой взгляд, будут полезны читателям, обновил некоторые разделы и представил новые, более простые средства для решения некоторых задач в языке Python. Также были доработаны разделы, в которых некоторые аспекты языка были представлены недостаточно точно. Все проекты были полностью обновлены; в них используются только популярные библиотеки с качественным сопровождением, которыми вы можете пользоваться при построении собственных проектов.

Ниже приведена краткая сводка изменений в третьем издании:

- В главе 1 инструкции по установке Python были упрощены для пользователей всех основных операционных систем. Теперь я рекомендую использовать текстовый редактор Sublime Text, который популярен как среди новичков, так и среди профессионалов и работает во всех операционных системах.
- В главе 2 приведено более подробное описание того, как переменные реализованы в языке Python. Переменные описываются как текстовые *метки* для значений, что позволяет читателю лучше понять поведение пере-

менных в Python. В книге широко используются f-строки, появившиеся в Python 3.6, — намного более простой и удобный механизм использования значений в строках. В Python 3.6 также появилась возможность использования символов подчеркивания для представления больших чисел (например, `1_000_000`); она также включена в это издание. Множественное присваивание переменных было представлено в одном из проектов первого издания; описание было обобщено и перемещено в главу 2 для удобства читателей. Наконец, в эту главу также включена понятная система соглашений для представления констант в Python.

- В главе 6 представлен метод `get()` для чтения значений из словаря, который может вернуть значение по умолчанию в случае отсутствия ключа.
- Проект «Инопланетное вторжение» (главы 12–14) теперь полностью базируется на использовании классов. Сама игра представляет собой класс (вместо серии функций, как в предыдущем издании). Это сильно упрощает общую структуру игры, уменьшает число вызовов функций и необходимых параметров. Читатели, знакомые с первым изданием, оценят простоту нового подхода, основанного на классах. Pygame теперь может устанавливаться однострочной командой во всех системах, а читатель может выбрать между запуском игры в полноэкранном или оконном режиме.
- В проектах визуализации данных инструкции по установке Matplotlib упростились для всех операционных систем. Визуализации, базирующиеся на Matplotlib, используют функцию `subplots()`, более удобную для обучения построению сложных визуализаций. В проекте Rolling Dice из главы 15 используется Plotly — библиотека визуализации с простым синтаксисом, качественным сопровождением и красивым выводом результатов с широкими возможностями настройки.
- В главе 16 метеорологический проект переведен на данные NOAA, которые продемонстрировали большую стабильность за прошедшие годы, чем сайт, использованный в первом издании. Картографический проект теперь посвящен глобальной сейсмической активности; к моменту завершения проекта будет построена великолепная визуализация, наглядно изображающая границы тектонических плит по координатам всех землетрясений за заданный период времени. Вы научитесь наносить на карту любые наборы данных, содержащие географические координаты.
- В главе 17 для визуализации активности, связанной с Python, в проектах с открытым кодом на GitHub используется библиотека Plotly.
- Проект Learning Log (главы 18–20) построен на базе новейшей версии Django, а для его оформления используется новейшая версия Bootstrap. Процесс развертывания проекта в Heroku был упрощен за счет использования пакета `django-heroku`, и в нем используются переменные среды вместо редактирования файлов `settings.py`. Такое решение получается более простым и лучше соответ-

ствует тому, как профессиональные программисты развертывают современные проекты Django.

- Приложение А было полностью переработано. В него были включены рекомендации по установке Python. В приложение Б включены подробные рекомендации по настройке Sublime Text и приведены краткие описания основных текстовых редакторов и интегрированных сред, используемых в наши дни. В приложении В приведены ссылки на более новые и популярные сетевые ресурсы для получения помощи, а приложение Г предлагает небольшой учебный курс по применению Git для контроля версий.

Спасибо за то, что читаете эту книгу! Если у вас появятся вопросы или вы захотите поделиться своим мнением, не стесняйтесь и пишите.

Благодарности

Эта книга никогда бы не появилась на свет без великолепных, чрезвычайно профессиональных сотрудников издательства *No Starch Press*. Билл Поллок (Bill Pollock) предложил мне написать вводный учебник, и я глубоко благодарен ему за это предложение. Тайлер Ортман (Tyler Ortman) помог привести в порядок мои идеи на ранней стадии подготовки чернового варианта. Лиз Чедвик (Liz Chadwick) и Лесли Шен (Leslie Shen) предоставили бесценные отзывы на исходные варианты каждой главы, а Энн Мэри Уокер (Anne Marie Walker) помогла прояснить многие части книги. Райли Хоффман (Riley Hoffman) отвечал на все вопросы, которые возникали у меня в процессе создания книги, и терпеливо превращал мою работу в прекрасный завершенный продукт.

Также хочу поблагодарить Кеннета Лава, научного редактора книги. Я познакомился с Кеннетом на конференции PyCon, и его энтузиазм в отношении языка и сообщества Python с тех пор неизменно оставался для меня источником профессионального вдохновения. Кеннет вышел за рамки простой проверки фактов; он следил за тем, чтобы книга помогала начинающим программистам сформировать основательное понимание языка Python и программирования в целом. Вместе с тем ответственность за все оставшиеся неточности лежит исключительно на мне.

Я хочу поблагодарить своего отца, который познакомил меня с программированием в раннем возрасте и не побоялся, что я сломаю его оборудование. Также хочу сказать спасибо своей жене Эрин за поддержку и помощь во время работы над книгой и своему сыну Эверу, чья любознательность постоянно служит мне примером.

Введение

У каждого программиста найдется своя история о том, как он написал свою первую программу. Я начал изучать программирование еще в детстве, когда мой отец работал на Digital Equipment Corporation, одну из ведущих компаний современной эры вычислительной техники. Я написал свою первую программу на компьютере, который был собран моим отцом из набора компонентов в подвале дома. Компьютер представлял собой системную плату (без корпуса), подключенную к клавиатуре, а в качестве монитора использовалась простейшая электронно-лучевая трубка. Моей первой программой стала игра по отгадыванию чисел, которая выглядела примерно так:

```
Я загадал число! Попробуйте отгадать мое число: 25
Слишком мало! Следующая попытка: 50
Слишком много! Следующая попытка: 42
Верно! Хотите сыграть снова? (да/нет) нет
Спасибо за игру!
```

Никогда не забуду, как это было здорово — моя семья играла в написанную мной игру, и все работало точно так, как я задумал.

Мои ранние переживания имели далекоидущие последствия. Очень приятно построить нечто, предназначенное для конкретной цели; нечто, успешно решающее свою задачу. Программы, которые я пишу сейчас, намного серьезнее моих детских попыток, но чувство удовлетворения, которое я ощущаю от вида работающей программы, остается практически тем же.

Для кого написана эта книга?

Цель этой книги — как можно быстрее ввести читателя в курс дела, чтобы он начал писать на Python работоспособные программы (игры, визуализации данных и веб-приложения), и одновременно заложить основу в области программирования, которая пригодится ему на протяжении всей жизни. Книга написана для людей любого возраста, которые прежде никогда не программировали на Python или вообще никогда не программировали. Если вы хотите быстро изучить азы программирования, чтобы сосредоточиться на интересных проектах, а также проверить свое понимание новых концепций на содержательных задачах, — эта книга для вас. Книга также

прекрасно подходит для преподавателей, желающих предложить вводный курс программирования, основанный на проектах. Наконец, если вы студент, которому хочется иметь более доступное введение в Python, чем предложенный ему учебник, эта книга также упростит ваши учебу.

Чему эта книга вас научит?

Цель книги — сделать вас хорошим программистом вообще и хорошим программистом Python в частности. Процесс обучения будет эффективным, и вы приобретете много полезных привычек, так как я представлю основательное введение в общие концепции программирования. После того как вы перевернете последнюю страницу, вы будете готовы к знакомству с более серьезными возможностями Python, а изучение вашего следующего языка программирования тоже упростится.

В части I книги будут представлены базовые концепции программирования, которые необходимо знать для написания программ Python. Эти концепции ничем не отличаются от тех, которые рассматриваются в начале изучения почти любого языка программирования. Вы познакомитесь с разными видами данных и возможностями хранения данных в списках и словарях. Вы научитесь создавать коллекции данных и эффективно работать с этими коллекциями. В частности, циклы `while` и `if` позволяют выполнять определенные фрагменты кода, если некоторое условие истинно, и выполнять другие фрагменты в противном случае — эти конструкции очень сильно помогают при автоматизации процессов.

Вы научитесь получать входные данные от пользователя, чтобы ваши программы стали интерактивными, и выполнять их до тех пор, пока пользователь остается активным. Также вы узнаете, как написать функции для многократного выполнения некоторых частей ваших программ, чтобы вы один раз программировали некоторое действие, а потом могли использовать его столько раз, сколько потребуется. Затем эта концепция будет распространена на более сложное поведение с классами, что позволит даже относительно простым программам реагировать на множество разнообразных ситуаций. Вы научитесь писать программы, корректно обрабатывающие многие типичные ошибки. После знакомства с базовыми концепциями мы напишем несколько коротких программ для решения конкретных задач. Наконец, вы сделаете первые шаги на пути к программированию среднего уровня: вы научитесь писать тесты для своего кода, чтобы продолжать разработку программ, не беспокоясь о возможном внесении ошибок. Вся информация части I подготовит вас к более сложным и масштабным проектам.

В части II знания, полученные в части I, будут применены для построения трех проектов. Вы можете взяться за любые из этих проектов в том порядке, который лучше подходит для вас. В первом проекте (главы 12–14) будет создана игра-стрелялка в стиле классического хита Space Invaders, состоящая из многих уровней с нарастающей сложностью. После завершения этого проекта вы будете знать многое из того, что необходимо знать для разработки собственных 2D-игр.

Второй проект (главы 15–17) познакомит вас с визуализацией данных. Чтобы разобраться в огромных объемах доступной информации, специалисты по анализу данных применяют различные средства визуализации. Вы будете работать с наборами данных, генерируемыми в программах; наборами данных, загруженными из сетевых источников; и наборами данных, которые загружаются вашей программой автоматически. После завершения этого проекта вы сможете писать программы, обрабатывающие большие наборы данных и строящие визуальные представления сохраненной информации.

В третьем проекте (главы 18–20) будет построено небольшое веб-приложение Learning Log. Этот проект позволяет вести журнал новых идей и концепций, которые вы узнали в ходе изучения конкретной темы. Пользователь приложения сможет вести разные журналы по разным темам, создавать учетные записи и начинать новые журналы. Вы также узнаете, как развернуть свой проект в интернете, чтобы любой желающий мог работать с ним откуда угодно.

Онлайн-ресурсы

Все информационные ресурсы, прилагаемые к книге, доступны в интернете по адресу https://nostarch.com/pythoncrashcourse2e/ или http://ehmatthes.github.io/pcc_2e/:

- **Инструкции по настройке**. Они совпадают с инструкциями, приведенными в книге, но в них включены активные ссылки, относящиеся к различным фазам настройки. Если у вас возникнут какие-либо проблемы с подготовкой и настройкой, обращайтесь к этому ресурсу.
- **Обновления**. Python, как и все языки, постоянно развивается. Я отслеживаю наборы обновлений, так что если что-то не работает — обращайтесь к этому ресурсу и проверьте, не изменилось ли что-то.
- **Решения к упражнениям**. Не жалейте времени на самостоятельное решение задач из разделов «Упражнения». Но если вы оказались в тупике и не знаете, что делать, ответы к большинству упражнений доступны в интернете.
- **Памятки**. Полный набор памяток, содержащих краткие сводки по основным концепциям.

Почему именно Python?

Каждый год я задумываюсь над тем, продолжать ли мне работать на Python или же перейти на другой язык — вероятно, более новый в мире программирования. И все же я продолжаю работать на Python по многим причинам. Язык Python невероятно эффективен: программы, написанные на нем, делают больше, чем многие на других языках и в меньшем объеме кода. Синтаксис Python также позволяет писать «чистый» код. Ваш код будет легко читаться, у вас будет меньше проблем с отладкой и расширением программ по сравнению с другими языками.

Python используется для разных целей: для создания игр, построения веб-приложений, решения бизнес-задач и разработки внутренних инструментов для всевозможных интересных проектов. Python также широко применяется в научной области для теоретических исследований и решения прикладных задач.

Впрочем, одной из самых важных причин для использования Python для меня остается сообщество Python, состоящее из невероятно разных и благожелательных людей. Сообщество играет исключительно важную роль в программировании, потому что программирование не является сугубо индивидуальным делом. Многим из нас, даже самым опытным программистам, приходится обращаться за советом к коллегам, которые уже решали похожие задачи. Существование связного, доброжелательного сообщества помогает решать задачи, и сообщество Python готово прийти на помощь людям, для которых Python является первым языком программирования.

Python — замечательный язык, так давайте браться за дело!

От издательства

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

Мы будем рады узнать ваше мнение!

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

Часть I
ОСНОВЫ

В части I этой книги представлены базовые концепции, необходимые для написания программ на языке Python. Многие из этих концепций встречаются во всех языках программирования, поэтому они пригодятся вам на протяжении всей карьеры в программировании.

В главе 1 вы установите Python на свой компьютер и запустите свою первую программу, которая выводит на экран сообщение `Hello world!`.

В главе 2 вы научитесь хранить информацию в переменных, работать с текстовыми и числовыми данными.

В главах 3 и 4 вы познакомитесь со списками. Списки позволяют хранить любой объем информации в одной переменной, что повышает эффективность работы с данными. Вы сможете работать с сотнями, тысячами и даже миллионами значений всего в нескольких строках кода.

В главе 5 будут представлены команды `if`. С их помощью вы сможете написать код, который делает что-то одно, если некоторое условие истинно, и что-то другое, если условие не выполняется.

Глава 6 показывает, как использовать словари Python, связывающие разные виды информации. Словари, как и списки, могут содержать столько информации, сколько вы захотите в них поместить.

В главе 7 вы научитесь получать данные от пользователей, чтобы ваши программы стали интерактивными. Также в этой главе описаны циклы `while`, многократно выполняющие блоки кода, пока некоторое условие остается истинным.

В главе 8 вы займетесь написанием функций — именованных блоков кода, которые решают конкретную задачу и запускаются тогда, когда потребуется.

В главе 9 представлены классы, предназначенные для моделирования реальных объектов: собак, кошек, людей, машин, ракет и т. д. С их помощью вы сможете представить в своем коде любые сущности, реальные или абстрактные.

Глава 10 научит вас работать с файлами и обрабатывать ошибки, чтобы ваши программы не завершались аварийно. Вы узнаете, как сохранить данные перед закрытием программы и снова загрузить их при запуске программы. В этой главе рассматриваются исключения Python; с их помощью вы сможете предвидеть возможные ошибки и организовать их корректную обработку в программах.

В главе 11 вы научитесь писать тесты для своего кода. Тесты проверяют, что ваша программа работает так, как было задумано. В результате вы сможете дорабатывать свои программы, не беспокоясь о возможном внесении новых ошибок. Тестирование — один из первых навыков, отличающих новичка от программиста среднего уровня.

1 Первые шаги

В этой главе вы запустите свою первую программу на языке Python, hello_world.py. Сначала вы проверите, установлен ли Python на вашем компьютере, и если нет — установите его. Также будет установлен текстовый редактор для подготовки программ Python. Текстовые редакторы распознают код Python и выделяют синтаксические конструкции во время работы, упрощая понимание структуры кода разработчиком.

Подготовка среды программирования

Поддержка Python слегка отличается в разных операционных системах, поэтому вы должны учитывать некоторые аспекты. В этой главе мы проверим, что Python правильно установлен в вашей системе.

Версии Python

Каждый язык программирования развивается с появлением новых идей и технологий, и разработчики Python неустанно трудятся над тем, чтобы сделать язык более мощным и гибким. На момент написания книги новейшей была версия 3.7, но все программы в книге должны нормально работать в версии 3.6. В этом разделе вы определите, установлена ли поддержка Python в вашей системе и нужно ли установить новую версию. В приложении A содержится подробное руководство по установке новейших версий Python во всех основных операционных системах.

Некоторые старые проекты Python продолжают использовать Python 2, но вам стоит остановиться на Python 3. Если в вашей системе установлена только версия Python 2, скорее всего, она нужна для поддержки других программ, необходимых для вашей системы. Мы оставим эту установку без изменений и позаботимся о том, чтобы в системе была более свежая версия для работы.

Выполнение фрагментов кода Python

Интерпретатор Python может выполняться в терминальном окне и позволяет опробовать фрагменты кода Python без сохранения и запуска всей программы.

В этой книге встречаются фрагменты следующего вида:

```
❶ >>> print("Hello Python interpreter!")
  Hello Python interpreter!
```

Приглашение >>> означает, что используется окно терминала, а жирным шрифтом выделена команда, которую вы вводите и выполняете нажатием клавиши Enter. Большинство примеров в книге представляет небольшие самостоятельные программы, которые запускаются из редактора, а не в терминале, потому что именно так вы будете писать большую часть своего кода. Но в некоторых случаях базовые концепции будут продемонстрированы серией фрагментов в терминальном сеансе Python, чтобы более эффективно показать отдельные концепции. Каждый раз, когда в листинге встречаются три угловые скобки ❶, это означает, что перед вами вывод терминального сеанса. Вскоре мы опробуем возможность программирования в интерпретаторе для вашей системы.

Также текстовый редактор будет использоваться для создания простой программы *Hello World!*. В мире программирования издавна принято начинать освоение нового языка с программы, выводящей на экран сообщение *Hello world!* — считается, что это принесет удачу. Даже такая простая программа выполняет вполне конкретную функцию. Если она запускается в вашей системе, то и любая программа, которую вы напишете на Python, тоже должна запускаться нормально.

О текстовом редакторе Sublime Text

Sublime Text — простой текстовый редактор, который устанавливается во всех основных операционных системах. Sublime Text позволяет запускать практически любые программы прямо из редактора (вместо терминала), а код выполняется в терминальном сеансе, встроенном в окно Sublime Text, что упрощает просмотр вывода.

Редактор Sublime Text особенно удобен для начинающих, но многие профессиональные программисты также пользуются им. Если вы привыкнете к нему во время изучения Python, возможно, вы будете пользоваться им и при переходе на более крупные и сложные проекты. Политика лицензирования Sublime Text более чем либеральна: вы можете бесплатно пользоваться редактором сколь угодно долго, но автор требует приобрести лицензию, если программа вам понравилась и вы собираетесь использовать ее в будущем.

В приложении Б приведена информация о других вариантах; возможно, вам стоит бегло просмотреть его. Если вы хотите быстро перейти к программированию, используйте Sublime Text на первых порах и рассмотрите переход на другие редакторы, когда у вас появится некоторый опыт программирования. В этой главе я опишу процесс установки Sublime Text для вашей операционной системы.

Python в разных операционных системах

Python является кросс-платформенным языком программирования; это означает, что он работает во всех основных операционных системах. Любая программа на

языке Python, написанная вами, должна выполняться на любом современном компьютере с установленной поддержкой Python. Впрочем, способы настройки Python для разных операционных систем слегка отличаются.

В этом разделе вы узнаете, как подготовить Python к работе в вашей системе. Сначала вы проверите, установлена ли новая версия Python в вашей системе, и если нет — установите ее. Затем вы установите Sublime Text. Процедура состоит всего из двух шагов, отличающихся для разных операционных систем.

Затем вы установите программу *Hello World* и устраните любые неполадки. Этот процесс будет описан для всех операционных систем, так что в итоге в вашем распоряжении появится простая и удобная среда программирования на Python.

Python в системе Windows

Windows далеко не всегда включает поддержку Python. Скорее всего, вам придется загрузить и установить Python, а затем загрузить и установить текстовый редактор Sublime Text.

Установка Python

Для начала проверьте, установлена ли поддержка Python в вашей системе. Откройте окно командной строки: введите `command` в меню Пуск или щелкните на рабочем столе с нажатой клавишей Shift и выберите команду Open command window here. Введите в окне командной строки команду `python` в нижнем регистре. Если на экране появится приглашение `>>>`, значит, в системе установлена поддержка Python. Впрочем, скорее всего, вместо приглашения появится сообщение об ошибке, в котором говорится, что команда python не опознана системой.

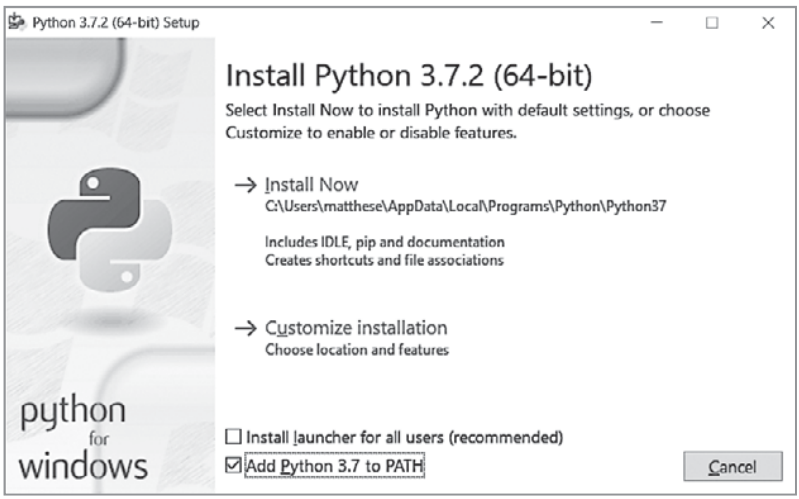

Рис. 1.1. Не забудьте установить флажок Add Python to PATH

В таком случае (а также если будет выведена версия Python менее 3.6) загрузите программу установки Python для Windows. Откройте страницу https://python.org/ и наведите указатель мыши на ссылку Downloads. Появляется кнопка для загрузки новейшей версии Python. Щелкните на кнопке, которая запускает автоматическую загрузку правильного установочного пакета для вашей системы. После того как загрузка файла будет завершена, запустите программу установки. Не забудьте установить флажок Add Python to PATH; это упростит правильную настройку системы. На рис. 1.1 изображено окно мастера установки с установленным флажком.

Запуск Python в терминальном сеансе

Откройте окно командной строки и введите команду `python` в нижнем регистре. Если на экране появится приглашение Python (`>>>`), значит, система Windows обнаружила установленную версию Python:

```
C:\> python
Python 3.7.2 (v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

ПРИМЕЧАНИЕ Если вы не увидите этот (или похожий) вывод, обращайтесь к более подробным инструкциям по установке в приложении А.

Введите в сеансе Python следующую строку и убедитесь в том, что на экране появился вывод `Hello Python interpreter!`

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Каждый раз, когда вы захотите выполнить фрагмент кода Python, откройте окно командной строки и запустите терминальный сеанс Python. Чтобы закрыть терминальный сеанс, нажмите Ctrl+Z, а затем Enter или введите команду `exit()`.

Установка Sublime Text

Программу установки Sublime Text можно загрузить по адресу https://sublimetext.com/. Щелкните на ссылке загрузки и найдите программу установки для Windows. После того как программа установки будет загружена, запустите ее и подтвердите все настройки по умолчанию.

Python в системе macOS

В большинстве систем macOS поддержка Python уже установлена, но скорее всего, это будет какая-нибудь устаревшая версия, которую не стоит использовать для обучения. В этом разделе мы установим новейшую версию Python, а затем текстовый редактор Sublime Text и убедимся в том, что он правильно настроен.

Проверка наличия Python 3 в системе

Откройте терминальное окно (команда Applications ▸ Utilities ▸ Terminal). Также можно нажать Command+пробел, ввести terminal и нажать Enter. Чтобы проверить, установлена ли поддержка Python в вашей системе, введите команду `python` (со строчной буквы `p`). На экране появится информация о том, какая версия Python у вас установлена, и приглашение `>>>`, в котором можно вводить команды Python:

```
$ python
Python 2.7.15 (default, Aug 17 2018, 22:39:05)
[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>>
```

Этот вывод сообщает, что Python 2.7.15 в настоящее время является версией Python по умолчанию, установленной на данном компьютере. Нажмите Ctrl+D или введите `exit()`, чтобы выйти из приглашения Python и вернуться к приглашению терминала.

Чтобы проверить наличие Python 3, попробуйте ввести команду `python3`. На экране может появиться сообщение об ошибке, но если из вывода следует, что в системе установлена версия Python 3.6 или более поздняя версия, вы можете перейти к разделу «Запуск Python в терминальном сеансе» этой главы. Если Python 3 не устанавливается по умолчанию, ее необходимо установить вручную. Каждый раз, когда в книге встречается команда `python`, вводите вместо нее команду `python3`, чтобы при запуске использовалась версия Python 3, а не Python 2; различия между этими версиями достаточно серьезны, чтобы при выполнении кода в книге с Python 2 могли возникнуть проблемы.

Если команда выводит любой номер версии, предшествующий 3.6, выполните инструкции в следующем разделе для установки новейшей версии.

Установка новейшей версии Python

Программа установки Python доступна на сайте https://python.org/. Наведите указатель мыши на ссылку Downloads. Появляется кнопка для загрузки новейшей версии Python. Щелкните на кнопке, которая запускает автоматическую загрузку правильного установочного пакета для вашей системы. После того как загрузка файла будет завершена, запустите программу установки.

Когда установка будет завершена, введите следующую команду в приглашении терминала:

```
$ python3 --version
Python 3.7.2
```

Если появится похожий результат, значит, все готово к работе с Python. Каждый раз, когда в книге встречается команда `python`, вводите вместо нее команду `python3`.

Запуск Python в терминальном сеансе

Для выполнения фрагментов кода Python можно открыть терминальное окно и ввести команду python3. Введите в терминальном сеансе следующую строку:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Сообщение выводится прямо в текущем терминальном окне. Вспомните, что интерпретатор Python закрывается комбинацией клавиш Ctrl+D или командой `exit()`.

Установка Sublime Text

Чтобы установить редактор Sublime Text, необходимо загрузить программу установки по адресу https://sublimetext.com/. Щелкните на ссылке Download и найдите программу установки для macOS. После того как программа установки будет загружена, откройте ее и перетащите значок Sublime Text в папку Applications.

Python в системе Linux

Системы семейства Linux ориентированы на программистов, поэтому поддержка Python уже установлена на большинстве компьютеров Linux. Люди, которые занимаются разработкой и сопровождением Linux, ожидают, что в какой-то момент вы займетесь программированием, и всячески способствуют этому. По этой причине для перехода к программированию вам почти ничего не придется устанавливать, а количество необходимых настроек будет минимальным.

Проверка версии Python

Откройте терминальное окно, запустив приложение Terminal в вашей системе (в Ubuntu нажмите клавиши Ctrl+Alt+T). Чтобы проверить, какая версия Python установлена в вашей системе, введите команду `python3` (со строчной буквы `p`). Если Python присутствует в системе, эта команда запустит интерпретатор Python. На экране появится информация о том, какая версия Python у вас установлена, и приглашение >>>, в котором можно вводить команды Python:

```
$ python3
Python 3.7.2 (default, Dec 27 2018, 04:01:51)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Этот вывод сообщает, что Python 3.7.2 в настоящее время является версией Python по умолчанию, установленной на данном компьютере. Нажмите Ctrl+D или введите `exit()`, чтобы выйти из приглашения Python и вернуться к приглашению терми-

нала. Каждый раз, когда в книге встречается команда `python`, вводите вместо нее команду `python3`.

Для запуска кода из книги необходима версия Python 3.6. Если в системе установлена более ранняя версия, обращайтесь к подробным инструкциям по установке в приложении А.

Запуск Python в терминальном сеансе

Для выполнения фрагментов кода Python можно открыть терминальное окно и ввести команду `python3`, как мы поступили при проверке версии. Сделайте то же самое, но на этот раз введите в терминальном сеансе следующую строку:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Сообщение выводится прямо в текущем терминальном окне. Вспомните, что интерпретатор Python закрывается комбинацией клавиш Ctrl+D или командой `exit()`.

Установка Sublime Text

В системе Linux редактор Sublime Text устанавливается из Ubuntu Software Center. Щелкните на значке Ubuntu Software в меню и найдите вариант Sublime Text. Щелкните на нем, чтобы установить, а потом запустить программу.

Запуск программы Hello World

После того как в вашей системе будут установлены последние версии Python и Sublime Text, все почти готово к запуску вашей первой программы Python, написанной в текстовом редакторе. Но перед этим необходимо убедиться в том, что в Sublime Text настроена правильная версия Python для вашей системы. После этого вы сможете написать программу *Hello World!* и запустить ее.

Настройка Sublime Text для использования правильной версии Python

Если команда `python` в вашей системе запускает Python 3, ничего настраивать не нужно и вы можете просто перейти к следующему разделу. Если же вы используете команду `python3`, необходимо настроить Sublime Text для использования правильной версии Python при запуске программ.

Щелкните на значке Sublime Text или проведите поиск `Sublime Text` в строке поиска вашей системы и запустите редактор. Выполните команду Tools ▸ Build System ▸ New Build System, которая откроет новый конфигурационный файл. Удалите текущее содержимое и введите следующий текст:

Python3.sublime-build

```
{
    "cmd": ["python3", "-u", "$file"],
}
```

Он приказывает Sublime Text использовать команду `python3` для запуска программных файлов Python. Сохраните этот файл под именем `Python3.sublime-build` в каталоге по умолчанию, который открывает Sublime Text при выполнении команды Save.

Запуск hello_world.py

Прежде чем писать первую программу, создайте где-нибудь в своей системе папку с именем python_work для своих проектов. В именах файлов и папок лучше использовать буквы нижнего регистра и символы подчеркивания в соответствии со схемой выбора имен в Python.

Откройте Sublime Text и сохраните пустой файл Python (File ▸ Save As) с именем hello_world.py в папке python_work. Расширение .py сообщает Sublime Text, что код в файле написан на языке Python; эта информация помогает редактору запустить программу и правильно выделить цветом элементы синтаксиса.

После того как файл будет сохранен, введите следующую строку в текстовом редакторе:

hello_world.py

```
print("Hello Python world!")
```

Если команда `python` работает в вашей системе, программу можно запустить командой меню Tools ▸ Build или комбинацией клавиш Ctrl+B (Command+B в macOS). Если вы настроили Sublime Text на использование другой команды вместо `python`, выберите команду меню Tools ▸ Build System, а затем Python 3. В дальнейшем вы сможете запускать свои программы командой Tools ▸ Build или комбинацией клавиш Ctrl+B (или Command+B).

В нижней части окна Sublime Text должно отображаться терминальное окно со следующим текстом:

```
Hello Python world!
[Finished in 0.1s]
```

Если вы не увидели это сообщение, проверьте каждый символ во введенной программе. Может, вы случайно набрали `print` с прописной буквы? Пропустили одну либо обе кавычки или круглые скобки? В языках программирования используется предельно конкретный синтаксис, и при малейшем его нарушении произойдет ошибка. Если программа так и не заработала, обращайтесь к следующему разделу.

Решение проблем с установкой

Если вам так и не удалось запустить программу hello_world.py, возможно, вам помогут следующие полезные советы (кстати, они могут пригодиться для решения любых проблем в программах).

- Если программа содержит серьезную ошибку, Python выводит данные *трассировки*. Python анализирует содержимое файла и пытается составить отчет о проблеме. Возможно, трассировка подскажет, что именно мешает выполнению программы.
- Отойдите от компьютера, отдохните и попробуйте снова. Помните, что синтаксис в программировании очень важен; даже пропущенное двоеточие, неверно расположенная кавычка или непарная скобка могут помешать нормальной работе программы. Перечитайте соответствующие части главы, еще раз проанализируйте, что было сделано, и попробуйте найти ошибку.
- Начните заново. Вероятно, ничего переустанавливать не придется, но попробуйте удалить файл hello_world.py и создать его с нуля.
- Попросите кого-нибудь повторить действия, описанные в этой главе, на вашем (или на другом) компьютере. Внимательно понаблюдайте за происходящим. Возможно, вы упустили какую-нибудь мелочь, которую заметят другие.
- Найдите специалиста, хорошо знающего Python, и попросите его помочь вам. Вполне может оказаться, что такой специалист есть среди ваших знакомых.
- Инструкции по настройке среды программирования, приведенные в этой главе, также доступны по адресу https://nostarch.com/pythoncrashcourse2e/. Возможно, сетевая версия будет более удобной для вас.
- Обратитесь за помощью в интернет. В приложении B перечислены некоторые ресурсы (форумы, чаты и т. д.), на которых вы сможете проконсультироваться у людей, уже сталкивавшихся с вашей проблемой.

Не стесняйтесь обращаться к опытным программистам. Любой программист в какой-то момент своей жизни заходил в тупик; многие программисты охотно помогут вам правильно настроить вашу систему. Если вы сможете четко объяснить, что вы хотите сделать, что уже пытались и какие результаты получили, скорее всего, кто-нибудь вам поможет. Как упоминалось во введении, сообщество Python доброжелательно относится к новичкам.

Python должен нормально работать на любом современном компьютере, и если у вас все же возникли проблемы — обращайтесь за помощью. На первых порах проблемы могут быть весьма неприятными, но с ними стоит разобраться. Когда программа hello_world.py заработает, вы сможете приступить к изучению Python, а ваша работа станет намного более интересной и принесет больше удовольствия.

Запуск программ Python в терминале

Большинство программ, написанных вами в текстовом редакторе, будут запускаться прямо из редактора. Тем не менее иногда бывает полезно запускать программы из терминала — например, если вы хотите просто выполнить готовую программу, не открывая ее для редактирования.

Это можно сделать в любой системе с установленной поддержкой Python; необходимо лишь знать путь к каталогу, в котором хранится файл программы. Приведенные ниже примеры предполагают, что вы сохранили файл hello_world.py в папке python_work на рабочем столе.

В Windows

Команда cd (Change Directory) используется для перемещения по файловой системе в окне командной строки. Команда dir (DIRectory) выводит список всех файлов в текущем каталоге.

Откройте новое терминальное окно и введите следующие команды для запуска программы hello_world.py:

```
❶ C:\> cd Desktop\python_work
❷ C:\Desktop\python_work> dir
    hello_world.py
❸ C:\Desktop\python_work> python hello_world.py
    Hello Python world!
```

Команда cd используется для перехода к папке python_work, находящейся в папке Desktop ❶. Затем команда dir проверяет, что файл hello_world.py действительно находится в этой папке ❷. Далее файл запускается командой python hello_world.py ❸.

Большинство программ будет нормально запускаться из редактора. Но со временем ваша работа станет более сложной, и возможно, вы предпочтете запускать некоторые из своих программ из терминала.

В macOS и Linux

Запуск программы Python в терминальном сеансе в системах Linux и macOS осуществляется одинаково. Команда cd (Change Directory) используется для перемещения по файловой системе в терминальном сеансе. Команда ls (LiSt) выводит список всех нескрытых файлов в текущем каталоге.

Откройте новое терминальное окно и введите следующие команды для запуска программы hello_world.py:

```
❶ ~$ cd Desktop/python_work/
❷ ~/Desktop/python_work$ ls
    hello_world.py
❸ ~/Desktop/python_work$ python hello_world.py
    Hello Python world!
```

Команда cd используется для перехода к папке python_work, находящейся в папке Desktop ❶. Затем команда ls проверяет, что файл hello_world.py действительно находится в этой папке ❷. Далее файл запускается командой python hello_world.py ❸.

Как видите, все просто. По сути, вы просто используете команду python (или python3) для запуска программ Python.

УПРАЖНЕНИЯ

Упражнения этой главы в основном направлены на самостоятельный поиск информации. Начиная с главы 2, упражнения будут ориентированы на решение задач по изложенному материалу.

1.1. python.org: изучите домашнюю страницу Python (https://python.org/) и найдите темы, которые вас заинтересуют. Со временем вы лучше узнаете Python, и другие разделы этого сайта покажутся вам более полезными.

1.2. Опечатки в Hello World: откройте только что созданный файл hello_world.py. Сделайте где-нибудь намеренную опечатку и снова запустите программу. Удастся ли вам сделать опечатку, которая приводит к ошибке? Поймете ли вы смысл сообщения об ошибке? Удастся ли вам сделать опечатку, которая не приводит к ошибке? Как вы думаете, почему на этот раз выполнение обходится без ошибки?

1.3. Бесконечное мастерство: если бы вы были программистом с неограниченными возможностями, за какой проект вы бы взялись? Вы сейчас учитесь программировать. Если у вас имеется ясное представление о конечной цели, вы сможете немедленно применить свои новые навыки на практике; попробуйте набросать общие описания тех программ, над которыми вам хотелось бы поработать. Заведите «блокнот идей», к которому вы сможете обращаться каждый раз, когда собираетесь начать новый проект. Выделите пару минут и составьте описания трех программ, которые вам хотелось бы создать.

Итоги

В этой главе вы познакомились с языком Python и установили поддержку Python в своей системе, если она не была установлена ранее. Также вы установили текстовый редактор, упрощающий работу над кодом Python. Вы научились выполнять фрагменты кода Python в терминальном сеансе и запустили свою первую настоящую программу hello_world.py. Скорее всего, попутно вы кое-что узнали о поиске и исправлении ошибок.

В следующей главе рассматриваются структуры данных, с которыми вы будете работать в программах Python. Кроме того, вы научитесь пользоваться переменными Python.

2 Переменные и простые типы данных

В этой главе представлены разные виды данных, с которыми вы будете работать в своих программах Python. Вы также научитесь использовать переменные для представления данных в своих программах.

Что происходит при запуске hello_world.py

Давайте повнимательнее разберемся с тем, что же делает Python при запуске hello_world.py. Оказывается, даже для такой простой программы Python выполняет достаточно серьезную работу:

hello_world.py

```
print("Hello Python world!")
```

При выполнении этого кода выводится следующий текст:

```
Hello Python world!
```

Суффикс .py в имени файла hello_world.py указывает, что файл является программой Python. Редактор запускает файл в *интерпретаторе* Python, который читает программу и определяет, что означает каждое слово в программе. Например, когда интерпретатор обнаруживает слово `print`, он выводит на экран текст, заключенный в скобки.

Во время написания программы редактор выделяет цветом разные части программы. Например, он понимает, что `print` является именем функции, и выводит это слово одним цветом. С другой стороны, `"Hello Python world!"` не является кодом Python, поэтому этот текст выделяется другим цветом. Этот механизм, называемый *цветовым выделением синтаксиса*, сильно поможет вам, когда вы возьметесь за самостоятельное программирование.

Переменные

Попробуем использовать переменную в программе hello_world.py. Добавьте новую строку в начало файла и измените вторую строку:

hello_world.py

```
message = "Hello Python world!"
print(message)
```

Запустите программу и посмотрите, что получится. Программа выводит уже знакомый результат:

```
Hello Python world!
```

В программу добавилась *переменная* с именем `message`. В каждой переменной хранится *значение*, то есть данные, связанные с переменной. В нашем случае значением является текст `"Hello Python world!"`.

Добавление переменной немного усложняет задачу интерпретатора Python. Во время обработки первой строки он связывает текст `"Hello Python world!"` с переменной `message`. А когда интерпретатор доберется до второй строки, он выводит на экран значение, связанное с именем `message`.

Давайте немного расширим эту программу hello_world.py, чтобы она выводила второе сообщение. Добавьте в hello_world.py пустую строку, а после нее еще две строки кода:

```
message = "Hello Python world!"
print(message)

message = "Hello Python Crash Course world!"
print(message)
```

Теперь при выполнении hello_world.py на экране должны появляться две строки:

```
Hello Python world!
Hello Python Crash Course world!
```

Вы можете в любой момент изменить значение переменной в своей программе; Python постоянно отслеживает его текущее состояние.

Выбор имен и использование переменных

При работе с переменными в языке Python необходимо соблюдать некоторые правила и рекомендации. Нарушение правил приведет к ошибке; рекомендации всего лишь помогают писать более понятный и удобочитаемый код. Работая с переменными, помните о следующем:

- Имена переменных могут состоять только из букв, цифр и символов подчеркивания. Они могут начинаться с буквы или символа подчеркивания, но не с цифры. Например, переменной можно присвоить имя `message_1`, но не `1_message`.
- Пробелы в именах переменных запрещены, а для разделения слов в именах переменных используются символы подчеркивания. Например, имя `greeting_message` допустимо, а имя `greeting message` вызовет ошибку.

- Не используйте имена функций и ключевые слова Python в качестве имен переменных; иначе говоря, не используйте слова, которые зарезервированы в Python для конкретной цели, — например, слово `print` (см. раздел «Ключевые слова и встроенные функции Python»).
- Имена переменных должны быть короткими, но содержательными. Например, имя `name` лучше `n`, имя `student_name` лучше `s_n`, а имя `name_length` лучше `length_of_persons_name`.
- Будьте внимательны при использовании строчной буквы `l` и прописной буквы `O`, потому что они похожи на цифры 1 и 0.

Вероятно, вы не сразу научитесь создавать хорошие имена переменных, особенно когда программы станут сложнее и интереснее. Но когда вы начнете писать свои программы и читать код, написанный другими разработчиками, ваши имена переменных станут более содержательными.

ПРИМЕЧАНИЕ Пока ограничьтесь именами переменных, записанными в нижнем регистре. Использование символов верхнего регистра не приведет к ошибке, но такие символы имеют специальное значение в именах переменных, и пока все же лучше обойтись без них.

Предотвращение ошибок в именах при использовании переменных

Каждый программист совершает ошибки, а большинство программистов совершает ошибки ежедневно. И хотя даже опытный программист не застрахован от ошибок, он знает, как эффективно реагировать на них. Рассмотрим типичную ошибку, которую вы довольно часто будете совершать на первых порах, и выясним, как эту ошибку исправить.

Для начала напишем код с намеренно внесенной ошибкой. Введите следующий фрагмент (неправильно написанное слово *mesage* выделено жирным шрифтом):

```
message = "Hello Python Crash Course reader!"
print(mesage)
```

Когда в программе происходит ошибка, интерпретатор Python всеми силами старается помочь вам в поиске причины. Если программа не выполняется нормально, интерпретатор предоставляет данные *трассировки* — информацию о том, в каком месте кода находился интерпретатор при возникновении проблем. Ниже приведен пример трассировки, которую выдает Python после случайной опечатки в имени переменной:

```
  Traceback (most recent call last):
❶   File "hello_world.py", line 2, in <module>
❷     print(mesage)
❸ NameError: name 'mesage' is not defined
```

Строка ❶ сообщает, что ошибка произошла в строке 2 файла hello_world.py. Интерпретатор выводит номер строки, чтобы вам было проще найти ошибку ❷, и сообщает тип обнаруженной ошибки ❸. В данном случае была обнаружена ошибка в имени: переменная с указанным именем (`mesage`) не определена. Другими словами, Python не распознает имя переменной. Обычно такие ошибки возникают в том случае, если вы забыли присвоить значение переменной перед ее использованием или ошиблись при вводе имени.

Конечно, в данном примере в имени переменной во второй строке пропущена буква s. Интерпретатор Python не проверяет код на наличие опечаток, но следит за тем, чтобы имена переменных записывались одинаково. Например, вот что происходит, если имя `message` будет неправильно записано еще в одном месте кода:

```
mesage = "Hello Python Crash Course reader!"
print(mesage)
```

На этот раз программа выполняется успешно!

```
Hello Python Crash Course reader!
```

Языки программирования не отличаются гибкостью, но орфография их совершенно не волнует. Как следствие, вам не нужно следить за тем, чтобы в именах переменных идеально соблюдались правила орфографии английского языка.

Многие ошибки программирования сводятся к простым опечаткам — случайной замене одного символа в одной строке программы. Если вы потратили много времени на поиск одной из таких ошибок, знайте, что вы не одиноки. Многие опытные и талантливые программисты тратят долгие часы на поиск подобных мелких ошибок. Нечто подобное будет часто происходить в ходе вашей работы — просто посмейтесь и идите дальше.

Переменные как метки

Переменные часто описывают как «ящики» для хранения значений. Такое сравнение может быть полезным на первых порах работы с переменными, но оно неточно описывает внутреннее представление переменных в Python. Намного правильнее представлять переменные как метки, которые можно назначать переменным. Также можно сказать, что переменная *содержит ссылку* на некоторое значение.

Вероятно, это различие ни на что не повлияет в ваших первых программах. И все же лучше узнать о нем раньше, чем позже. В какой-то момент вы столкнетесь с неожиданным поведением переменных, и более точное понимание работы переменных поможет вам разобраться в том, что же происходит в вашем коде.

ПРИМЕЧАНИЕ Как лучше всего освоить новые концепции программирования? Попытайтесь использовать их в своей программе. Если в ходе работы над упражнением вы зайдете в тупик, попробуйте на какое-то время заняться чем-нибудь другим. Если это не поможет, перечитайте соответствующую часть этой главы. Если и это не помогло, обращайтесь к рекомендациям из приложения B.

УПРАЖНЕНИЯ

Напишите отдельную программу для выполнения каждого из следующих упражнений. Сохраните каждую программу в файле, имя которого подчиняется стандартным правилам Python по использованию строчных букв и символов подчеркивания — например, simple_message.py и simple_messages.py.

2.1. Простое сообщение: сохраните текстовое сообщение в переменной и выведите его на экран.

2.2. Простые сообщения: сохраните сообщение в переменной и выведите это сообщение. Затем замените значение переменной другим сообщением и выведите новое сообщение.

Строки

Так как многие программы определяют и собирают некие данные, а затем делают с ними что-то полезное, желательно выделить основные разновидности данных. Начнем со строковых данных. На первый взгляд строки достаточно просты, но с ними можно работать многими разными способами.

Строка представляет собой простую последовательность символов. Любая последовательность символов, заключенная в кавычки, в Python считается строкой; при этом строки могут быть заключены как в одиночные, так и в двойные кавычки:

```
"This is a string."
'This is also a string.'
```

Это правило позволяет использовать внутренние кавычки и апострофы в строках:

```
'I told my friend, "Python is my favorite language!"'
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive community."
```

Рассмотрим некоторые типичные операции со строками.

Изменение регистра символов в строках

Одна из простейших операций, выполняемых со строками, — изменение регистра символов. Взгляните на следующий фрагмент кода и попробуйте определить, что в нем происходит:

name.py

```
name = "ada lovelace"
print(name.title())
```

Сохраните файл с именем name.py и запустите его. Вывод программы должен выглядеть так:

```
Ada Lovelace
```

В этом примере в переменной `name` сохраняется строка, состоящая из букв нижнего регистра `"ada lovelace"`. За именем переменной в команде `print()` следует вызов метода `title()`. *Метод* представляет собой действие, которое Python выполняет с данными. Точка (`.`) после `name` в конструкции `name.title()` приказывает Python применить метод `title()` к переменной `name`. За именем метода всегда следует пара круглых скобок, потому что методам для выполнения их работы часто требуется дополнительная информация. Эта информация указывается в скобках. Функции `title()` дополнительная информация не нужна, поэтому в круглых скобках ничего нет.

Метод `title()` преобразует первый символ каждого слова в строке к верхнему регистру, тогда как все остальные символы выводятся в нижнем регистре. Например, данная возможность может быть полезна, если в вашей программе входные значения `Ada`, `ADA` и `ada` должны рассматриваться как одно и то же имя и все они должны отображаться в виде `Ada`.

Для работы с регистром также существуют другие полезные методы. Например, все символы строки можно преобразовать к верхнему или нижнему регистру:

```
name = "Ada Lovelace"
print(name.upper())
print(name.lower())
```

Программа выводит следующий результат:

```
ADA LOVELACE
ada lovelace
```

Метод `lower()` особенно полезен для хранения данных. Нередко программист не может рассчитывать на то, что пользователи введут все данные с точным соблюдением регистра, поэтому строки перед сохранением преобразуются к нижнему регистру. Затем, когда потребуется вывести информацию, используется регистр, наиболее подходящий для каждой строки.

Использование переменных в строках

В некоторых ситуациях требуется использовать значения переменных внутри строки. Представьте, что имя и фамилия хранятся в разных переменных и вы хотите объединить их для вывода полного имени:

full_name.py

```
  first_name = "ada"
  last_name = "lovelace"
❶ full_name = f"{first_name} {last_name}"
  print(full_name)
```

Чтобы вставить значение переменной в строку, поставьте букву `f` непосредственно перед открывающей кавычкой ❶. Заключите имя (или имена) переменных, кото-

рые должны использоваться внутри строки, в фигурные скобки. Python заменит каждую переменную ее значением при выводе строки.

Такие строки называются *f-строками*. Буква *f* происходит от слова «format», потому что Python форматирует строку, заменяя имена переменных в фигурных скобках их значениями. Приведенный код выводит следующий результат:

```
ada lovelace
```

С f-строками можно сделать много интересного. Например, с их помощью можно строить сложные сообщения с информацией, хранящейся в переменных. Рассмотрим пример:

```
  first_name = "ada"
  last_name = "lovelace"
  full_name = f"{first_name} {last_name}"
❶ print(f"Hello, {full_name.title()}!")
```

Полное имя используется в точке ❶ для вывода приветственного сообщения, а метод `title()` обеспечивает правильное форматирование имени. Этот фрагмент возвращает простое, хорошо отформатированное сообщение:

```
Hello, Ada Lovelace!
```

F-строками также можно воспользоваться для построения сообщения, которое затем сохраняется в переменной:

```
  first_name = "ada"
  last_name = "lovelace"
  full_name = f"{first_name} {last_name}"
❶ message = f"Hello, {full_name.title()}!"
❷ print(message)
```

Этот код также выводит сообщение «Hello, Ada Lovelace!», но сохранение текста сообщения в переменной ❶ существенно упрощает завершающую команду печати ❷.

ПРИМЕЧАНИЕ F-строки впервые появились в Python 3.6. Если вы используете Python 3.5 или более раннюю версию, используйте метод `format()` вместо синтаксиса f-строк. Чтобы использовать `format()`, перечислите переменные, которые должны использоваться, в строке, в круглых скобках после `format`. Каждая переменная обозначается парой фигурных скобок; эти позиции будут заполняться значениями, перечисленными в круглых скобках в указанном порядке:

```
full_name = "{} {}".format(first_name, last_name)
```

Табуляции и разрывы строк

В программировании термином *пропуск* (whitespace) называются такие непечатаемые символы, как пробелы, табуляции и символы конца строки. Пропуски структурируют текст, чтобы пользователю было удобнее читать его.

Для включения в текст позиции табуляции используется комбинация символов `\t`, как в точке ❶:

```
  >>> print("Python")
  Python
❶ >>> print("\tPython")
      Python
```

Разрывы строк добавляются с помощью комбинации символов `\n`:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

Табуляции и разрывы строк могут сочетаться в тексте. Скажем, последовательность `"\n\t"` приказывает Python начать текст с новой строки, в начале которой располагается табуляция. Следующий пример демонстрирует вывод одного сообщения с разбиением на четыре строки:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

Разрывы строк и табуляции часто встречаются в двух следующих главах, когда наши программы начнут выводить относительно длинный текст.

Удаление пропусков

Лишние пропуски могут вызвать путаницу в программах. Для программиста строки `'python'` и `'python '` внешне неотличимы, но для программы это совершенно разные строки. Python видит лишний пробел в `'python '` и считает, что он действительно важен — до тех пор, пока вы не сообщите о противоположном.

Обращайте внимание на пропуски, потому что в программах часто приходится сравнивать строки, чтобы проверить на совпадение их содержимое. Типичный пример — проверка имен пользователей при входе на сайт. Лишние пропуски могут создавать путаницу и в более простых ситуациях. К счастью, Python позволяет легко удалить лишние пропуски из данных, введенных пользователем.

Python может искать лишние пропуски у левого и правого края строки. Чтобы убедиться в том, что у правого края (в конце) строки нет пропусков, вызовите метод `rstrip()`.

```
❶ >>> favorite_language = 'python '
❷ >>> favorite_language
  'python '
❸ >>> favorite_language.rstrip()
  'python'
❹ >>> favorite_language
  'python '
```

Значение, хранящееся в переменной `favorite_language` в точке ❶, содержит лишние пропуски в конце строки. Когда вы приказываете Python вывести это значение в терминальном сеансе, вы видите пробел в конце значения ❷. Когда метод `rstrip()` работает с переменной `favorite_language` в точке ❸, этот лишний пробел удаляется. Впрочем, удаление лишь временное — если снова запросить значение `favorite_language`, мы видим, что строка не отличается от исходной, включая лишний пропуск ❹.

Чтобы навсегда исключить пропуск из строки, следует записать усеченное значение обратно в переменную:

```
  >>> favorite_language = 'python '
❶ >>> favorite_language = favorite_language.rstrip()
  >>> favorite_language
  'python'
```

Сначала пропуски удаляются в конце строки, а потом значение записывается в исходную переменную ❶. Операция изменения значения переменной с последующим его сохранением в исходной переменной часто выполняется в программировании. Так, значение переменной может изменяться в ходе выполнения программы или в ответ на действия пользователя.

Пропуски также можно удалить у левого края (в начале) строки при помощи метода `lstrip()`, а метод `strip()` удаляет пропуски с обоих концов:

```
❶ >>> favorite_language = ' python '
❷ >>> favorite_language.rstrip()
  ' python'
❸ >>> favorite_language.lstrip()
  'python '
❹ >>> favorite_language.strip()
  'python'
```

В этом примере исходное значение содержит пропуски в начале и в конце ❶. Затем пропуски удаляются у правого края ❷, у левого края ❸ и с обоих концов строки ❹. Поэкспериментируйте с функциями удаления пропусков, это поможет вам освоиться с работой со строками. На практике эти функции чаще всего применяются для «очистки» пользовательского ввода перед его сохранением в программе.

Предотвращение синтаксических ошибок в строках

Синтаксические ошибки встречаются в программах относительно регулярно. *Синтаксическая ошибка* происходит тогда, когда Python не распознает часть вашей программы как действительный код. Например, если заключить апостроф в одиночные кавычки, произойдет ошибка. Это происходит из-за того, что Python интерпретирует все символы от первой одиночной кавычки до апострофа как строку. После этого Python пытается интерпретировать остаток текста строки как код Python, что порождает ошибки.

Разберемся, как же правильно использовать одиночные или двойные кавычки. Сохраните следующую программу в файле apostrophe.py и запустите ее:

apostrophe.py

```
message = "One of Python's strengths is its diverse community."
print(message)
```

Апостроф находится в строке, заключенной в двойные кавычки, так что у интерпретатора Python не возникает проблем с правильной интерпретацией следующей строки:

```
One of Python's strengths is its diverse community.
```

Однако при использовании одиночных кавычек Python не сможет определить, где должна заканчиваться строка:

```
message = 'One of Python's strengths is its diverse community.'
print(message)
```

Программа выводит следующий результат:

```
  File "apostrophe.py", line 1
    message = 'One of Python's strengths is its diverse community.'
                             ^❶
SyntaxError: invalid syntax
```

Из выходных данных видно, что ошибка происходит в позиции ❶ сразу же после второй одиночной кавычки. Эта синтаксическая ошибка указывает, что интерпретатор не распознает какую-то конструкцию как действительный код Python. Ошибки могут возникать по разным причинам; я буду выделять наиболее распространенные источники по мере того, как они будут встречаться нам. Синтаксические ошибки будут часто встречаться вам в то время, пока вы учитесь писать правильный код Python. Кроме того, ошибки этой категории также являются наиболее расплывчатыми и неконкретными, поэтому их особенно трудно находить и исправлять. Если вы зайдете в тупик из-за особенно коварной ошибки, обращайтесь к рекомендациям в приложении B.

ПРИМЕЧАНИЕ Функция цветового выделения синтаксиса ускоряет выявление некоторых синтаксических ошибок прямо во время написания программы. Если вы увидите, что код Python выделяется как обычный текст (или обычный текст выделяется как код Python), скорее всего, в вашем файле где-то пропущена кавычка.

УПРАЖНЕНИЯ

Сохраните код каждого из следующих упражнений в отдельном файле с именем вида name_cases.py. Если у вас возникнут проблемы, сделайте перерыв или обратитесь к рекомендациям в приложении B.

2.3. Личное сообщение: сохраните имя пользователя в переменной и выведите сообщение, предназначенное для конкретного человека. Сообщение должно быть простым — например, «Hello Eric, would you like to learn some Python today?”.

2.4. Регистр символов в именах: сохраните имя пользователя в переменной и выведите его в нижнем регистре, в верхнем регистре и с капитализацией начальных букв каждого слова.

2.5. Знаменитая цитата: найдите известное высказывание, которое вам понравилось. Выведите текст цитаты с именем автора. Результат должен выглядеть примерно так (включая кавычки):

> *Albert Einstein once said, "A person who never made*
> *a mistake never tried anything new."*

2.6. Знаменитая цитата 2: повторите упражнение 2.5, но на этот раз сохраните имя автора цитаты в переменной `famous_person`. Затем составьте сообщение и сохраните его в новой переменной с именем message. Выведите свое сообщение.

2.7. Удаление пропусков: сохраните имя пользователя в переменной. Добавьте в начале и в конце имени несколько пропусков. Проследите за тем, чтобы каждая служебная последовательность , `"\t"` и `"\n"`, встречалась по крайней мере один раз.

Выведите имя, чтобы были видны пропуски в начале и конце строки. Затем выведите его снова с использованием каждой из функций удаления пропусков: `lstrip()`, `rstrip()` и `strip()`.

Числа

Числа очень часто применяются в программировании для ведения счета в играх, представления данных в визуализациях, хранения информации в веб-приложениях и т. д. В Python числовые данные делятся на несколько категорий в соответствии со способом их использования. Для начала посмотрим, как Python работает с целыми числами, потому что с ними возникает меньше всего проблем.

Целые числа

В Python с целыми числами можно выполнять операции сложения (`+`), вычитания (`-`), умножения (`*`) и деления(`/`).

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

В терминальном сеансе Python просто возвращает результат операции. Для представления операции возведения в степень в Python используется сдвоенный знак умножения:

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

В Python также существует определенный порядок операций, что позволяет использовать несколько операций в одном выражении. Круглые скобки используются для изменения следования операций, чтобы выражение могло вычисляться в нужном порядке. Пример:

```
>>> 2 + 3*4
14
>>> (2 + 3) * 4
20
```

Пробелы в этих примерах не влияют на то, как Python вычисляет выражения; они просто помогают быстрее найти приоритетные операции при чтении кода.

Вещественные числа

В Python числа, имеющие дробную часть, называются *вещественными* (или «числами с плавающей точкой»). Обычно разработчик может просто пользоваться дробными значениями, не особенно задумываясь об их поведении. Просто введите нужные числа, а Python, скорее всего, сделает именно то, что вы от него хотите:

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

Однако в некоторых ситуациях вдруг оказывается, что результат содержит неожиданно большое количество разрядов в дробной части:

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

Нечто подобное может произойти в любом языке; для беспокойства нет причин. Python пытается подобрать как можно более точное представление результата, что иногда бывает нелегко из-за особенностей внутреннего представления чисел в компьютерах. Пока просто не обращайте внимания на «лишние» разряды; вы узнаете, как поступать в подобных ситуациях, когда эта проблема станет актуальной для вас в проектах части II.

Целые и вещественные числа

При делении двух любых чисел — даже если это целые числа, частным от деления которых является целое число, — вы всегда получаете вещественное число:

```
>>> 4/2
2.0
```

При смешении целого и вещественного числа в любой другой операции вы также получаете вещественное число:

```
>>> 1 + 2.0
3.0
>>> 2 * 3.0
6.0
>>> 3.0 ** 2
9.0
```

Python по умолчанию использует вещественный тип для результата любой операции, в которой задействовано вещественное число, даже если результат является целым числом.

Символы подчеркивания в числах

В записи целых чисел можно группировать цифры при помощи символов подчеркивания, чтобы числа лучше читались:

```
>>> universe_age = 14_000_000_000
```

При выводе числа, определяемого с символами подчеркивания, Python выводит только цифры:

```
>>> print(universe_age)
14000000000
```

Python игнорирует символы подчеркивания при хранении таких значений. Даже если цифры не группируются в тройках, это никак не повлияет на значение. С точки зрения Python `1000` ничем не отличается от записи `1_000`, которая эквивалентна `10_00`. Этот вариант записи работает как для целых, так и для вещественных чисел, но он доступен только в Python 3.6 и выше.

Множественное присваивание

В одной строке программы можно присвоить значения сразу нескольким переменным. Этот синтаксис сократит длину программы и упростит ее чтение; чаще всего он применяется при инициализации наборов чисел.

Например, следующая строка инициализирует переменные `x`, `y` и `z` нулями:

```
>>> x, y, z = 0, 0, 0
```

Имена переменных должны разделяться запятыми; точно так же должны разделяться значения. Python присваивает каждое значение переменной в соответствующей позиции. Если количество значений соответствует количеству переменных, Python правильно сопоставит их друг с другом.

Константы

Константа представляет собой переменную, значение которой остается неизменным на протяжении всего срока жизни программы. В Python нет встроенных типов констант, но у программистов Python принято записывать имена переменных, которые должны рассматриваться как константы и оставаться неизменными, буквами верхнего регистра:

```
MAX_CONNECTIONS = 5000
```

Если вы собираетесь работать с переменной в коде как с константой, не забудьте записать ее имя буквами верхнего регистра.

УПРАЖНЕНИЯ

2.8. Число 8: напишите операции сложения, вычитания, умножения и деления, результатом которых является число `8`. Не забудьте заключить операции в команды `print()`, чтобы проверить результат. Вы должны написать четыре строки кода, которые выглядят примерно так:

```
print(5 + 3)
```

Результатом должны быть четыре строки, в каждой из которых выводится число `8`.

2.9. Любимое число: сохраните свое любимое число в переменной. Затем при помощи переменной создайте сообщение для вывода этого числа. Выведите это сообщение.

Комментарии

Комментарии чрезвычайно полезны в любом языке программирования. До сих пор ваши программы состояли только из кода Python. По мере роста объема и сложности кода в программы следует добавлять *комментарии*, описывающие общий подход к решаемой задаче, — своего рода заметки, написанные на понятном языке.

Как создаются комментарии?

В языке Python признаком комментария является символ «решетка» (`#`). Интерпретатор Python игнорирует все символы, следующие в коде после `#` до конца строки. Пример:

comment.py

```
# Say hello to everyone.
print("Hello Python people!")
```

Python игнорирует первую строку и выполняет вторую.

```
Hello Python people!
```

Какие комментарии следует писать?

Главная задача комментария — объяснить, что должен делать ваш код и как он работает. В разгаре работы над проектом вы понимаете, как работают все его компоненты. Но если вернуться к проекту спустя некоторое время, скорее всего, некоторые подробности будут забыты. Конечно, всегда можно изучить код и разобраться в том, как должны работать его части, но хорошие комментарии с доступным изложением общих принципов работы кода сэкономят немало времени.

Если вы хотите стать профессиональным программистом или участвовать в совместной работе с другими программистами, научитесь писать осмысленные комментарии. В наши дни почти все программы разрабатываются коллективно в группах — либо группами работников одной компании, либо группами энтузиастов, совместно работающих над проектом с открытым кодом. Опытные программисты ожидают увидеть комментарии в коде, поэтому лучше привыкайте добавлять содержательные комментарии прямо сейчас. Написание простых, лаконичных комментариев — одна из самых полезных привычек, необходимых начинающему программисту.

Принимая решение о том, нужно ли писать комментарий или нет, спросите себя, пришлось ли вам перебрать несколько вариантов в поисках разумного решения для некоторой задачи; если ответ будет положительным, напишите комментарий по поводу вашего решения. Удалить лишние комментарии позднее намного проще, чем возвращаться и добавлять комментарии в программу. С этого момента я буду использовать комментарии в примерах для пояснения смысла некоторых частей кода.

УПРАЖНЕНИЯ

2.10. Добавление комментариев: выберите две программы из написанных вами и добавьте в каждую хотя бы один комментарий. Если вы не найдете, что написать в комментариях, потому что программы были слишком просты, добавьте свое имя и текущую дату в начало кода. Затем добавьте одно предложение с описанием того, что делает программа.

Дзен Python

Опытные программисты Python рекомендуют избегать лишних сложностей и по возможности выбирать простые решения там, где это возможно. Философия сообщества Python выражена в очерке Тима Питерса «The Zen of Python». Чтобы просмотреть этот краткий набор принципов написания хорошего кода Python, достаточно ввести команду `import this` в интерпретаторе. Я не стану воспроизводить все принципы, но приведу несколько строк, чтобы вы поняли, почему они важны для вас как для начинающего программиста Python.

```
>>> import this
The Zen of Python, by Tim Peters
Красивое лучше, чем уродливое.
```

Программисты Python считают, что код может быть красивым и элегантным. В программировании люди занимаются решением задач. Программисты всегда ценили хорошо спроектированные, эффективные и даже красивые решения. Со временем вы больше узнаете о Python, начнете писать больше кода — и когда-нибудь ваш коллега посмотрит на экран вашего компьютера и скажет: «Ого, какой красивый код!»

```
Простое лучше, чем сложное.
```

Если у вас есть выбор между простым и сложным решением и оба работают, используйте простое решение. Код будет проще в сопровождении, а у вас и других разработчиков будет меньше проблем с обновлением этого кода в будущем.

```
Сложное лучше, чем запутанное.
```

Реальность создает свои сложности; иногда простое решение задачи невозможно. В таком случае используйте самое простое решение, которое работает.

```
Удобочитаемость имеет значение.
```

Даже если ваш код сложен, постарайтесь сделать так, чтобы он нормально читался. Работая над проектом, требующим написания сложного кода, постарайтесь написать содержательные комментарии для этого кода.

```
Должен существовать один — и желательно только один — очевидный способ сделать это.
```

Если предложить двум программистам Python решить одну и ту же задачу, они должны выработать похожие решения. Это не значит, что в программировании нет места для творчества. Наоборот! Но большая часть работы программиста заключается в применении небольших стандартных решений для простых ситуаций в контексте большого, более творческого проекта. Внутренняя организация ваших программ должна выглядеть логично с точки зрения других программистов Python.

```
Сейчас лучше, чем никогда.
```

Вы можете потратить весь остаток жизни на изучение всех тонкостей Python и программирования в целом, но в таком случае вы никогда не закончите ни одного проекта. Не пытайтесь написать идеальный код; напишите код, который работает, а потом решите, стоит ли доработать его для текущего проекта или перейти на что-то другое.

Когда вы перейдете к следующей главе и займетесь изучением более сложных тем, постарайтесь не забывать об этой философии простоты и ясности. Опытные программисты будут с большим уважением относиться к вашему коду, более охотно поделятся своим мнением и скорее будут сотрудничать с вами в интересных проектах.

УПРАЖНЕНИЯ

2.11. Дзен Python: введите команду `import this` в терминальном сеансе Python и просмотрите другие принципы.

Итоги

В этой главе вы научились работать с переменными. Вы узнали, как использовать содержательные имена переменных и как исправлять ошибки в именах и синтаксические ошибки в случае их возникновения. Вы узнали, что такое строки и как выводить их в нижнем/верхнем регистре и с капитализацией всех слов. Мы рассмотрели способы аккуратного оформления вывода с применением пропусков, а также удаления лишних пропусков из разных частей строки. Вы начали работать с целыми и вещественными числами и узнали о некоторых неожиданностях, встречающихся при работе с числовыми данными. Вы научились писать содержательные комментарии, которые упрощают написание кода для вас и его чтение для других разработчиков. В завершение главы была представлена философия максимальной простоты кода.

В главе 3 рассматривается хранение наборов данных в переменных, называемых *списками*. Вы узнаете, как перебрать содержимое списка и обработать хранящуюся в нем информацию.

3 Списки

В этой и следующей главе вы узнаете, что собой представляют списки и как начать работать с элементами списка. Списки позволяют хранить в одном месте взаимосвязанные данные, сколько бы их ни было — несколько элементов или несколько миллионов элементов. Работа со списками принадлежит к числу самых выдающихся возможностей Python, доступных для начинающего программиста. Операции со списками связывают воедино многие важные концепции в программировании.

Что такое список?

Список представляет собой набор элементов, следующих в определенном порядке. Вы можете создать список для хранения букв алфавита, цифр от 0 до 9 или имен всех членов вашей семьи. В список можно поместить любую информацию, причем данные в списке даже не обязаны быть как-то связаны друг с другом. Так как список обычно содержит более одного элемента, рекомендуется присваивать спискам имена во множественном числе: `letters`, `digits`, `names` и т. д.

В языке Python список обозначается квадратными скобками (`[]`), а отдельные элементы списка разделяются запятыми. Простой пример списка с названиями моделей велосипедов:

bicycles.py

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

Если вы прикажете Python вывести список, то на экране появится перечисление элементов списка в квадратных скобках:

```
['trek', 'cannondale', 'redline', 'specialized']
```

Конечно, вашим пользователям такое представление не подойдет; разберемся, как обратиться отдельным элементам в списке.

Обращение к элементам списка

Списки представляют собой упорядоченные наборы данных, поэтому для обращения к любому элементу списка следует сообщить Python позицию (*индекс*) нужного элемента. Чтобы обратиться к элементу в списке, укажите имя списка, за которым следует индекс элемента в квадратных скобках.

Например, название первого велосипеда в списке `bicycles` выводится следующим образом:

```
  bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ print(bicycles[0])
```

Синтаксис обращения к элементу показан в точке ❶. Когда мы запрашиваем один элемент из списка, Python возвращает только этот элемент без квадратных скобок или кавычек:

```
trek
```

Именно такой результат должны увидеть пользователи — чистый, аккуратно отформатированный вывод.

Также можно использовать строковые методы из главы 2 с любым элементом списка. Например, элемент `'trek'` можно более аккуратно отформатировать при помощи метода `title()`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0].title())
```

Этот пример выдает такой же результат, как и предыдущий, только название `'Trek'` выводится с прописной буквы.

Индексы начинаются с 0, а не с 1

Python считает, что первый элемент списка находится в позиции 0, а не в позиции 1. Этот принцип встречается в большинстве языков программирования и объясняется особенностями низкоуровневой реализации операций со списками. Если вы получаете неожиданные результаты, определите, не допустили ли вы простую ошибку «смещения на 1».

Второму элементу списка соответствует индекс 1. В этой простой схеме индекс любого элемента вычисляется уменьшением на 1 его позиции в списке. Например, чтобы обратиться к четвертому элементу списка, следует запросить элемент с индексом 3.

В следующем примере выводятся названия велосипедов с индексами 1 и 3:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])
```

При этом выводятся второй и четвертый элементы списка:

```
cannondale
specialized
```

В Python также существует специальный синтаксис для обращения к последнему элементу списка. Если запросить элемент с индексом –1, Python всегда возвращает последний элемент в списке:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

Фрагмент вернет значение `'specialized'`. Этот синтаксис весьма полезен, потому что при работе со списками часто требуется обратиться к последним элементам, не зная точного количества элементов в списке. Синтаксис также распространяется на другие отрицательные значения индексов. Индекс `-2` возвращает второй элемент от конца списка, индекс `-3` — третий элемент от конца, и т. д.

Использование отдельных элементов из списка

Отдельные значения из списка используются так же, как и любые другие переменные. Например, вы можете воспользоваться f-строками для построения сообщения, содержащего значение из списка.

Попробуем извлечь название первого велосипеда из списка и составить сообщение, включающее это значение.

```
  bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ message = f"My first bicycle was a {bicycles[0].title()}."

  print(message)
```

В точке ❶ программа строит сообщение, содержащее значение из `bicycles[0]`, и сохраняет его в переменной `message`. Так создается простое предложение с упоминанием первого велосипеда из списка:

```
My first bicycle was a Trek.
```

УПРАЖНЕНИЯ

Попробуйте написать несколько коротких программ, чтобы получить предварительное представление о списках Python. Возможно, для упражнений каждой главы стоит создать отдельную папку, чтобы избежать неразберихи.

3.1. Имена: сохраните имена нескольких своих друзей в списке с именем names. Выведите имя каждого друга, обратившись к каждому элементу списка (по одному за раз).

3.2. Сообщения: начните со списка, использованного в упражнении 3.1, но вместо вывода имени каждого человека выведите сообщение. Основной текст всех сообщений должен быть одинаковым, но каждое сообщение должно включать имя адресата.

3.3. Собственный список: выберите свой любимый вид транспорта (например, мотоциклы или машины) и создайте список с примерами. Используйте свой список для вывода утверждений об элементах типа: «Я хотел бы купить мотоцикл Honda».

Изменение, добавление и удаление элементов

Как правило, вы будете создавать *динамические* списки; это означает, что во время выполнения программы в созданном вами списке будут добавляться и удаляться элементы. Например, вы можете создать игру, в которой игрок должен стрелять по кораблям космических захватчиков. Исходный набор кораблей сохраняется в списке; каждый раз, когда вы сбиваете корабль, он удаляется из списка. Каждый раз, когда на экране появляется новый враг, он включается в список. Длина списка кораблей будет уменьшаться и увеличиваться по ходу игры.

Изменение элементов в списке

Синтаксис изменения элемента напоминает синтаксис обращения к элементу списка. Чтобы изменить элемент, укажите имя списка и индекс изменяемого элемента в квадратных скобках; далее задайте новое значение, которое должно быть присвоено элементу.

Допустим, имеется список мотоциклов и первым элементом списка хранится строка `'honda'`. Как изменить значение первого элемента?

motorcycles.py

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

❷ motorcycles[0] = 'ducati'
  print(motorcycles)
```

В точке ❶ определяется исходный список, в котором первый элемент содержит строку `'honda'`. В точке ❷ значение первого элемента заменяется строкой `'ducati'`. Из вывода видно, что первый элемент действительно изменился, тогда как остальные элементы списка сохранили прежние значения:

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

Изменить можно значение любого элемента в списке, не только первого.

Добавление элементов в список

Новые элементы могут добавляться в списки по разным причинам: например, для появления на экране новых космических кораблей, включения новых данных в визуализацию или добавления новых зарегистрированных пользователей на по-

строенный вами сайт. Python предоставляет несколько способов добавления новых данных в существующие списки.

Присоединение элементов в конец списка

Простейший способ добавления новых элементов в список — *присоединение* элемента в конец списка. Используя список из предыдущего примера, добавим новый элемент `'ducati'` в конец списка:

```
  motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)
❶ motorcycles.append('ducati')
  print(motorcycles)
```

Метод `append()` в точке ❶ присоединяет строку `'ducati'` в конец списка, другие элементы в списке при этом остаются неизменными:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

Метод `append()` упрощает динамическое построение списков. Например, вы можете начать с пустого списка и добавлять в него элементы серией команд `append()`. В следующем примере в пустой список добавляются элементы `'honda'`, `'yamaha'` и `'suzuki'`:

```
motorcycles = []

motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

Полученный список выглядит точно так же, как и списки из предыдущих примеров:

```
['honda', 'yamaha', 'suzuki']
```

Такой способ построения списков встречается очень часто, потому что данные, которые пользователь захочет сохранить в программе, часто становятся известными только после запуска программы. Чтобы пользователь мог управлять содержимым списка, начните с определения пустого списка, а затем присоединяйте к нему каждое новое значение.

Вставка элементов в список

Метод `insert()` позволяет добавить новый элемент в произвольную позицию списка. Для этого следует указать индекс и значение нового элемента.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

❶ motorcycles.insert(0, 'ducati')
  print(motorcycles)
```

В этом примере в точке ❶ значение `'ducati'` вставляется в начало списка. Метод `insert()` выделяет свободное место в позиции 0 и сохраняет в нем значение `'ducati'`. Все остальные значения списка при этом сдвигаются на одну позицию вправо:

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

Удаление элементов из списка

Нередко возникает необходимость в удалении одного или нескольких элементов из списка. Например, когда игрок сбивает корабль пришельца, этот корабль стоит удалить из списка активных врагов. Или когда пользователь решает удалить свою учетную запись в созданном вами веб-приложении, этот пользователь должен быть удален из списка активных пользователей. Элементы удаляются из списка по позиции или по значению.

Удаление элемента с использованием команды del

Если вам известна позиция элемента, который должен быть удален из списка, воспользуйтесь командой `del`.

```
  motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

❶ del motorcycles[0]
  print(motorcycles)
```

В точке ❶ вызов `del` удаляет первый элемент, `'honda'`, из списка `motorcycles`:

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

Команда `del` позволяет удалить элемент из любой позиции списка, если вам известен его индекс. Например, вот как из списка удаляется второй элемент, `'yamaha'`:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[1]
print(motorcycles)
```

Второй элемент исчез из списка:

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

В обоих примерах значение, удаленное из списка после использования команды `del`, становится недоступным.

Удаление элемента с использованием метода pop()

Иногда значение, удаляемое из списка, должно как-то использоваться. Допустим, вы хотите получить координаты *x* и *y* только что сбитого корабля пришельцев, чтобы изобразить взрыв в этой позиции. В веб-приложении пользователь, удаленный из списка активных участников, может быть добавлен в список неактивных, и т. д.

Метод `pop()` удаляет последний элемент из списка, но позволяет работать с ним после удаления. Удалим мотоцикл из списка:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

❷ popped_motorcycle = motorcycles.pop()
❸ print(motorcycles)
❹ print(popped_motorcycle)
```

Сначала в точке ❶ определяется и выводится содержимое списка `motorcycles`. В точке ❷ значение извлекается из списка и сохраняется в переменной с именем `popped_motorcycle`. Вывод измененного списка в точке ❸ показывает, что значение было удалено из списка. Затем мы выводим извлеченное значение в точке ❹, демонстрируя, что удаленное из списка значение остается доступным в программе.

Из вывода видно, что значение `'suzuki'`, удаленное в конце списка, теперь хранится в переменной `popped_motorcycle`:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

Для чего может понадобиться метод `pop()`? Представьте, что мотоциклы в списке хранятся в хронологическом порядке в соответствии с датой их покупки. В таком случае команда `pop()` может использоваться для вывода сообщения о последнем купленном мотоцикле:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print(f"The last motorcycle I owned was a {last_owned.title()}.")
```

Программа выводит простое сообщение:

```
The last motorcycle I owned was a Suzuki.
```

Извлечение элементов из произвольной позиции списка

Вызов `pop()` может использоваться для удаления элемента в произвольной позиции списка; для этого следует указать индекс удаляемого элемента в круглых скобках.

```
  motorcycles = ['honda', 'yamaha', 'suzuki']

❶ first_owned = motorcycles.pop(0)
❷ print(f"The first motorcycle I owned was a {first_owned.title()}.")
```

Сначала первый элемент извлекается из списка в точке ❶, а затем в точке ❷ выводится сообщение об этом мотоцикле. Программа выводит простое сообщение о первом мотоцикле:

```
The first motorcycle I owned was a Honda.
```

Помните, что после каждого вызова `pop()` элемент, с которым вы работаете, уже не находится в списке.

Если вы не уверены в том, какой из двух способов выбрать, команду `del` или метод `pop()`, — простое правило поможет вам определиться: если вы собираетесь просто удалить элемент из списка, никак не используя его, выбирайте команду `del`; если же вы намерены использовать элемент после удаления из списка, выбирайте метод `pop()`.

Удаление элементов по значению

Иногда позиция удаляемого элемента неизвестна. Если вы знаете только значение элемента, используйте метод `remove()`.

Допустим, из списка нужно удалить значение `'ducati'`:

```
  motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
  print(motorcycles)

❶ motorcycles.remove('ducati')
  print(motorcycles)
```

Код в точке ❶ приказывает Python определить, в какой позиции списка находится значение `'ducati'`, и удалить этот элемент:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

Метод `remove()` также может использоваться для работы со значением, которое удаляется из списка. Следующая программа удаляет значение `'ducati'` и выводит причину удаления:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
  print(motorcycles)

❷ too_expensive = 'ducati'
❸ motorcycles.remove(too_expensive)
  print(motorcycles)
❹ print(f"\nA {too_expensive.title()} is too expensive for me.")
```

После определения списка в точке ❶ значение `'ducati'` сохраняется в переменной с именем `too_expensive` в точке ❷. Затем эта переменная сообщает Python, какое значение должно быть удалено из списка ❸. В точке ❹ значение `'ducati'` было удалено из списка, но продолжает храниться в переменной `too_expensive`,

что позволяет вывести сообщение с причиной удаления `'ducati'` из списка мотоциклов:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']

A Ducati is too expensive for me.
```

ПРИМЕЧАНИЕ Метод `remove()` удаляет только первое вхождение заданного значения. Если существует вероятность того, что значение встречается в списке более одного раза, используйте цикл для определения того, были ли удалены все вхождения данного значения. О том, как это делать, рассказано в главе 7.

УПРАЖНЕНИЯ

Следующие упражнения немного сложнее упражнений из главы 2, но они предоставляют возможность попрактиковаться в выполнении всех описанных операций со списками.

3.4. Список гостей: если бы вы могли пригласить кого угодно (из живых или умерших) на обед, то кого бы вы пригласили? Создайте список, включающий минимум трех людей, которых вам хотелось бы пригласить на обед. Затем используйте этот список для вывода пригласительного сообщения каждому участнику.

3.5. Изменение списка гостей: вы только что узнали, что один из гостей прийти не сможет, поэтому вам придется разослать новые приглашения. Отсутствующего гостя нужно заменить кем-то другим.

- Начните с программы из упражнения 3.4. Добавьте в конец программы команду `print` для вывода имени гостя, который прийти не сможет.
- Измените список и замените имя гостя, который прийти не сможет, именем нового приглашенного.
- Выведите новый набор сообщений с приглашениями — по одному для каждого участника, входящего в список.

3.6. Больше гостей: вы решили купить обеденный стол большего размера. Дополнительные места позволяют пригласить на обед еще трех гостей.

- Начните с программы из упражнения 3.4 или 3.5. Добавьте в конец программы команду `print`, которая выводит сообщение о расширении списка гостей.
- Добавьте вызов `insert()` для добавления одного гостя в начало списка.
- Добавьте вызов `insert()` для добавления одного гостя в середину списка.
- Добавьте вызов `append()` для добавления одного гостя в конец списка.
- Выведите новый набор сообщений с приглашениями — по одному для каждого участника, входящего в список.

3.7. Сокращение списка гостей: только что выяснилось, что новый обеденный стол привезти вовремя не успеют и места хватит только для двух гостей.

- Начните с программы из упражнения 3.6. Добавьте команду для вывода сообщения о том, что на обед приглашаются всего два гостя.
- Используйте метод `pop()` для последовательного удаления гостей из списка до тех пор, пока в списке не останутся только два человека. Каждый раз, когда из списка удаляется очередное имя, выведите для этого человека сообщение о том, что вы сожалеете об отмене приглашения.

- Выведите сообщение для каждого из двух человек, остающихся в списке. Сообщение должно подтверждать, что более раннее приглашение остается в силе.
- Используйте команду `del` для удаления двух последних имен, чтобы список остался пустым. Выведите список, чтобы убедиться в том, что в конце работы программы список действительно не содержит ни одного элемента.

Упорядочение списка

Нередко список создается в непредсказуемом порядке, потому что порядок получения данных от пользователя не всегда находится под вашим контролем. И хотя во многих случаях такое положение дел неизбежно, часто требуется вывести информацию в определенном порядке. В одних случаях требуется сохранить исходный порядок элементов в списке, в других исходный порядок должен быть изменен. Python предоставляет в распоряжение программиста несколько разных способов упорядочения списка в зависимости от ситуации.

Постоянная сортировка списка методом sort()

Метод `sort()` позволяет относительно легко отсортировать список. Предположим, имеется список машин и вы хотите переупорядочить эти элементы по алфавиту. Чтобы упростить задачу, предположим, что все значения в списке состоят из символов нижнего регистра.

cars.py

```
  cars = ['bmw', 'audi', 'toyota', 'subaru']
❶ cars.sort()
  print(cars)
```

Метод `sort()` в точке ❶ осуществляет постоянное изменение порядка элементов в списке. Названия машин располагаются в алфавитном порядке, и вернуться к исходному порядку уже не удастся:

```
['audi', 'bmw', 'subaru', 'toyota']
```

Список также можно отсортировать в обратном алфавитном порядке; для этого методу `sort()` следует передать аргумент `reverse=True`. В следующем примере список сортируется в порядке, обратном алфавитному:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True)
print(cars)
```

И снова порядок элементов изменяется на постоянной основе:

```
['toyota', 'subaru', 'bmw', 'audi']
```

Временная сортировка списка функцией sorted()

Чтобы сохранить исходный порядок элементов списка, но временно представить их в отсортированном порядке, можно воспользоваться функцией `sorted()`. Функция `sorted()` позволяет представить список в определенном порядке, но не изменяет фактический порядок элементов в списке.

Попробуем применить эту функцию к списку машин.

```
  cars = ['bmw', 'audi', 'toyota', 'subaru']

❶ print("Here is the original list:")
  print(cars)

❷ print("\nHere is the sorted list:")
  print(sorted(cars))

❸ print("\nHere is the original list again:")
  print(cars)
```

Сначала список выводится в исходном порядке ❶, а затем в алфавитном порядке ❷. После того как список будет выведен в новом порядке, в точке ❸ мы убеждаемся в том, что список все еще хранится в исходном порядке.

```
  Here is the original list:
  ['bmw', 'audi', 'toyota', 'subaru']

  Here is the sorted list:
  ['audi', 'bmw', 'subaru', 'toyota']

❹ Here is the original list again:
  ['bmw', 'audi', 'toyota', 'subaru']
```

Обратите внимание: после вызова функции `sorted()` список продолжает храниться в исходном порядке ❹. Функции `sorted()` также можно передать аргумент `reverse=True`, чтобы список был представлен в порядке, обратном алфавитному.

ПРИМЕЧАНИЕ Если не все значения записаны в нижнем регистре, алфавитная сортировка списка немного усложняется. При определении порядка сортировки появляются разные способы интерпретации прописных букв, и точное определение порядка уже не столь тривиально (во всяком случае, чтобы отвлекаться на него сейчас). Впрочем, большинство способов сортировки напрямую следует из того, о чем вы узнали в этом разделе.

Вывод списка в обратном порядке

Чтобы переставить элементы списка в обратном порядке, используйте метод `reverse()`. Скажем, если список машин первоначально хранился в хронологическом порядке даты приобретения, элементы можно легко переупорядочить в обратном хронологическом порядке:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)

cars.reverse()
print(cars)
```

Обратите внимание: метод `reverse()` не сортирует элементы в обратном алфавитном порядке, а просто переходит к обратному порядку списка:

```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
```

Метод `reverse()` осуществляет постоянное изменение порядка элементов, но вы можете легко вернуться к исходному порядку, снова применив `reverse()` к обратному списку.

Определение длины списка

Вы можете быстро определить длину списка с помощью функции `len()`. Список в нашем примере состоит из четырех элементов, поэтому его длина равна 4:

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
4
```

Метод `len()` может пригодиться для определения количества пришельцев, которых необходимо сбить в игре; объема данных, которыми необходимо управлять в визуализации; количества зарегистрированных пользователей на веб-сайте и т. д.

ПРИМЕЧАНИЕ Python подсчитывает элементы списка, начиная с 1, поэтому при определении длины списка ошибок «смещения на 1» уже быть не должно.

УПРАЖНЕНИЯ

3.8. Повидать мир: вспомните хотя бы пять стран, в которых вам хотелось бы побывать.

- Сохраните названия стран в списке. Проследите за тем, чтобы список не хранился в алфавитном порядке.
- Выведите список в исходном порядке. Не беспокойтесь об оформлении, просто выведите его как обычный список Python.
- Используйте функцию `sorted()` для вывода списка в алфавитном порядке без изменения списка.
- Снова выведите список, чтобы показать, что он по-прежнему хранится в исходном порядке.
- Используйте функцию `sorted()` для вывода списка в обратном алфавитном порядке без изменения порядка исходного списка.
- Снова выведите список, чтобы показать, что исходный порядок не изменился.
- Измените порядок элементов вызовом `reverse()`. Выведите список, чтобы показать, что элементы следуют в другом порядке.

- Измените порядок элементов повторным вызовом `reverse()`. Выведите список, чтобы показать, что список вернулся к исходному порядку.
- Отсортируйте список в алфавитном порядке вызовом `sort()`. Выведите список, чтобы показать, что элементы следуют в другом порядке.
- Вызовите `sort()` для перестановки элементов списка в обратном алфавитном порядке. Выведите список, чтобы показать, что порядок элементов изменился.

3.9. Количество гостей: в одной из программ из упражнений с 3.4 по 3.7 используйте `len()` для вывода сообщения с количеством людей, приглашенных на обед.

3.10. Все функции: придумайте информацию, которую можно было бы хранить в списке. Например, создайте список гор, рек, стран, городов, языков... словом, чего угодно. Напишите программу, которая создает список элементов, а затем вызывает каждую функцию, упоминавшуюся в этой главе, хотя бы один раз.

Ошибки индексирования при работе со списками

Когда программист только начинает работать со списками, он часто допускает одну характерную ошибку. Допустим, имеется список с тремя элементами и программа запрашивает четвертый элемент:

motorcycles.py

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[3])
```

В этом случае происходит *ошибка индексирования*:

```
Traceback (most recent call last):
  File "motorcycles.py", line 3, in <module>
    print(motorcycles[3])
IndexError: list index out of range
```

Python пытается вернуть элемент с индексом 3. Однако при поиске по списку ни один элемент `motorcycles` не обладает индексом 3. Из-за смещения индексов на 1 эта ошибка весьма распространена. Люди думают, что третьим элементом является элемент с индексом 3, потому что они начинают отсчет с 1. Но для Python третьим является элемент с индексом 2, потому что индексирование начинается с 0.

Ошибка индексирования означает, что Python не может понять, какой индекс запрашивается в программе. Если в вашей программе происходит ошибка индексирования, попробуйте уменьшить запрашиваемый индекс на 1. Затем снова запустите программу и проверьте правильность результатов.

Помните, что для обращения к последнему элементу в списке используется индекс –1. Этот способ работает всегда, даже если размер списка изменился с момента последнего обращения к нему:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1])
```

Индекс –1 всегда возвращает последний элемент списка, в данном случае значение `'suzuki'`:

```
'suzuki'
```

Этот синтаксис порождает ошибку только в одном случае — при попытке получить последний элемент пустого списка:

```
motorcycles = []
print(motorcycles[-1])
```

В списке `motorcycles` нет ни одного элемента, поэтому Python снова выдает ошибку индексирования:

```
Traceback (most recent call last):
  File "motorcyles.py", line 3, in <module>
    print(motorcycles[-1])
IndexError: list index out of range
```

ПРИМЕЧАНИЕ Если в вашей программе произошла ошибка индексирования и вы не знаете, как с ней справиться, попробуйте вывести список или хотя бы его длину. Возможно, ваш список выглядит совсем не так, как вы думаете, особенно если его содержимое динамически определялось программой. Фактическое состояние списка или точное количество элементов в нем поможет вам выявить логические ошибки такого рода.

УПРАЖНЕНИЯ

3.11. Намеренная ошибка: если ни в одной из предшествующих программ вы еще не сталкивались с ошибками индексирования, попробуйте создать такую ошибку искусственно. Измените индекс в одной из программ, чтобы вызвать ошибку индексирования. Не забудьте исправить ошибку перед тем, как закрывать программу.

Итоги

В этой главе вы узнали, что собой представляют списки и как работать с отдельными элементами в списках. Вы научились определять списки, добавлять и удалять элементы, выполнять сортировку (постоянную или временную для отображения). Также вы узнали, как определить длину списка и как избежать ошибок индексирования при работе со списком.

В главе 4 рассматриваются приемы более эффективной работы со списками. Перебор всех элементов списка всего в нескольких строках кода, даже если список содержит тысячи или миллионы элементов, сокращает объем программы.

4 Работа со списками

В главе 3 вы научились создавать простые списки и работать с отдельными элементами списков. В этой главе вы узнаете, как *перебрать* весь список в нескольких строках кода (независимо от длины списка). Механизм перебора позволяет выполнить одно действие или набор действий с каждым элементом в списке. С его помощью вы сможете эффективно работать со списками любой длины, даже состоящими из тысяч и миллионов элементов.

Перебор всего списка

Типичная задача из области программирования — перебрать все элементы списка и выполнить с каждым элементом одну и ту же операцию. Например, в компьютерной игре все экранные объекты могут смещаться на одинаковую величину или в списке чисел к каждому элементу может применяться одна и та же статистическая операция. А может быть, вам потребовалось вывести все заголовки из списка статей на сайте. В ситуациях, требующих применения одного действия к каждому элементу списка, можно воспользоваться циклами `for`.

Допустим, имеется список с именами фокусников и вы хотите вывести каждое имя из списка. Конечно, можно обратиться к каждому элементу по отдельности, но такой подход создает ряд проблем. Во-первых, для очень длинных списков все сведется к однообразным повторениям. Во-вторых, при любом изменении длины списка в программу придется вносить изменения. Цикл `for` решает обе проблемы: Python будет следить за всеми техническими деталями в своей внутренней реализации.

В следующем примере цикл `for` используется для вывода имен фокусников:

magicians.py

```
❶ magicians = ['alice', 'david', 'carolina']
❷ for magician in magicians:
❸     print(magician)
```

Все начинается с определения списка ❶, как и в главе 3. В точке ❷ определяется цикл `for`. Эта строка приказывает Python взять очередное имя из списка и сохранить его в переменной `magician`. В точке ❸ выводится имя, только что сохранен-

ное в переменной magician. Затем строки ❷ и ❸ повторяются для каждого имени в списке. Этот код можно описать так: «Для каждого фокусника в списке вывести его имя». Результат представляет собой простой перечень имен из списка:

```
alice
david
carolina
```

Подробнее о циклах

Концепция циклов очень важна, потому что она представляет один из основных способов автоматизации повторяющихся задач компьютером. Например, в простом цикле, использованном в magicians.py, Python сначала читает первую строку цикла:

```
for magician in magicians:
```

Эта строка означает, что нужно взять первое значение из списка magicians и сохранить его в переменной magician. Первое значение в списке — 'alice'. Затем Python читает следующую строку:

```
    print(magician)
```

Python выводит текущее значение magician, которое все еще равно 'alice'. Так как в списке еще остались другие значения, Python возвращается к первой строке цикла:

```
for magician in magicians:
```

Python берет следующее значение из списка, 'david', и сохраняет его в magician. Затем выполняется строка:

```
    print(magician)
```

Python снова выводит текущее значение magician; теперь это строка 'david'. Весь цикл повторяется еще раз с последним значением в списке, 'carolina'. Так как других значений в списке не осталось, Python переходит к следующей строке в программе. В данном случае после цикла for ничего нет, поэтому программа просто завершается.

Помните, что все действия повторяются по одному разу для каждого элемента в списке независимо от их количества. Если список содержит миллион элементов, Python повторит эти действия миллион раз — обычно это происходит очень быстро.

Также помните, что при написании собственных циклов for временной переменной для текущего значения из списка можно присвоить любое имя. Однако на практике рекомендуется выбирать осмысленное имя, описывающее отдельный элемент списка. Несколько примеров:

```
for cat in cats:
for dog in dogs:
for item in list_of_items:
```

Выполнение этого правила поможет вам проследить за тем, какие действия выполняются с каждым элементом в цикле `for`. В зависимости от того, какое число используется — одиночное или множественное, вы сможете понять, работает ли данная часть кода с отдельным элементом из списка или со всем списком.

Более сложные действия в циклах for

В цикле `for` с каждым элементом списка может выполняться практически любое действие. Дополним предыдущий пример, чтобы программа выводила для каждого фокусника отдельное сообщение:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
❶     print(f"{magician.title()}, that was a great trick!")
```

Единственное отличие этого кода от предыдущего заключается в том, что в точке ❶ для каждого фокусника строится сообщение с его именем. При первом проходе цикла переменная `magician` содержит значение `'alice'`, поэтому Python начинает первое сообщение с имени `'Alice'`. При втором проходе сообщение будет начинаться с имени `'David'`, а при третьем — с имени `'Carolina'`:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
```

Тело цикла `for` может содержать сколько угодно строк кода. Каждая строка с начальным отступом после строки `for magician in magicians` считается находящейся *в цикле* и выполняется по одному разу для каждого значения в списке. Таким образом, с каждым значением в списке можно выполнить любые операции на ваше усмотрение.

Включим в сообщение для каждого фокусника вторую строку:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
❶     print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

Так как оба вызова `print()` снабжены отступами, каждая строка будет выполнена по одному разу для каждого фокусника в списке. Символ новой строки (`"\n"`) во второй команде print ❶ вставляет пустую строку после каждого прохода цикла. В результате будет создан набор сообщений, аккуратно сгруппированных для каждого фокусника в списке:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.
```

```
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Тело цикла `for` может содержать сколько угодно строк кода. На практике часто требуется выполнить в цикле `for` несколько разных операций для каждого элемента списка.

Выполнение действий после цикла for

Что происходит после завершения цикла `for`? Обычно программа выводит некую сводную информацию или переходит к другим операциям.

Каждая строка кода после цикла `for`, не имеющая отступа, выполняется без повторения. Допустим, вы хотите вывести сообщение для всей группы фокусников и поблагодарить их за превосходное представление. Чтобы вывести общее сообщение после всех отдельных сообщений, разместите его после цикла `for` без отступа:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")

❶ print("Thank you, everyone. That was a great magic show!")
```

Первые две команды `print` повторяются по одному разу для каждого фокусника в списке, как было показано ранее. Но поскольку строка ❶ отступа не имеет, это сообщение выводится только один раз:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!
```

При обработке данных в циклах `for` завершающее сообщение позволяет подвести итог операции, выполненной со всем набором данных. Например, цикл `for` может инициализировать игру, перебирая список персонажей и изображая каждого персонажа на экране. После цикла выполняется блок без отступа, который выводит кнопку Начать игру, после того как все персонажи появятся на экране.

Предотвращение ошибок с отступами

В Python связь одной строки кода с предшествующей строкой обозначается отступами. В приведенных примерах строки, выводившие сообщения для отдельных фокусников, были частью цикла, потому что они были снабжены отступами. Применение отступов в Python сильно упрощает чтение кода. Фактически отступы

заставляют разработчика писать аккуратно отформатированный код с четкой визуальной структурой. В более длинных программах Python могут встречаться блоки кода с отступами нескольких разных уровней. Эти уровни способствуют пониманию общей структуры программы.

Когда разработчики только начинают писать код, работа которого зависит от правильности отступов, в их коде нередко встречаются распространенные ошибки. Например, иногда они расставляют отступы в коде, в котором эти отступы не нужны, или наоборот — забывают ставить отступы в блоках, в которых это необходимо. Несколько примеров помогут вам избежать подобных ошибок в будущем и успешно исправлять их, когда они встретятся в ваших программах.

Итак, рассмотрим несколько типичных ошибок при использовании отступов.

Пропущенный отступ

Строка после команды `for` в цикле всегда должна снабжаться отступом. Если вы забудете поставить отступ, Python напомнит вам об этом:

magicians.py

```
  magicians = ['alice', 'david', 'carolina']
  for magician in magicians:
❶ print(magician)
```

Команда `print` в точке ❶ должна иметь отступ, но здесь его нет. Когда Python ожидает увидеть блок с отступом, но не находит его, появляется сообщение с указанием номера строки:

```
  File "magicians.py", line 3
    print(magician)
        ^
IndentationError: expected an indented block
```

Обычно для устранения подобных ошибок достаточно поставить отступ в строке (или строках), следующей непосредственно после команды `for`.

Пропущенные отступы в других строках

Иногда цикл выполняется без ошибок, но не выдает ожидаемых результатов. Такое часто происходит, когда вы пытаетесь выполнить несколько операций в цикле, но забываете снабдить отступом некоторые из строк.

Например, вот что происходит, если вы забудете снабдить отступом вторую строку в цикле:

```
  magicians = ['alice', 'david', 'carolina']
  for magician in magicians:
      print(magician.title() + ", that was a great trick!")
❶ print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

Команда `print` в точке ❶ должна быть снабжена отступом, но поскольку Python находит хотя бы одну строку с отступом после команды `for`, сообщение об ошибке не выдается. В результате первая команда `print` будет выполнена для каждого элемента в списке, потому что в ней есть отступ. Вторая команда `print` отступа не имеет, поэтому она будет выполнена только один раз после завершения цикла. Так как последним значением `magician` является строка `'carolina'`, второе сообщение будет выведено только с этим именем:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Это пример *логической ошибки*. Код имеет действительный синтаксис, но он не приводит к желаемому результату, потому что проблема кроется в его логике. Если некоторое действие должно повторяться для каждого элемента в списке, но выполняется только один раз, проверьте, не нужно ли добавить отступы в строке или нескольких строках кода.

Лишние отступы

Если вы случайно поставите отступ в строке, в которой он не нужен, Python сообщит об этом:

hello_world.py

```
  message = "Hello Python world!"
❶     print(message)
```

Отступ команды `print` в точке ❶ не нужен, потому что эта строка не подчинена предшествующей; Python сообщает об ошибке:

```
  File "hello_world.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

Чтобы избежать непредвиденных ошибок с отступами, используйте их только там, где для этого существуют конкретные причины. В тех программах, которые вы пишете на этой стадии изучения Python, отступы нужны только в строках действий, повторяемых для каждого элемента в цикле `for`.

Лишние отступы после цикла

Если вы случайно снабдите отступом код, который должен выполняться *после* завершения цикла, то этот код будет выполнен для каждого элемента. Иногда Python выводит сообщение об ошибке, но часто дело ограничивается простой логической ошибкой.

Например, что произойдет, если случайно снабдить отступом строку с выводом завершающего приветствия для группы фокусников?

magicians.py

```
  magicians = ['alice', 'david', 'carolina']
  for magician in magicians:
      print(f"{magician.title()}, that was a great trick!")
      print(f"I can't wait to see your next trick, {magician.title()}.\n")

❶     print("Thank you everyone, that was a great magic show!")
```

Так как строка ❶ имеет отступ, сообщение будет продублировано для каждого фокусника в списке:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

Thank you everyone, that was a great magic show!
David, that was a great trick!
I can't wait to see your next trick, David.

Thank you everyone, that was a great magic show!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you everyone, that was a great magic show!
```

Это еще один пример логической ошибки наподобие описанной в разделе «Пропущенные отступы в других строках». Python не знает, что вы пытаетесь сделать в своем коде, поэтому он просто выполняет весь код, не нарушающий правил синтаксиса. Если действие, которое должно выполняться один раз, выполняется многократно, проверьте, нет ли лишнего отступа в соответствующей строке кода.

Пропущенное двоеточие

Двоеточие в конце команды `for` сообщает Python, что следующая строка является началом цикла.

```
  magicians = ['alice', 'david', 'carolina']
❶ for magician in magicians
      print(magician)
```

Если вы случайно забудете поставить двоеточие, как в примере ❶, произойдет синтаксическая ошибка, так как полученная команда нарушает правила языка. И хотя такие ошибки легко исправляются, найти их бывает достаточно трудно. Вы не поверите, сколько времени тратят программисты на поиск подобных «односимвольных» ошибок. Поиск таких ошибок усложняется еще и тем, что человек обычно склонен видеть то, что он ожидает увидеть.

УПРАЖНЕНИЯ

4.1. Пицца: вспомните по крайней мере три названия ваших любимых видов пиццы. Сохраните их в списке и используйте цикл `for` для вывода всех названий.

- Измените цикл for так, чтобы вместо простого названия пиццы выводилось сообщение, включающее это название. Таким образом, для каждого элемента должна выводиться строка с простым текстом, например: «I like pepperoni pizza».
- Добавьте в конец программы (после цикла for) строку с завершающим сообщением. Таким образом, вывод должен состоять из трех (и более) строк с названиями пиццы и дополнительного сообщения — скажем, «I really love pizza!».

4.2. Животные: создайте список из трех (и более) животных, обладающих общей характеристикой. Используйте цикл for для вывода названий всех животных.

- Измените программу так, чтобы вместо простого названия выводилось сообщение, включающее это название, — например, «A dog would make a great pet».
- Добавьте в конец программы строку с описанием общего свойства. Например, можно вывести сообщение «Any of these animals would make a great pet!».

Создание числовых списков

Необходимость хранения наборов чисел возникает в программах по многим причинам. Например, в компьютерной игре могут храниться координаты каждого персонажа на экране, таблицы рекордов и т. д. В программах визуализации данных пользователь почти всегда работает с наборами чисел: температурой, расстоянием, численностью населения, широтой/долготой и другими числовыми данными.

Списки идеально подходят для хранения наборов чисел, а Python предоставляет специальные средства для эффективной работы с числовыми списками. Достаточно один раз понять, как эффективно пользоваться этими средствами, и ваш код будет хорошо работать даже в том случае, если список содержит миллионы элементов.

Функция range()

Функция `range()` упрощает построение числовых последовательностей. Например, с ее помощью можно легко вывести серию чисел:

first_numbers.py

```
for value in range(1,5):
    print(value)
```

И хотя на первый взгляд может показаться, что он должен вывести числа от 1 до 5, на самом деле число 5 не выводится:

```
1
2
3
4
```

В этом примере `range()` выводит только числа от 1 до 4. Перед вами еще один пример явления «смещения на 1», часто встречающегося в языках программирования. При выполнении функции `range()` Python начинает отсчет от первого переданного значения и прекращает его при достижении второго. Так как на втором значении происходит остановка, конец интервала (5 в данном случае) не встречается в выводе.

Чтобы вывести числа от 1 до 5, используйте вызов `range(1,6)`:

```
for value in range(1,6):
    print(value)
```

На этот раз вывод начинается с 1 и завершается 5:

```
1
2
3
4
5
```

Если ваша программа при использовании `range()` выводит не тот результат, на который вы рассчитывали, попробуйте увеличить конечное значение на 1.

При вызове `range()` также можно передать только один аргумент; в этом случае последовательность чисел будет начинаться с 0. Например, `range(6)` вернет числа от 0 до 5.

Использование range() для создания числового списка

Если вы хотите создать числовой список, преобразуйте результаты `range()` в список при помощи функции `list()`. Если заключить вызов `range()` в `list()`, то результат будет представлять собой список с числовыми элементами.

В примере из предыдущего раздела числовая последовательность просто выводилась на экран. Тот же набор чисел можно преобразовать в список вызовом `list()`:

```
numbers = list(range(1,6))
print(numbers)
```

Результат:

```
[1, 2, 3, 4, 5]
```

Функция `range()` также может генерировать числовые последовательности, пропуская числа в заданном диапазоне. Например, построение списка четных чисел от 1 до 10 происходит так:

even_numbers.py

```
even_numbers = list(range(2,11,2))
print(even_numbers)
```

В этом примере функция `range()` начинает со значения 2, а затем увеличивает его на 2. Приращение 2 последовательно применяется до тех пор, пока не будет достигнуто или пройдено конечное значение 11, после чего выводится результат:

```
[2, 4, 6, 8, 10]
```

С помощью функции `range()` можно создать практически любой диапазон чисел. Например, как бы вы создали список квадратов всех целых чисел от 1 до 10? В языке Python операция возведения в степень обозначается двумя звездочками (**). Один из возможных вариантов выглядит так:

squares.py

```
❶ squares = []
❷ for value in range(1,11):
❸     square = value**2
❹     squares.append(square)

❺ print(squares)
```

Сначала в точке ❶ создается пустой список с именем `squares`. В точке ❷ вы приказываете Python перебрать все значения от 1 до 10 при помощи функции `range()`. В цикле текущее значение возводится во вторую степень, а результат сохраняется в переменной `square` в точке ❸. В точке ❹ каждое новое значение `square` присоединяется к списку `squares`. Наконец, после завершения цикла список квадратов выводится в точке ❺:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Чтобы сделать код более компактным, можно опустить временную переменную `square` и присоединять каждое новое значение прямо к списку:

```
  squares = []
  for value in range(1,11):
❶     squares.append(value**2)

  print(squares)
```

Конструкция ❶ выполняет ту же работу, что и строки ❸ и ❹ в squares.py. Каждое значение в цикле возводится во вторую степень, а затем немедленно присоединяется к списку квадратов.

При создании более сложных списков можно использовать любой из двух подходов. Иногда использование временной переменной упрощает чтение кода; в других случаях оно приводит лишь к напрасному удлинению кода. Сначала сосредоточьтесь на написании четкого и понятного кода, который делает именно то, что нужно, и только потом переходите к анализу кода и поиску более эффективных решений.

Простая статистика с числовыми списками

Некоторые функции Python предназначены для работы с числовыми списками. Например, вы можете легко узнать минимум, максимум и сумму числового списка:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

ПРИМЕЧАНИЕ В примерах этой главы используются короткие списки чисел, но это делается только для того, чтобы данные помещались на странице. Примеры также будут работать и в том случае, если список содержит миллионы чисел.

Генераторы списков

Описанный выше способ генерирования списка `squares` состоял из трех или четырех строк кода. *Генератор списка* (list comprehension) позволяет сгенерировать тот же список всего в одной строке. Генератор списка объединяет цикл `for` и создание новых элементов в одну строку и автоматически присоединяет к списку все новые элементы. Учебники не всегда рассказывают о генераторах списка начинающим программистам, но я привожу этот материал, потому что вы с большой вероятностью встретите эту конструкцию, как только начнете просматривать код других разработчиков.

В следующем примере список квадратов, знакомый вам по предыдущим примерам, строится с использованием генератора списка:

squares.py

```
squares = [value**2 for value in range(1,11)]
print(squares)
```

Чтобы использовать этот синтаксис, начните с содержательного имени списка, например `squares`. Затем откройте квадратные скобки и определите выражение для значений, которые должны быть сохранены в новом списке. В данном примере это выражение `value**2`, которое возводит значение во вторую степень. Затем напишите цикл `for` для генерирования чисел, которые должны передаваться выражению, и закройте квадратные скобки. Цикл `for` в данном примере — `for value in range(1,11)` — передает значения с 1 до 10 выражению `value**2`. Обратите внимание на отсутствие двоеточия в конце команды `for`.

Результатом будет уже знакомый вам список квадратов:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Чтобы успешно писать собственные генераторы списков, необходим определенный опыт. Тем не менее, как только вы освоитесь с созданием обычных

списков, вы оцените возможности генераторов. Когда после очередного трех-четырехстрочного блока вам это надоест, подумайте о написании собственных генераторов списков.

УПРАЖНЕНИЯ

4.3. Считаем до 20: используйте цикл `for` для вывода чисел от 1 до 20 включительно.

4.4. Миллион: создайте список чисел от 1 до 1 000 000, затем воспользуйтесь циклом for для вывода чисел. (Если вывод занимает слишком много времени, остановите его нажатием Ctrl+C или закройте окно вывода.)

4.5. Суммирование миллиона чисел: создайте список чисел от 1 до 1 000 000, затем воспользуйтесь функциями `min()` и `max()` и убедитесь в том, что список действительно начинается с 1 и заканчивается 1 000 000. Вызовите функцию sum() и посмотрите, насколько быстро Python сможет просуммировать миллион чисел.

4.6. Нечетные числа: воспользуйтесь третьим аргументом функции range() для создания списка нечетных чисел от 1 до 20. Выведите все числа в цикле for.

4.7. Тройки: создайте список чисел, кратных 3, в диапазоне от 3 до 30. Выведите все числа своего списка в цикле `for`.

4.8. Кубы: результат возведения числа в третью степень называется кубом. Например, куб 2 записывается в языке Python в виде `2**3`. Создайте список первых 10 кубов (то есть кубов всех целых чисел от 1 до 10) и выведите значения всех кубов в цикле for.

4.9. Генератор кубов: используйте конструкцию генератора списка для создания списка первых 10 кубов.

Работа с частью списка

В главе 3 вы узнали, как обращаться к отдельным элементам списка, а в этой главе мы занимались перебором всех элементов списка. Также можно работать с конкретным подмножеством элементов списка; в Python такие подмножества называются *сегментами* (slices).

Создание сегмента

Чтобы создать сегмент на основе списка, следует задать индексы первого и последнего элементов, с которыми вы намереваетесь работать. Как и в случае с функцией `range()`, Python останавливается на элементе, предшествующем второму индексу. Скажем, чтобы вывести первые три элемента списка, запросите индексы с 0 по 3, и вы получите элементы 0, 1 и 2.

В следующем примере используется список игроков команды:

players.py

```
  players = ['charles', 'martina', 'michael', 'florence', 'eli']
❶ print(players[0:3])
```

В точке ❶ выводится сегмент, включающий только первых трех игроков. Вывод сохраняет структуру списка, но включает только первых трех игроков:

```
['charles', 'martina', 'michael']
```

Подмножество может включать любую часть списка. Например, чтобы ограничиться вторым, третьим и четвертым элементами списка, создайте сегмент, который начинается с индекса 1 и заканчивается на индексе 4:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

На этот раз сегмент начинается с элемента `'martina'` и заканчивается элементом `'florence'`:

```
['martina', 'michael', 'florence']
```

Если первый индекс сегмента не указан, то Python автоматически начинает сегмент от начала списка:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

Без начального индекса Python берет элементы от начала списка:

```
['charles', 'martina', 'michael', 'florence']
```

Аналогичный синтаксис работает и для сегментов, включающих конец списка. Например, если вам нужны все элементы с третьего до последнего, начните с индекса 2 и не указывайте второй индекс:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

Python возвращает все элементы с третьего до конца списка:

```
['michael', 'florence', 'eli']
```

Этот синтаксис позволяет вывести все элементы от любой позиции до конца списка независимо от его длины. Вспомните, что отрицательный индекс возвращает элемент, находящийся на заданном расстоянии от конца списка; следовательно, вы можете получить любой сегмент от конца списка. Например, чтобы отобрать последних трех игроков из списка, используйте сегмент `players[-3:]`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

Программа выводит имена трех последних игроков, причем продолжает работать даже при изменении размера списка.

ПРИМЕЧАНИЕ В квадратные скобки, определяющие сегмент, также можно включить третье значение. Это значение, если оно присутствует, сообщает Python, сколько элементов следует пропускать при выборе элементов в заданном диапазоне.

Перебор содержимого сегмента

Если вы хотите перебрать элементы, входящие в подмножество элементов, используйте сегмент в цикле `for`. В следующем примере программа перебирает первых трех игроков и выводит их имена:

```
  players = ['charles', 'martina', 'michael', 'florence', 'eli']

  print("Here are the first three players on my team:")
❶ for player in players[:3]:
      print(player.title())
```

Вместо того чтобы перебирать весь список игроков ❶, Python ограничивается первыми тремя именами:

```
Here are the first three players on my team:
Charles
Martina
Michael
```

Сегменты приносят огромную пользу во многих ситуациях. Например, при создании компьютерной игры итоговый счет игрока может добавляться в список после окончания текущей партии. После этого программа может получить три лучших результата игрока, отсортировав список по уменьшению и получив сегмент, включающий только три элемента. При работе с данными сегменты могут использоваться для обработки данных блоками заданного размера. Или при построении веб-приложения сегменты могут использоваться для постраничного вывода информации так, чтобы на каждой странице выводился соответствующий объем информации.

Копирование списка

Часто разработчик берет существующий список и создает на его основе совершенно новый список. Посмотрим, как работает копирование списков, и рассмотрим одну ситуацию, в которой копирование списка может принести пользу.

Чтобы скопировать список, создайте сегмент, включающий весь исходный список без указания первого и второго индекса (`[:]`). Эта конструкция создает сегмент, который начинается с первого элемента и завершается последним; в результате создается копия всего списка.

Представьте, что вы создали список своих любимых блюд и теперь хотите создать отдельный список блюд, которые нравятся вашему другу. Пока вашему другу

нравятся все блюда из нашего списка, поэтому вы можете создать другой список простым копированием нашего:

foods.py

```
❶ my_foods = ['pizza', 'falafel', 'carrot cake']
❷ friend_foods = my_foods[:]

  print("My favorite foods are:")
  print(my_foods)

  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

В точке ❶ создается список блюд с именем `my_foods`. В точке ❷ создается другой список с именем `friend_foods`. Чтобы создать копию `my_foods`, программа запрашивает сегмент `my_foods` без указания индексов и сохраняет копию в `friend_foods`. При выводе обоих списков становится видно, что оба списка содержат одинаковые данные:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']
```

Чтобы доказать, что речь в действительности идет о двух разных списках, добавим новое блюдо в каждый список:

```
  my_foods = ['pizza', 'falafel', 'carrot cake']
❶ friend_foods = my_foods[:]

❷ my_foods.append('cannoli')
❸ friend_foods.append('ice cream')

  print("My favorite foods are:")
  print(my_foods)

  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

В точке ❶ исходные элементы `my_foods` копируются в новый список `friend_foods`, как было сделано в предыдущем примере. Затем в ❷ каждый список добавляется новый элемент: `'cannoli'` в `my_foods`, а в точке ❸ `'ice cream'` добавляется в `friend_foods`. После этого вывод двух списков наглядно показывает, что каждое блюдо находится в соответствующем списке.

```
  My favorite foods are:
❹ ['pizza', 'falafel', 'carrot cake', 'cannoli']

  My friend's favorite foods are:
❺ ['pizza', 'falafel', 'carrot cake', 'ice cream']
```

Вывод в точке ❹ показывает, что элемент `'cannoli'` находится в списке `my_foods`, а элемент `'ice cream'` в этот список не входит. В точке ❺ видно, что `'ice cream'` входит в список `friend_foods`, а элемент `'cannoli'` в этот список не входит. Если бы эти два списка просто совпадали, то их содержимое уже не различалось бы. Например, вот что происходит при попытке копирования списка без использования сегмента:

```
my_foods = ['pizza', 'falafel', 'carrot cake']

# Не работает:
❶ friend_foods = my_foods

my_foods.append('cannoli')
friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

Вместо того чтобы сохранять копию `my_foods` в `friend_foods` в точке ❶, мы задаем `friend_foods` равным `my_foods`. Этот синтаксис в действительности сообщает Python, что новая переменная `friend_foods` должна быть связана со списком, уже хранящимся в `my_foods`, поэтому теперь обе переменные связаны с одним списком. В результате при добавлении элемента `'cannoli'` в `my_foods` этот элемент также появляется в `friend_foods`. Аналогичным образом элемент `'ice cream'` появляется в обоих списках, хотя на первый взгляд он был добавлен только в `friend_foods`.

Вывод показывает, что оба списка содержат одинаковые элементы, а это совсем не то, что требовалось:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

ПРИМЕЧАНИЕ Если какие-то подробности в этом примере кажутся непонятными, не огорчайтесь. В двух словах, если при работе с копией списка происходит что-то непредвиденное, убедитесь в том, что список копируется с использованием сегмента, как это делается в нашем первом примере.

УПРАЖНЕНИЯ

4.10. Сегменты: добавьте в конец одной из программ, написанных в этой главе, фрагмент, который делает следующее:

- Выводит сообщение «The irst three items in the list are:», а затем использует сегмент для вывода первых трех элементов из списка.
- Выводит сообщение «Three items from the middle of the list are:», а затем использует сегмент для вывода первых трех элементов из середины списка.
- Выводит сообщение «The last three items in the list are:», а затем использует сегмент для вывода последних трех элементов из списка.

4.11. Моя пицца, твоя пицца: начните с программы из упражнения 4.1. Создайте копию списка с видами пиццы, присвойте ему имя `friend_pizzas`. Затем сделайте следующее:

- Добавьте новую пиццу в исходный список.
- Добавьте другую пиццу в список `friend_pizzas`.
- Докажите, что в программе существуют два разных списка. Выведите сообщение «My favorite pizzas are:», а затем первый список в цикле `for`. Выведите сообщение «My friend's favorite pizzas are:», а затем второй список в цикле for. Убедитесь в том, что каждая новая пицца находится в соответствующем списке.

4.12. Больше циклов: во всех версиях foods.py из этого раздела мы избегали использования цикла for при выводе для экономии места. Выберите версию foods.py и напишите два цикла `for` для вывода каждого списка.

Кортежи

Списки хорошо подходят для хранения наборов элементов, которые могут изменяться на протяжении жизненного цикла программы. Например, возможность модификации списков жизненно необходима при работе со списками пользователей сайта или списками персонажей игры. Однако в некоторых ситуациях требуется создать список элементов, который не может изменяться. *Кортежи* (tuples) предоставляют именно такую возможность. В языке Python значения, которые не могут изменяться, называются *неизменяемыми* (immutable), а неизменяемый список называется *кортежем*.

Определение кортежа

Кортеж выглядит как список, не считая того, что вместо квадратных скобок используются круглые скобки. После определения кортежа вы можете обращаться к его отдельным элементам по индексам точно так же, как это делается при работе со списком.

Допустим, имеется прямоугольник, который в программе всегда должен иметь строго определенные размеры. Чтобы гарантировать неизменность размеров, можно объединить размеры в кортеж:

dimensions.py

```
❶ dimensions = (200, 50)
❷ print(dimensions[0])
  print(dimensions[1])
```

В точке ❶ определяется кортеж `dimensions`, при этом вместо квадратных скобок используются круглые. В точке ❷ каждый элемент кортежа выводится по отдельности с использованием того же синтаксиса, который использовался для обращения к элементу списка:

```
200
50
```

Посмотрим, что произойдет при попытке изменения одного из элементов в кортеже `dimensions`:

```
dimensions = (200, 50)
❶ dimensions[0] = 250
```

Код в точке ❶ пытается изменить первое значение, но Python возвращает ошибку типа. По сути, так как мы пытаемся изменить кортеж, а эта операция недопустима для объектов этого типа, Python сообщает о невозможности присваивания нового значения элементу в кортеже:

```
Traceback (most recent call last):
  File "dimensions.py", line 3, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

И это хорошо, потому что мы хотим, чтобы Python сообщал о попытке изменения размеров прямоугольника в программе, выдавая сообщение об ошибке.

ПРИМЕЧАНИЕ Формально кортеж определяется наличием запятой; круглые скобки просто делают запись более аккуратной и понятной. Если вы хотите определить кортеж, состоящий из одного элемента, включите завершающую запятую:

```
my_t = (3,)
```

Обычно создание кортежа из одного элемента не имеет особого смысла. Тем не менее это может произойти при автоматическом генерировании кортежей.

Перебор всех значений в кортеже

Для перебора всех значений в кортеже используется цикл `for`, как и при работе со списками:

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

Python возвращает все элементы кортежа по аналогии с тем, как это делается со списком:

```
200
50
```

Замена кортежа

Элементы кортежа не могут изменяться, но вы можете присвоить новое значение переменной, в которой хранится кортеж. Таким образом, для изменения размеров прямоугольника следует переопределить весь кортеж:

```
❶ dimensions = (200, 50)

  print("Original dimensions:")
  for dimension in dimensions:
      print(dimension)

❷ dimensions = (400, 100)
❸ print("\nModified dimensions:")
  for dimension in dimensions:

      print(dimension)
```

Блок, начинающийся в точке ❶, определяет исходный кортеж и выводит исходные размеры. В точке ❷ в переменной `dimensions` сохраняется новый кортеж, после чего в точке ❸ выводятся новые размеры. На этот раз Python не выдает сообщений об ошибке, потому что замена значения переменной является допустимой операцией:

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

По сравнению со списками структуры данных кортежей относительно просты. Используйте их для хранения наборов значений, которые не должны изменяться на протяжении жизненного цикла программы.

УПРАЖНЕНИЯ

4.13. Шведский стол: меню «шведского стола» в ресторане состоит всего из пяти пунктов. Придумайте пять простых блюд и сохраните их в кортеже.

- Используйте цикл for для вывода всех блюд, предлагаемых рестораном.
- Попробуйте изменить один из элементов и убедитесь в том, что Python отказывается вносить изменения.
- Ресторан изменяет меню, заменяя два элемента другими блюдами. Добавьте блок кода, который заменяет кортеж, и используйте цикл for для вывода каждого элемента обновленного меню.

Стиль программирования

Итак, вы постепенно начинаете писать более длинные программы, и вам стоит познакомиться с некоторыми рекомендациями по стилевому оформлению кода. Не жалейте времени на то, чтобы ваш код читался как можно проще. Понятный код помогает следить за тем, что делает ваша программа, и упрощает изучение вашего кода другими разработчиками.

Программисты Python выработали ряд соглашений по стилю, чтобы весь код имел хотя бы отдаленно похожую структуру. Научившись писать «чистый» код Python,

вы сможете понять общую структуру кода Python, написанного любым другим программистом, соблюдающим те же рекомендации. Если вы рассчитываете когда-нибудь стать профессиональным программистом, привыкайте соблюдать эти рекомендации как можно раньше, чтобы выработать полезную привычку.

Рекомендации по стилю

Когда кто-нибудь хочет внести изменения в язык Python, он пишет документ *PEP* (Python Enhancement Proposal). Одним из самых старых PEP является документ *PEP 8* с рекомендациями по стилевому оформлению кода. PEP 8 довольно длинный, но большая часть документа посвящена более сложным программным структурам, нежели те, которые встречались вам до настоящего момента.

Руководство по стилю Python было написано с пониманием того факта, что код читается чаще, чем пишется. Вы пишете свой код один раз, а потом начинаете читать его, когда переходите к отладке. При расширении функциональности программы вы снова тратите время на чтение своего кода. А когда вашим кодом начинают пользоваться другие программисты, они тоже читают его.

Выбирая между написанием кода, который проще пишется, и кодом, который проще читается, программисты Python почти всегда рекомендуют второй вариант. Следующие советы помогут вам с самого начала писать чистый, понятный код.

Отступы

PEP 8 рекомендует обозначать уровень отступа четырьмя пробелами. Использование четырех пробелов упрощает чтение программы и при этом оставляет достаточно места для нескольких уровней отступов в каждой строке.

В программах форматирования текста для создания отступов часто используются табуляции вместо пробелов. Такой способ хорошо работает в текстовых процессорах, но интерпретатор Python приходит в замешательство, когда табуляции смешиваются с пробелами. В каждом текстовом редакторе имеется параметр конфигурации, который заменяет нажатие клавиши табуляции заданным количеством пробелов. Конечно, клавиша табуляции удобна, но вы должны проследить за тем, чтобы редактор вставлял в документ пробелы вместо табуляций.

Смешение табуляций и пробелов в файле может создать проблемы, сильно затрудняющие диагностику. Если вы думаете, что в программе табуляции смешались с пробелами, в большинстве редакторов существует возможность преобразования всех табуляций в пробелы.

Длина строк

Многие программисты Python рекомендуют ограничивать длину строк 80 символами. Исторически эта рекомендация появилась из-за того, что в большинстве

компьютеров в одной строке терминального окна помещалось всего 79 символов. В настоящее время на экранах помещаются куда более длинные строки, но для применения стандартной длины строки в 79 символов существуют и другие причины. Профессиональные программисты часто открывают на одном экране сразу несколько файлов; стандартная длина строки позволяет видеть все строки в двух или трех файлах, открытых на экране одновременно. PEP 8 также рекомендует ограничивать комментарии 72 символами на строку, потому что некоторые служебные программы, автоматически генерирующие документацию в больших проектах, добавляют символы форматирования в начале каждой строки комментария.

Рекомендации PEP 8 по выбору длины строки не являются незыблемыми, и некоторые программисты предпочитают ограничение в 99 символов. Пока вы учитесь, длина строки в коде не так важна, но учтите, что при совместной работе в группах почти всегда соблюдаются рекомендации PEP 8. В большинстве редакторов можно установить визуальный ориентир (обычно вертикальную линию на экране), показывающий, где проходит граница.

ПРИМЕЧАНИЕ В приложении Б показано, как настроить текстовый редактор, чтобы он всегда вставлял четыре пробела при нажатии клавиши табуляции и отображал вертикальную линию для соблюдения ограничения длины строки 79 символами.

Пустые строки

Пустые строки применяются для визуальной группировки частей программы. Используйте пустые строки для структурирования файлов, но не злоупотребляйте ими. Примеры, приведенные в книге, помогут вам выработать нужный баланс. Например, если в программе пять строк кода создают список, а затем следующие три строки что-то делают с этим списком, два фрагмента уместно разделить пустой строкой. Тем не менее между ними не стоит вставлять три или четыре пустые строки.

Пустые строки не влияют на работу кода, но отражаются на его удобочитаемости. Интерпретатор Python использует горизонтальные отступы для интерпретации смысла кода, но игнорирует вертикальные интервалы.

Другие рекомендации

PEP 8 содержит много других рекомендаций по стилю, но эти рекомендации в основном относятся к программам более сложным, чем те, которые вы пишете на данный момент. По мере изучения более сложных возможностей Python я буду приводить соответствующие фрагменты рекомендаций PEP 8.

УПРАЖНЕНИЯ

4.14. Просмотрите исходное руководство по стилю PEP 8 по адресу https://python.org/dev/peps/pep-0008/. Пока вы будете пользоваться им относительно редко, но просмотреть его будет интересно.

4.15. Анализ кода: выберите три программы, написанные в этой главе, и измените каждую в соответствии с рекомендациями PEP 8:

- Используйте четыре пробела для каждого уровня отступов. Настройте текстовый редактор так, чтобы он вставлял четыре пробела при каждом нажатии клавиши табуляции, если это не было сделано ранее (за инструкциями обращайтесь к приложению Б).
- Используйте менее 80 символов в каждой строке. Настройте редактор так, чтобы он отображал вертикальную черту в позиции 80-го символа.
- Не злоупотребляйте пустыми строками в файлах программ.

Итоги

В этой главе вы научились эффективно работать с элементами списка. Вы узнали, как работать со списком в цикле `for`, как Python использует отступы для определения структуры программы и как избежать некоторых типичных ошибок при использовании отступов. Вы научились создавать простые числовые списки, а также изучили некоторые операции с числовыми списками. Вы узнали, как создать сегмент списка для работы с подмножеством элементов и как правильно копировать списки с использованием сегмента. Глава завершается описанием кортежей, до определенной степени защищающих наборы значений, которые не должны изменяться, и рекомендациями по стилевому оформлению вашего кода (сложность которого со временем только возрастает) для упрощения его чтения.

В главе 5 мы займемся обработкой различных условий с использованием команд `if`. Вы научитесь группировать относительно сложные наборы проверок для обработки именно той ситуации или информации, которая вам нужна. Также в этой главе рассматривается использование команд `if` при переборе элементов списка для выполнения действий с элементами, отобранными по некоторому критерию.

5 Команды if

Программисту часто приходится проверять наборы условий и принимать решения в зависимости от этих условий. Команда `if` в языке Python позволяет проверить текущее состояние программы и выбрать дальнейшие действия в зависимости от результатов проверки.

В этой главе вы научитесь писать условные проверки для любых интересующих вас условий. Мы начнем с простых команд `if`, а затем перейдем к более сложным сериям команд `if` для проверки комбинированных условий. Затем эта концепция будет применена к спискам; вы узнаете, как написать цикл, который выполняет с большинством элементов списка одну операцию, но для некоторых элементов с конкретными значениями применяется особая обработка.

Простой пример

Следующий короткий пример показывает, как правильно организовать обработку специальных ситуаций с использованием `if`. Допустим, у вас имеется список машин и вы хотите вывести название каждой машины. Названия большинства машин должны записываться с капитализацией (первая буква в верхнем регистре, остальные в нижнем). С другой стороны, значение `'bmw'` должно записываться в верхнем регистре. Следующий код перебирает список названий машин и ищет в нем значение `'bmw'`. Для всех элементов, содержащих значение `'bmw'`, значение выводится в верхнем регистре:

cars.py

```
cars = ['audi', 'bmw', 'subaru', 'toyota']

for car in cars:
❶     if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

Цикл в этом примере ❶ сначала проверяет, содержит ли `car` значение `'bmw'`. Если проверка дает положительный результат, то значение выводится в верхнем регистре. Если `car` содержит все, что угодно, кроме `'bmw'`, то при выводе значения применяется капитализация:

```
Audi
BMW
Subaru
Toyota
```

В этом примере объединяются несколько концепций, о которых вы узнаете в этой главе. Для начала рассмотрим основные конструкции, применяемые для проверки условий в программах.

Проверка условий

В каждой команде `if` центральное место занимает выражение, результатом которого является логическая истина (`True`) или логическая ложь (`False`); это выражение называется *условием*. В зависимости от результата проверки Python решает, должен ли выполняться код в команде `if`. Если результат условия равен `True`, то Python выполняет код, следующий за командой `if`. Если же будет получен результат `False`, то Python игнорирует код, следующий за командой `if`.

Проверка равенства

Во многих условиях текущее значение переменной сравнивается с конкретным значением, интересующим вас. Простейшее условие проверяет, равно ли значение переменной конкретной величине:

```
❶ >>> car = 'bmw'
❷ >>> car == 'bmw'
  True
```

В строке ❶ переменной `car` присваивается значение `'bmw'`; операция выполняется одним знаком `=`, как вы уже неоднократно видели. Строка ❷ проверяет, равно ли значение `car` строке `'bmw'`; для проверки используется двойной знак равенства (`==`). Этот оператор возвращает `True`, если значения слева и справа от оператора равны; если же значения не совпадают, оператор возвращает `False`. В нашем примере значения совпадают, поэтому Python возвращает `True`.

Если `car` принимает любое другое значение вместо `'bmw'`, проверка возвращает `False`:

```
❶ >>> car = 'audi'
❷ >>> car == 'bmw'
  False
```

Одиночный знак равенства выполняет операцию; код ❶ можно прочитать в форме «Присвоить `car` значение `'audi'`». С другой стороны, двойной знак равенства, как в строке ❷, задает вопрос: «Значение `car` равно `'bmw'`?». Такое применение знаков равенства встречается во многих языках программирования.

Проверка равенства без учета регистра

В языке Python проверка равенства выполняется с учетом регистра. Например, два значения с разным регистром символов равными не считаются:

```
>>> car = 'Audi'
>>> car == 'audi'
False
```

Если регистр символов важен, такое поведение приносит пользу. Но если проверка должна выполняться на уровне символов без учета регистра, преобразуйте значение переменной к нижнему регистру перед выполнением сравнения:

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Условие возвращает `True` независимо от регистра символов `'Audi'`, потому что проверка теперь выполняется без учета регистра. Функция `lower()` не изменяет значение, которое изначально хранилось в `car`, так что сравнение не отражается на исходной переменной:

```
❶ >>> car = 'Audi'
❷ >>> car.lower() == 'audi'
  True
❸ >>> car
  'Audi'
```

В точке ❶ строка `'Audi'` сохраняется в переменной `car`. В точке ❷ значение `car` приводится к нижнему регистру и сравнивается со значением строки `'audi'`, также записанным в нижнем регистре. Две строки совпадают, поэтому Python возвращает `True`. Вывод в точке ❸ показывает, что значение, хранящееся в `car`, не изменилось при вызове `lower()`.

Веб-сайты устанавливают определенные правила для данных, вводимых пользователями подобным образом. Например, сайт может использовать проверку условия, чтобы убедиться в том, что имя каждого пользователя уникально (а не совпадает с именем другого пользователя, отличаясь от него только регистром символов). Когда кто-то указывает новое имя пользователя, это имя преобразуется к нижнему регистру и сравнивается с версиями всех существующих имен в нижнем регистре. Во время такой проверки имя `'John'` будет отклонено, если в системе уже используется любая разновидность `'john'`.

Проверка неравенства

Если вы хотите проверить, что два значения *различны*, используйте комбинацию из восклицательного знака и знака равенства (`!=`). Восклицательный знак представляет отрицание, как и во многих языках программирования.

Для знакомства с оператором неравенства мы воспользуемся другой командой `if`. В переменной хранится заказанный топпинг к пицце; если клиент не заказал анчоусы (anchovies), программа выводит сообщение:

toppings.py

```
  requested_topping = 'mushrooms'

❶ if requested_topping != 'anchovies':
      print("Hold the anchovies!")
```

Строка ❶ сравнивает значение `requested_topping` со значением `'anchovies'`. Если эти два значения не равны, Python возвращает `True` и выполняет код после команды `if`. Если два значения равны, Python возвращает `False` и не выполняет код после команды `if`.

Так как значение `requested_topping` отлично от `'anchovies'`, команда `print` будет выполнена:

```
Hold the anchovies!
```

В большинстве условных выражений, которые вы будете использовать в программах, будет проверяться равенство, но иногда проверка неравенства оказывается более эффективной.

Сравнения чисел

Проверка числовых значений достаточно прямолинейна. Например, следующий код проверяет, что переменная `age` равна 18:

```
>>> age = 18
>>> age == 18
True
```

Также можно проверить условие неравенства двух чисел. Например, следующий код выводит сообщение, если значение переменной `answer` отлично от ожидаемого:

magic_ number.py

```
  answer = 17

❶ if answer != 42:
      print("That is not the correct answer. Please try again!")
```

Условие ❶ выполняется, потому что значение `answer` (17) не равно 42. Так как условие истинно, блок с отступом выполняется:

```
That is not the correct answer. Please try again!
```

В условные команды также можно включать всевозможные математические сравнения: меньше, меньше или равно, больше, больше или равно:

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Все эти математические сравнения могут использоваться в условиях `if`, что повышает точность формулировки интересующих вас условий.

Проверка нескольких условий

Иногда требуется проверить несколько условий одновременно. Например, для выполнения действия бывает нужно, чтобы истинными были сразу два условия; в других случаях достаточно, чтобы истинным было хотя бы одно из двух условий. Ключевые слова `and` и `or` помогут вам в подобных ситуациях.

Использование and для проверки нескольких условий

Чтобы проверить, что два условия истинны *одновременно*, объедините их ключевым словом `and`; если оба условия истинны, то и все выражение тоже истинно. Если хотя бы одно (или оба) условие ложно, то и результат всего выражения равен `False`.

Например, чтобы убедиться в том, что каждому из двух людей больше 21 года, используйте следующую проверку:

```
❶ >>> age_0 = 22
  >>> age_1 = 18
❷ >>> age_0 >= 21 and age_1 >= 21
  False
❸ >>> age_1 = 22
  >>> age_0 >= 21 and age_1 >= 21
  True
```

В точке ❶ определяются две переменные, `age_0` и `age_1`. В точке ❷ программа проверяет, что оба значения равны 21 и более. Левое условие выполняется, а правое нет, поэтому все условное выражение дает результат `False`. В точке ❸ переменной `age_1` присваивается значение 22. Теперь значение `age_1` больше 21; обе проверки проходят, а все условное выражение дает истинный результат.

Чтобы код лучше читался, отдельные условия можно заключить в круглые скобки, но это не обязательно. С круглыми скобками проверка может выглядеть так:

```
(age_0 >= 21) and (age_1 >= 21)
```

Использование or для проверки нескольких условий

Ключевое слово `or` тоже позволяет проверить несколько условий, но результат общей проверки является истинным в том случае, когда истинно хотя бы одно или оба условия. Ложный результат достигается только в том случае, если оба условия ложны.

Вернемся к примеру с возрастом, но на этот раз проверим, что хотя бы одна из двух переменных больше 21:

```
❶ >>> age_0 = 22
  >>> age_1 = 18
❷ >>> age_0 >= 21 or age_1 >= 21
  True
❸ >>> age_0 = 18
  >>> age_0 >= 21 or age_1 >= 21
  False
```

Как и в предыдущем случае, в точке ❶ определяются две переменные. Так как условие для `age_0` в точке ❷ истинно, все выражение также дает истинный результат. Затем значение `age_0` уменьшается до 18. При проверке ❸ оба условия оказываются ложными, и общий результат всего выражения тоже ложен.

Проверка вхождения значений в список

Иногда бывает важно проверить, содержит ли список некоторое значение, прежде чем выполнять действие. Например, перед завершением регистрации нового пользователя на сайте можно проверить, существует ли его имя в списке имен действующих пользователей, или в картографическом проекте определить, входит ли передаваемое место в список известных мест на карте.

Чтобы узнать, присутствует ли заданное значение в списке, воспользуйтесь ключевым словом `in`. Допустим, вы пишете программу для пиццерии. Вы создали список дополнений к пицце, заказанных клиентом, и хотите проверить, входят ли некоторые дополнения в этот список.

```
  >>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
❶ >>> 'mushrooms' in requested_toppings
  True
❷ >>> 'pepperoni' in requested_toppings
  False
```

В точках ❶ и ❷ ключевое слово `in` приказывает Python проверить, входят ли значения `'mushrooms'` и `'pepperoni'` в список `requested_toppings`. Это весьма полезно,

потому что вы можете создать список значений, критичных для вашей программы, а затем легко проверить, присутствует ли проверяемое значение в списке.

Проверка отсутствия значения в списке

В других случаях программа должна убедиться в том, что значение *не входит* в список. Для этого используется ключевое слово `not`. Для примера рассмотрим список пользователей, которым запрещено писать комментарии на форуме. Прежде чем разрешить пользователю отправку комментария, можно проверить, не был ли пользователь включен в черный список:

banned_users.py

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'

❶ if user not in banned_users:
      print(f"{user.title()}, you can post a response if you wish.")
```

Строка ❶ достаточно четко читается: если пользователь не входит в черный список `banned_users`, то Python возвращает `True` и выполняет строку с отступом.

Пользователь `'marie'` в этот список не входит, поэтому программа выводит соответствующее сообщение:

```
Marie, you can post a response if you wish.
```

Логические выражения

В процессе изучения программирования вы рано или поздно услышите термин «логическое выражение». По сути, это всего лишь другое название для проверки условия. Результат логического выражения равен `True` или `False`, как и результат условного выражения после его вычисления.

Логические выражения часто используются для проверки некоторых условий — например, запущена ли компьютерная игра или разрешено ли пользователю редактирование некоторой информации на сайте:

```
game_active = True
can_edit = False
```

Логические выражения предоставляют эффективные средства для контроля состояния программы или определенного условия, играющего важную роль в вашей программе.

УПРАЖНЕНИЯ

5.1. Проверка условий: напишите последовательность условий. Выведите описание каждой проверки и ваш прогноз относительно ее результата. Код должен выглядеть примерно так:

```
car = 'subaru'
print("Is car == 'subaru'? I predict True.")
print(car == 'subaru')

print("\nIs car == 'audi'? I predict False.")
print(car == 'audi')
```

- Внимательно просмотрите результаты. Убедитесь в том, что вы понимаете, почему результат каждой строки равен `True` или `False`.
- Создайте как минимум 10 условий. Не менее пяти одних должны давать результат `True`, а не менее пяти других — результат `False`.

5.2. Больше условий: количество условий не ограничивается десятью. Попробуйте написать другие условия и включить их в conditional_tests.py. Программа должна выдавать по крайней мере один истинный и один ложный результат для следующих видов проверок:

- Проверка равенства и неравенства строк.
- Проверки с использованием функции `lower()`.
- Числовые проверки равенства и неравенства, условий «больше», «меньше», «больше или равно», «меньше или равно».
- Проверки с ключевым словом and и or.
- Проверка вхождения элемента в список.
- Проверка отсутствия элемента в списке.

Команды if

Когда вы поймете, как работают проверки условий, можно переходить к написанию команд `if`. Существуют несколько разновидностей команд `if`, и выбор варианта зависит от количества проверяемых условий. Примеры команд `if` уже встречались вам при обсуждении проверки условий, но сейчас эта тема будет рассмотрена более подробно.

Простые команды if

Простейшая форма команды `if` состоит из одного условия и одного действия:

```
if условие:
    действие
```

В первой строке размещается условие, а в блоке с отступом — практически любое действие. Если условие истинно, то Python выполняет код в блоке после команды `if`, а если ложно, этот код игнорируется.

Допустим, имеется переменная, представляющая возраст человека. Следующий код проверяет, что этот возраст достаточен для голосования:

voting.py

```
  age = 19
❶ if age >= 18:
❷     print("You are old enough to vote!")
```

В точке ❶ Python проверяет, что значение переменной `age` больше или равно 18. В таком случае выполняется команда `print` ❷ в строке с отступом:

```
You are old enough to vote!
```

Отступы в командах `if` играют ту же роль, что и в циклах `for`. Если условие истинно, то все строки с отступом после команды `if` выполняются, а если ложно — весь блок с отступом игнорируется.

Блок команды `if` может содержать сколько угодно строк. Добавим еще одну строку для вывода дополнительного сообщения в том случае, если возраст достаточен для голосования:

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

Условие выполняется, а обе команды `print` снабжены отступом, поэтому выводятся оба сообщения:

```
You are old enough to vote!
Have you registered to vote yet?
```

Если значение `age` меньше 18, программа ничего не выводит.

Команды if-else

Часто в программе необходимо выполнить одно действие в том случае, если условие истинно, и другое действие, если оно ложно. С синтаксисом `if-else` это возможно. Блок `if-else` в целом похож на команду `if`, но секция `else` определяет действие или набор действий, выполняемых при неудачной проверке.

В следующем примере выводится то же сообщение, которое выводилось ранее, если возраст достаточен для голосования, но на этот раз при любом другом возрасте выводится другое сообщение:

```
  age = 17
❶ if age >= 18:
      print("You are old enough to vote!")
      print("Have you registered to vote yet?")
❷ else:
      print("Sorry, you are too young to vote.")
      print("Please register to vote as soon as you turn 18!")
```

Если условие ❶ истинно, то выполняется первый блок с командами `print`. Если же условие ложно, выполняется блок `else` в точке ❷. Так как значение `age` на этот раз меньше 18, условие оказывается ложным и выполняется код в блоке `else`:

```
Sorry, you are too young to vote.
Please register to vote as soon as you turn 18!
```

Этот код работает, потому что существуют обе возможные ситуации: возраст либо достаточен для голосования, либо недостаточен. Структура `if-else` хорошо подходит для тех ситуаций, в которых Python всегда выполняет только одно из двух возможных действий. В подобных простых цепочках `if-else` всегда выполняется одно из двух возможных действий.

Цепочки if-elif-else

Нередко в программе требуется проверять более двух возможных ситуаций; для таких ситуаций в Python предусмотрен синтаксис `if-elif-else`. Python выполняет только один блок в цепочке `if-elif-else`. Все условия проверяются по порядку до тех пор, пока одно из них не даст истинный результат. Далее выполняется код, следующий за этим условием, а все остальные проверки Python пропускает.

Во многих реальных ситуациях существует более двух возможных результатов. Представьте себе парк аттракционов, который взимает разную плату за вход для разных возрастных групп:

- Для посетителей младше 4 лет вход бесплатный.
- Для посетителей от 4 до 18 лет билет стоит $25.
- Для посетителей от 18 лет и старше билет стоит $40.

Как использовать команду `if` для определения платы за вход? Следующий код определяет, к какой возрастной категории относится посетитель, и выводит сообщение со стоимостью билета:

amusement_park.py

```
age = 12

❶ if age < 4:
      print("Your admission cost is $0.")
❷ elif age < 18:
      print("Your admission cost is $25.")
❸ else:
      print("Your admission cost is $40.")
```

Условие `if` в точке ❶ проверяет, что возраст посетителя меньше 4 лет. Если условие истинно, то программа выводит соответствующее сообщение и Python пропускает остальные проверки. Строка `elif` в точке ❷ в действительности является еще одной проверкой `if`, которая выполняется только в том случае, если предыдущая проверка завершилась неудачей. В этом месте цепочки известно, что возраст посетителя не меньше 4 лет, потому что первое условие было ложным. Если посетителю меньше 18 лет, программа выводит соответствующее сообщение и Python пропускает блок `else`. Если ложны оба условия — `if` и `elif`, то Python выполняет код в блоке `else` в точке ❸.

В данном примере условие ❶ дает ложный результат, поэтому его блок не выполняется. Однако второе условие оказывается истинным (12 меньше 18), поэтому код будет выполнен. Вывод состоит из одного сообщения с ценой билета:

```
Your admission cost is $25.
```

При любом значении возраста больше 17 первые два условия ложны. В таких ситуациях блок `else` будет выполнен и цена билета составит $40.

Вместо того чтобы выводить сообщение с ценой билета в блоках `if-elif-else`, лучше использовать другое, более компактное решение: присвоить цену в цепочке `if-elif-else`, а затем добавить одну команду `print` после выполнения цепочки:

```
  age = 12

  if age < 4:
❶     price = 0
  elif age < 18:
❷     price = 25
  else:
❸     price = 40

❹ print(f"Your admission cost is ${price}.")
```

Строки ❶, ❷ и ❸ присваивают значение `price` в зависимости от значения `age`, как и в предыдущем примере. После присваивания цены в цепочке `if-elif-else` отдельная команда `print` без отступа ❹ использует это значение для вывода сообщения с ценой билета.

Этот пример выводит тот же результат, что и предыдущий, но цепочка `if-elif-else` имеет более четкую специализацию. Вместо того чтобы определять цену и выводить сообщения, она просто определяет цену билета. Кроме повышения эффективности, у этого кода есть дополнительное преимущество: его легче модифицировать. Чтобы изменить текст выходного сообщения, достаточно будет отредактировать всего одну команду `print` — вместо трех разных команд.

Серии блоков elif

Код может содержать сколько угодно блоков `elif`. Например, если парк аттракционов введет особую скидку для пожилых посетителей, вы можете добавить в свой код еще одну проверку для определения того, распространяется ли скидка на текущего посетителя. Допустим, посетители возрастом 65 и выше платят половину обычной цены билета, или $40:

```
  age = 12

  if age < 4:
      price = 0
  elif age < 18:
      price = 25
❶ elif age < 65:
```

```
      price = 40
❷ else:
      price = 20

  print(f"Your admission cost is ${price}.")
```

Большая часть кода осталась неизменной. Второй блок `elif` в точке ❶ теперь убеждается в том, что посетителю меньше 65 лет, прежде чем назначить ему полную цену билета $40. Обратите внимание: значение, присвоенное в блоке `else` ❷, должно быть заменено на $20, потому что до этого блока доходят только посетители с возрастом 65 и выше.

Отсутствие блока else

Python не требует, чтобы цепочка `if-elif` непременно завершалась блоком `else`. Иногда блок `else` удобен; иногда бывает лучше использовать дополнительную секцию `elif` для обработки конкретного условия:

```
  age = 12

  if age < 4:
      price = 0
  elif age < 18:
      price = 25
  elif age < 65:
      price = 40
❶ elif age >= 65:
      price = 20

  print(f"Your admission cost is ${price}.")
```

Блок `elif` в точке ❶ назначает цену $20, если возраст посетителя равен 65 и выше; смысл такого кода более понятен, чем у обобщенного блока `else`. С таким изменением выполнение каждого блока возможно только при истинности конкретного условия.

Блок `else` «универсален»: он обрабатывает все условия, не подходящие ни под одну конкретную проверку `if` или `elif`, причем в эту категорию иногда могут попасть недействительные или даже вредоносные данные. Если у вас имеется завершающее конкретное условие, лучше используйте завершающий блок `elif` и опустите блок `else`. В этом случае вы можете быть уверены в том, что ваш код будет выполняться только в правильных условиях.

Проверка нескольких условий

Цепочки `if-elif-else` эффективны, но они подходят в том случае, если истинным должно быть только одно условие. Когда Python находит выполняющееся условие, все остальные проверки пропускаются. Такое поведение достаточно эффективно, потому что оно позволяет проверить одно конкретное условие.

Однако иногда бывает важно проверить *все* условия, представляющие интерес. В таких случаях следует применять серии простых команд `if` без блоков `elif` или `else`. Такое решение уместно, когда истинными могут быть сразу несколько условий и вы хотите отреагировать на все истинные.

Вернемся к примеру с пиццей. Если кто-то закажет пиццу с двумя топпингами, программа должна обработать оба топпинга:

toppings.py

```
❶ requested_toppings = ['mushrooms', 'extra cheese']

❷ if 'mushrooms' in requested_toppings:
      print("Adding mushrooms.")
❸ if 'pepperoni' in requested_toppings:
      print("Adding pepperoni.")
❹ if 'extra cheese' in requested_toppings:
      print("Adding extra cheese.")

  print("\nFinished making your pizza!")
```

Обработка начинается в точке ❶ со списка, содержащего заказанные топпинги. Команды `if` в точке ❷ проверяют, включает ли заказ конкретные топпинги — грибы и пепперони, и если включает, выводят подтверждающее сообщение. Проверка в точке ❸ реализована простой командой `if`, а не `elif` или `else`, поэтому условие будет проверяться независимо от того, было ли предыдущее условие истинным или ложным. Код в точке ❹ проверяет, была ли заказана дополнительная порция сыра, независимо от результата первых двух проверок. Эти три независимых условия проверяются при каждом выполнении программы.

Так как в этом коде проверяются все возможные варианты топпингов, в заказ будут включены два топпинга из трех:

```
Adding mushrooms.
Adding extra cheese.

Finished making your pizza!
```

Если бы в программе использовался блок `if-elif-else`, код работал бы неправильно, потому что он прерывал бы работу после обнаружения первого истинного условия. Вот как это выглядело бы:

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

Первое же проверяемое условие (для 'mushrooms') оказывается истинным. Однако значения 'extra cheese' и 'pepperoni' после этого не проверяются, потому что в цепочках if-elif-else после обнаружения первого истинного условия все остальные условия пропускаются. В результате в пиццу будет включено только первый из заказанных топпингов:

```
Adding mushrooms.

Finished making your pizza!
```

Итак, если вы хотите, чтобы в программе выполнялся только один блок кода, используйте цепочку if-elif-else. Если же выполняться должны несколько блоков, используйте серию независимых команд if.

УПРАЖНЕНИЯ

5.3. Цвета 1: представьте, что в вашей компьютерной игре только что был подбит корабль пришельцев. Создайте переменную с именем alien_color и присвойте ей значение 'green', 'yellow' или 'red'.

- Напишите команду if для проверки того, что переменная содержит значение 'green'. Если условие истинно, выведите сообщение о том, что игрок только что заработал 5 очков.
- Напишите одну версию программы, в которой условие if выполняется, и другую версию, в которой оно не выполняется. (Во второй версии никакое сообщение выводиться не должно.)

5.4. Цвета 2: выберите цвет, как это было сделано в упражнении 5.3, и напишите цепочку if-else.

- Напишите команду if для проверки того, что переменная содержит значение 'green'. Если условие истинно, выведите сообщение о том, что игрок только что заработал 5 очков.
- Если переменная содержит любое другое значение, выведите сообщение о том, что игрок только что заработал 10 очков.
- Напишите одну версию программы, в которой выполняется блок if, и другую версию, в которой выполняется блок else.

5.5. Цвета 3: преобразуйте цепочку if-else из упражнения 5.4 в цепочку if-elif-else.

- Если переменная содержит значение 'green', выведите сообщение о том, что игрок только что заработал 5 очков.
- Если переменная содержит значение 'yellow', выведите сообщение о том, что игрок только что заработал 10 очков.
- Если переменная содержит значение 'red', выведите сообщение о том, что игрок только что заработал 15 очков.
- Напишите три версии программы и проследите за тем, чтобы для каждого цвета пришельца выводилось соответствующее сообщение.

5.6. Периоды жизни: напишите цепочку if-elif-else для определения периода жизни человека. Присвойте значение переменной age, а затем выведите сообщение:

- Если значение меньше 2 — младенец.
- Если значение больше или равно 2, но меньше 4 — малыш.
- Если значение больше или равно 4, но меньше 13 — ребенок.
- Если значение больше или равно 13, но меньше 20 — подросток.
- Если значение больше или равно 20, но меньше 65 — взрослый.
- Если значение больше или равно 65 — пожилой человек.

5.7. Любимый фрукт: составьте список своих любимых фруктов. Напишите серию независимых команд if для проверки того, присутствуют ли некоторые фрукты в списке.

- Создайте список трех своих любимых фруктов и назовите его `favorite_fruits`.
- Напишите пять команд `if`. Каждая команда должна проверять, входит ли определенный тип фрукта в список. Если фрукт входит в список, блок if должен выводить сообщение вида «You really like bananas!».

Использование команд if со списками

Объединение команд `if` со списками открывает ряд интересных возможностей. Например, вы можете отслеживать специальные значения, для которых необходима особая обработка по сравнению с другими значения в списке, или эффективно управлять изменяющимися условиями — например, наличием некоторых блюд в ресторане. Также объединение команд `if` со списками помогает продемонстрировать, что ваш код корректно работает во всех возможных ситуациях.

Проверка специальных значений

Эта глава началась с простого примера, показывающего, как обрабатывать особые значения (такие, как `'bmw'`), которые должны выводиться в другом формате по сравнению с другими значениями в списке. Теперь, когда вы лучше разбираетесь в проверках условий и командах `if`, давайте повнимательнее рассмотрим процесс поиска и обработки особых значений в списке.

Вернемся к примеру с пиццерией. Программа выводит сообщение каждый раз, когда пицца снабжается топпингом в процессе приготовления. Код этого действия можно записать чрезвычайно эффективно: нужно создать список топпингов, заказанных клиентом, и использовать цикл для перебора всех заказанных:

toppings.py

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

Вывод достаточно тривиален, поэтому код сводится к простому циклу `for`:

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.

Finished making your pizza!
```

А если в пиццерии вдруг закончится зеленый перец? Команда `if` в цикле `for` может правильно обработать эту ситуацию:

```
  requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

  for requested_topping in requested_toppings:
❶     if requested_topping == 'green peppers':
          print("Sorry, we are out of green peppers right now.")
❷     else:
          print(f"Adding {requested_topping}.")

  print("\nFinished making your pizza!")
```

На этот раз программа проверяет каждый заказанный элемент перед добавлением его к пицце. В точке ❶ программа проверяет, заказал ли клиент зеленый перец, и если заказал, выводит сообщение о том, что этого дополнения нет. Блок `else` в точке ❷ гарантирует, что все другие дополнения будут включены в заказ.

Из выходных данных видно, что все заказанные топпинги обрабатываются правильно:

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!
```

Проверка наличия элементов в списке

Для всех списков, с которыми мы работали до сих пор, действовало одно простое предположение: мы считали, что в каждом списке есть хотя бы один элемент. Скоро мы предоставим пользователю возможность вводить информацию, хранящуюся в списке, поэтому мы уже не можем предполагать, что при каждом выполнении цикла в списке есть хотя бы один элемент. В такой ситуации перед выполнением цикла `for` будет полезно проверить, есть ли в списке хотя бы один элемент.

Проверим, есть ли элементы в списке заказанных топпингов, перед изготовлением пиццы. Если список пуст, программа предлагает пользователю подтвердить, что он хочет базовую пиццу без топпингов. Если список не пуст, пицца готовится так же, как в предыдущих примерах:

```
❶ requested_toppings = []
❷ if requested_toppings:
      for requested_topping in requested_toppings:
          print(f"Adding {requested_topping}.")
      print("\nFinished making your pizza!")
❸ else:
      print("Are you sure you want a plain pizza?")
```

На этот раз мы начинаем с пустого списка заказанных топпингов в точке ❶. Вместо того чтобы сразу переходить к циклу `for`, программа выполняет проверку в точке ❷. Когда имя списка используется в условии `if`, Python возвращает `True`, если список содержит хотя бы один элемент; если список пуст, возвращается значение `False`. Если `requested_toppings` проходит проверку условия, выполняется тот же цикл `for`, который мы использовали в предыдущем примере. Если же условие ложно, то программа выводит сообщение, которое предлагает клиенту подтвердить, действительно ли он хочет получить базовую пиццу без дополнений ❸.

В данном примере список пуст, поэтому выводится сообщение:

```
Are you sure you want a plain pizza?
```

Если в списке есть хотя бы один элемент, то в выходные данные включается каждый заказанный топпинг.

Множественные списки

Посетители способны заказать что угодно, особенно когда речь заходит о топпингах к пицце. Что, если клиент захочет положить на пиццу картофель-фри? Списки и команды `if` позволят вам убедиться в том, что входные данные имеют смысл, прежде чем обрабатывать их.

Давайте проверим наличие нестандартных дополнений перед тем, как готовить пиццу. В следующем примере определяются два списка. Первый список содержит перечень доступных топпингов, а второй — список топпингов, заказанных клиентом. На этот раз каждый элемент из `requested_toppings` проверяется по списку доступных топпингов перед добавлением в пиццу:

```
❶ available_toppings = ['mushrooms', 'olives', 'green peppers',
                        'pepperoni', 'pineapple', 'extra cheese']

❷ requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

❸ for requested_topping in requested_toppings:
❹     if requested_topping in available_toppings:
          print(f"Adding {requested_topping}.")
❺     else:
          print(f"Sorry, we don't have {requested_topping}.")

  print("\nFinished making your pizza!")
```

В точке ❶ определяется список доступных топпингов к пицце. Стоит заметить, что если в пиццерии используется постоянный ассортимент топпингов, этот список можно реализовать в виде кортежа. В точке ❷ создается список топпингов, заказанных клиентом. Обратите внимание на необычный заказ `'french fries'`. В точке ❸ программа перебирает список заказанных топпингов. Внутри цикла программа сначала проверяет, что каждый заказанный топпинг присутствует в списке доступных топпингов ❹. Если топпинг доступен, он добавляется в пиццу. Если заказанный топпинг не входит в список, выполняется блок `else` ❺. Блок `else` выводит сообщение о том, что топпинг недоступен.

С этим синтаксисом программа выдает четкий, содержательный вывод:

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

Всего в нескольких строках кода нам удалось эффективно решить вполне реальную проблему!

УПРАЖНЕНИЯ

5.8. Hello Admin: создайте список из пяти и более имен пользователей, включающий имя `'admin'`. Представьте, что вы пишете код, который выводит приветственное сообщение для каждого пользователя после его входа на сайт. Переберите элементы списка и выведите сообщение для каждого пользователя:

- Для пользователя с именем 'admin' выведите особое сообщение — например, «Hello admin, would you like to see a status report?».
- В остальных случаях выводите универсальное приветствие — например, «Hello Jaden, thank you for logging in again».

5.9. Без пользователей: добавьте в hello_admin.py команду `if`, которая проверит, что список пользователей не пуст.

- Если список пуст, выведите сообщение «We need to ind some users!».
- Удалите из списка все имена пользователей и убедитесь в том, что программа выводит правильное сообщение.

5.10. Проверка имен пользователей: выполните следующие действия для создания программы, моделирующей проверку уникальности имен пользователей.

- Создайте список `current_users`, содержащий пять и более имен пользователей.
- Создайте другой список, `new_users`, содержащий пять имен пользователей. Убедитесь в том, что одно или два новых имени также присутствуют в списке `current_users`.
- Переберите список `new_users` и для каждого имени в этом списке проверьте, было ли оно использовано ранее. Если имя уже использовалось, выведите сообщение о том, что пользователь должен выбрать новое имя. Если имя не использовалось, выведите сообщение о его доступности.

- Проследите за тем, чтобы сравнение выполнялось без учета регистра символов. Если имя `'John'` уже используется, в регистрации имени `'JOHN'` следует отказать. (Для этого необходимо создать копию `current_users`, содержащую версию всех существующих имен пользователей в нижнем регистре.)

5.11. Порядковые числительные: порядковые числительные в английском языке заканчиваются суффиксом th (кроме 1st, 2nd и 3rd).

- Сохраните числа от 1 до 9 в списке.
- Переберите элементы списка.
- Используйте цепочку `if-elif-else` в цикле для вывода правильного окончания числительного для каждого числа. Программа должна выводить числительные `"1st 2nd 3rd 4th 5th 6th 7th 8th 9th"`, причем каждый результат должен располагаться в отдельной строке.

Оформление команд if

Во всех примерах этой главы применялись правила стилевого оформления. В PEP 8 приведена только одна рекомендация, касающаяся проверки условий: заключать операторы сравнения (такие, как `==`, `>=`, `<=` и т. д.) в одиночные пробелы. Например, запись

```
if age < 4:
```

лучше, чем:

```
if age<4:
```

Пробелы не влияют на интерпретацию вашего кода Python; они только упрощают чтение кода вами и другими разработчиками.

УПРАЖНЕНИЯ

5.12. Стиль оформления команд if: проанализируйте программы, написанные в этой главе, и проверьте, правильно ли вы оформляли условия.

5.13. Ваши идеи: к этому моменту вы уже стали более квалифицированным программистом, чем когда начинали читать книгу. Теперь вы лучше представляете, как в программах моделируются явления реального мира, и сможете сами придумать задачи, которые будут решаться в ваших программах. Запишите несколько задач, которые вам хотелось бы решить по мере роста вашего профессионального мастерства. Может быть, это какие-то компьютерные игры, задачи анализа наборов данных или веб-приложения?

Итоги

В этой главе вы научились писать условия, результатом которых всегда является логическое значение (`True` или `False`). Вы научились писать простые команды `if`, цепочки `if-else` и цепочки `if-elif-else`. Вы начали использовать эти структуры

для выявления конкретных условий, которые необходимо проверить, и проверки этих условий в ваших программах. Вы узнали, как обеспечить специальную обработку некоторых элементов в списке, с сохранением эффективности циклов `for`. Также мы вернулись к стилевым рекомендациям Python, с которыми более сложные программы становятся относительно простыми для чтения и понимания.

В главе 6 рассматриваются *словари* Python. Словарь отчасти напоминает список, но он позволяет связывать разные виды информации. Вы научитесь создавать словари, перебирать их элементы, использовать их в сочетании со списками и командами `if`. Словари помогут вам моделировать еще более широкий спектр реальных ситуаций.

6 Словари

В этой главе речь пойдет о словарях — структурах данных, предназначенных для объединения взаимосвязанной информации. Вы узнаете, как получить доступ к информации, хранящейся в словаре, и как изменить эту информацию. Так как объем данных в словаре практически безграничен, мы рассмотрим средства перебора данных в словарях. Кроме того, вы научитесь использовать вложенные словари в списках, вложенные списки в словарях и даже словари в других словарях.

Операции со словарями позволяют моделировать всевозможные реальные объекты с большей точностью. Вы узнаете, как создать словарь, описывающий человека, и сохранить в нем сколько угодно информации об этом человеке. В словаре может храниться имя, возраст, место жительства, профессия и любые другие атрибуты. Вы узнаете, как сохранить любые два вида информации, способные образовать пары: список слов и их значений, список имен людей и их любимых чисел, список гор и их высот и т. д.

Простой словарь

Возьмем игру с инопланетными пришельцами, которые имеют разные цвета и приносят разное количество очков игроку. В следующем простом словаре хранится информация об одном конкретном пришельце:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

В словаре `alien_0` хранятся два атрибута: цвет (`color`) и количество очков (`points`). Следующие две команды `print` читают эту информацию из словаря и выводят ее на экран:

```
green
5
```

Работа со словарями, как и большинство других новых концепций, требует определенного опыта. Стоит вам немного поработать со словарями, и вы увидите, как эффективно они работают при моделировании реальных ситуаций.

Работа со словарями

Словарь в языке Python представляет собой совокупность пар «ключ-значение». Каждый ключ связывается с некоторым значением, и программа может получить значение, связанное с заданным ключом. Значением может быть число, строка, список и даже другой словарь. Собственно, *любой* объект, создаваемый в программе Python, может стать значением в словаре.

В Python словарь заключается в фигурные скобки `{}`, в которых приводится последовательность пар «ключ-значение», как в предыдущем примере:

```
alien_0 = {'color': 'green', 'points': 5}
```

Пара «ключ-значение» представляет данные, связанные друг с другом. Если вы укажете ключ, то Python вернет значение, связанное с этим ключом. Ключ отделяется от значения двоеточием, а отдельные пары разделяются запятыми. В словаре может храниться любое количество пар «ключ-значение».

Простейший словарь содержит ровно одну пару «ключ-значение», как в следующей измененной версии словаря `alien_0`:

```
alien_0 = {'color': 'green'}
```

В этом словаре хранится ровно один фрагмент информации о пришельце `alien_0`, а именно его цвет. Строка `'color'` является ключом в словаре; с этим ключом связано значение `'green'`.

Обращение к значениям в словаре

Чтобы получить значение, связанное с ключом, укажите имя словаря, а затем ключ в квадратных скобках:

alien.py

```
alien_0 = {'color': 'green'}
print(alien_0['color'])
```

Эта конструкция возвращает значение, связанное с ключом `'color'` из словаря `alien_0`:

```
green
```

Количество пар «ключ-значение» в словаре не ограничено. Например, вот как выглядит исходный словарь `alien_0` с двумя парами «ключ-значение»:

```
alien_0 = {'color': 'green', 'points': 5}
```

Теперь программа может получить значение, связанное с любым из ключей в `alien_0`: `color` или `points`. Если игрок сбивает корабль пришельца, то для получения количества заработанных им очков может использоваться код следующего вида:

```
  alien_0 = {'color': 'green', 'points': 5}

❶ new_points = alien_0['points']
❷ print(f"You just earned {new_points} points!")
```

После того как словарь будет определен, код ❶ извлекает значение, связанное с ключом `'points'`, из словаря. Затем это значение сохраняется в переменной `new_points`. Строка ❷ преобразует целое значение в строку и выводит сообщение с количеством заработанных очков:

```
You just earned 5 points!
```

Если этот код будет выполняться каждый раз, когда игрок сбивает очередного пришельца, программа будет получать правильное количество очков.

Добавление новых пар «ключ-значение»

Словари относятся к динамическим структурам данных: в словарь можно в любой момент добавлять новые пары «ключ-значение». Для этого указывается имя словаря, за которым в квадратных скобках следует новый ключ с новым значением.

Добавим в словарь `alien_0` еще два атрибута: координаты `x` и `y` для вывода изображения пришельца в определенной позиции экрана. Допустим, пришелец должен отображаться у левого края экрана, в 25 пикселах от верхнего края. Так как система экранных координат обычно располагается в левом верхнем углу, для размещения пришельца у левого края координата `x` должна быть равна 0, а координата `y` — 25:

alien.py

```
  alien_0 = {'color': 'green', 'points': 5}
  print(alien_0)

❶ alien_0['x_position'] = 0
❷ alien_0['y_position'] = 25
  print(alien_0)
```

Программа начинается с определения того же словаря, с которым мы уже работали ранее. После этого выводится «снимок» текущего состояния словаря. В точке ❶ в словарь добавляется новая пара «ключ-значение»: ключ `'x_position'` и значение `0`. То же самое делается для ключа `'y_position'` в точке ❷. При выводе измененного словаря мы видим две дополнительные пары «ключ-значение»:

```
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'y_position': 25, 'x_position': 0}
```

Окончательная версия словаря содержит четыре пары «ключ-значение». Первые две определяют цвет и количество очков, а другие две — координаты.

ПРИМЕЧАНИЕ В версии Python 3.7 словари сохраняют исходный порядок добавления пар «ключ-значение». Когда вы выводите словарь или перебираете его элементы, вы будете получать элементы в том порядке, в каком они добавлялись в словарь.

Создание пустого словаря

В некоторых ситуациях бывает удобно (или даже необходимо) начать с пустого словаря, а затем добавлять в него новые элементы. Чтобы начать заполнение пустого словаря, определите словарь с пустой парой фигурных скобок, а затем добавляйте новые пары «ключ-значение» (каждая пара в отдельной строке). Например, вот как строится словарь `alien_0`:

alien.py

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Программа определяет пустой словарь `alien_0`, после чего добавляет в него значения для цвета и количества очков. В результате создается словарь, который использовался в предыдущих примерах:

```
{'color': 'green', 'points': 5}
```

Обычно пустые словари используются при хранении данных, введенных пользователем, или при написании кода, автоматически генерирующего большое количество пар «ключ-значение».

Изменение значений в словаре

Чтобы изменить значение в словаре, укажите имя словаря с ключом в квадратных скобках, а затем новое значение, которое должно быть связано с этим ключом. Допустим, в процессе игры цвет пришельца меняется с зеленого на желтый:

alien.py

```
alien_0 = {'color': 'green'}
print(f"The alien is {alien_0['color']}.")

alien_0['color'] = 'yellow'
print(f"The alien is now {alien_0['color']}.")
```

Сначала определяется словарь `alien_0`, который содержит только цвет пришельца; затем значение, связанное с ключом `'color'`, меняется на `'yellow'`. Из выходных данных видно, что цвет пришельца действительно сменился с зеленого на желтый:

```
The alien is green.
The alien is now yellow.
```

Рассмотрим более интересный пример: отслеживание позиции пришельца, который может двигаться с разной скоростью. Мы сохраним значение, представляющее текущую скорость пришельца, и используем его для определения величины горизонтального смещения:

```
  alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}
  print(f"Original position: {alien_0['x_position']}")

 # Пришелец перемещается вправо.
 # Вычисляем величину смещения на основании текущей скорости.
❶ if alien_0['speed'] == 'slow':
      x_increment = 1
  elif alien_0['speed'] == 'medium':
      x_increment = 2
  else:
      # Пришелец двигается быстро.
      x_increment = 3

  # Новая позиция равна сумме старой позиции и приращения.
❷ alien_0['x_position'] = alien_0['x_position'] + x_increment

  print(f"New position: {alien_0['x_position']}")
```

Сначала определяется словарь с исходной позицией (координаты x и y) и скоростью `'medium'`. Значения цвета и количества очков для простоты опущены, но с ними этот пример работал бы точно так же. Аналогично выводится исходное значение `x_position`.

В точке ❶ цепочка `if-elif-else` определяет, на какое расстояние пришелец должен переместиться вправо; полученное значение сохраняется в переменной `x_increment`. Если пришелец двигается медленно (`'slow'`), то он перемещается на одну единицу вправо; при средней скорости (`'medium'`) он перемещается на две единицы вправо; наконец, при высокой скорости (`'fast'`) он перемещается на три единицы вправо. Вычисленное смещение прибавляется к значению `x_position` в ❷, а результат сохраняется в словаре с ключом `x_position`.

Для пришельца со средней скоростью позиция смещается на две единицы:

```
Original x-position: 0
New x-position: 2
```

Получается, что изменение одного значения в словаре изменяет все поведение пришельца. Например, чтобы превратить пришельца со средней скоростью в быстрого, добавьте следующую строку:

```
alien_0['speed'] = fast
```

При следующем выполнении кода блок `if-elif-else` присвоит `x_increment` большее значение.

Удаление пар «ключ-значение»

Когда информация, хранящаяся в словаре, перестает быть нужной, пару «ключ-значение» можно полностью удалить при помощи команды `del`. При вызове достаточно передать имя словаря и удаляемый ключ.

Например, в следующем примере из словаря `alien_0` удаляется ключ `'points'` вместе со значением:

alien.py

```
  alien_0 = {'color': 'green', 'points': 5}
  print(alien_0)

❶ del alien_0['points']
  print(alien_0)
```

Строка ❶ приказывает Python удалить ключ `'points'` из словаря `alien_0`, а также удалить значение, связанное с этим ключом. Из вывода видно, что ключ `'points'` и его значение 5 исчезли из словаря, но остальные данные остались без изменений:

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

ПРИМЕЧАНИЕ Учтите, что удаление пары «ключ-значение» отменить уже не удастся.

Словарь с однотипными объектами

В предыдущем примере в словаре сохранялась разнообразная информация об одном объекте (пришельце из компьютерной игры). Словарь также может использоваться для хранения одного вида информации о многих объектах. Допустим, вы хотите провести опрос среди коллег и узнать их любимый язык программирования. Результаты простого опроса удобно сохранить в словаре:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }
```

Пары в словаре в этой записи разбиты по строкам. Ключами являются имена участников опроса, а значениями — выбранные ими языки. Если вы знаете, что для определения словаря потребуется более одной строки, нажмите клавишу Enter после ввода открывающей фигурной скобки. Снабдите следующую строку отступом на один уровень (четыре пробела) и запишите первую пару «ключ-

значение», поставив за ней запятую. После этого при нажатии Enter ваш текстовый редактор будет автоматически снабжать все последующие пары таким же отступом, как у первой.

Завершив определение словаря, добавьте закрывающую фигурную скобку в новой строке после последней пары «ключ-значение» и снабдите ее отступом на один уровень, чтобы она была выровнена по ключам. За последней парой также рекомендуется поставить запятую, чтобы при необходимости вы смогли легко добавить новую пару «ключ-значение» в следующей строке.

ПРИМЕЧАНИЕ Во многих редакторах предусмотрены функции, упрощающие форматирование расширенных списков и словарей в описанном стиле. Также существуют другие распространенные способы форматирования длинных словарей — вы можете столкнуться с ними в вашем редакторе или в другом источнике.

Для заданного имени участника опроса этот словарь позволяет легко определить его любимый язык:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

❶ language = favorite_languages['sarah'].title()
print(f"Sarah's favorite language is {language}.")
```

Чтобы узнать, какой язык выбран пользователем с именем `Sarah`, мы запрашиваем следующее значение:

```
favorite_languages['sarah']
```

Этот синтаксис используется для получения соответствующего языка программирования из словаря ❶ и присваивания его переменной `language`. Создание новой переменной существенно упрощает вызов `print()`. В выходных данных показывается значение, связанное с ключом:

```
Sarah's favorite language is C.
```

Тот же синтаксис может использоваться с любым участником опроса, содержащимся в словаре.

Обращение к значениям методом get()

Использование синтаксиса с ключом в квадратных скобках для получения интересующего вас значения из словаря имеет один потенциальный недостаток: если запрашиваемый ключ не существует, то вы получите сообщение об ошибке.

Посмотрим, что произойдет при запросе количества очков для пришельца, для которого оно не задано:

alien_no_points.py

```
alien_0 = {'color': 'green', 'speed': 'slow'}
print(alien_0['points'])
```

На экране появляется трассировка с сообщением об ошибке `KeyError`:

```
Traceback (most recent call last):
  File "alien_no_points.py", line 2, in <module>
    print(alien_0['points'])
KeyError: 'points'
```

Более общие способы обработки подобных ошибок рассматриваются в главе 10. Конкретно для словарей можно воспользоваться методом `get()` для назначения значения по умолчанию, которое будет возвращено при отсутствии заданного ключа в словаре.

В первом аргументе метода `get()` передается ключ. Во втором необязательном аргументе можно передать значение, которое должно возвращаться при отсутствии ключа:

```
alien_0 = {'color': 'green', 'speed': 'slow'}

point_value = alien_0.get('points', 'No point value assigned.')
print(point_value)
```

Если ключ `'points'` существует в словаре, вы получите соответствующее значение; если нет — будет получено значение по умолчанию. В данном случае ключ `'points'` не существует, поэтому вместо ошибки выводится понятное сообщение:

```
No point value assigned.
```

Если есть вероятность того, что запрашиваемый ключ не существует, возможно, стоит использовать метод `get()` вместо синтаксиса с квадратными скобками.

ПРИМЕЧАНИЕ Если второй аргумент при вызове `get()` опущен, а ключ не существует, то Python вернет специальное значение `None` — признак того, что значение не существует. Это не ошибка, а специальное значение, указывающее на отсутствие значения. Другие применения `None` описаны в главе 8.

УПРАЖНЕНИЯ

6.1. Человек: используйте словарь для сохранения информации об известном вам человеке. Сохраните имя, фамилию, возраст и город, в котором живет этот человек. Словарь должен содержать ключи с такими именами, как `first_name`, `last_name`, `age` и `city`. Выведите каждый фрагмент информации, хранящийся в словаре.

6.2. Любимые числа: используйте словарь для хранения любимых чисел. Возьмите пять имен и используйте их как ключи словаря. Придумайте любимое число для каждого человека и сохраните его как значение в словаре. Выведите имя каждого человека и его любимое число. Чтобы задача стала более интересной, опросите нескольких друзей и соберите реальные данные для своей программы.

6.3. Глоссарий: словари Python могут использоваться для моделирования «настоящего» словаря (чтобы не создавать путаницу, назовем его глоссарием):

- Вспомните пять терминов из области программирования, которые вы узнали в предыдущих главах. Используйте эти слова как ключи глоссария, а их определения — как значения.
- Выведите каждое слово и его определение в аккуратно отформатированном виде. Например, вы можете вывести слово, затем двоеточие и определение или же слово в одной строке, а его определение — с отступом в следующей строке. Используйте символ новой строки (`\n`) для вставки пустых строк между парами «слово — определение» в выходных данных.

Перебор словаря

Словарь Python может содержать как несколько пар «ключ-значение», так и миллионы таких пар. Поскольку словарь может содержать большие объемы данных, Python предоставляет средства для перебора элементов словаря. Информация может храниться в словарях по-разному, поэтому предусмотрены разные способы перебора. Программа может перебрать все пары «ключ-значение» в словаре, только ключи или только значения.

Перебор всех пар «ключ-значение»

Прежде чем рассматривать разные способы перебора, рассмотрим новый словарь, предназначенный для хранения информации о пользователе веб-сайта. В следующем словаре хранится имя пользователя, его имя и фамилия:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }
```

То, что вы уже узнали в этой главе, позволит вам обратиться к любому отдельному атрибуту `user_0`. Но что, если вы хотите просмотреть *все* данные из словаря этого пользователя? Для этого можно воспользоваться перебором в цикле `for`:

user.py

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
```

```
    }

❶ for key, value in user_0.items():
❷     print(f"\nKey: {key}")
❸     print(f"Value: {value}")
```

Как мы видим в точке ❶, чтобы написать цикл `for` для словаря, необходимо создать имена для двух переменных, в которых будет храниться ключ и значение из каждой пары «ключ-значение». Этим двум переменным можно присвоить любые имена — с короткими однобуквенными именами код будет работать точно так же:

```
for k, v in user_0.items()
```

Вторая половина команды `for` в точке ❶ включает имя словаря, за которым следует вызов метода `items()`, возвращающий список пар «ключ-значение». Цикл `for` сохраняет компоненты пары в двух указанных переменных. В предыдущем примере мы используем переменные для вывода каждого ключа ❷, за которым следует связанное значение ❸. `"\n"` в первой команде `print` гарантирует, что перед каждой парой «ключ-значение» в выводе будет вставлена пустая строка:

```
Key: last
Value: fermi

Key: first
Value: enrico

Key: username
Value: efermi
```

Перебор всех пар «ключ-значение» особенно хорошо работает для таких словарей, как в примере `favorite_languages.py` на с. ___: то есть для словарей, хранящих один вид информации со многими разными ключами. Перебрав словарь `favorite_languages`, вы получите имя каждого человека и его любимый язык программирования. Так как ключ всегда содержит имя, а значение — язык программирования, в цикле вместо имен `key` и `value` используются переменные `name` и `language`. С таким выбором имен читателю кода будет проще следить за тем, что происходит в цикле:

favorite_languages.py

```
  favorite_languages = {
      'jen': 'python',
      'sarah': 'c',
      'edward': 'ruby',
      'phil': 'python',
      }

❶ for name, language in favorite_languages.items():
❷     print(f"{name.title()}'s favorite language is {language.title()}.")
```

Код в точке ❶ приказывает Python перебрать все пары «ключ-значение» в словаре. В процессе перебора пар ключ сохраняется в переменной `name`, а значение — в пере-

менной language. С этими содержательными именами намного проще понять, что делает команда print в точке ❷.

Всего в нескольких строках кода выводится вся информация из опроса:

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Ruby.
Phil's favorite language is Python.
```

Такой способ перебора точно так же работает и в том случае, если в словаре будут храниться результаты опроса тысяч и даже миллионов людей.

Перебор всех ключей в словаре

Метод keys() удобен в тех случаях, когда вы не собираетесь работать со всеми значениями в словаре. Переберем словарь favorite_languages и выведем имена всех людей, участвовавших в опросе:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

❶ for name in favorite_languages.keys():
    print(name.title())
```

Строка ❶ приказывает Python извлечь из словаря favorite_languages все ключи и последовательно сохранять их в переменной name. В выходных данных представлены имена всех людей, участвовавших в опросе:

```
Jen
Sarah
Edward
Phil
```

На самом деле перебор ключей используется по умолчанию при переборе словаря, так что этот код будет работать точно так же, как если бы вы написали

```
for name in favorite_languages:
```

вместо...

```
for name in favorite_languages.keys():
```

Используйте явный вызов метода keys(), если вы считаете, что он упростит чтение вашего кода, или опустите его при желании.

Чтобы обратиться в цикле к значению, связанному с интересующим вас ключом, используйте текущий ключ. Для примера выведем для пары друзей сообщение о выбранном ими языке. Мы переберем имена в словаре, как это делалось ранее, но когда имя совпадает с именем одного из друзей, программа будет выводить специальное сообщение об их любимом языке:

```
  favorite_languages = {
  ...
  }

❶ friends = ['phil', 'sarah']
  for name in favorite_languages.keys():
      print(name.title())

❷     if name in friends:
❸         language = favorite_languages[name].title()
          print(f"\t{name.title()}, I see you love {language}!")
```

В точке ❶ строится список друзей, для которых должно выводиться сообщение. В цикле выводится имя очередного участника опроса, а затем в точке ❷ программа проверяет, входит ли текущее имя в список `friends`. Если имя входит в список, выводится специальное приветствие с упоминанием выбранного языка ❸.

Выводятся все имена, но для наших друзей выводится специальное сообщение:

```
Hi Jen.
Hi Sarah.
    Sarah, I see you love C!
Hi Edward.
Hi Phil.
    Phil, I see you love Python!
```

Метод `keys()` также может использоваться для проверки того, участвовал ли конкретный человек в опросе:

```
  favorite_languages = {
      'jen': 'python',
      'sarah': 'c',
      'edward': 'ruby',
      'phil': 'python',
      }

❶ if 'erin' not in favorite_languages.keys():
      print("Erin, please take our poll!")
```

Метод `keys()` не ограничивается перебором: он возвращает список всех ключей, и строка ❶ просто проверяет, входит ли ключ `'erin'` в список. Так как ключ в списке отсутствует, программа выводит сообщение:

```
Erin, please take our poll!
```

Перебор ключей словаря в определенном порядке

Начиная с Python версии 3.7, перебор содержимого словаря возвращает элементы в том порядке, в каком они вставлялись. Тем не менее иногда требуется перебрать элементы словаря в другом порядке.

Один из способов получения элементов в определенном порядке основан на сортировке ключей, возвращаемых циклом `for`. Для получения упорядоченной копии ключей можно воспользоваться функцией `sorted()`:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

for name in sorted(favorite_languages.keys()):
    print(f"{name.title()}, thank you for taking the poll.")
```

Эта команда `for` не отличается от других команд `for`, если не считать того, что метод `dictionary.keys()` заключен в вызов функции `sorted()`. Эта конструкция приказывает Python выдать список всех ключей в словаре и отсортировать его перед тем, как перебирать элементы. В выводе перечислены все пользователи, участвовавшие в опросе, а их имена упорядочены по алфавиту:

```
Edward, thank you for taking the poll.
Jen, thank you for taking the poll.
Phil, thank you for taking the poll.
Sarah, thank you for taking the poll.
```

Перебор всех значений в словаре

Если вас прежде всего интересуют значения, содержащиеся в словаре, используйте метод `values()` для получения списка значений без ключей. Допустим, вы хотите просто получить список всех языков, выбранных в опросе, и вас не интересуют имена людей, выбравших каждый язык:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

print("The following languages have been mentioned:")
for language in favorite_languages.values():
    print(language.title())
```

Команда `for` читает каждое значение из словаря и сохраняет его в переменной `language`. При выводе этих значений будет получен список всех выбранных языков:

```
The following languages have been mentioned:
Python
C
Python
Ruby
```

Значения извлекаются из словаря без проверки на возможные повторения. Для небольших словарей это может быть приемлемо, но в опросах с большим количеством респондентов список будет содержать слишком много дубликатов. Чтобы получить список выбранных языков без повторений, можно воспользоваться *множеством* (set). Множество в целом похоже на список, но все его элементы должны быть уникальными:

```
  favorite_languages = {
      ...
      }

  print("The following languages have been mentioned:")
❶ for language in set(favorite_languages.values()):
      print(language.title())
```

Когда список, содержащий дубликаты, заключается в вызов `set()`, Python находит уникальные элементы списка и строит множество из этих элементов. В точке ❶ `set()` используется для извлечения уникальных языков из `favorite_languages.values()`.

В результате создается не содержащий дубликатов список языков программирования, упомянутых участниками опроса:

```
The following languages have been mentioned:
Python
C
Ruby
```

В ходе дальнейшего изучения Python вы часто будете обнаруживать встроенные возможности языка, которые помогают сделать с данными именно то, что вам требуется.

ПРИМЕЧАНИЕ Множество можно построить прямо в фигурных скобках с разделением элементов запятыми:

```
>>> languages = {'python', 'ruby', 'python', 'c'}
>>> languages
{'ruby', 'python', 'c'}
```

Словари легко перепутать с множествами, потому что обе структуры заключаются в фигурные скобки. Когда вы видите фигурные скобки без пар «ключ-значение», скорее всего, перед вами множество. В отличие от списков и словарей, элементы множеств не хранятся в каком-либо определенном порядке.

УПРАЖНЕНИЯ

6.4. Глоссарий 2: теперь, когда вы знаете, как перебрать элементы словаря, упростите код из упражнения 6.3, заменив серию команд `print` циклом, перебирающим ключи и значения словаря. Когда вы будете уверены в том, что цикл работает, добавьте в глоссарий еще пять терминов Python. При повторном запуске программы новые слова и значения должны быть автоматически включены в вывод.

6.5. Реки: создайте словарь с названиями трех больших рек и стран, по которым протекает каждая река. Одна из возможных пар «ключ-значение» — `'nile': 'egypt'`.

- Используйте цикл для вывода сообщения с упоминанием реки и страны — например, «The Nile runs through Egypt».
- Используйте цикл для вывода названия каждой реки, включенной в словарь.
- Используйте цикл для вывода названия каждой страны, включенной в словарь.

6.6. Опрос: возьмите за основу код favorite_languages.py (с. 115).

- Создайте список людей, которые должны участвовать в опросе по поводу любимого языка программирования. Включите некоторые имена, которые уже присутствуют в списке, и некоторые имена, которых в списке еще нет.
- Переберите список людей, которые должны участвовать в опросе. Если они уже прошли опрос, выведите сообщение с благодарностью за участие. Если они еще не проходили опрос, выведите сообщение с предложением принять участие.

Вложение

Иногда бывает нужно сохранить множество словарей в списке или сохранить список как значение элемента словаря. Создание сложных структур такого рода называется *вложением*. Вы можете вложить множество словарей в список, список элементов в словарь или даже словарь внутрь другого словаря. Как наглядно показывают следующие примеры, вложение — чрезвычайно мощный механизм.

Список словарей

Словарь `alien_0` содержит разнообразную информацию об одном пришельце, но в нем нет места для хранения информации о втором пришельце, не говоря уже о целом экране, забитом пришельцами. Как смоделировать флот вторжения? Например, можно создать список пришельцев, в котором каждый элемент представляет собой словарь с информацией о пришельце. Например, следующий код строит список из трех пришельцев:

aliens.py

```
alien_0 = {'color': 'green', 'points': 5}
alien_1 = {'color': 'yellow', 'points': 10}
alien_2 = {'color': 'red', 'points': 15}
```

```
❶ aliens = [alien_0, alien_1, alien_2]

  for alien in aliens:
      print(alien)
```

Сначала создаются три словаря, каждый из которых представляет отдельного пришельца. В точке ❶ каждый словарь заносится в список с именем `aliens`. Наконец, программа перебирает список и выводит каждого пришельца:

```
{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}
```

Конечно, в реалистичном примере будут использоваться более трех пришельцев, которые будут генерироваться автоматически. В следующем примере функция `range()` создает флот из 30 пришельцев:

```
# Создание пустого списка для хранения пришельцев.
aliens = []

# Создание 30 зеленых пришельцев.
❶ for alien_number in range(30):
❷     new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
❸     aliens.append(new_alien)

  # Вывод первых 5 пришельцев:
❹ for alien in aliens[:5]:
      print(alien)
  print("...")

  # Вывод количества созданных пришельцев.
❺ print(f"Total number of aliens: {len(aliens)}")
```

В начале примера список для хранения всех пришельцев, которые будут созданы, пуст. В точке ❶ функция `range()` возвращает множество чисел, которое просто сообщает Python, сколько раз должен повторяться цикл. При каждом выполнении цикла создается новый пришелец ❷, который затем добавляется в список `aliens` ❸. В точке ❹ сегмент используется для вывода первых пяти пришельцев, а в точке ❺ выводится длина списка (для демонстрации того, что программа действительно сгенерировала весь флот из 30 пришельцев):

```
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
...

Total number of aliens: 30
```

Все пришельцы обладают одинаковыми характеристиками, но Python рассматривает каждого пришельца как отдельный объект, что позволяет изменять атрибуты каждого владельца по отдельности.

Как работать с таким множеством? Представьте, что в этой игре некоторые пришельцы изменяют цвет и начинают двигаться быстрее. Когда приходит время смены цветов, мы можем воспользоваться циклом `for` и командой `if` для изменения цвета. Например, чтобы превратить первых трех пришельцев в желтых, двигающихся со средней скоростью и приносящих игроку по 10 очков, можно действовать так:

```
# Создание пустого списка для хранения пришельцев.
aliens = []

# Создание 30 зеленых пришельцев.
for alien_number in range (0,30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[0:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

# Вывод первых 5 пришельцев:
for alien in aliens[0:5]:
    print(alien)
print("...")
```

Чтобы изменить первых трех пришельцев, мы перебираем элементы сегмента, включающего только первых трех пришельцев. В данный момент все пришельцы зеленые (`'green'`), но так будет не всегда, поэтому мы пишем команду `if`, которая гарантирует, что изменяться будут только зеленые пришельцы. Если пришелец зеленый, то его цвет меняется на желтый (`'yellow'`), скорость — на среднюю (`'medium'`), а награда увеличивается до 10 очков:

```
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
...
```

Цикл можно расширить, добавив блок `elif` для превращения желтых пришельцев в красных — быстрых и приносящих игроку по 15 очков. Мы не станем приводить весь код, а цикл выглядит так:

```
for alien in aliens[0:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10
```

```
    elif alien['color'] == 'yellow':
        alien['color'] = 'red'
        alien['speed'] = 'fast'
        alien['points'] = 15
```

Решение с хранением словарей в списке достаточно часто встречается тогда, когда каждый словарь содержит разные атрибуты одного объекта. Например, вы можете создать словарь для каждого пользователя сайта, как это было сделано в программе user.py на с. 114, и сохранить отдельные словари в списке с именем `users`. Все словари в списке должны иметь одинаковую структуру, чтобы вы могли перебрать список и выполнить с каждым объектом словаря одни и те же операции.

Список в словаре

Вместо того чтобы помещать словарь в список, иногда бывает удобно поместить список в словарь. Представьте, как бы вы описали в программе заказанную пиццу. Если ограничиться только списком, сохранить удастся разве что список топпингов к пицце. При использовании словаря список топпингов может быть всего лишь одним аспектом описания пиццы.

В следующем примере для каждой пиццы сохраняются два вида информации: основа и список топпингов. Список топпингов представляет собой значение, связанное с ключом `'toppings'`. Чтобы использовать элементы в списке, нужно указать имя словаря и ключ `'toppings'`, как и для любого другого значения в словаре. Вместо одного значения будет получен список топпингов:

pizza.py

```
  # Сохранение информации о заказанной пицце.
❶ pizza = {
      'crust': 'thick',
      'toppings': ['mushrooms', 'extra cheese'],
      }

  # Описание заказа.
❷ print(f"You ordered a {pizza['crust']}-crust pizza "
      "with the following toppings:")

❸ for topping in pizza['toppings']:
      print("\t" + topping)
```

Работа начинается в точке ❶ со словаря с информацией о заказанной пицце. С ключом в словаре `'crust'` связано строковое значение `'thick'`. С другим ключом `'toppings'` связано значение-список, в котором хранятся все заказанные топпинги. В точке ❷ выводится описание заказа перед созданием пиццы. Если вам нужно разбить длинную строку в вызове `print()`, выберите точку для разбиения выводимой строки и закончите строку кавычкой. Снабдите следующую строку отступом, добавьте открывающую кавычку и продолжите строку. Python автоматически объединяет все строки, обнаруженные в круглых скобках. Для вывода дополне-

ний пишется цикл `for` ❸. Чтобы вывести список топпингов, мы используем ключ `'toppings'`, а Python берет список топпингов из словаря.

Следующее сообщение описывает пиццу, которую мы собираемся создать:

```
You ordered a thick-crust pizza with the following toppings:
    mushrooms
    extra cheese
```

Вложение списка в словарь может применяться каждый раз, когда с одним ключом словаря должно быть связано более одного значения. Если бы в предыдущем примере с языками программирования ответы сохранялись в списке, один участник опроса мог бы выбрать сразу несколько любимых языков. При переборе словаря значение, связанное с каждым человеком, представляло бы собой список языков (вместо одного языка). В цикле `for` словаря создается другой цикл для перебора списка языков, связанных с каждым участником:

favorite_languages.py

```
❶ favorite_languages = {
      'jen': ['python', 'ruby'],
      'sarah': ['c'],
      'edward': ['ruby', 'go'],
      'phil': ['python', 'haskell'],
      }

❷ for name, languages in favorite_languages.items():
      print(f"\n{name.title()}'s favorite languages are:")
❸     for language in languages:
          print(f"\t{language.title()}")
```

Вы видите в точке ❶, что значение, связанное с каждым именем, теперь представляет собой список. У некоторых участников один любимый язык программирования, у других таких языков несколько. При переборе словаря в точке ❷ переменная с именем `languages` используется для хранения каждого значения из словаря, потому что мы знаем, что каждое значение будет представлять собой список. В основном цикле по элементам словаря другой цикл ❸ перебирает элементы списка любимых языков каждого участника. Теперь каждый участник опроса может указать сколько угодно любимых языков программирования:

```
Jen's favorite languages are:
    Python
    Ruby

Sarah's favorite languages are:
    C

Phil's favorite languages are:
    Python
    Haskell
```

```
Edward's favorite languages are:
    Ruby
    Go
```

Чтобы дополнительно усовершенствовать программу, включите в начало цикла `for` словаря команду `if` для проверки того, выбрал ли данный участник более одного языка программирования (проверка основана на значении `len(languages)`). Если у участника только один любимый язык, текст сообщения изменяется для единственного числа (например, «Sarah's favorite language is C»).

ПРИМЕЧАНИЕ Глубина вложения списков и словарей не должна быть слишком большой. Если вам приходится вкладывать элементы на глубину существенно большую, чем в предыдущих примерах, или если вы работаете с чужим кодом со значительной глубиной вложения, скорее всего, у задачи существует более простое решение.

Словарь в словаре

Словарь также можно вложить в другой словарь, но в таких случаях код быстро усложняется. Например, если на сайте есть несколько пользователей с уникальными именами, вы можете использовать имена пользователей как ключи в словаре. Информация о каждом пользователе при этом хранится в словаре, который используется как значение, связанное с именем. В следующем примере о каждом пользователе хранится три вида информации: имя, фамилия и место жительства. Чтобы получить доступ к этой информации, переберите имена пользователей и словарь с информацией, связанной с каждым именем:

many_users.py

```
  users = {
      'aeinstein': {
          'first': 'albert',
          'last': 'einstein',
          'location': 'princeton',
          },

      'mcurie': {
          'first': 'marie',
          'last': 'curie',
          'location': 'paris',
          },
      }

❶ for username, user_info in users.items():
❷     print(f"\nUsername: {username}")
❸     full_name = f"{user_info['first']} {user_info['last']}"
      location = user_info['location']
❹     print(f"\tFull name: {full_name.title()}")
      print(f"\tLocation: {location.title()}")
```

В программе определяется словарь с именем `users`, содержащий два ключа: для пользователей `'aeinstein'` и `'mcurie'`. Значение, связанное с каждым ключом, представляет собой словарь с именем, фамилией и местом жительства пользователя. В процессе перебора словаря `users` в точке ❶ Python сохраняет каждый ключ в переменной `username`, а словарь, связанный с каждым именем пользователя, сохраняется в переменной `user_info`. Внутри основного цикла в словаре выводится имя пользователя ❷.

В точке ❸ начинается работа с внутренним словарем. Переменная `user_info`, содержащая словарь с информацией о пользователе, содержит три ключа: `'first'`, `'last'` и `'location'`. Каждый ключ используется для построения аккуратно отформатированных данных с полным именем и местом жительства пользователя, с последующим выводом сводки известной информации о пользователе ❹:

```
Username: aeinstein
    Full name: Albert Einstein
    Location: Princeton

Username: mcurie
    Full name: Marie Curie
    Location: Paris
```

Обратите внимание на идентичность структур словарей всех пользователей. Хотя Python этого и не требует, наличие единой структуры упрощает работу с вложенными словарями. Если словари разных пользователей будут содержать разные ключи, то код в цикле `for` заметно усложнится.

УПРАЖНЕНИЯ

6.7. Люди: начните с программы, написанной для упражнения 6.1 (с. 113). Создайте два новых словаря, представляющих разных людей, и сохраните все три словаря в списке с именем `people`. Переберите элементы списка людей. В процессе перебора выведите всю имеющуюся информацию о каждом человеке.

6.8. Домашние животные: создайте несколько словарей, имена которых представляют клички домашних животных. В каждом словаре сохраните информацию о виде животного и имени владельца. Сохраните словари в списке с именем `pets`. Переберите элементы списка. В процессе перебора выведите всю имеющуюся информацию о каждом животном.

6.9. Любимые места: создайте словарь с именем `favorite_places`. Придумайте названия трех мест, которые станут ключами словаря, и сохраните для каждого человека от одного до трех любимых мест. Чтобы задача стала более интересной, опросите нескольких друзей и соберите реальные данные для своей программы. Переберите данные в словаре, выведите имя каждого человека и его любимые места.

6.10. Любимые числа: измените программу из упражнения 6.2 (с. 114), чтобы для каждого человека можно было хранить более одного любимого числа. Выведите имя каждого человека в списке и его любимые числа.

6.11. Города: создайте словарь с именем `cities`. Используйте названия трех городов в качестве ключей словаря. Создайте словарь с информацией о каждом городе; включите в него страну, в которой расположен город, примерную численность населения и один примеча-

тельный факт, относящийся к этому городу. Ключи словаря каждого города должны называться `country`, `population` и `fact`. Выведите название каждого города и всю сохраненную информацию о нем.

6.12. Расширение: примеры, с которыми мы работаем, стали достаточно сложными, и в них можно вносить разного рода усовершенствования. Воспользуйтесь одним из примеров этой главы и расширьте его: добавьте новые ключи и значения, измените контекст программы или улучшите форматирование вывода.

Итоги

В этой главе вы научились определять словари и работать с хранящейся в них информацией. Вы узнали, как обращаться к отдельным элементам словаря и изменять их, как перебрать всю информацию в словаре. Вы научились перебирать пары «ключ-значение», ключи и значения словаря. Также были рассмотрены возможности вложения словарей в список, вложения списков в словари и вложения словарей в другие словари.

В следующей главе будут рассмотрены циклы `while` и получение входных данных от пользователей программ. Эта глава будет особенно интересной, потому что вы наконец-то сможете сделать свои программы интерактивными: они начнут реагировать на действия пользователя.

7 Ввод данных и циклы while

Программы обычно пишутся для решения задач конечного пользователя. Для этого им нужна некоторая информация, которую должен ввести пользователь. Простой пример: допустим, пользователь хочет узнать, достаточен ли его возраст для голосования. Если вы пишете программу для ответа на этот вопрос, то вам нужно будет узнать возраст пользователя. Программа должна запросить у пользователя значение — его возраст; когда у программы появятся данные, она может сравнить их с возрастом, дающим право на голосование, и сообщить результат.

В этой главе вы узнаете, как получить пользовательский ввод (то есть входные данные), чтобы программа могла работать с ним. Если программа хочет получить отдельное имя, она запрашивает отдельное имя; если ей нужен список имен — она также выводит соответствующее сообщение. Для получения данных в программах используется функция `input()`.

Вы также научитесь продолжать работу программы, пока пользователь вводит новые данные; после получения всех данных программа переходит к работе с полученной информацией. Цикл `while` в языке Python позволяет выполнять программу, пока некоторое условие остается истинным.

Когда вы научитесь работать с пользовательским вводом и управлять продолжительностью выполнения программы, вы сможете создавать полностью интерактивные программы.

Как работает функция input()

Функция `input()` приостанавливает выполнение программы и ожидает, пока пользователь введет некоторый текст. Получив ввод, Python сохраняет его в переменной, чтобы вам было удобнее работать с ним.

Например, следующая программа предлагает пользователю ввести текст, а затем выводит сообщение для пользователя:

parrot.py

```
message = input("Tell me something, and I will repeat it back to you: ")
print(message)
```

Функция `input()` получает один аргумент: текст подсказки (или инструкции), который выводится на экран, чтобы пользователь понимал, что от него требуется. В данном примере при выполнении первой строки пользователь видит подсказку с предложением ввести любой текст. Программа ожидает, пока пользователь введет ответ, и продолжает работу после нажатия Enter. Ответ сохраняется в переменной `message`, после чего вызов `print(message)` дублирует введенные данные:

```
Tell me something, and I will repeat it back to you: Hello everyone!
Hello everyone!
```

ПРИМЕЧАНИЕ Sublime Text и многие другие текстовые редакторы не запускают программы, запрашивающие входные данные у пользователя. Вы можете использовать эти редакторы для создания таких программ, но запускать их придется из терминального окна. См. «Запуск программ Python в терминале», с. 31.

Содержательные подсказки

Каждый раз, когда в вашей программе используется функция `input()`, вы должны включать четкую, понятную подсказку, которая точно сообщит пользователю, какую информацию вы от него хотите получить. Подойдет любое предложение, которое объяснит пользователю, что нужно вводить. Пример:

greeter.py

```
name = input("Please enter your name: ")
print(f"\nHello, {name}!")
```

Добавьте пробел в конце подсказки (после двоеточия в предыдущем примере), чтобы отделить подсказку от данных, вводимых пользователем, и четко показать, где должен вводиться текст. Пример:

```
Please enter your name: Eric
Hello, Eric!
```

Иногда подсказка занимает более одной строки. Например, вы можете сообщить пользователю, для чего программа запрашивает данные. Текст подсказки можно сохранить в переменной и передать эту переменную функции `input()`: вы строите длинное приглашение из нескольких строк, а потом выполняете одну компактную команду `input()`.

greeter.py

```
prompt = "If you tell us who you are, we can personalize the messages you see."
prompt += "\nWhat is your first name? "

name = input(prompt)
print(f"\nHello, {name}!")
```

В этом примере продемонстрирован один из способов построения длинных строк. Первая часть длинного сообщения сохраняется в переменной `prompt`. Затем оператор `+=` объединяет текст, хранящийся в `prompt`, с новым фрагментом текста.

Теперь содержимое `prompt` занимает две строки (вопросительный знак снова отделяется от ввода пробелом для наглядности):

```
If you tell us who you are, we can personalize the messages you see.
What is your first name? Eric

Hello, Eric!
```

Использование int() для получения числового ввода

При использовании функции `input()` Python интерпретирует все данные, введенные пользователем, как строку. В следующем сеансе интерпретатора программа запрашивает у пользователя возраст:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age
'21'
```

Пользователь вводит число 21, но когда мы запрашиваем у Python значение `age`, выводится `'21'` — представление введенного числа в строковом формате. Кавычки, в которые заключены данные, указывают на то, что Python интерпретирует ввод как строку. Но попытка использовать данные как число приведет к ошибке:

```
  >>> age = input("How old are you? ")
  How old are you? 21
❶ >>> age >= 18
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
❷ TypeError: unorderable types: str() >= int()
```

Когда вы пытаетесь сравнить введенные данные с числом в точке ❶, Python выдает ошибку, потому что не может сравнить строку с числом: строка `'21'`, хранящаяся в `age`, не сравнивается с числовым значением 18; происходит ошибка ❷.

Проблему можно решить при помощи функции `int()`, интерпретирующей строку как числовое значение. Функция `int()` преобразует строковое представление числа в само число:

```
  >>> age = input("How old are you? ")
  How old are you? 21
❶ >>> age = int(age)
  >>> age >= 18
  True
```

В этом примере введенный текст `21` интерпретируется как строка, но затем он преобразуется в числовое представление вызовом `int()` в точке ❶. Теперь Python

может проверить условие: сравнить переменную age (которая теперь содержит числовое значение 21) с 18. Условие «значение age больше или равно 18» выполняется, и результат проверки равен True.

Как использовать функцию int() в реальной программе? Допустим, программа проверяет рост пользователя и определяет, достаточен ли он для катания на аттракционе:

rollercoaster.py

```
height = input("How tall are you, in inches? ")
height = int(height)

if height >= 48:
    print("\nYou're tall enough to ride!")
else:
    print("\nYou'll be able to ride when you're a little older.")
```

Программа может сравнить height с 48, потому что строка height = int(height) преобразует входное значение в число перед проведением сравнения. Если введенное число больше или равно 36, программа сообщает пользователю, что он прошел проверку:

```
How tall are you, in inches? 71

You're tall enough to ride!
```

Если пользователь вводит числовые данные, которые используются в вашей программе для вычислений и сравнений, обязательно преобразуйте введенное значение в его числовой эквивалент.

Оператор вычисления остатка

При работе с числовыми данными может пригодиться *оператор вычисления остатка* (%), который делит одно число на другое и возвращает остаток:

```
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
```

Оператор % не сообщает частное от целочисленного деления; он возвращает только остаток.

Когда одно число нацело делится на другое, остаток равен 0, и оператор % возвращает 0. Например, этот факт может использоваться для проверки четности или нечетности числа:

even_or_odd.py

```
number = input("Enter a number, and I'll tell you if it's even or odd: ")
number = int(number)

if number % 2 == 0:
    print(f"\nThe number {number} is even.")
else:
    print(f"\nThe number {number} is odd.")
```

Четные числа всегда делятся на 2. Следовательно, если остаток от деления на 2 равен 0 (`number % 2 == 0`), число четное, а если нет — нечетное.

```
Enter a number, and I'll tell you if it's even or odd: 42

The number 42 is even.
```

УПРАЖНЕНИЯ

7.1. Прокат машин: напишите программу, которая спрашивает у пользователя, какую машину он бы хотел взять напрокат. Выведите сообщение с введенными данными (например, «Let me see if I can find you a Subaru").

7.2. Заказ стола: напишите программу, которая спрашивает у пользователя, на сколько мест он хочет забронировать стол в ресторане. Если введенное число больше 8, выведите сообщение о том, что пользователю придется подождать. В противном случае сообщите, что стол готов.

7.3. Числа, кратные 10: запросите у пользователя число и сообщите, кратно оно 10 или нет.

Циклы while

Цикл `for` получает коллекцию элементов и выполняет блок кода по одному разу для каждого элемента в коллекции. В отличие от него, цикл `while` продолжает выполняться, пока остается истинным некоторое условие.

Цикл while в действии

Цикл `while` может использоваться для перебора числовой последовательности. Например, следующий цикл считает от 1 до 5:

counting.py

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

В первой строке отсчет начинается с 1, для чего `current_number` присваивается значение 1. Далее запускается цикл `while`, который продолжает работать, пока значение `current_number` остается меньшим или равным 5. Код в цикле выводит значение `current_number` и увеличивает его на 1 командой `current_number += 1`. (Оператор `+=` является сокращенной формой записи для `current_number = current_number + 1`.)

Цикл повторяется, пока условие `current_number <= 5` остается истинным. Так как 1 меньше 5, Python выводит 1, а затем увеличивает значение на 1, отчего `current_number` становится равным 2. Так как 2 меньше 5, Python выводит 2 и снова прибавляет 1, и т. д. Как только значение `current_number` превысит 5, цикл останавливается, а программа завершается:

```
1
2
3
4
5
```

Очень многие повседневные программы содержат циклы `while`. Например, представьте компьютерную игру: цикл `while` выполняется, пока игра продолжается, и завершается, как только игрок захочет остановить игру. Вряд ли кого-нибудь обрадует, если программа завершает работу преждевременно или продолжает работать, когда ей приказали остановиться, так что циклы `while` весьма полезны.

Пользователь решает прервать работу программы

Программа parrot.py может выполняться, пока пользователь не захочет остановить ее, — для этого большая часть кода заключается в цикл `while`. В программе определяется *признак завершения*, и программа работает, пока пользователь не введет нужное значение:

parrot.py

```
❶ prompt = "\nTell me something, and I will repeat it back to you:"
  prompt += "\nEnter 'quit' to end the program. "
❷ message = ""
❸ while message != 'quit':
      message = input(prompt)
      print(message)
```

В точке ❶ определяется сообщение, которое объясняет, что у пользователя есть два варианта: ввести сообщение или ввести признак завершения (в данном случае это строка `'quit'`). Затем переменной `message` ❷ присваивается значение, введенное пользователем. В программе переменная `message` инициализируется пустой строкой `""`, чтобы значение проверялось без ошибок при первом выполнении строки `while`. Когда программа только запускается и выполнение достигает команды `while`, значение `message` необходимо сравнить с `'quit'`, но пользователь еще не вводил никакие данные. Если у Python нет данных для сравнения, продолжение выполнения становится невозможным. Чтобы решить эту проблему, необходимо предоставить `message` исходное значение. И хотя это всего лишь пустая строка, для Python такое значение выглядит вполне осмысленно; программа сможет выполнить сравнение, на котором основана работа цикла `while`. Цикл `while` ❸ выполняется, пока значение `message` не равно `'quit'`.

При первом выполнении цикла `message` содержит пустую строку, и Python входит в цикл. При выполнении команды `message = input(prompt)` Python отображает подсказку и ожидает, пока пользователь введет данные. Эти данные сохраняются в `message` и выводятся командой `print`; после этого Python снова проверяет условие команды `while`. Пока пользователь не введет слово `'quit'`, приглашение будет выводиться снова и снова, а Python будет ожидать новых данных. При вводе слова `'quit'` Python перестает выполнять цикл `while`, а программа завершается:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
quit
```

Программа работает неплохо, если не считать того, что она выводит слово `'quit'`, словно оно является обычным сообщением. Простая проверка `if` решает проблему:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Теперь программа проводит проверку перед выводом сообщения и выводит сообщение только в том случае, если оно не совпадает с признаком завершения:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

Флаги

В предыдущем примере программа выполняла некоторые операции, пока заданное условие оставалось истинным. А если вы пишете более сложную программу, выполнение которой может прерываться по нескольким разным условиям?

Например, компьютерная игра может завершаться по разным причинам: у игрока кончились все «жизни»; прошло отведенное время; все города, которые он должен был защищать, были уничтожены и т. д. Игра должна завершаться при выполнении любого из этих условий. Попытки проверять все возможные условия в одной команде `while` быстро усложняются и становятся слишком громоздкими.

Если программа должна выполняться только при истинности нескольких условий, определите одну переменную-*флаг*. Эта переменная сообщает, должна ли программа выполняться далее. Программу можно написать так, чтобы она продолжала выполнение, если флаг находится в состоянии `True`, и завершалась, если любое из нескольких событий перевело флаг в состояние `False`. В результате в команде `while` достаточно проверить всего одно условие: находится ли флаг в состоянии `True`. Все остальные проверки (которые должны определить, произошло ли событие, переводящее флаг в состояние `False`) удобно организуются в остальном коде.

Добавим флаг в программу parrot.py из предыдущего раздела. Этот флаг, который мы назовем `active` (хотя переменная может называться как угодно), управляет тем, должно ли продолжаться выполнение программы:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

❶ active = True
❷ while active:
      message = input(prompt)

❸     if message == 'quit':
          active = False
❹     else:
          print(message)
```

В точке ❶ переменной `active` присваивается `True`, чтобы программа начинала работу в активном состоянии. Это присваивание упрощает команду `while`, потому что в самой команде `while` никакие сравнения не выполняются; вся логика реализуется в других частях программы. Пока переменная `active` остается равной `True`, цикл выполняется ❷.

В команде `if` внутри цикла `while` значение `message` проверяется после того, как пользователь введет данные. Если пользователь ввел строку `'quit'` ❸, флаг `active` переходит в состояние `False`, а цикл `while` останавливается. Если пользователь ввел любой текст, кроме `'quit'` ❹, то введенные им данные выводятся как сообщение.

Результаты работы этой программы ничем не отличаются от предыдущего примера, в котором условная проверка выполняется прямо в команде `while`. Но теперь в программе имеется флаг, указывающий, находится ли она в активном состоянии, и вы сможете легко добавить новые проверки (в форме команд `elif`) для событий, с которыми переменная `active` может перейти в состояние `False`. Это может быть удобно в сложных программах — например, в компьютерных играх с многочисленными событиями, каждое из которых может привести к завершению программы. Когда по любому из этих событий флаг `active` переходит в состояние `False`,

основной игровой цикл прерывается, выводится сообщение о завершении игры и у игрока появляется возможность сыграть еще раз.

Команда break и выход из цикла

Чтобы немедленно прервать цикл `while` без выполнения оставшегося кода в цикле независимо от состояния условия, используйте команду `break`. Команда `break` управляет ходом выполнения программы; она позволит вам управлять тем, какая часть кода выполняется, а какая нет.

Рассмотрим пример — программу, которая спрашивает у пользователя, в каких городах он бывал. Чтобы прервать цикл `while`, программа выполняет команду `break`, как только пользователь введет значение `'quit'`:

cities.py

```
  prompt = "\nPlease enter the name of a city you have visited:"
  prompt += "\n(Enter 'quit' when you are finished.) "

❶ while True:
      city = input(prompt)

      if city == 'quit':
          break
      else:
          print(f"I'd love to go to {city.title()}!")
```

Цикл, который начинается с `while True` ❶, будет выполняться бесконечно — если только в нем не будет выполнена команда `break`. Цикл в программе продолжает запрашивать у пользователя названия городов, пока пользователь не введет строку `'quit'`. При вводе строки `'quit'` выполняется команда `break`, по которой Python выходит из цикла:

```
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) New York
I'd love to go to New York!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) San Francisco
I'd love to go to San Francisco!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) quit
```

ПРИМЕЧАНИЕ Команда `break` может использоваться в любых циклах Python. Например, ее можно включить в цикл `for` для перебора элементов словаря.

Команда continue и продолжение цикла

Вместо того чтобы полностью прерывать цикл без выполнения оставшейся части кода, вы можете воспользоваться командой `continue` для возвращения к началу

цикла и проверке условия. Например, возьмем цикл, который считает от 1 до 10, но выводит только нечетные числа в этом диапазоне:

counting.py

```
  current_number = 0
  while current_number < 10:
❶     current_number += 1
      if current_number % 2 == 0:
          continue

      print(current_number)
```

Сначала переменной `current_number` присваивается 0. Так как значение меньше 10, Python входит в цикл `while`. При входе в цикл счетчик увеличивается на 1 в точке ❶, поэтому `current_number` принимает значение 1. Затем команда `if` проверяет остаток от деления `current_number` на 2. Если остаток равен 0 (это означает, что `current_number` делится на 2), команда `continue` приказывает Python проигнорировать оставшийся код цикла и вернуться к началу. Если счетчик не делится на 2, то оставшаяся часть цикла выполняется и Python выводит текущее значение счетчика:

```
1
3
5
7
9
```

Предотвращение зацикливания

У каждого цикла `while` должна быть предусмотрена возможность завершения, чтобы цикл не выполнялся бесконечно. Например, следующий цикл считает от 1 до 5:

counting.py

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

Но если случайно пропустить строку `x += 1` (см. далее), то цикл будет выполняться бесконечно:

```
# Бесконечный цикл!
x = 1
while x <= 5:
    print(x)
```

Теперь переменной `x` присваивается начальное значение 1, но это значение никогда не изменяется в программе. В результате проверка условия `x <= 5` всегда дает результат `True`, и цикл `while` выводит бесконечную серию единиц:

```
1
1
1
1
...
```

Любой программист время от времени пишет бесконечный цикл, особенно если в программе используются неочевидные условия завершения. Если ваша программа зациклилась, нажмите Ctrl+C или просто закройте терминальное окно с выводом программы.

Чтобы избежать зацикливания, тщательно проверьте каждый цикл `while` и убедитесь в том, что цикл прерывается именно тогда, когда предполагается. Если программа должна завершаться при вводе некоторого значения, запустите программу и введите это значение. Если программа не завершилась, проанализируйте обработку значения, которое должно приводить к выходу из цикла. Проверьте, что хотя бы одна часть программы может привести к тому, что условие цикла станет равно `False` или будет выполнена команда `break`.

ПРИМЕЧАНИЕ В некоторых редакторах — в частности, в Sublime Text — используется встроенное окно вывода. Оно может усложнить прерывание бесконечных циклов; возможно, для выхода из цикла придется закрыть редактор. Прежде чем нажимать Ctrl+C, попробуйте щелкнуть в области вывода; возможно, вам удастся прервать бесконечный цикл.

УПРАЖНЕНИЯ

7.4. Топпинг для пиццы: напишите цикл, который предлагает пользователю вводить дополнения для пиццы до тех пор, пока не будет введено значение `'quit'`. При вводе каждого дополнения выведите сообщение о том, что это дополнение включено в заказ.

7.5. Билеты в кино: кинотеатр установил несколько вариантов цены на билеты в зависимости от возраста посетителя. Для посетителей младше 3 лет билет бесплатный; в возрасте от 3 до 12 билет стоит $10; наконец, если возраст посетителя больше 12, билет стоит $15. Напишите цикл, который предлагает пользователю ввести возраст и выводит цену билета.

7.6. Три выхода: напишите альтернативную версию упражнения 7.4 или упражнения 7.5, в которой каждый пункт следующего списка встречается хотя бы один раз:

- Завершение цикла по проверке условия в команде `while`.
- Управление продолжительностью выполнения цикла в зависимости от переменной `active`.
- Выход из цикла по команде break, если пользователь вводит значение `'quit'`.

7.7. Бесконечный цикл: напишите цикл, который никогда не завершается, и выполните его. (Чтобы выйти из цикла, нажмите Ctrl+C или закройте окно с выводом.)

Использование цикла while со списками и словарями

До настоящего момента мы работали только с одним фрагментом информации, полученной от пользователя. Мы получали ввод пользователя, а затем выво-

дили ответ на него. При следующем проходе цикла `while` программа получала новое входное значение и реагировала на него. Но чтобы работать с несколькими фрагментами информации, необходимо использовать в циклах `while` списки и словари.

Цикл `for` хорошо подходит для перебора списков, но скорее всего, список не должен изменяться в цикле, потому что у Python возникнут проблемы с отслеживанием элементов. Чтобы изменять список в процессе обработки, используйте цикл `while`. Использование циклов `while` со списками и словарями позволяет собирать, хранить и упорядочивать большие объемы данных для последующего анализа и обработки.

Перемещение элементов между списками

Возьмем список недавно зарегистрированных, но еще не проверенных пользователей сайта. Как переместить пользователей после проверки в отдельный список проверенных пользователей? Одно из возможных решений: используем цикл `while` для извлечения пользователей из списка непроверенных, проверяем их и включаем в отдельный список проверенных пользователей. Код может выглядеть так:

confirmed_users.py

```
  # Начинаем с двух списков: пользователей для проверки
  #  и пустого списка для хранения проверенных пользователей.
❶ unconfirmed_users = ['alice', 'brian', 'candace']
  confirmed_users = []

  # Проверяем каждого пользователя, пока остаются непроверенные
  #  пользователи. Каждый пользователь, прошедший проверку,
  #  перемещается в список проверенных.
❷ while unconfirmed_users:
❸     current_user = unconfirmed_users.pop()

      print(f"Verifying user: {current_user.title()}")
❹     confirmed_users.append(current_user)

  # Вывод всех проверенных пользователей.
  print("\nThe following users have been confirmed:")
  for confirmed_user in confirmed_users:
      print(confirmed_user.title())
```

Работа программы начинается с двух списков: непроверенных пользователей ❶ и пустого списка для проверенных пользователей. Цикл `while` в точке ❷ выполняется, пока в списке `unconfirmed_users` остаются элементы. Внутри этого списка функция `pop()` в точке ❸ извлекает очередного непроверенного пользователя из конца списка `unconfirmed_users`. В данном примере список `unconfirmed_users` завершается пользователем `Candace`; это имя первым извлекается из списка, сохраняется в `current_user` и добавляется в список `confirmed_users` в точке ❹. Далее следуют пользователи `Brian` и `Alice`.

Программа моделирует проверку каждого пользователя выводом сообщения, после чего переносит пользователя в список проверенных. По мере сокращения списка непроверенных пользователей список проверенных пользователей растет. Когда в списке непроверенных пользователей не остается ни одного элемента, цикл останавливается и выводится список проверенных пользователей:

```
Verifying user: Candace
Verifying user: Brian
Verifying user: Alice

The following users have been confirmed:
Candace
Brian
Alice
```

Удаление всех вхождений конкретного значения из списка

В главе 3 функция `remove()` использовалась для удаления конкретного значения из списка. Функция `remove()` работала, потому что интересующее нас значение встречалось в списке только один раз. Но что, если вы захотите удалить все вхождения значения из списка?

Допустим, имеется список `pets`, в котором значение `'cat'` встречается многократно. Чтобы удалить все экземпляры этого значения, можно выполнять цикл `while` до тех пор, пока в списке не останется ни одного экземпляра `'cat'`:

pets.py

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

Программа начинает со списка, содержащего множественные экземпляры `'cat'`. После вывода списка Python входит в цикл `while`, потому что значение `'cat'` присутствует в списке хотя бы в одном экземпляре. После входа цикл Python удаляет первое вхождение `'cat'`, возвращается к строке `while`, а затем обнаруживает, что экземпляры `'cat'` все еще присутствуют в списке, и проходит цикл заново. Вхождения `'cat'` удаляются до тех пор, пока не окажется, что в списке значений `'cat'` не осталось; в этот момент Python завершает цикл и выводит список заново:

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']
```

Заполнение словаря данными, введенными пользователем

При каждом проходе цикла `while` ваша программа может запрашивать любое необходимое количество данных. Напишем программу, которая при каждом про-

ходе цикла запрашивает имя участника и его ответ. Собранные данные будут сохраняться в словаре, потому что каждый ответ должен быть связан с конкретным пользователем:

mountain_poll.py

```
responses = {}

# Установка флага продолжения опроса.
polling_active = True

while polling_active:
    # Запрос имени и ответа пользователя.
❶   name = input("\nWhat is your name? ")
    response = input("Which mountain would you like to climb someday? ")

    # Ответ сохраняется в словаре:
❷   responses[name] = response

    # Проверка продолжения опроса.
❸   repeat = input("Would you like to let another person respond? (yes/ no) ")
    if repeat == 'no':
        polling_active = False

# Опрос завершен, вывести результаты.
print("\n--- Poll Results ---")
❹ for name, response in responses.items():
    print(f"{name} would like to climb {response}.")
```

Сначала программа определяет пустой словарь (`responses`) и устанавливает флаг (`polling_active`), показывающий, что опрос продолжается. Пока `polling_active` содержит `True`, Python будет выполнять код в цикле `while`.

В цикле пользователю предлагается ввести имя и название горы, на которую ему хотелось бы подняться ❶. Эта информация сохраняется в словаре `responses` в строке ❷, после чего программа спрашивает у пользователя, нужно ли продолжать опрос ❸. Если пользователь отвечает положительно, то программа снова входит в цикл `while`. Если же ответ отрицателен, флаг `polling_active` переходит в состояние `False`, цикл `while` перестает выполняться и завершающий блок кода ❹ выводит результаты опроса.

Если вы запустите эту программу и введете пару ответов, результат будет выглядеть примерно так:

```
What is your name? Eric
Which mountain would you like to climb someday? Denali
Would you like to let another person respond? (yes/ no) yes

What is your name? Lynn
Which mountain would you like to climb someday? Devil's Thumb
Would you like to let another person respond? (yes/ no) no

--- Poll Results ---
```

```
Lynn would like to climb Devil's Thumb.
Eric would like to climb Denali.
```

УПРАЖНЕНИЯ

7.8. Сэндвичи: создайте список с именем `sandwich_orders`, заполните его названиями различных видов сэндвичей. Создайте пустой список с именем `finished_sandwiches`. В цикле переберите элементы первого списка и выведите сообщение для каждого элемента (например, «I made your tuna sandwich»). После этого каждый сэндвич из первого списка перемещается в список `finished_sandwiches`. После того как все элементы первого списка будут обработаны, выведите сообщение с перечислением всех изготовленных сэндвичей.

7.9. Без пастрами: используя список `sandwich_orders` из упражнения 7.8, проследите за тем, чтобы значение `'pastrami'` встречалось в списке как минимум три раза. Добавьте в начало программы код для вывода сообщения о том, что пастрами больше нет, и напишите цикл `while` для удаления всех вхождений `'pastrami'` из `sandwich_orders`. Убедитесь в том, что в `finished_sandwiches` значение `'pastrami'` не встречается ни одного раза.

7.10. Отпуск мечты: напишите программу, которая опрашивает пользователей, где бы они хотели провести отпуск. Включите блок кода для вывода результатов опроса.

Итоги

В этой главе вы научились использовать `input()` для того, чтобы пользователи могли вводить собственную информацию в своих программах. Вы научились работать с числовыми и текстовыми данными, а также управлять продолжительностью выполнения своих программ с помощью циклов `while`. Также мы рассмотрели несколько способов управления циклами `while`: установка флага, команда `break` и команда `continue`. Вы узнали, как использовать цикл `while` для перемещения элементов из одного списка в другой и как удалить все вхождения некоторого значения из списка. Также были рассмотрены возможности применения циклов `while` со словарями.

Глава 8 посвящена *функциям*. Функции позволяют разделить программу на меньшие части, каждая из которых решает одну конкретную задачу. Функции можно хранить в отдельных файлах и вызывать их столько раз, сколько потребуется. Благодаря функциям вы сможете писать более эффективный, более простой в отладке и сопровождении код, который к тому же можно повторно использовать в разных программах.

8 Функции

Эта глава посвящена *функциям* — именованным блокам кода, предназначенным для решения одной конкретной задачи. Чтобы выполнить задачу, определенную в виде функции, вы *вызываете* функцию, отвечающую за эту задачу. Если задача должна многократно выполняться в программе, вам не придется заново вводить весь необходимый код; просто вызовите функцию, предназначенную для решения задачи, и этот вызов прикажет Python выполнить код, содержащийся внутри функции. Как вы вскоре убедитесь, использование функций упрощает чтение, написание, тестирование кода и исправление ошибок.

В этой главе также рассматриваются возможности передачи информации функциям. Вы узнаете, как писать функции, основной задачей которых является вывод информации, и другие функции, предназначенные для обработки данных и возвращения значения (или набора значений). Наконец, вы научитесь хранить функции в отдельных файлах, называемых *модулями*, для упорядочения файлов основной программы.

Определение функции

Вот простая функция с именем `greet_user ()`, которая выводит приветствие:

greeter.py

```
❶ def greet_user():
❷       """Выводит простое приветствие."""
❸       print("Hello!")

❹ greet_user()
```

В этом примере представлена простейшая структура функции. Строка ❶ при помощи ключевого слова `def` сообщает Python, что вы определяете функцию. В *определении функции* указывается имя функции и если нужно — описание информации, необходимой функции для решения ее задачи. Эта информация заключается в круглые скобки. В данном примере функции присвоено имя `greet_user()` и она не нуждается в дополнительной информации для решения своей задачи, поэтому

круглые скобки пусты. (Впрочем, даже в этом случае они обязательны.) Наконец, определение завершается двоеточием.

Все строки с отступами, следующие за `def greet_user():`, образуют *тело* функции. Текст в точке ❷ представляет собой комментарий — *строку документации* с описанием функции. Строки документации заключаются в утроенные кавычки; Python опознает их по этой последовательности символов во время генерирования документации к функциям в ваших программах.

«Настоящий» код в теле этой функции состоит всего из одной строки `print("Hello!")` — см. ❸. Таким образом, функция `greet_user()` решает всего одну задачу: выполнение команды `print("Hello!")`.

Когда потребуется использовать эту функцию, вызовите ее. *Вызов функции* приказывает Python выполнить содержащийся в ней код. Чтобы вызвать функцию, укажите ее имя, за которым следует вся необходимая информация, заключенная в круглые скобки, как показано в строке ❹. Так как никакая дополнительная информация не нужна, вызов функции эквивалентен простому выполнению команды `greet_user()`. Как и ожидалось, функция выводит сообщение `Hello!`:

```
Hello!
```

Передача информации функции

С небольшими изменениями функция `greet_user()` сможет не только сказать «Привет!» пользователю, но и поприветствовать его по имени. Для этого следует включить имя `username` в круглых скобках в определении функции `def greet_user()`. С добавлением `username` функция примет любое значение, которое будет заключено в скобки при вызове. Теперь функция ожидает, что при каждом вызове будет передаваться имя пользователя. При вызове `greet_user()` укажите имя (например, `'jesse'`) в круглых скобках:

```
def greet_user(username):
    """Выводит простое приветствие."""
    print(f"Hello, {username.title()}!")

greet_user('jesse')
```

Команда `greet_user('jesse')` вызывает функцию `greet_user()` и передает ей информацию, необходимую для выполнения команды `print`. Функция получает переданное имя и выводит приветствие для этого имени:

```
Hello, Jesse!
```

Точно так же команда `greet_user('sarah')` вызывает функцию `greet_user()` и передает ей строку `'sarah'`, в результате чего будет выведено сообщение `Hello, Sarah!` Функцию `greet_user()` можно вызвать сколько угодно раз и передать ей любое имя на ваше усмотрение — и вы будете получать ожидаемый результат.

Аргументы и параметры

Функция `greet_user()` определена так, что для работы она должна получить значение переменной `username`. После того как функция будет вызвана и получит необходимую информацию (имя пользователя), она выведет правильное приветствие.

Переменная `username` в определении `greet_user()` — *параметр*, то есть условные данные, необходимые функции для выполнения ее работы. Значение `'jesse'` в `greet_user('jesse')` — *аргумент*, то есть конкретная информация, переданная при вызове функции. Вызывая функцию, вы заключаете значение, с которым функция должна работать, в круглые скобки. В данном случае аргумент `'jesse'` был передан функции `greet_user()`, а его значение было сохранено в переменной `username`.

ПРИМЕЧАНИЕ Иногда в литературе термины «аргумент» и «параметр» используются как синонимы. Не удивляйтесь, если переменные в определении функции вдруг будут названы аргументами, а значения, переданные при вызове функции, — параметрами.

УПРАЖНЕНИЯ

8.1. Сообщение: напишите функцию `display_message()` для вывода сообщения по теме, рассматриваемой в этой главе. Вызовите функцию и убедитесь в том, что сообщение выводится правильно.

8.2. Любимая книга: напишите функцию `favorite_book()`, которая получает один параметр `title`. Функция должна выводить сообщение вида «One of my favorite books is Alice in Wonderland». Вызовите функцию и убедитесь в том, что название книги правильно передается как аргумент при вызове функции.

Передача аргументов

Определение функции может иметь несколько параметров, и может оказаться, что при вызове функции должны передаваться несколько аргументов. Существуют несколько способов передачи аргументов функциям. *Позиционные аргументы* перечисляются в порядке, точно соответствующем порядку записи параметров; *именованные аргументы* состоят из имени переменной и значения; наконец, существуют списки и словари значений. Рассмотрим все эти способы.

Позиционные аргументы

При вызове функции каждому аргументу должен быть поставлен в соответствие параметр в определении функции. Проще всего сделать это на основании порядка перечисления аргументов. Значения, связываемые с аргументами подобным образом, называются *позиционными аргументами*.

Чтобы понять, как работает эта схема, рассмотрим функцию для вывода информации о домашних животных. Функция сообщает тип животного и его имя:

pets.py

```
❶ def describe_pet(animal_type, pet_name):
      """Выводит информацию о животном."""
      print(f"\nI have a {animal_type}.")
      print(f"My {animal_type}'s name is {pet_name.title()}.")

❷ describe_pet('hamster', 'harry')
```

Из определения ❶ видно, что функции должен передаваться тип животного (`animal_type`) и его имя (`pet_name`). При вызове `describe_pet()` необходимо передать тип и имя — именно в таком порядке. В этом примере аргумент `'hamster'` сохраняется в параметре `animal_type`, а аргумент `'harry'` сохраняется в параметре `pet_name` ❷. В теле функции эти два параметра используются для вывода информации:

```
I have a hamster.
My hamster's name is Harry.
```

Многократные вызовы функций

Функция может вызываться в программе столько раз, сколько потребуется. Для вывода информации о другом животном достаточно одного вызова `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Выводит информацию о животном."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

Во втором вызове функции `describe_pet()` передаются аргументы `'dog'` и `'willie'`. По аналогии с предыдущей парой аргументов Python сопоставляет аргумент `'dog'` с параметром `animal_type`, а аргумент `'willie'` — с параметром `pet_name`.

Как и в предыдущем случае, функция выполняет свою задачу, но на этот раз выводятся другие значения:

```
I have a hamster.
My hamster's name is Harry.

I have a dog.
My dog's name is Willie.
```

Многократный вызов функции — чрезвычайно эффективный механизм. Код вывода информации о домашнем животном пишется один раз в функции. Каждый раз, когда вам понадобится вывести информацию о новом животном, вы вызываете функцию с данными нового животного. Даже если код вывода информации разрастется до 10 строк, вы все равно сможете вывести информацию всего одной командой — для этого достаточно снова вызвать функцию.

Функция может иметь любое количество позиционных аргументов. При вызове функции Python перебирает аргументы, приведенные в вызове, и сопоставляет каждый аргумент с соответствующим параметром из определения функции.

О важности порядка позиционных аргументов

Если нарушить порядок следования аргументов в вызове при использовании позиционных аргументов, возможны неожиданные результаты:

```
def describe_pet(animal_type, pet_name):
    """Выводит информацию о животном."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('harry', 'hamster')
```

В этом вызове функции сначала передается имя, а потом тип животного. Так как аргумент `'harry'` находится в первой позиции, значение сохраняется в параметре `animal_type`, а аргумент `'hamster'` — в `pet_name`. На этот раз вывод получается бессмысленным:

```
I have a harry.
My harry's name is Hamster.
```

Если вы получили подобные странные результаты, проверьте, что порядок следования аргументов в вызове функции соответствует порядку параметров в ее определении.

Именованные аргументы

Именованный аргумент представляет собой пару «имя-значение», передаваемую функции. Имя и значение связываются с аргументом напрямую, так что при передаче аргумента путаница с порядком исключается. Именованные аргументы избавляют от хлопот с порядком аргументов при вызове функции, а также проясняют роль каждого значения в вызове функции.

Перепишем программу pets.py с использованием именованных аргументов при вызове `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Выводит информацию о животном."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(animal_type='hamster', pet_name='harry')
```

Функция `describe_pet()` не изменилась. Однако на этот раз при вызове функции мы явно сообщаем Python, с каким параметром должен быть связан каждый аргу-

мент. При обработке вызова функции Python знает, что аргумент `'hamster'` должен быть сохранен в параметре `animal_type`, а аргумент `'harry'` — в параметре `pet_name`.

Порядок следования именованных аргументов в данном случае неважен, потому что Python знает, где должно храниться каждое значение. Следующие два вызова функции эквивалентны:

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

ПРИМЕЧАНИЕ При использовании именованных аргументов будьте внимательны — имена должны точно совпадать с именами параметров из определения функции.

Значения по умолчанию

Для каждого параметра вашей функции можно определить значение по умолчанию. Если при вызове функции передается аргумент, соответствующий данному параметру, Python использует значение аргумента, а если нет — использует значение по умолчанию. Таким образом, если для параметра определено значение по умолчанию, вы можете опустить соответствующий аргумент, который обычно включается в вызов функции. Значения по умолчанию упрощают вызовы функций и проясняют типичные способы использования функций.

Например, если вы заметили, что большинство вызовов `describe_pet()` используется для описания собак, задайте `animal_type` значение по умолчанию `'dog'`. Теперь в любом вызове `describe_pet()` для собаки эту информацию можно опустить:

```
def describe_pet(pet_name, animal_type='dog'):
    """Выводит информацию о животном."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(pet_name='willie')
```

Мы изменили определение `describe_pet()` и включили для параметра `animal_type` значение по умолчанию `'dog'`. Если теперь функция будет вызвана без указания `animal_type`, Python знает, что для этого параметра следует использовать значение `'dog'`:

```
I have a dog.
My dog's name is Willie.
```

Обратите внимание: в определении функции пришлось изменить порядок параметров. Так как благодаря значению по умолчанию указывать аргумент с типом животного не обязательно, единственным оставшимся аргументом в вызове функции остается имя домашнего животного. Python интерпретирует его как позиционный аргумент, и если функция вызывается только с именем животного, этот аргумент ставится в соответствие с первым параметром в определении функции. Именно по этой причине имя животного должно быть первым параметром.

В простейшем варианте использования этой функции при вызове передается только имя собаки:

```
describe_pet('willie')
```

Вызов функции выводит тот же результат, что и в предыдущем примере. Единственный переданный аргумент `'willie'` ставится в соответствие с первым параметром в определении, `pet_name`. Так как для `animal_type` аргумент не указан, Python использует значение по умолчанию `'dog'`.

Для вывода информации о любом другом животном, кроме собаки, используется вызов функции следующего вида:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Так как аргумент для параметра `animal_type` задан явно, Python игнорирует значение параметра по умолчанию.

ПРИМЕЧАНИЕ Если вы используете значения по умолчанию, все параметры со значением по умолчанию должны следовать после параметров, у которых значений по умолчанию нет. Это необходимо для того, чтобы Python правильно интерпретировал позиционные аргументы.

Эквивалентные вызовы функций

Так как позиционные аргументы, именованные аргументы и значения по умолчанию могут использоваться одновременно, часто существуют несколько эквивалентных способов вызова функций. Возьмем следующий оператор `describe_pets()` с одним значением по умолчанию:

```
def describe_pet(pet_name, animal_type='dog'):
```

При таком определении аргумент для параметра `pet_name` должен задаваться в любом случае, но это значение может передаваться как в позиционном, так и в именованном формате. Если описываемое животное не является собакой, то аргумент `animal_type` тоже должен быть включен в вызов, и этот аргумент тоже может быть задан как в позиционном, так и в именованном формате.

Все следующие вызовы являются допустимыми для данной функции:

```
# Пес по имени Вилли.
describe_pet('willie')
describe_pet(pet_name='willie')

# Хомяк по имени Гарри.
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

Все вызовы функции выдадут такой же результат, как и в предыдущих примерах.

ПРИМЕЧАНИЕ На самом деле не так важно, какой стиль вызова вы используете. Если ваша функция выдает нужный результат, выберите тот стиль, который вам кажется более понятным.

Предотвращение ошибок в аргументах

Не удивляйтесь, если на первых порах вашей работы с функциями будут встречаться ошибки несоответствия аргументов. Такие ошибки происходят в том случае, если вы передали меньше или больше аргументов, чем необходимо функции для выполнения ее работы. Например, вот что произойдет при попытке вызвать `describe_pet()` без аргументов:

```
def describe_pet(animal_type, pet_name):
    """Выводит информацию о животном."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet()
```

Python понимает, что при вызове функции часть информации отсутствует, и мы видим это в данных трассировки:

```
  Traceback (most recent call last):
❶   File "pets.py", line 6, in <module>
❷     describe_pet()
❸ TypeError: describe_pet() missing 2 required positional arguments: 'animal_
  type' and 'pet_name'
```

В точке ❶ сообщается местонахождение проблемы, чтобы вы поняли, что с вызовом функции что-то пошло не так. В точке ❷ приводится вызов функции, приведший к ошибке. В точке ❸ Python сообщает, что при вызове пропущены два аргумента, и сообщает имена этих аргументов. Если бы функция размещалась в отдельном файле, вероятно, вы смогли бы исправить вызов и вам не пришлось бы открывать этот файл и читать код функции.

Python помогает еще и тем, что он читает код функции и сообщает имена аргументов, которые необходимо передать при вызове. Это еще одна причина для того, чтобы присваивать переменным и функциям содержательные имена. В этом случае сообщения об ошибках Python принесут больше пользы как вам, так и любому другому разработчику, который будет использовать ваш код.

Если при вызове будут переданы лишние аргументы, вы получите похожую трассировку, которая поможет привести вызов функции в соответствие с ее определением.

УПРАЖНЕНИЯ

8.3. Футболка: напишите функцию `make_shirt()`, которая получает размер футболки и текст, который должен быть напечатан на ней. Функция должна выводить сообщение с размером и текстом.

Вызовите функцию с использованием позиционных аргументов. Вызовите функцию во второй раз с использованием именованных аргументов.

8.4. Большие футболки: измените функцию `make_shirt()`, чтобы по умолчанию футболки имели размер L и на них выводился текст «I love Python». Создайте футболку с размером L и текстом по умолчанию, а также футболку любого размера с другим текстом.

8.5. Города: напишите функцию `describe_city()`, которая получает названия города и страны. Функция должна выводить простое сообщение (например, «Reykjavik is in Iceland»). Задайте параметру страны значение по умолчанию. Вызовите свою функцию для трех разных городов, по крайней мере один из которых не находится в стране по умолчанию.

Возвращаемое значение

Функция не обязана выводить результаты своей работы напрямую. Вместо этого она может обработать данные, а затем вернуть значение или набор сообщений. Значение, возвращаемое функцией, называется *возвращаемым значением*. Команда `return` передает значение из функции в точку программы, в которой эта функция была вызвана. Возвращаемые значения помогают переместить большую часть рутинной работы в вашей программе в функции, чтобы упростить основной код программы.

Возвращение простого значения

Рассмотрим функцию, которая получает имя и фамилию и возвращает аккуратно отформатированное полное имя:

formatted_name.py

```
❶ def get_formatted_name(first_name, last_name):
      """Возвращает аккуратно отформатированное полное имя."""
❷     full_name = f"{first_name} {last_name}"
❸     return full_name.title()

❹ musician = get_formatted_name('jimi', 'hendrix')
  print(musician)
```

Определение `get_formatted_name()` получает в параметрах имя и фамилию ❶. Функция объединяет эти два имени, добавляет между ними пробел и сохраняет результат в `full_name` ❷. Значение `full_name` преобразуется в формат с начальной буквой верхнего регистра, а затем возвращается в точку вызова ❸.

Вызывая функцию, которая возвращает значение, необходимо предоставить переменную, в которой должно храниться возвращаемое значение. В данном случае возвращаемое значение сохраняется в переменной `musician` ❹. Результат содержит аккуратно отформатированное полное имя, построенное из имени и фамилии:

```
Jimi Hendrix
```

Может показаться, что все эти хлопоты излишни — с таким же успехом можно было использовать команду:

```
print("Jimi Hendrix")
```

Но если представить, что вы пишете большую программу, в которой многочисленные имена и фамилии должны храниться по отдельности, такие функции, как `get_formatted_name()`, становятся чрезвычайно полезными. Вы храните имена отдельно от фамилий, а затем вызываете функцию везде, где потребуется вывести полное имя.

Необязательные аргументы

Иногда бывает удобно сделать аргумент необязательным, чтобы разработчик, использующий функцию, мог передать дополнительную информацию только в том случае, если он этого захочет. Чтобы сделать аргумент необязательным, можно воспользоваться значением по умолчанию.

Допустим, вы захотели расширить функцию `get_formatted_name()`, чтобы она также работала и со вторыми именами. Первая попытка могла бы выглядеть так:

```
def get_formatted_name(first_name, middle_name, last_name):
    """Возвращает аккуратно отформатированное полное имя."""
    full_name = f"{first_name} {middle_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

Функция работает при получении имени, второго имени и фамилии. Она получает все три части имени, а затем строит из них строку. Функция добавляет пробелы там, где это уместно, и преобразует полное имя в формат с капитализацией:

```
John Lee Hooker
```

Однако вторые имена нужны не всегда, а в такой записи функция не будет работать, если при вызове ей передается только имя и фамилия. Чтобы средний аргумент был необязательным, можно присвоить аргументу `middle_name` пустое значение по умолчанию; этот аргумент игнорируется, если пользователь не передал для него значение. Чтобы функция `get_formatted_name()` работала без второго имени, следует назначить для параметра `middle_name` пустую строку значением по умолчанию и переместить его в конец списка параметров:

```
❶ def get_formatted_name(first_name, last_name, middle_name=''):
      """Возвращает аккуратно отформатированное полное имя."""
❷     if middle_name:
          full_name = f"{first_name} {middle_name} {last_name}"
❸     else:
          full_name = f"{first_name} {last_name}"
      return full_name.title()
```

```
  musician = get_formatted_name('jimi', 'hendrix')
  print(musician)

❹ musician = get_formatted_name('john', 'hooker', 'lee')
  print(musician)
```

В этом примере имя строится из трех возможных частей. Поскольку имя и фамилия указываются всегда, эти параметры стоят в начале списка в определении функции. Второе имя не обязательно, поэтому оно находится на последнем месте в определении, а его значением по умолчанию является пустая строка ❶.

В теле функции мы сначала проверяем, было ли задано второе имя. Python интерпретирует непустые строки как истинное значение, и если при вызове задан аргумент второго имени, `middle_name` дает результат `True` ❷. Если второе имя указано, то из имени, второго имени и фамилии строится полное имя. Затем имя преобразуется с капитализацией символов и возвращается в строку вызова функции, где оно сохраняется в переменной `musician` и выводится. Если второе имя не указано, то пустая строка не проходит проверку `if` и выполняет блок `else` ❸. В этом случае полное имя строится только из имени и фамилии и отформатированное имя возвращается в строку вызова, где оно сохраняется в переменной `musician` и выводится.

Вызов этой функции с именем и фамилией достаточно тривиален. Но при использовании второго имени придется проследить за тем, чтобы второе имя было последним из передаваемых аргументов. Это необходимо для правильного связывания позиционных аргументов в строке ❹.

Обновленная версия этой функции подойдет как для людей, у которых задается только имя и фамилия, так и для людей со вторым именем:

```
Jimi Hendrix
John Lee Hooker
```

Необязательные значения позволяют функциям работать в максимально широком спектре сценариев использования без усложнения вызовов.

Возвращение словаря

Функция может вернуть любое значение, которое вам потребуется, в том числе и более сложную структуру данных (например, список или словарь). Так, следующая функция получает части имени и возвращает словарь, представляющий человека:

person.py

```
  def build_person(first_name, last_name):
      """Возвращает словарь с информацией о человеке."""
❶     person = {'first': first_name, 'last': last_name}
❷     return person

  musician = build_person('jimi', 'hendrix')
❸ print(musician)
```

Функция `build_person()` получает имя и фамилию и сохраняет полученные значения в словаре в точке ❶. Значение `first_name` сохраняется с ключом `'first'`, а значение `last_name` — с ключом `'last'`. Весь словарь с описанием человека возвращается в точке ❷. Возвращаемое значение выводится в точке ❸ с двумя исходными фрагментами текстовой информации, теперь хранящимися в словаре:

```
{'first': 'jimi', 'last': 'hendrix'}
```

Функция получает простую текстовую информацию и помещает ее в более удобную структуру данных, которая позволяет работать с информацией (помимо простого вывода). Строки `'jimi'` и `'hendrix'` теперь помечены как имя и фамилия. Функцию можно легко расширить так, чтобы она принимала дополнительные значения — второе имя, возраст, профессию или любую другую информацию о человеке, которую вы хотите сохранить. Например, следующее изменение позволяет также сохранить возраст человека:

```
def build_person(first_name, last_name):
    """Возвращает словарь с информацией о человеке."""
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

В определение функции добавляется новый необязательный параметр `age`, которому присваивается специальное значение по умолчанию `None` — оно используется для переменных, которым не присвоено никакое значение. При проверке условий `None` интерпретируется как `False`. Если вызов функции включает значение этого параметра, то значение сохраняется в словаре. Функция всегда сохраняет имя, но ее также можно модифицировать, чтобы она сохраняла любую необходимую информацию о человеке.

Использование функции в цикле while

Функции могут использоваться со всеми структурами Python, уже известными вам. Например, используем функцию `get_formatted_name()` в цикле `while`, чтобы поприветствовать пользователей более официально. Первая версия программы, приветствующей пользователей по имени и фамилии, может выглядеть так:

greeter.py

```
def get_formatted_name(first_name, last_name):
    """Возвращает аккуратно отформатированное полное имя."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# Бесконечный цикл!
```

```
  while True:
❶     print("\nPlease tell me your name:")
      f_name = input("First name: ")
      l_name = input("Last name: ")

      formatted_name = get_formatted_name(f_name, l_name)
      print(f"\nHello, {formatted_name}!")
```

В этом примере используется простая версия `get_formatted_name()`, не использующая вторые имена. В цикле `while` ❶ имя и фамилия пользователя запрашиваются по отдельности.

Но у этого цикла `while` есть один недостаток: в нем не определено условие завершения. Где следует разместить условие завершения при запросе серии данных? Пользователю нужно предоставить возможность выйти из цикла как можно раньше, так что в приглашении должен содержаться способ завершения. Команда `break` позволяет немедленно прервать цикл при запросе любого из компонентов:

```
def get_formatted_name(first_name, last_name):
    """Возвращает аккуратно отформатированное полное имя."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
        break

    l_name = input("Last name: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

В программу добавляется сообщение, которое объясняет пользователю, как завершить ввод данных, и при вводе признака завершения в любом из приглашений цикл прерывается. Теперь программа будет приветствовать пользователя до тех пор, пока вместо имени или фамилии не будет введен символ `'q'`:

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: eric
Last name: matthes

Hello, Eric Matthes!

Please tell me your name:
(enter 'q' at any time to quit)
First name: q
```

УПРАЖНЕНИЯ

8.6. Названия городов: напишите функцию `city_country()`, которая получает название города и страну. Функция должна возвращать строку в формате `"Santiago, Chile"`. Вызовите свою функцию по крайней мере для трех пар «город — страна» и выведите возвращенное значение.

8.7. Альбом: напишите функцию `make_album()`, которая строит словарь с описанием музыкального альбома. Функция должна получать имя исполнителя и название альбома и возвращать словарь, содержащий эти два вида информации. Используйте функцию для создания трех словарей, представляющих разные альбомы. Выведите все возвращаемые значения, чтобы показать, что информация правильно сохраняется во всех трех словарях.

Добавьте в `make_album()` дополнительный параметр для сохранения количества дорожек в альбоме, имеющий значение по умолчанию `None`. Если в строку вызова включено значение количества дорожек, добавьте это значение в словарь альбома. Создайте как минимум один новый вызов функции с передачей количества дорожек в альбоме.

8.8. Пользовательские альбомы: начните с программы из упражнения 8.7. Напишите цикл `while`, в котором пользователь вводит исполнителя и название альбома. Затем в цикле вызывается функция `make_album()` для введенных пользователей и выводится созданный словарь. Не забудьте предусмотреть признак завершения в цикле while.

Передача списка

Часто при вызове функции удобно передать список — имен, чисел или более сложных объектов (например, словарей). При передаче списка функция получает прямой доступ ко всему его содержимому. Мы воспользуемся функциями для того, чтобы сделать работу со списком более эффективной.

Допустим, вы хотите вывести приветствие для каждого пользователя из списка. В следующем примере список имен передается функции `greet_users()`, которая выводит приветствие для каждого пользователя по отдельности:

greet_users.py

```
def greet_users(names):
    """Вывод простого приветствия для каждого пользователя в списке."""
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)

❶ usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

В соответствии со своим определением функция `greet_users()` рассчитывает получить список имен, который сохраняется в параметре `names`. Функция перебирает полученный список и выводит приветствие для каждого пользователя. В точке ❶ мы определяем список пользователей `usernames`, который затем передается `greet_users()` в вызове функции:

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

Результат выглядит именно так, как ожидалось. Каждый пользователь получает персональное сообщение, и эту функцию можно вызвать для каждого нового набора пользователей.

Изменение списка в функции

Если вы передаете список функции, код функции сможет изменить список. Все изменения, внесенные в список в теле функции, закрепляются, что позволяет эффективно работать со списком даже при больших объемах данных.

Допустим, компания печатает на 3D-принтере модели, предоставленные пользователем. Проекты хранятся в списке, а после печати перемещаются в отдельный список. В следующем примере приведена реализация, не использующая функции:

printing_models.py

```
# Список моделей, которые необходимо напечатать.
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

# Цикл последовательно печатает каждую модель до конца списка.
#  После печати каждая модель перемещается в список completed_models.
while unprinted_designs:
    current_design = unprinted_designs.pop()
    print(f"Printing model: {current_design}")
    completed_models.append(current_design)

# Вывод всех готовых моделей.
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)
```

В начале программы создается список моделей и пустой список `completed_models`, в который каждая модель перемещается после печати. Пока в `unprinted_designs` остаются модели, цикл `while` имитирует печать каждой модели: модель удаляется из конца списка, сохраняется в `current_design`, а пользователь получает сообщение о том, что текущая модель была напечатана. Затем модель перемещается в список напечатанных. После завершения цикла выводится список напечатанных моделей:

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case

The following models have been printed:
dodecahedron
robot pendant
phone case
```

Мы можем изменить структуру этого кода: для этого следует написать две функции, каждая из которых решает одну конкретную задачу. Большая часть кода останется неизменной; просто программа становится более эффективной. Первая функция занимается печатью, а вторая выводит сводку напечатанных моделей:

```
❶ def print_models(unprinted_designs, completed_models):
      """
      Имитирует печать моделей, пока список не станет пустым.
      Каждая модель после печати перемещается в completed_models.
      """
      while unprinted_designs:
          current_design = unprinted_designs.pop()
          print(f"Printing model: {current_design}")
          completed_models.append(current_design)

❷ def show_completed_models(completed_models):
      """Выводит информацию обо всех напечатанных моделях."""
      print("\nThe following models have been printed:")
      for completed_model in completed_models:
          print(completed_model)

  unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
  completed_models = []

  print_models(unprinted_designs, completed_models)
  show_completed_models(completed_models)
```

В точке ❶ определяется функция `print_models()` с двумя параметрами: список моделей для печати и список готовых моделей. Функция имитирует печать каждой модели, последовательно извлекая модели из первого списка и перемещая их во второй список. В точке ❷ определяется функция `show_completed_models()` с одним параметром: списком напечатанных моделей. Функция `show_completed_models()` получает этот список и выводит имена всех напечатанных моделей.

Программа выводит тот же результат, что и версия без функций, но структура кода значительно улучшилась. Код, выполняющий большую часть работы, разнесен по двум разным функциям; это упрощает чтение основной части программы. Теперь любому разработчику будет намного проще просмотреть код программы и понять, что делает программа:

```
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Программа создает список моделей для печати и пустой список для готовых моделей. Затем, поскольку обе функции уже определены, остается вызвать их и передать правильные аргументы. Мы вызываем `print_models()` и передаем два необходимых списка; как и ожидалось, `print_models()` имитирует печать моделей. Затем вызы-

вается функция show_completed_models() и ей передается список готовых моделей, чтобы функция могла вывести информацию о напечатанных моделях. Благодаря содержательным именам функций другой разработчик сможет прочитать этот код и понять его даже без комментариев.

Вдобавок эта программа создает меньше проблем с расширением и сопровождением, чем версия без функций. Если позднее потребуется напечатать новую партию моделей, достаточно снова вызвать print_models(). Если окажется, что код печати необходимо модифицировать, изменения достаточно внести в одном месте, и они автоматически распространятся на все вызовы функции. Такой подход намного эффективнее независимой правки кода в нескольких местах программы.

Этот пример также демонстрирует принцип, в соответствии с которым каждая функция должна решать одну конкретную задачу. Первая функция печатает каждую модель, а вторая выводит информацию о готовых моделях. Такой подход предпочтительнее решения обеих задач в функции. Если вы пишете функцию и видите, что она решает слишком много разных задач, попробуйте разделить ее код на две функции. Помните, что функции всегда можно вызывать из других функций. Эта возможность может пригодиться для разбиения сложных задач на серию составляющих.

Запрет изменения списка в функции

Иногда требуется предотвратить изменение списка в функции. Допустим, у вас имеется список моделей для печати и вы пишете функцию для перемещения их в список готовых моделей, как в предыдущем примере. Возможно, даже после печати всех моделей исходный список нужно оставить для отчетности. Но поскольку все имена моделей были перенесены из списка unprinted_designs, остался только пустой список; исходная версия списка потеряна. Проблему можно решить передачей функции копии списка вместо оригинала. В этом случае все изменения, вносимые функцией в список, будут распространяться только на копию, а оригинал списка остается неизменным.

Чтобы передать функции копию списка, можно поступить так:

```
имя_функции(имя_списка[:])
```

Синтаксис сегмента [:] создает копию списка для передачи функции. Если удаление элементов из списка unprinted_designs в print_models.py нежелательно, функцию print_models() можно вызвать так:

```
print_models(unprinted_designs[:], completed_models)
```

Функция print_models() может выполнить свою работу, потому что она все равно получает имена всех ненапечатанных моделей. Но на этот раз она использует не сам список unprinted_designs, а его копию. Список completed_models заполняется именами напечатанных моделей, как и в предыдущем случае, но исходный список функцией не изменяется.

Несмотря на то что передача копии позволяет сохранить содержимое списка, обычно функциям следует передавать исходный список (если у вас нет веских причин для передачи копии). Работа с существующим списком более эффективна, потому что программе не приходится тратить время и память на создание отдельной копии (лишние затраты особенно заметны при работе с большими списками).

УПРАЖНЕНИЯ

8.9. Сообщения: создайте список с серией коротких сообщений. Передайте список функции `show_messages()`, которая выводит текст каждого сообщения в списке.

8.10. Отправка сообщений: начните с копии вашей программы из упражнения 8.9. Напишите функцию `send_messages()`, которая выводит каждое сообщение и перемещает его в новый список с именем `sent_messages`. После вызова функции выведите оба списка и убедитесь в том, что перемещение прошло успешно.

8.11. Архивированные сообщения: начните с программы из упражнения 8.10. Вызовите функцию `send_messages()` для копии списка сообщений. После вызова функции выведите оба списка и убедитесь в том, что в исходном списке остались все сообщения.

Передача произвольного набора аргументов

В некоторых ситуациях вы не знаете заранее, сколько аргументов должно быть передано функции. К счастью, Python позволяет функции получить произвольное количество аргументов из вызывающей команды.

Для примера рассмотрим функцию для создания пиццы. Функция должна получить набор топпингов к пицце, но вы не знаете заранее, сколько топпингов закажет клиент. Функция в следующем примере получает один параметр `*toppings`, но этот параметр объединяет все аргументы, заданные в командной строке:

pizza.py

```
def make_pizza(*toppings):
    """Вывод списка заказанных топпингов."""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

Звездочка в имени параметра `*toppings` приказывает Python создать пустой кортеж с именем `toppings` и упаковать в него все полученные значения. Результат команды `print` в теле функции показывает, что Python успешно справляется и с вызовом функции с одним значением, и с вызовом с тремя значениями. Разные вызовы обрабатываются похожим образом. Обратите внимание: Python упаковывает аргументы в кортеж даже в том случае, если функция получает всего одно значение:

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Теперь команду `print` можно заменить циклом, который перебирает список топпингов и выводит описание заказанной пиццы:

```
def make_pizza(*toppings):
    """Выводит описание пиццы."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

Функция реагирует соответственно независимо от того, сколько значений она получила — одно или три:

```
Making a pizza with the following toppings:
- pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Этот синтаксис работает независимо от количества аргументов, переданных функции.

Позиционные аргументы с произвольными наборами аргументов

Если вы хотите, чтобы функция могла вызываться с разными количествами аргументов, параметр для получения произвольного количества аргументов должен стоять на последнем месте в определении функции. Python сначала подбирает соответствия для позиционных и именованных аргументов, а потом объединяет все остальные аргументы в последнем параметре.

Например, если функция должна получать размер пиццы, этот параметр должен стоять в списке до параметра `*toppings`:

```
def make_pizza(size, *toppings):
    """Выводит описание пиццы."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

В определении функции Python сохраняет первое полученное значение в параметре `size`. Все остальные значения, следующие за ним, сохраняются в кортеже `toppings`. В вызовах функций на первом месте располагается аргумент для параметра `size`, а за ним следуют сколько угодно дополнений.

В итоге для каждой пиццы указывается размер и количество дополнений, и каждый фрагмент информации выводится в положенном месте: сначала размер, а потом топпинги:

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

ПРИМЕЧАНИЕ В программах часто используется имя обобщенного параметра `*args` для хранения произвольного набора позиционных аргументов.

Использование произвольного набора именованных аргументов

Иногда программа должна получать произвольное количество аргументов, но вы не знаете заранее, какая информация будет передаваться функции. В таких случаях можно написать функцию, получающую столько пар «ключ-значение», сколько указано в команде вызова. Один из возможных примеров — построение пользовательских профилей: вы знаете, что вы получите информацию о пользователе, но не знаете заранее, какую именно. Функция `build_profile()` в следующем примере всегда получает имя и фамилию, но также может получать произвольное количество именованных аргументов:

user_profile.py

```
  def build_profile(first, last, **user_info):
      """Строит словарь с информацией о пользователе."""
❶     user_info['first_name'] = first
      user_info['last_name'] = last
      return user_info

  user_profile = build_profile('albert', 'einstein',
                               location='princeton',
                               field='physics')
  print(user_profile)
```

Определение `build_profile()` ожидает получить имя и фамилию пользователя, а также позволяет передать любое количество пар «имя-значение». Две звездочки перед параметром `**user_info` заставляют Python создать пустой словарь с именем `user_info` и упаковать в него все полученные пары «имя-значение». Внутри функции вы можете обращаться к парам «имя-значение» из `user_info` точно так же, как в любом словаре.

В теле `build_profile()` в словарь `user_info` добавляются имя и фамилия, потому что эти два значения всегда передаются пользователем ❶ и они еще не были помещены в словарь. Затем словарь `user_info` возвращается в точку вызова функции.

Вызовем функцию `build_profile()` и передадим ей имя `'albert'`, фамилию `'einstein'` и еще две пары «ключ-значение» — `location='princeton'` и `field='physics'`. Программа сохраняет возвращенный словарь в `user_profile` и выводит его содержимое:

```
{'location': 'princeton', 'field': 'physics',
'first_name': 'albert', 'last_name': 'einstein'}
```

Возвращаемый словарь содержит имя и фамилию пользователя, а в данном случае еще и местонахождение и область исследований. Функция будет работать, сколько бы дополнительных пар «ключ-значение» ни было передано при вызове функции.

При написании функций допускаются самые разнообразные комбинации позиционных, именованных и произвольных значений. Полезно знать о существовании всех этих типов аргументов, потому что они часто будут встречаться вам при чтении чужого кода. Только с практическим опытом вы научитесь правильно использовать разные типы аргументов и поймете, когда следует применять каждый тип; а пока просто используйте самый простой способ, который позволит решить задачу. С течением времени вы научитесь выбирать наиболее эффективный вариант для каждой конкретной ситуации.

ПРИМЕЧАНИЕ В программах часто используется имя обобщенного параметра `**kwargs` для хранения произвольного набора ключевых аргументов.

УПРАЖНЕНИЯ

8.12. Сэндвичи: напишите функцию, которая получает список компонентов сэндвича. Функция должна иметь один параметр для любого количества значений, переданных при вызове функции, и выводить описание заказанного сэндвича. Вызовите функцию три раза с разным количеством аргументов.

8.13. Профиль: начните с копии программы user_profile.py (с. 162). Создайте собственный профиль вызовом `build_profile()`, укажите имя, фамилию и три другие пары «ключ-значение» для вашего описания.

8.14. Автомобили: напишите функцию для сохранения информации об автомобиле в словаре. Функция всегда должна возвращать производителя и название модели, но при этом она может получать произвольное количество именованных аргументов. Вызовите функцию с передачей обязательной информации и еще двух пар «имя-значение» (например, цвет и комплектация). Ваша функция должна работать для вызовов следующего вида:

```
car = make_car('subaru', 'outback', color='blue', tow_package=True)
```

Выведите возвращаемый словарь и убедитесь в том, что вся информация была сохранена правильно.

Хранение функций в модулях

Одно из преимуществ функций заключается в том, что они отделяют блоки кода от основной программы. Если для функций были выбраны содержательные имена,

ваша программа будет намного проще читаться. Можно пойти еще дальше и сохранить функции в отдельном файле, называемом *модулем*, а затем *импортировать* модуль в свою программу. Команда `import` сообщает Python, что код модуля должен быть доступен в текущем выполняемом программном файле.

Хранение функций в отдельных файлах позволяет скрыть второстепенные детали кода и сосредоточиться на логике более высокого уровня. Кроме того, функции можно использовать во множестве разных программ. Функции, хранящиеся в отдельных файлах, можно передать другим программистам без распространения полного кода программы. А умение импортировать функции позволит вам использовать библиотеки функций, написанные другими программистами.

Существует несколько способов импортирования модулей; все они кратко рассматриваются ниже.

Импортирование всего модуля

Чтобы заняться импортированием функций, сначала необходимо создать модуль. *Модуль* представляет собой файл с расширением .py, содержащий код, который вы хотите импортировать в свою программу. Давайте создадим модуль с функцией `make_pizza()`. Для этого из файла pizza.py следует удалить все, кроме функции `make_pizza()`:

pizza.py

```
def make_pizza(size, *toppings):
    """Выводит описание пиццы."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
```

Теперь создайте отдельный файл с именем making_pizzas.py в одном каталоге с pizza.py. Файл импортирует только что созданный модуль, а затем дважды вызывает `make_pizza()`:

making_pizzas.py

```
  import pizza

❶ pizza.make_pizza(16, 'pepperoni')
  pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

В процессе обработки этого файла строка `import pizza` говорит Python открыть файл pizza.py и скопировать все функции из него в программу. Вы не видите, как происходит копирование, потому что Python копирует код незаметно для пользователя во время выполнения программы. Вам необходимо знать одно: что любая функция, определенная в pizza.py, будет доступна в making_pizzas.py.

Чтобы вызвать функцию из импортированного модуля, укажите имя модуля (`pizza`), точку и имя функции (`make_pizza()`), как показано в строке ❶. Код выдает тот же результат, что и исходная программа, в которой модуль не импортировался:

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Первый способ импортирования, при котором записывается команда `import` с именем модуля, открывает доступ программе ко всем функциям из модуля. Если вы используете эту разновидность команды `import` для импортирования всего модуля *имя_модуля*.py, то каждая функция модуля будет доступна в следующем синтаксисе:

```
имя_модуля.имя_функции()
```

Импортирование конкретных функций

Также возможно импортировать конкретную функцию из модуля. Общий синтаксис выглядит так:

```
from имя_модуля import имя_функции
```

Вы можете импортировать любое количество функций из модуля, разделив их имена запятыми:

```
from имя_модуля import функция_0, функция_1, функция_2
```

Если ограничиться импортированием только той функции, которую вы намереваетесь использовать, пример making_pizzas.py будет выглядеть так:

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

При таком синтаксисе использовать точечную запись при вызове функции не обязательно. Так как функция `make_pizza()` явно импортируется в команде `import`, при использовании ее можно вызывать прямо по имени.

Назначение псевдонима для функции

Если имя импортируемой функции может конфликтовать с именем существующей функции или функция имеет слишком длинное имя, его можно заменить коротким уникальным *псевдонимом* (alias) — альтернативным именем для функции. Псевдоним назначается функции при импортировании.

В следующем примере функции `make_pizza()` назначается псевдоним `mp()`, для чего при импортировании используется конструкция `make_pizza as mp`. Ключевое слово `as` переименовывает функцию, используя указанный псевдоним:

```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Команда `import` в этом примере назначает функции `make_pizza()` псевдоним `mp()` для этой программы. Каждый раз, когда потребуется вызвать `make_pizza()`, достаточно включить вызов `mp()` — Python выполнит код `make_pizza()` без конфликтов с другой функцией `make_pizza()`, которую вы могли включить в этот файл программы.

Общий синтаксис назначения псевдонима выглядит так:

```
from имя_модуля import имя_функции as псевдоним
```

Назначение псевдонима для модуля

Псевдоним также можно назначить для всего модуля. Назначение короткого имени для модуля — скажем, `p` для `pizza` — позволит вам быстрее вызывать функции модуля. Вызов `p.make_pizza()` получается более компактным, чем `pizza.make_pizza()`:

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Модулю `pizza` в команде `import` назначается псевдоним `p`, но все функции модуля сохраняют свои исходные имена. Вызов функций в записи `p.make_pizza()` не только компактнее `pizza.make_pizza()`; он также отвлекает внимание от имени модуля и помогает сосредоточиться на содержательных именах функций. Эти имена функций, четко показывающие, что делает каждая функция, важнее для удобочитаемости вашего кода, чем использование полного имени модуля.

Общий синтаксис выглядит так:

```
import имя_модуля as псевдоним
```

Импортирование всех функций модуля

Также можно приказать Python импортировать каждую функцию в модуле; для этого используется оператор *:

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Звездочка в команде `import` приказывает Python скопировать каждую функцию из модуля `pizza` в файл программы. После импортирования всех функций вы сможете

вызывать каждую функцию по имени без точечной записи. Тем не менее лучше не использовать этот способ с большими модулями, написанными другими разработчиками; если модуль содержит функцию, имя которой совпадает с существующим именем из вашего проекта, возможны неожиданные результаты. Python обнаруживает несколько функций или переменных с одинаковыми именами, и вместо импортирования всех функций по отдельности происходит замена этих функций.

В таких ситуациях лучше всего импортировать только нужную функцию или функции или же импортировать весь модуль с последующим применением точечной записи. При этом создается чистый код, легко читаемый и понятный. Я включил этот раздел только для того, чтобы вы понимали команды `import` вроде следующей, когда вы встретите их в чужом коде:

```
from имя_модуля import *
```

Стилевое оформление функций

В стилевом оформлении функций необходимо учитывать некоторые подробности. Функции должны иметь содержательные имена, состоящие из букв нижнего регистра и символов подчеркивания. Содержательные имена помогают вам и другим разработчикам понять, что же делает ваш код. Эти соглашения следует соблюдать и в именах модулей.

Каждая функция должна быть снабжена комментарием, который кратко поясняет, что же делает эта функция. Комментарий должен следовать сразу же за определением функции в формате строк документации. Если функция хорошо документирована, другие разработчики смогут использовать ее, прочитав только описание. Конечно, для этого они должны доверять тому, что код работает в соответствии с описанием, но если знать имя функции, то, какие аргументы ей нужны и какое значение она возвращает, они смогут использовать ее в своих программах.

Если для параметра задается значение по умолчанию, слева и справа от знака равенства не должно быть пробелов:

```
def имя_функции(параметр_0, параметр_1='значение_по_умолчанию')
```

Те же соглашения должны применяться для именованных аргументов в вызовах функций:

```
имя_функции(значение_0, параметр_1='значение')
```

Документ PEP 8 (https://www.python.org/dev/peps/pep-0008/) рекомендует ограничить длину строк кода 79 символами, чтобы строки были полностью видны в окне редактора нормального размера. Если из-за параметров длина определения функции превышает 79 символов, нажмите Enter после открывающей круглой скобки в строке определения. В следующей строке дважды нажмите Tab, чтобы отделить список аргументов от тела функции, которое должно быть снабжено отступом только на один уровень.

Многие редакторы автоматически выравнивают дополнительные строки параметров по отступам, установленным в первой строке:

```
def имя_функции(
        параметр_0, параметр_1, параметр_2,
        параметр_3, параметр_4, параметр_5):
    тело функции...
```

Если программа или модуль состоят из нескольких функций, эти функции можно разделить двумя пустыми строками. Так вам будет проще увидеть, где кончается одна функция и начинается другая.

Все команды `import` следует записывать в начале файла. У этого правила есть только одно исключение: файл может начинаться с комментариев, описывающих программу в целом.

УПРАЖНЕНИЯ

8.15. Печать моделей: выделите функции примера print_models.py в отдельный файл с именем printing_functions.py. Разместите команду `import` в начале файла print_models.py и измените файл так, чтобы в нем использовались импортированные функции.

8.16. Импортирование: возьмите за основу одну из написанных вами программ с одной функцией. Сохраните эту функцию в отдельном файле. Импортируйте функцию в файл основной программы и вызовите функцию каждым из следующих способов:

```
import имя_модуля
from имя_модуля import имя_функции
from имя_модуля import имя_функции as псевдоним
import имя_модуля as псевдоним
from имя_модуля import *
```

8.17. Стилевое оформление функций: выберите любые три программы, написанные для этой главы. Убедитесь в том, что в них соблюдаются рекомендации стилевого оформления, представленные в этом разделе.

Итоги

В этой главе вы научились писать функции и передавать аргументы, в которых функциям передается информация, необходимая для их работы. Вы узнали, как использовать позиционные и именованные аргументы и как передать функции произвольное количество аргументов. Вы видели функции, которые выводят данные, и функции, которые возвращают значения. Вы научились использовать функции со списками, словарями, командами `if` и циклами `while`. Также вы научились сохранять функции в отдельных файлах, называемых *модулями*, чтобы код ваших программ стал проще и понятнее. Глава завершается рекомендациями по стилевому оформлению функций, чтобы ваши программы были хорошо структурированы и легко читались вами и другими разработчиками.

Каждый программист должен стремиться к написанию простого кода, который справляется с поставленной задачей, и функции помогают вам в этом. Вы сможете писать блоки кода и оставлять их на будущее. Когда вы знаете, что функция правильно справляется со своей задачей, считайте, что она работает, и переходите к следующей задаче.

В программе с использованием функций единожды написанный код может заново использоваться столько раз, сколько потребуется. Чтобы выполнить код, содержащийся в функции, достаточно написать всего одну строку с вызовом, а функция сделает все остальное. Если же потребуется модифицировать поведение функции, достаточно внести изменения всего в одном месте; они вступят в силу повсюду, где вызывается эта функция.

С функциями ваши программы проще читаются, а хорошо выбранные имена функций описывают, что делает та или иная часть программы. Прочитав серию вызовов функций, вы гораздо быстрее поймете, что делает функция, чем при чтении длинной серии программных блоков.

Функции также упрощают тестирование и отладку кода. Когда основная работа программы выполняется набором функций, каждая из которых решает одну конкретную задачу, вам будет намного проще организовать тестирование и сопровождение вашего кода. Напишите отдельную программу, которая вызывает каждую функцию и проверяет ее работоспособность во всех типичных ситуациях. В этом случае вы можете быть уверены в том, что ваши функции всегда работают правильно.

В главе 9 вы научитесь писать классы. *Классы* объединяют функции и данные в один удобный пакет, с которым программист может работать гибко и эффективно.

9 Классы

Объектно-ориентированное программирование по праву считается одной из самых эффективных методологий создания программных продуктов. В объектно-ориентированном программировании вы создаете классы, описывающие реально существующие предметы и ситуации, а затем создаете *объекты* на основе этих описаний. При написании класса определяется общее поведение для целой категории объектов.

Когда вы создаете конкретные объекты на базе этих классов, каждый объект автоматически наделяется общим поведением; после этого вы можете наделить каждый объект уникальными особенностями на свой выбор. Просто невероятно, насколько хорошо реальные ситуации моделируются в объектно-ориентированном программировании.

Создание объекта на основе класса называется *созданием экземпляра*; таким образом, вы работаете с экземплярами класса. В этой главе вы будете писать классы и создавать экземпляры этих классов. Вы будете указывать, какая информация может храниться в экземплярах, и определять действия, которые могут выполняться с экземплярами. Также вы будете писать классы, расширяющие функциональность существующих классов; это позволяет организовать эффективное совместное использование кода похожими классами. Вы будете сохранять классы в модулях и импортировать классы, написанные другими программистами, в ваши программные файлы.

С хорошим пониманием объектно-ориентированного программирования вы взглянете на мир с точки зрения программиста. Вам будет проще понять свой код — увидеть не только то, что происходит в каждой его строке, а более масштабные концепции, лежащие в его основе. Логика, заложенная в основу классов, научит вас мыслить логически, чтобы ваши программы эффективно решали практически любые задачи, с которыми вы можете столкнуться.

Кроме того, классы упрощают жизнь вам и другим программистам, с которыми вам придется работать совместно над более серьезными проектами. Когда вы и другие программисты пишете код, базирующийся на сходной логике, вам будет проще разобраться в коде, написанном другими людьми. Ваши программы будут понятны коллегам, и в результате все вы сможете добиться больших результатов.

Создание и использование класса

Классы позволяют моделировать практически все что угодно. Начнем с написания простого класса `Dog`, представляющего собаку — не какую-то конкретную, а собаку вообще. Что мы знаем о собаках? У них есть кличка и возраст. Также известно, что большинство собак умеют садиться и перекатываться по команде. Эти два вида информации (кличка и возраст) и два вида поведения (сидеть и перекатываться) будут включены в класс `Dog`, потому что они являются общими для большинства собак. Класс сообщает Python, как создать объект, представляющий собаку. После того как класс будет написан, мы используем его для создания экземпляров, каждый из которых представляет одну конкретную собаку.

Создание класса Dog

В каждом экземпляре, созданном на основе класса `Dog`, будет храниться кличка (`name`) и возраст (`age`); кроме того, в нем будут присутствовать методы `sit()` и `roll_over()`:

dog.py

```
❶ class Dog():
❷     """Простая модель собаки."""

❸     def __init__(self, name, age):
          """Инициализирует атрибуты name и age."""
❹         self.name = name
          self.age = age

❺     def sit(self):
          """Собака садится по команде."""
          print(f"{self.name} is now sitting.")

      def roll_over(self):
          """Собака перекатывается по команде."""
          print(f"{self.name} rolled over!")
```

В этом коде есть много мест, заслуживающих вашего внимания, но не беспокойтесь. Эта структура неоднократно встретится вам в этой главе, и вы еще успеете к ней привыкнуть. В точке ❶ определяется класс с именем `Dog`. По общепринятым соглашениям имена, начинающиеся с символа верхнего регистра, в Python обозначают классы. Круглые скобки в определении класса пусты, потому что класс создается с нуля. В точке ❷ приведена строка документации с кратким описанием класса.

Метод __init__()

Функция, являющаяся частью класса, называется *методом*. Все, что вы узнали ранее о функциях, также относится и к методам; единственное практическое различие — способ вызова методов. Метод `__init__()` в точке ❸ — специальный метод,

который автоматически выполняется при создании каждого нового экземпляра на базе класса `Dog`. Имя метода начинается и заканчивается двумя символами подчеркивания; эта схема предотвращает конфликты имен стандартных методов Python и методов ваших классов. Будьте внимательны: два символа подчеркивания должны стоять на *каждой* стороне `__init__()`. Если вы поставите только один символ подчеркивания с каждой стороны, то метод не будет вызываться автоматически при использовании класса, что может привести к появлению коварных ошибок.

Метод `__init__()` определяется с тремя параметрами: `self`, `name` и `age`. Параметр `self` обязателен в определении метода; он должен предшествовать всем остальным параметрам. Он должен быть включен в определение, потому что при будущем вызове метода `__init__()` (для создания экземпляра `Dog`) Python автоматически передает аргумент `self`. При каждом вызове метода, связанного с классом, автоматически передается `self` — ссылка на экземпляр; она предоставляет конкретному экземпляру доступ к атрибутам и методам класса. Когда вы создаете экземпляр `Dog`, Python вызывает метод `__init__()` из класса `Dog`. Мы передаем `Dog()` кличку и возраст в аргументах; значение `self` передается автоматически, так что его передавать не нужно. Каждый раз, когда вы захотите создать экземпляр на основе класса `Dog`, необходимо предоставить значения только двух последних аргументов, `name` и `age`.

Каждая из двух переменных, определяемых в точке ❹, снабжена префиксом `self`. Любая переменная с префиксом `self` доступна для каждого метода в классе, и вы также сможете обращаться к этим переменным в каждом экземпляре, созданном на основе класса. Конструкция `self.name = name` берет значение, хранящееся в параметре `name`, и сохраняет его в переменной `name`, которая затем связывается с создаваемым экземпляром. Процесс также повторяется с `self.age = age`. Переменные, к которым вы обращаетесь через экземпляры, также называются *атрибутами*.

В классе `Dog` также определяются два метода: `sit()` и `roll_over()` ❺. Так как этим методам не нужна дополнительная информация (кличка или возраст), они определяются с единственным параметром `self`. Экземпляры, которые будут созданы позднее, смогут вызывать эти методы. Пока методы `sit()` и `roll_over()` ограничиваются простым выводом сообщения о том, что собака садится или перекатывается. Тем не менее концепцию легко расширить для практического применения: если бы этот класс был частью компьютерной игры, то эти методы вполне могли бы содержать код для создания анимации садящейся или перекатывающейся собаки. А если бы класс был написан для управления роботом, то методы могли бы управлять механизмами, заставляющими робота-собаку выполнить соответствующую команду.

Создание экземпляра класса

Считайте, что класс — это своего рода инструкция по созданию экземпляров. Соответственно класс `Dog` — инструкция по созданию экземпляров, представляющих конкретных собак.

Создадим экземпляр, представляющий конкретную собаку:

```
  class Dog():
      ...
❶ my_dog = Dog('willie', 6)

❷ print(f"My dog's name is {my_dog.name}.")
❸ print(f"My dog is {my_dog.age} years old.")
```

Использованный в данном случае класс `Dog` был написан в предыдущем примере. В точке ❶ мы приказываем Python создать экземпляр собаки с кличкой `'willie'` и возрастом 6 лет. В процессе обработки этой строки Python вызывает метод `__init__()` класса `Dog` с аргументами `'willie'` и 6. Метод `__init__()` создает экземпляр, представляющий конкретную собаку, и присваивает его атрибутам `name` и `age` переданные значения. Затем Python возвращает экземпляр, представляющий собаку. Этот экземпляр сохраняется в переменной `my_dog`. Здесь нелишне вспомнить соглашения по записи имен: обычно считается, что имя, начинающееся с символа верхнего регистра (например, `Dog`), обозначает класс, а имя, записанное в нижнем регистре (например, `my_dog`), обозначает отдельный экземпляр, созданный на базе класса.

Обращение к атрибутам

Для обращения к атрибутам экземпляра используется «точечная» запись. В строке ❷ мы обращаемся к значению атрибута `name` экземпляра `my_dog`:

```
my_dog.name
```

Точечная запись часто используется в Python. Этот синтаксис показывает, как Python ищет значения атрибутов. В данном случае Python обращается к экземпляру `my_dog` и ищет атрибут `name`, связанный с экземпляром `my_dog`. Это тот же атрибут, который обозначался `self.name` в классе `Dog`. В точке ❸ тот же прием используется для работы с атрибутом `age`.

Пример выводит известную информацию о `my_dog`:

```
My dog's name is Willie.
My dog is 6 years old.
```

Вызов методов

После создания экземпляра на основе класса `Dog` можно применять точечную запись для вызова любых методов, определенных в `Dog`:

```
class Dog():
...

my_dog = Dog('willie', 6)
my_dog.sit()
my_dog.roll_over()
```

Чтобы вызвать метод, укажите экземпляр (в данном случае `my_dog`) и вызываемый метод, разделив их точкой. В ходе обработки `my_dog.sit()` Python ищет метод `sit()` в классе `Dog` и выполняет его код. Строка `my_dog.roll_over()` интерпретируется аналогичным образом.

Теперь экземпляр послушно выполняет полученные команды:

```
Willie is now sitting.
Willie rolled over!
```

Это очень полезный синтаксис. Если атрибутам и методам были присвоены содержательные имена (например, `name`, `age`, `sit()` и `roll_over()`), разработчик сможет легко понять, что делает блок кода — даже если он видит этот блок впервые.

Создание нескольких экземпляров

На основе класса можно создать столько экземпляров, сколько вам потребуется. Создадим второй экземпляр `Dog` с именем `your_dog`:

```
class Dog():
...

my_dog = Dog('willie', 6)
your_dog = Dog('lucy', 3)

print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()

print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

В этом примере создаются два экземпляра с именами `Willie` и `Lucy`. Каждый экземпляр обладает своим набором атрибутов и способен выполнять действия из общего набора:

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

Даже если второй собаке будет назначено то же имя и возраст, Python все равно создаст отдельный экземпляр класса `Dog`. Вы можете создать сколько угодно экземпляров одного класса при условии, что эти экземпляры хранятся в переменных с разными именами или занимают разные позиции в списке либо словаре.

УПРАЖНЕНИЯ

9.1. Ресторан: создайте класс с именем `Restaurant`. Метод `__init__()` класса `Restaurant` должен содержать два атрибута: `restaurant_name` и `cuisine_type`. Создайте метод `describe_restaurant()`, который выводит два атрибута, и метод `open_restaurant()`, который выводит сообщение о том, что ресторан открыт.

Создайте на основе своего класса экземпляр с именем `restaurant`. Выведите два атрибута по отдельности, затем вызовите оба метода.

9.2. Три ресторана: начните с класса из упражнения 9.1. Создайте три разных экземпляра, вызовите для каждого экземпляра метод `describe_restaurant()`.

9.3. Пользователи: создайте класс с именем `User`. Создайте два атрибута `first_name` и `last_name`, а затем еще несколько атрибутов, которые обычно хранятся в профиле пользователя. Напишите метод `describe_user()`, который выводит сводку с информацией о пользователе. Создайте еще один метод `greet_user()` для вывода персонального приветствия для пользователя.

Создайте несколько экземпляров, представляющих разных пользователей. Вызовите оба метода для каждого пользователя.

Работа с классами и экземплярами

Классы могут использоваться для моделирования многих реальных ситуаций. После того как класс будет написан, разработчик проводит большую часть времени за работой с экземплярами, созданными на основе этого класса. Одной из первых задач станет изменение атрибутов, связанных с конкретным экземпляром. Атрибуты экземпляра можно изменять напрямую или же написать методы, изменяющие атрибуты по особым правилам.

Класс Car

Напишем класс, представляющий автомобиль. Этот класс будет содержать информацию о типе машины, а также метод для вывода краткого описания:

car.py

```
    class Car():
        """Простая модель автомобиля."""

❶       def __init__(self, make, model, year):
            """Инициализирует атрибуты описания автомобиля."""
            self.make = make
            self.model = model
            self.year = year

❷       def get_descriptive_name(self):
            """Возвращает аккуратно отформатированное описание."""
            long_name = f"{self.year} {self.manufacturer} {self.model}"
            return long_name.title()
```

```
❸ my_new_car = Car('audi', 'a4', 2019)
  print(my_new_car.get_descriptive_name())
```

В точке ❶ в классе `Car` определяется метод `__init__()`; его список параметров начинается с `self`, как и в классе `Dog`. За ним следуют еще три параметра: `make`, `model` и `year`. Метод `__init__()` получает эти параметры и сохраняет их в атрибутах, которые будут связаны с экземплярами, созданными на основе класса. При создании нового экземпляра `Car` необходимо указать фирму-производитель, модель и год выпуска для данного экземпляра.

В точке ❷ определяется метод `get_descriptive_name()`, который объединяет год выпуска, фирму-производитель и модель в одну строку с описанием. Это избавит вас от необходимости выводить значение каждого атрибута по отдельности. Для работы со значениями атрибутов в этом методе используется синтаксис `self.make`, `self.model` и `self.year`. В точке ❸ создается экземпляр класса `Car`, который сохраняется в переменной `my_new_car`. Затем вызов метода `get_descriptive_name()` показывает, с какой машиной работает программа:

```
2019 Audi A4
```

Чтобы класс был более интересным, добавим атрибут, изменяющийся со временем, — в нем будет храниться пробег машины в милях.

Назначение атрибуту значения по умолчанию

Каждый атрибут класса должен иметь исходное значение, даже если оно равно 0 или пустой строке. В некоторых случаях (например, при задании значений по умолчанию) это исходное значение есть смысл задавать в теле метода `__init__()`; в таком случае передавать параметр для этого атрибута при создании объекта не обязательно.

Добавим атрибут с именем `odometer_reading`, исходное значение которого всегда равно 0. Также в класс будет включен метод `read_odometer()` для чтения текущих показаний одометра:

```
  class Car():

      def __init__(self, make, model, year):
          """Инициализирует атрибуты описания автомобиля."""
          self.make = make
          self.model = model
          self.year = year
❶         self.odometer_reading = 0

      def get_descriptive_name(self):
          ...

❷     def read_odometer(self):
          """Выводит пробег машины в милях."""
```

```
        print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

Когда Python вызывает метод `__init__()` для создания нового экземпляра, этот метод сохраняет фирму-производителя, модель и год выпуска в атрибутах, как и в предыдущем случае. Затем Python создает новый атрибут с именем `odometer_reading` и присваивает ему исходное значение 0 ❶. Также в класс добавляется новый метод `read_odometer()` ❷, который упрощает чтение пробега машины в милях.

Сразу же после создания машины ее пробег равен 0:

```
2019 Audi A4
This car has 0 miles on it.
```

Впрочем, у продаваемых машин одометр редко показывает ровно 0, поэтому нам понадобится способ изменения значения этого атрибута.

Изменение значений атрибутов

Значение атрибута можно изменить одним из трех способов: изменить его прямо в экземпляре, задать значение при помощи метода или изменить его с приращением (то есть прибавлением определенной величины) при помощи метода. Рассмотрим все эти способы.

Прямое изменение значения атрибута

Чтобы изменить значение атрибута, проще всего обратиться к нему прямо через экземпляр. В следующем примере на одометре напрямую выставляется значение 23:

```
class Car:
--snip--

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

❶ my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

В точке ❶ точечная запись используется для обращения к атрибуту `odometer_reading` экземпляра и прямого присваивания его значения. Эта строка приказывает Python взять экземпляр `my_new_car`, найти связанный с ним атрибут `odometer_reading` и задать значение атрибута равным 23:

```
2019 Audi A4
This car has 23 miles on it.
```

Иногда подобные прямые обращения к атрибутам допустимы, но чаще разработчик пишет вспомогательный метод, который изменяет значение за него.

Изменение значения атрибута с использованием метода

В класс можно включить методы, которые изменяют некоторые атрибуты за вас. Вместо того чтобы изменять атрибут напрямую, вы передаете новое значение методу, который берет обновление атрибута на себя.

В следующем примере в класс включается метод `update_odometer()` для изменения показаний одометра:

```
  class Car:
      ...

❶     def update_odometer(self, mileage):
          """Устанавливает заданное значение на одометре."""
          self.odometer_reading = mileage

  my_new_car = Car('audi', 'a4', 2019)
  print(my_new_car.get_descriptive_name())

❷ my_new_car.update_odometer(23)
  my_new_car.read_odometer()
```

Класс `Car` почти не изменился, в нем только добавился метод `update_odometer()`❶. Этот метод получает пробег в милях и сохраняет его в `self.odometer_reading`. В точке ❷ мы вызываем метод `update_odometer()` и передаем ему значение 23 в аргументе (соответствующем параметру `mileage` в определении метода). Метод устанавливает на одометре значение 23, а метод `read_odometer()` выводит текущие показания:

```
2019 Audi A4
This car has 23 miles on it.
```

Метод `update_odometer()` можно расширить так, чтобы при каждом изменении показаний одометра выполнялась некоторая дополнительная работа. Добавим проверку, которая гарантирует, что никто не будет пытаться сбрасывать показания одометра:

```
  class Car():
      ...

      def update_odometer(self, mileage):
          """
          Устанавливает на одометре заданное значение.
          При попытке обратной подкрутки изменение отклоняется.
          """
❶         if mileage >= self.odometer_reading:
              self.odometer_reading = mileage
          else:
❷             print("You can't roll back an odometer!")
```

Теперь `update_odometer()` проверяет новое значение перед изменением атрибута. Если новое значение `mileage` больше или равно текущему, `self.odometer_reading`, показания одометра можно обновить новым значением ❶. Если же новое значение меньше текущего, вы получите предупреждение о недопустимости обратной подкрутки ❷.

Изменение значения атрибута с приращением

Иногда значение атрибута требуется изменить с заданным приращением (вместо того, чтобы присваивать атрибуту произвольное новое значение). Допустим, вы купили подержанную машину и проехали на ней 100 миль с момента покупки. Следующий метод получает величину приращения и прибавляет ее к текущим показаниям одометра:

```
  class Car():
      ...

      def update_odometer(self, mileage):
          ...

❶     def increment_odometer(self, miles):
          """Увеличивает показания одометра с заданным приращением."""
          self.odometer_reading += miles

❷ my_used_car = Car('subaru', 'outback', 2015)
  print(my_used_car.get_descriptive_name())

❸ my_used_car.update_odometer(23_500)
  my_used_car.read_odometer()

❹ my_used_car.increment_odometer(100)
  my_used_car.read_odometer()
```

Новый метод `increment_odometer()` в точке ❶ получает расстояние в милях и прибавляет его к `self.odometer_reading`. В точке ❷ создается экземпляр `my_used_car`. Мы инициализируем показания его одометра значением 23 500; для этого вызывается метод `update_odometer()`, которому передается значение 23 500 ❸. В точке ❹ вызывается метод `increment_odometer()`, которому передается значение 100, чтобы увеличить показания одометра на 100 миль, пройденные с момента покупки:

```
2015 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

При желании можно легко усовершенствовать этот метод, чтобы он отклонял отрицательные приращения; тем самым вы предотвратите обратную подкрутку одометра.

ПРИМЕЧАНИЕ Подобные методы управляют обновлением внутренних значений экземпляров (таких, как показания одометра), однако любой пользователь, имеющий доступ к программному коду, сможет напрямую задать атрибуту любое значение. Эффективная схема безопасности должна уделять особое внимание таким подробностям, не ограничиваясь простейшими проверками.

УПРАЖНЕНИЯ

9.4. Посетители: начните с программы из упражнения 9.1 (с. 175). Добавьте атрибут `number_served` со значением по умолчанию `0`; он представляет количество обслуженных посетителей. Создайте экземпляр с именем `restaurant`. Выведите значение `number_served`, потом измените и выведите снова.

Добавьте метод с именем `set_number_served()`, позволяющий задать количество обслуженных посетителей. Вызовите метод с новым числом, снова выведите значение.

Добавьте метод с именем `increment_number_served()`, который увеличивает количество обслуженных посетителей на заданную величину. Вызовите этот метод с любым числом, которое могло бы представлять количество обслуженных клиентов, — скажем, за один день.

9.5. Попытки входа: добавьте атрибут `login_attempts` в класс `User` из упражнения 9.3 (с. 175). Напишите метод `increment_login_attempts()`, увеличивающий значение `login_attempts` на `1`. Напишите другой метод с именем `reset_login_attempts()`, обнуляющий значение `login_attempts`.

Создайте экземпляр класса `User` и вызовите `increment_login_attempts()` несколько раз. Выведите значение `login_attempts`, чтобы убедиться в том, что значение было изменено правильно, а затем вызовите `reset_login_attempts()`. Снова выведите `login_attempts` и убедитесь в том, что значение обнулилось.

Наследование

Работа над новым классом не обязана начинаться с нуля. Если класс, который вы пишете, представляет собой специализированную версию ранее написанного класса, вы можете воспользоваться *наследованием*. Один класс, *наследующий* от другого, автоматически получает все атрибуты и методы первого класса. Исходный класс называется *родителем*, а новый класс — *потомком*. Класс-потомок наследует атрибуты и методы родителя, но при этом также может определять собственные атрибуты и методы.

Метод __init__() класса-потомка

При написании нового класса на базе существующего класса часто приходится вызывать метод `__init__()` класса-родителя. При этом происходит инициализация любых атрибутов, определенных в методе `__init__()` родителя, и эти атрибуты становятся доступными для класса-потомка.

Например, попробуем построить модель электромобиля. Электромобиль представляет собой специализированную разновидность автомобиля, поэтому новый класс `ElectricCar` можно создать на базе класса `Car`, написанного ранее. Тогда

нам останется добавить в него код атрибутов и поведения, относящегося только к электромобилям.

Начнем с создания простой версии класса `ElectricCar`, который делает все, что делает класс `Car`:

electric_car.py

```
❶ class Car():
      """Простая модель автомобиля."""

      def __init__(self, make, model, year):
          self.make = make
          self.model = model
          self.year = year
          self.odometer_reading = 0

      def get_descriptive_name(self):
          long_name = f"{self.year} {self.manufacturer} {self.model}"
          return long_name.title()

      def read_odometer(self):
          print(f"This car has {self.odometer_reading} miles on it.")

      def update_odometer(self, mileage):
          if mileage >= self.odometer_reading:
              self.odometer_reading = mileage
          else:
              print("You can't roll back an odometer!")

      def increment_odometer(self, miles):
          self.odometer_reading += miles

❷ class ElectricCar(Car):
      """Представляет аспекты машины, специфические для электромобилей."""

❸     def __init__(self, make, model, year):
          """инициализирует атрибуты класса-родителя."""
❹         super().__init__(make, model, year)

❺ my_tesla = ElectricCar('tesla', 'model s', 2019)
  print(my_tesla.get_descriptive_name())
```

В точке ❶ строится экземпляр `Car`. При создании класса-потомка класс-родитель должен быть частью текущего файла, а его определение должно предшествовать определению класса-потомка в файле. В точке ❷ определяется класс-потомок `ElectricCar`. В определении потомка имя класса-родителя заключается в круглые скобки. Метод `__init__()` в точке ❸ получает информацию, необходимую для создания экземпляра `Car`.

Функция `super()` в строке ❹ — специальная функция, которая позволяет вызвать метод родительского класса. Эта строка приказывает Python вызвать метод `__init__()` класса `Car`, в результате чего экземпляр `ElectricCar` получает доступ

ко всем атрибутам класса-родителя. Имя `super` происходит из распространенной терминологии: класс-родитель называется *суперклассом*, а класс-потомок — *подклассом*.

Чтобы проверить, правильно ли сработало наследование, попробуем создать электромобиль с такой же информацией, которая передается при создании обычного экземпляра `Car`. В точке ❺ мы создаем экземпляр класса `ElectricCar` и сохраняем его в `my_tesla`. Эта строка вызывает метод `__init__()`, определенный в `ElectricCar`, который, в свою очередь, приказывает Python вызвать метод `__init__()`, определенный в классе-родителе `Car`. При вызове передаются аргументы `'tesla'`, `'model s'` и `2019`.

Кроме `__init__()`, класс еще не содержит никаких атрибутов или методов, специфических для электромобилей. Пока мы просто убеждаемся в том, что класс электромобиля содержит все поведение, присущее классу автомобиля:

```
2019 Tesla Model S
```

Экземпляр `ElectricCar` работает так же, как экземпляр `Car`; можно переходить к определению атрибутов и методов, специфических для электромобилей.

Определение атрибутов и методов класса-потомка

После создания класса-потомка, наследующего от класса-родителя, можно переходить к добавлению новых атрибутов и методов, необходимых для того, чтобы потомок отличался от родителя.

Добавим атрибут, специфический для электромобилей (например, мощность аккумулятора), и метод для вывода информации об этом атрибуте:

```
class Car():
    ...

class ElectricCar(Car):
    """Представляет аспекты машины, специфические для электромобилей."""

    def __init__(self, make, model, year):
        """
        Инициализирует атрибуты класса-родителя.
        Затем инициализирует атрибуты, специфические для электромобиля.
        """
        super().__init__(make, model, year)
❶       self.battery_size = 75

❷   def describe_battery(self):
        """Выводит информацию о мощности аккумулятора."""
        print(f"This car has a {self.battery_size}-kWh battery.")

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

В точке ❶ добавляется новый атрибут `self.battery_size`, которому присваивается исходное значение — скажем, 75. Этот атрибут будет присутствовать во всех экземплярах, созданных на основе класса `ElectricCar` (но не во всяком экземпляре `Car`). Также добавляется метод с именем `describe_battery()`, который выводит информацию об аккумуляторе в точке ❷. При вызове этого метода выводится описание, которое явно относится только к электромобилям:

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

Возможности специализации класса `ElectricCar` беспредельны. Вы можете добавить сколько угодно атрибутов и методов, чтобы моделировать электромобиль с любой нужной точностью. Атрибуты или методы, которые могут принадлежать любой машине (а не только электромобилю), должны добавляться в класс `Car` вместо `ElectricCar`. Тогда эта информация будет доступна всем пользователям класса `Car`, а класс `ElectricCar` будет содержать только код информации и поведения, специфических для электромобилей.

Переопределение методов класса-родителя

Любой метод родительского класса, который в моделируемой ситуации делает не то, что нужно, можно переопределить. Для этого в классе-потомке определяется метод с тем же именем, что и у метода класса-родителя. Python игнорирует метод родителя и обращает внимание только на метод, определенный в потомке.

Допустим, в классе `Car` имеется метод `fill_gas_tank()`. Для электромобилей заправка бензином бессмысленна, поэтому этот метод логично переопределить. Например, это можно сделать так:

```
class ElectricCar(Car):
    ...

    def fill_gas_tank(self):
        """У электромобилей нет бензобака."""
        print("This car doesn't need a gas tank!")
```

И если кто-то попытается вызвать метод `fill_gas_tank()` для электромобиля, Python проигнорирует метод `fill_gas_tank()` класса `Car` и выполнит вместо него этот код. С применением наследования потомок сохраняет те аспекты родителя, которые вам нужны, и переопределяет все ненужное.

Экземпляры как атрибуты

При моделировании явлений реального мира в программах классы нередко дополняются все большим количеством подробностей. Списки атрибутов и методов растут, и через какое-то время файлы становятся длинными и громоздкими. В такой ситуации часть одного класса нередко можно записать в виде отдельного

класса. Большой код разбивается на меньшие классы, которые работают во взаимодействии друг с другом.

Например, при дальнейшей доработке класса `ElectricCar` может оказаться, что в нем появилось слишком много атрибутов и методов, относящихся к аккумулятору. В таком случае можно остановиться и переместить все эти атрибуты и методы в отдельный класс с именем `Battery`. Затем экземпляр `Battery` становится атрибутом класса `ElectricCar`:

```
class Car():
...

❶ class Battery():
    """Простая модель аккумулятора электромобиля."""

❷     def __init__(self, battery_size=75):
        """Инициализирует атрибуты аккумулятора."""
        self.battery_size = battery_size

❸     def describe_battery(self):
        """Выводит информацию о мощности аккумулятора."""
        print(f"This car has a {self.battery_size}-kWh battery.")

class ElectricCar(Car):
    """Представляет аспекты машины, специфические для электромобилей."""

    def __init__(self, make, model, year):
        """
        Инициализирует атрибуты класса-родителя.
        Затем инициализирует атрибуты, специфические для электромобиля.
        """
        super().__init__(make, model, year)
❹       self.battery = Battery()

my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
```

В точке ❶ определяется новый класс с именем `Battery`, который не наследует ни один из других классов. Метод `__init__()` в точке ❷ получает один параметр `battery_size`, кроме `self`. Если значение не предоставлено, этот необязательный параметр задает `battery_size` значение 75. Метод `describe_battery()` также перемещен в этот класс ❸.

Затем в класс `ElectricCar` добавляется атрибут с именем `self.battery` ❹. Эта строка приказывает Python создать новый экземпляр `Battery` (со значением `battery_size` по умолчанию, равным 75, потому что значение не задано) и сохранить его в атрибуте `self.battery`. Это будет происходить при каждом вызове `__init__()`; теперь любой экземпляр `ElectricCar` будет иметь автоматически создаваемый экземпляр `Battery`.

Программа создает экземпляр электромобиля и сохраняет его в переменной `my_tesla`. Когда потребуется вывести описание аккумулятора, необходимо обратиться к атрибуту `battery`:

```
my_tesla.battery.describe_battery()
```

Эта строка приказывает Python обратиться к экземпляру `my_tesla`, найти его атрибут `battery` и вызвать метод `describe_battery()`, связанный с экземпляром `Battery` из атрибута.

Результат выглядит так же, как и в предыдущей версии:

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

Казалось бы, новый вариант требует большой дополнительной работы, но теперь аккумулятор можно моделировать с любой степенью детализации без загромождения класса `ElectricCar`. Добавим в `Battery` еще один метод, который выводит запас хода на основании мощности аккумулятора:

```
  class Car():
  ...

  class Battery():
  ...

❶     def get_range(self):
          """Выводит приблизительный запас хода для аккумулятора."""
         if self.battery_size == 75:
              range = 260
          elif self.battery_size == 100:
              range = 315

          print(f"This car can go about {range} miles on a full charge.")

  class ElectricCar(Car):
  ...

  my_tesla = ElectricCar('tesla', 'model s', 2019)
  print(my_tesla.get_descriptive_name())
  my_tesla.battery.describe_battery()
❷ my_tesla.battery.get_range()
```

Новый метод `get_range()` в точке ❶ проводит простой анализ. Если мощность равна 75, то `get_range()` устанавливает запас хода 260 миль, а при мощности 100 кВт/ч запас хода равен 315 милям. Затем программа выводит это значение. Когда вы захотите использовать этот метод, его придется вызывать через атрибут `battery` в точке ❷.

Результат сообщает запас хода машины в зависимости от мощности аккумулятора:

```
2019 Tesla Model S
This car has a 75-kWh battery.
This car can go approximately 260 miles on a full charge.
```

Моделирование объектов реального мира

Занявшись моделированием более сложных объектов, таких как электромобили, вы столкнетесь с множеством интересных вопросов. Является ли запас хода электромобиля свойством аккумулятора или машины? Если вы описываете только одну машину, вероятно, можно связать метод `get_range()` с классом `Battery`. Но если моделируется целая линейка машин от производителя, вероятно, метод `get_range()` правильнее будет переместить в класс `ElectricCar`. Метод `get_range()` по-прежнему будет проверять мощность аккумулятора перед определением запаса хода, но он будет сообщать запас хода для той машины, с которой он связан. Также возможно связать метод `get_range()` с аккумулятором, но передавать ему параметр (например, `car_model`). Метод `get_range()` будет определять запас хода на основании мощности аккумулятора и модели автомобиля.

Если вы начнете ломать голову над такими вопросами, это означает, что вы мыслите на более высоком логическом уровне, не ограничиваясь уровнем синтаксиса. Вы думаете уже не о Python, а о том, как лучше представить реальный мир в своем коде. И достигнув этой точки, вы поймете, что однозначно правильного или неправильного подхода к моделированию реальных ситуаций часто не существует. Некоторые методы эффективнее других, но для того, чтобы найти наиболее эффективную реализацию, необходим практический опыт. Если ваш код работает именно так, как вы хотели, — значит, у вас все получается! Не огорчайтесь, если окажется, что вы по несколько раз переписываете свои классы для разных решений. На пути к написанию точного, эффективного кода все программисты проходят через этот процесс.

УПРАЖНЕНИЯ

9.6. Киоск с мороженым: киоск с мороженым — особая разновидность ресторана. Напишите класс IceCreamStand, наследующий от класса Restaurant из упражнения 9.1 (с. 175) или упражнения 9.4 (с. 180). Подойдет любая версия класса; просто выберите ту, которая вам больше нравится. Добавьте атрибут с именем `flavors` для хранения списка сортов мороженого. Напишите метод, который выводит этот список. Создайте экземпляр `IceCreamStand` и вызовите этот метод.

9.7. Администратор: администратор — особая разновидность пользователя. Напишите класс с именем Admin, наследующий от класса `User` из упражнения 9.3 или упражнения 9.5 (с. 180). Добавьте атрибут `privileges` для хранения списка строк вида `"разрешено добавлять сообщения"`, `"разрешено удалять пользователей"`, `"разрешено банить пользователей"` и т. д. Напишите метод `show_privileges()` для вывода набора привилегий администратора. Создайте экземпляр `Admin` и вызовите свой метод.

9.8. Привилегии: напишите класс `Privileges`. Класс должен содержать всего один атрибут `privileges` со списком строк из упражнения 9.7. Переместите метод `show_privileges()` в этот класс. Создайте экземпляр `Privileges` как атрибут класса `Admin`. Создайте новый экземпляр `Admin` и используйте свой метод для вывода списка привилегий.

9.9. Обновление аккумулятора: используйте окончательную версию программы electric_car.py из этого раздела. Добавьте в класс `Battery` метод с именем `upgrade_battery()`. Этот метод должен проверять размер аккумулятора и устанавливать мощность равной 100, если она имеет другое значение. Создайте экземпляр электромобиля с аккумулятором по умолчанию, вызовите `get_range()`, а затем вызовите `get_range()` во второй раз после вызова `upgrade_battery()`. Убедитесь в том, что запас хода увеличился.

Импортирование классов

С добавлением новой функциональности в классы файлы могут стать слишком длинными, даже при правильном использовании наследования. В соответствии с общей философией Python файлы не должны загромождаться лишними подробностями. Для этого Python позволяет хранить классы в модулях и импортировать нужные классы в основную программу.

Импортирование одного класса

Начнем с создания модуля, содержащего только класс `Car`. При этом возникает неочевидный конфликт имен: в этой главе уже был создан файл с именем car.py, но этот модуль тоже должен называться car.py, потому что в нем содержится код класса Car. Мы решим эту проблему, сохранив класс `Car` в модуле с именем car.py, заменяя им файл car.py, который использовался ранее. В дальнейшем любой программе, использующей этот модуль, придется присвоить более конкретное имя файла — например, my_car.py. Ниже приведен файл car.py с кодом класса `Car`:

car.py

```
❶ """Класс для представления автомобиля."""

  class Car():
      """Простая модель автомобиля."""

      def __init__(self, make, model, year):
          """Инициализирует атрибуты описания автомобиля."""
          self.make = make
          self.model = model
          self.year = year
          self.odometer_reading = 0

      def get_descriptive_name(self):
          """Возвращает аккуратно отформатированное описание."""
          long_name = f"{self.year} {self.manufacturer} {self.model}"
          return long_name.title()

      def read_odometer(self):
          """Выводит пробег машины в милях."""
          print(f"This car has {self.odometer_reading} miles on it.")

      def update_odometer(self, mileage):
```

```
        """
        Устанавливает на одометре заданное значение.
        При попытке обратной подкрутки изменение отклоняется.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Увеличивает показания одометра с заданным приращением."""
        self.odometer_reading += miles
```

В точке ❶ включается строка документации уровня модуля с кратким описанием содержимого модуля. Пишите строки документации для каждого созданного вами модуля.

Теперь мы создадим отдельный файл с именем my_car.py. Этот файл импортирует класс `Car` и создает экземпляр этого класса:

my_car.py

```
❶ from car import Car

  my_new_car = Car('audi', 'a4', 2019)
  print(my_new_car.get_descriptive_name())

  my_new_car.odometer_reading = 23
  my_new_car.read_odometer()
```

Команда `import` в точке ❶ приказывает Python открыть модуль `car` и импортировать класс `Car`. Теперь мы можем использовать класс `Car` так, как если бы он был определен в этом файле. Результат остается тем же, что и в предыдущей версии:

```
2019 Audi A4
This car has 23 miles on it.
```

Импортирование классов повышает эффективность программирования. Представьте, каким длинным получился бы файл этой программы, если бы в него был включен весь класс `Car`. Перемещая класс в модуль и импортируя этот модуль, вы получаете ту же функциональность, но основной файл программы при этом остается чистым и удобочитаемым. Большая часть логики также может храниться в отдельных файлах; когда ваши классы работают так, как положено, вы можете забыть об этих файлах и сосредоточиться на высокоуровневой логике основной программы.

Хранение нескольких классов в модуле

В одном модуле можно хранить сколько угодно классов, хотя все эти классы должны быть каким-то образом связаны друг с другом. Оба класса, `Battery`

и `ElectricCar`, используются для представления автомобилей, поэтому мы добавим их в модуль car.py:

car.py

```
"""Классы для представления машин с бензиновым и электродвигателем."""

class Car():
...

class Battery():
    """Простая модель аккумулятора электромобиля."""

    def __init__(self, battery_size=70):
        """Инициализация атрибутов аккумулятора."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Выводит информацию о мощности аккумулятора."""
        print(f"This car has a {self.battery_size}-kWh battery.")

        def get_range(self):
        """Выводит приблизительный запас хода для аккумулятора."""
        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315

        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    """Представляет аспекты машины, специфические для электромобилей."""

    def __init__(self, make, model, year):
        """
        Инициализирует атрибуты класса-родителя.
        Затем инициализирует атрибуты, специфические для электромобиля.
        """
        super().__init__(make, model, year)
        self.battery = Battery()
```

Теперь вы можете создать новый файл с именем my_electric_car.py, импортировать класс `ElectricCar` и создать новый экземпляр электромобиля:

my_electric_car.py

```
from car import ElectricCar

my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

Программа выводит тот же результат, что и в предыдущем случае, хотя большая часть ее логики скрыта в модуле:

```
2019 Tesla Model S
This car has a 75-kWh battery.
This car can go approximately 260 miles on a full charge.
```

Импортирование нескольких классов из модуля

В файл программы можно импортировать столько классов, сколько потребуется. Если вы захотите создать обычный автомобиль и электромобиль в одном файле, потребуется импортировать оба класса, `Car` и `ElectricCar`:

my_cars.py

```
❶ from car import Car, ElectricCar

❷ my_beetle = Car('volkswagen', 'beetle', 2019)
  print(my_beetle.get_descriptive_name())

❸ my_tesla = ElectricCar('tesla', 'roadster', 2019)
  print(my_tesla.get_descriptive_name())
```

Чтобы импортировать несколько классов из модуля, разделите их имена запятыми ❶. После того как необходимые классы будут импортированы, вы можете создать столько экземпляров каждого класса, сколько вам потребуется.

В этом примере создается обычный автомобиль Volkswagen Beetle ❷ и электромобиль Tesla Roadster ❸:

```
2019 Volkswagen Beetle
2019 Tesla Roadster
```

Импортирование всего модуля

Также возможно импортировать весь модуль, а потом обращаться к нужным классам с использованием точечной записи. Этот способ прост, а полученный код легко читается. Так как каждый вызов, создающий экземпляр класса, включает имя модуля, в программе не будет конфликтов с именами, используемыми в текущем файле.

my_cars.py

```
❶ import car

❷ my_beetle = car.Car('volkswagen', 'beetle', 2019)
  print(my_beetle.get_descriptive_name())

❸ my_tesla = car.ElectricCar('tesla', 'roadster', 2019)
  print(my_tesla.get_descriptive_name())
```

В точке ❶ импортируется весь модуль `car`, после чего программа обращается к нужным классам с использованием синтаксиса *имя_модуля.имя_класса*. В точке ❷ снова создается экземпляр Volkswagen Beetle, а в точке ❸ — экземпляр Tesla Roadster.

Импортирование всех классов из модуля

Для импортирования всех классов из модуля используется следующий синтаксис:

```
from имя_модуля import *
```

Использовать этот способ не рекомендуется по двум причинам. Прежде всего, бывает полезно прочитать команды `import` в начале файла и получить четкое представление о том, какие классы используются в программе, а при таком подходе неясно, какие классы из модуля нужны программе. Также возможны конфликты с именами в файле. Если вы случайно импортируете класс с именем, уже присутствующим в файле, в программе могут возникнуть коварные ошибки. Почему я привожу описание этого способа? Хотя использовать его не рекомендуется, скорее всего, вы встретите его в коде других разработчиков.

Итак, если вам нужно импортировать большое количество классов из модуля, лучше импортировать весь модуль и воспользоваться синтаксисом *имя_модуля*`.`*имя_класса*. Хотя вы не видите перечень всех используемых классов в начале файла, по крайней мере ясно видно, где модуль используется в программе. Также предотвращаются потенциальные конфликты имен, которые могут возникнуть при импортировании каждого класса в модуле.

Импортирование модуля в модуль

Иногда классы приходится распределять по нескольким модулям, чтобы избежать чрезмерного разрастания одного файла и хранения несвязанных классов в одном модуле. При хранении классов в нескольких модулях может оказаться, что один класс из одного модуля зависит от класса из другого модуля. В таких случаях необходимый класс можно импортировать в первый модуль.

Допустим, класс `Car` хранится в одном модуле, а классы `ElectricCar` и `Battery` — в другом. Мы создадим новый модуль с именем electric_car.py (он заменит файл electric_car.py, созданный ранее) и скопируем в него только классы `Battery` и `ElectricCar`:

electric_car.py

```
"""Набор классов для представления электромобилей."""

❶ from car import Car

class Battery():
...

class ElectricCar(Car):
...
```

Классу `ElectricCar` необходим доступ к классу-родителю `Car`, поэтому класс `Car` импортируется прямо в модуль в точке ❶. Если вы забудете вставить эту команду,

при попытке создания экземпляра `ElectricCar` произойдет ошибка. Также необходимо обновить модуль `Car`, чтобы он содержал только класс `Car`:

car.py

```
"""Простая модель автомобиля."""

class Car():
...
```

Теперь вы можете импортировать классы из каждого модуля по отдельности и создать ту разновидность машины, которая вам нужна:

my_cars.py

```
❶ from car import Car
  from electric_car import ElectricCar

  my_beetle = Car('volkswagen', 'beetle', 2019)
  print(my_beetle.get_descriptive_name())

  my_tesla = ElectricCar('tesla', 'roadster', 2019)
  print(my_tesla.get_descriptive_name())
```

В точке ❶ класс `Car` импортируется из своего модуля, а класс `ElectricCar` — из своего. После этого создаются экземпляры обоих разновидностей. Вывод показывает, что экземпляры были созданы правильно:

```
2019 Volkswagen Beetle
2019 Tesla Roadster
```

Использование псевдонимов

Как было показано в главе 8, псевдонимы весьма полезны при использовании модулей для организации кода проектов. Псевдонимы также могут использоваться и при импортировании классов.

Для примера возьмем программу, которая должна создать группу экземпляров электрических машин. Многократно вводить (и читать) имя `ElectricCar` будет очень утомительно. Имени `ElectricCar` можно назначить псевдоним в команде `import`:

```
from electric_car import ElectricCar as EC
```

С этого момента вы сможете использовать этот псевдоним каждый раз, когда вам потребуется создать экземпляр `ElectricCar`:

```
my_tesla = EC('tesla', 'roadster', 2019)
```

Выработка рабочего процесса

Как видите, Python предоставляет много возможностей структурирования кода в крупных проектах. Вы должны знать все эти возможности, чтобы найти удачные

способы организации своих проектов, а также лучше понимать код других разработчиков.

На первых порах постарайтесь поддерживать простую структуру своего кода. Попробуйте разместить весь код в одном файле, и только когда все заработает, переместите классы в отдельные модули. Если вам нравится схема взаимодействия между модулями и файлами, попробуйте сохранить классы в модулях в начале работы над проектом. Найдите подход, при котором у вас получается работоспособный код, и двигайтесь дальше.

УПРАЖНЕНИЯ

9.10. Импортирование класса Restaurant: возьмите последнюю версию класса `Restaurant` и сохраните ее в модуле. Создайте отдельный файл, импортирующий класс `Restaurant`. Создайте экземпляр `Restaurant` и вызовите один из методов `Restaurant`, чтобы показать, что команда `import` работает правильно.

9.11. Импортирование класса Admin: начните с версии класса из упражнения 9.8 (с. 186). Сохраните классы `User`, `Privileges` и `Admin` в одном модуле. Создайте отдельный файл, создайте экземпляр `Admin` и вызовите метод `show_privileges()`, чтобы показать, что все работает правильно.

9.12. Множественные модули: сохраните класс User в одном модуле, а классы `Privileges` и `Admin` в другом модуле. В отдельном файле создайте экземпляр `Admin` и вызовите метод `show_privileges()`, чтобы показать, что все работает правильно.

Стандартная библиотека Python

Стандартная библиотека Python представляет собой набор модулей, включаемых в каждую установленную копию Python. Сейчас вы уже примерно понимаете, как работают классы, и можете начать использовать модули, написанные другими программистами. Чтобы использовать любую функцию или класс из стандартной библиотеки, достаточно включить простую команду `import` в начало файла. Для примера рассмотрим модуль `random`, который может пригодиться для моделирования многих реальных ситуаций.

В частности, модуль `random` содержит интересную функцию `randint()`. Эта функция получает два целочисленных аргумента и возвращает случайно выбранное целое число в диапазоне, определяемом этими двумя числами (включительно).

В следующем примере генерируется случайное число в диапазоне от 1 до 6:

```
>>> from random import randint
>>> randint(1, 6)
3
```

Другая полезная функция, `choice()`, получает список или кортеж и возвращает случайно выбранный элемент:

```
>>> from random import choice
>>> players = ['charles', 'martina', 'michael', 'florence', 'eli']
>>> first_up = choice(players)
>>> first_up
'florence'
```

Модуль `random` не должен использоваться при построении приложений, связанных с безопасностью, но его возможностей достаточно для многих интересных и увлекательных проектов.

ПРИМЕЧАНИЕ Модули также можно загружать из внешних источников. Соответствующие примеры встретятся вам в части II, в которой для завершения работы над проектами мы будем использовать внешние модули.

УПРАЖНЕНИЯ

9.13. Кубики: создайте класс `Die` с одним атрибутом `sides`, который имеет значение по умолчанию 6. Напишите метод `roll_die()` для вывода случайного числа от 1 до количества граней на кубике. Создайте экземпляр, представляющий 6-гранный кубик, и смоделируйте 10 бросков.

Создайте экземпляры, представляющие 10- и 20-гранный кубик. Смоделируйте 10 бросков каждого кубика.

9.14. Лотерея: создайте список или кортеж, содержащий серию из 10 чисел и 5 букв. Случайным образом выберите 4 числа или буквы из списка. Выведите сообщение о том, что билет, содержащий эту комбинацию из четырех цифр или букв, является выигрышным.

9.15. Анализ лотереи: напишите цикл, который проверяет, насколько сложно выиграть в смоделированной вами лотерее. Создайте список или кортеж с именем `my_ticket`. Напишите цикл, который продолжает генерировать комбинации до тех пор, пока не выпадет выигрышная комбинация. Выведите сообщение с информацией о том, сколько выполнений цикла понадобилось для получения выигрышной комбинации.

9.16. Модуль недели: для знакомства со стандартной библиотекой Python отлично подойдет сайт Python Module of the Week. Откройте сайт http://pymotw.com/ и просмотрите оглавление. Найдите модуль, который покажется вам интересным; прочитайте его описание или изучите документацию по модулю `random`.

Оформление классов

В стилевом оформлении классов есть несколько моментов, о которых стоит упомянуть отдельно, особенно с усложнением ваших программ.

Имена классов должны записываться в «верблюжьем» регистре: первая буква каждого слова записывается в верхнем регистре, слова не разделяются пробелами. Имена экземпляров и модулей записываются в нижнем регистре с разделением слов символами подчеркивания.

Каждый класс должен иметь строку документации, следующую сразу же за определением класса. Строка документации должна содержать краткое описание того, что

делает класс, и в ней должны соблюдаться те же соглашения по форматированию, которые вы использовали при написании строк документации в функциях. Каждый модуль также должен содержать строку документации с описанием возможных применений классов в модуле.

Пустые строки могут использоваться для структурирования кода, но злоупотреблять ими не стоит. В классах можно разделять методы одной пустой строкой, а в модулях для разделения классов можно использовать две пустые строки.

Если вам потребуется импортировать модуль из стандартной библиотеки и модуль из библиотеки, написанной вами, начните с команды `import` для модуля стандартной библиотеки. Затем добавьте пустую строку и команду `import` для модуля, написанного вами. В программах с несколькими командами `import` выполнение этого соглашения поможет понять, откуда берутся разные модули, использованные в программе.

Итоги

В этой главе вы узнали, как написать собственные классы. Вы научились хранить информацию в классе с использованием атрибутов и наделять свои классы нужным поведением. Вы узнали, как написать методы `__init__()` для создания экземпляров ваших классов с нужными значениями атрибутов и как изменять атрибуты экземпляров напрямую и через методы. Также было показано, что наследование может упростить создание логически связанных классов и что экземпляры одного класса могут использоваться как атрибуты другого класса для упрощения кода классов.

Вы узнали, что хранение классов в модулях и импортирование необходимых классов в файлы, где они будут использоваться, улучшает организацию проектов. Вы познакомились со стандартной библиотекой Python и рассмотрели пример, основанный на модуле `random`. Наконец, вы научились оформлять свои классы с использованием общепринятых соглашений Python.

В главе 10 вы научитесь работать с файлами и сохранять результаты работы, выполненной в программе. Также будут рассмотрены *исключения* — экземпляры специального класса Python, предназначенного для передачи информации о возникающих ошибках.

10 Файлы и исключения

Вы уже овладели основными навыками, необходимыми для создания хорошо структурированных и удобных в использовании программ; теперь пора подумать о том, как сделать ваши программы еще более удобными и полезными. В этой главе вы научитесь работать с файлами, чтобы ваши программы могли быстро анализировать большие объемы данных.

Вы научитесь обрабатывать ошибки, чтобы возникновение аномальных ситуаций не приводило к аварийному завершению ваших программ. Мы рассмотрим *исключения* — специальные объекты, которые создаются для управления ошибками, возникающими во время выполнения программ Python. Также будет описан модуль `json`, позволяющий сохранять пользовательские данные, чтобы они не терялись по завершении работы программы.

Работа с файлами и сохранение данных упрощают использование ваших программ. Пользователь сам выбирает, какие данные и когда нужно вводить. Он может запустить вашу программу, выполнить некоторую работу, потом закрыть программу и позднее продолжить работу с того момента, на котором он прервался. Умение обрабатывать исключения поможет справиться с такими ситуациями, как отсутствие нужных файлов, а также другими проблемами, приводящими к сбою программ. Обработка исключений повысит устойчивость ваших программ при работе с некорректными данными — появившимися как из-за случайных ошибок, так и в результате злонамеренных попыток взлома ваших программ. Материал, представленный в этой главе, сделает ваши программы более практичными, удобными и надежными.

Чтение из файла

Гигантские объемы данных доступны в текстовых файлах. В них могут храниться погодные данные, социально-экономическая информация, литературные произведения и многое другое. Чтение из файла особенно актуально для приложений, предназначенных для анализа данных, но оно также может пригодиться в любой ситуации, требующей анализа или изменения информации, хранящейся в файле. Например, программа может читать содержимое текстового файла и переписывать его с форматированием, рассчитанным на отображение информации в браузере.

Работа с информацией в текстовом файле начинается с чтения данных в память. Вы можете прочитать все содержимое файла или же читать данные по строкам.

Чтение всего файла

Для начала нам понадобится файл с несколькими строками текста. Пусть это будет файл с числом «пи» с точностью до 30 знаков, по 10 знаков на строку:

pi_digits.txt

```
3.1415926535
  8979323846
  2643383279
```

Чтобы опробовать эти примеры, либо введите данные в редакторе и сохраните файл с именем pi_digits.txt, либо загрузите файл из ресурсов книги на странице https://www.nostarch.com/pythoncrashcourse2e/. Сохраните файл в каталоге, в котором будут храниться программы этой главы.

Следующая программа открывает этот файл, читает его и выводит содержимое на экран:

file_reader.py

```
with open('pi_digits.txt') as file_object:
    contents = file_object.read()
print(contents)
```

В первой строке этой программы многое заслуживает вашего внимания. Начнем с функции `open()`. Чтобы выполнить любые операции с файлом — даже просто вывести его содержимое, — сначала необходимо *открыть* файл. Функция `open()` получает один аргумент: имя открываемого файла. Python ищет файл с указанным именем в каталоге, в котором находится файл текущей программы. В данном примере выполняется программа file_reader.py, поэтому Python ищет файл pi_digits.txt в каталоге, в котором хранится file_reader.py. Функция `open()` возвращает объект, представляющий файл. В данном случае `open('pi_digits.txt')` возвращает объект, представляющий файл pi_digits.txt. Python сохраняет этот объект в переменной `file_object`, с которой мы будем работать позднее в программе.

Конструкция с ключевым словом `with` закрывает файл после того, как надобность в нем отпадет. Обратите внимание: в этой программе есть вызов `open()`, но нет вызова `close()`. Файлы можно открывать и закрывать явными вызовами `open()` и `close()`; но если из-за ошибки в программе команда `close()` останется невыполненной, то файл не будет закрыт. На первый взгляд это не страшно, но некорректное закрытие файлов может привести к потере или порче данных. А если функция `close()` будет вызвана слишком рано, программа попытается работать с *закрытым* (то есть недоступным) файлом, что приведет к новым ошибкам. Не всегда можно заранее определить, когда нужно закрывать файл, но с приведенной конструкцией Python сделает это за вас. Вам остается лишь открыть файл и работать с ним так,

как требуется, надеясь на то, что Python закроет его автоматически при завершении блока `with`.

После того как в программе появится объект, представляющий файл pi_digits.txt, во второй строке программы используется метод `read()`, который читает все содержимое файла и сохраняет его в одной длинной строке в переменной `contents`. При выводе значения `contents` на экране появляется все содержимое файла :

```
3.1415926535
  8979323846
  2643383279
```

Единственное различие между выводом и исходным файлом — лишняя пустая строка в конце вывода. Откуда она взялась? Метод `read()` возвращает ее при чтении, если достигнут конец файла. Если вы хотите удалить лишнюю пустую строку, включите вызов `rstrip()` в вызов `print()`:

```
with open('pi_digits.txt') as file_object:
    contents = file_object.read()
    print(contents.rstrip())
```

Напомним, что метод `rstrip()` удаляет все пропуски в конце строки. Теперь вывод точно соответствует содержимому исходного файла:

```
3.1415926535
  8979323846
  2643383279
```

Пути к файлам

Если передать функции `open()` простое имя файла, такое как pi_digits.txt, Python ищет файл в том каталоге, в котором находится файл, выполняемый в настоящий момент (то есть файл программы .py).

В некоторых случаях (в зависимости от того, как организованы ваши рабочие файлы) открываемый файл может и не находиться в одном каталоге с файлом программы. Например, файл программы может находиться в каталоге python_work; в каталоге python_work создается другой каталог с именем text_files для текстовых файлов, с которыми работает программа. И хотя папка text_files находится в python_work, простая передача `open()` имени файла из text_files не подойдет, потому что Python произведет поиск файла в python_work и на этом остановится; поиск не будет продолжен во вложенном каталоге text_files. Чтобы открыть файлы из каталога, отличного от того, в котором хранится файл программы, необходимо указать *путь* — то есть приказать Python искать файлы в конкретном месте файловой системы.

Так как каталог text_files находится в python_work, для открытия файла из text_files можно воспользоваться *относительным* путем. Относительный путь приказывает Python искать файлы в каталоге, который задается *относительно* каталога, в котором находится текущий файл программы. Например, это может выглядеть так:

```
with open('text_files/имя_файла.txt') as file_object:
```

Эта строка означает, что файл .txt следует искать в каталоге text_files; она предполагает, что каталог text_files находится в python_work (так оно и есть).

ПРИМЕЧАНИЕ В системе Windows при отображении путей файлов используется обратный слеш. Но в своем коде вы можете использовать и обычный слеш.

Также можно точно определить местонахождение файла в вашей системе независимо от того, где хранится выполняемая программа. Такие пути называются *абсолютными* и используются в том случае, если относительный путь не работает. Например, если каталог text_files находится не в python_work, а в другом каталоге (скажем, в каталоге с именем other_files), то передать `open()` путь `'text_files/filename.txt'` не получится, потому что Python будет искать указанный каталог только внутри python_work. Чтобы объяснить Python, где следует искать файл, необходимо записать полный путь.

Абсолютные пути обычно длиннее относительных, поэтому их лучше сохранять в переменных, которые затем передаются `open()`:

```
file_path = '/home/ehmatthes/other_files/text_files/имя_файла.txt'
with open(file_path) as file_object:
```

С абсолютными путями вы сможете читать файлы из любого каталога вашей системы. Пока будет проще хранить файлы в одном каталоге с файлами программ или в каталогах, вложенных в каталог с файлами программ (таких, как text_files из рассмотренного примера).

ПРИМЕЧАНИЕ Если вы попытаетесь использовать символ \ в пути, произойдет ошибка, потому что этот символ используется для экранирования служебных символов в строках. Например, в пути `'C:\path\to\file.txt'` последовательность `\t` интерпретируется как символ табуляции. Если вам потребуется включить литерал \ в строку, экранируйте каждое вхождение такого символа: `'C:\\path\\to\\file.txt'`.

Чтение по строкам

В процессе чтения файла часто бывает нужно обработать каждую строку. Возможно, вы ищете некую информацию в файле или собираетесь каким-то образом изменить текст — например, при чтении файла с метеорологическими данными вы обрабатываете каждую строку, в которой в описании погоды встречается слово «солнечно». Или, допустим, в новостях вы ищете каждую строку с тегом заголовка и заменяете ее специальными элементами форматирования.

Для последовательной обработки каждой строки в файле можно воспользоваться циклом `for`:

file_reader.py

```
❶ filename = 'pi_digits.txt'
```

```
❷ with open(filename) as file_object:
❸     for line in file_object:
          print(line)
```

В точке ❶ имя файла, из которого читается информация, сохраняется в переменной `filename`. Это стандартный прием при работе с файлами: так как переменная `filename` не представляет конкретный файл (это всего лишь строка, которая сообщает Python, где найти файл), вы сможете легко заменить `'pi_digits.txt'` именем другого файла, с которым вы собираетесь работать. После вызова `open()` объект, представляющий файл и его содержимое, сохраняется в переменной `file_object` ❷. Мы снова используем синтаксис `with`, чтобы поручить Python открывать и закрывать файл в нужный момент. Для просмотра содержимого все строки файла перебираются в цикле `for` по объекту файла ❸.

На этот раз пустых строк оказывается еще больше:

```
3.1415926535

  8979323846

  2643383279
```

Пустые строки появляются из-за того, что каждая строка в текстовом файле завершается невидимым символом новой строки. Команда `print` добавляет свой символ новой строки при каждом вызове, поэтому в результате каждая строка завершается *двумя* символами новой строки: один прочитан из файла, а другой добавлен командой `print`. Вызов `rstrip()` в команде `print` удаляет лишние пустые строки:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    for line in file_object:
        print(line.rstrip())
```

Теперь вывод снова соответствует содержимому файла:

```
3.1415926535
  8979323846
  2643383279
```

Создание списка строк по содержимому файла

При использовании `with` объект файла, возвращаемый вызовом `open()`, доступен только в пределах содержащего его блока `with`. Если вы хотите, чтобы содержимое файла оставалось доступным за пределами блока `with`, сохраните строки файла в списке внутри блока и в дальнейшем работайте с полученным списком. Одни части файла можно обработать немедленно, а другие отложить для обработки в будущем.

В следующем примере строки pi_digits.txt сохраняются в списке в блоке `with`, после чего выводятся за пределами этого блока:

```
  filename = 'pi_digits.txt'

  with open(filename) as file_object:
❶     lines = file_object.readlines()

❷ for line in lines:
      print(line.rstrip())
```

В точке ❶ метод `readlines()` последовательно читает каждую строку из файла и сохраняет ее в списке. Список сохраняется в переменной `lines`, с которой можно продолжить работу после завершения блока `with`. В точке ❷ в простом цикле `for` выводятся все элементы списка `lines`. Так как каждый элемент `lines` соответствует ровно одной строке файла, вывод точно соответствует его содержимому.

Работа с содержимым файла

После того как файл будет прочитан в память, вы сможете обрабатывать данные так, как считаете нужным. Для начала попробуем построить одну строку со всеми цифрами из файла без промежуточных пропусков:

pi_string.py

```
  filename = 'pi_digits.txt'

  with open(filename) as file_object:
      lines = file_object.readlines()

❶ pi_string = ''
❷ for line in lines:
      pi_string += line.rstrip()

❸ print(pi_string)
  print(len(pi_string))
```

Сначала программа открывает файл и сохраняет каждую строку цифр в списке — точно так же, как это делалось в предыдущем примере. В точке ❶ создается переменная `pi_string` для хранения цифр числа «пи». Далее следует цикл, который добавляет к `pi_string` каждую серию цифр, из которой удаляется символ новой строки ❷. В точке ❸ программа выводит строку и ее длину:

```
3.1415926535  8979323846  2643383279
36
```

Переменная `pi_string` содержит пропуски, которые присутствовали в начале каждой строки цифр. Чтобы удалить их, достаточно использовать `strip()` вместо `rstrip()`:

```
...
for line in lines:
    pi_string += line.strip()

print(pi_string)
print(len(pi_string))
```

В итоге мы получаем строку, содержащую значение «пи» с точностью до 30 знаков. Длина строки равна 32 символам, потому что в нее также включается начальная цифра 3 и точка:

```
3.14159265358979323846264338327
32
```

ПРИМЕЧАНИЕ Читая данные из текстового файла, Python интерпретирует весь текст в файле как строку. Если вы читаете из текстового файла число и хотите работать с ним в числовом контексте, преобразуйте его в целое число функцией `int()` или в вещественное число функцией `float()`.

Большие файлы: миллион цифр

До настоящего момента мы ограничивались анализом текстового файла, который состоял всего из трех строк, но код этих примеров будет работать и с много большими файлами. Начиная с текстового файла, содержащего значение «пи» до 1 000 000 знаков (вместо 30), вы сможете создать одну строку, которая содержит все эти цифры. Изменять программу вообще не придется — достаточно передать ей другой файл. Также мы ограничимся выводом первых 50 цифр, чтобы не пришлось ждать, пока в терминале не прокрутится миллион знаков:

pi_string.py

```
filename = 'pi_million_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()
pi_string = ''
for line in lines:
    pi_string += line.strip()

print(f"{pi_string[:52]}...")
print(len(pi_string))
```

Из выходных данных видно, что строка действительно содержит значение «пи» с точностью до 1 000 000 знаков:

```
3.14159265358979323846264338327950288419716939937510...
1000002
```

Python не устанавливает никаких ограничений на длину данных, с которыми вы можете работать. Она ограничивается разве что объемом памяти вашей системы.

ПРИМЕЧАНИЕ Для запуска этой программы (и многих других примеров, приведенных ниже) необходимо загрузить ресурсы по адресу https://www.nostarch.com/pythoncrashcourse2e/.

Проверка дня рождения

Меня всегда интересовало, не встречается ли мой день рождения среди цифр числа «пи»? Воспользуемся только что созданной программой для проверки того, входит ли запись дня рождения пользователя в первый миллион цифр. Для этого можно записать день рождения в виде строки из цифр и посмотреть, присутствует ли эта строка в `pi_string`:

```
  ...
  for line in lines:
      pi_string += line.strip()

❶ birthday = input("Enter your birthday, in the form mmddyy: ")
❷ if birthday in pi_string:
      print("Your birthday appears in the first million digits of pi!")
  else:
      print("Your birthday does not appear in the first million digits of pi.")
```

В точке ❶ программа запрашивает день рождения пользователя, а затем в точке ❷ проверяет вхождение этой строки в `pi_string`. Пробуем:

```
Enter your birthdate, in the form mmddyy: 120372
Your birthday appears in the first million digits of pi!
```

Оказывается, мой день рождения встречается среди цифр «пи»! После того, как данные будут прочитаны из файла, вы сможете делать с ними все, что сочтете нужным.

УПРАЖНЕНИЯ

10.1. Изучение Python: откройте пустой файл в текстовом редакторе и напишите несколько строк текста о возможностях Python. Каждая строка должна начинаться с фразы «In Python you can...». Сохраните файл под именем learning_python.txt в каталоге, использованном для примеров этой главы. Напишите программу, которая читает файл и выводит текст три раза: с чтением всего файла, с перебором строк объекта файла и с сохранением строк в списке с последующим выводом списка вне блока `with`.

10.2. Изучение C: метод `replace()` может использоваться для замены любого слова в строке другим словом. В следующем примере слово `'dog'` заменяется словом `'cat'`:

```
>>> message = "I really like dogs."
>>> message.replace('dog', 'cat')
'I really like cats.'
```

Прочитайте каждую строку из только что созданного файла learning_python.txt и замените слово Python названием другого языка, например C. Выведите каждую измененную строку на экран.

Запись в файл

Один из простейших способов сохранения данных — запись в файл. Текст, записанный в файл, останется доступным и после закрытия терминала с выводом вашей программы. Вы сможете проанализировать результаты после завершения программы или передать свои файлы другим. Вы также сможете написать программы, которые снова читают сохраненный текст в память и работают с ним.

Запись в пустой файл

Чтобы записать текст в файл, необходимо вызвать `open()` со вторым аргументом, который сообщает Python, что вы собираетесь записывать данные в файл. Чтобы увидеть, как это делается, напишем простое сообщение и сохраним его в файле (вместо того, чтобы просто вывести на экран):

write_message.py

```
  filename = 'programming.txt'

❶ with open(filename, 'w') as file_object:
❷     file_object.write("I love programming.")
```

При вызове `open()` в этом примере передаются два аргумента ❶. Первый аргумент, как и прежде, содержит имя открываемого файла. Второй аргумент `'w'` сообщает Python, что файл должен быть открыт в режиме *записи*. Файлы можно открывать в режиме *чтения* (`'r'`), *записи* (`'w'`), *присоединения* (`'a'`) или в режиме, допускающем *как чтение, так и запись* в файл (`'r+'`). Если аргумент режима не указан, Python по умолчанию открывает файл в режиме только для чтения.

Если файл, открываемый для записи, еще не существует, функция `open()` автоматически создает его. Будьте внимательны, открывая файл в режиме записи (`'w'`): если файл существует, то Python уничтожит его данные перед возвращением объекта файла.

В точке ❷ метод `write()` используется с объектом файла для записи строки в файл. Программа не выводит данные на терминал, но открыв файл programming.txt, вы увидите в нем одну строку:

programming.txt

```
I love programming.
```

Этот файл ничем не отличается от любого другого текстового файла на вашем компьютере. Его можно открыть, записать в него новый текст, скопировать/вставить текст и т. д.

ПРИМЕЧАНИЕ Python может записывать в текстовые файлы только строковые данные. Если вы захотите сохранить в текстовом файле числовую информацию, данные придется предварительно преобразовать в строки функцией `str()`.

Многострочная запись

Функция `write()` не добавляет символы новой строки в записываемый текст. А это означает, что если вы записываете сразу несколько строк без включения символов новой строки, полученный файл может выглядеть не так, как вы рассчитывали:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
    file_object.write("I love creating new games.")
```

Открыв файл programming.txt, вы увидите, что две строки «склеились»:

```
I love programming.I love creating new games.
```

Если включить символы новой строки в команды `write()`, текст будет состоять из двух строк:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.\n")
    file_object.write("I love creating new games.\n")
```

Результат выглядит так:

```
I love programming.
I love creating new games.
```

Для форматирования вывода также можно использовать пробелы, символы табуляции и пустые строки по аналогии с тем, как это делалось с выводом на терминал.

Присоединение данных к файлу

Если вы хотите добавить в файл новые данные вместо того, чтобы перезаписывать существующее содержимое, откройте файл в режиме присоединения. В этом случае Python не уничтожает содержимое файла перед возвращением объекта файла. Все строки, выводимые в файл, будут добавлены в конец файла. Если файл еще не существует, то Python автоматически создаст пустой файл.

Изменим программу write_message.py и дополним существующий файл programming.txt новыми данными:

write_message.py

```
  filename = 'programming.txt'

❶ with open(filename, 'a') as file_object:
❷     file_object.write("I also love finding meaning in large datasets.\n")
      file_object.write("I love creating apps that can run in a browser.\n")
```

В точке ❶ аргумент `'a'` используется для открытия файла в режиме присоединения (вместо перезаписи существующего файла). В точке ❷ записываются две новые строки, которые добавляются к содержимому programming.txt:

programming.txt

```
I love programming.
I love creating new games.
I also love finding meaning in large datasets.
I love creating apps that can run in a browser.
```

В результате к исходному содержимому файла добавляется новый текст.

УПРАЖНЕНИЯ

10.3. Гость: напишите программу, которая запрашивает у пользователя его имя. Введенный ответ сохраняется в файле с именем guest.txt.

10.4. Гостевая книга: напишите цикл `while`, который запрашивает у пользователей имена. При вводе каждого имени выведите на экран приветствие и добавьте строку с сообщением в файл с именем guest_book.txt. Проследите за тем, чтобы каждое сообщение размещалось в отдельной строке файла.

10.5. Опрос: напишите цикл `while`, в котором программа спрашивает у пользователя, почему ему нравится программировать. Каждый раз, когда пользователь вводит очередную причину, сохраните текст его ответа в файле.

Исключения

Для управления ошибками, возникающими в ходе выполнения программы, в Python используются специальные объекты, называемые *исключениями*. Если при возникновении ошибки Python не знает, что делать дальше, создается объект исключения. Если в программу включен код обработки исключения, то выполнение программы продолжится, а если нет — программа останавливается и выводит *трассировку* с отчетом об исключении.

Исключения обрабатываются в блоках `try-except`. Блок `try-except` приказывает Python выполнить некоторые действия, но при этом также сообщает, что делать при возникновении исключения. С блоками `try-except` ваши программы будут работать даже в том случае, если что-то пошло не так. Вместо невразумительной трассировки выводится понятное сообщение об ошибке, которое вы определяете в программе.

Обработка исключения ZeroDivisionError

Рассмотрим простую ошибку, при которой Python инициирует исключение. Конечно, вы знаете, что деление на ноль невозможно, но мы все же прикажем Python выполнить эту операцию:

division_calculator.py

```
print(5/0)
```

Конечно, из этого ничего не выйдет, поэтому на экран выводятся данные трассировки:

```
  Traceback (most recent call last):
    File "division.py", line 1, in <module>
      print(5/0)
❶ ZeroDivisionError: division by zero
```

Ошибка, упоминаемая в трассировке ❶, — `ZeroDivisionError` — является объектом исключения. Такие объекты создаются в том случае, если Python не может выполнить ваши распоряжения. Обычно в таких случаях Python прерывает выполнение программы и сообщает тип обнаруженного исключения. Эта информация может использоваться в программе; по сути, вы сообщаете Python, как следует поступить при возникновении исключения данного типа. В таком случае ваша программа будет подготовлена к его появлению.

Блоки try-except

Если вы предполагаете, что в программе может произойти ошибка, напишите блок `try-except` для обработки возникающего исключения. Такой блок приказывает Python выполнить некоторый код, а также сообщает, что нужно делать, если при его выполнении произойдет исключение конкретного типа.

Вот как выглядит блок `try-except` для обработки исключений `ZeroDivisionError`:

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Команда `print(5/0)`, порождающая ошибку, находится в блоке `try`. Если код в блоке `try` выполнен успешно, то Python пропускает блок `except`. Если код в блоке `try` порождает ошибку, то Python ищет блок `except` с соответствующей ошибкой и выпускает код в этом блоке.

В этом примере код блока `try` порождает ошибку `ZeroDivisionError`, поэтому Python ищет блок `except` с описанием того, как следует действовать в такой ситуации. При выполнении кода этого блока пользователь видит понятное сообщение об ошибке вместо данных трассировки:

```
You can't divide by zero!
```

Если за кодом `try-except` следует другой код, то выполнение программы продолжится, потому что мы объяснили Python, как обрабатывать эту ошибку. В следующем примере обработка ошибки позволяет программе продолжить выполнение.

Использование исключений для предотвращения аварийного завершения программы

Правильная обработка ошибок особенно важна в том случае, если программа должна продолжить работу после возникновения ошибки. Такая ситуация часто встречается в программах, запрашивающих данные у пользователя. Если программа правильно среагировала на некорректный ввод, она может запросить новые данные после сбоя.

Создадим простой калькулятор, который выполняет только операцию деления:

division_calculator.py

```
  print("Give me two numbers, and I'll divide them.")
  print("Enter 'q' to quit.")

  while True:
❶     first_number = input("\nFirst number: ")
      if first_number == 'q':
          break
❷     second_number = input("Second number: ")
      if second_number == 'q':
          break
❸     answer = int(first_number) / int(second_number)
      print(answer)
```

Программа запрашивает у пользователя первое число, `first_number` ❶, а затем, если пользователь не ввел `q` для завершения работы, — второе число, `second_number` ❷. Далее одно число делится на другое для получения результата `answer` ❸. Программа никак не пытается обрабатывать ошибки, так что попытка деления на ноль приводит к ее аварийному завершению:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
Traceback (most recent call last):
  File "division.py", line 9, in <module>
    answer = int(first_number) / int(second_number)
ZeroDivisionError: division by zero
```

Конечно, аварийное завершение — это плохо, но еще хуже, что пользователь увидит данные трассировки. Неопытного пользователя они собьют с толку, а при сознательной попытке взлома злоумышленник сможет получить из них больше информации, чем вам хотелось бы. Например, он узнает имя файла программы и увидит некорректно работающую часть кода. На основании этой информации опытный хакер иногда может определить, какие атаки следует применять против вашего кода.

Блок else

Для повышения устойчивости программы к ошибкам можно заключить строку, выдающую ошибки, в блок `try-except`. Ошибка происходит в строке, выполняющей деление; следовательно, именно эту строку следует заключить в блок `try-except`. Данный пример также включает блок `else`. Любой код, зависящий от успешного выполнения блока `try`, размещается в блоке `else`:

```
  ...
  while True:
  ...
      if second_number == 'q':
          break
❶     try:
          answer = int(first_number) / int(second_number)
❷     except ZeroDivisionError:
          print("You can't divide by 0!")
❸     else:
          print(answer)
```

Программа пытается выполнить операцию деления в блоке `try` ❶, который включает только код, способный породить ошибку. Любой код, зависящий от успешного выполнения блока `try`, добавляется в блок `else`. В данном случае, если операция деления выполняется успешно, блок `else` используется для вывода результата ❸.

Блок `except` сообщает Python, как следует поступать при возникновении ошибки `ZeroDivisionError` ❷. Если при выполнении команды из блока `try` происходит ошибка, связанная с делением на 0, программа выводит понятное сообщение, которое объясняет пользователю, как избежать подобных ошибок. Выполнение программы продолжается, и пользователь не сталкивается с трассировкой:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
You can't divide by 0!
First number: 5
Second number: 2
2.5

First number: q
```

Блок `try-except-else` работает так: Python пытается выполнить код в блоке `try`. В блоках `try` следует размещать только тот код, при выполнении которого может возникнуть исключение. Иногда некоторый код должен выполняться только в том случае, если выполнение `try` прошло успешно; такой код размещается в блоке `else`. Блок `except` сообщает Python, что делать, если при выполнении кода `try` произошло определенное исключение.

Заранее определяя вероятные источники ошибок, вы повышаете надежность своих программ, которые продолжают работать даже при вводе некорректных данных или при недоступности ресурсов. Ваш код оказывается защищенным от случайных ошибок пользователей и сознательных атак.

Обработка исключения FileNotFoundError

Одна из стандартных проблем при работе с файлами — отсутствие необходимых файлов. Тот файл, который вам нужен, может находиться в другом месте, в имени файла может быть допущена ошибка или файл может вообще не существовать. Все эти ситуации достаточно прямолинейно обрабатываются в блоках `try-except`.

Попробуем прочитать данные из несуществующего файла. Следующая программа пытается прочитать содержимое файла с текстом «Алисы в Стране чудес», но я не сохранил файл alice.txt в одном каталоге с файлом alice.py:

alice.py

```
filename = 'alice.txt'

with open(filename, encoding='utf-8') as f:
    contents = f.read()
```

В программе видны два изменения. Во-первых, объект файла представляется переменной с именем `f` — это общепринятое соглашение. Во-вторых, в программе используется аргумент `encoding`. Он необходим в тех случаях, когда кодировка вашей системы по умолчанию не совпадает с кодировкой читаемого файла.

Прочитать данные из несуществующего файла нельзя, поэтому Python выдает исключение:

```
Traceback (most recent call last):
  File "alice.py", line 3, in <module>
    with open(filename, encoding='utf-8') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

В последней строке трассировки упоминается `FileNotFoundError`: это исключение выдается в том случае, если Python не может найти открываемый файл. В данном примере функция `open()` порождает ошибку, и чтобы обработать ее, блок `try` начинается перед строкой с вызовом `open()`:

```
filename = 'alice.txt'

try:
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    print(f"Sorry, the file {filename} does not exist.")
```

В этом примере код блока `try` выдает исключение `FileNotFoundError`, поэтому Python ищет блок `except` для этой ошибки. Затем выполняется код этого

блока, в результате чего вместо трассировки выдается доступное сообщение об ошибке:

```
Sorry, the file alice.txt does not exist.
```

Если файл не существует, программе больше делать нечего, поэтому код обработки ошибок почти ничего не добавляет в эту программу. Доведем до ума этот пример и посмотрим, как обработка исключений помогает при работе с несколькими файлами.

Анализ текста

Программа может анализировать текстовые файлы, содержащие целые книги. Многие классические произведения, ставшие общественным достоянием, доступны в виде простых текстовых файлов. Тексты, использованные в этом разделе, взяты с сайта проекта «Гутенберг» (http://gutenberg.org/). На этом сайте хранится подборка литературных произведений, не защищенных авторским правом; это превосходный ресурс для разработчиков, которые собираются работать с литературными текстами в своих программных проектах.

Прочитаем текст «Алисы в Стране чудес» и попробуем подсчитать количество слов в тексте. Мы воспользуемся методом `split()`, предназначенным для построения списка слов на основе строки. Вот как метод `split()` работает со строкой, содержащей только название книги:

```
>>> title = "Alice in Wonderland"
>>> title.split()
['Alice', 'in', 'Wonderland']
```

Метод `split()` разделяет строку на части по всем позициям, в которых обнаружит пробел, и сохраняет все части строки в элементах списка. В результате создается список слов, входящих в строку (впрочем, вместе с некоторыми словами могут храниться знаки препинания.) Для подсчета слов в книге мы вызовем `split()` для всего текста, а затем подсчитаем элементы списка, чтобы получить примерное количество слов в тексте:

```
  filename = 'alice.txt'

  try:
      with open(filename, encoding='utf-8') as f:
          contents = f.read()
  except FileNotFoundError:
      print(f"Sorry, the file {filename} does not exist.")
  else:
      # Подсчет приблизительного количества строк в файле.
❶     words = contents.split()
❷     num_words = len(words)
❸     print(f"The file {filename} has about {num_words} words.")
```

Затем я переместил файл alice.txt в правильный каталог, чтобы код в блоке `try` был выполнен без ошибок. В точке ❶ программа загружает текст в переменную

`contents`, которая теперь содержит весь текст в виде одной длинной строки и использует метод `split()` для получения списка всех слов в книге. Запрашивая длину этого списка при помощи функции `len()`, мы получаем неплохое приближенное значение количества слов в исходной строке ❷. В точке ❸ выводится сообщение с количеством слов, найденных в файле. Этот код помещен в блок `else`, потому что он должен выводиться только в случае успешного выполнения блока `try`. Выходные данные программы сообщают, сколько слов содержит файл alice.txt:

```
The file alice.txt has about 29465 words.
```

Количество слов немного завышено, потому что в нем учитывается дополнительная информация, включенная в текстовый файл издателем, но в целом оно довольно точно оценивает объем «Алисы в Стране чудес».

Работа с несколькими файлами

Добавим еще несколько файлов с книгами для анализа. Но для начала переместим основной код программы в функцию с именем `count_words()`. Это упростит проведение анализа для нескольких книг:

word_count.py

```
  def count_words(filename):
❶     """Подсчет приблизительного количества строк в файле."""
      try:
          with open(filename,encoding='utf-8') as f:
              contents = f.read()
      except FileNotFoundError:
          print(f"Sorry, the file {filename} does not exist.")
      else:
          words = contents.split()
          num_words = len(words)
          print(f"The file {filename} has about {num_words} words.")

  filename = 'alice.txt'
  count_words(filename)
```

Большая часть кода не изменилась. Мы просто снабдили код отступом и переместили его в тело `count_words()`. При внесении изменений в программу желательно обновлять комментарии, поэтому мы преобразовали комментарий в строку документации и слегка переформулировали его ❶.

Теперь мы можем написать простой цикл для подсчета слов в любом тексте, который нужно проанализировать. Для этого имена анализируемых файлов сохраняются в списке, после чего для каждого файла в списке вызывается функция `count_words()`. Мы попробуем подсчитать слова в «Алисе в Стране чудес», «Сиддхартхе», «Моби Дике» и «Маленьких женщинах» — все эти книги распространяются в свободном доступе. Я намеренно не стал копировать файл siddhartha.txt в каталог с программой word_count.py, чтобы выяснить, насколько хорошо наша программа справляется с отсутствием файла:

```
def count_words(filename):
...

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt',
             'little_women.txt']
for filename in filenames:
    count_words(filename)
```

Отсутствие файла siddhartha.txt не влияет на выполнение программы:

```
The file alice.txt has about 29465 words.
Sorry, the file siddhartha.txt does not exist.
The file moby_dick.txt has about 215830 words.
The file little_women.txt has about 189079 words.
```

Использование блока `try-except` в данном примере предоставляет два важных преимущества: программа ограждает пользователя от вывода трассировки и продолжает выполнение, анализируя тексты, которые ей удается найти. Если бы в программе не перехватывалось исключение `FileNotFoundError`, инициированное из-за отсутствия siddhartha.txt, то пользователь увидел бы полную трассировку, а работа программы прервалась бы после попытки подсчитать слова в тексте «Сиддхартхи»; до анализа «Моби Дика» или «Маленьких женщин» дело не дошло бы.

Ошибки без уведомления пользователя

В предыдущем примере мы сообщили пользователю о том, что один из файлов оказался недоступен. Тем не менее вы не обязаны сообщать о каждом обнаруженном исключении. Иногда при возникновении исключения программа должна просто проигнорировать сбой и продолжать работу, словно ничего не произошло. Для этого блок `try` пишется так же, как обычно, но в блоке `except` вы явно приказываете Python не предпринимать никаких особых действий в случае ошибки. В языке Python существует команда `pass`, с которой блок ничего не делает:

```
def count_words(filename):
    """Подсчет приблизительного количества строк в файле."""
    try:
        ...
    except FileNotFoundError:
❶       pass
    else:
        ...

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    count_words(filename)
```

Единственное отличие этого листинга от предыдущего — команда `pass` в точке ❶. Теперь при возникновении ошибки `FileNotFoundError` выполняется код в блоке `except`, но при этом ничего не происходит. Программа не выдает данных трассировки и вообще никаких результатов, указывающих на возникновение ошибки.

Пользователи получают данные о количестве слов во всех существующих файлах, однако ничто не сообщает о том, что какой-то файл не был найден:

```
The file alice.txt has about 29465 words.
The file moby_dick.txt has about 215830 words.
The file little_women.txt has about 189079 words.
```

Команда `pass` также может служить временным заполнителем. Она напоминает, что в этот конкретный момент выполнения вашей программы вы решили ничего не предпринимать, хотя, возможно, решите сделать что-то позднее. Например, эта программа может записать все имена отсутствующих файлов в файл с именем missing_files.txt. Пользователи этот файл не увидят, но создатель программы сможет прочитать его и разобраться с отсутствующими текстами.

О каких ошибках нужно сообщать?

Как определить, в каком случае следует сообщить об ошибке пользователю, а когда можно просто проигнорировать ее незаметно для пользователя? Если пользователь знает, с какими текстами должна работать программа, вероятно, он предпочтет получить сообщение, объясняющее, почему некоторые тексты были пропущены при анализе. Пользователь ожидает увидеть какие-то результаты, но не знает, какие книги должны быть проанализированы. Возможно, ему и не нужно знать о недоступности каких-то файлов. Лишняя информация только сделает вашу программу менее удобной для пользователя. Средства обработки ошибок Python позволяют достаточно точно управлять тем, какой объем информации следует предоставить пользователю.

Хорошо написанный, правильно протестированный код редко содержит внутренние ошибки (например, синтаксические или логические). Но в любой ситуации, в которой ваша программа зависит от внешних факторов (пользовательского ввода, существования файла, доступности сетевого подключения), существует риск возникновения исключения. С накоплением практического опыта вы начнете видеть, в каких местах программы следует разместить блоки обработки исключений и сколько информации предоставлять пользователям о возникающих ошибках.

УПРАЖНЕНИЯ

10.6. Сложение: при вводе числовых данных часто встречается типичная проблема: пользователь вводит текст вместо чисел. При попытке преобразовать данные в `int` происходит исключение `ValueError`. Напишите программу, которая запрашивает два числа, складывает их и выводит результат. Перехватите исключение `ValueError`, если какое-либо из входных значений не является числом, и выведите удобное сообщение об ошибке. Протестируйте свою программу: сначала введите два числа, а потом введите текст вместо одного из чисел.

10.7. Калькулятор: заключите код из упражнения 10.6 в цикл `while`, чтобы пользователь мог продолжать вводить числа, даже если он допустил ошибку и ввел текст вместо числа.

10.8. Кошки и собаки: создайте два файла с именами cats.txt и dogs.txt. Сохраните по крайней мере три клички кошек в первом файле и три клички собак во втором. Напишите программу, которая пытается прочитать эти файлы и выводит их содержимое на экран. Заключите свой код в блок `try-except` для перехвата исключения `FileNotFoundError` и вывода понятного сообщения об отсутствии файла. Переместите один из файлов в другое место файловой системы; убедитесь в том, что код блока except выполняется как положено.

10.9. Ошибки без уведомления: измените блок `except` из упражнения 10.8 так, чтобы при отсутствии файла программа продолжала работу, не уведомляя пользователя о проблеме.

10.10. Частые слова: зайдите на сайт проекта «Гутенберг» (http://gutenberg.org/) и найдите несколько книг для анализа. Загрузите текстовые файлы этих произведений или скопируйте текст из браузера в текстовый файл на вашем компьютере.

Для подсчета количества вхождений слова или выражения в строку можно воспользоваться методом `count()`. Например, следующий код подсчитывает количество вхождений `'row'` в строке:

```
>>> line = "Row, row, row your boat"
>>> line.count('row')
2
>>> line.lower().count('row')
3
```

Обратите внимание: преобразование строки к нижнему регистру функцией `lower()` позволяет найти все вхождения искомого слова независимо от регистра.

Напишите программу, которая читает файлы из проекта «Гутенберг» и определяет количество вхождений слова `'the'` в каждом тексте. Результат будет приближенным, потому что программа также будет учитывать такие слова, как `'then'` и `'there'`. Попробуйте повторить поиск для строки `'the '` (с пробелом в строке) и посмотрите, насколько уменьшится количество найденных результатов.

Сохранение данных

Многие ваши программы будут запрашивать у пользователя информацию. Например, пользователь может вводить настройки для компьютерной игры или данные для визуального представления. Чем бы ни занималась ваша программа, информация, введенная пользователем, будет сохраняться в структурах данных (таких, как списки или словари). Когда пользователь закрывает программу, введенную им информацию почти всегда следует сохранять на будущее. Простейший способ сохранения данных основан на использовании модуля `json`.

Модуль `json` обеспечивает запись простых структур данных Python в файл и загрузку данных из файла при следующем запуске программы. Модуль `json` также может использоваться для обмена данными между программами Python. Более того, формат данных JSON не привязан к Python, поэтому данные в этом формате можно передавать программам, написанным на многих других языках программирования. Это полезный и универсальный формат, который к тому же легко изучается.

ПРИМЕЧАНИЕ Формат JSON (JavaScript Object Notation) был изначально разработан для JavaScript. Впрочем, с того времени он стал использоваться во многих языках, включая Python.

Функции json.dump() и json.load()

Напишем короткую программу для сохранения набора чисел и другую программу, которая будет читать эти числа обратно в память. Первая программа использует функцию `json.dump()`, а вторая — функцию `json.load()`.

Функция `json.dump()` получает два аргумента: сохраняемые данные и объект файла, используемый для сохранения. В следующем примере `json.dump()` используется для сохранения списка чисел:

number_writer.py

```
import json

numbers = [2, 3, 5, 7, 11, 13]

❶ filename = 'numbers.json'
❷ with open(filename, 'w') as f:
❸     json.dump(numbers, f)
```

Программа импортирует модуль `json` и создает список чисел для работы. В точке ❶ выбирается имя файла для хранения списка. Обычно для таких файлов принято использовать расширение .json, указывающее, что данные в файле хранятся в формате JSON. Затем файл открывается в режиме записи, чтобы модуль `json` мог записать в него данные ❷. В точке ❸ функция `json.dump()` используется для сохранения списка `numbers` в файле numbers.json.

Программа ничего не выводит, но давайте откроем файл numbers.json и посмотрим на его содержимое. Данные хранятся в формате, очень похожем на код Python:

```
[2, 3, 5, 7, 11, 13]
```

А теперь напишем следующую программу, которая использует `json.load()` для чтения списка обратно в память:

number_reader.py

```
import json

❶ filename = 'numbers.json'
❷ with open(filename) as f:
❸     numbers = json.load(f)

print(numbers)
```

В точке ❶ для чтения данных используется тот же файл, в который эти данные были записаны. На этот раз файл открывается в режиме чтения, потому что Python

нужно только прочитать данные из файла ❷. В точке ❸ функция `json.load()` используется для загрузки информации из numbers.json; эта информация сохраняется в переменной `numbers`. Наконец, программа выводит прочитанный список. Как видите, это тот же список, который был создан в программе number_writer.py:

```
[2, 3, 5, 7, 11, 13]
```

Модуль `json` позволяет организовать простейший обмен данными между программами.

Сохранение и чтение данных, сгенерированных пользователем

Сохранение с использованием модуля `json` особенно полезно при работе с данными, сгенерированными пользователем, потому что без сохранения эта информация будет потеряна при остановке программы. В следующем примере программа запрашивает у пользователя имя при первом запуске программы и «вспоминает» его при повторных запусках.

Начнем с сохранения имени пользователя:

remember_me.py

```
  import json

❶ username = input("What is your name? ")

  filename = 'username.json'
  with open(filename, 'w') as f:
❷     json.dump(username, f)
❸     print(f"We'll remember you when you come back, {username}!")
```

В точке ❶ программа запрашивает имя пользователя для сохранения. Затем вызывается функция `json.dump()`, которой передается имя пользователя и объект файла; функция сохраняет имя пользователя в файле ❷. Далее выводится сообщение о том, что имя пользователя было сохранено ❸:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

А теперь напишем другую программу, которая приветствует пользователя по ранее сохраненному имени:

greet_user.py

```
  import json

  filename = 'username.json'

  with open(filename) as f:
❶     username = json.load(f)
❷     print(f"Welcome back, {username}!")
```

В точке ❶ вызов `json.load()` читает информацию из файла username.json в переменную `username`. После того как данные будут успешно прочитаны, мы можем поприветствовать пользователя по имени ❷:

```
Welcome back, Eric!
```

Теперь эти две программы необходимо объединить в один файл. Когда пользователь запускает remember_me.py, программа должна взять имя пользователя из памяти, если это возможно; соответственно, программа начинается с блока `try`, который пытается прочитать имя пользователя. Если файл username.json не существует, блок `except` запросит имя пользователя и сохранит его в username.json на будущее:

remember_me.py

```
  import json

  # Программа загружает имя пользователя, если оно было сохранено ранее.
  #  В противном случае она запрашивает имя пользователя и сохраняет его.
  filename = 'username.json'
  try:
❶     with open(filename) as f:
❷         username = json.load(f)
❸ except FileNotFoundError:
❹     username = input("What is your name? ")
❺     with open(filename, 'w') as f:
         json.dump(username, f)
         print(f"We'll remember you when you come back, {username}!")
  else:
      print(f"Welcome back, {username}!")
```

Никакого нового кода здесь нет; просто блоки кода из двух предыдущих примеров были объединены в один файл. В точке ❶ программа пытается открыть файл username.json. Если файл существует, программа читает имя пользователя в память ❷ и выводит сообщение, приветствующее пользователя, в блоке `else`. Если программа запускается впервые, то файл username.json не существует и происходит исключение `FileNotFoundError` ❸. Python переходит к блоку `except`, в котором пользователю предлагается ввести имя ❹. Затем программа вызывает `json.dump()` для сохранения имени пользователя и выводит приветствие ❺.

Какой бы блок ни выполнялся, результатом является имя пользователя и соответствующее сообщение. При первом запуске программы результат выглядит так:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Если же программа уже была выполнена хотя бы один раз, то результат будет таким:

```
Welcome back, Eric!
```

Рефакторинг

Часто возникает типичная ситуация: код работает, но вы понимаете, что его структуру можно усовершенствовать, разбив его на функции, каждая из которых решает свою конкретную задачу. Этот процесс называется *рефакторингом* (или переработкой). Рефакторинг делает ваш код более чистым, понятным и простым в расширении.

В процессе рефакторинга remember_me.py мы можем переместить основную часть логики в одну или несколько функций. Основной задачей remember_me.py является вывод приветствия для пользователя, поэтому весь существующий код будет перемещен в функцию `greet_user()`:

remember_me.py

```
import json

def greet_user():
❶    """Приветствует пользователя по имени."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
        username = input("What is your name? ")
        with open(filename, 'w') as f:
            json.dump(username, f)
            print(f"We'll remember you when you come back, {username}!")
    else:
        print(f"Welcome back, {username}!")

greet_user()
```

С переходом на функцию комментарии дополняются строкой документации, которая описывает работу кода в текущей версии ❶. Код становится немного чище, но функция `greet_user()` не только приветствует пользователя — она также загружает хранимое имя пользователя, если оно существует, и запрашивает новое имя, если оно не было сохранено ранее.

Переработаем функцию `greet_user()`, чтобы она не решала столько разных задач. Начнем с перемещения кода загрузки хранимого имени пользователя в отдельную функцию:

```
import json

def get_stored_username():
❶    """Получает хранимое имя пользователя, если оно существует."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
```

```
❷         return None
      else:
          return username

  def greet_user():
      """Приветствует пользователя по имени."""
      username = get_stored_username()
❸     if username:
          print(f"Welcome back, {username}!")
      else:
          username = input("What is your name? ")
          filename = 'username.json'
          with open(filename, 'w') as f:
              json.dump(username, f)
              print(f"We'll remember you when you come back, {username}!")
  greet_user()
```

Новая функция `get_stored_username()` имеет четкое предназначение, изложенное в строке документации ❶. Эта функция читает и возвращает сохраненное имя пользователя, если его удается найти. Если файл `username.json` не существует, то функция возвращает `None` ❷. И это правильно: функция должна возвращать либо ожидаемое значение, либо `None`. Это позволяет провести простую проверку возвращаемого значения функции. В точке ❸ программа выводит приветствие для пользователя, если попытка получения имени пользователя была успешной; в противном случае программа запрашивает новое имя пользователя.

Из функции `greet_user()` стоит вынести еще один блок кода. Если имя пользователя не существует, то код запроса нового имени должен размещаться в функции, специализирующейся на решении этой задачи:

```
import json

def get_stored_username():
    """Получает хранимое имя пользователя, если оно существует."""
    ...

def get_new_username():
    """Запрашивает новое имя пользователя."""
    username = input("What is your name? ")
    filename = 'username.json'
    with open(filename, 'w') as f:
        json.dump(username, f)
    return username

def greet_user():
    """Приветствует пользователя по имени."""
    username = get_stored_username()
    if username:
        print(f"Welcome back, {username}!")
    else:
        username = get_new_username()
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

Каждая функция в окончательной версии remember_me.py имеет четкое, конкретное предназначение. Мы вызываем `greet_user()`, и эта функция выводит нужное приветствие: либо для уже знакомого, либо для нового пользователя. Для этого функция вызывает функцию `get_stored_username()`, которая отвечает только за чтение хранимого имени пользователя (если оно есть). Наконец, функция `greet_user()` при необходимости вызывает функцию `get_new_username()`, которая отвечает только за получение нового имени пользователя и его сохранение. Такое «разделение обязанностей» является важнейшим аспектом написания чистого кода, простого в сопровождении и расширении.

УПРАЖНЕНИЯ

10.11. Любимое число: напишите программу, которая запрашивает у пользователя его любимое число. Воспользуйтесь функцией `json.dump()` для сохранения этого числа в файле. Напишите другую программу, которая читает это значение и выводит сообщение: «Я знаю ваше любимое число! Это _____».

10.12. Сохраненное любимое число: объедините две программы из упражнения 10.11 в один файл. Если число уже сохранено, сообщите его пользователю, а если нет — запросите любимое число пользователя и сохраните в файле. Выполните программу дважды, чтобы убедиться в том, что она работает.

10.13. Проверка пользователя: последняя версия remember_me.py предполагает, что пользователь либо уже ввел свое имя, либо программа выполняется впервые. Ее нужно изменить на тот случай, если текущий пользователь не является тем человеком, который последним использовал программу.

Прежде чем выводить приветствие в `greet_user()`, спросите, правильно ли определено имя пользователя. Если ответ будет отрицательным, вызовите `get_new_username()` для получения правильного имени пользователя.

Итоги

В этой главе вы научились работать с файлами. Вы узнали, как прочитать сразу весь файл и как читать его содержимое по строкам. Вы научились записывать в файл и присоединять текст в конец файла, познакомились с исключениями и средствами обработки исключений, возникающих в программе. В завершающей части главы рассматриваются структуры данных Python для сохранения введенной информации, чтобы пользователю не приходилось каждый раз вводить данные заново при каждом запуске программы.

В главе 11 мы займемся эффективной организацией тестирования вашего кода. Тестирование поможет убедиться в том, что написанный код работает правильно, а также выявит ошибки, внесенные в процессе расширения уже написанных программ.

11 Тестирование

Вместе с функциями и классами вы также можете написать тесты для своего кода. Тестирование доказывает, что код работает так, как положено, для любых разновидностей входных данных, которые он может получать. Тесты позволят вам быть уверенными в том, что код будет работать правильно и тогда, когда вашими программами начнут пользоваться другие люди. Тестирование при добавлении нового кода гарантирует, что внесенные изменения не изменят текущее поведение программы. Все программисты допускают ошибки, поэтому каждый программист должен часто тестировать свой код и выявлять ошибки до того, как с ними столкнутся другие пользователи.

В этой главе вы научитесь тестировать код средствами модуля Python `unittest`. Вы узнаете, как построить тестовые сценарии, как проверить, что для конкретных входных данных программа выдает ожидаемый результат. Вы поймете, как выглядят успешно проходящие или сбойные тесты, и узнаете, как сбойный тест помогает усовершенствовать код. Также вы научитесь тестировать функции и классы и оценивать примерное количество необходимых тестов для проекта.

Тестирование функции

Чтобы потренироваться в тестировании, нам понадобится код. Ниже приведена простая функция, которая получает имя и фамилию и возвращает отформатированное полное имя:

name_function.py

```
def get_formatted_name(first, last):
    """Строит отформатированное полное имя."""
    full_name = f"{first} {last}"
    return full_name.title()
```

Функция `get_formatted_name()` строит полное имя из имени и фамилии, разделив их пробелом, преобразует первый символ каждого слова к верхнему регистру и возвращает полученный результат. Чтобы убедиться в том, что функция `get_formatted_name()` работает правильно, мы напишем программу, использующую

эту функцию. Программа names.py запрашивает у пользователя имя и фамилию и выдает отформатированное полное имя:

names.py

```
from name_function import get_formatted_name

print("Enter 'q' at any time to quit.")
while True:
    first = input("\nPlease give me a first name: ")
    if first == 'q':
        break
    last = input("Please give me a last name: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print(f"\tNeatly formatted name: {formatted_name}.")
```

Программа импортирует функцию `get_formatted_name()` из модуля name_function.py. Пользователь вводит последовательность имен и фамилий и видит, что программа сгенерировала отформатированные полные имена:

```
Enter 'q' at any time to quit.

Please give me a first name: janis
Please give me a last name: joplin
        Neatly formatted name: Janis Joplin.

Please give me a first name: bob
Please give me a last name: dylan
        Neatly formatted name: Bob Dylan.

Please give me a first name: q
```

Как видно из листинга, имена сгенерированы правильно. Но допустим, вы решили изменить функцию `get_formatted_name()`, чтобы она также работала со вторыми именами. При этом необходимо проследить за тем, чтобы функция не перестала правильно работать для имен, состоящих только из имени и фамилии. Чтобы протестировать код, можно запустить names.py и для проверки вводить имя из двух компонентов (скажем, `Janis Joplin`) при каждом изменении `get_formatted_name()`, но это довольно утомительно. К счастью, Python предоставляет эффективный механизм автоматизации тестирования вывода функций. При автоматизации тестирования `get_formatted_name()` вы будете уверены в том, что функция успешно работает для всех видов имен, для которых написаны тесты.

Модульные тесты и тестовые сценарии

Модуль `unittest` из стандартной библиотеки Python предоставляет функциональность для тестирования вашего кода. *Модульный тест* проверяет правильность ра-

боты одного конкретного аспекта поведения функции. *Тестовый сценарий* представляет собой совокупность модульных тестов, которые совместно доказывают, что функция ведет себя так, как положено, во всем диапазоне ситуаций, которые она должна обрабатывать. Хороший тестовый сценарий учитывает все возможные виды ввода, которые может получать функция, и включает тесты для представления всех таких ситуаций. Тестовый сценарий *с полным покрытием* включает полный спектр модульных тестов, покрывающих все возможные варианты использования функции. Обеспечение полного покрытия для крупного проекта может быть весьма непростой задачей. Часто бывает достаточно написать модульные тесты для критичных аспектов поведения вашего кода, а затем стремиться к полному покрытию только в том случае, если проект перейдет в фазу масштабного использования.

Прохождение теста

Вы не сразу привыкнете к синтаксису создания тестовых сценариев, но после того, как тестовый сценарий будет создан, вы сможете легко добавить новые модульные тесты для своих функций. Чтобы написать тестовый сценарий для функции, импортируйте модуль `unittest` и функцию, которую необходимо протестировать. Затем создайте класс, наследующий от `unittest.TestCase`, и напишите серию методов для тестирования различных аспектов поведения своей функции.

Ниже приведен тестовый сценарий с одним методом, который проверяет, что функция `get_formatted_name()` правильно работает при передаче имени и фамилии:

test_name_function.py

```
  import unittest
  from name_function import get_formatted_name

❶ class NamesTestCase(unittest.TestCase):
      """Тесты для 'name_function.py'."""

      def test_first_last_name(self):
          """Имена вида 'Janis Joplin' работают правильно?"""
❷         formatted_name = get_formatted_name('janis', 'joplin')
❸         self.assertEqual(formatted_name, 'Janis Joplin')

❹ if __name__ == '__main__':
      unittest.main()
```

Сначала мы импортируем `unittest` и тестируемую функцию `get_formatted_name()`. В точке ❶ создается класс `NamesTestCase`, который содержит серию модульных тестов для `get_formatted_name()`. Имя класса выбирается произвольно, но лучше выбрать имя, связанное с функцией, которую вы собираетесь тестировать, и включить в имя класса слово `Test`. Этот класс должен наследовать от класса `unittest.TestCase`, чтобы Python знал, как запустить написанные вами тесты.

Класс `NamesTestCase` содержит один метод, который тестирует всего один аспект `get_formatted_name()` — правильность форматирования имен, состоящих только из

имени и фамилии. Мы назвали этот метод `test_first_last_name()`. Любой метод, имя которого начинается с `test_`, будет выполняться автоматически при запуске test_name_function.py. В тестовом методе вызывается тестируемая функция и сохраняется возвращаемое значение, которое необходимо проверить. В данном примере вызывается функция `get_formatted_name()` с аргументами `'janis'` и `'joplin'`, а результат сохраняется в переменной `formatted_name` ❷.

В точке ❸ используется одна из самых полезных особенностей `unittest`: метод `assert`. Методы `assert` проверяют, что полученный результат соответствует тому результату, который вы рассчитывали получить. В данном случае известно, что функция `get_formatted_name()` должна вернуть полное имя с пробелами и капитализацией слов, поэтому переменная `formatted_name` должна содержать текст «Janis Joplin». Чтобы убедиться в этом, мы используем метод `assertEqual()` из модуля `unittest` и передаем ему переменную `formatted_name` и строку `'Janis Joplin'`. Вызов

```
self.assertEqual(formatted_name, 'Janis Joplin')
```

означает: «Сравни значение `formatted_name` со строкой `'Janis Joplin'`. Если они равны, как и ожидалось, — хорошо. Но если они не равны, обязательно сообщи мне!»

Мы запустим этот файл напрямую, но важно заметить, что многие тестовые фреймворки импортируют ваши тестовые файлы перед их выполнением. При импортировании файла интерпретатор выполняет файл в процессе импортирования. Блок `if` в точке ❹ проверяет специальную переменную `__name__`, значение которой задается при выполнении программы. Если файл выполняется как главная программа, то переменной `__name__` будет присвоено значение `'__main__'`. В этом случае вызывается метод `unittest.main()`, который выполняет тестовый сценарий. Если файл импортируется тестовым сценарием, то переменная `__name__` будет содержать значение `'__main__'`, и этот блок выполняться не будет.

При запуске test_name_function.py будет получен следующий результат:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Точка в первой строке вывода сообщает, что один тест прошел успешно. Следующая строка говорит, что для выполнения одного теста Python потребовалось менее 0,001 секунды. Наконец, завершающее сообщение `OK` говорит о том, что прошли все модульные тесты в тестовом сценарии.

Этот результат показывает, что функция `get_formatted_name()` успешно работает для полных имен, состоящих из имени и фамилии, если только функция не была изменена. В случае внесения изменений в `get_formatted_name()` тест можно запустить снова. И если тестовый сценарий снова пройдет, мы будем знать, что функция продолжает успешно работать с полными именами из двух компонентов.

Сбой теста

Как выглядит сбойный тест? Попробуем изменить функцию `get_formatted_name()`, чтобы она работала со вторыми именами, но сделаем это так, чтобы она перестала работать с полными именами из двух компонентов.

Новая версия `get_formatted_name()` с дополнительным аргументом второго имени выглядит так:

name_function.py

```
def get_formatted_name(first, middle, last):
    """Строит отформатированное полное имя."""
    full_name = f"{first} {middle} {last}"
    return full_name.title()
```

Эта версия должна работать для полных имен из трех компонентов, но тестирование показывает, что она перестала работать для полных имен из двух компонентов. На этот раз файл test_name_function.py выдает следующий результат:

```
❶ E
======================================================================
❷ ERROR: test_first_last_name (__main__.NamesTestCase)
----------------------------------------------------------------------
❸ Traceback (most recent call last):
    File "test_name_function.py", line 8, in test_first_last_name
      formatted_name = get_formatted_name('janis', 'joplin')
  TypeError: get_formatted_name() missing 1 required positional argument:
             'last'
----------------------------------------------------------------------
❹ Ran 1 test in 0.000s

❺ FAILED (errors=1)
```

Теперь информации гораздо больше, потому что при сбое теста разработчик должен знать, почему это произошло. Вывод начинается с одной буквы E ❶, которая сообщает, что один модульный тест в тестовом сценарии привел к ошибке. Затем мы видим, что ошибка произошла в тесте `test_first_last_name()` в `NamesTestCase` ❷. Конкретная информация о сбойном тесте особенно важна в том случае, если тестовый сценарий состоит из нескольких модульных тестов. В точке ❸ — стандартная трассировка, из которой видно, что вызов функции `get_formatted_name('janis', 'joplin')` перестал работать из-за необходимого позиционного аргумента.

Также из вывода следует, что был выполнен один модульный тест ❹. Наконец, дополнительное сообщение информирует, что тестовый сценарий в целом не прошел и в ходе выполнения произошла одна ошибка при выполнении тестового сценария ❺. Эта информация размещается в конце вывода, чтобы она была видна сразу; разработчику не придется прокручивать длинный протокол, чтобы узнать количество сбойных тестов.

Реакция на сбойный тест

Что делать, если тест не проходит? Если предположить, что проверяются правильные условия, прохождение теста означает, что функция работает правильно, а сбой — что в новом коде добавилась ошибка. Итак, если тест не прошел, изменять нужно не тест, а код, который привел к сбою теста. Проанализируйте изменения, внесенные в функцию, и разберитесь, как они привели к нарушению ожидаемого поведения.

В данном случае у функции `get_formatted_name()` было всего два обязательных параметра: имя и фамилия. Теперь она требует три обязательных параметра: имя, второе имя и фамилию. Добавление обязательного параметра для второго имени нарушило ожидаемое поведение `get_formatted_name()`. В таком случае лучше всего сделать параметр второго имени необязательным. После этого тесты для имен с двумя компонентами снова будут проходить, и программа сможет получать также вторые имена. Изменим функцию `get_formatted_name()`, чтобы параметр второго имени перестал быть обязательным, и снова выполним тестовый сценарий. Если он пройдет, можно переходить к проверке правильности обработки вторых имен.

Чтобы сделать второе имя необязательным, нужно переместить параметр `middle` в конец списка параметров в определении функции и задать ему пустое значение по умолчанию. Также будет добавлена проверка `if`, которая правильно строит полное имя в зависимости от того, передается второе имя или нет:

name_function.py

```
def get_formatted_name(first, last, middle=''):
    """Строит отформатированное полное имя."""
    if middle:
        full_name = f"{first} {middle} {last}"
    else:
        full_name = f"{first} {last}"
    return full_name.title()
```

В новой версии `get_formatted_name()` параметр `middle` необязателен. Если второе имя передается функции, то полное имя будет содержать имя, второе имя и фамилию. В противном случае полное имя состоит только из имени и фамилии. Теперь функция должна работать для обеих разновидностей имен. Чтобы узнать, работает ли функция для имен из двух компонентов, снова запустите test_name_function.py:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Теперь тестовый сценарий проходит. Такой исход идеален; он означает, что функция снова работает для имен из двух компонентов и нам не придется тестировать функцию вручную. Исправить ошибку было несложно, потому что сбойный тест помог выявить новый код, нарушивший существующее поведение.

Добавление новых тестов

Теперь мы знаем, что `get_formatted_name()` работает для простых имен, и можем написать второй тест для имен из трех компонентов. Для этого в класс `NamesTestCase` добавляется еще один метод:

test_name_function.py

```
  ...
  class NamesTestCase(unittest.TestCase):
      """Тесты для 'name_function.py'."""

      def test_first_last_name(self):
      ...

      def test_first_last_middle_name(self):
          """Работают ли такие имена, как 'Wolfgang Amadeus Mozart'?"""
❶         formatted_name = get_formatted_name(
              'wolfgang', 'mozart', 'amadeus')
          self.assertEqual(formatted_name, 'Wolfgang Amadeus Mozart')

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

Новому методу присваивается имя `test_first_last_middle_name()`. Имя метода должно начинаться с `test_`, чтобы этот метод выполнялся автоматически при запуске test_name_function.py. В остальном имя выбирается так, чтобы оно четко показывало, какое именно поведение `get_formatted_name()` мы тестируем. В результате при сбое теста вы сразу видите, к каким именам он относится. Не нужно опасаться длинных имен методов в классах `TestCase`: имена должны быть содержательными, чтобы донести информацию до разработчика в случае сбоя, а поскольку Python вызывает их автоматически, вам никогда не придется вручную вводить эти имена при вызове.

Чтобы протестировать функцию, мы вызываем `get_formatted_name()` с тремя компонентами ❶, после чего используем `assertEqual()` для проверки того, что возвращенное полное имя совпадает с ожидаемым. При повторном запуске test_name_function.py оба теста проходят успешно:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Отлично! Теперь мы знаем, что функция по-прежнему работает с именами из двух компонентов, как `Janis Joplin`, но можем быть уверены в том, что она сработает и для имен с тремя компонентами, такими как `Wolfgang Amadeus Mozart`.

УПРАЖНЕНИЯ

11.1. Город, страна: напишите функцию, которая получает два параметра: название страны и название города. Функция должна возвращать одну строку в формате «Город, Страна» — например, «Santiago, Chile». Сохраните функцию в модуле с именем city_functions.py.

Создайте файл test_cities.py для тестирования только что написанной функции (не забудьте импортировать unittest и тестируемую функцию). Напишите метод `test_city_country()` для проверки того, что вызов функции с такими значениями, как `'santiago'` и `'chile'`, дает правильную строку. Запустите test_cities.py и убедитесь в том, что тест `test_city_country()` проходит успешно.

11.2. Население: измените свою функцию так, чтобы у нее был третий обязательный параметр — население. В новой версии функция должна возвращать одну строку вида «Santiago, Chile — population 5 000 000». Снова запустите программу test_cities.py. Убедитесь в том, что тест `test_city_country()` на этот раз не проходит.

Измените функцию так, чтобы параметр населения стал необязательным. Снова запустите test_cities.py и убедитесь в том, что тест `test_city_country()` снова проходит успешно.

Напишите второй тест `test_city_country_population()`, который проверяет вызов функции со значениями `'santiago'`, `'chile'` и `'population=5000000'`. Снова запустите test_cities.py и убедитесь в том, что новый тест проходит успешно.

Тестирование класса

В первой части этой главы мы писали тесты для отдельной функции. Сейчас мы займемся написанием тестов для класса. Классы будут использоваться во многих ваших программах, поэтому возможность доказать, что ваши классы работают правильно, будет безусловно полезной. Если тесты для класса, над которым вы работаете, проходят успешно, вы можете быть уверены в том, что дальнейшая доработка класса не приведет к случайному нарушению его текущего поведения.

Разные методы assert

Класс `unittest.TestCase` содержит целое семейство проверочных методов `assert`. Как упоминалось ранее, эти методы проверяют, выполняется ли условие, которое должно выполняться в определенной точке вашего кода. Если условие истинно, как и предполагалось, то ваши ожидания относительно поведения части вашей программы подтверждаются; вы можете быть уверены в отсутствии ошибок. Если же условие, которое должно быть истинным, окажется ложным, то Python выдает исключение.

В табл. 11.1 перечислены шесть часто используемых методов `assert`. С их помощью можно проверить, что возвращаемые значения равны или не равны ожидаемым, что значения равны `True` или `False` или что значения входят или не входят в заданный список. Эти методы могут использоваться только в классах, наследующих от `unittest.TestCase`; рассмотрим пример использования такого метода в контексте тестирования реального класса.

Таблица 11.1. Методы assert, предоставляемые модулем unittest

Метод	Использование
`assertEqual(a, b)`	Проверяет, что `a == b`
`assertNotEqual(a, b)`	Проверяет, что `a != b`
`assertTrue(x)`	Проверяет, что значение `x` истинно
`assertFalse(x)`	Проверяет, что значение `x` ложно
`assertIn(элемент, список)`	Проверяет, что элемент входит в список
`assertNotIn(элемент, список)`	Проверяет, что элемент не входит в список

Класс для тестирования

Тестирование класса имеет много общего с тестированием функции — значительная часть работы направлена на тестирование поведения методов класса. Впрочем, существуют и различия, поэтому мы напишем отдельный класс для тестирования. Возьмем класс для управления проведением анонимных опросов:

survey.py

```
class AnonymousSurvey():
    """Сбор анонимных ответов на опросы."""

❶   def __init__(self, question):
        """Сохраняет вопрос и готовится к сохранению ответов."""
        self.question = question
        self.responses = []

❷   def show_question(self):
        """Выводит вопрос."""
        print(self.question)

❸   def store_response(self, new_response):
        """Сохраняет один ответ на опрос."""
        self.responses.append(new_response)

❹   def show_results(self):
        """Выводит все полученные ответы."""
        print("Survey results:")
        for response in self.responses:
            print(f"- {response}")
```

Класс начинается с вопроса, который вы предоставили ❶, и включает пустой список для хранения ответов. Класс содержит методы для вывода вопроса ❷, добавления нового ответа в список ответов ❸ и вывода всех ответов, хранящихся в списке ❹. Чтобы создать экземпляр на основе этого класса, необходимо предоставить вопрос. После того как будет создан экземпляр, представляющий конкретный опрос, программа выводит вопрос методом `show_question()`, сохраняет ответ методом `store_response()` и выводит результаты вызовом `show_results()`.

Чтобы продемонстрировать, что класс `AnonymousSurvey` работает, напишем программу, которая использует этот класс:

language_survey.py

```
from survey import AnonymousSurvey

# Определение вопроса с созданием экземпляра AnonymousSurvey.
question = "What language did you first learn to speak?"
my_survey = AnonymousSurvey(question)

# Вывод вопроса и сохранение ответов.
my_survey.show_question()
print("Enter 'q' at any time to quit.\n")
while True:
    response = input("Language: ")
    if response == 'q':
        break
    my_survey.store_response(response)

# Вывод результатов опроса.
print("\nThank you to everyone who participated in the survey!")
my_survey.show_results()
```

Программа определяет вопрос и создает объект `AnonymousSurvey` на базе этого вопроса. Программа вызывает метод `show_question()` для вывода вопроса, после чего переходит к получению ответов. Каждый ответ сохраняется сразу же при получении. Когда ввод ответов был завершен (пользователь ввел `q`), метод `show_results()` выводит результаты опроса:

```
What language did you first learn to speak?
Enter 'q' at any time to quit.

Language: English
Language: Spanish
Language: English
Language: Mandarin
Language: q

Thank you to everyone who participated in the survey!
Survey results:
- English
- Spanish
- English
- Mandarin
```

Этот класс работает для простого анонимного опроса. Но допустим, вы решили усовершенствовать класс `AnonymousSurvey` и модуль `survey`, в котором он находится. Например, каждому пользователю будет разрешено ввести несколько ответов. Или вы напишете метод, который будет выводить только уникальные ответы и сообщать, сколько раз был дан тот или иной ответ. Или вы напишете другой класс для проведения неанонимных опросов.

Реализация таких изменений грозит повлиять на текущее поведение класса `AnonymousSurvey`. Например, может оказаться, что поддержка ввода нескольких ответов случайно повлияет на процесс обработки одиночных ответов. Чтобы гарантировать, что доработка модуля не нарушит существующее поведение, для класса нужно написать тесты.

Тестирование класса AnonymousSurvey

Напишем тест, проверяющий всего один аспект поведения `AnonymousSurvey`. Этот тест будет проверять, что один ответ на опрос сохраняется правильно. После того как метод будет сохранен, метод `assertIn()` проверяет, что он действительно находится в списке ответов:

test_survey.py

```
  import unittest
  from survey import AnonymousSurvey
❶ class TestAnonmyousSurvey(unittest.TestCase):
      """Тесты для класса AnonymousSurvey"""

❷     def test_store_single_response(self):
          """Проверяет, что один ответ сохранен правильно."""
          question = "What language did you first learn to speak?"
❸         my_survey = AnonymousSurvey(question)
          my_survey.store_response('English')
❹         self.assertIn('English', my_survey.responses)

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

Программа начинается с импортирования модуля `unittest` и тестируемого класса `AnonymousSurvey`. Тестовый сценарий `TestAnonymousSurvey`, как и в предыдущих случаях, наследует от `unittest.TestCase` ❶. Первый тестовый метод проверяет, что сохраненный ответ действительно попадает в список ответов опроса. Этому методу присваивается хорошее содержательное имя `test_store_single_response()` ❷. Если тест не проходит, имя метода в выходных данных сбойного теста ясно показывает, что проблема связана с сохранением отдельного ответа на опрос.

Чтобы протестировать поведение класса, необходимо создать экземпляр класса. В точке ❸ создается экземпляр с именем `my_survey` для вопроса `"What language did you first learn to speak?"`. Один ответ (`English`) сохраняется с использованием метода `store_response()`. Затем программа убеждается в том, что ответ был сохранен правильно; для этого она проверяет, что значение `English` присутствует в списке `my_survey.responses` ❹.

При запуске программы test_survey.py тест проходит успешно:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Неплохо, но опрос с одним ответом вряд ли можно назвать полезным. Убедимся в том, что три ответа сохраняются правильно. Для этого в `TestAnonymousSurvey` добавляется еще один метод:

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Тесты для класса AnonymousSurvey"""

    def test_store_single_response(self):
    ...

    def test_store_three_responses(self):
        """Проверяет, что три ответа были сохранены правильно."""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
❶       responses = ['English', 'Spanish', 'Mandarin']
        for response in responses:
            my_survey.store_response(response)

❷       for response in responses:
            self.assertIn(response, my_survey.responses)

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

Новому методу присваивается имя `test_store_three_responses()`. Мы создаем объект опроса по аналогии с тем, как это делалось в `test_store_single_response()`. Затем определяется список, содержащий три разных ответа ❶, и для каждого из этих ответов вызывается метод `store_response()`. После того как ответы будут сохранены, следующий цикл проверяет, что каждый ответ теперь присутствует в `my_survey.responses` ❷.

Если снова запустить test_survey.py, оба теста (для одного ответа и для трех ответов) проходят успешно:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Все прекрасно работает. Тем не менее тесты выглядят немного однообразно, поэтому мы воспользуемся еще одной возможностью `unittest` для повышения их эффективности.

Метод setUp()

В программе test_survey.py в каждом тестовом методе создавался новый экземпляр `AnonymousSurvey`, а также новые ответы. Класс `unittest.TestCase` содержит метод

setUp(), который позволяет создать эти объекты один раз, а затем использовать их в каждом из тестовых методов. Если в класс TestCase включается метод setUp(), Python выполняет метод setUp() перед запуском каждого метода, имя которого начинается с test_. Все объекты, созданные методом setUp(), становятся доступными во всех написанных вами тестовых методах.

Применим метод setUp() для создания экземпляра AnonymousSurvey и набора ответов, которые могут использоваться в test_store_single_response() и test_store_three_responses():

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Тесты для класса AnonymousSurvey."""

    def setUp(self):
        """
        Создание опроса и набора ответов для всех тестовых методов.
        """
        question = "What language did you first learn to speak?"
❶       self.my_survey = AnonymousSurvey(question)
❷       self.responses = ['English', 'Spanish', 'Mandarin']

    def test_store_single_response(self):
        """Проверяет, что один ответ сохранен правильно."""
        self.my_survey.store_response(self.responses[0])
        self.assertIn(self.responses[0], self.my_survey.responses)

    def test_store_three_responses(self):
        """Проверяет, что три ответа были сохранены правильно."""
        for response in self.responses:
            self.my_survey.store_response(response)
        for response in self.responses:
            self.assertIn(response, self.my_survey.responses)

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

Метод setUp() решает две задачи: он создает экземпляр опроса ❶ и список ответов ❷. Каждый из этих атрибутов снабжается префиксом self, поэтому он может использоваться где угодно в классе. Это обстоятельство упрощает два тестовых метода, потому что им уже не нужно создавать экземпляр опроса или ответы. Метод test_store_single_response() убеждается в том, что первый ответ в self.responses — self.responses[0] — сохранен правильно, а метод test_store_single_response() убеждается в том, что правильно были сохранены все три ответа в self.responses.

При повторном запуске test_survey.py оба теста по-прежнему проходят. Эти тесты будут особенно полезными при расширении AnonymousSurvey с поддержкой нескольких ответов для каждого участника. После внесения изменений вы можете

повторить тесты и убедиться в том, что изменения не повлияли на возможность сохранения отдельного ответа или серии ответов.

При тестировании классов, написанных вами, метод `setUp()` упрощает написание тестовых методов. Вы создаете один набор экземпляров и атрибутов в `setUp()`, а затем используете эти экземпляры во всех тестовых методах. Это намного проще и удобнее, чем создавать новый набор экземпляров и атрибутов в каждом тестовом методе.

ПРИМЕЧАНИЕ Во время работы тестового сценария Python выводит один символ для каждого модульного теста после его завершения. Для прошедшего теста выводится точка; если при выполнении произошла ошибка, выводится символ E, а если не прошла проверка условия assert, выводится символ F. Вот почему вы увидите другое количество точек и символов в первой строке вывода при выполнении ваших тестовых сценариев. Если выполнение тестового сценария занимает слишком много времени, потому что сценарий содержит слишком много тестов, эти символы дадут некоторое представление о количестве прошедших тестов.

УПРАЖНЕНИЯ

11.3. Работник: напишите класс `Employee`, представляющий работника. Метод `__init__()` должен получать имя, фамилию и ежегодный оклад; все эти значения должны сохраняться в атрибутах. Напишите метод `give_raise()`, который по умолчанию увеличивает ежегодный оклад на $5000 — но при этом может получать другую величину прибавки.

Напишите тестовый сценарий для `Employee`. Напишите два тестовых метода, `test_give_default_raise()` и `test_give_custom_raise()`. Используйте метод `setUp()`, чтобы вам не приходилось заново создавать экземпляр `Employee` в каждом тестовом метода. Запустите свой тестовый сценарий и убедитесь в том, что оба теста прошли успешно.

Итоги

В этой главе вы научились писать тесты для функций и классов с использованием средств модуля `unittest`. Вы узнали, как написать класс, наследующий от `unittest.TestCase`, и как писать тестовые методы для проверки конкретных аспектов поведения ваших функций и классов. Вы научились использовать метод `setUp()` для эффективного создания экземпляров и атрибутов, которые могут использоваться во всех методах для тестирования класса.

Тестирование — важная тема, на которую многие новички не обращают внимания. Пока вы делаете свои первые шаги в программировании, писать тесты для простых проектов не нужно. Но как только вы начинаете работать над проектами, требующими значительных затрат ресурсов на разработку, непременно обеспечьте тестирование критических аспектов поведения ваших функций и классов. С эффективными тестами вы можете быть уверены в том, что изменения в проекте не повредят тому, что уже работает, а это развяжет вам руки для усовершенствования кода. Случайно нарушив существующую функциональность, вы немедленно

узнаете об этом, что позволит вам быстро исправить проблему. Отреагировать на сбой теста всегда намного проще, чем на отчет об ошибке от недовольного пользователя.

Другие программисты будут более уважительно относиться к вашим проектам, если вы включите исходные тесты. Они будут чувствовать себя более комфортно, экспериментируя с вашим кодом, и с большей готовностью присоединятся к участию в ваших проектах. Если вы будете участвовать в проекте, над которым работают другие программисты, вам придется продемонстрировать, что ваш код проходит существующие тесты; кроме того, от вас будут ждать, что вы напишете тесты для нового поведения, добавленного вами в проект.

Поэкспериментируйте с тестами и освойтесь с процессом тестирования кода. Пишите тесты для критических аспектов поведения ваших функций и классов, но не стремитесь к полному тестовому покрытию своих ранних проектов (если только у вас для этого нет особых причин).

Часть II
ПРОЕКТЫ

Поздравляем! Вы уже знаете о Python достаточно, чтобы взяться за построение интерактивных осмысленных проектов. Создание собственных проектов закрепит новые навыки и упрочит ваше понимание концепций, представленных в части I.

В части II представлены три типа проектов; вы можете взяться за любые из них в том порядке, который вам больше нравится. Ниже приведено краткое описание каждого проекта, чтобы вам было проще решить, с чего начать.

Программирование игры на языке Python

В проекте Alien Invasion (главы 12, 13 и 14) мы воспользуемся пакетом Pygame для написания 2D-игры, в которой игрок должен сбивать корабли пришельцев, падающие по экрану с нарастающей скоростью и сложностью. К концу этого проекта вы будете знать достаточно для того, чтобы создавать собственные 2D-игры с использованием Pygame.

Визуализация данных

Проект Data Visualization начинается с главы 15. В этом проекте вы научитесь генерировать данные и создавать практичные, элегантные визуализации этих данных с использованием пакетов matplotlib и Pygal. Глава 16 научит вас работать с данными из сетевых источников и передавать их пакету визуализации для построения графиков погодных данных и карты глобальной сейсмической активности. Наконец, глава 17 показывает, как написать программу для автоматической загрузки и визуализации данных. Навыки визуализации пригодятся вам в изучении области анализа данных — в современном мире это умение ценится очень высоко.

Веб-приложения

В проекте Web Applications (главы 18, 19 и 20) мы при помощи пакета Django создадим простое веб-приложение для ведения веб-дневника по произвольным темам. Пользователь создает учетную запись с именем и паролем, вводит тему и делает заметки. Вы также научитесь развертывать свои приложения так, чтобы сделать их доступными для потенциальных пользователей со всего мира.

После завершения проекта вы сможете заняться построением собственных простых веб-приложений. Кроме того, вы будете готовы к изучению более серьезных ресурсов, посвященных построению приложений с использованием Django.

Проект 1
Игра «Инопланетное вторжение»

12 Инопланетное вторжение

Давайте создадим собственную игру! Мы воспользуемся Pygame — подборкой интересных мощных модулей Python для управления графикой, анимацией и даже звуком, упрощающей построение сложных игр. Pygame берет на себя такие задачи, как перерисовка изображений на экране, что позволяет вам пропустить большую часть рутинного, сложного программирования и сосредоточиться на высокоуровневой логике игровой динамики.

В этой главе мы настроим Pygame и создадим корабль, который движется влево и вправо и стреляет по приказу пользователя. В следующих двух главах вы создадите флот инопланетного вторжения, а затем займетесь внесением усовершенствований, например ограничением количества попыток и добавлением таблицы рекордов.

Эта глава также научит вас управлять большими проектами, состоящими из многих файлов. Мы часто будем проводить рефакторинг и изменять структуру содержимого файлов, чтобы проект был четко организован, а код оставался эффективным.

Программирование игр — идеальный способ совместить изучение языка с развлечением. Написание простой игры поможет вам понять, как пишутся профессиональные игры. В процессе работы над этой главой вводите и запускайте код, чтобы понять, как каждый блок кода участвует в общем игровом процессе. Экспериментируйте с разными значениями и настройками, чтобы лучше понять, как следует организовать взаимодействие с пользователем в ваших собственных играх.

ПРИМЕЧАНИЕ Игра Alien Invasion состоит из множества файлов; создайте в своей системе новый каталог с именем `alien_invasion`. Чтобы команды `import` работали правильно, все файлы проекта должны находиться в этой папке.

Кроме того, если вы уверенно работаете с системами контроля версий, возможно, вам стоит использовать такую систему в этом проекте. Если ранее вы никогда не использовали системы контроля версий, обратитесь к краткому обзору в приложении Г.

Планирование проекта

Построение крупного проекта должно начинаться не с написания кода, а с планирования. План поможет вам сосредоточить усилия в нужном направлении и повысит вероятность успешного завершения проекта.

Итак, напишем общее описание игрового процесса. Хотя это описание не затрагивает все аспекты игры, оно дает достаточно четкое представление о том, с чего начинать работу:

> Каждый игрок управляет кораблем, который находится в середине нижнего края экрана. Игрок перемещает корабль вправо и влево клавишами управления курсором; клавиша «пробел» используется для стрельбы. В начале игры флот пришельцев находится в верхней части экрана и постепенно опускается вниз, также смещаясь в сторону. Игрок выстрелами уничтожает пришельцев. Если ему удается сбить всех пришельцев, появляется новый флот, который движется быстрее предыдущего. Если пришелец сталкивается с кораблем игрока или доходит до нижнего края экрана, игрок теряет корабль. Если игрок теряет все три корабля, игра заканчивается.

В первой фазе разработки мы создадим корабль, который может двигаться вправо и влево. Корабль должен стрелять из пушки, когда игрок нажимает клавишу «пробел». Когда это поведение будет реализовано, мы можем заняться пришельцами и доработкой игрового процесса.

Установка Pygame

Прежде чем браться за программирование, установите пакет Pygame. Модуль `pip` помогает управлять загрузкой и установкой пакетов Python. Чтобы установить Pygame, введите следующую команду в приглашении терминала:

```
$ python -m pip install --user pygame
```

Эта команда приказывает Python запустить модуль `pip` и включить пакет `pygame` в установленный экземпляр Python текущего пользователя. Если для запуска программ или сеанса терминала вместо `python` используется другая команда (например, `python3`), команда будет выглядеть так:

```
$ python3 -m pip install --user pygame
```

ПРИМЕЧАНИЕ Если команда не работает в macOS, попробуйте снова выполнить команду без флага `--user`.

Создание проекта игры

Построение игры начнется с создания пустого окна Pygame, в котором позднее будут отображаться игровые элементы, — прежде всего корабль и пришельцы.

Также игра должна реагировать на действия пользователя, назначать цвет фона и загружать изображение корабля.

Создание окна Pygame и обработка ввода

Начнем с создания пустого окна Pygame, для чего будет создан класс, представляющий окно. Создайте в текстовом редакторе новый файл и сохраните его с именем alien_invasion.py, после чего введите следующий код:

alien_invasion.py

```
import sys

import pygame

class AlienInvasion:
    """Класс для управления ресурсами и поведением игры."""

    def __init__(self):
        """Инициализирует игру и создает игровые ресурсы."""
❶       pygame.init()

❷       self.screen = pygame.display.set_mode((1200, 800))
        pygame.display.set_caption("Alien Invasion")

    def run_game(self):
        """Запуск основного цикла игры."""
❸       while True:
            # Отслеживание событий клавиатуры и мыши.
❹           for event in pygame.event.get():
❺               if event.type == pygame.QUIT:
                    sys.exit()

            # Отображение последнего прорисованного экрана.
❻           pygame.display.flip()

if __name__ == '__main__':
    # Создание экземпляра и запуск игры.
    ai = AlienInvasion()
    ai.run_game()
```

Программа начинается с импортирования модуля `sys` и `pygame`. Модуль `pygame` содержит функциональность, необходимую для создания игры, а модуль `sys` завершает игру по команде игрока.

Игра Alien Invasion начинается с класса с именем `AlienInvasion`. В методе `__init__()` функция `pygame.init()` инициализирует настройки, необходимые Pygame для нормальной работы ❶. В точке ❷ вызов `pygame.display.set_mode()` создает окно, в котором прорисовываются все графические элементы игры. Аргумент `(1200, 800)` представляет собой кортеж, определяющий размеры игрового окна. (Вы можете изменить эти значения в соответствии с размерами своего мони-

тора.) Объект окна присваивается атрибуту `self.screen`, что позволяет работать с ним во всех методах класса.

Объект, присвоенный `self.screen`, называется *поверхностью* (surface). Поверхность в Pygame представляет часть экрана, на которой отображается игровой элемент. Каждый элемент в игре (например, пришелец или корабль игрока) представлен поверхностью. Поверхность, возвращаемая `display.set_mode()`, представляет все игровое окно. При активизации игрового цикла анимации эта поверхность автоматически перерисовывается при каждом проходе цикла, чтобы она обновлялась всеми изменениями, обусловленными вводом от пользователя.

Процессом игры управляет метод `run_game()`. Метод содержит непрерывно выполняемый цикл `while` ❸, который содержит цикл событий и код, управляющий обновлениями экрана. *Событием* называется действие, выполняемое пользователем во время игры (например, нажатие клавиши или перемещение мыши). Чтобы наша программа реагировала на события, мы напишем *цикл событий* для *прослушивания* событий и выполнения соответствующей операции в зависимости от типа произошедшего события. Этим циклом событий является цикл `for` в точке ❹.

Для получения доступа к событиям, обнаруженным Pygame, используется метод `pygame.event.get()`. Он возвращает список событий, произошедших с момента последнего вызова этой функции. При любом событии клавиатуры или мыши отрабатывает цикл `for`. В этом цикле записывается серия команд `if` для обнаружения и обработки конкретных событий. Например, когда игрок щелкает на кнопке закрытия игрового окна, программа обнаруживает событие `pygame.QUIT` и вызывает метод `sys.exit()` для выхода из игры ❺.

Вызов `pygame.display.flip()` ❻ приказывает Pygame отобразить последний отрисованный экран. В данном случае при каждом выполнении цикла `while` будет отображаться пустой экран со стиранием старого экрана, так что виден будет только новый экран. При перемещении игровых элементов вызов `pygame.display.flip()` будет постоянно обновлять экран, отображая игровые элементы в новых позициях и скрывая старые изображения; таким образом создается иллюзия плавного движения.

В последней строке файла создается экземпляр игры, после чего вызывается метод `run_game()`. Вызов `run_game()` заключается в блок `if`, чтобы он выполнялся только при прямом вызове функции. Запустив файл alien_invasion.py, вы увидите пустое окно Pygame.

Назначение цвета фона

Pygame по умолчанию создает черный экран, но это банально — выберем другой цвет фона. Это делается в методе `__init__()`:

alien_invasion.py

```
  def __init__(self):
      ...
      pygame.display.set_caption("Alien Invasion")
```

```
        # Назначение цвета фона.
❶       self.bg_color = (230, 230, 230)

    def run_game(self):
        ...
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    sys.exit()

            # При каждом проходе цикла перерисовывается экран.
❷           self.screen.fill(self.bg_color)

            # Отображение последнего прорисованного экрана.
            pygame.display.flip()
```

Цвета в Pygame задаются в схеме RGB: тройками интенсивности красной, зеленой и синей составляющих цвета. Значение каждой составляющей лежит в диапазоне от 0 до 255. Цветовое значение (255, 0, 0) соответствует красному цвету, (0, 255, 0) — зеленому и (0, 0, 255) — синему. Разные сочетания составляющих RGB позволяют создать до 16 миллионов цветов. В цветовом значении (230, 230, 230) красная, синяя и зеленая составляющие смешиваются в равных долях, давая светло-серый цвет фона. Этот цвет сохраняется в переменной `self.bg_color` ❶.

В точке ❷ экран заполняется цветом фона. Для этого вызывается метод `fill()`, получающий всего один аргумент: цвет фона.

Создание класса Settings

Каждый раз, когда в нашу игру добавляется новая функциональность, также в нее обычно добавляются новые настройки (параметры конфигурации). Вместо того чтобы добавлять настройки в коде, мы напишем модуль с именем `settings`; этот модуль содержит класс с именем `Settings` для хранения всех настроек. Такое решение позволит передавать один объект вместо множества отдельных настроек. Кроме того, оно упрощает вызовы функций и изменение внешнего вида игры с ростом проекта. Чтобы внести изменения в игру, достаточно будет изменить некоторые значения в settings.py вместо того, чтобы искать разные настройки в файлах.

Создайте новый файл с именем settings.py в папке alien_invasion. Исходная версия класса `Settings` выглядит так:

settings.py

```
class Settings():
    """Класс для хранения всех настроек игры Alien Invasion."""

    def __init__(self):
        """Инициализирует настройки игры."""
        # Параметры экрана
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)
```

Чтобы создать экземпляр `Settings` и использовать его для обращения к настройкам, внесите изменения в alien_invasion.py:

alien_invasion.py

```
  ...
  import pygame

  from settings import Settings

  class AlienInvasion:
      """Класс для управления ресурсами и поведением игры."""

      def __init__(self):
          """Инициализирует игру и создает игровые ресурсы."""
          pygame.init()
❶         self.settings = Settings()

❷         self.screen = pygame.display.set_mode(
              (self.settings.screen_width, self.settings.screen_height))
          pygame.display.set_caption("Alien Invasion")

      def run_game(self):
              ...
              # При каждом проходе цикла перерисовывается экран.
❸             self.screen.fill(self.settings.bg_color)

              # Отображение последнего прорисованного экрана.
              pygame.display.flip()
  ...
```

Класс `Settings` импортируется в основной файл программы, после чего программа создает экземпляр `Settings` и сохраняет его в `self.settings` ❶после вызова `pygame.init()`. При создании экрана ❷ используются атрибуты `screen_width` и `screen_height` объекта `self.settings`, после чего объект `self.settings` также используется для получения цвета фона при заполнении экрана ❸.

Запустив файл alien_invasion.py, вы не заметите никаких изменений, потому что в этом разделе мы всего лишь переместили настройки, уже использованные в другом месте. Теперь можно переходить к добавлению новых элементов на экран.

Добавление изображения корабля

А теперь добавим в игру космический корабль, которым управляет игрок. Чтобы вывести его на экран, мы загрузим изображение, после чего воспользуемся методом Pygame `blit()` для вывода изображения.

Выбирая графику для своих игр, обязательно обращайте внимание на условия лицензирования. Самый безопасный и дешевый начальный вариант — использование бесплатной графики с таких сайтов, как http://pixabay.com/.

В игре можно использовать практически любые графические форматы, но проще всего использовать файлы в формате .bmp, потому что этот формат Pygame загружает по умолчанию. И хотя Pygame можно настроить для других типов файлов, некоторые типы зависят от установки на компьютере определенных графических библиотек. (Большинство изображений, которые вы найдете, имеют формат .jpg, .png или .gif, но их можно преобразовать в формат .bmp при помощи таких программ, как Photoshop, GIMP или Paint.)

Обратите особое внимание на цвет фона вашего изображения. Попробуйте найти файл с прозрачным фоном, который можно заменить любым цветом фона в графическом редакторе. Чтобы ваша игра хорошо смотрелась, цвет фона изображения должен соответствовать цвету фона игры. Также можно подобрать цвет фона игры под цвет фона изображения.

В игре Alien Invasion используется файл ship.bmp (рис. 12.1), который можно загрузить в числе ресурсов книги по адресу https://www.nostarch.com/pythoncrashcourse2e/. Цвет фона файла соответствует настройкам, используемым в проекте. Создайте в главном каталоге проекта (alien_invasion) каталог с именем images. Сохраните файл ship.bmp в каталоге images.

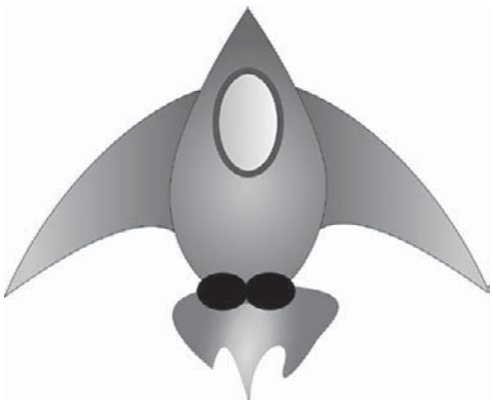

Рис. 12.1. Корабль для игры Alien Invasion

Создание класса Ship

После того как изображение корабля будет выбрано, его необходимо вывести на экран. Для работы с кораблем мы напишем модуль `ship`, содержащий класс `Ship`. Этот класс реализует большую часть поведения корабля.

ship.py

```
import pygame

class Ship():
    """Класс для управления кораблем."""
```

```
    def __init__(self, ai_game):
        """Инициализирует корабль и задает его начальную позицию."""
❶       self.screen = ai_game.screen
❷       self.screen_rect = ai_game.screen.get_rect()

        # Загружает изображение корабля и получает прямоугольник.
❸       self.image = pygame.image.load('images/ship.bmp')
        self.rect = self.image.get_rect()
        # Каждый новый корабль появляется у нижнего края экрана.
❹       self.rect.midbottom = self.screen_rect.midbottom

❺   def blitme(self):
        """Рисует корабль в текущей позиции."""
        self.screen.blit(self.image, self.rect)
```

Один из факторов эффективности Pygame заключается в том, что программист может выполнять операции с игровыми элементами как с прямоугольниками даже в том случае, если они имеют другую форму. Операции с прямоугольниками эффективны, потому что прямоугольник — простая геометрическая фигура. Обычно этот подход работает достаточно хорошо, и игроки не замечают, что программа не отслеживает точную геометрическую форму каждого игрового элемента. В этом классе корабль и экран будут рассматриваться как прямоугольные объекты.

Перед определением класса программа импортирует модуль `pygame`. Метод `__init__()` класса `Ship` получает два параметра: ссылку `self` и ссылку на текущий экземпляр класса `AlienInvasion`, так класс `Ship` получает доступ ко всем игровым ресурсам, определенным в `AlienInvasion`. В точке ❶ экран присваивается атрибуту `Ship`, чтобы к нему можно было легко обращаться во всех модулях класса. В точке ❷ программа обращается к атрибуту `rect` объекта экрана при помощи метода `get_rect()` и присваивает его `self.screen_rect`. Это позволяет разместить корабль в нужной позиции экрана.

Чтобы загрузить изображение, мы вызываем метод `pygame.image.load()` ❸ и передаем ему местоположение изображения корабля. Функция возвращает поверхность, представляющую корабль, которая присваивается `self.image`.

Когда изображение будет загружено, программа вызывает `get_rect()` для получения атрибута `rect` поверхности корабля, чтобы позднее использовать ее для позиционирования корабля.

При работе с объектом `rect` вам доступны координаты x и y верхней, нижней, левой и правой сторон, а также центра. Присваивая любые из этих значений, вы задаете текущую позицию прямоугольника. Местонахождение центра игрового элемента определяется атрибутами `center`, `centerx` или `centery` прямоугольника. Стороны определяются атрибутами `top`, `bottom`, `left` и `right`. Также имеются атрибуты, которые являются комбинацией этих свойств, например `midbottom`, `midtop`, `midleft` и `midright`. Для изменения горизонтального или вертикального расположения прямоугольника достаточно задать атрибуты `x` и `y`, содержащие координаты левого верхнего угла. Эти атрибуты избавляют вас от вычислений, которые раньше разработчикам игр приходилось выполнять вручную, притом достаточно часто.

ПРИМЕЧАНИЕ В Pygame начало координат (0, 0) находится в левом верхнем углу экрана, а оси направлены сверху вниз и слева направо. На экране размером 1200 × 800 начало координат располагается в левом верхнем углу, а правый нижний угол имеет координаты (1200, 800). Эти координаты относятся к игровому окну, а не к физическому экрану.

Корабль будет расположен в середине нижней стороны экрана. Для этого значение `self.rect.midbottom` выравнивается по атрибуту `midbottom` прямоугольника экрана ❹. Pygame использует эти атрибуты `rect` для позиционирования изображения, чтобы корабль был выровнен по центру, а его нижний край совпадал с нижним краем экрана.

В точке ❺ определяется метод `blitme()`, который выводит изображение на экран в позиции, заданной `self.rect`.

Вывод корабля на экран

Изменим программу alien_invasion.py, чтобы в ней создавался корабль и вызывался метод `blitme()` класса `Ship`:

alien_invasion.py

```
  ...
  from settings import Settings
  from ship import Ship

  class AlienInvasion:
      """Класс для управления ресурсами и поведением игры."""

      def __init__(self):
          ...
          pygame.display.set_caption("Alien Invasion")

❶         self.ship = Ship(screen)

      def run_game(self):
              ...
              # При каждом проходе цикла перерисовывается экран.
              self.screen.fill(self.settings.bg_color)
❷             self.ship.blitme()

              # Отображение последнего прорисованного экрана.
              pygame.display.flip()
  ...
```

После создания экрана программа импортирует класс `Ship` и создает его экземпляр ❶. При вызове `Ship` передается один аргумент — экземпляр `AlienInvasion`. Аргумент `self` относится к текущему экземпляру `AlienInvasion`. Этот параметр предоставляет `Ship` доступ к ресурсам игры, например к объекту `screen`. Экземпляр `Ship` присваивается `self.ship`.

После заполнения фона корабль рисуется на экране вызовом `ship.blitme()`, так что корабль выводится поверх фона ❷.

Если запустить alien_invasion.py сейчас, вы увидите пустой игровой экран, в центре нижней стороны которого находится корабль (рис. 12.2).

Рис. 12.2. Корабль в середине нижней стороны экрана

Рефакторинг: методы _check_events() и _update_screen()

В больших проектах перед добавлением нового кода часто проводится рефакторинг уже написанного кода. Рефакторинг упрощает структуру существующего кода и дальнейшее развитие проекта. В этом разделе метод `run_game()`, который становится слишком длинным, будет разбит на два вспомогательных метода. *Вспомогательный метод* работает во внутренней реализации класса, но не должен вызываться через экземпляр. В Python имена вспомогательных методов обозначаются начальным символом подчеркивания (_).

Метод _check_events()

Начнем с перемещения кода управления событиями в отдельный метод `_check_events()`. Тем самым вы упростите `run_game()` и изолируете цикл управления событиями от остального кода. Изоляция цикла событий позволит организовать

управление событиями отдельно от других аспектов игры (например, обновления экрана).

Ниже приведен класс `AlienInvasion` с новым методом `_check_events()`, который используется только в коде `run_game()`:

alien_invasion.py

```
    def run_game(self):
        """Запуск основного цикла игры."""
        while True:
❶           self._check_events()
        # При каждом проходе цикла перерисовывается экран.
        ...

❷   def _check_events(self):
        """Обрабатывает нажатия клавиш и события мыши."""
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
```

Мы определяем новый метод `_check_events()` ❷ и перемещаем строки, которые проверяют, не закрыл ли игрок окно щелчком кнопки мыши, в этот новый метод.

Для вызова метода внутри класса используется точечный синтаксис с переменной `self` и именем метода ❶. Затем метод вызывается в цикле `while` метода `run_game()`.

Метод _update_screen()

Для дальнейшего упрощения `run_game()` выделим код обновления экрана в отдельный метод `_update_screen()`:

alien_invasion.py

```
    def run_game(self):
        """Запуск основного цикла игры."""
        while True:
            self._check_events()
            self._update_screen()

    def _check_events(self):
        ...

    def _update_screen(self):
        """Обновляет изображения на экране и отображает новый экран."""
        self.screen.fill(self.settings.bg_color)
        self.ship.blitme()

        pygame.display.flip()
```

Код прорисовки фона и переключения экрана перемещен в `_update_screen()`. Тело основного цикла в `run_game()` серьезно упростилось. С первого взгляда видно, что программа отслеживает новые события и обновляет экран при каждом проходе цикла.

Если вы уже написали несколько игр, вероятно, вы с самого начала начнете разбивать свой код на такие методы. Но если вы никогда не брались за подобный проект, вероятно, вы не знаете, как структурировать этот код. Эта последовательность дает представление о реальном процессе разработки: сначала вы пишете свой код в самом простом виде, а потом подвергаете его рефакторингу по мере роста сложности проекта.

Теперь, когда мы изменили структуру кода и упростили его расширение, можно переходить к динамическим аспектам игры.

УПРАЖНЕНИЯ

12.1. Синее небо: создайте окно Pygame с синим фоном.

12.2. Игровой персонаж: найдите изображение игрового персонажа, который вам нравится, в формате .bmp (или преобразуйте существующее изображение). Создайте класс, который рисует персонажа в центре экрана, и приведите цвет фона изображения в соответствие с цветом фона экрана (или наоборот).

Управление кораблем

Реализуем возможность перемещения корабля по горизонтали. Для этого мы напишем код, реагирующий на нажатие клавиш ← или →. Начнем с движения вправо, а потом применим те же принципы к движению влево. Заодно вы научитесь управлять перемещением изображений на экране.

Обработка нажатия клавиши

Каждый раз, когда пользователь нажимает клавишу, это нажатие регистрируется в Pygame как событие. Каждое событие идентифицируется методом `pygame.event.get()`, поэтому в методе `_check_events()` необходимо указать, какие события должны отслеживаться. Каждое нажатие клавиши регистрируется как событие `KEYDOWN`.

При обнаружении события `KEYDOWN` необходимо проверить, была ли нажата клавиша, инициирующая некоторое игровое событие. Например, при нажатии клавиши → значение `rect.x` корабля увеличивается для перемещения корабля вправо:

alien_invasion.py

```
    def _check_events(self):
        """Обрабатывает нажатия клавиш и события мыши."""
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
❶           elif event.type == pygame.KEYDOWN:
❷               if event.key == pygame.K_RIGHT:
                    # Переместить корабль вправо.
❸                   self.ship.rect.x += 1
```

Внутри `_check_events()` в цикл событий добавляется блок `elif` для выполнения кода при обнаружении события `KEYDOWN` ❶. Чтобы проверить, является ли нажатая клавиша клавишей → (`pygame.K_RIGHT`), мы читаем атрибут `event.key` ❷. Если нажата клавиша →, корабль перемещается вправо, для чего значение `self.ship.rect.x` увеличивается на 1 ❸.

Если запустить программу alien_invasion.py сейчас, вы увидите, что корабль перемещается вправо на 1 пиксел при каждом нажатии клавиши →. Неплохо для начала, но это не лучший способ управления кораблем. Чтобы управление было более удобным, следует реализовать возможность непрерывного перемещения.

Непрерывное перемещение

Если игрок удерживает клавишу →, корабль должен двигаться вправо до тех пор, пока клавиша не будет отпущена. Чтобы узнать, когда клавиша → будет отпущена, наша игра отслеживает событие `pygame.KEYUP`; таким образом, реализация непрерывного движения будет основана на отслеживании событий `KEYDOWN` и `KEYUP` в сочетании с флагом `moving_right`.

В неподвижном состоянии корабля флаг `moving_right` равен `False`. При нажатии клавиши → флагу присваивается значение `True`, а когда клавиша будет отпущена, флаг возвращается в состояние `False`.

Класс `Ship` управляет всеми атрибутами корабля, и мы добавим в него атрибут с именем `moving_right` и метод `update()` для проверки состояния флага `moving_right`. Метод `update()` изменяет позицию корабля, если флаг содержит значение `True`. Этот метод будет вызываться каждый раз, когда вы хотите обновить позицию корабля.

Ниже приведены изменения в классе `Ship`:

ship.py

```
  class Ship():
      """ Класс для управления кораблем"""

      def __init__(self, ai_game):
          ...
          # Каждый новый корабль появляется у нижнего края экрана.
          self.rect.midbottom = self.screen_rect.midbottom

          # Флаг перемещения
❶         self.moving_right = False

❷     def update(self):
          """Обновляет позицию корабля с учетом флага."""
          if self.moving_right:
              self.rect.x += 1

      def blitme(self):
          ...
```

Мы добавляем атрибут `self.moving_right` в метод `__init__()` и инициализируем его значением `False` ❶. Затем вызывается метод `update()`, который перемещает корабль вправо, если флаг равен `True` ❷. Метод `update()` будет вызываться через экземпляр `Ship`, поэтому он не считается вспомогательным методом.

Теперь внесем изменения в `run_game()`, чтобы при нажатии клавиши → `moving_right` присваивалось значение `True`, а при ее отпускании — `False`:

alien_invasion.py

```
    def _check_events(ship):
        """Обрабатывает нажатия клавиш и события мыши."""
        for event in pygame.event.get():
            ...
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RIGHT:
❶                   self.ship.moving_right = True
❷           elif event.type == pygame.KEYUP:
                if event.key == pygame.K_RIGHT:
                    self.ship.moving_right = False
```

В точке ❶ изменяется реакция игры при нажатии клавиши →; вместо непосредственного изменения позиции корабля программа просто присваивает `moving_right` значение `True`. В точке ❷ добавляется новый блок `elif`, реагирующий на события `KEYUP`. Когда игрок отпускает клавишу → (`K_RIGHT`), `moving_right` присваивается значение `False`.

Остается изменить цикл `while` в alien_invasion.py, чтобы при каждом проходе цикла вызывался метод `update()` корабля:

alien_invasion.py

```
def run_game(self):
    # Запуск основного цикла игры.
    while True:
        self._check_events()
        self.ship.update()
        self._update_screen()
```

Позиция корабля будет обновляться после проверки событий клавиатуры, но перед обновлением экрана. Таким образом, позиция корабля обновляется в ответ на действия пользователя и будет использоваться при перерисовке корабля на экране.

Если запустить alien_invasion.py и удерживать клавишу →, корабль непрерывно двигается вправо, пока клавиша не будет отпущена.

Перемещение влево и вправо

Теперь, когда мы реализовали непрерывное движение вправо, добавить движение влево относительно несложно. Для этого нужно снова изменить класс `Ship` и метод

_check_events(). Ниже приведены необходимые изменения в __init__() и update() в классе Ship:

ship.py

```
    def __init__(self, ai_game):
        ...
        # Флаги перемещения
        self.moving_right = False
        self.moving_left = False

    def update(self):
        """Обновляет позицию корабля с учетом флагов."""
        if self.moving_right:
            self.rect.x += 1
        if self.moving_left:
            self.rect.x -= 1
```

В методе __init__() добавляется флаг self.moving_left. В update() используются два отдельных блока if вместо elif, чтобы при нажатии обеих клавиш со стрелками атрибут rect.x сначала увеличивался, а потом уменьшался. В результате корабль остается на месте. Если бы для движения влево использовался блок elif, то клавиша → всегда имела бы приоритет. Такая реализация повышает точность перемещения при переключении направления, когда игрок может ненадолго удерживать нажатыми обе клавиши.

В _check_events() необходимо внести два изменения:

alien_invasion.py

```
def _check_events(self):
    """Обрабатывает нажатия клавиш и события мыши."""
    for event in pygame.event.get():
        ...
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                self.ship.moving_right = True
            elif event.key == pygame.K_LEFT:
                self.ship.moving_left = True

        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_RIGHT:
                self.ship.moving_right = False
            elif event.key == pygame.K_LEFT:
                self.ship.moving_left = False
```

Если событие KEYDOWN происходит для события K_LEFT, то moving_left присваивается True. Если событие KEYUP происходит для события K_LEFT, то moving_left присваивается False. Здесь возможно использовать блоки elif, потому что каждое событие связано только с одной клавишей. Если же игрок нажимает обе клавиши одновременно, то программа обнаруживает два разных события.

Если вы запустите alien_invasion.py, то увидите, что корабль может непрерывно двигаться влево и вправо. Если же нажать обе клавиши, корабль останавливается.

Следующий шаг — доработка движения корабля. Внесем изменения в скорость и ограничим величину перемещения, чтобы корабль не выходил за края экрана.

Регулировка скорости корабля

В настоящий момент корабль смещается на 1 пиксел за каждый проход цикла `while`, но для повышения точности управления скоростью можно добавить в класс `Settings` атрибут `ship_speed`. Этот атрибут определяет величину смещения корабля при каждом проходе цикла. Новый атрибут settings.py выглядит так:

settings.py

```
class Settings():
    """Класс для хранения всех настроек игры Alien Invasion."""

    def __init__(self):
        ...

        # Настройки корабля
        self.ship_speed = 1.5
```

Переменной `ship_speed_factor` присваивается значение 1.5. При перемещении корабля его позиция изменяется на 1,5 пиксела вместо 1.

Дробные значения скорости позволят лучше управлять скоростью корабля при последующем повышении темпа игры. Однако атрибуты прямоугольников (такие, как `x`) принимают только целочисленные значения, поэтому в `Ship` необходимо внести ряд изменений:

ship.py

```
  class Ship():
      """Класс для управления кораблем."""

❶     def __init__(self, ai_game):
          """Инициализирует корабль и задает его начальную позицию."""
          self.screen = ai_game.screen
          self.settings = ai_game.settings
          ...

          # Каждый новый корабль появляется у нижнего края экрана
          ...

          # Сохранение вещественной координаты центра корабля.
❷         self.x = float(self.rect.x)

          # Флаги перемещения
          self.moving_right = False
          self.moving_left = False

      def update(self):
          """Обновляет позицию корабля с учетом флагов."""
          # Обновляется атрибут x, не rect.
          if self.moving_right:
```

```
❸           self.x += self.settings.ship_speed
        if self.moving_left:
            self.x -= self.settings.ship_speed

        # Обновление атрибута rect на основании self.x.
❹       self.rect.x = self.x

    def blitme(self):
        ...
```

В точке ❶ в `Ship` создается атрибут `settings`, чтобы он мог использоваться в `update()`. Так как позиция корабля изменяется с нецелым приращением пикселов, ее следует присвоить переменной, способной хранить дробные значения. Формально атрибутам `rect` можно присвоить дробные значения, но `rect` сохранит только целую часть этого значения. Для точного хранения позиции корабля определяется новый атрибут `self.x`, способный хранить дробные значения ❷. Функция `float()` используется для преобразования значения `self.rect.x` в вещественный формат и сохранения этого значения в `self.x`.

После изменения позиции корабля в `update()` значение `self.x` изменяется на величину, хранящуюся в `settings.ship_speed` ❸. После обновления `self.x` новое значение используется для обновления атрибута `self.rect.x`, управляющего позицией корабля ❹. В `self.rect.x` будет сохранена только целая часть `self.x`, но для отображения корабля этого достаточно.

Теперь можно изменить значение `ship_speed`; при любом значении, большем 1, корабль начинает двигаться быстрее. Эта возможность ускорит реакцию корабля на действия игрока, а также позволит нам изменить темп игры с течением времени.

ПРИМЕЧАНИЕ Если вы используете macOS, может оказаться, что корабль двигается очень медленно даже при высоких значениях скорости. Проблема решается запуском игры в полноэкранном режиме, который мы вскоре реализуем.

Ограничение перемещений

Если удерживать какую-нибудь клавишу со стрелкой достаточно долго, корабль выйдет за край экрана. Давайте сделаем так, чтобы корабль останавливался при достижении края экрана. Задача решается изменением метода `update()` в классе `Ship`:

ship.py

```
    def update(self):
        """Обновляет позицию корабля с учетом флагов."""
        # Обновляется атрибут x объекта ship, не rect.
❶       if self.moving_right and self.rect.right < self.screen_rect.right:
            self.x += self.settings.ship_speed
❷       if self.moving_left and self.rect.left > 0:
            self.x -= self.settings.ship_speed

        # Обновление атрибута rect на основании self.x
        self.rect.x = self.x
```

Этот код проверяет позицию корабля перед изменением значения `self.x`. Выражение `self.rect.right` возвращает координату `x` правого края прямоугольника корабля. Если это значение меньше значения, возвращаемого `self.screen_rect.right`, значит, корабль еще не достиг правого края экрана ❶. То же относится и к левому краю: если координата `x` левой стороны прямоугольника больше 0, значит, корабль еще не достиг левого края экрана ❷. Проверка гарантирует, что корабль будет оставаться в пределах экрана перед изменением значения `self.x`.

Если вы запустите alien_invasion.py сейчас, то движение корабля будет останавливаться у края экрана. Согласитесь, эффектно: мы всего лишь добавили условную проверку в команду `if`, но все выглядит так, словно у края экрана корабль наталкивается на невидимую стену или силовое поле!

Рефакторинг _check_events()

В ходе разработки метод `_check_events()` будет становиться все длиннее, поэтому мы выделим из `_check_events()` еще два метода: для обработки событий `KEYDOWN` и для обработки событий `KEYUP`:

alien_invasion.py

```
def _check_events(self):
    """Реагирует на нажатие клавиш и события мыши."""
    for event in pygame.event.get():
    if event.type == pygame.QUIT:
        sys.exit()
    elif event.type == pygame.KEYDOWN:
        self._check_keydown_events(event)
    elif event.type == pygame.KEYUP:
        self._check_keyup_events(event)

def _check_keydown_events(self, event):
    """Реагирует на нажатие клавиш."""
    if event.key == pygame.K_RIGHT:
        self.ship.moving_right = True
    elif event.key == pygame.K_LEFT:
        self.ship.moving_left = True

def _check_keyup_events(self, event):
    """Реагирует на отпускание клавиш."""
    if event.key == pygame.K_RIGHT:
        self.ship.moving_right = False
    elif event.key == pygame.K_LEFT:
        self.ship.moving_left = False
```

В программе появились два вспомогательных метода: `_check_keydown_events()` и `_check_keyup_events()`. Каждый метод получает параметр `self` и параметр `event`. Тела двух методов скопированы из `_check_events()`, а старый код заменен вызовами новых методов. Новая структура кода упрощает метод `_check_events()` и облегчает последующее программирование реакции на действия игрока.

Нажатие клавиши Q для завершения

Итак, теперь программа реагирует на нажатия клавиш, и мы можем добавить еще один способ завершения игры. Было бы утомительно щелкать на кнопке X в верхней части игрового окна каждый раз, когда в игру добавляется новая функциональность, поэтому мы добавим специальную клавишу для завершения игры при нажатии клавиши Q:

alien_invasion.py

```
def _check_keydown_events(self, event):
    ...
    elif event.key == pygame.K_LEFT:
        self.ship.moving_left = True
    elif event.key == pygame.K_q:
        sys.exit()
```

В код `_check_keydown_events()` добавляется новый блок. Теперь в процессе тестирования можно закрыть игру нажатием клавиши Q, вместо того чтобы пользоваться кнопкой закрытия окна.

Запуск игры в полноэкранном режиме

В Pygame поддерживается полноэкранный режим, который, возможно, понравится вам больше запуска в обычном окне. Некоторые игры лучше смотрятся в полноэкранном режиме, а у пользователей macOS в полноэкранном режиме может улучшиться быстродействие.

Чтобы запустить игру в полноэкранном режиме, внесите следующие изменения в `__init__()`:

alien_invasion.py

```
    def __init__(self):
        """Инициализирует игру и создает игровые ресурсы."""
        pygame.init()
        self.settings = Settings()

❶       self.screen = pygame.display.set_mode((0, 0), pygame.FULLSCREEN)
❷       self.settings.screen_width = self.screen.get_rect().width
        self.settings.screen_height = self.screen.get_rect().height
        pygame.display.set_caption("Alien Invasion")
```

При создании экранной поверхности передается размер (0, 0) и параметр `pygame.FULLSCREEN` ❶. Эти значения приказывают Pygame вычислить размер окна, заполняющего весь экран. Так как ширина и высота экрана неизвестны заранее, эти настройки обновляются после создания экрана ❷. Атрибуты `width` и `height` прямоугольника экрана используются для обновления объекта `settings`.

Если вам понравится, как игра выглядит или работает в полноэкранном режиме, оставьте новые настройки. Если вы предпочитаете, чтобы игра работала в отдель-

ном окне, — вернитесь к исходной реализации с назначением конкретных размеров экрана.

ПРИМЕЧАНИЕ Прежде чем запускать игру в полноэкранном режиме, убедитесь в том, что она закрывается при нажатии клавиши Q; в Pygame не существует стандартных средств завершения игры в полноэкранном режиме.

В двух словах

В следующем разделе мы реализуем стрельбу, для чего нам потребуется новый файл с именем bullet.py и изменения в некоторых уже имеющихся файлах. В настоящее время программа состоит из трех файлов с разными классами и методами. Чтобы вы четко представляли себе структуру проекта, кратко проанализируем каждый из этих файлов перед добавлением новой функциональности.

alien_invasion.py

Главный файл программы alien_invasion.py содержит класс `AlienInvasion`. Этот класс содержит ряд важных атрибутов, используемых в процессе игры: настройки хранятся в `settings`, основная поверхность для вывода изображения хранится в `screen`, а экземпляр `ship` тоже создается в этом файле. Также в alien_invasion.py содержится главный цикл игры — цикл `while` с вызовами `_check_events()`, `ship.update()` и `_update_screen()`.

Метод `_check_events()` обнаруживает важные события (например, нажатия и отпускания клавиш) и обрабатывает все эти типы событий с использованием методов `_check_keydown_events()` и `_check_keyup_events()`. Пока эти методы управляют движением корабля. Класс `AlienInvasion` также содержит метод `_update_screen()`, который перерисовывает экран при каждом проходе основного цикла.

Файл alien_invasion.py — единственный файл, который должен запускаться для игры в Alien Invasion. Все остальные файлы — settings.py и ship.py — содержат код, который импортируется в этот файл.

settings.py

Файл settings.py содержит класс `Settings`. Этот класс содержит только метод `__init__()`, инициализирующий атрибуты, которые управляют внешним видом и скоростью игры.

ship.py

Файл ship.py содержит класс `Ship`. В этом классе определен метод `__init__()`, метод `update()` для управления позицией корабля и метод `blitme()` для вывода

изображения корабля на экран. Изображение корабля хранится в файле ship.bmp, который находится в папке images.

УПРАЖНЕНИЯ

12.3. Документация Pygame: разработка игры зашла уже достаточно далеко, и вам стоит просмотреть документацию Pygame. Домашняя страница Pygame находится по адресу https://www.pygame.org/, а домашняя страница документации — по адресу https://www.pygame.org/docs/. Пока вы можете ограничиться простым просмотром документации. Она не понадобится вам для завершения этого проекта, но пригодится, если вы захотите внести изменения в игру или займетесь созданием собственной игры.

12.4. Ракета: создайте игру, в которой в исходном состоянии в центре экрана находится ракета. Игрок может перемещать ракету вверх, вниз, вправо и влево четырьмя клавишами со стрелками. Проследите за тем, чтобы ракета не выходила за края экрана.

12.4. Клавиши: создайте файл Pygame, который создает пустой экран. В цикле событий выводите значение атрибута `event.key` при обнаружении события `pygame.KEYDOWN`. Запустите программу, нажимайте различные клавиши и понаблюдайте за реакцией Pygame.

Стрельба

А теперь добавим в игру функциональность стрельбы. Мы напишем код, который выпускает снаряд (маленький прямоугольник) при нажатии игроком клавиши «пробел». Снаряды летят вертикально вверх, пока не исчезнут у верхнего края экрана.

Добавление настроек

Сначала добавим в settings.py новые настройки для значений, управляющих поведением класса `Bullet`. Эти настройки добавляются в конец метода `__init__()`:

settings.py

```
def __init__(self):
    ...
    # Параметры снаряда
    self.bullet_speed = 1
    self.bullet_width = 3
    self.bullet_height = 15
    self.bullet_color = (60, 60, 60)
```

Эти настройки создают темно-серые снаряды шириной 3 пиксела и высотой 15 пикселов. Они двигаются немного медленнее, чем корабль.

Создание класса Bullet

Теперь создадим файл bullet.py для хранения класса `Bullet`. Первая часть файла bullet.py выглядит так:

bullet.py

```
import pygame
from pygame.sprite import Sprite

class Bullet(Sprite):
    """Класс для управления снарядами, выпущенными кораблем."""

    def __init__(self, ai_game):
        """Создает объект снарядов в текущей позиции корабля."""
        super().__init__()
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        self.color = self.settings.bullet_color

        # Создание снаряда в позиции (0,0) и назначение правильной позиции.
❶       self.rect = pygame.Rect(0, 0, self.settings.bullet_width,
            self.settings.bullet_height)
❷       self.rect.midtop = ai_game.ship.rect.midtop

        # Позиция снаряда хранится в вещественном формате.
❸       self.y = float(self.rect.y)
```

Класс `Bullet` наследует от класса `Sprite`, импортируемого из модуля `pygame.sprite`. Работая со *спрайтами* (sprite), разработчик группирует связанные элементы в своей игре и выполняет операции со всеми сгруппированными элементами одновременно. Чтобы создать экземпляр снаряда, методу `__init__()` необходим текущий экземпляр `AlienInvasion`, а вызов `super()` необходим для правильной реализации наследования от `Sprite`. Также задаются атрибуты для объектов экрана и настроек, а также цвета снаряда.

В точке ❶ создается атрибут `rect` снаряда. Снаряд не создается на основе готового изображения, поэтому прямоугольник приходится строить с нуля при помощи класса `pygame.Rect()`. При создании экземпляра этого класса необходимо задать координаты левого верхнего угла прямоугольника, его ширину и высоту. Прямоугольник инициализируется в точке (0, 0), но в следующих двух строках он перемещается в нужное место, так как позиция снаряда зависит от позиции корабля. Ширина и высота снаряда определяются значениями, хранящимися в `self.settings`.

В точке ❷ атрибуту `midtop` снаряда присваивается значение `midtop` корабля. Снаряд должен появляться у верхнего края корабля, поэтому верхний край снаряда совмещается с верхним краем прямоугольника корабля для имитации выстрела из корабля ❸.

А вот как выглядит вторая часть bullet.py, `update()` и `draw_bullet()`:

bullet.py

```
    def update(self):
        """Перемещает снаряд вверх по экрану."""
        # Обновление позиции снаряда в вещественном формате.
❶       self.y -= self.settings.bullet_speed
        # Обновление позиции прямоугольника.
```

```
❷         self.rect.y = self.y

      def draw_bullet(self):
          """Вывод снаряда на экран."""
❸         pygame.draw.rect(self.screen, self.color, self.rect)
```

Метод `update()` управляет позицией снаряда. Когда происходит выстрел, снаряд двигается вверх по экрану, что соответствует уменьшению координаты `y`; следовательно, для обновления позиции снаряда следует вычесть величину, хранящуюся в `settings.bullet_speed`, из `self.y` ❶. Затем значение `self.y` используется для изменения значения `self.rect.y` ❷.

Атрибут `bullet_speed` позволяет увеличить скорость снарядов по ходу игры или при изменении ее поведения. Координата `x` снаряда после выстрела не изменяется, поэтому снаряд летит вертикально по прямой линии.

Для вывода снаряда на экран вызывается функция `draw_bullet()`. Функция `draw_rect()` заполняет часть экрана, определяемую прямоугольником снаряда, цветом из `self.color` ❸.

Группировка снарядов

Класс `Bullet` и все необходимые настройки готовы; можно переходить к написанию кода, который будет выпускать снаряд каждый раз, когда игрок нажимает клавишу «пробел». Сначала мы создадим в `AlienInvasion` группу для хранения всех летящих снарядов, чтобы программа могла управлять их полетом. Эта группа будет представлена экземпляром класса `pygame.sprite.Group` — своего рода списком с расширенной функциональностью, которая может быть полезна при построении игр. Мы воспользуемся группой для прорисовки снарядов на экране при каждом проходе основного цикла и обновления текущей позиции каждого снаряда.

Группа будет создаваться в `__init__()`:

alien_invasion.py

```
def __init__(self):
    ...
    self.ship = Ship(self)
    self.bullets = pygame.sprite.Group()
```

Позиция снаряда будет обновляться при каждом проходе цикла `while`:

alien_invasion.py

```
  def run_game():
      """Запуск основного цикла игры."""
      while True:
          self._check_events()
          self.ship.update()
❶         self.bullets.update()
          self._update_screen()
```

Вызов `update()` для группы ❶ приводит к автоматическому вызову `update()` для каждого спрайта в группе. Строка `self.bullets.update()` вызывает `bullet.update()` для каждого снаряда, включенного в группу `bullets`.

Обработка выстрелов

В классе `AlienInvasion` необходимо внести изменения в метод `_check_keydown_events()`, чтобы при нажатии клавиши «пробел» происходил выстрел. Изменять `_check_keyup_events()` не нужно, потому что при отпускании клавиши ничего не происходит. Также необходимо изменить `_update_screen()` и вывести каждый снаряд на экран перед вызовом `flip()`.

При обработке выстрела придется выполнить довольно значительную работу, для которой мы напишем новый метод `fire_bullet()`:

alien_invasion.py

```
  ...
  from ship import Ship
❶ from bullet import Bullet

  class AlienInvasion:
      ...
      def _check_keydown_events(self, event):
          ...
          elif event.key == pygame.K_q:
              sys.exit()
❷         elif event.key == pygame.K_SPACE:
              self._fire_bullet()

      def _check_keyup_events(self, event):
          ...

      def _fire_bullet(self):
          """Создание нового снаряда и включение его в группу bullets."""
❸         new_bullet = Bullet(self)
❹         self.bullets.add(new_bullet)

      def _update_screen(self):
          """Обновляет изображения на экране и отображает новый экран."""
          self.screen.fill(self.settings.bg_color)
          self.ship.blitme()
❺         for bullet in self.bullets.sprites():
              bullet.draw_bullet()
      pygame.display.flip()
  ...
```

Сначала импортируется класс `Bullet` ❶. Затем при нажатии клавиши «пробел» вызывается `_fire_bullet()` ❷. В коде `_fire_bullet()` мы создаем экземпляр `Bullet`, которому присваивается имя `new_bullet` ❸. Он включается в группу `bullets` вызовом метода `add()` ❹. Метод `add()` похож на `append()`, но этот метод написан специально для групп Pygame.

Метод `bullets.sprites()` возвращает список всех спрайтов в группе `bullets`. Чтобы нарисовать все выпущенные снаряды на экране, программа перебирает спрайты в `bullets` и вызывает для каждого `draw_bullet()` ❺.

Если запустить alien_invasion.py сейчас, вы сможете двигать корабль влево и вправо и выпускать сколько угодно снарядов. Снаряды перемещаются вверх по экрану и исчезают при достижении верхнего края (рис. 12.3). Размер, цвет и скорость можно изменить при помощи настроек в settings.py.

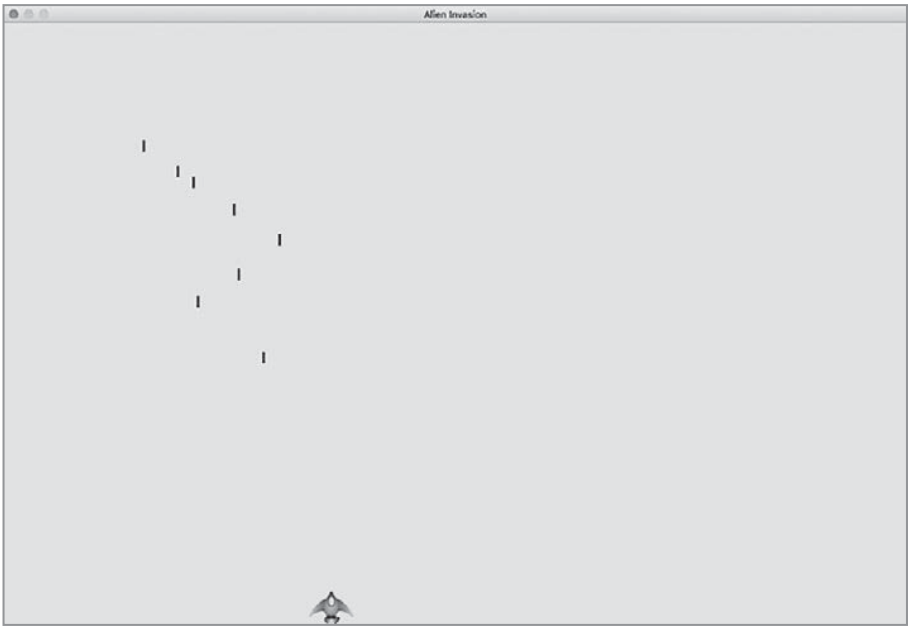

Рис. 12.3. Экран игры после серии выстрелов

Удаление старых снарядов

На данный момент снаряды исчезают при достижении верхнего края, но только потому, что Pygame не может нарисовать их выше края экрана. На самом деле снаряды продолжают существовать; их координата `y` продолжает уменьшаться. И это создает проблему, потому что снаряды продолжают потреблять память и вычислительные мощности.

От старых снарядов необходимо избавиться, иначе игра замедлится из-за большого объема лишней работы. Для этого необходимо определить момент, когда атрибут `bottom` прямоугольника снаряда достигнет 0 — это означает, что снаряд вышел за верхний край экрана:

alien_invasion.py

```
  def run_game(self):
      # Запуск основного цикла игры.
      while True:
          self._check_events()
          self.ship.update()
          self.bullets.update()

          # Удаление снарядов, вышедших за край экрана.
❶         for bullet in self.bullets.copy():
❷             if bullet.rect.bottom <= 0:
❸                 self.bullets.remove(bullet)
❹         print(len(self.bullets))

          self._update_screen()
```

При использовании цикла `for` со списком (или группой в Pygame) Python ожидает, что список сохраняет постоянную длину во время выполнения цикла. Так как элементы из списка или группы в цикле `for` не должны удаляться, перебирать нужно копию группы. Метод `copy()` используется для создания цикла `for` ❶, в котором возможно изменять содержимое `bullets`. Программа проверяет каждый снаряд и определяет, вышел ли он за верхний край экрана ❷. Если снаряд пересек границу, он удаляется из `bullets` ❸. В точке ❹ добавляется команда `print`, которая сообщает, сколько снарядов сейчас существует в игре; по выведенному значению можно убедиться в том, что снаряды действительно удаляются при достижении верхнего края экрана.

Если код работает правильно, вы можете понаблюдать за выводом на терминале и убедиться в том, что количество снарядов уменьшается до 0 после того, как очередной залп уходит за верхний край экрана. После запуска игры, когда вы убедитесь в том, что снаряды правильно удаляются из группы, удалите команду `print`. Если команда останется в программе, она существенно замедлит игру, потому что вывод на терминал занимает больше времени, чем отображение графики в игровом окне.

Ограничение количества снарядов

Многие игры-стрелялки ограничивают количество снарядов, одновременно находящихся на экране, чтобы у игроков появился стимул стрелять более метко. То же самое будет сделано и в игре Alien Invasion.

Сначала сохраним максимально допустимое количество снарядов в settings.py:

settings.py

```
# Параметры снаряда
...
self.bullet_color = (60, 60, 60)
self.bullets_allowed = 3
```

В любой момент времени на экране может находиться не более трех снарядов. Эта настройка будет использоваться в `AlienInvasion` для проверки количества существующих снарядов перед созданием нового снаряда в `_fire_bullet()`:

alien_invasion.py

```
def _fire_bullet(self):
    """Создание нового снаряда и включение его в группу bullets."""
    if len(self.bullets) < self.settings.bullets_allowed:
        new_bullet = Bullet(self)
        self.bullets.add(new_bullet).
```

При нажатии клавиши «пробел» программа проверяет длину `bullets`. Если значение `len(self.bullets)` меньше трех, создается новый снаряд. Но если на экране уже находятся три активных снаряда, при нажатии пробела ничего не происходит. Если вы запустите игру сейчас, вы сможете выпускать снаряды только группами по три.

Создание метода _update_bullets()

Мы хотим, чтобы класс `AlienInvasion` был как можно более простым, поэтому после написания и проверки кода управления снарядами этот код можно переместить в отдельный метод. Мы создадим новый метод `_update_bullets()` и добавим его непосредственно перед `_update_screen()`:

alien_invasion.py

```
def _update_bullets(self):
    """Обновляет позиции снарядов и уничтожает старые снаряды."""
    # Обновление позиций снарядов.
    self.bullets.update()

    # Удаление снарядов, вышедших за край экрана.
    for bullet in self.bullets.copy():
        if bullet.rect.bottom <= 0:
            self.bullets.remove(bullet)
```

Код `_update_bullets()` вырезается и вставляется из `run_game()`; мы всего лишь немного уточнили комментарии.

Цикл `while` в `run_game()` снова выглядит просто:

alien_invasion.py

```
while True:
    self._check_events()
    self.ship.update()
    self._update_bullets()
    self._update_screen()
```

В результате преобразования основной цикл содержит минимум кода, чтобы можно было легко прочитать имена функций и понять, что происходит в игре. Основной цикл проверяет ввод, полученный от игрока, а затем обновляет позицию корабля

и всех выпущенных снарядов. Затем обновленные позиции игровых элементов используются для вывода нового экрана в точке.

Снова запустите alien_invasion.py и убедитесь в том, что стрельба происходит без ошибок.

УПРАЖНЕНИЯ

12.6. Боковая стрельба: напишите игру, в которой корабль размещается у левого края экрана, а игрок может перемещать корабль вверх и вниз. При нажатии клавиши «пробел» корабль стреляет и снаряд двигается вправо по экрану. Проследите за тем, чтобы снаряды удалялись при выходе за край экрана.

Итоги

В этой главе вы научились планировать ход игры, а также усвоили базовую структуру игры, написанной с использованием Pygame. Вы узнали, как задать цвет фона и как сохранить настройки в отдельном классе, чтобы они были доступны для всех частей игры. Вы научились выводить изображения на экран и управлять перемещением игровых элементов. Также вы узнали, как создавать элементы, двигающиеся самостоятельно (например, снаряды, летящие по экрану), и как удалять объекты, которые стали лишними. Также в этой главе рассматривалась методика регулярного рефакторинга кода для упрощения текущей разработки.

В главе 13 в игру Alien Invasion будут добавлены пришельцы. К концу главы 13 игрок сможет сбивать корабли пришельцев — конечно, если они не доберутся до него первыми!

13 Осторожно, пришельцы!

В этой главе в игру Alien Invasion будут добавлены пришельцы. Сначала мы добавим одного пришельца у верхнего края экрана, а потом сгенерируем целый флот. Пришельцы будут перемещаться в сторону и вниз; при этом пришельцы, в которых попадают снаряды, исчезают с экрана. Наконец, мы ограничим количество кораблей у игрока, так что при гибели последнего корабля игра завершается.

В этой главе вы узнаете больше о Pygame и о ведении крупного проекта. Вы также научитесь обнаруживать *коллизии* (столкновения) игровых объектов, например снарядов и пришельцев. Обнаружение коллизий помогает определять взаимодействия между элементами игры: например, ограничить перемещение персонажа областью между стенами лабиринта или организовать передачу мяча между двумя персонажами. Работа будет продолжаться на основе плана, к которому мы будем время от времени возвращаться, чтобы не отклоняться от цели во время написания кода.

Прежде чем браться за новый код для добавления флота пришельцев на экран, рассмотрим проект и обновим план.

Анализ проекта

Приступая к новой фазе разработки крупного проекта, всегда полезно вернуться к исходному плану и уточнить, чего же вы хотите добиться в том коде, который собираетесь написать. В этой главе мы:

- проанализируем код и определим, нужно ли провести рефакторинг перед реализацией новых возможностей;
- добавим в левом верхнем углу экрана одного пришельца, отделив его от краев экрана интервалами;
- по величине интервалов вокруг первого пришельца и общим размерам экрана вычислим, сколько пришельцев поместится на экране. Для создания пришельцев, заполняющих верхнюю часть экрана, будет написан цикл;
- организуем перемещение флота пришельцев в сторону и вниз, пока весь флот не будет уничтожен, или пока пришелец не столкнется с кораблем игрока, или

пока пришелец не достигнет земли. Если весь флот будет уничтожен, программа создает новый флот. Если пришелец сталкивается с кораблем или с землей, программа уничтожает корабль и создает новый флот;

- ограничим количество кораблей, которые могут использоваться игроком, и завершаем игру в конце последней попытки.

Этот план будет уточняться по мере реализации новых возможностей, но для начала и этого достаточно.

Также проводите анализ кода, когда вы начинаете работу над новой серией возможностей проекта. Так как с каждой новой фазой проект обычно становится более сложным, лучше всего заняться расчисткой излишне громоздкого или неэффективного кода. Ранее мы уже проводили рефакторинг, так что сейчас особой расчистки не потребуется.

Создание пришельца

Размещение одного пришельца на экране мало чем отличается от размещения корабля. Поведением каждого пришельца будет управлять класс с именем `Alien`, который по своей структуре очень похож на класс `Ship`. Для простоты мы снова воспользуемся готовыми графическими изображениями. Вы можете найти собственное изображение пришельца или использовать изображение на рис. 13.1, доступное в ресурсах книги по адресу https://www.nostarch.com/pythoncrashcourse2e/. Это изображение имеет серый фон, совпадающий с цветом фона экрана. Не забудьте сохранить выбранный файл в каталоге images.

Рис. 13.1. Пришелец, который будет использоваться для создания флота

Создание класса Alien

Теперь можно написать класс `Alien` и сохранить его в файле alien.py:

alien.py

```
import pygame
from pygame.sprite import Sprite

class Alien(Sprite):
    """Класс, представляющий одного пришельца."""

    def __init__(self, ai_game):
        """Инициализирует пришельца и задает его начальную позицию."""
        super().__init__()
        self.screen = ai_game.screen

        # Загрузка изображения пришельца и назначение атрибута rect.
        self.image = pygame.image.load('images/alien.bmp')
        self.rect = self.image.get_rect()

        # Каждый новый пришелец появляется в левом верхнем углу экрана.
❶       self.rect.x = self.rect.width
        self.rect.y = self.rect.height

        # Сохранение точной горизонтальной позиции пришельца.
❷       self.x = float(self.rect.x)
```

В основном этот класс похож на класс `Ship` (если не считать размещения пришельца). Изначально каждый пришелец размещается в левом верхнем углу экрана, при этом слева от него добавляется интервал, равный ширине пришельца, а над ним — интервал, равный высоте ❶. Нас в первую очередь интересует горизонтальная скорость пришельца, поэтому горизонтальная позиция каждого пришельца отслеживается точно ❷.

Классу `Alien` не нужен метод для вывода на экран; вместо этого мы воспользуемся методом групп Pygame, который автоматически рисует все элементы группы на экране.

Создание экземпляра Alien

Начнем с создания экземпляра `Alien`, чтобы первый пришелец появился на экране. Так как эта операция входит в подготовительную часть, код для текущего экземпляра будет добавлен в конец метода `__init__()` в `AlienInvasion`. Позднее будет создан целый флот вторжения, что потребует определенной работы, поэтому мы определим новый вспомогательный метод с именем `_create_fleet()`.

Порядок следования методов в классе может быть любым — важно лишь, чтобы в этом порядке существовала некая закономерность. Я размещу `_create_fleet()` непосредственно перед методом `_update_screen()`, но с таким же успехом его можно разместить в любой точке `AlienInvasion`. Начнем с импортирования класса `Alien`.

Обновленные команды импортирования в файле alien_invasion.py:

alien_invasion.py

```
...
from bullet import Bullet
from alien import Alien
```

Обновленный метод `__init__()`:

alien_invasion.py

```
def __init__(self):
    ...
    self.ship = Ship(self)
    self.bullets = pygame.sprite.Group()
    self.aliens = pygame.sprite.Group()

    self._create_fleet()
```

Создадим группу для хранения флота вторжения и вызовем метод `_create_fleet()`, который будет вскоре написан.

Новый метод `_create_fleet()` выглядит так:

alien_invasion.py

```
    def _create_fleet(self):
        """Создание флота вторжения."""
        # Создание пришельца.
        alien = Alien(self)
        self.aliens.add(alien)
```

В этом методе создается один экземпляр `Alien`, который затем добавляется в группу для хранения флота. По умолчанию объект размещается в левом верхнем углу экрана — эта позиция прекрасно подходит для первого пришельца.

Чтобы пришелец появился на экране, программа вызывает метод `draw()` группы в `_update_screen()`:

alien_invasion.py

```
    def _update_screen(self):
        ...
        for bullet in self.bullets.sprites():
            bullet.draw_bullet()
        self.aliens.draw(self.screen)

        pygame.display.flip()
```

При вызове `draw()` для группы Pygame выводит каждый элемент группы в позиции, определяемой его атрибутом `rect`. Метод `draw()` получает один аргумент — поверхность для вывода элементов группы. На рис. 13.2 изображен первый пришелец.

Рис. 13.2. Появился первый пришелец

После того как первый пришелец появится на экране, мы напишем код для вывода всего флота.

Построение флота

Чтобы нарисовать флот пришельцев, необходимо вычислить, сколько пришельцев поместится в одном ряду и сколько рядов поместится по высоте. Сначала мы вычислим горизонтальные интервалы между пришельцами и создадим ряд; затем будет вычислен вертикальный интервал и создан весь флот.

Вычисление количества пришельцев в одном ряду

Чтобы определить, сколько пришельцев помещается в одном ряду, сначала вычислим доступное горизонтальное пространство. Ширина экрана хранится в `settings.screen_width`, но с обеих сторон экрана необходимо зарезервировать пустые интервалы. Определим их равными ширине одного пришельца. Так как ширина уменьшается на величину двух интервалов, доступное пространство равно ширине экрана за вычетом удвоенной ширины пришельца:

```
available_space_x = settings.screen_width – (2 * alien_width)
```

Также необходимо зарезервировать интервалы между пришельцами; они будут составлять одну ширину пришельца. Пространство, необходимое для вывода одного

пришельца, равно его удвоенной ширине: одна ширина для самого пришельца и еще одна для пустого интервала справа. Чтобы определить количество пришельцев на экране, разделим доступное пространство на удвоенную ширину пришельца. При этом будет использоваться *целочисленное деление* `//` с потерей остатка, чтобы полученное количество пришельцев было целым:

```
number_aliens_x = available_space_x // (2 * alien_width)
```

Эти вычисления будут включены в программу при создании флота.

ПРИМЕЧАНИЕ У вычислений в программировании есть одна замечательная особенность: не обязательно быть полностью уверенными в правильности формулы, когда вы ее пишете. Вы можете опробовать формулу на практике и посмотреть, что из этого получится. В худшем случае получится экран, до отказа забитый пришельцами или, наоборот, пустой. В этом случае вы пересмотрите формулу на основании полученных результатов.

Создание ряда

Все готово к тому, чтобы сгенерировать полный ряд пришельцев. Так как наш код создания одного пришельца работает правильно, мы перепишем `_create_fleet()` для создания ряда пришельцев:

alien_invasion.py

```
  def _create_fleet(self):
      """Создает флот пришельцев."""
      # Создание пришельца и вычисление количества пришельцев в ряду
      # Интервал между соседними пришельцами равен ширине пришельца.
❶     alien = Alien(self)
❷     alien_width = alien.rect.width
❸     available_space_x = self.settings.screen_width - (2 * alien_width)
      number_aliens_x = available_space_x // (2 * alien_width)

      # Создание первого ряда пришельцев.
❹     for alien_number in range(number_aliens_x):
          # Создание пришельца и размещение его в ряду.
          alien = Alien(self)
❺         alien.x = alien_width + 2 * alien_width * alien_number
          alien.rect.x = alien.x
          self.aliens.add(alien)
```

Большая часть этого кода уже была описана ранее. Для размещения пришельцев необходимо знать ширину и высоту одного пришельца, и мы создаем его в точке ❶ перед выполнением вычислений. Этот пришелец не войдет во флот, поэтому он не включается в группу `aliens`. В точке ❷ ширина пришельца определяется по его атрибуту `rect`, а полученное значение сохраняется в `alien_width`, чтобы избежать лишних обращений к атрибуту `rect`. В точке ❸ вычисляется доступное горизонтальное пространство и количество пришельцев, которые в нем поместятся.

Затем создается цикл от 0 до количества создаваемых пришельцев ❹. В теле цикла создается новый пришелец, после чего задается его координата `x` для размещения

его в ряду ❺. Каждый пришелец сдвигается вправо на одну ширину от левого поля. Затем ширина пришельца умножается на 2, чтобы учесть полное пространство, выделенное для одного пришельца, включая пустой интервал справа, а полученная величина умножается на позицию пришельца в ряду. Атрибут `x` пришельца используется для назначения позиции его прямоугольника. После этого новый пришелец добавляется в группу `aliens`.

Запустив программу Alien Invasion, вы увидите, что на экране появился первый ряд пришельцев (рис. 13.3).

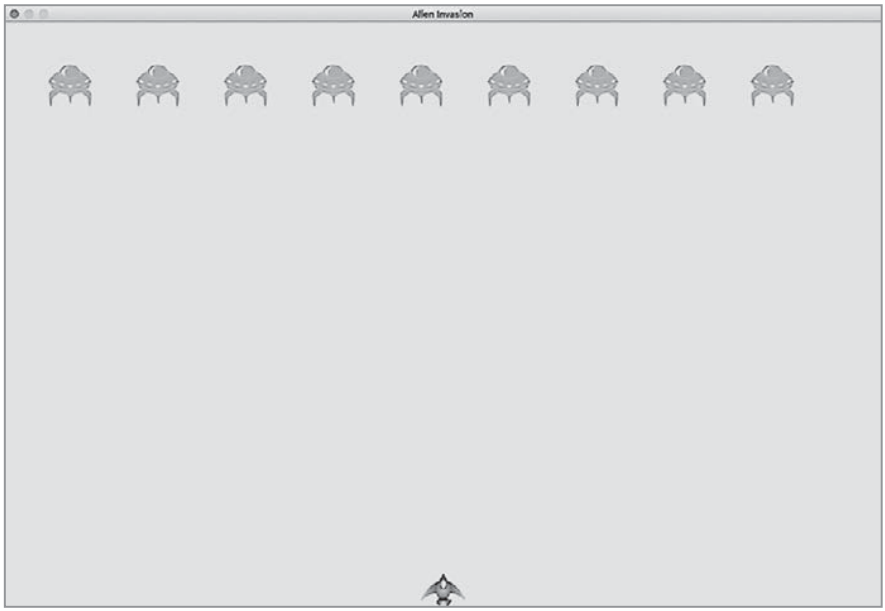

Рис. 13.3. Первый ряд пришельцев

Первый ряд сдвинут влево, и это полезно для игрового процесса, потому что флот пришельцев должен двигаться вправо, пока не дойдет до края экрана, затем немного опусти́ться вниз, затем двигаться влево и т. д. Как и в классической игре Space Invaders, такое перемещение интереснее, чем постепенное снижение по прямой. Движение будет продолжаться до тех пор, пока все пришельцы не будут сбиты или пока пришелец не столкнется с кораблем либо нижним краем экрана.

ПРИМЕЧАНИЕ В зависимости от выбранной ширины экрана расположение первого ряда пришельцев в вашей системе может выглядеть немного иначе.

Рефакторинг _create_fleet()

Если бы создание флота на этом было завершено, то функцию `_create_fleet()`, пожалуй, можно было бы оставить в таком виде, но работа еще не закончена, по-

этому мы немного подчистим код функции. Добавим новый вспомогательный метод `_create_alien()` и вызовем его из `_create_fleet()`:

alien_invasion.py

```
def _create_fleet(self):
    ...
    # Создание первого ряда пришельцев.
    for alien_number in range(number_aliens_x):
        self._create_alien(alien_number)

def _create_alien(self, alien_number):
    """Создание пришельца и размещение его в ряду."""
    alien = Alien(self)
    alien_width = alien.rect.width
    alien.x = alien_width + 2 * alien_width * alien_number
    alien.rect.x = alien.x
    self.aliens.add(alien)
```

Метод `_create_alien()` должен получать еще один параметр, кроме `self`: номер пришельца, создаваемого в настоящий момент. Мы используем тот же код, созданный для `_create_fleet()`, не считая того, что ширина пришельца определяется внутри метода, а не передается в аргументе. Рефакторинг упрощает добавление новых строк и создание всего флота.

Добавление рядов

Чтобы завершить построение флота, определите количество рядов на экране и повторите цикл (создания пришельцев одного ряда) полученное количество раз. Чтобы определить количество рядов, мы вычисляем доступное вертикальное пространство, вычитая высоту пришельца (сверху), высоту корабля (снизу) и удвоенную высоту пришельца (снизу):

```
available_space_y = settings.screen_height – (3 * alien_height) – ship_height
```

В результате вокруг корабля образуется пустое пространство, чтобы у игрока было время начать стрельбу по пришельцам в начале каждого уровня.

Под каждым рядом должно быть пустое место, равное высоте пришельца. Чтобы вычислить количество строк, мы делим свободное пространство на удвоенную высоту пришельца. Мы снова используем целочисленное деление, потому что количество создаваемых рядов должно быть целым. (Как и прежде, если формула содержит ошибку, мы это немедленно увидим и внесем изменения, пока не получим нужные интервалы):

```
number_rows = available_height_y // (2 * alien_height)
```

Зная количество рядов во флоте, мы можем повторить код создания ряда:

alien_invasion.py

```
  def _create_fleet(self):
      ...
```

```
        alien = Alien(self)
❶       alien_width, alien_height = alien.rect.size
        available_space_x = self.settings.screen_width - (2 * alien_width)
        number_aliens_x = available_space_x // (2 * alien_width)

        """Определяет количество рядов, помещающихся на экране."""
        ship_height = self.ship.rect.height
❷       available_space_y = (self.settings.screen_height -
                             (3 * alien_height) - ship_height)
        number_rows = available_space_y // (2 * alien_height)

        # Создание флота вторжения.
❸       for row_number in range(number_rows):
            for alien_number in range(number_aliens_x):
                self._create_alien(alien_number, row_number)

    def _create_alien(self, alien_number, row_number):
        """Создание пришельца и размещение его в ряду."""
        alien = Alien(self)
        alien_width, alien_height = alien.rect.size
        alien.x = alien_width + 2 * alien_width * alien_number
        alien.rect.x = alien.x
❹       alien.rect.y = alien.rect.height + 2 * alien.rect.height * row_number
        self.aliens.add(alien)
```

Для дальнейших вычислений понадобятся значения ширины и высоты пришельца, поэтому в точке ❶ используется атрибут `size`, который содержит кортеж с шириной и высотой объекта `rect`. Чтобы вычислить количество рядов, помещающихся на экране, мы включаем вычисление `available_space_y` сразу же после вычисления `available_space_x` ❷. Вычисления заключаются в круглые скобки, чтобы их можно было разбить на две строки длиной 79 символов и менее, как указано в рекомендациях.

Чтобы создать несколько рядов, мы используем два вложенных цикла: внешний и внутренний ❸. Внутренний цикл создает один ряд пришельцев. Внешний цикл считает от 0 до количества рядов; Python использует код создания одного ряда и повторяет его `number_rows` раз.

Чтобы создать вложенный цикл, напишите новый цикл `for` и снабдите повторяемый код отступом. (В большинстве текстовых редакторов операции создания и удаления блоков кода выполняются просто, но если вам понадобится помощь, обращайтесь к приложению Б.) Затем при вызове `_create_alien()` передается аргумент с номером ряда, чтобы каждый ряд находился на экране ниже предыдущих.

Определению `_create_alien()` необходим параметр с номером ряда. В `_create_alien()` мы изменяем координату y пришельца, если он не находится в первом ряду ❹. Сначала прибавляется одна высота пришельца, чтобы создать пустое место у верхнего края экрана. Каждый новый ряд начинается на две высоты пришельца ниже последнего ряда, поэтому мы умножаем высоту пришельца на 2, а затем на номер ряда. Номер первого ряда равен 0, так что вертикальное расположение первого ряда остается неизменным. Все последующие ряды размещаются ниже на экране.

Если теперь запустить игру, вы увидите целый флот пришельцев (рис. 13.4).

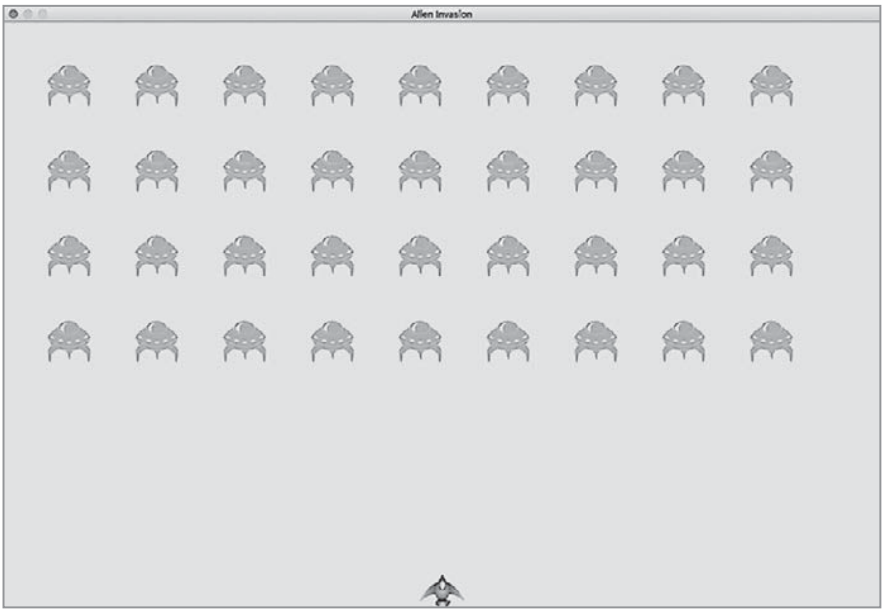

Рис. 13.4. На экране появился весь флот пришельцев

В следующем разделе мы приведем флот в движение.

УПРАЖНЕНИЯ

13.1. Звезды: найдите изображение звезды. Создайте на экране сетку из звезд.

13.2. Звезды-2: чтобы звезды выглядели более реалистично, следует внести случайное отклонение при размещении звезд. Вспомните, что случайные числа генерируются следующим образом:

```
from random import randint
random_number = randint(-10,10)
```

Этот код возвращает случайное целое число в диапазоне от 10 до 10. Используя свой код из упражнения 13.1, измените позицию каждой звезды на случайную величину.

Перемещение флота

Флот пришельцев должен двигаться вправо по экрану, пока не дойдет до края; тогда флот опускается на заданную величину и начинает двигаться в обратном направлении. Это продолжается до тех пор, пока все пришельцы не будут сбиты, один из них столкнется с кораблем или не достигнет низа экрана. Начнем с перемещения флота вправо.

Перемещение вправо

Чтобы корабли пришельцев перемещались по экрану, мы воспользуемся методом `update()` из alien.py, который будет вызываться для каждого пришельца в группе. Сначала добавим настройку для управления скоростью каждого пришельца:

settings.py

```
def __init__(self):
    ...
    # Настройки пришельцев
    self.alien_speed = 1.0
```

Настройка используется в реализации `update()`:

alien.py

```
  def __init__(self, ai_game):
      """Инициализирует пришельца и задает его начальную позицию."""
      super().__init__()
      self.screen = ai_game.screen
      self.settings = ai_game.settings
      ...

  def update(self):
      """Перемещает пришельца вправо."""
❶     self.x += self.settings.alien_speed
❷     self.rect.x = self.x
```

Параметр `settings` создается в `__init__()`, чтобы к скорости пришельца можно было обратиться в `update()`. При каждом обновлении позиции пришельца мы смещаем его вправо на величину, хранящуюся в `alien_speed`. Точная позиция пришельца хранится в атрибуте `self.x`, который может принимать вещественные значения ❶. Затем значение `self.x` используется для обновления позиции прямоугольника пришельца ❷.

В основном цикле `while` уже содержатся вызовы обновления корабля и снарядов. Теперь необходимо также обновить позицию каждого пришельца:

alien_invasion.py

```
while True:
    self._check_events()
    self.ship.update()
    self._update_bullets()
    self._update_aliens()
    self._update_screen()
```

Сейчас мы напишем код управления флотом, для которого будет создан новый метод с именем `_update_aliens()`. Позиции пришельцев обновляются после обновления снарядов, потому что скоро мы будем проверять, попали ли какие-либо снаряды в пришельцев.

Местоположение этого метода в модуле некритично. Но для улучшения структуры кода мы разместим его сразу же после `_update_bullets()` в соответствии с порядком вызова методов в цикле `while`. Первая версия `_update_aliens()` выглядит так:

alien_invasion.py

```
def _update_aliens(self):
    """Обновляет позиции всех пришельцев во флоте."""
    self.aliens.update()
```

Мы используем метод `update()` для группы `aliens`, что приводит к автоматическому вызову метода `update()` каждого пришельца. Если запустить Alien Invasion сейчас, вы увидите, как флот двигается вправо и исчезает за краем экрана.

Создание настроек для направления флота

Теперь мы создадим настройки, которые перемещают флот вниз по экрану, а потом влево при достижении правого края экрана. Вот как реализуется это поведение:

settings.py

```
# Настройки пришельцев
self.alien_speed = 1.0
self.fleet_drop_speed = 10
# fleet_direction = 1 обозначает движение вправо; а -1 - влево.
self.fleet_direction = 1
```

Настройка `fleet_drop_speed` управляет величиной снижения флота при достижении им края. Эту скорость полезно отделить от горизонтальной скорости пришельцев, чтобы эти две скорости можно было изменять независимо.

Для настройки `fleet_direction` можно использовать текстовое значение (например, `'left'` или `'right'`), но скорее всего, в итоге придется использовать набор команд `if-elif` для проверки направления. Так как в данном случае направлений всего два, мы используем значения 1 и –1 и будем переключаться между ними при каждом изменении направления флота. (Числа в данном случае особенно удобны, потому что при движении вправо координата `x` каждого пришельца должна увеличиваться, а при перемещении влево — уменьшаться.)

Проверка достижения края

Также нам понадобится метод для проверки того, достиг ли пришелец одного из двух краев. Для этого необходимо внести в метод `update()` изменение, позволяющее каждому пришельцу двигаться в соответствующем направлении. Этот код является частью класса `Alien`:

alien.py

```
  def check_edges(self):
      """Возвращает True, если пришелец находится у края экрана."""
```

```
        screen_rect = self.screen.get_rect()
❶       if self.rect.right >= screen_rect.right or self.rect.left <= 0:
            return True

    def update(self):
        """Перемещает пришельца влево или вправо."""
❷       self.x += (self.settings.alien_speed *
                   self.settings.fleet_direction)
        self.rect.x = self.x
```

Вызов нового метода `check_edges()` для любого пришельца позволяет проверить, достиг ли он левого или правого края. У пришельца, находящегося у правого края, атрибут `right` его атрибута `rect` больше или равен атрибуту `right` атрибута `rect` экрана. У пришельца, находящегося у левого края, значение `left` меньше либо равно 0 ❶.

В метод `update()` будут внесены изменения, обеспечивающие перемещение влево и вправо; для этого скорость пришельца умножается на значение `fleet_direction` ❷. Если значение `fleet_direction` равно 1, то значение `alien_speed` прибавляется к текущей позиции пришельца и пришелец перемещается вправо; если же значение `fleet_direction` равно –1, то значение вычитается из позиции пришельца (который перемещается влево).

Снижение флота и смена направления

Когда пришелец доходит до края, весь флот должен опуститься вниз и изменить направление движения. Это означает, что в `AlienInvasion` придется внести изменения, потому что именно здесь программа проверяет, достиг ли какой-либо пришелец левого или правого края. Для этого мы напишем функции `_check_fleet_edges()` и `_change_fleet_direction()`, а затем изменим `_update_aliens()`. Новые методы будут располагаться после `_create_alien()`, но я еще раз подчеркну, что конкретное размещение этих методов в классе несущественно.

alien_invasion.py

```
    def _check_fleet_edges(self):
        """Реагирует на достижение пришельцем края экрана."""
❶       for alien in self.aliens.sprites():
            if alien.check_edges():
❷               self.change_fleet_direction()
                break

    def _change_fleet_direction(self):
        """Опускает весь флот и меняет направление флота."""
        for alien in self.aliens.sprites():
❸           alien.rect.y += self.settings.fleet_drop_speed
        self.settings.fleet_direction *= -1
```

Код `_check_fleet_edges()` перебирает флот и вызывает `check_edges()` для каждого пришельца ❶. Если `check_edges()` возвращает `True`, значит, пришелец находится у края и весь флот должен сменить направление, поэтому вызывается

функция `_change_fleet_direction()` и происходит выход из цикла ❷. Функция `_change_fleet_direction()` перебирает пришельцев и уменьшает высоту каждого из них с использованием настройки `fleet_drop_speed` ❸; затем направление `fleet_direction` меняется на противоположное, для чего текущее значение умножается на –1. Строка, изменяющая направление, не является частью цикла `for`. Вертикальная позиция должна изменяться для каждого пришельца, но направление всего флота должно измениться однократно.

Изменения в `_update_aliens()`:

alien_invasion.py

```
def _update_aliens(self):
    """
    Проверяет, достиг ли флот края экрана,
      с последующим обновлением позиций всех пришельцев во флоте.
    """
    self._check_fleet_edges()
    self.aliens.update()
```

Перед обновлением позиции каждого пришельца будет вызываться метод `_check_fleet_edges()`.

Если запустить игру сейчас, флот будет двигаться влево-вправо между краями экрана и опускаться каждый раз, когда он доберется до края. Теперь можно переходить к реализации уничтожения пришельцев и отслеживания пришельцев, сталкивающихся с кораблем или достигающих нижнего края экрана.

УПРАЖНЕНИЯ

13.3. Капли: найдите изображение дождевой капли и создайте сетку из капель. Капли должны постепенно опускаться вниз и исчезать у нижнего края экрана.

13.4. Дождь: измените свой код в упражнении 13.3, чтобы при исчезновении ряда капель у нижнего края экрана новый ряд появлялся у верхнего края и начинал свое падение.

Уничтожение пришельцев

Итак, мы создали корабль и флот пришельцев, но когда снаряды достигают пришельцев, они просто проходят насквозь, потому что программа не проверяет коллизии. В игровом программировании *коллизией* называется перекрытие игровых элементов. Чтобы снаряды сбивали пришельцев, метод `sprite.groupcollide()` используется для выявления коллизий между элементами двух групп.

Выявление коллизий

Когда снаряд попадает в пришельца, программа должна немедленно узнать об этом, чтобы сбитый пришелец исчез с экрана. Для этого мы будем проверять коллизии сразу же после обновления позиции снаряда.

Метод `sprite.groupcollide()` сравнивает прямоугольник `rect` каждого элемента с прямоугольником `rect` каждого элемента другой группы. В данном случае он сравнивает прямоугольник каждого снаряда с прямоугольником каждого пришельца и возвращает словарь со снарядами и пришельцами, между которыми обнаружены коллизии. Каждый ключ в словаре представляет снаряд, а ассоциированное с ним значение — пришельца, в которого попал снаряд. (Этот словарь будет использоваться в реализации системы подсчета очков в главе 14.)

Для проверки коллизий в конец функции `update_bullets()` добавляется следующий код:

alien_invasion.py

```
def _update_bullets(self):
    """Обновляет позиции снарядов и удаляет старые пули."""
    ...

    # Проверка попаданий в пришельцев.
    # При обнаружении попадания удалить снаряд и пришельца.
    collisions = pygame.sprite.groupcollide(
            self.bullets, self.aliens, True, True)
```

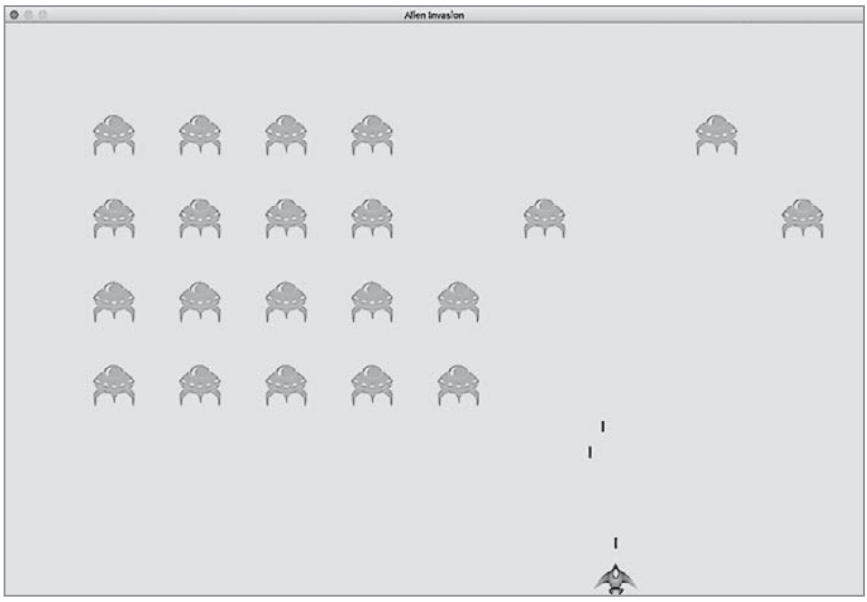

Рис. 13.5. Снаряды уничтожают пришельцев!

Новая строка сначала перебирает все снаряды в `self.bullets`, а затем перебирает всех пришельцев в `self.aliens`. Каждый раз, когда между прямоугольником снаряда и пришельца обнаруживается перекрытие, `groupcollide()` добавляет пару «ключ-значение» в возвращаемый словарь. Два аргумента `True` сообщают Pygame, нужно ли

удалять столкнувшиеся объекты: снаряд и пришельца. (Чтобы создать сверхмощный снаряд, который будет уничтожать всех пришельцев на своем пути, можно передать в первом аргументе `False`, а во втором `True`. Пришельцы, в которых попадает снаряд, будут исчезать, но все снаряды будут оставаться активными до верхнего края экрана.)

Если запустить Alien Invasion сейчас, пришельцы, в которых попадает снаряд, будут исчезать с экрана. На рис. 13.5 изображен частично уничтоженный флот.

Создание больших снарядов для тестирования

Многие игровые возможности тестируются простым запуском игры, но некоторые аспекты слишком утомительно тестировать в обычной версии игры. Например, чтобы проверить, правильно ли обрабатывается уничтожение последнего пришельца, нам пришлось бы несколько раз сбивать всех пришельцев на экране.

Для тестирования конкретных аспектов игры можно изменить игровые настройки так, чтобы упростить конкретную область. Например, можно уменьшить экран, чтобы на нем было меньше пришельцев, или увеличить скорость снаряда и количество снарядов, одновременно находящихся на экране.

Мое любимое изменение при тестировании Alien Invasion — использование сверхширокого снарядов, которые остаются активными даже после попадания в пришельца (рис. 13.6). Попробуйте задать настройке `bullet_width` значение 300 (или даже 3000!) и посмотрите, сколько времени вам понадобится для уничтожения флота пришельцев!

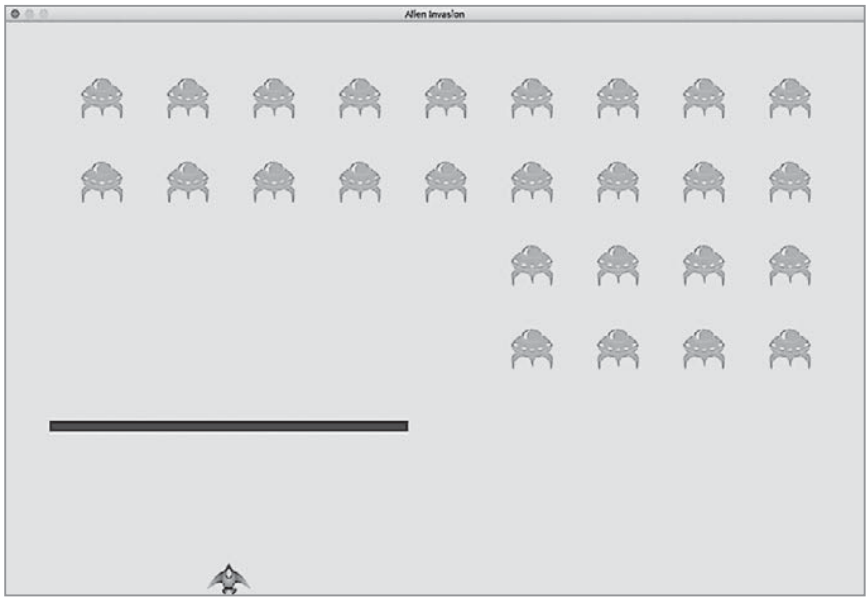

Рис. 13.6. Сверхмощные снаряды упрощают тестирование некоторых аспектов игры

Такие изменения повышают эффективность тестирования, а заодно могут подсказать идеи для всевозможных игровых бонусов. (Только не забудьте восстановить нормальное состояние настроек после завершения тестирования.)

Восстановление флота

Одна из ключевых особенностей Alien Invasion — бесконечные орды пришельцев: каждый раз, когда вы уничтожаете один флот, на его месте появляется другой.

Чтобы после уничтожения одного флота появлялся другой, сначала нужно убедиться в том, что группа `aliens` пуста. Если она пуста, вызывается метод `_create_fleet()`. Проверка будет выполняться в конце `_update_bullets()`, потому что именно здесь уничтожаются отдельные пришельцы:

alien_invasion.py

```
  def _update_bullets(self):
      ...
❶     if not self.aliens:
          # Уничтожение существующих снарядов и создание нового флота.
❷         self.bullets.empty()
          self._create_fleet()
```

В точке ❶ программа проверяет, пуста ли группа `aliens`. Пустая группа интерпретируется как `False`; это самый простой способ проверить группу на наличие элементов. Если группа пуста, то все существующие снаряды убираются методом `empty()`, который удаляет все существующие спрайты из группы ❷. Вызов метода `_create_fleet()` снова заполняет экран пришельцами.

Теперь сразу же после уничтожения текущего флота на экране появляется новый флот.

Ускорение снарядов

Попытавшись стрелять по пришельцам в текущем состоянии игры, можно заметить, что скорость движения снарядов не оптимальна — в вашей системе она может быть слишком высокой или слишком низкой. На этой стадии можно изменить настройки, чтобы игра была более интересной и приятной.

Скорость снарядов можно увеличить настройкой `bullet_speed` в settings.py. Например, если задать в моей системе значение 1.5, снаряды будут двигаться по экрану немного быстрее:

settings.py

```
# Настройки снарядов
    self.bullet_speed = 1.5
self.bullet_width = 3
...
```

Оптимальное значение этой настройки зависит от производительности вашей системы. Найдите значение, которое лучше подходит для вашей конкретной конфигурации.

Рефакторинг _update_bullets()

Переработаем метод `_update_bullets()`, чтобы он не решал такое количество разных задач. Код обработки коллизий будет выделен в отдельный метод:

alien_invasion.py

```
def _update_bullets(self):
    ...
    # Уничтожение исчезнувших снарядов.
    for bullet in self.bullets.copy():
        if bullet.rect.bottom <= 0:
            self.bullets.remove(bullet)

    self._check_bullet_alien_collisions()

def _check_bullet_alien_collisions(self):
    """Обработка коллизий снарядов с пришельцами."""
    # Удаление снарядов и пришельцев, участвующих в коллизиях.
    collisions = pygame.sprite.groupcollide(
            self.bullets, self.aliens, True, True)

    if not self.aliens:
        # Уничтожение существующих снарядов и создание нового флота.
        self.bullets.empty()
        self._create_fleet()
```

Мы создали новый метод `check_bullet_alien_collisions()` для выявления коллизий между снарядами и пришельцами и для реакции на уничтожение всего флота. Это сделано для того, чтобы сократить длину `_update_bullets()` и упростить дальнейшую разработку.

УПРАЖНЕНИЯ

13.5. Боковая стрельба-2: после упражнения 12.6 «Боковая стрельба» программа была серьезно доработана. В этом упражнении вам предлагается усовершенствовать «Боковую стрельбу» до того же состояния, к которому была приведена игра Alien Invasion. Добавьте флот пришельцев и заставьте их перемещаться по горизонтали по направлению к кораблю. Другой вариант: напишите код, который размещает пришельцев в случайных позициях у правого края экрана, а затем заставляет их двигаться к кораблю. Также напишите код, который заставляет пришельцев исчезать при попадании в них.

Завершение игры

Какое удовольствие от игры, в которой невозможно проиграть? Если игрок не успеет сбить флот достаточно быстро, пришельцы уничтожат корабль при столк-

новении. При этом количество кораблей, используемых игроком, ограниченно, и корабль уничтожается, когда пришелец достигает нижнего края экрана. Игра завершается в тот момент, когда у игрока кончатся все корабли.

Обнаружение коллизий с кораблем

Начнем с проверки коллизий между пришельцами и кораблем, чтобы мы могли правильно обработать столкновения с пришельцами. Коллизии «пришелец — корабль» проверяются немедленно после обновления позиции каждого пришельца в `AlienInvasion`:

alien_invasion.py

```
  def _update_aliens(self):
      ...
      self.aliens.update()

      # Проверка коллизий "пришелец – корабль".
❶     if pygame.sprite.spritecollideany(self.ship, self.aliens):
❷         print("Ship hit!!!")
```

Функция `spritecollideany()` получает два аргумента: спрайт и группу. Функция пытается найти любой элемент группы, вступивший в коллизию со спрайтом, и останавливает цикл по группе сразу же после обнаружения столкнувшегося элемента. В данном случае он перебирает группу `aliens` и возвращает первого пришельца, столкнувшегося с кораблем `ship`.

Если ни одна коллизия не обнаружена, `spritecollideany()` возвращает `None`, и блок `if` в точке ❶ не выполняется. Если же будет обнаружен пришелец, столкнувшийся с кораблем, метод возвращает этого пришельца и выполняется блок `if`: выводится сообщение `Ship hit!!!` ❷. При столкновении пришельца с кораблем необходимо выполнить ряд операций: удалить всех оставшихся пришельцев и снаряды, вернуть корабль в центр и создать новый флот. Прежде чем писать код всех этих операций, необходимо убедиться в том, что решение с обнаружением коллизий с кораблем работает правильно. Вызов `print()` всего лишь позволяет легко проверить правильность обнаружения коллизий.

Если вы запустите Alien Invasion, при столкновении пришельца с кораблем в терминальном окне появляется сообщение `Ship hit!!!`. В ходе тестирования этого аспекта присвойте `alien_drop_speed` более высокое значение (например, 50 или 100), чтобы пришельцы быстрее добирались до вашего корабля.

Обработка столкновений с кораблем

Теперь нужно разобраться, что же происходит при столкновении пришельца с кораблем. Вместо того чтобы уничтожать экземпляр `ship` и создавать новый, мы будем подсчитывать количество уничтоженных кораблей; для этого следует организовать сбор статистики по игре. (Статистика также пригодится для подсчета очков.)

Напишем новый класс `GameStats` для ведения статистики и сохраним его в файле game_stats.py:

game_stats.py

```
class GameStats():
    """Отслеживание статистики для игры Alien Invasion."""

    def __init__(self, ai_game):
        """Инициализирует статистику."""
        self.settings = ai_game.settings
❶       self.reset_stats()

    def reset_stats(self):
        """Инициализирует статистику, изменяющуюся в ходе игры."""
        self.ships_left = self.settings.ship_limit
```

На все время работы Alien Invasion будет создаваться один экземпляр `GameStats`, но часть статистики должна сбрасываться в начале каждой новой игры. Для этого большая часть статистики будет инициализироваться в методе `reset_stats()` вместо `__init__()`. Этот метод будет вызываться из `__init__()`, чтобы статистика правильно инициализировалась при первом создании экземпляра `GameStats` ❶, а метод `reset_stats()` будет вызываться в начале каждой новой игры.

Пока в игре используется всего один вид статистики — значение `ships_left`, изменяющееся в ходе игры. Количество кораблей в начале игры хранится в settings.py под именем `ship_limit`:

settings.py

```
# Настройки корабля
self.ship_speed = 1.5
self.ship_limit = 3
```

Также необходимо внести ряд изменений в alien_invasion.py для создания экземпляра `GameStats`. Начнем с обновления команд `import` в начале файла:

alien_invasion.py

```
import sys
from time import sleep

import pygame

from settings import Settings
from game_stats import GameStats
from ship import Ship
...
```

Мы импортируем функцию `sleep()` из модуля `time` стандартной библиотеки Python, чтобы игру можно было ненадолго приостановить в момент столкновения с кораблем. Также импортируется класс `GameStats`.

Экземпляр `GameStats` создается в `__init__()`:

alien_invasion.py

```
def __init__(self):
    ...
    self.screen = pygame.display.set_mode(
        (self.settings.screen_width, self.settings.screen_height))
    pygame.display.set_caption("Alien Invasion")

        # Создание экземпляра для хранения игровой статистики.
        self.stats = GameStats(self)

        self.ship = Ship(self)
        ...
```

Экземпляр создается после создания игрового окна, но перед определением других игровых элементов (например, корабля).

Когда пришелец сталкивается с кораблем, программа уменьшает количество оставшихся кораблей на 1, уничтожает всех существующих пришельцев и снаряды, создает новый флот и возвращает корабль в середину экрана. Также игра ненадолго приостанавливается, чтобы игрок заметил столкновение и перестроился перед появлением нового флота.

Большая часть этого кода будет вынесена в новый метод `_ship_hit()`. Этот метод вызывается из `_update_aliens()` при столкновении пришельца с кораблем:

alien_invasion.py

```
  def _ship_hit(self):
      """Обрабатывает столкновение корабля с пришельцем."""
      # Уменьшение ships_left.
❶     self.stats.ships_left -= 1

      # Очистка списков пришельцев и снарядов.
❷     self.aliens.empty()
      self.bullets.empty()

      # Создание нового флота и размещение корабля в центре.
❸     self._create_fleet()
      self.ship.center_ship()

      # Пауза.
❹     sleep(0.5)
```

Новый метод `_ship_hit()` управляет реакцией игры на столкновение корабля с пришельцем. Внутри `_ship_hit()` число оставшихся кораблей уменьшается на 1 ❶, после чего происходит очистка групп `aliens` и `bullets` ❷.

Затем программа создает новый флот и выравнивает корабль по центру нижнего края ❸. (Вскоре мы добавим метод `center_ship()` в класс `Ship`.) Наконец, после внесения изменений во все игровые элементы, но до перерисовки изменений на

экране делается короткая пауза, чтобы игрок увидел, что его корабль столкнулся с пришельцем ❹. Вызов `sleep()` приостанавливает программу на 0,5 секунды. После завершения паузы управление передается методу `_update_screen()`, который перерисовывает новый флот на экране.

Внутри `_update_aliens()` вызов `print()` заменяется вызовом `_ship_hit()` при столкновении пришельца с кораблем:

alien_invasion.py

```
def _update_aliens(self):
    ...
    if pygame.sprite.spritecollideany(self.ship, self.aliens):
        self._ship_hit()
```

Новый метод `center_ship()` добавляется в конец файла ship.py:

```
def center_ship(self):
    """Размещает корабль в центре нижней стороны."""
    self.rect.midbottom = self.screen_rect.midbottom
    self.x = float(self.rect.x)
```

Выравнивание корабля по центру выполняется так же, как и в `__init__()`. После выравнивания сбрасывается атрибут `self.x`, чтобы в программе отслеживалась точная позиция корабля.

ПРИМЕЧАНИЕ Обратите внимание: программа никогда не создает более одного корабля. Один экземпляр `ship` используется на протяжении всей игры, а при столкновении с пришельцем он просто возвращается к центру экрана. О том, что у игрока не осталось ни одного корабля, программа узнает из атрибута `ships_left`.

Запустите игру, подбейте нескольких пришельцев, а затем позвольте пришельцу столкнуться с кораблем. Происходит небольшая пауза, на экране появляется новый флот вторжения, а корабль возвращается в центр нижней части экрана.

Достижение нижнего края экрана

Если пришелец добирается до нижнего края экрана, программа будет реагировать так же, как при столкновении с кораблем. Добавьте для проверки этого условия новый метод в alien_invasion.py:

alien_invasion.py

```
  def _check_aliens_bottom(self):
      """Проверяет, добрались ли пришельцы до нижнего края экрана."""
      screen_rect = self.screen.get_rect()
      for alien in self.aliens.sprites():
❶         if alien.rect.bottom >= screen_rect.bottom:
              # Происходит то же, что при столкновении с кораблем.
              self._ship_hit()
              break
```

Метод `check_aliens_bottom()` проверяет, есть ли хотя бы один пришелец, добравшийся до нижнего края экрана. Условие выполняется, когда атрибут `rect.bottom` пришельца больше или равен атрибуту `rect.bottom` экрана ❶. Если пришелец добрался до низа, вызывается функция `_ship_hit()`. Если хотя бы один пришелец добрался до нижнего края, проверять остальных уже не нужно, поэтому после вызова `_ship_hit()` цикл прерывается.

Этот метод вызывается из `_update_aliens()`:

alien_invasion.py

```
def _update_aliens(self):
    ...
    # Проверка коллизий "пришелец – корабль".
    if pygame.sprite.spritecollideany(self.ship, self.aliens):
        self._ship_hit()

    # Проверить, добрались ли пришельцы до нижнего края экрана.
    self._check_aliens_bottom()
```

Метод `_check_aliens_bottom()` вызывается после обновления позиций всех пришельцев и после проверки столкновений «пришелец — корабль» ❷. Теперь новый флот будет появляться как при столкновении корабля с пришельцем, так и в том случае, если кто-то из пришельцев смог добраться до нижнего края экрана.

Конец игры

Программа Alien Invasion уже на что-то похожа, но игра длится бесконечно. Значение `ships_left` просто продолжает уходить в отрицательную бесконечность. Добавим в `GameStats` новый атрибут — флаг `game_active`, который завершает игру после потери последнего корабля. Этот флаг устанавливается в конце метода `__init__()` в GameStats:

game_stats.py

```
def __init__(self, ai_game):
    ...
    # Игра Alien Invasion запускается в активном состоянии.
    self.game_active = True
```

Добавим в `ship_hit()` код, который сбрасывает флаг `game_active` в состояние `False` при потере игроком последнего корабля:

alien_invasion.py

```
def _ship_hit(self):
    """Обрабатывает столкновение корабля с пришельцем."""
    if stats.ships_left > 0:
        # Уменьшение ships_left.
        self.stats.ships_left -= 1
        ...
```

```
            # Пауза.
            sleep(0.5)
        else:
            self.stats.game_active = False
```

Большая часть кода `_ship_hit()` осталась неизменной. Весь существующий код был перемещен в блок `if`, который проверяет, что у игрока остался хотя бы один корабль. Если корабли еще остаются, программа создает новый флот, делает паузу и продолжает игру. Если же игрок потерял последний корабль, флаг `game_active` переводится в состояние `False`.

Определение исполняемых частей игры

В файле alien_invasion.py необходимо определить части игры, которые должны выполняться всегда, и те части, которые должны выполняться только при активной игре:

alien_invasion.py

```
def run_game(self):
    """ Запуск основного цикла игры."""
    while True:
        self._check_events()

        if self.stats.game_active:
            self.ship.update()
            self._update_bullets()
            self._update_aliens()

        self._update_screen()
```

В основном цикле всегда должна вызываться функция `_check_events()`, даже если игра находится в неактивном состоянии. Например, программа все равно должна узнать о том, что пользователь нажал клавишу Q для завершения игры или щелкнул на кнопке закрытия окна. Также экран должен обновляться в то время, пока игрок решает, хочет ли он начать новую игру. Остальные вызовы функций должны происходить только при активной игре, потому что в то время, когда игра неактивна, обновлять позиции игровых элементов не нужно.

В обновленной версии игра должна останавливаться после потери игроком последнего корабля.

УПРАЖНЕНИЯ

13.6. Конец игры: в упражнении «Боковая стрельба» в коде из упражнения 13.5 (с. 285) подсчитывайте, сколько пришельцев было сбито игроком и сколько раз произошло столкновение с кораблем. Определите разумное условие завершения игры и останавливайте игру при возникновении этой ситуации.

Итоги

В этой главе вы научились добавлять в игру большое количество одинаковых элементов на примере флота пришельцев. Вы узнали, как использовать вложенные циклы для создания сетки с элементами, а также привели игровые элементы в движение, вызывая метод `update()` каждого элемента. Вы научились управлять перемещением объектов на экране и обрабатывать различные события (например, достижение края экрана). Вы также узнали, как обнаруживать коллизии и реагировать на них (на примере попадания снарядов в пришельцев и столкновения пришельцев с кораблем). В завершение главы рассматривалась тема ведения игровой статистики и использования флага `game_active` для проверки окончания игры.

В последней главе этого проекта будет добавлена кнопка Play, чтобы игрок мог самостоятельно запустить свою первую игру, а также повторить игру после ее завершения. После каждого уничтожения вражеского флота скорость игры будет возрастать, а мы реализуем систему подсчета очков. В результате вы получите полностью работоспособную игру.

14 Ведение счета

В этой главе построение игры Alien Invasion будет завершено. Мы добавим кнопку Play для запуска игры по желанию игрока или перезапуска игры после ее завершения. Мы также изменим игру, чтобы она ускорялась при переходе игрока на следующий уровень, а также реализуем систему подсчета очков. К концу главы вы будете знать достаточно, чтобы заняться разработкой игр, сложность которых нарастает по ходу игры и в которых реализована система подсчета очков.

Добавление кнопки Play

В этом разделе мы добавим кнопку Play, которая отображается перед началом игры и появляется после ее завершения, чтобы игрок мог сыграть снова.

В текущей версии игра начинается сразу же после запуска alien_invasion.py. После очередных изменений игра будет запускаться в неактивном состоянии и предлагать игроку нажать кнопку Play для запуска. Для этого включите следующий код в метод `__init__()` класса `GameStats`:

game_stats.py

```
def __init__(self, ai_game):
    """Инициализирует статистику."""
    self.settings = ai_game.settings
    self.reset_stats()

    # Игра запускается в неактивном состоянии.
    self.game_active = False
```

Итак, программа запускается в неактивном состоянии, а игру можно запустить только нажатием кнопки Play.

Создание класса Button

Так как в Pygame не существует встроенного метода создания кнопок, мы напишем класс `Button` для создания заполненного прямоугольника с текстовой надписью.

Следующий код может использоваться для создания кнопок в любой игре. Ниже приведена первая часть класса `Button`; сохраните ее в файле button.py:

button.py

```
import pygame.font

class Button():

❶     def __init__(self, ai_game, msg):
          """Инициализирует атрибуты кнопки."""
          self.screen = ai_game.screen
          self.screen_rect = self.screen.get_rect()

          # Назначение размеров и свойств кнопок.
❷         self.width, self.height = 200, 50
          self.button_color = (0, 255, 0)
          self.text_color = (255, 255, 255)
❸         self.font = pygame.font.SysFont(None, 48)

          # Построение объекта rect кнопки и выравнивание по центру экрана.
❹         self.rect = pygame.Rect(0, 0, self.width, self.height)
          self.rect.center = self.screen_rect.center

          # Сообщение кнопки создается только один раз.
❺         self.prep_msg(msg)
```

Сначала программа импортирует модуль pygame.font, который позволяет Pygame выводить текст на экран. Метод `__init__()` получает параметры `self`, объект `ai_game` и строку `msg` с текстом кнопки ❶. Размеры кнопки задаются в точке ❷, после чего атрибуты `button_color` и `text_color` задаются так, чтобы прямоугольник кнопки был окрашен в ярко-зеленый цвет, а текст выводился белым цветом.

В точке ❸ происходит подготовка атрибута `font` для вывода текста. Аргумент `None` сообщает Pygame, что для вывода текста должен использоваться шрифт по умолчанию, а значение `48` определяет размер текста. Чтобы выровнять кнопку по центру экрана, мы создаем объект `rect` для кнопки ❹ и задаем его атрибут `center` в соответствии с одноименным атрибутом экрана.

Pygame выводит строку текста в виде графического изображения. В точке ❺ эта задача решается методом `_prep_msg()`.

Код `_prep_msg()` выглядит так:

button.py

```
  def _prep_msg(self, msg):
      """Преобразует msg в прямоугольник и выравнивает текст по центру."""
❶     self.msg_image = self.font.render(msg, True, self.text_color,
              self.button_color)
❷     self.msg_image_rect = self.msg_image.get_rect()
      self.msg_image_rect.center = self.rect.center
```

Метод `_prep_msg()` должен получать параметр `self` и текст, который нужно вывести в графическом виде (`msg`). Вызов `font.render()` преобразует текст, хранящийся в `msg`, в изображение, которое затем сохраняется в `self.msg_image` ❶. Методу `font.render()` также передается логический признак режима сглаживания текста. В остальных аргументах передаются цвет шрифта и цвет фона. В нашем примере режим сглаживания включен (`True`), а цвет фона совпадает с цветом фона кнопки. (Если цвет фона не указан, Pygame пытается вывести шрифт с прозрачным фоном.)

В точке ❷ изображение текста выравнивается по центру кнопки, для чего создается объект `rect` изображения, а его атрибут `center` приводится в соответствие с одноименным атрибутом кнопки.

Остается создать метод `draw_button()`, который может вызываться для отображения кнопки на экране:

button.py

```
def draw_button(self):
    # Отображение пустой кнопки и вывод сообщения.
    self.screen.fill(self.button_color, self.rect)
    self.screen.blit(self.msg_image, self.msg_image_rect)
```

Вызов метода `screen.fill()` рисует прямоугольную часть кнопки. Затем вызов `screen.blit()` выводит изображение текста на экран, с передачей изображения и объекта `rect`, связанного с изображением. Класс `Button` готов.

Вывод кнопки на экран

В программе класс `Button` используется для создания кнопки Play. Мы создадим кнопку прямо в файле alien_invasion.py:

alien_invasion.py

```
...
from game_stats import GameStats
from button import Button
```

Так как нам нужна только одна кнопка Play, мы создадим ее в методе `__init__()` класса `AlienInvasion`. Этот код можно разместить в самом конце `__init__()`:

alien_invasion.py

```
def __init__(self):
    ...
    self._create_fleet()

    # Создание кнопки Play.
    self.play_button = Button(self, "Play")
```

Программа создает экземпляр `Button` с текстом `Play`, но не выводит кнопку на экран. Чтобы кнопка появилась на экране, мы вызовем метод `draw_button()` кнопки в `_update_screen()`:

alien_invasion.py

```
def _update_screen(self):
    ...
    self.aliens.draw(self.screen)

    # Кнопка Play отображается в том случае, если игра неактивна.
    if not self.stats.game_active:
        self.play_button.draw_button()

    pygame.display.flip()
```

Чтобы кнопка Play не закрывалась другими элементами экрана, мы отображаем ее после всех остальных игровых элементов, но перед переключением на новый экран. Код заключается в блок `if`, чтобы кнопка отображалась только в неактивном состоянии игры.

Теперь при запуске Alien Invasion в центре экрана отображается кнопка Play (рис. 14.1).

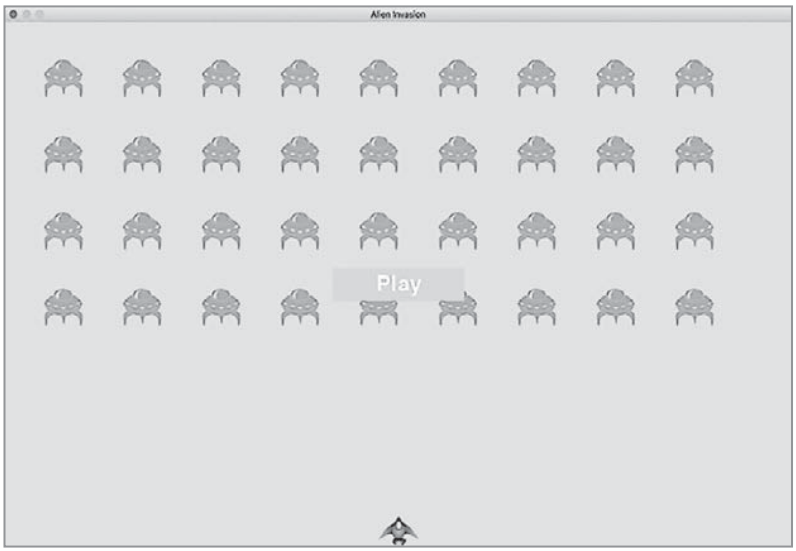

Рис. 14.1. Кнопка Play выводится тогда, когда игра неактивна

Запуск игры

Чтобы при нажатии кнопки Play запускалась новая игра, добавьте в конец `_check_events()` следующий блок `elif` для отслеживания событий мыши над кнопкой:

alien_invasion.py

```
  def _check_events(self):
      """Обрабатывает нажатия клавиш и события мыши."""
      for event in pygame.event.get():
          if event.type == pygame.QUIT:
              ...
❶         elif event.type == pygame.MOUSEBUTTONDOWN:
❷             mouse_pos = pygame.mouse.get_pos()
❸             self._check_play_button(mouse_pos)
```

Pygame обнаруживает событие `MOUSEBUTTONDOWN`, когда игрок щелкает в любой точке экрана ❶, но мы хотим ограничить игру, чтобы она реагировала только на щелчки на кнопке Play. Для этого будет использоваться метод `pygame.mouse.get_pos()`, возвращающий кортеж с координатами `x` и `y` точки щелчка ❷. Эти значения передаются новому методу `_check_play_button()` ❸.

Ниже приведен код `_check_play_button()`, который я решил разместить после `_check_events()`:

alien_invasion.py

```
  def _check_play_button(self, mouse_pos):
      """Запускает новую игру при нажатии кнопки Play."""
❶     if self.play_button.rect.collidepoint(mouse_pos):
          self.stats.game_active = True
```

Метод `collidepoint()` используется для проверки того, находится ли точка щелчка в пределах области, определяемой прямоугольником кнопки Play ❶. Если точка находится в пределах кнопки, флаг `game_active` переводится в состояние `True` и игра начинается!

К этому моменту вы сможете запустить и сыграть полноценную игру. После завершения игры значение `game_active` становится равным `False`, а кнопка Play снова появится на экране.

Сброс игры

Только что написанный нами код работает при первом нажатии кнопки Play, но не работает после завершения первой игры, потому что условия, приводящие к окончанию игры, еще не были сброшены.

Чтобы игра сбрасывалась при каждом нажатии кнопки Play, необходимо сбросить игровую статистику, стереть старых пришельцев и снаряды, построить новый флот и вернуть корабль в центр нижней стороны:

alien_invasion.py

```
  def _check_play_button(self,mouse_pos):
      """Запускает новую игру при нажатии кнопки Play."""
      if self.play_button.rect.collidepoint(mouse_pos):
          # Сброс игровой статистики.
```

```
❶         self.stats.reset_stats()
          self.stats.game_active = True

          # Очистка списков пришельцев и снарядов.
❷         self.aliens.empty()
          self.bullets.empty()

          # Создание нового флота и размещение корабля в центре.
❸         self._create_fleet()
          self.ship.center_ship()
```

В точке ❶ обновляется игровая статистика, вследствие чего игрок получает три новых корабля. После этого флаг `game_active` переводится в состояние `True` (чтобы игра началась сразу же после выполнения кода функции), группы `aliens` и `bullets` очищаются ❷, создается новый флот, а корабль выравнивается по центру ❸.

После этих изменений игра будет правильно переходить в исходное состояние при каждом нажатии Play, и вы сможете сыграть столько раз, сколько вам захочется!

Блокировка кнопки Play

У кнопки Play в нашем приложении есть одна проблема: область кнопки на экране продолжает реагировать на щелчки, даже если кнопка Play не отображается. Если случайно щелкнуть на месте кнопки Play после начала игры, то игра перезапустится!

Чтобы исправить этот недостаток, следует запускать игру только в том случае, если флаг `game_active` находится в состоянии `False`:

alien_invasion.py

```
  def _check_play_button(self, mouse_pos):
      """Запускает новую игру при нажатии кнопки Play."""
❶     button_clicked = self.play_button.rect.collidepoint(mouse_pos)
❷     if button_clicked and not self.stats.game_active:
          # Сброс игровой статистики.
          self.stats.reset_stats()
          ...
```

Флаг `button_clicked` содержит значение `True` или `False` ❶, а игра перезапускается только в том случае, если пользователь нажал кнопку Play и *при этом* игра неактивна в настоящий момент ❷. Чтобы протестировать это поведение, запустите новую игру и многократно щелкайте в том месте, где должна находиться кнопка Play. Если все работает так, как положено, нажатия кнопки Play не должны влиять на ход игры.

Сокрытие указателя мыши

Указатель мыши должен быть видимым, чтобы пользователь мог начать игру, но после начала игры он только мешает. Чтобы исправить этот недостаток, мы скроем указатель мыши после того, как игра станет активной. Это можно сделать в блоке `if` в конце `_check_play_button()`:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    """Запускает новую игру при нажатии кнопки Play."""
    button_clicked = self.play_button.rect.collidepoint(mouse_pos)
    if button_clicked and not self.stats.game_active:
        ...
        # Указатель мыши скрывается.
        pygame.mouse.set_visible(False)
```

Вызов `set_visible()` со значением `False` приказывает Pygame скрыть указатель, когда он находится над окном игры.

После завершения игры указатель должен появляться снова, чтобы игрок мог нажать кнопку Play для запуска новой игры. Эту задачу решает следующий код:

alien_invasion.py

```
def _ship_hit(self):
    """Обрабатывает столкновение корабля с пришельцем."""
    if self.stats.ships_left > 0:
        ...
    else:
        self.stats.game_active = False
        pygame.mouse.set_visible(True)
```

Указатель снова появляется сразу же после того, как игра становится неактивной, что происходит в `_ship_hit()`. Внимание к подобным деталям сделает вашу игру более профессиональной, а игрок сможет сосредоточиться на игре вместо того, чтобы разбираться в сложностях пользовательского интерфейса.

УПРАЖНЕНИЯ

14.1. Запуск игры клавишей P: так как в Alien Invasion игрок управляет кораблем с клавиатуры, для запуска игры также лучше использовать клавиатуру. Добавьте код, с которым игрок сможет запустить игру нажатием клавиши P. Возможно, часть кода из `_check_play_button()` стоит переместить в функцию `start_game()`, которая будет вызываться из `_check_play_button()` и `_check_keydown_events()`.

14.2. Стрельба по мишени: создайте у правого края экрана прямоугольник, который двигается вверх и вниз с постоянной скоростью. У левого края располагается корабль, который перемещается вверх и вниз игроком и стреляет по движущейся прямоугольной мишени. Добавьте кнопку Play для запуска игры. После трех промахов игра заканчивается, а на экране снова появляется кнопка Play. Нажатие этой кнопки перезапускает игру.

Повышение сложности

В текущей версии после того, как весь флот пришельцев будет уничтожен, игрок переходит на новый уровень, но сложность игры остается неизменной. Давайте немного оживим игру и усложним ее; для этого скорость игры будет повышаться каждый раз, когда игрок уничтожает весь флот.

Изменение настроек скорости

Начнем с реорганизации класса `Settings` и разделения настроек игры на две категории: постоянные и изменяющиеся. Также необходимо проследить за тем, чтобы настройки, изменяющиеся в ходе игры, сбрасывались в исходное состояние в начале новой игры. Метод `__init__()` из файла settings.py выглядит так:

settings.py

```
  def __init__(self):
      """Инициализирует статические настройки игры."""
      # Настройки экрана
      self.screen_width = 1200
      self.screen_height = 800
      self.bg_color = (230, 230, 230)

      # Настройки корабля
      self.ship_limit = 3

      # Настройки снарядов
      self.bullet_width = 3
      self.bullet_height = 15
      self.bullet_color = 60, 60, 60
      self.bullets_allowed = 3

      # Настройки пришельцев
      self.fleet_drop_speed = 10

      # Темп ускорения игры
❶     self.speedup_scale = 1.1

❷     self.initialize_dynamic_settings()
```

Значения, которые остаются неизменными, по-прежнему инициализируются в методе `__init__()`. В точке ❶ добавляется настройка `speedup_scale`, управляющая быстротой нарастания скорости; со значением 2 скорость удваивается каждый раз, когда игрок переходит на следующий уровень, а со значением 1 скорость остается постоянной. С таким значением, как 1,1, скорость будет увеличиваться в достаточной степени, чтобы игра усложнилась, но не стала невозможной. Наконец, вызов `initialize_dynamic_settings()` инициализирует значения атрибутов, которые должны изменяться в ходе игры ❷.

Код `initialize_dynamic_settings()` выглядит так:

settings.py

```
def initialize_dynamic_settings(self):
    """Инициализирует настройки, изменяющиеся в ходе игры."""
    self.ship_speed_factor = 1.5
    self.bullet_speed_factor = 3.0
    self.alien_speed_factor = 1.0

    # fleet_direction = 1 обозначает движение вправо; а -1 - влево.
    self.fleet_direction = 1
```

Метод задает исходные значения скоростей корабля, снарядов и пришельцев. Эти скорости будут увеличиваться по ходу игры и сбрасываться каждый раз, когда игрок запускает новую игру. Мы включаем в этот метод `fleet_direction`, чтобы пришельцы в начале новой игры всегда двигались вправо. Увеличивать значение `fleet_drop_speed` не нужно, потому что когда пришельцы быстрее двигаются по горизонтали, они также будут быстрее перемещаться по вертикали.

Для увеличения скорости корабля, снарядов и пришельцев каждый раз, когда игрок достигает нового уровня, мы напишем новый метод `increase_speed()`:

settings.py

```
def increase_speed(self):
    """Увеличивает настройки скорости."""
    self.ship_speed_factor *= self.speedup_scale
    self.bullet_speed_factor *= self.speedup_scale
    self.alien_speed_factor *= self.speedup_scale
```

Чтобы увеличить скорость этих игровых элементов, мы умножаем каждую настройку скорости на значение `speedup_scale`.

Темп игры повышается вызовом `increase_speed()` в `check_bullet_alien_collisions()` при уничтожении последнего пришельца во флоте, но перед созданием нового флота:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    ...
    if not self.aliens:
        # Уничтожение снарядов, повышение скорости и создание нового флота.
        self.bullets.empty()
        self._create_fleet()
        self.settings.increase_speed()
```

Изменения значений настроек скорости `ship_speed`, `alien_speed` и `bullet_speed` достаточно для того, чтобы ускорить всю игру.

Сброс скорости

Каждый раз, когда игрок начинает новую игру, все измененные настройки должны вернуться к исходным значениям, иначе каждая новая игра будет начинаться с повышенными настройками скорости предыдущей игры:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    """Запускает новую игру при нажатии кнопки Play."""
    button_clicked = self.play_button.rect.collidepoint(mouse_pos)
    if button_clicked and not self.stats.game_active:
        # Сброс игровых настроек.
        self.settings.initialize_dynamic_settings()
        ...
```

Игра Alien Invasion стала достаточно сложной и интересной. Каждый раз, когда игрок очищает экран, игра должна слегка ускориться, а ее сложность — слегка возрасти. Если сложность игры возрастает слишком быстро, уменьшите значение `settings.speedup_scale`, а если, наоборот, сложность недостаточна — слегка увеличьте это значение. Найдите оптимальное значение, оценивая сложность игры за разумный промежуток времени. Первая пара флотов должна быть простой, несколько следующих — сложными, но возможными, а при последующих попытках сложность должна становиться практически безнадежной.

УПРАЖНЕНИЯ

14.3. Учебная стрельба с нарастающей сложностью: начните с кода упражнения 14.2 (с. 299). Скорость мишени должна увеличиваться по ходу игры, а при нажатии игроком кнопки Play мишень должна возвращаться к исходной скорости.

14.4. Уровни сложности: создайте в Alien Invasion набор кнопок для выбора начальной сложности игры. Каждая кнопка должна присваивать атрибутам Settings значения, необходимые для создания различных уровней сложности.

Подсчет очков

Система подсчета очков позволит отслеживать счет игры в реальном времени; также на экране будет выводиться текущий рекорд, уровень и количество оставшихся кораблей.

Счет игры также относится к игровой статистике, поэтому мы добавим атрибут `score` в класс `GameStats`:

game_stats.py

```
class GameStats():

    ...
    def reset_stats(self):
        """Инициализирует статистику, изменяющуюся в ходе игры."""
        self.ships_left = self.ai_settings.ship_limit
        self.score = 0
```

Чтобы счет сбрасывался при запуске новой игры, мы инициализируем `score` в `reset_stats()` вместо `__init__()`.

Вывод счета

Чтобы вывести счет на экран, мы сначала создаем новый класс `Scoreboard`. Пока этот класс ограничивается выводом текущего счета, но мы используем его для вывода рекордного счета, уровня и количества оставшихся кораблей. Ниже приведена первая часть класса; сохраните ее под именем scoreboard.py:

scoreboard.py

```
  import pygame.font

  class Scoreboard():
      """Класс для вывода игровой информации."""

❶     def __init__(self, ai_game):
          """Инициализирует атрибуты подсчета очков."""
          self.screen = ai_game.screen
          self.screen_rect = self.screen.get_rect()
          self.settings = ai_game.settings
          self.stats = ai_game.stats

          # Настройки шрифта для вывода счета.
❷         self.text_color = (30, 30, 30)
❸         self.font = pygame.font.SysFont(None, 48)
          # Подготовка исходного изображения.
❹         self.prep_score()
```

Так как `Scoreboard` выводит текст на экран, код начинается с импортирования модуля `pygame.font`. Затем `__init__()` передается параметр `ai_game` для обращения к объектам `settings`, `screen` и `stats`, чтобы класс мог выводить информацию об отслеживаемых показателях ❶. Далее назначается цвет текста ❷ и создается экземпляр объекта шрифта ❸.

Чтобы преобразовать выводимый текст в изображение, мы вызываем метод `prep_score()` ❹, который определяется следующим образом:

scoreboard.py

```
      def prep_score(self):
          """Преобразует текущий счет в графическое изображение."""
❶         score_str = str(self.stats.score)
❷         self.score_image = self.font.render(score_str, True,
                  self.text_color, self.settings.bg_color)

          # Вывод счета в правой верхней части экрана.
❸         self.score_rect = self.score_image.get_rect()
❹         self.score_rect.right = self.screen_rect.right - 20
❺         self.score_rect.top = 20
```

В методе `prep_score()` числовое значение `stats.score` преобразуется в строку ❶; эта строка передается методу `render()`, создающему изображение ❷. Чтобы счет был хорошо виден на экране, `render()` передается цвет фона и цвет текста.

Счет размещается в правой верхней части экрана и расширяется влево с ростом значения и ширины числа. Чтобы счет всегда оставался выровненным по правой стороне, мы создаем прямоугольник `rect` с именем `score_rect` ❸ и смещаем его правую сторону на 20 пикселов от правого края экрана ❹. Затем верхняя сторона прямоугольника смещается на 20 пикселов вниз от верхнего края экрана ❺.

Остается создать метод `show_score()` для вывода построенного графического изображения:

scoreboard.py

```
def show_score(self):
    """Выводит счет на экран."""
    self.screen.blit(self.score_image, self.score_rect)
```

Метод выводит счет на экран в позиции, определяемой `score_rect`.

Создание экземпляра Scoreboard

Чтобы вывести счет, мы создадим в `AlienInvasion` экземпляр `Scoreboard`. Начнем с обновления команд импортирования:

alien_invasion.py

```
...
from game_stats import GameStats
from scoreboard import Scoreboard
...
```

Затем создадим экземпляр `Scoreboard` в `__init__()`:

alien_invasion.py

```
def __init__(self):
    ...
    pygame.display.set_caption("ALien Invasion")

    # Создание экземпляров для хранения статистики
    # и панели результатов.
    self.stats = GameStats(self)
    self.sb = Scoreboard(self)
    ...
```

Затем панель результатов выводится на экран в `_update_screen()`:

alien_invasion.py

```
def _update_screen(self):
    ...
    self.aliens.draw(self.screen)

        # Вывод информации о счете.
        self.sb.show_score()

    # Кнопка Play отображается в том случае, если игра неактивна.
    ...
```

Метод `show_score()` вызывается непосредственно перед отображением кнопки Play.

Если запустить Alien Invasion сейчас, в правом верхнем углу экрана отображается счет 0. (Пока мы просто хотим убедиться в том, что счет отображается в нужном месте, прежде чем заниматься дальнейшей доработкой системы подсчета очков.) На рис. 14.2 изображено окно игры перед ее началом.

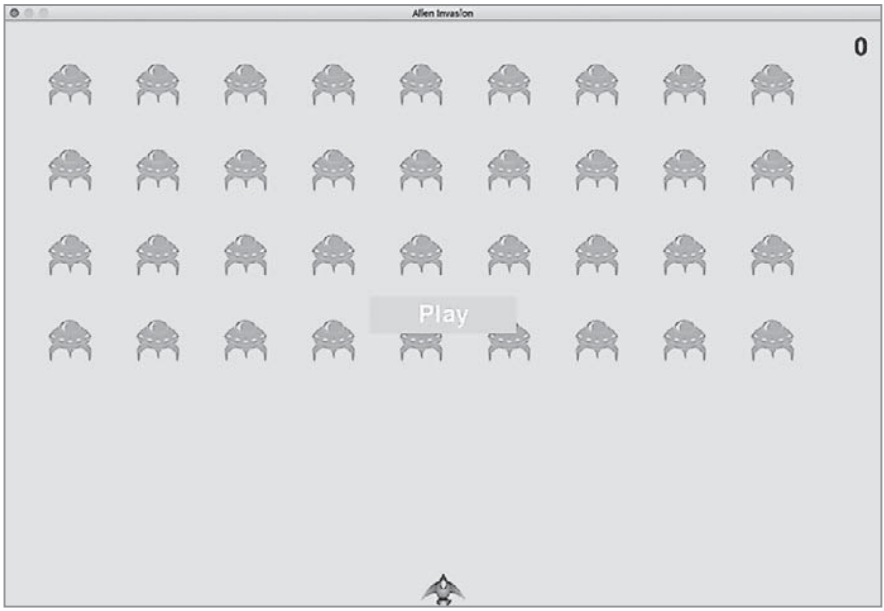

Рис. 14.2. Счет отображается в правом верхнем углу экрана

А теперь нужно организовать начисление очков за каждого пришельца!

Обновление счета при уничтожении пришельцев

Чтобы на экране выводился оперативно обновляемый счет, мы будем обновлять значение `stats.score` при каждом попадании в пришельца, а затем вызывать `prep_score()` для обновления изображения счета. Но сначала нужно определить, сколько очков игрок будет получать за каждого пришельца:

settings.py

```
def initialize_dynamic_settings(self):
    ...

    # Подсчет очков
    self.alien_points = 50
```

Стоимость каждого пришельца в очках будет увеличиваться по ходу игры. Чтобы значение сбрасывалось в начале каждой новой игры, мы задаем значение в `initialize_dynamic_settings()`.

Счет будет обновляться за каждого сбитого пришельца в `_check_bullet_alien_collisions()`:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    """Обработка коллизий снарядов с пришельцами."""
    # Удаление снарядов и пришельцев, участвующих в коллизиях.
    collisions = pygame.sprite.groupcollide(
            self.bullets, self.aliens, True, True)

    if collisions:
        self.stats.score += self.settings.alien_points
        self.sb.prep_score()
    ...
```

При попадании снаряда в пришельца Pygame возвращает словарь `collisions`. Программа проверяет, существует ли словарь, и если существует, стоимость пришельца добавляется к счету. Затем вызов `prep_score()` создает новое изображение для обновленного счета.

Теперь во время игры вы сможете набирать очки!

Сброс счета

В текущей версии счет обновляется только после попадания в пришельца; в основном такой подход работает нормально. Но старый счет выводится и после попадания в первого пришельца в новой игре.

Проблема решается инициализацией счета при создании новой игры:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    ...
    if button_clicked and not self.stats.game_active:
        ...
        # сброс игровой статистики.
        self.stats.reset_stats()
        self.stats.game_active = True
        self.sb.prep_score()
        ...
```

Метод `prep_score()` вызывается при сбросе игровой статистики в начале новой игры. Счет, выводимый на экран, обнуляется.

Начисление очков за все попадания

В том виде, в каком написан код, некоторые пришельцы будут пропускаться при подсчете. Например, если два снаряда попадают в пришельцев во время одного прохода цикла или если вы создадите широкий «снаряд» для поражения несколь-

ких пришельцев одновременно, игрок получит очки только за одного подбитого пришельца. Чтобы устранить этот недостаток, нужно доработать механизм обнаружения коллизий между снарядами и пришельцами.

В коде `_check_bullet_alien_collisions()` любой снаряд, столкнувшийся с пришельцем, становится ключом словаря `collisions`. С каждым снарядом связывается значение — список пришельцев, участвующих в коллизии. Переберем словарь `collisions` и убедимся в том, что очки начисляются за каждого подбитого пришельца:

alien_invasion.py

```
  def _check_bullet_alien_collisions(self):
      ...
      if collisions:
❶         for aliens in collisions.values():
              self.stats.score += self.settings.alien_points * len(aliens)
          self.sb.prep_score()
      ...
```

Если словарь `collisions` был определен, программа перебирает все значения в словаре ❶. Вспомните, что каждое значение представляет собой список пришельцев, в которых попал один снаряд. Стоимость каждого пришельца умножается на количество пришельцев в списке, а результат прибавляется к текущему счету. Чтобы протестировать эту систему, увеличьте ширину снаряда до 300 пикселов и убедитесь в том, что игра начисляет очки за каждого пришельца, в которого попал этот большой снаряд; затем верните ширину снаряда к нормальному состоянию.

Увеличение стоимости пришельцев

Так как с каждым достижением нового уровня игра становится более сложной, за пришельцев на этих уровнях следует давать больше очков. Чтобы реализовать эту функциональность, мы добавим код, увеличивающий стоимость пришельцев при возрастании скорости игры:

settings.py

```
  class Settings():
      """Класс для хранения всех настроек игры Alien Invasion."""

      def __init__(self):
          ...
          # Темп ускорения игры
          self.speedup_scale = 1.1
          # Темп роста стоимости пришельцев
❶         self.score_scale = 1.5

          self.initialize_dynamic_settings()

      def initialize_dynamic_settings(self):
          ...
```

```
    def increase_speed(self):
        """Увеличивает настройки скорости и стоимость пришельцев."""
        self.ship_speed_factor *= self.speedup_scale
        self.bullet_speed_factor *= self.speedup_scale
        self.alien_speed_factor *= self.speedup_scale

❷       self.alien_points = int(self.alien_points * self.score_scale)
```

В программе определяется коэффициент прироста начисляемых очков, он называется `score_scale` ❶. С небольшим увеличением скорости (1,1) игра быстро усложняется, но чтобы увидеть заметную разницу в очках, необходимо изменять стоимость пришельцев в большем темпе (1,5). После увеличения скорости игры стоимость каждого попадания также увеличивается ❷. Чтобы счет возрастал на целое количество очков, в программе используется функция `int()`.

Чтобы увидеть стоимость каждого пришельца, добавьте в метод `increase_speed()` в классе `Settings` команду `print`:

settings.py

```
def increase_speed(self):
    ...
    self.alien_points = int(self.alien_points * self.score_scale)
    print(self.alien_points)
```

Новое значение должно выводиться в терминальном окне каждый раз, когда игрок переходит на новый уровень.

ПРИМЕЧАНИЕ Убедившись, что стоимость пришельцев действительно возрастает, не забудьте удалить команду `print`; в противном случае лишний вывод повлияет на быстродействие игры и будет отвлекать игрока.

Округление счета

В большинстве аркадных «стрелялок» счет ведется значениями, кратными 10, и мы воспользуемся этой схемой в своей игре. Давайте отформатируем счет так, чтобы в больших числах группы разрядов разделялись запятыми. Изменения вносятся в классе `Scoreboard`:

scoreboard.py

```
  def prep_score(self):
      """Преобразует текущий счет в графическое изображение."""
❶     rounded_score = round(self.stats.score, -1)
❷     score_str = "{:,}".format(rounded_score)
      self.score_image = self.font.render(score_str, True,
              self.text_color, self.ai_settings.bg_color)
      ...
```

Функция `round()` обычно округляет дробное число до заданного количества знаков, переданного во втором аргументе. Но если во втором аргументе передается

отрицательное число, `round()` округляет значение до ближайших десятков, сотен, тысяч и т. д. Код ❶ приказывает Python округлить значение `stats.score` до десятков и сохранить его в `rounded_score`.

В точке ❷ директива форматирования строки приказывает Python вставить запятые при преобразовании числового значения в строку — например, чтобы вместо 1000000 выводилась строка 1,000,000. Теперь при запуске игры всегда будет отображаться аккуратно отформатированный, округленный счет (рис. 14.3).

Рис. 14.3. Округленный счет с разделителями разрядов

Рекорды

Каждый игрок желает превзойти предыдущий рекорд игры, поэтому мы будем отслеживать и выводить рекорды, чтобы у игрока была ясная цель. Рекорды будут храниться в классе `GameStats`:

game_stats.py

```
def __init__(self, ai_game):
    ...
    # Рекорд не должен сбрасываться.
    self.high_score = 0
```

Так как рекорд не должен сбрасываться при повторном запуске, значение `high_score` инициализируется в `__init__()`, а не в `reset_stats()`.

Теперь изменим класс `Scoreboard` для отображения рекорда. Начнем с метода `__init__()`:

scoreboard.py

```
  def __init__(self, ai_game):
      ...
      # Подготовка изображений счетов.
      self.prep_score()
❶     self.prep_high_score()
```

Рекорд должен отображаться отдельно от текущего счета, поэтому для подготовки его изображения понадобится новый метод `prep_high_score()`❶:

scoreboard.py

```
  def prep_high_score(self):
      """Преобразует рекордный счет в графическое изображение."""
❶     high_score = round(self.stats.high_score, -1)
      high_score_str = "{:,}".format(high_score)
❷     self.high_score_image = self.font.render(high_score_str, True,
              self.text_color, self.ai_settings.bg_color)

      # Рекорд выравнивается по центру верхней стороны.
      self.high_score_rect = self.high_score_image.get_rect()
❸     self.high_score_rect.centerx = self.screen_rect.centerx
❹     self.high_score_rect.top = self.score_rect.top
```

Рекорд округляется до десятков и форматируется с запятыми ❶. Затем для рекорда строится графическое изображение ❷, выполняется горизонтальное выравнивание прямоугольника по центру экрана ❸, а атрибут `top` прямоугольника приводится в соответствие с верхней стороной изображения счета ❹.

Теперь метод `show_score()` выводит текущий счет в правом верхнем углу, а рекорд — в центре верхней стороны:

scoreboard.py

```
def show_score(self):
    """Выводит счет на экран."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
```

Для обновления рекорда в `Scoreboard` добавляется новая функция `check_high_score()`:

scoreboard.py

```
def check_high_score(self):
    """Проверяет, появился ли новый рекорд."""
    if self.stats.score > self.stats.high_score:
        self.stats.high_score = self.stats.score
        self.prep_high_score()
```

Метод `check_high_score()` сравнивает текущий счет с рекордом. Если текущий счет выше, мы обновляем значение `high_score` и вызываем `prep_high_score()` для обновления изображения рекорда.

Метод `check_high_score()` должен вызываться при каждом попадании в пришельца после обновления счета в `_check_bullet_alien_collisions()`:

alien_invasion.py

```
def check_bullet_alien_collisions(self):
    ...
    if collisions:
        for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
        self.check_high_score()
    ...
```

Метод `check_high_score()` должен вызываться только в том случае, если словарь `collisions` присутствует, причем вызов выполняется после обновления счета для всех подбитых пришельцев.

Когда вы впервые играете в Alien Invasion, текущий счет одновременно будет наивысшим, поэтому он будет отображаться и как текущий счет, и как рекорд. Но в начале второй игры ваш предыдущий рекорд должен отображаться в середине, а текущий счет справа, как показано на рис. 14.4.

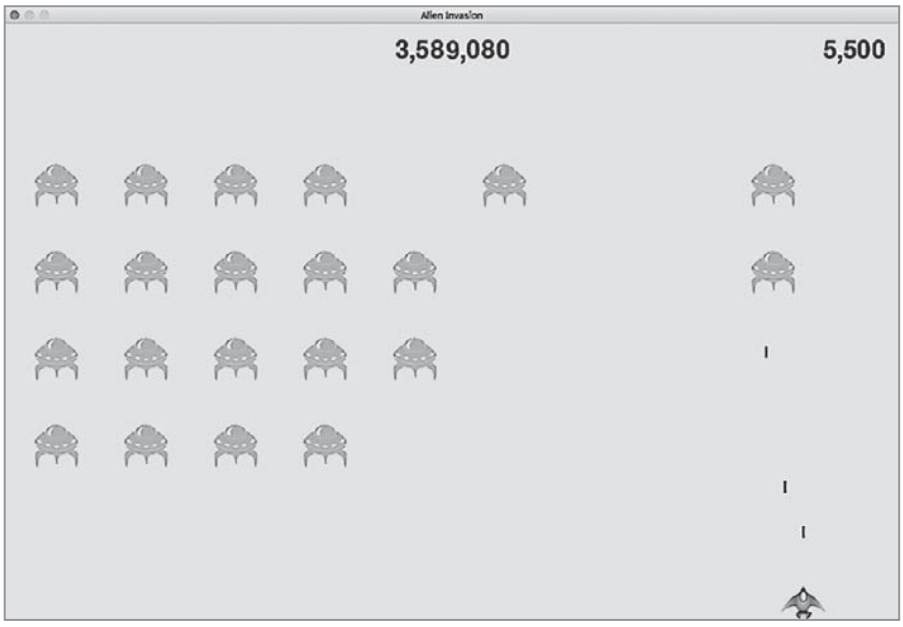

Рис. 14.4. Рекордный счет выводится в середине экрана

Вывод уровня

Чтобы в игре выводился текущий уровень, сначала в класс `GameStats` следует включить атрибут для его представления. Чтобы уровень сбрасывался в начале каждой игры, инициализируйте его в `reset_stats()`:

game_stats.py

```
def reset_stats(self):
    """Инициализирует статистику, изменяющуюся в ходе игры."""
    self.ships_left = self.ai_settings.ship_limit
    self.score = 0
    self.level = 1
```

Чтобы класс `Scoreboard` выводил текущий уровень, мы вызываем новый метод `prep_level()` из `__init__()`:

scoreboard.py

```
def __init__(self, ai_game):
    ...
    self.prep_high_score()
    self.prep_level()
```

Метод `prep_level()` выглядит так:

scoreboard.py

```
  def prep_level(self):
      """Преобразует уровень в графическое изображение."""
      level_str = str(self.stats.level)
❶     self.level_image = self.font.render(level_str, True,
              self.text_color, self.settings.bg_color)

      # Уровень выводится под текущим счетом.
      self.level_rect = self.level_image.get_rect()
❷     self.level_rect.right = self.score_rect.right
❸     self.level_rect.top = self.score_rect.bottom + 10
```

Метод `prep_level()` создает изображение на базе значения, хранящегося в `stats.level` ❶, и приводит атрибут `right` изображения в соответствие с атрибутом `right` счета ❷. Затем атрибут `top` сдвигается на 10 пикселов ниже нижнего края изображения текущего счета, чтобы между счетом и уровнем оставался пустой интервал ❸.

В метод `show_score()` также необходимо внести изменения:

scoreboard.py

```
def show_score(self):
    """Выводит текущий счет, рекорд и число оставшихся кораблей."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
    self.screen.blit(self.level_image, self.level_rect)
```

Добавленная строка выводит на экран изображение, представляющее уровень.

Увеличение `stats.level` и обновление изображения уровня выполняются в `_check_bullet_alien_collisions()`:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    ...
    if not self.aliens:
        # Уничтожить существующие снаряды и создать новый флот.
        self.bullets.empty()
        self.create_fleet()
        self.settings.increase_speed()

        # Увеличение уровня.
        self.stats.level += 1
        self.sb.prep_level()
```

Если все пришельцы уничтожены, программа увеличивает значение `stats.level` и вызывает `prep_level()` для обновления уровня.

Чтобы убедиться в том, что изображения текущего счета и уровня правильно обновляются в начале новой игры, мы вызываем `prep_level()` при нажатии кнопки Play:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    ...
    if button_clicked and not stats.game_active:
        ...
        self.sb.prep_score()
        sb.prep_level()
        ...
```

Метод `prep_level()` вызывается сразу же после вызова `prep_score()`.

Теперь количество пройденных уровней отображается на экране (рис. 14.5).

ПРИМЕЧАНИЕ В некоторых классических играх выводимая информация снабжается текстовыми метками: «Уровень», «Рекорд» и т. д. Мы эти метки опустили, потому что смысл каждого числа понятен каждому, кто сыграл в Alien Invasion. Если вы включили эти метки, добавьте их в строки непосредственно перед вызовами `font.render()` в Scoreboard.

Вывод количества кораблей

Остается вывести количество кораблей, оставшихся у игрока, но на этот раз информация будет выводиться в графическом виде. Как во многих классических аркадных играх, в левом верхнем углу экрана программа рисует несколько изображений корабля. Каждый корабль обозначает одну оставшуюся попытку.

Рис. 14.5. Текущий уровень выводится под текущим счетом

Для начала нужно сделать так, чтобы класс `Ship` наследовал от `Sprite` — это необходимо для создания группы кораблей:

ship.py

```
  import pygame
  from pygame.sprite import Sprite

❶ class Ship(Sprite):
      # Класс для управления кораблем.

      def __init__(self, ai_game):
          """Инициализирует корабль и задает его начальную позицию."""
❷         super().__init__()
          ...
```

Здесь мы импортируем `Sprite`, объявляем о наследовании `Ship` от `Sprite` ❶ и вызываем `super()` в начале `__init__()` ❷.

Далее необходимо изменить `Scoreboard` и создать группу кораблей для вывода на экран. Команды `import` выглядят так:

scoreboard.py

```
import pygame.font
from pygame.sprite import Group

from ship import Ship
```

Так как мы собираемся создать группу кораблей, программа импортирует классы `Group` и `Ship`.

Метод `__init__()` выглядит так:

scoreboard.py

```
def __init__(self, ai_game):
    """Инициализирует атрибуты подсчета очков."""
    self.ai_game = ai_game
    self.screen = ai_game.screen
    ...
    self.prep_level()
    self.prep_ships()
```

Экземпляр игры присваивается атрибуту, так как он понадобится нам для создания кораблей. Метод `prep_ships()` будет вызываться после `prep_level()`. Он выглядит так:

scoreboard.py

```
  def prep_ships(self):
      """Сообщает количество оставшихся кораблей."""
❶     self.ships = Group()
❷     for ship_number in range(self.stats.ships_left):
          ship = Ship(self.ai_game)
❸         ship.rect.x = 10 + ship_number * ship.rect.width
❹         ship.rect.y = 10
❺         self.ships.add(ship)
```

Метод `prep_ships()` создает пустую группу `self.ships` для хранения экземпляров кораблей ❶. В ходе заполнения этой группы цикл выполняется по одному разу для каждого корабля, оставшегося у игрока ❷. В цикле создается новый корабль, а координата `x` этого корабля задается так, чтобы корабли размещались рядом друг с другом, разделенные интервалами величиной 10 пикселов ❸. Координата `y` задается так, чтобы корабли были смещены на 10 пикселов от верхнего края экрана и были выровнены по изображению текущего счета ❹. Наконец, каждый корабль добавляется в группу `ships` ❺.

Следующим шагом становится вывод кораблей на экран:

scoreboard.py

```
def show_score(self):
    """Выводит очки, уровень и количество кораблей на экран."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
    self.screen.blit(self.level_image, self.level_rect)
    self.ships.draw(self.screen)
```

При выводе кораблей на экран мы вызываем метод `draw()` для группы, а Pygame рисует каждый отдельный корабль.

Чтобы игрок видел, сколько попыток у него в начале игры, мы вызываем `prep_ships()` при запуске новой игры. Это происходит в функции `_check_play_button()` в `AlienInvasion`:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    ...
    if button_clicked and not self.stats.game_active:
        ...
        self.sb.prep_score()
        self.sb.prep_level()
        self.sb.prep_ships()
        ...
```

Метод `prep_ships()` также вызывается при столкновении пришельца с кораблем, чтобы изображение обновлялось при потере корабля:

alien_invasion.py

```
def _ship_hit(self):
    """Обрабатывает столкновение корабля с пришельцем."""
    if self.stats.ships_left > 0:
        # Уменьшение ships_left и обновление панели счета
        self.stats.ships_left -= 1
        self.sb.prep_ships()
        ...
```

Метод `prep_ships()` вызывается после уменьшения значения `ships_left`, так что при каждой потере корабля выводится правильное количество изображений.

На рис. 14.6 показана полная игровая информация на экране, с количеством оставшихся кораблей в левой верхней части экрана.

УПРАЖНЕНИЯ

14.5. Исторический рекорд: в текущей версии рекорд сбрасывается каждый раз, когда игрок закрывает и перезапускает Alien Invasion. Чтобы этого не происходило, запишите рекорд в файл перед вызовом `sys.exit()` и загрузите его при инициализации значения в GameStats.

14.6. Рефакторинг: найдите функции и методы, которые решают более одной задачи, и проведите рефакторинг, улучшающий структуру и эффективность кода. Например, переместите часть кода функции `_check_bullet_alien_collisions()`, которая запускает новый уровень при уничтожении флота, в функцию `start_new_level()`. Также переместите четыре метода, вызываемых в методе `__init__()` класса Scoreboard, в метод `prep_images()` для сокращения длины `__init__()`. Метод `prep_images()` также может оказать помощь `_check_play_button()` или `start_game()`, если вы уже провели рефакторинг `_check_play_button()`.

ПРИМЕЧАНИЕ Прежде чем браться за рефакторинг проекта, обратитесь к приложению Г. В нем рассказано, как восстановить рабочее состояние проекта, если в ходе рефакторинга были допущены ошибки.

Рис. 14.6. Полная игровая информация в Alien Invasion

14.7. Расширение Alien Invasion: подумайте над возможными расширениями Alien Invasion. Например, пришельцы тоже могут стрелять по кораблю, или же вы можете добавить укрытия, за которыми может скрываться корабль (укрытия могут разрушаться снарядами с обеих сторон). Или добавьте звуковые эффекты (например, взрывы или звуки выстрелов) средствами модуля `pygame.mixer`.

14.8. Боковая стрельба, финальная версия: продолжайте разрабатывать приложение с боковой стрельбой, используя все, чему вы научились в этом проекте. Добавьте кнопку Play, обеспечьте ускорение игры в нужных местах и разработайте систему начисления очков. Не забывайте проводить рефакторинг в процессе работы и ищите возможности настройки игры за рамками того, что было показано в этой главе.

Итоги

В этой главе вы узнали, как создать кнопку для запуска новой игры, как обнаруживать события мыши и скрывать указатель мыши в активных играх. Полученные знания помогут вам создать другие кнопки в играх, например кнопку для вывода инструкций по игре. Также вы научились изменять скорость по ходу игры, создавать прогрессивную систему подсчета очков и выводить информацию в текстовом и графическом виде.

Проект 2
Визуализация данных

15 Генерирование данных

Под *визуализацией данных* понимается исследование данных через их визуальное представление. Визуализация тесно связана с *анализом данных* (data mining), использующим программный код для изучения закономерностей и связей в наборе данных. Набором данных может быть как маленький список чисел, помещающийся в одной строке кода, так и массив из многих гигабайтов.

Качественное представление данных не сводится к красивой картинке. Если для набора данных подобрано простое, визуально привлекательное представление, его смысл становится очевидным для зрителя. Люди замечают в наборе данных закономерности, о которых они и не подозревали.

К счастью, для визуализации сложных данных не нужен суперкомпьютер. Благодаря эффективности Python вы сможете быстро исследовать наборы данных из миллионов отдельных *элементов данных* (точек данных) на обычном ноутбуке. Элементы данных даже не обязаны быть числовыми. Приемы, о которых вы узнали в части I книги, позволят вам проанализировать даже нечисловые данные.

Python используется для обработки данных в генетике, исследовании климата, политическом и экономическом анализе и множестве других областей. Специалисты по обработке данных написали на Python впечатляющий инструментарий визуализации и анализа, и многие из этих разработок также доступны и для вас. Один из самых популярных инструментов такого рода — `matplotlib`, математическая библиотека построения диаграмм. С помощью `matplotlib` можно строить простые диаграммы, графики, диаграммы разброса данных и т. д. После этого будет создан более интересный набор данных, основанный на концепции *случайного блуждания* — визуализации, генерируемой на основе серии случайных решений.

Также в этом проекте будет использоваться пакет Plotly, ориентированный на создание визуализаций, хорошо работающих с цифровыми устройствами. Plotly генерирует визуализации, автоматически масштабируемые по размерам экранов различных цифровых устройств. Визуализации также могут включать различные интерактивные возможности, например выделение различных аспектов данных набора данных при наведении указателя мыши на разные части визуализации. Мы используем Plotly для исследования закономерностей различных бросков кубиков.

Установка matplotlib

Чтобы использовать библиотеку Matplotlib для исходных визуализаций, необходимо установить ее при помощи `pip` — модуля для загрузки и установки пакетов Python. Введите следующую команду в приглашении терминала:

```
$ python -m pip install --user matplotlib
```

Эта команда приказывает Python запустить модуль `pip` и добавить пакет `matplotlib` к установке Python текущего пользователя. Если вы используете для запуска программ или запуска терминального сеанса другую команду вместо `python` (например, `python3`), ваша команда будет выглядеть так:

```
$ python3 -m pip install --user matplotlib
```

ПРИМЕЧАНИЕ Если команда не работает в macOS, попробуйте запустить ее без флага `--user`.

Чтобы получить представление о визуализациях, которые можно построить средствами Matplotlib, посетите галерею по адресу https://matplotlib.org/gallery/. Щелкая на визуализации в галерее, вы сможете просмотреть код, использованный для ее построения.

Построение простого графика

Начнем с построения простого линейного графика с использованием Matplotlib, а затем настроим его для более содержательной визуализации данных. В качестве данных для графика будет использоваться последовательность квадратов 1, 4, 9, 16 и 25.

Передайте Matplotlib числа так, как показано ниже, а Matplotlib сделает все остальное:

mpl_squares.py

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]
❶ fig, ax = plt.subplots()
ax.plot(squares)

plt.show()
```

Сначала импортируйте модуль `pyplot` с псевдонимом `plt`, чтобы вам не приходилось многократно вводить имя `pyplot`. (Это сокращение часто встречается в примерах на сайте, поэтому мы поступим так же.) Модуль `pyplot` содержит ряд функций для построения диаграмм и графиков.

Мы создаем список `squares` для хранения данных, которые будут наноситься на график. Затем используется еще одно общепринятое соглашение Matplotlib — вызов функции `subplots()`❶. Эта функция позволяет сгенерировать одну или несколько поддиаграмм на одной диаграмме. Переменная `fig` представляет весь рисунок или набор генерируемых диаграмм. Переменная `ax` представляет одну диаграмму на рисунке; эта переменная будет использоваться чаще всего в нашем примере.

Затем вызывается функция `plot()`, которая пытается построить осмысленное графическое представление для заданных чисел. Вызов `plt.show()` открывает окно просмотра Matplotlib и выводит график (рис. 15.1). В окне просмотра можно изменять масштаб и перемещаться по построенному графику, а кнопка с диском позволяет сохранить любое изображение по вашему выбору.

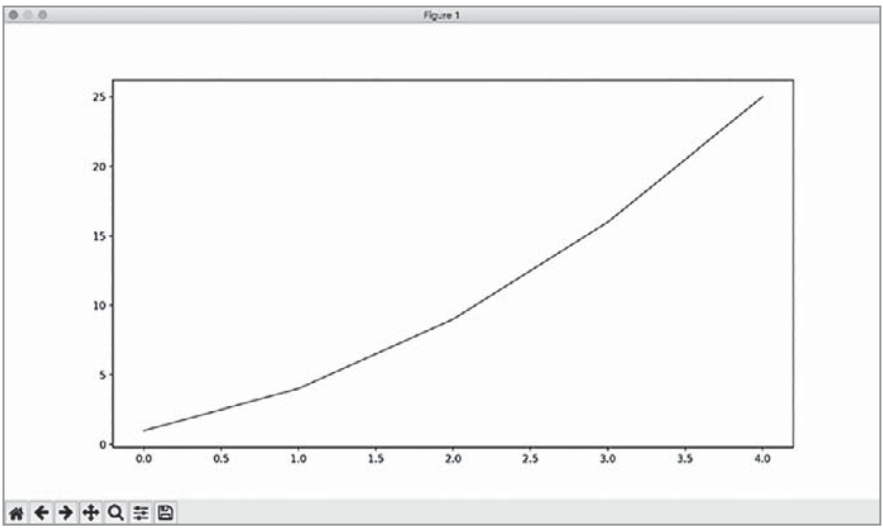

Рис. 15.1. Пример простейшего графика в Matplotlib

Изменение типа надписей и толщины графика

Хотя из графика на рис. 15.1 видно, что числовая последовательность возрастает, текст надписей слишком мелкий, а линия слишком тонкая. К счастью, Matplotlib позволяет настроить практически каждый аспект визуализации.

Мы используем эти возможности настройки для того, чтобы сделать график более выразительным:

mpl_squares.py

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]
```

```
  fig, ax = plt.subplots()
❶ ax.plot(squares, linewidth=3)

  # Назначение заголовка диаграммы и меток осей.
❷ ax.set_title("Square Numbers", fontsize=24)
❸ ax.set_xlabel("Value", fontsize=14)
  ax .set_ylabel("Square of Value", fontsize=14)

  # Назначение размера шрифта делений на осях.
❹ ax.tick_params(axis='both', labelsize=14)

  plt.show()
```

Параметр `linewidth` ❶ управляет толщиной линии, которая строится вызовом `plot()`. Метод `set_title()` ❷ назначает заголовок диаграммы. Параметры `fontsize`, неоднократно встречающиеся в коде, управляют размером текста различных элементов диаграммы.

Методы `xlabel()` и `ylabel()` позволяют назначить метки (заголовки) каждой из осей ❸, а функция `tick_params()` определяет оформление делений на осях ❹. Аргументы, использованные в данном примере, относятся к делениям на обеих осях (`axis='both'`) и устанавливают для меток делений размер шрифта 14 (`labelsize=14`).

Как видно из рис. 15.2, график выглядит гораздо лучше. Текст надписей стал крупнее, а линия графика толще. Часто стоит поэкспериментировать с этими значениями, чтобы получить представление о том, какой вариант оформления будет лучше смотреться на полученной диаграмме.

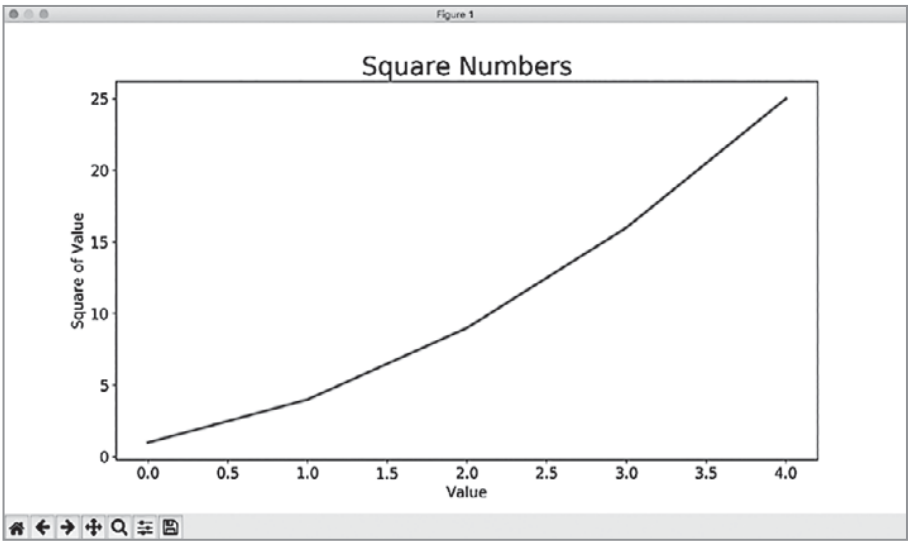

Рис. 15.2. График выглядит гораздо лучше

Корректировка графика

Теперь, когда текст на графике стал нормально читаться, мы видим, что данные помечены неправильно. Обратите внимание: для точки 4,0 в конце графика указан квадрат 25! Давайте исправим эту проблему.

Если `plot()` передается числовая последовательность, функция считает, что первый элемент данных соответствует координате `x` со значением 0, но в нашем примере первая точка соответствует значению 1. Чтобы переопределить значение по умолчанию, передайте `plot()` как входные значения, так и квадраты:

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

fig, ax = plt.subplots()
ax.plot(input_values, squares, linewidth=3)

# Назначение заголовка диаграммы и меток осей.
...
```

Теперь `plot()` правильно строит график, потому что мы предоставили оба набора значений и функции не нужно предполагать, как был сгенерирован выходной набор чисел. На рис. 15.3 изображен правильный график.

При вызове `plot()` можно передавать многочисленные аргументы, а также использовать различные функции для настройки графиков. Знакомство с этими

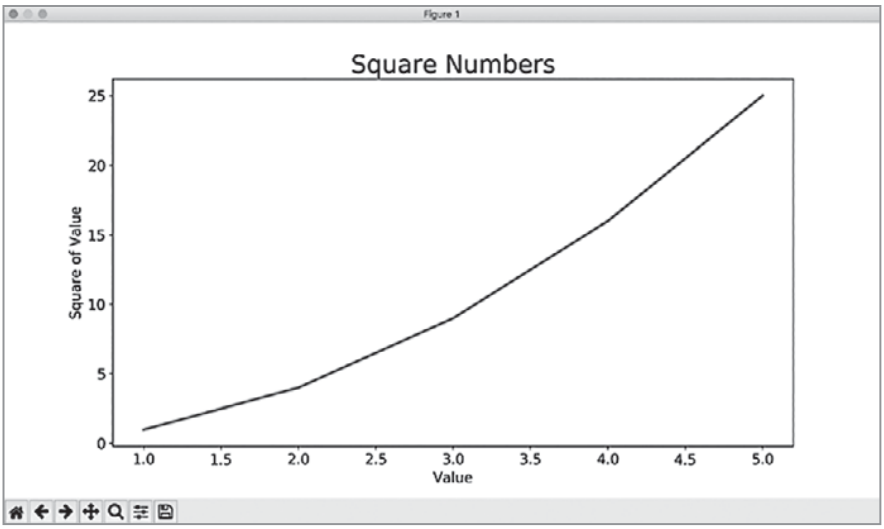

Рис. 15.3. График с правильными данными

функциями продолжится позднее, когда мы начнем работать с более интересными наборами данных в этой главе.

Встроенные стили

В Matplotlib существует целый ряд заранее определенных стилей оформления с хорошей подборкой настроек для цвета фона, линий сетки, толщины линий, шрифтов, размера шрифтов и т. д.; готовые настройки позволят вам создавать привлекательные диаграммы без возни с настройкой. Чтобы узнать, какие стили доступны в вашей системе, выполните следующие команды в терминальном сеансе:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.available
['seaborn-dark', 'seaborn-darkgrid', 'seaborn-ticks', 'fivethirtyeight',
...
```

Чтобы использовать эти стили, добавьте одну строку кода перед построением диаграммы:

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
...
```

Этот код строит график, изображенный на рис. 15.4. Существует множество разнообразных стилей; поэкспериментируйте и найдите те, которые вам больше нравятся.

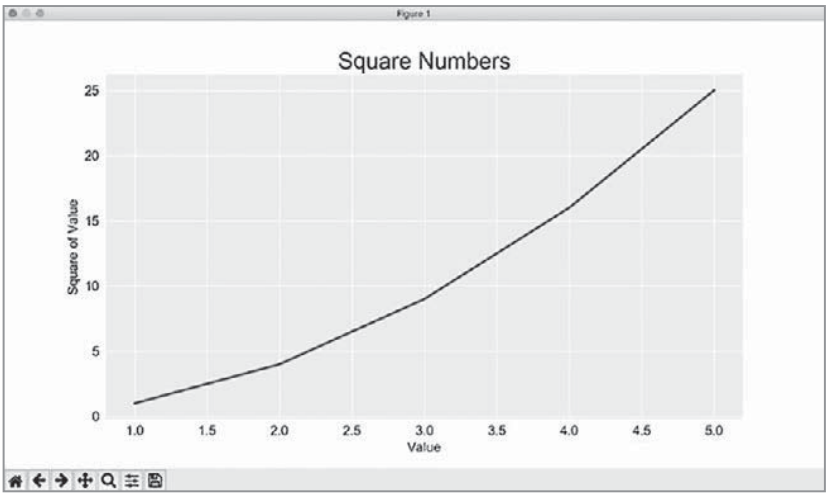

Рис. 15.4. Встроенный стиль seaborn

Нанесение и оформление отдельных точек функцией scatter()

Иногда бывает полезно нанести на график отдельные точки, основанные на некоторых характеристиках, и определить их оформление. Например, на графике малые и большие значения могут отображаться разными цветами. Возможны и другие варианты: например, сначала нанести множество точек с одним типом оформления, а затем выделить отдельные точки набора, перерисовав их с другим оформлением.

Для нанесения на диаграмму отдельной точки используется функция `scatter()`. Передайте `scatter()` координаты (x, y) нужной точки, и функция нанесет эти значения на диаграмму:

scatter_squares.py

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(2, 4)

plt.show()
```

Применим оформление, чтобы результат выглядел более интересно. Мы добавим название, метки осей, а также увеличим шрифт, чтобы текст нормально читался:

```
  import matplotlib.pyplot as plt
  plt.style.use('seaborn')
  fig, ax = plt.subplots()
❶ ax.scatter(2, 4, s=200)

  # Назначение заголовка диаграммы и меток осей.
  ax.set_title("Square Numbers", fontsize=24)
  ax.set_xlabel("Value", fontsize=14)
  ax.set_ylabel("Square of Value", fontsize=14)

  # Назначение размера шрифта делений на осях.
  ax.tick_params(axis='both', which='major', labelsize=14)

  plt.show()
```

В точке ❶ вызывается функция; аргумент `s` задает размер точек, используемых для рисования диаграммы. Если запустить программу scatter_squares.py в текущем состоянии, вы увидите одну точку в середине диаграммы (рис. 15.5).

Вывод серии точек функцией scatter()

Чтобы вывести на диаграмме серию точек, передайте `scatter()` списки значений координат x и y:

scatter_squares.py

```
import matplotlib.pyplot as plt
```

```
x_values = [1, 2, 3, 4, 5]
y_values = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, y_values, s=100)

# Назначение заголовка диаграммы и меток осей.
...
```

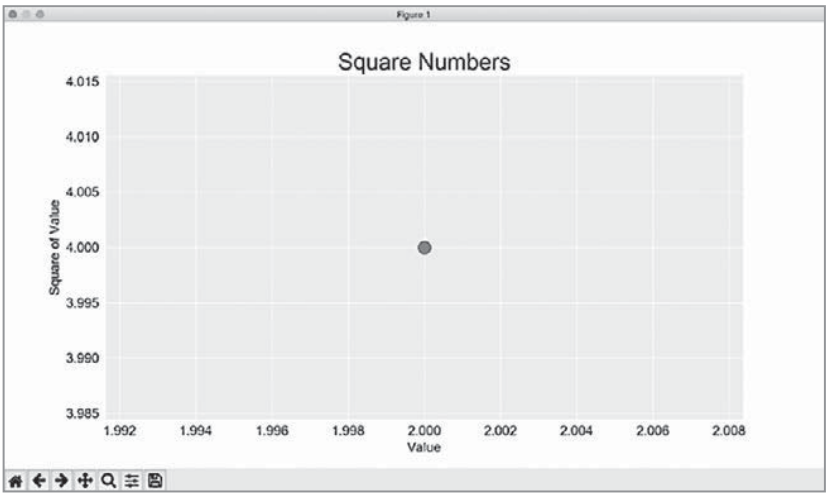

Рис. 15.5. Вывод одной точки

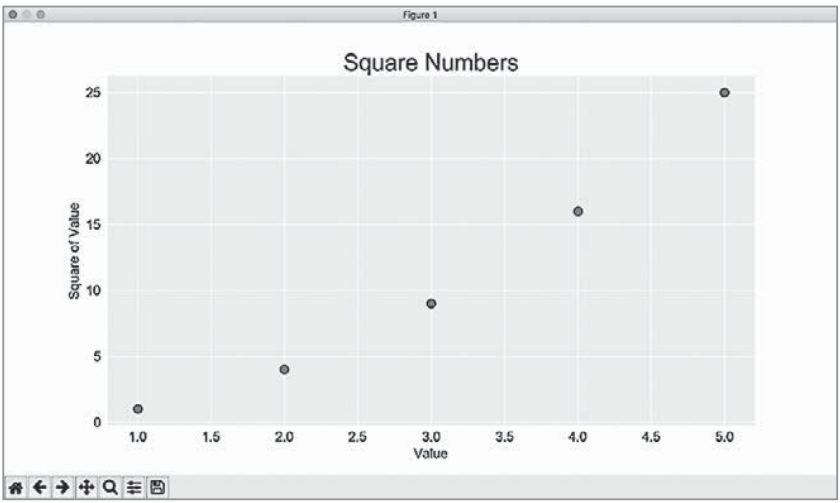

Рис. 15.6. Точечная диаграмма с несколькими точками

Список `x_values` содержит числа, возводимые в квадрат, а в `y_values` содержатся квадраты. При передаче этих списков `scatter()` библиотека Matplotlib читает по одному значению из каждого списка и наносит их на диаграмму как точку. Таким образом, на диаграмму будут нанесены точки (1, 1), (2, 4), (3, 9), (4, 16) и (5, 25); результат показан на рис. 15.6.

Автоматическое вычисление данных

Строить списки вручную неэффективно, особенно при большом объеме данных. Вместо того чтобы передавать данные в виде списка, мы воспользуемся циклом Python, который выполнит вычисления за нас. Вот как выглядит такой цикл для 1000 точек:

scatter_squares.py

```
import matplotlib.pyplot as plt

❶ x_values = list(range(1, 1001))
  y_values = [x**2 for x in x_values]

  plt.style.use('seaborn')
  fig, ax = plt.subplots()
❷ ax.scatter(x_values, y_values, s=10)

  # Назначение заголовка диаграммы и меток осей.
  ...

  # Назначение диапазона для каждой оси.
❸ ax.axis([0, 1100, 0, 1100000])

  plt.show()
```

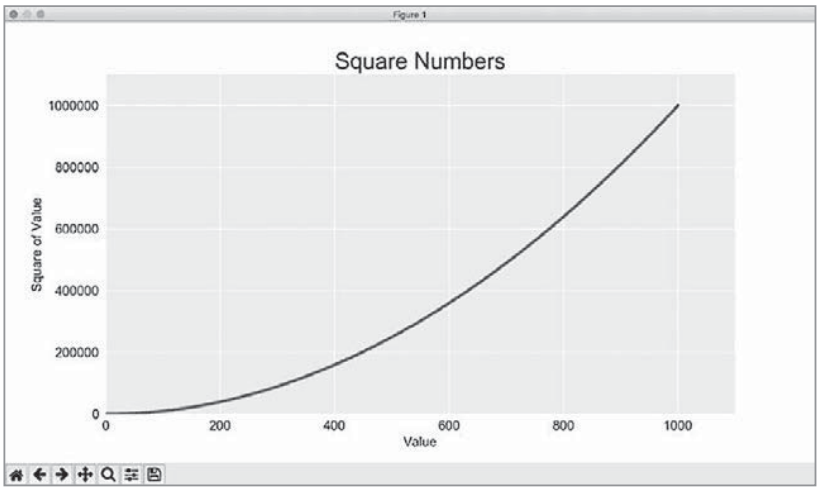

Рис. 15.7. Диаграмма с 1000 точками строится так же легко, как и диаграмма с 5 точками

Все начинается со списка значений координаты x с числами от 1 до 1000 ❶. Затем генератор списка строит значения y, перебирая значения x (for x in x_values), возводя каждое число в квадрат (x**2) и сохраняя результаты в y_values. Затем оба списка (входной и выходной) передаются scatter() ❷. Так как набор данных велик, мы используем меньший размер точек.

В точке ❸ метод axis() используется для задания диапазона каждой оси. Метод axis() получает четыре значения: минимум и максимум по осям x и y. В данном случае по оси x откладывается диапазон от 0 до 1100, а по оси y — диапазон от 0 до 1 100 000. На рис. 15.7 показан результат.

Определение пользовательских цветов

Чтобы изменить цвет точек, передайте scatter() аргумент c с именем используемого цвета, заключенным в одинарные кавычки:

```
ax.scatter(x_values, y_values, c='red', s=10)
```

Также возможно определять пользовательские цвета в цветовой модели RGB. Чтобы определить цвет, передайте аргумент c с кортежем из трех дробных значений (для красной, зеленой и синей составляющих) в диапазоне от 0 до 1. Например, следующая строка создает диаграмму со светло-зелеными точками:

```
ax.scatter(x_values, y_values, c=(0, 0.8, 0), s=10)
```

Значения, близкие к 0, дают более темные цвета, а со значениями, близкими к 1, цвета получаются более светлыми.

Цветовые карты

Цветовая карта (colormap) представляет собой серию цветов градиента, определяющую плавный переход от начального цвета к конечному. Цветовые карты используются в визуализациях для выделения закономерностей в данных. Например, малые значения можно обозначить светлыми цветами, а большие — темными.

Модуль pyplot включает набор встроенных цветовых карт. Чтобы воспользоваться одной из готовых карт, вы должны указать, как модуль pyplot должен присваивать цвет каждой точке набора данных. В следующем примере цвет каждой точки присваивается на основании значения y:

scatter_squares.py

```
import matplotlib.pyplot as plt

x_values = list(range(1001))
y_values = [x**2 for x in x_values]

ax.scatter(x_values, y_values, c=y_values, cmap=plt.cm.Blues, s=10)

# Назначение заголовка диаграммы и меток осей.
...
```

Мы передаем в c список значений по оси y, а затем указываем pyplot, какая цветовая карта должна использоваться, при помощи аргумента cmap. Следующий код окрашивает точки с меньшими значениями y в светло-синий цвет, а точки с большими значениями y — в темно-синий цвет. Полученная диаграмма изображена на рис. 15.8.

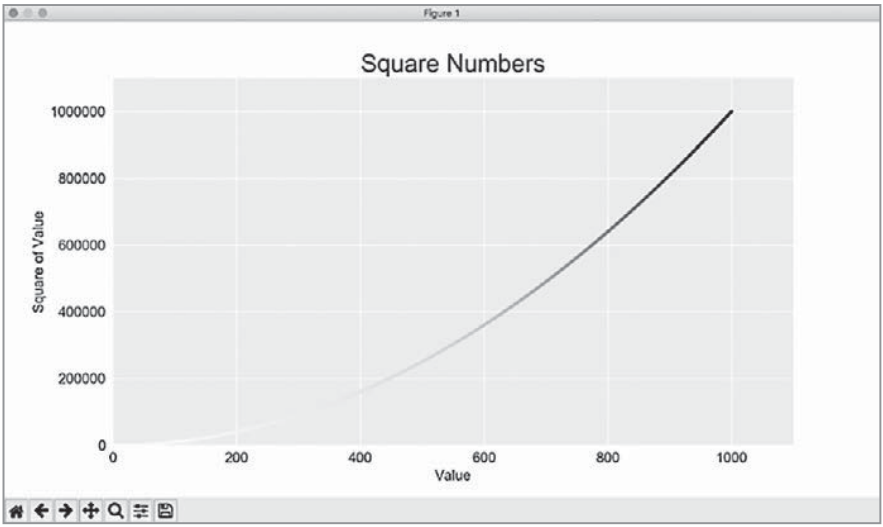

Рис. 15.8. Точечная диаграмма с цветовой картой Blues

ПРИМЕЧАНИЕ Все цветовые карты, доступные в pyplot, можно просмотреть на сайте http://matplotlib.org/; откройте раздел Examples, прокрутите содержимое до пункта Color и щелкните на ссылке Colormaps_reference.

Автоматическое сохранение диаграмм

Если вы хотите, чтобы программа автоматически сохраняла диаграмму в файле, замените вызов `plt.show()` вызовом `plt.savefig()`:

```
plt.savefig('squares_plot.png', bbox_inches='tight')
```

Первый аргумент содержит имя файла для сохранения диаграммы; файл будет сохранен в одном каталоге с scatter_squares.py. Второй аргумент отсекает от диаграммы лишнее пространство. Если вы хотите оставить пустые места вокруг диаграммы, этот аргумент можно опустить.

УПРАЖНЕНИЯ

15.1. Кубы: число, возведенное в третью степень, называется «кубом». Нанесите на диаграмму первые пять кубов, а затем первые 5000 кубов.

15.2. Цветные кубы: примените цветовую карту к диаграмме с кубами.

Случайное блуждание

В этом разделе мы используем Python для генерирования данных для случайного обхода, а затем при помощи Matplotlib создадим привлекательное представление сгенерированных данных. *Случайным блужданием* (random walk) называется путь, который не имеет четкого направления, но определяется серией полностью случайных решений. Представьте, что муравей сошел с ума и делает каждый новый шаг в случайном направлении; его путь напоминает случайное блуждание.

Случайное блуждание находит практическое применение в естественных науках: физике, биологии, химии и экономике. Например, пыльцевое зерно на поверхности водяной капли движется по поверхности воды, потому что его постоянно подталкивают молекулы воды. Молекулярное движение в капле воды случайно, поэтому путь пыльцевого зерна на поверхности представляет собой случайное блуждание. Код, который мы напишем, найдет применение при моделировании многих реальных ситуаций.

Создание класса RandomWalk()

Чтобы создать путь случайного блуждания, мы напишем класс `RandomWalk`, который принимает случайные решения по выбору направления. Классу нужны три атрибута: переменная для хранения количества точек в пути и два списка для координат `x` и `y` каждой точки.

Класс `RandomWalk` содержит всего два метода: `__init__()` и `fill_walk()` для вычисления точек случайного блуждания. Начнем с метода `__init__()`:

random_walk.py

```
❶ from random import choice

  class RandomWalk():
      """Класс для генерирования случайных блужданий."""

❷     def __init__(self, num_points=5000):
          """Инициализирует атрибуты блуждания."""
          self.num_points = num_points

          # Все блуждания начинаются с точки (0, 0).
❸         self.x_values = [0]
          self.y_values = [0]
```

Чтобы принимать случайные решения, мы сохраним возможные варианты в списке и используем функцию `choice()` из модуля `random` для принятия решения ❶. Затем для списка устанавливается количество точек по умолчанию, равное 5000, — достаточно большое, чтобы генерировать интересные закономерности, но достаточно малое, чтобы блуждания генерировались быстро ❷. Затем в точке ❸ создаются два списка для хранения значений x и y, после чего каждый путь начинается с точки $(0, 0)$.

Выбор направления

Метод `fill_walk()`, как показано ниже, заполняет путь точками и определяет направление каждого шага. Добавьте этот метод в random_walk.py:

random_walk.py

```
  def fill_walk(self):
      """Вычисляет все точки блуждания."""

      # Шаги генерируются до достижения нужной длины.
❶     while len(self.x_values) < self.num_points:

          # Определение направления и длины перемещения.
❷         x_direction = choice([1, -1])
          x_distance = choice([0, 1, 2, 3, 4])
❸         x_step = x_direction * x_distance

          y_direction = choice([1, -1])
          y_distance = choice([0, 1, 2, 3, 4])
❹         y_step = y_direction * y_distance

          # Отклонение нулевых перемещений.
❺         if x_step == 0 and y_step == 0:
              continue

          # Вычисление следующих значений x и y.
❻         x = self.x_values[-1] + x_step
          y = self.y_values[-1] + y_step

          self.x_values.append(x)
          self.y_values.append(y)
```

В точке ❶ запускается цикл, который выполняется вплоть до заполнения пути правильным количеством точек. Главная часть метода `fill_walk()` сообщает Python, как следует моделировать четыре случайных решения: двигаться вправо или влево? как далеко идти в этом направлении? двигаться вверх или вниз? как далеко идти в этом направлении?

Выражение `choice([1, -1])` выбирает значение `x_direction`; оно возвращает 1 для перемещения вправо или –1 для движения влево ❷. Затем выражение `choice([0, 1, 2, 3, 4])` определяет дальность перемещения в этом направлении (`x_distance`) случайным выбором целого числа от 0 до 4. (Включение 0 позволяет выполнять шаги по оси `y`, а также шаги со смещением по обеим осям.)

В точках ❸ и ❹ определяется длина каждого шага в направлениях `x` и `y`, для чего направление движения умножается на выбранное расстояние. При положительном результате `x_step` смещает вправо, при отрицательном — влево и при нулевом — вертикально. При положительном результате `y_step` смещает вверх, при отрицательном — вниз и при нулевом — горизонтально. Если оба значения, `x_step` и `y_step`, равны 0, то блуждание останавливается, но цикл продолжается ❺.

Чтобы получить следующее значение x, мы прибавляем значение x_step к последнему значению, хранящемуся в x_values ❻, и делаем то же самое для значений y. После того как значения будут получены, они присоединяются к x_values и y_values.

Вывод случайного блуждания

Ниже приведен код отображения всех точек блуждания:

rw_visual.py

```
  import matplotlib.pyplot as plt

  from random_walk import RandomWalk

  # Построение случайного блуждания.
❶ rw = RandomWalk()
  rw.fill_walk()

  # Нанесение точек на диаграмму.
  plt.style.use('classic')
  fig, ax = plt.subplots()
❷ ax.scatter(rw.x_values, rw.y_values, s=15)
  plt.show()
```

Сначала программа импортирует pyplot и RandomWalk. Затем она создает случайное блуждание и сохраняет его в rw ❶, не забывая вызвать fill_walk(). В точке ❷ программа передает scatter() координаты x и y блуждания и выбирает подходящий размер точки. На рис. 15.9 показана диаграмма с 5000 точками. (В изображениях этого раздела область просмотра Matplotlib не показана, но вы увидите ее при запуске rw_visual.py.)

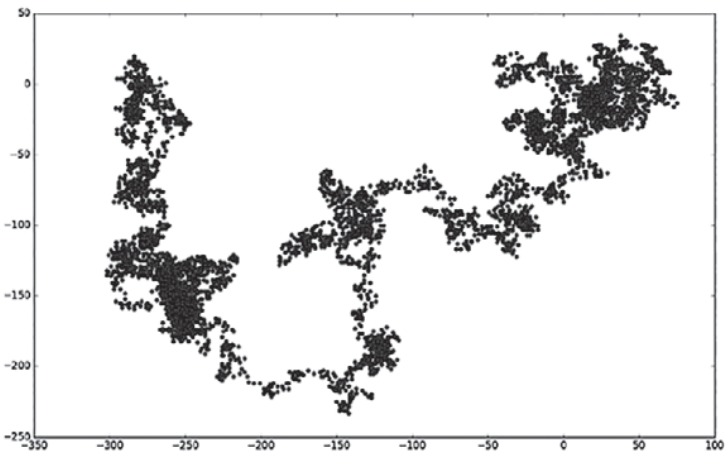

Рис. 15.9. Случайное блуждание с 5000 точек

Генерирование нескольких случайных блужданий

Все случайные блуждания отличаются друг от друга; интересно понаблюдать за тем, какие узоры генерирует программа. Один из способов использования предыдущего кода — построить несколько блужданий без многократного запуска программы в цикле `while`:

rw_visual.py

```
import matplotlib.pyplot as plt

from random_walk import RandomWalk

# Новые блуждания строятся до тех пор, пока программа остается активной.
while True:
    # Построение случайного блуждания.
    rw = RandomWalk()
    rw.fill_walk()

    # Нанесение точк на диаграмму.
    plt.style.use('classic')
    fig, ax = plt.subplots()
    ax.scatter(rw.x_values, rw.y_values, s=15)
    plt.show()

❶   keep_running = input("Make another walk? (y/n): ")
    if keep_running == 'n':
        break
```

Код генерирует случайное блуждание, отображает его в области просмотра Matplotlib и делает паузу с открытой областью просмотра. Когда вы закрываете область просмотра, программа спрашивает, хотите ли вы сгенерировать следующее блуждание. Введите значение `y`, и вы сможете сгенерировать блуждания, которые начинаются рядом с начальной точкой, а затем отклоняются преимущественно в одном направлении; при этом большие группы будут соединяться тонкими секциями. Чтобы завершить программу, введите `n`.

Оформление случайного блуждания

В этом разделе мы настроим диаграмму так, чтобы подчеркнуть важные характеристики каждого блуждания и отвести на второй план несущественные элементы. Для этого мы выделим характеристики, которые нужно подчеркнуть (например, откуда началось блуждание, где оно закончилось и по какому пути следовало). Затем определяются характеристики, которые нужно ослабить (например, деления шкалы и метки). Результатом должно быть простое визуальное представление, которое четко описывает путь, использованный в каждом случайном блуждании.

Назначение цветов

Мы используем цветовую карту для отображения точек блуждания, а также удаляем черный контур из каждой точки, чтобы цвет точек был лучше виден. Чтобы

точки окрашивались в соответствии с их позицией в блуждании, мы передаем в аргументе `c` список с позицией каждой точки. Так как точки выводятся по порядку, список просто содержит числа от 1 до 4999:

rw_visual.py

```
...
while True:
    # Построение случайного блуждания
    rw = RandomWalk()
    rw.fill_walk()

    # Нанесение точек на диаграмму.
    plt.style.use('classic')
    fig, ax = plt.subplots()
❶   point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolors='none', s=15)
    plt.show()

    keep_running = input("Make another walk? (y/n): ")
    ...
```

В точке ❶ функция `range()` используется для генерирования списка чисел, размер которого равен количеству точек в блуждании. Полученный результат сохраняется в списке `point_numbers`, который используется для назначения цвета каждой точке в блуждании. Мы передаем `point_numbers` в аргументе `c`, используем цветовую карту `Blues` и затем передаем `edgecolor=none` для удаления черного контура вокруг каждой точки. В результате создается диаграмма блуждания с градиентным переходом от светло-синего к темно-синему (рис. 15.10).

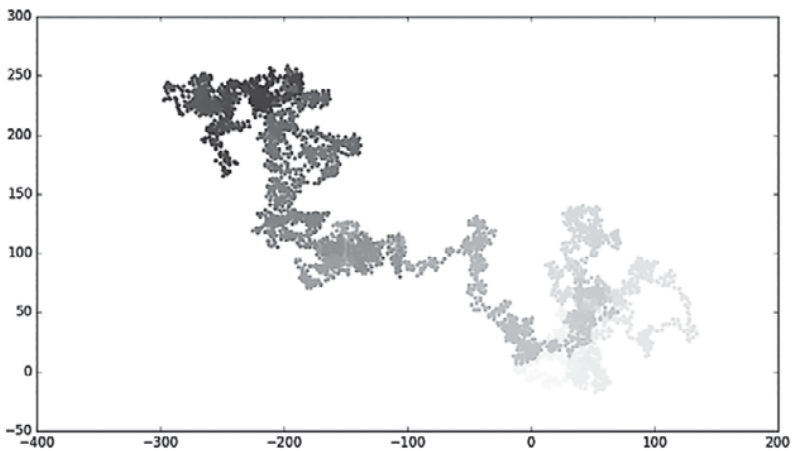

Рис. 15.10. Случайное блуждание, окрашенное с применением цветовой карты Blues

Начальные и конечные точки

Помимо раскраски точек, обозначающей их позицию, было бы неплохо видеть, где начинается и заканчивается каждое блуждание. Для этого можно прорисовать первую и последнюю точки отдельно, после нанесения на диаграмму основной серии. Мы выведем конечные точки с бóльшим размером и другим цветом, чтобы они выделялись на общем фоне:

rw_visual.py

```
...
while True:
    ...
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolors='none', s=15)

    # Выделение первой и последней точек.
    ax.scatter(0, 0, c='green', edgecolors='none', s=100)
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
        s=100)

    plt.show()
    ...
```

Чтобы вывести начальную точку, мы рисуем точку (0, 0) зеленым цветом с большим размером (`s`=100) по сравнению с остальными точками. Для выделения конечной точки последняя пара координат `x` и `y` выводится с размером 100. Обязательно вставьте этот код непосредственно перед вызовом `plt.show()`, чтобы начальная и конечная точки выводились поверх всех остальных точек.

При выполнении этого кода вы будете точно видеть, где начинается и кончается каждое блуждание. (Если конечные точки не выделяются достаточно четко, настраивайте их цвет и размер, пока не достигнете желаемого результата.)

Удаление осей

Уберем оси с диаграммы, чтобы они не отвлекали зрителя от общей картины. Для удаления осей используется следующий код:

rw_visual.py

```
  ...
  while True:
      ...
      ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
          s=100)

      # Удаление осей.
❶     ax.get_xaxis().set_visible(False)
      ax.get_yaxis().set_visible(False)

      plt.show()
      ...
```

Методы `ax.get_xaxis()` и `ax.get_yaxis()` ❶ переводят флаг видимости каждой оси в состояние `False`. При работе с визуализацией подобные цепочки вызовов встречаются очень часто.

Запустите программу rw_visual.py; теперь выводимые диаграммы не имеют осей.

Добавление точек

Увеличим количество точек, чтобы работать с большим объемом данных. Для этого мы увеличим значение `num_points` при создании экземпляра `RandomWalk` и отрегулируем размер каждой точки при выводе диаграммы:

rw_visual.py

```
...
while True:
    # Построение случайного блуждания.
    rw = RandomWalk(50000)
    rw.fill_walk()

    # Вывод точек и отображение диаграммы.
    plt.style.use('classic')
    fig, ax = plt.subplots()
    point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolor='none', s=1)
    ...
```

В этом примере создается случайное блуждание из 50 000 точек (что в большей степени соответствует реальным данным), и каждая точка рисуется размером `s=1`. Как видно из рис. 15.11, изображение получается эфемерным и туманным. Простая точечная диаграмма превратилась в произведение искусства!

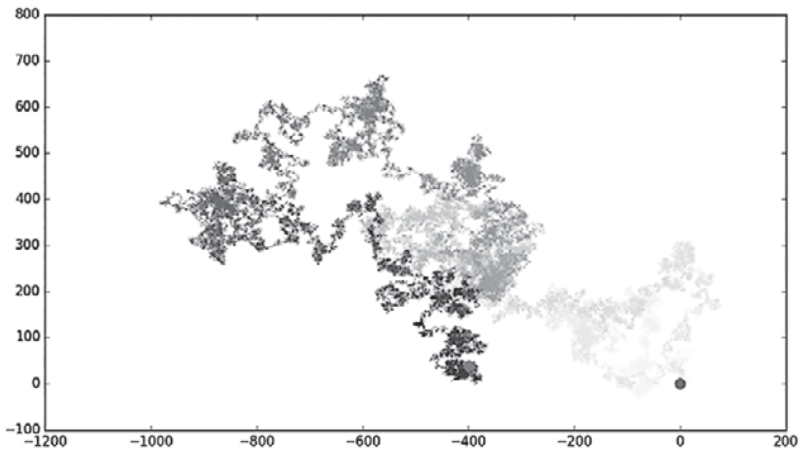

Рис. 15.11. Случайное блуждание с 50 000 точек

Поэкспериментируйте с этим кодом и посмотрите, насколько вам удастся увеличить количество точек в случайном блуждании, прежде чем система начнет заметно тормозить или диаграмма потеряет свою визуальную привлекательность.

Изменение размера диаграммы для заполнения экрана

Визуализация гораздо эффективнее передает закономерности в данных, если она адаптирована под размер экрана. Чтобы диаграмма лучше смотрелась на экране, измените размер области просмотра Matplotlib:

rw_visual.py

```
...
while True:
    # Построение случайного блуждания.
    rw = RandomWalk()
    rw.fill_walk()

    # Назначение размера области просмотра.
    plt.style.use('classic')
    fig, ax = plt.subplots(figsize=(15, 9))
    ...
```

Функция `figure()` управляет шириной, высотой, разрешением и цветом фона диаграммы. Параметр `figsize` получает кортеж с размерами окна диаграммы в дюймах.

Matplotlib предполагает, что разрешение экрана составляет 100 пикселов на дюйм; если этот код не дает точного размера, внесите необходимые изменения в числа. Или, если вы знаете разрешение экрана в вашей системе, передайте его `plt.subplots()` в параметре `dpi` для выбора размера, эффективно использующего доступное пространство:

```
fig, ax = plt.subplots(figsize=(10, 6), dpi=128)
```

УПРАЖНЕНИЯ

15.3. Молекулярное движение: измените программу rw_visual.py и замените `plt.scatter()` вызовом `plt.plot()`. Чтобы смоделировать путь пыльцевого зерна на поверхности водяной капли, передайте значения `rw.x_values` и `rw.y_values` и включите аргумент `linewidth`. Используйте 5000 точек вместо 50 000.

15.4. Измененные случайные блуждания: в классе `RandomWalk` значения `x_step` и `y_step` генерируются по единому набору условий. Направление выбирается случайно из списка `[1, -1]`, а расстояние — из списка `[0, 1, 2, 3, 4]`. Измените значения в этих списках и посмотрите, что произойдет с общей формой диаграммы. Попробуйте применить расширенный список вариантов расстояния (например, от 0 до 8) или удалите `-1` из списка направлений по оси x или y.

15.5. Рефакторинг: метод `fill_walk()` получился слишком длинным. Создайте новый метод с именем `get_step()`, который определяет расстояние и направление для каждого шага, после чего вычисляет этот шаг. В результате метод `fill_walk()` должен содержать два вызова `get_step()`:

```
    x_step = self.get_step()
    y_step = self.get_step()
```

Рефакторинг сокращает размер `fill_walk()`, а метод становится более простым и понятным.

Моделирование бросков кубиков в Plotly

В этом разделе мы воспользуемся пакетом визуализации Plotly для построения интерактивных визуализаций. Пакет Plotly особенно хорошо подходит для визуализаций, которые будут отображаться в браузере, потому что изображение автоматически масштабируется по размерам экрана зрителя. Кроме того, Plotly генерирует интерактивные визуализации; когда пользователь наводит указатель мыши на некоторые элементы, на экране появляется расширенная информация об этом элементе.

В этом проекте мы займемся анализом результатов бросков кубиков. При броске одного шестигранного кубика существует равная вероятность выпадения любого числа от 1 до 6. С другой стороны, при броске двух кубиков некоторые суммы выпадают с большей вероятностью, чем другие. Чтобы определить, какие числа наиболее вероятны, мы сгенерируем набор данных, представляющих брошенные кубики. Затем на базе данных большого количества бросков будет построена диаграмма, по которой можно определить, какие результаты более вероятны.

Броски кубиков часто используются в математике для пояснения различных типов анализа данных. Впрочем, они также находят применение в реальных приложениях — скажем, в казино и в обычных играх («Монополия», ролевые игры и т. д.).

Установка Plotly

Установите Plotly при помощи pip по аналогии с тем, как это делалось с Matplotlib:

```
$ python -m pip install --user plotly
```

(Если при установке Matplotlib использовалась команда `python3` или другая, убедитесь в том, что в данном случае используется то же самое.)

Примеры визуализаций, которые могут быть построены с использованием Plotly, представлены в галерее диаграмм: зайдите на сайт https://plot.ly/python/. Каждый пример сопровождается исходным кодом, так что вы сможете увидеть, как была построена каждая из визуализаций.

Создание класса кубика

Для моделирования броска одного кубика будет использоваться класс `Die`:

die.py

```
from random import randint
```

```
  class Die():
      """Класс, представляющий один кубик."""

❶     def __init__(self, num_sides=6):
          """По умолчанию используется шестигранный кубик."""
          self.num_sides = num_sides

      def roll(self):
          """Возвращает случайное число от 1 до числа граней."""
❷         return randint(1, self.num_sides)
```

Метод `__init__()` получает один необязательный аргумент. Если при создании экземпляра кубика аргумент с количеством сторон не передается, по умолчанию создается шестигранный кубик. Если же аргумент *присутствует*, то переданное значение используется для определения количества граней ❶. (Кубики принято обозначать по количеству граней: шестигранный кубик — D6, восьмигранный — D8 и т. д.)

Метод `roll()` использует функцию `randint()` для получения случайного числа в диапазоне от 1 до количества граней ❷. Функция может вернуть начальное значение (1), конечное значение (`num_sides`) или любое целое число в этом диапазоне.

Бросок кубика

Прежде чем строить визуализацию на основе этого класса, бросим кубик D6, выведем результаты и убедимся в том, что они выглядят разумно:

die_visual.py

```
  from die import Die

  # Создание кубика D6.
❶ die = Die()

  # Моделирование серии бросков с сохранением результатов в списке.
  results = []
❷ for roll_num in range(100):
      result = die.roll()
      results.append(result)

  print(results)
```

В точке ❶ создается экземпляр `Die` с шестью гранями по умолчанию. В точке ❷ моделируются 100 бросков кубика, а результат каждого броска сохраняется в списке `results`. Выборка выглядит примерно так:

```
[4, 6, 5, 6, 1, 5, 6, 3, 5, 3, 5, 3, 2, 2, 1, 3, 1, 5, 3, 6, 3, 6, 5, 4,
 1, 1, 4, 2, 3, 6, 4, 2, 6, 4, 1, 3, 2, 5, 6, 3, 6, 2, 1, 1, 3, 4, 1, 4,
 3, 5, 1, 4, 5, 5, 2, 3, 3, 1, 2, 3, 5, 6, 2, 5, 6, 1, 3, 2, 1, 1, 1, 6,
 5, 5, 2, 2, 6, 4, 1, 4, 5, 1, 1, 1, 4, 5, 3, 3, 1, 3, 5, 4, 5, 6, 5, 4,
 1, 5, 1, 2]
```

Беглое знакомство с результатами показывает, что класс `Die` работает. В результатах встречаются граничные значения 1 и 6, то есть модель возвращает наименьшее и наибольшее возможное значение; значения 0 и 7 не встречаются, а значит, все результаты лежат в диапазоне допустимых значений. Также в выборке встречаются все числа от 1 до 6, то есть представлены все возможные результаты.

Анализ результатов

Чтобы проанализировать результаты бросков одного кубика D6, мы подсчитаем, сколько раз выпадало каждое число:

die_visual.py

```
--snip--
# Моделирование серии бросков с сохранением результатов в списке.
results = []
❶ for roll_num in range(1000):
    result = die.roll()
    results.append(result)

# Анализ результатов.
frequencies = []
❷ for value in range(1, die.num_sides+1):
❸     frequency = results.count(value)
❹     frequencies.append(frequency)

print(frequencies)
```

Так как Plotly используется для анализа, а не для вывода результатов, количество моделируемых бросков можно увеличить до 1000 ❶. Для анализа создается пустой список `frequencies`, в котором хранится количество выпадений каждого значения. Программа перебирает возможные значения (от 1 до 6 в данном случае) в цикле ❷, подсчитывает количество вхождений каждого числа в результатах ❸, после чего присоединяет полученное значение к списку `frequencies` ❹. Содержимое списка выводится перед построением визуализации:

```
[155, 167, 168, 170, 159, 181]
```

Результаты выглядят разумно: мы видим все шесть частот, по одной для каждого возможного результата при броске D6, и ни одна из частот не выделяется на общем фоне. А теперь займемся наглядным представлением результатов.

Построение гистограммы

Имея список частот, можно построить *гистограмму* результатов. Гистограмма представляет собой столбцовую диаграмму, наглядно отображающую относительные частоты результатов. Код построения гистограммы выглядит так:

die_visual.py

```
  from plotly.graph_objs import Bar, Layout
  from plotly import offline

  from die import Die
  ...

  # Анализ результатов.
  frequencies = []
  for value in range(1, die.num_sides+1):
      frequency = results.count(value)
      frequencies.append(frequency)

  # Визуализация результатов.
❶ x_values = list(range(1, die.num_sides+1))
❷ data = [Bar(x=x_values, y=frequencies)]

❸ x_axis_config = {'title': 'Result'}
  y_axis_config = {'title': 'Frequency of Result'}
❹ my_layout = Layout(title='Results of rolling one D6 1000 times',
          xaxis=x_axis_config, yaxis=y_axis_config)
❺ offline.plot({'data': data, 'layout': my_layout}, filename='d6.html')
```

Чтобы построить столбцовую диаграмму, необходимо создать столбец для каждого из возможных результатов. Эти результаты сохраняются в списке `x_values`, который начинается с 1 и заканчивается количеством граней кубика ❶. Plotly не может получить результат функции `range()` напрямую, поэтому необходимо явно преобразовать диапазон в список при помощи функции `list`. Класс `Bar` из Plotly представляет набор данных, который будет форматироваться в виде столбцовой диаграммы ❷. Класс должен быть заключен в квадратные скобки, поскольку набор данных может состоять из нескольких элементов.

Для осей предусмотрены различные возможности настройки, и каждый параметр конфигурации сохраняется в виде элемента в словаре. На данный момент мы только задаем заголовок каждой оси ❸. Класс `Layout()` возвращает объект, который задает макет и конфигурацию диаграммы в целом ❹. Здесь также задается заголовок диаграммы и передаются словари конфигурации осей `x` и `y`.

Диаграмма строится вызовом функции `offline.plot()` ❺. Этой функции передается словарь с объектами данных и макета; она также получает имя файла для сохранения результата. В нашем примере результат сохраняется в файле с именем d6.html.

При запуске программы die_visual.py должен открыться браузер с файлом d6.html. Если это не происходит автоматически, откройте новую вкладку в любом браузере, а затем откройте файл d6.html (из папки, в которой был сохранен файл die_visual.py). Диаграмма должна выглядеть примерно так, как на рис. 15.12. (Изображение было слегка изменено для печати; по умолчанию Plotly генерирует диаграммы с более темным фоном, чем на иллюстрации.)

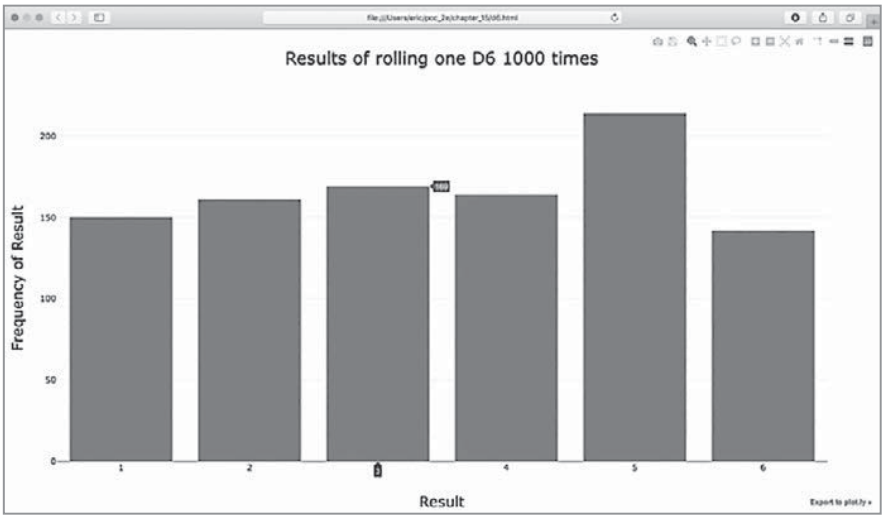

Рис. 15.12. Простая гистограмма, созданная с использованием Plotly

Обратите внимание на интерактивность диаграмм, построенных с использованием Plotly: если навести указатель мыши на столбец диаграммы, вы увидите данные, связанные с этим столбцом. Данная возможность особенно полезна при нанесении нескольких наборов данных на одну диаграмму. Также обратите внимание на кнопки в правом верхнем углу: они позволяют изменить масштаб визуализации и сохранить ее в графическом файле.

Бросок двух кубиков

При броске двух кубиков вы получаете большие значения с другим распределением результатов. Изменим наш код и создадим два кубика D6, моделирующих бросок пары кубиков. При броске каждой пары программа складывает два числа (по одному с каждого кубика) и сохраняет сумму в `results`. Сохраните копию die_visual.py под именем dice_visual.py и внесите следующие изменения:

dice_visual.py

```
from plotly.graph_objs import Bar, Layout
from plotly import offline

from die import Die

# Создание двух кубиков D6.
die_1 = Die()
die_2 = Die()

# Моделирование серии бросков с сохранением результатов в списке.
results = []
for roll_num in range(1000):
```

```
❶     result = die_1.roll() + die_2.roll()
      results.append(result)

  # Анализ результатов.
  frequencies = []
❷ max_result = die_1.num_sides + die_2.num_sides
❸ for value in range(2, max_result+1):
      frequency = results.count(value)
      frequencies.append(frequency)

  # Визуализация результатов.
  x_values = list(range(2, max_result+1))
  data = [Bar(x=x_values, y=frequencies)]

❹ x_axis_config = {'title': 'Result', 'dtick': 1}
  y_axis_config = {'title': 'Frequency of Result'}
  my_layout = Layout(title='Results of rolling two D6 dice 1000 times',
          xaxis=x_axis_config, yaxis=y_axis_config)
  offline.plot({'data': data, 'layout': my_layout}, filename='d6_d6.html')
```

Создав два экземпляра `Die`, мы бросаем кубики и вычисляем сумму для каждого броска ❶. Самый большой возможный результат (12) вычисляется суммированием наибольших результатов на обоих кубиках; мы сохраняем его в `max_result` ❷. Наименьший возможный результат (2) равен сумме наименьших результатов на обоих кубиках. В процессе анализа подсчитывается количество результатов для каждого значения от 2 до `max_result` ❸. (Также можно было использовать диапазон `range(2, 13)`, но он работал бы только для двух кубиков D6. При моделировании реальных ситуаций лучше писать код, который легко адаптируется для разных ситуаций. В частности, этот код позволяет смоделировать бросок пары кубиков с *любым* количеством граней.)

При создании диаграммы мы включаем ключ `dtick` в словарь `x_axis_config` ❹. Этот параметр управляет расстоянием между делениями на оси x. С появлением дополнительных столбцов на гистограмме в конфигурации по умолчанию Plotly будет снабжать метками только часть столбцов. Параметр `'dtick': 1` приказывает Plotly помечать все деления. Также мы обновляем заголовок диаграммы и изменяем имя выходного файла.

После выполнения кода в браузере должна появиться диаграмма, примерный вид которой показан на рис. 15.13.

На диаграмме показаны примерные результаты, которые могут быть получены для пары кубиков D6. Как видите, реже всего выпадают результаты 2 и 12, а чаще всего 7, потому что эта комбинация может быть выброшена шестью способами, а именно: 1+6, 2+5, 3+4, 4+3, 5+2 и 6+1.

Броски кубиков с разным числом граней

Создадим кубики с 6 и 10 гранями и посмотрим, что произойдет, если бросить их 50 000 раз:

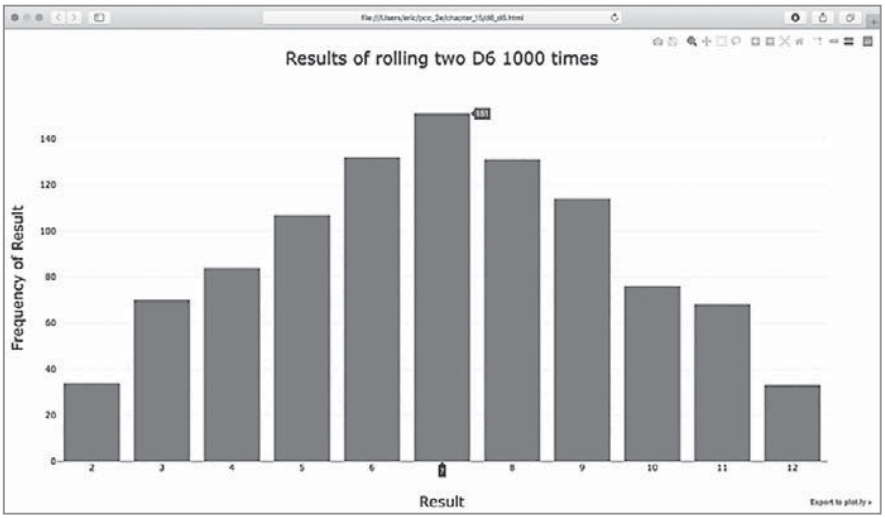

Рис. 15.13. Смоделированные результаты 1000 бросков двух шестигранных кубиков

dice_visual.py

```
  from plotly.graph_objs import Bar, Layout
  from plotly import offline
  from die import Die

  # Создание кубиков D6 и D10.
  die_1 = Die()
❶ die_2 = Die(10)

  # Моделирование серии бросков с сохранением результатов в списке.
  results = []
  for roll_num in range(50000):
      result = die_1.roll() + die_2.roll()
      results.append(result)

  # Анализ результатов.
  ...

  # Визуализация результатов.
  X_values = list(range(2, max_result+1))
  data = [Bar(x=x_values, y=frequencies)]

  x_axis_config = {'title': 'Result', 'dtick': 1}
  y_axis_config = {'title': 'Frequency of Result'}
❷ my_layout = Layout(title='Results of rolling a D6 and a D10 50000 times',
          xaxis=x_axis_config, yaxis=y_axis_config)
  offline.plot({'data': data, 'layout': my_layout}, filename='d6_d10.html')
```

Чтобы создать модель кубика D10, мы передаем аргумент 10 при создании второго экземпляра `Die` ❶ и изменяем первый цикл для моделирования 50 000 бросков

вместо 1000. Наименьший возможный результат, как и прежде, равен 2, зато наибольший увеличился до 16; мы соответственно изменяем заголовок, метки оси x и метки серии данных ❷.

На рис. 15.14 показана полученная диаграмма. Вместо одного наиболее вероятного результата их стало целых пять. Это объясняется тем, что наименьшее (1+1) и наибольшее (6+10) значения по-прежнему могут быть получены только одним способом, но кубик D6 ограничивает количество способов генерирования средних чисел: суммы 7, 8, 9, 10 и 11 можно выбросить шестью способами. Следовательно, именно эти результаты являются наиболее частыми и все эти числа выпадают с равной вероятностью.

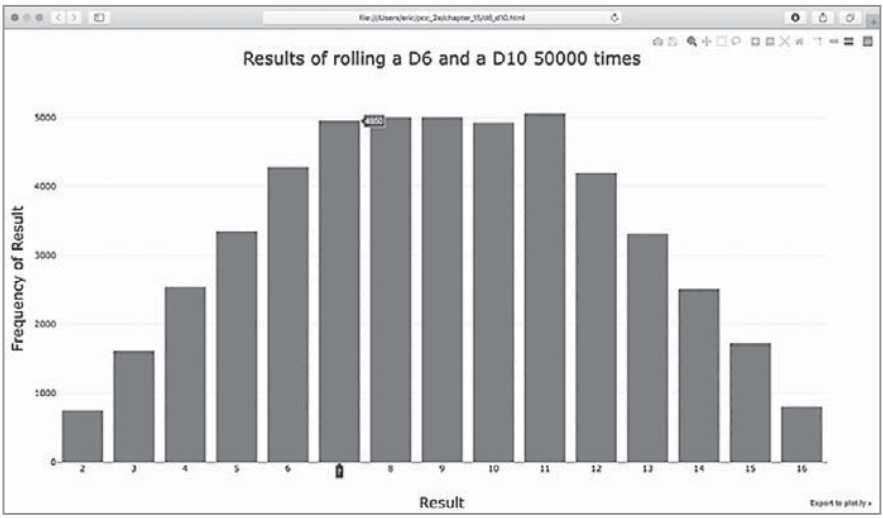

Рис. 15.14. Результаты 50 000 бросков шести- и десятигранного кубиков

Возможность применения Plotly для моделирования бросков кубиков дает существенную свободу при исследовании этого явления. За считаные минуты вы сможете смоделировать огромное количество бросков с разнообразными кубиками.

УПРАЖНЕНИЯ

15.6. Два кубика D8: создайте модель, которая показывает, что происходит при 1000-кратном бросании двух восьмигранных кубиков. Попробуйте заранее представить, как будет выглядеть визуализация, перед моделированием; проверьте правильность своих интуитивных представлений. Постепенно наращивайте количество бросков, пока не начнете замечать ограничения, связанные с ресурсами вашей системы.

15.7. Три кубика: при броске трех кубиков D6 наименьший возможный результат равен 3, а наибольший — 18. Создайте визуализацию, которая показывает, что происходит при броске трех кубиков D6.

15.8. Умножение: при броске двух кубиков результат обычно определяется суммированием двух чисел. Создайте визуализацию, которая показывает, что происходит при умножении этих чисел.

15.9. Генераторы: для наглядности в списках этого раздела используется длинная форма цикла `for`. Если вы уверенно работаете с генераторами списков, попробуйте написать генератор для одного или обоих циклов в каждой из этих программ.

15.10. Эксперименты с библиотеками: попробуйте использовать Matplotlib для создания визуализации бросков кубиков, а Plotly — для создания визуализации случайного блуждания. (Для выполнения этого упражнения вам придется обратиться к документациям обеих библиотек.)

Итоги

В этой главе вы научились генерировать наборы данных и строить визуализации этих данных. Вы научились строить простые диаграммы с использованием Matplotlib и применять точечные диаграммы для анализа случайных блужданий. Вы узнали, как построить гистограмму с использованием Plotly и как исследовать результаты бросков кубиков с разным количеством граней при помощи гистограммы.

Генерирование собственных наборов данных в программах — интересный и мощный способ моделирования и анализа различных реальных ситуаций. В дальнейших проектах визуализации данных обращайте особое внимание на ситуации, которые могут быть смоделированы на программном уровне. Присмотритесь к визуализациям, встречающимся в выпусках новостей, — возможно, они были сгенерированы методами, сходными с теми, о которых вы узнали в этих проектах?

В главе 16 мы загрузим данные из сетевого источника и продолжим использовать Matplotlib и Plotly для анализа данных.

16 Загрузка данных

В этой главе мы загрузим наборы данных из сетевого источника и создадим работоспособные визуализации этих данных. В интернете можно найти невероятно разнообразную информацию, большая часть которой еще не подвергалась основательному анализу. Умение анализировать данные позволит вам выявить связи и закономерности, не найденные никем другим.

В этой главе рассматривается работа с данными в двух популярных форматах, CSV и JSON. Модуль Python `csv` будет применен для обработки погодных данных в формате CSV (с разделением запятыми) и анализа динамики высоких и низких температур в двух разных местах. Затем библиотека Matplotlib будет использована для построения на базе загруженных данных диаграммы колебания температур в двух разных местах: в Ситке (Аляска) и в Долине Смерти (Калифорния). Позднее в этой главе модуль `json` будет использован для обращения к данным численности населения, хранимым в формате JSON, а при помощи модуля Plotly будет построена карта с данными местоположения и магнитуд недавних землетрясений.

К концу этой главы вы будете готовы к работе с разными типами и форматами наборов данных и начнете лучше понимать принципы построения сложных визуализаций. Возможность загрузки и визуализации сетевых данных разных типов и форматов крайне важна для работы с разнообразными массивами данных в реальном мире.

Формат CSV

Один из простейших вариантов хранения — запись данных в текстовый файл как серий значений, разделенных запятыми; такой формат хранения получил название *CSV* (от Comma Separated Values, то есть «значения, разделенные запятыми»). Например, одна строка погодных данных в формате CSV может выглядеть так:

```
"USW00025333","SITKA AIRPORT, AK US","2018-01-01","0.45",,"48","38"
```

Это погодные данные за 1 января 2018 г. в Ситке (Аляска). В данных указаны максимальная и минимальная температура, а также ряд других показателей за этот день. У человека могут возникнуть проблемы с чтением данных CSV, но этот

формат хорошо подходит для программной обработки и извлечения значений, а это ускоряет процесс анализа.

Начнем с небольшого набора погодных данных в формате CSV, записанного в Ситке; файл с данными можно загрузить по адресу https://www.nostarch.com/pythoncrashcourse2e/. Создайте подкаталог с именем data в каталоге, в котором сохраняются программы этой главы. Скопируйте в созданный каталог файл sitka_weather_07-2018_simple.csv. (После загрузки ресурсов книги в вашем распоряжении появятся все необходимые файлы для этого проекта.)

ПРИМЕЧАНИЕ Погодные данные для этого проекта были загружены с сайта https://ncdc.noaa.gov/cdo-web/.

Разбор заголовка файлов CSV

Модуль Python `csv` из стандартной библиотеки разбирает строки файла CSV и позволяет быстро извлечь нужные значения. Начнем с первой строки файла, которая содержит серию заголовков данных. Заголовки описывают информацию, хранящуюся в данных:

sitka_highs.py

```
  import csv

  filename = 'data/sitka_weather_07-2018_simple.csv'
❶ with open(filename) as f:
❷     reader = csv.reader(f)
❸     header_row = next(reader)
      print(header_row)
```

После импортирования модуля `csv` имя обрабатываемого файла сохраняется в переменной `filename`. Затем файл открывается, а полученный объект сохраняется в переменной `f` ❶. Далее программа вызывает метод `csv.reader()` и передает ему объект файла в аргументе, чтобы создать объект чтения данных для этого файла ❷. Объект чтения данных сохраняется в переменной `reader`.

Модуль `csv` содержит функцию `next()`, которая возвращает следующую строку файла для полученного объекта чтения данных. В следующем листинге функция `next()` вызывается только один раз для получения первой строки файла, содержащей заголовки ❸. Возвращенные данные сохраняются в `header_row`. Как видите, `header_row` содержит осмысленные имена заголовков, которые сообщают, какая информация содержится в каждой строке данных:

```
['STATION', 'NAME', 'DATE', 'PRCP', 'TAVG', 'TMAX', 'TMIN']
```

Объект `reader` обрабатывает первую строку значений, разделенных запятыми, и сохраняет все значения из строки в списке. Заголовок `STATION` представляет код метеорологической станции, зарегистрировавшей данные. Позиция заголовка указывает на то, что первым значением в каждой из следующих строк является код метеостанции.

Остальные заголовки сообщают, какая информация хранится в соответствующем поле. В данном примере нас интересуют значения даты, высокой температуры (`TMAX`) и низкой температуры (`TMIN`). Мы используем простой набор данных, содержащий информацию только об уровне осадков и температуре. Вы также можете загрузить собственный набор погодных данных и включить в обработку другие показатели: скорость и направление ветра, расширенные данные осадков и т. д.

Печать заголовков и их позиций

Чтобы читателю было проще понять структуру данных в файле, выведем каждый заголовок и его позицию в списке:

sitka_highs.py

```
  ...
  with open(filename) as f:
      reader = csv.reader(f)
      header_row = next(reader)

❶     for index, column_header in enumerate(header_row):
          print(index, column_header)
```

Функция `enumerate()` возвращает индекс каждого элемента и его значение при переборе списка ❶. (Обратите внимание: строка `print(header_row)` удалена ради этой, более подробной версии.)

Результат с индексами всех заголовков выглядит так:

```
0 STATION
1 NAME
2 DATE
3 PRCP
4 TAVG
5 TMAX
6 TMIN
```

Из этих данных видно, что даты и максимальные температуры за эти дни находятся в столбцах 2 и 5. Чтобы проанализировать температурные данные, мы обработаем каждую запись данных в файле sitka_weather_07-2018_simple.csv и извлечем элементы с индексами 2 и 5.

Извлечение и чтение данных

Итак, нужные столбцы данных известны; попробуем прочитать часть этих данных. Начнем с чтения максимальной температуры за каждый день:

sitka_highs.py

```
  ...
  with open(filename) as f:
```

```
    reader = csv.reader(f)
    header_row = next(reader)

    # Чтение максимальных температур
❶   highs = []
❷   for row in reader:
❸       high = int(row[5])
        highs.append(high)

print(highs)
```

Программа создает пустой список с именем `highs` ❶ и перебирает остальные строки в файле ❷. Объект `reader` продолжает с того места, на котором он остановился в ходе чтения файла CSV, и автоматически возвращает каждую строку после текущей позиции. Так как заголовок уже прочитан, цикл продолжается со второй строки, в которой начинаются фактические данные. При каждом проходе цикла значение с индексом 5 (заголовок `TMAX`) присваивается переменной `high` ❸. Функция `int()` преобразует данные, хранящиеся в строковом виде, в числовой формат для последующего использования. Значение присоединяется к списку `highs`.

В результате будет получен список `highs` со следующим содержимым:

```
[62, 58, 70, 70, 67, 59, 58, 62, 66, 59, 56, 63, 65, 58, 56, 59, 64, 60, 60,
 61, 65, 65, 63, 59, 64, 65, 68, 66, 64, 67, 65]
```

Мы извлекли максимальную температуру для каждого дня и аккуратно сохранили полученные данные в строковом формате в списке. Следующим шагом станет построение визуализации этих данных.

Нанесение данных на диаграмму

Для наглядного представления температурных данных мы сначала создадим простую диаграмму дневных максимумов температуры с использованием Matplotlib:

sitka_highs.py

```
import csv

from matplotlib import pyplot as plt

filename = 'data/sitka_weather_07-2018_simple.csv'
with open(filename) as f:
    ...

# Нанесение данных на диаграмму.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.plot(highs, c='red')

# Форматирование диаграммы.
❷ plt.title("Daily high temperatures, July 2018", fontsize=24)
❸ plt.xlabel('', fontsize=16)
```

```
plt.ylabel("Temperature (F)", fontsize=16)
plt.tick_params(axis='both', which='major', labelsize=16)

plt.show()
```

Мы передаем при вызове `plot()` список `highs` ❶ и аргумент `c='red'` для отображения точек красным цветом. (Максимумы будут выводиться красным цветом, а минимумы синим.) Затем указываются другие аспекты форматирования (например, размер шрифта и метки) ❷, уже знакомые вам по главе 15. Так как даты еще не добавлены, метки для оси `x` не задаются, но вызов `plt.xlabel()` изменяет размер шрифта, чтобы метки по умолчанию лучше читались ❸. На рис. 16.1 показана полученная диаграмма: это простой график температурных максимумов за июль 2018 г. в Ситке (штат Аляска).

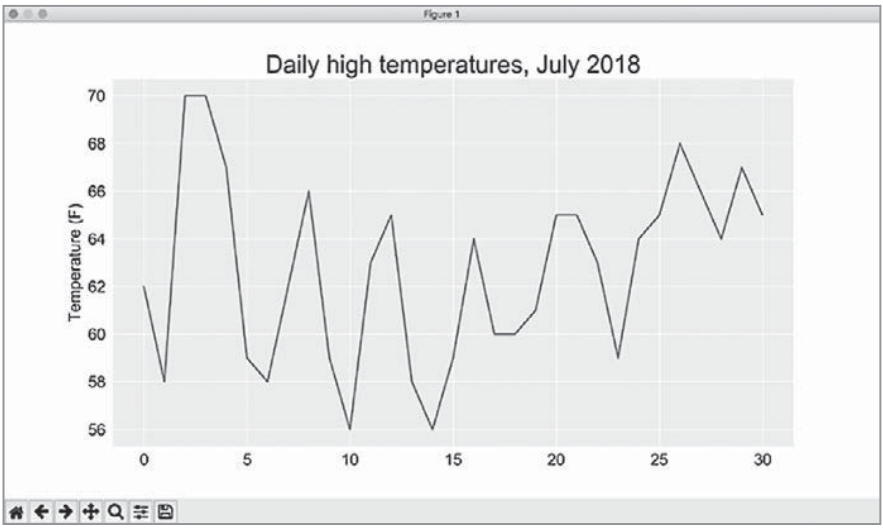

Рис. 16.1. График ежедневных температурных максимумов в июле 2018 г. в Ситке (штат Аляска)

Модуль datetime

Теперь нанесем даты на график, чтобы с ним было удобнее работать. Первая дата из файла погодных данных хранится во второй строке файла:

```
"USW00025333","SITKA AIRPORT, AK US","2018-07-01","0.25",,"62","50"
```

Данные будут читаться в строковом формате, поэтому нам понадобится способ преобразовать строку `'2018-07-1'` в объект, представляющий эту дату. Чтобы построить объект, соответствующий 1 июля 2018 г., мы воспользуемся методом `strptime()` из модуля `datetime`. Посмотрим, как работает `strptime()` в терминальном окне:

```
>>> from datetime import datetime
>>> first_date = datetime.strptime('2018-07-01', '%Y-%m-%d')
>>> print(first_date)
2018-07-01 00:00:00
```

Сначала необходимо импортировать класс `datetime` из модуля `datetime`. Затем вызывается метод `strptime()`, первый аргумент которого содержит строку с датой. Второй аргумент сообщает Python, как отформатирована дата. В данном примере значение `'%Y-'` сообщает Python, что часть строки, предшествующая первому дефису, должна интерпретироваться как год из четырех цифр; `'%m-'` приказывает Python интерпретировать часть строки перед вторым дефисом как число, представляющее месяц; наконец, `'%d'` приказывает Python интерпретировать последнюю часть строки как день месяца от 1 до 31.

Метод `strptime()` может получать различные аргументы, которые описывают, как должна интерпретироваться запись даты. В табл. 16.1 перечислены некоторые из таких аргументов.

Таблица 16.1. Аргументы форматирования даты и времени из модуля datetime

Аргумент	Описание
`%A`	Название дня недели (например, Monday)
`%B`	Название месяца (например, January)
`%m`	Порядковый номер месяца (от 01 до 12)
`%d`	День месяца (от 01 до 31)
`%Y`	Год из четырех цифр (например, 2019)
`%y`	Две последние цифры года (например, 19)
`%H`	Часы в 24-часовом формате (от 00 до 23)
`%I`	Часы в 12-часовом формате (от 01 до 12)
`%p`	AM или PM
`%M`	Минуты (от 00 до 59)
`%S`	Секунды (от 00 до 61)

Представление дат на диаграмме

Научившись обрабатывать данные в файлах CSV, вы сможете улучшить диаграмму температурных данных. Для этого мы извлечем из файла даты ежедневных максимумов и передадим даты и максимумы функции `plot()`:

sitka_highs.py

```
import csv
from datetime import datetime

from matplotlib import pyplot as plt

filename = 'data/sitka_weather_07-2018_simple.csv'
with open(filename) as f:
```

```
    reader = csv.reader(f)
    header_row = next(reader)

    # Чтение дат и максимальных температур из файла.
❶   dates, highs = [], []
    for row in reader:
❷       current_date = datetime.strptime(row[2], "%Y-%m-%d")
        high = int(row[5])
        dates.append(current_date)
        highs.append(high)

# Нанесение данных на диаграмму.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❸ ax.plot(dates, highs, c='red')

# Форматирование диаграммы.
plt.title("Daily high temperatures, July 2018", fontsize=24)
plt.xlabel('', fontsize=16)
❹ fig.autofmt_xdate()
plt.ylabel("Temperature (F)", fontsize=16)
plt.tick_params(axis='both', which='major', labelsize=16)

plt.show()
```

Мы создаем два пустых списка для хранения дат и температурных максимумов из файла ❶. Затем программа преобразует данные, содержащие информацию даты (`row[2]`), в объект `datetime` ❷, который присоединяется к `dates`. Значения дат и температурных максимумов передаются `plot()` в точке ❸. Вызов `fig.autofmt_xdate()` в точке ❹ выводит метки дат по диагонали, чтобы они не перекрывались. На рис. 16.2 изображена новая версия графика.

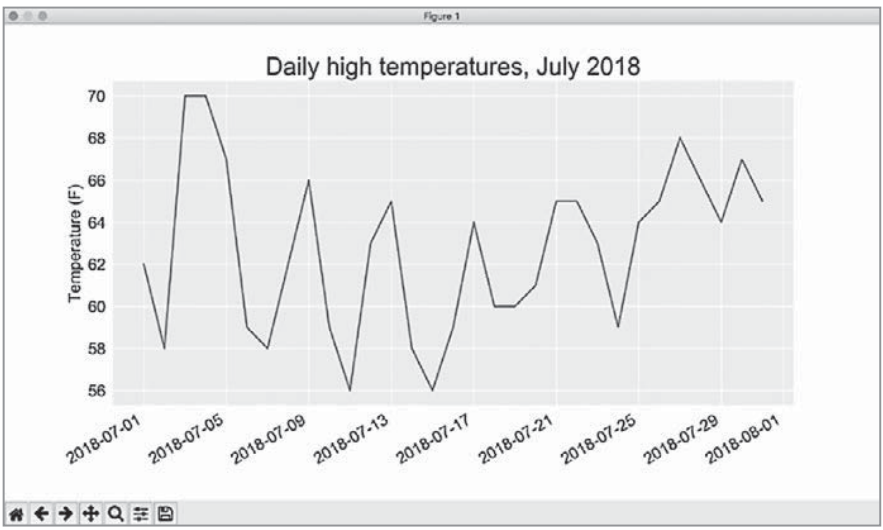

Рис. 16.2. График с датами на оси x стал более понятным

Расширение временного диапазона

Итак, график успешно создан. Добавим на него новые данные для получения более полной картины погоды в Ситке. Скопируйте файл sitka_weather_2018_simple.csv, содержащий погодные данные для Ситки за целый год, в каталог с программами этой главы.

А теперь мы можем сгенерировать график с погодными данными за год:

sitka_highs.py

```
  ...
❶ filename = 'data/sitka_weather_2018_simple.csv'
  with open(filename) as f:
  ...
  # Форматирование диаграммы.
❷ plt.title("Daily high temperatures - 2018", fontsize=24)
  plt.xlabel('', fontsize=16)
  ...
```

Значение `filename` было изменено, чтобы в программе использовался новый файл данных sitka_weather_2018_simple.csv ❶, а заголовок диаграммы приведен в соответствие с содержимым ❷. На рис. 16.3 изображена полученная диаграмма.

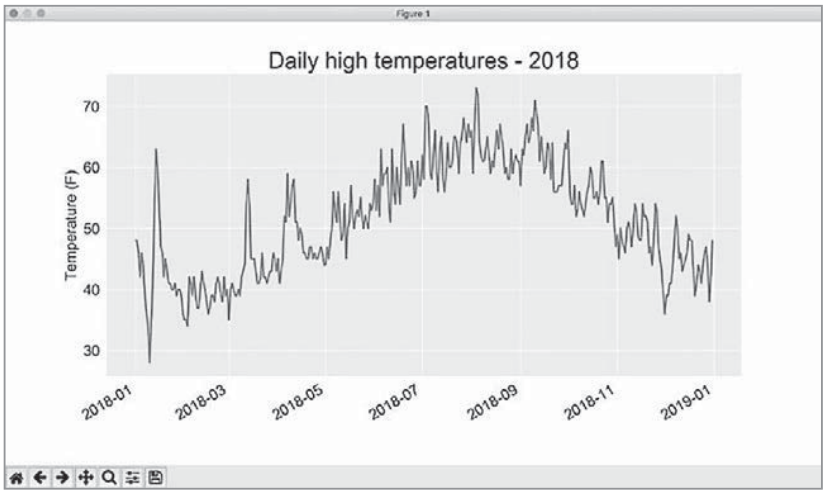

Рис. 16.3. Данные за год

Нанесение на диаграмму второй серии данных

Обновленный график на рис. 16.3 содержит немало полезной информации, но график можно сделать еще полезнее, добавив на него данные температурных минимумов. Для этого необходимо прочитать температурные минимумы из файла данных и нанести их на график:

sitka_highs_lows.py

```
  ...
  filename = 'sitka_weather_2018_simple.csv'
  with open(filename) as f:
      reader = csv.reader(f)
      header_row = next(reader)

      # Получение дат, температурных минимумов и максимумов из файла.
❶     dates, highs, lows = [], [], []
      for row in reader:
          current_date = datetime.strptime(row[2], "%Y-%m-%d")
          high = int(row[5])
❷         low = int(row[6])
          dates.append(current_date)
          highs.append(high)
          lows.append(low)

  # Нанесение данных на диаграмму.
  plt.style.use('seaborn')
  fig, ax = plt.subplots()
  ax.plot(dates, highs, c='red')
❸ plt.plot(dates, lows, c='blue')

  # Форматирование диаграммы.
❹ plt.title("Daily high and low temperatures - 2018", fontsize=24)
  ...
```

В точке ❶ создается пустой список `lows` для хранения температурных минимумов, после чего программа извлекает и сохраняет температурный минимум для каждой даты из седьмой позиции каждой строки данных (`row[6]`) ❷. В точке ❸ добавляется вызов `plot()` для температурных минимумов, которые окрашиваются в синий цвет. Затем остается лишь обновить заголовок диаграммы ❹. На рис. 16.4 изображена полученная диаграмма.

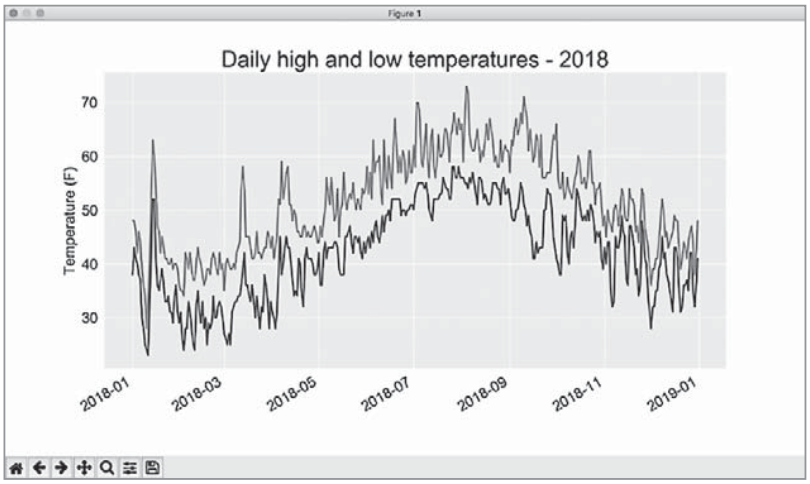

Рис. 16.4. Две серии данных на одной диаграмме

Цветовое выделение части диаграммы

После добавления двух серий данных можно переходить к анализу диапазона температур по дням. Пора сделать последний штрих в оформлении диаграммы: затушевать диапазон между минимальной и максимальной дневной температурой. Для этого мы воспользуемся методом `fill_between()`, который получает серию значений `x` и две серии значений `y` и заполняет область между двумя значениями `y`:

sitka_highs_lows.py

```
  ...
  # Нанесение данных на диаграмму.
  plt.style.use('seaborn')
  fig, ax = plt.subplots()
❶ ax.plot(dates, highs, c='red', alpha=0.5)
  ax.plot(dates, lows, c='blue', alpha=0.5)
❷ plt.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)
  ...
```

Аргумент `alpha` ❶ определяет степень прозрачности вывода. Значение 0 означает полную прозрачность, а 1 (по умолчанию) — полную непрозрачность. Со значением `alpha=0.5` красные и синие линии на графике становятся более светлыми.

В точке ❷ `fill_between()` передается список `dates` для значений `x` и две серии значений `y` `highs` и `lows`. Аргумент `facecolor` определяет цвет закрашиваемой области; мы назначаем ему низкое значение `alpha=0.1`, чтобы заполненная область соединяла две серии данных, не отвлекая зрителя от передаваемой информации. На рис. 16.5 показана диаграмма с закрашенной областью между `highs` и `lows`.

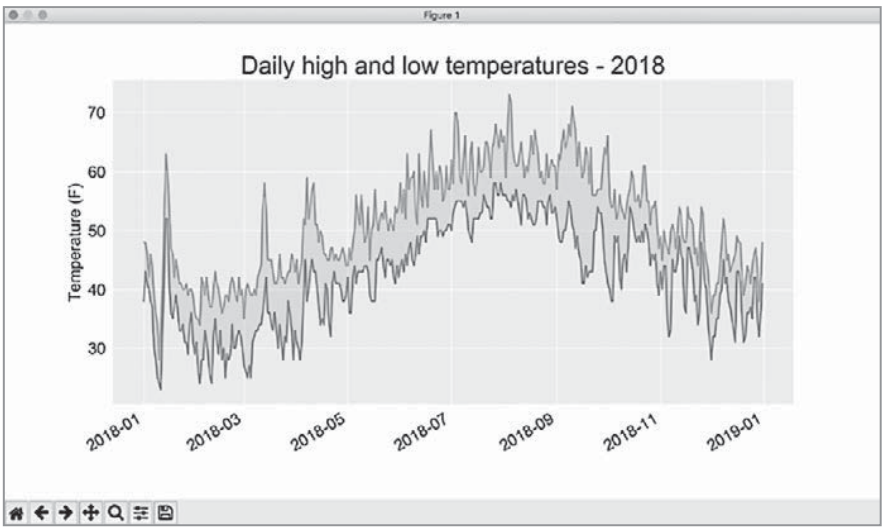

Рис. 16.5. Область между двумя наборами данных закрашена

Закрашенная область подчеркивает величину расхождения между двумя наборами данных.

Проверка ошибок

Программа sitka_highs_lows.py должна нормально работать для погодных данных любого места. Однако некоторые метеорологические станции собирают данные по особым правилам, а на других станциях их не удается собрать из-за сбоев (полностью или частично). Отсутствие данных может привести к исключениям; если исключения не будут обработаны, то программа аварийно завершится.

Для примера попробуем построить график температур для Долины Смерти (штат Калифорния). Скопируйте файл death_valley_2018_simple.csv в каталог с программами этой главы.

Затем выполните следующий код, чтобы просмотреть состав заголовков из файла данных:

death_valley_highs_lows.py

```
import csv

filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    for index, column_header in enumerate(header_row):
        print(index, column_header)
```

Результат выглядит так:

```
0 STATION
1 NAME
2 DATE
3 PRCP
4 TMAX
5 TMIN
6 TOBS
```

Дата находится в той же позиции с индексом 2. Но температурные максимумы и минимумы находятся в позициях с индексами 4 и 5, поэтому нам придется изменить индексы в программе в соответствии с новыми позициями. Вместо того чтобы включать средние показания температуры за день, эта станция регистрирует TOBS — денные за конкретное время наблюдений.

Я удалил одно из показаний температуры из файла, чтобы продемонстрировать, что происходит при отсутствии данных в файле. Внесите изменения в sitka_highs_lows.py, чтобы построить график температур для Долины Смерти по только что определенным индексам, и проследите за происходящим:

death_valley_highs_lows.py

```
  ...
  filename = 'data/death_valley_2018_simple.csv'
  with open(filename) as f:
      ...
      # Получение дат, температурных минимумов и максимумов из файла.
      dates, highs, lows = [], [], []
      for row in reader:
          current_date = datetime.strptime(row[2], '%Y-%m-%d')
❶         high = int(row[4])
          low = int(row[5])
          dates.append(current_date)
  ...
```

В точке ❶ индексы обновляются в соответствии с позициями `TMAX` и `TMIN` в данном файле.

При запуске программы происходит ошибка, как видно из последней строки следующего вывода:

```
Traceback (most recent call last):
  File "death_valley_highs_lows.py", line 15, in <module>
    high = int(row[4])
ValueError: invalid literal for int() with base 10: ''
```

Трассировка показывает, что Python не может обработать максимальную температуру для одной из дат, потому что не может преобразовать пустую строку (`''`) в целое число. Вместо того чтобы копаться в данных и искать отсутствующее значение, мы обработаем ситуации с отсутствием данных напрямую.

При чтении данных из CSV-файла будет выполняться код проверки ошибок для обработки исключений, которые могут возникнуть при разборе наборов данных. Вот как это делается:

death_valley_highs_lows.py

```
...
# Чтение дат, температурных максимумов и минимумов из файла.
filename = 'death_valley_2018_simple.csv'
with open(filename) as f:
    ...
    for row in reader:
            current_date = datetime.strptime(row[2], '%Y-%m-%d')
❶           try:
                high = int(row[4])
                low = int(row[5])
            except ValueError:
❷               print(f"Missing data for {current_date}")
❸           else:
                dates.append(current_date)
                highs.append(high)
                lows.append(low)
```

```
    # Нанесение данных на диаграмму.
    ...

    # Форматирование диаграммы
❹   title = "Daily high and low temperatures - 2018\nDeath Valley, CA"
    plt.title(title, fontsize=20)
    plt.xlabel('', fontsize=16)
    ...
```

При анализе каждой строки данных мы пытаемся извлечь дату, максимальную и минимальную температуру ❶. Если каких-либо данных не хватает, Python выдает ошибку `ValueError`, а мы обрабатываем ее — выводим сообщение с датой, для которой отсутствуют данные ❷. После вывода ошибки цикл продолжает обработку следующей порции данных. Если все данные, относящиеся к некоторой дате, прочитаны без ошибок, выполняется блок `else`, а данные присоединяются к соответствующим спискам ❸. Так как на диаграмме отображается информация для нового места, заголовок изменяется, в него включается название места, а для вывода длинного заголовка используется уменьшенный шрифт ❹.

При выполнении death_valley_highs_lows.py мы видим, что данные отсутствуют только для одной даты:

```
Missing data for 2018-02-18 00:00:00
```

Так как ошибка была обработана корректно, наш код может сгенерировать диаграмму, в которой пропущены отсутствующие данные. Полученная диаграмма изображена на рис. 16.6.

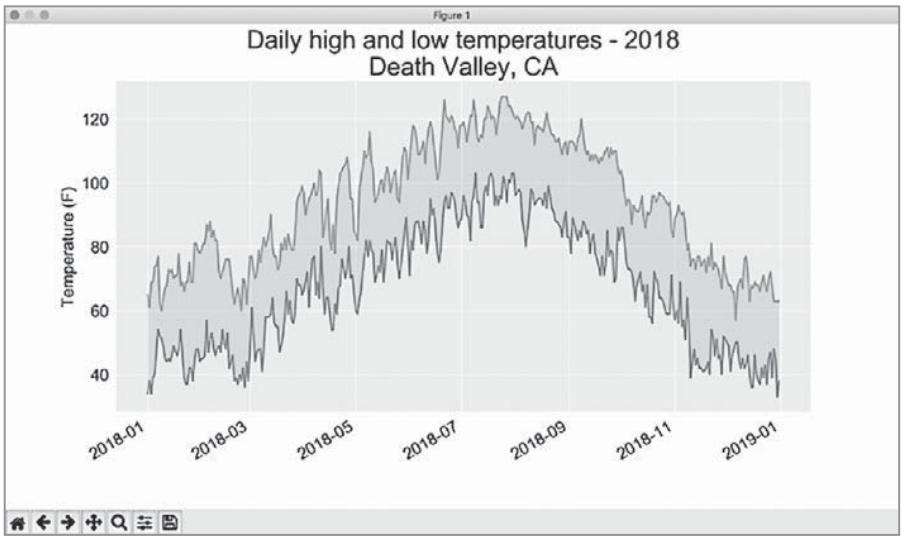

Рис. 16.6. Максимальная и минимальная температура в Долине Смерти

Сравнивая эту диаграмму с диаграммой для Ситки, мы видим, что в Долине Смерти теплее, чем на юго-востоке Аляски (как и следовало ожидать), но при этом температурный диапазон в пустыне более широкий. Высота закрашенной области наглядно демонстрирует этот факт.

Во многих наборах данных, с которыми вы будете работать, вам встретятся отсутствующие, неправильно отформатированные или некорректные данные. В таких ситуациях воспользуйтесь теми инструментами, которые вы освоили в первой половине книги. В данном примере для обработки отсутствующих данных использовался блок `try-except-else`. Иногда команда `continue` используется для пропуска части данных или же данные удаляются после извлечения вызовом `remove()` или `del`. Используйте любое работающее решение — лишь бы в результате у вас получилась осмысленная, точная визуализация.

Загрузка собственных данных

Если вы предпочитаете загрузить собственные погодные данные, выполните следующие действия:

1. Посетите сайт NOAA Climate Data Online по адресу https://www.ncdc.noaa.gov/cdo-web/. В разделе Discover Data щелкните на кнопке Search Tool. В поле Select a Dataset выберите вариант Daily Summaries.
2. Выберите диапазон дат. В разделе Search For выберите вариант ZIP Codes. Введите zip-код интересующего вас места и щелкните на кнопке Search.
3. На следующей странице отображается карта и информация об области, на которой вы хотите сосредоточиться. Под названием места щелкните на кнопке View Full Details либо щелкните на карте, а затем выберите вариант Full Details.
4. Прокрутите данные и щелкните на кнопке Station List, чтобы просмотреть список метеорологических станций в этой области. Выберите одну из станций и щелкните на кнопке Add to Cart. Данные распространяются бесплатно, несмотря на то что на сайте используется обозначение покупательской корзины. Щелкните на изображении корзины в правом верхнем углу.
5. Выберите Output, затем Custom GHCN-Daily CSV. Проверьте на правильность диапазон дат и щелкните на кнопке Continue.
6. На следующей странице вы можете выбрать нужные разновидности данных. Например, можно загрузить один тип данных, ограничившись температурой воздуха, или же загрузить все данные, собираемые станцией. Выберите нужный вариант и щелкните на кнопке Continue.
7. На последней странице выводится сводка запроса. Введите свой адрес электронной почты и щелкните на кнопке Submit Order. Вы получите подтверждение запроса, а через несколько минут придет другое сообщение электронной почты со ссылкой для загрузки данных.

Загруженные данные будут иметь такую же структуру, как и те данные, с которыми вы работали в этом разделе. Их заголовки могут отличаться от представленных в этом разделе, но если вы последуете описанной процедуре, вы сможете построить визуализации интересующих вас данных.

УПРАЖНЕНИЯ

16.1. Осадки в Ситке: Ситка находится в зоне умеренных лесов, поэтому в этой местности выпадает достаточно осадков. В файле данных sitka_weather_2018_simple.csv присутствует заголовок PRCP, представляющий величину ежедневных осадков. Постройте диаграмму по данным этого столбца. Повторите упражнение для Долины Смерти, если вас интересует, сколько осадков выпадает в пустыне.

16.2. Сравнение Ситки с Долиной Смерти: разные масштабы температур отражают разные диапазоны данных. Чтобы точно сравнить температурный диапазон в Ситке с температурным диапазоном Долины Смерти, необходимо установить одинаковый масштаб по оси y. Измените параметры оси y для одной или обеих диаграмм на рис. 16.5 и 16.6 и проведите прямое сравнение температурных диапазонов в этих двух местах (или любых других, которые вас интересуют).

16.3. Сан-Франциско: к какому месту ближе температура в Сан-Франциско: к Ситке или Долине Смерти? Загрузите данные для Сан-Франциско, постройте температурную диаграмму для Сан-Франциско и сравните.

16.4. Автоматические индексы: в этом разделе индексы, соответствующие столбцам TMIN и TMAX, были жестко зафиксированы в коде. Используйте строку данных заголовка для определения индексов этих значений, чтобы ваша программа работала как для Ситки, так и для Долины Смерти. Используйте название станции, чтобы автоматически сгенерировать подходящий заголовок для вашей диаграммы.

16.5. Исследования: постройте еще несколько визуализаций, отражающих любые другие аспекты погоды для интересующих вас мест.

Построение карт с глобальными наборами данных: формат JSON

В этом разделе мы загрузим данные о землетрясениях, произошедших в мире за последний месяц. Затем построим карту, на которой будут обозначены эти землетрясения с указанием их силы. Так как данные хранятся в формате JSON, для работы с ними будет использован модуль `json`. С помощью удобных средств Plotly для работы с картами, мы построим визуализации, отражающие глобальное распределение землетрясений.

Загрузка данных землетрясений

Скопируйте файл eq_1_day_m1.json в каталог, в котором хранятся данные программ этой главы. Землетрясения классифицируются по магнитуде по шкале Рихтера. Файл включает данные по всем землетрясениям с магнитудой M1 и выше, про-

изошедшим за последние 24 часа (на момент написания книги). Данные взяты из одного потока данных Геологического управления США, доступных по адресу https://earthquake.usgs.gov/earthquakes/feed/.

Знакомство с данными

Открыв файл eq_1_day_m1.json, вы увидите, что данные упакованы очень плотно и плохо читаются:

```
{"type":"FeatureCollection","metadata":{"generated":1550361461000,...
{"type":"Feature","properties":{"mag":1.2,"place":"11km NNE of Nor...
{"type":"Feature","properties":{"mag":4.3,"place":"69km NNW of Ayn...
{"type":"Feature","properties":{"mag":3.6,"place":"126km SSE of Co...
{"type":"Feature","properties":{"mag":2.1,"place":"21km NNW of Teh...
{"type":"Feature","properties":{"mag":4,"place":"57km SSW of Kakto...
...
```

Формат этого файла больше предназначен для машин, нежели для людей. Но мы видим, что файл содержит словари, а также информацию, которая нас интересует, включая магнитуды и местоположения землетрясений.

Модуль `json` предоставляет разнообразные инструменты для анализа и обработки данных JSON. Некоторые из этих инструментов позволяют переформатировать файл, чтобы вам было удобнее работать с необработанными данными, прежде чем вы начнете работать с ними на программном уровне.

Для начала загрузим данные и выведем их в формате, лучше подходящем для чтения. Файл очень длинный, поэтому вместо того, чтобы выводить его, запишем данные в новый файл. После этого вы сможете открыть этот файл и прокрутить его к нужной позиции:

eq_explore_data.py

```
  import json

  # Изучение структуры данных.
  filename = 'data/eq_data_1_day_m1.json'
  with open(filename) as f:
❶     all_eq_data = json.load(f)

❷ readable_file = 'data/readable_eq_data.json'
  with open(readable_file, 'w') as f:

❸     json.dump(all_eq_data, f, indent=4)
```

Начнем с импортирования модуля `json`, чтобы загрузить данные из файла, а затем сохранить весь набор данных в `all_eq_data` ❶. Функция `json.load()` преобразует данные в формат, с которым может работать Python: в данном случае огромный словарь. В точке ❷ создается файл для записи тех же данных в более удобочитаемый формат. Функция `json.dump()` получает объект данных JSON и объект файла

и записывает данные в этот файл ❸. Аргумент `indent=4` приказывает `dump()` форматировать данные с отступами, соответствующими структуре данных.

Перейдите в каталог data и откройте файл readable_eq_data.json. Начальная часть выглядит так:

readable_eq_data.json

```
{
      "type": "FeatureCollection",
❶     "metadata": {
          "generated": 1550361461000,
          "url": "https://earthquake.usgs.gov/earthquakes/.../1.0_day.geojson",
          "title": "USGS Magnitude 1.0+ Earthquakes, Past Day",
          "status": 200,
          "api": "1.7.0",
          "count": 158
      },
❷     "features": [
      ...
```

В первую часть файла включена секция с ключом `"metadata"` ❶. По ней можно определить, когда файл был сгенерирован и где можно найти данные в интернете. Также в ней содержится понятный заголовок и количество землетрясений, включенных в файл. За этот 24-часовой период было зарегистрировано 158 землетрясений.

Структура файла geoJSON хорошо подходит для географических данных. Информация хранится в списке, связанном с ключом `"features"` ❷. Так как в файле хранится информация о землетрясениях, эти данные имеют форму списка, в котором каждый элемент соответствует одному землетрясению. На первый взгляд структура кажется запутанной, но она весьма полезна. Например, геолог может сохранить в словаре столько информации о каждом землетрясении, сколько потребуется, а затем объединить все словари в один большой список.

Рассмотрим словарь, представляющий одно землетрясение:

readable_eq_data.json

```
  ...
      {
          "type": "Feature",
❶         "properties": {
              "mag": 0.96,
              ...
❷             "title": "M 1.0 - 8km NE of Aguanga, CA"
          },
❸         "geometry": {
              "type": "Point",
              "coordinates": [
❹                 -116.7941667,
❺                 33.4863333,
                  3.22
              ]
```

```
        },
        "id": "ci37532978"
    },
```

Ключ `"properties"` содержит подробную информацию о каждом землетрясении ❶. Нас прежде всего интересует магнитуда каждого землетрясения, связанная с ключом `"mag"`. Также представляет интерес заголовок каждого землетрясения, содержащий удобную сводку магнитуды и координат ❷.

Ключ `"geometry"` помогает определить, где произошло землетрясение ❸. Эта информация потребуется для географической привязки событий. Долгота `x` ❹ и широта ❺ `y` для каждого землетрясения содержатся в списке, связанном с ключом `"coordinates"`.

Уровень вложенности в этом коде намного выше, чем мы использовали бы в своем коде, и если он покажется запутанным — не огорчайтесь; Python берет на себя большую часть сложности. В любой момент времени мы будем работать с одним или двумя уровнями. Мы начнем с извлечения словаря для каждого землетрясения, зарегистрированного за 24-часовой период.

ПРИМЕЧАНИЕ В географических координатах часто сначала указывается широта, а затем долгота. Вероятно, эта система обозначений возникла из-за того, что люди обнаружили широту задолго до того, как была изобретена концепция долготы. Тем не менее во многих геопространственных библиотеках сначала указывается долгота, а потом широта, потому что этот порядок соответствует системе обозначений (x, y), используемой в математических представлениях. Формат geoJSON использует систему записи (долгота, широта), и если вы будете работать с другой библиотекой, очень важно заранее узнать, какую систему использует эта библиотека.

Построение списка всех землетрясений

Начнем с построения списка, содержащего всю информацию обо всех произошедших землетрясениях.

eq_explore_data.py

```
import json

# Изучение структуры данных.
filename = 'data/eq_data_1_day_m1.json'
with open(filename) as f:
    all_eq_data = json.load(f)

all_eq_dicts = all_eq_data['features']
print(len(all_eq_dicts))
```

Мы берем данные, ассоциированные с ключом `'features'`, и сохраняем их в `all_eq_dicts`. Известно, что файл содержит данные около 158 землетрясений, а вывод подтверждает, что были прочитаны данные всех землетрясений из файла:

```
158
```

Обратите внимание на то, каким коротким получился код. Аккуратно отформатированный файл readable_eq_data.json содержит более 6000 строк. Всего в нескольких строках кода мы прочитали все данные и сохранили их в списке Python. Теперь извлечем данные магнитуд по каждому землетрясению.

Извлечение магнитуд

Имея список, содержащий данные по всем землетрясениям, мы можем перебрать содержимое списка и извлечь всю необходимую информацию. В данном случае это будет магнитуда каждого землетрясения:

eq_explore_data.py

```
  ...
  all_eq_dicts = all_eq_data['features']
❶ mags = []
  for eq_dict in all_eq_dicts:
❷     mag = eq_dict['properties']['mag']
      mags.append(mag)

  print(mags[:10])
```

Создадим пустой список для хранения магнитуд, а затем переберем в цикле словарь `all_eq_dicts` ❶. Внутри цикла каждое землетрясение представляется словарем `eq_dict`. Магнитуда каждого землетрясения хранится в секции `'properties'` словаря с ключом `'mag'` ❷. Каждая магнитуда сохраняется в переменной `mag` и присоединяется к списку `mags`.

Выведем первые 10 магнитуд, чтобы убедиться в том, что были получены правильные данные:

```
[0.96, 1.2, 4.3, 3.6, 2.1, 4, 1.06, 2.3, 4.9, 1.8]
```

Затем мы извлечем данные местоположения (то есть координаты) для каждого землетрясения, а затем построим карту землетрясений.

Извлечение данных местоположения

Данные местоположения хранятся с ключом `"geometry"`. В словаре `geometry` присутствует ключ `"coordinates"`, первыми двумя значениями которого являются долгота и широта. Извлечение данных происходит следующим образом:

eq_explore_data.py

```
  ...
  all_eq_dicts = all_eq_data['features']

  mags, lons, lats = [], [], []
  for eq_dict in all_eq_dicts:
      mag = eq_dict['properties']['mag']
```

```
❶     lon = eq_dict['geometry']['coordinates'][0]
      lat = eq_dict['geometry']['coordinates'][1]
      mags.append(mag)
      lons.append(lon)
      lats.append(lat)

  print(mags[:10])
  print(lons[:5])
  print(lats[:5])
```

Для долгот и широт создаются пустые списки (`lons` и `lats`). Выражение `eq_dict['geometry']` обращается к словарю, представляющему элемент `geometry` данных землетрясения (1). Второй ключ `'coordinates'` извлекает список значений, связанных с ключом `'coordinates'`. Наконец, индекс 0 запрашивает первое значение в списке координат, соответствующее долготе землетрясения.

При выводе первых пяти долгот и широт становится видно, что данные были извлечены правильно:

```
[0.96, 1.2, 4.3, 3.6, 2.1, 4, 1.06, 2.3, 4.9, 1.8]
[-116.7941667, -148.9865, -74.2343, -161.6801, -118.5316667]
[33.4863333, 64.6673, -12.1025, 54.2232, 35.3098333]
```

С этими данными можно переходить к нанесению координат землетрясений на географическую карту.

Построение карты мира

На основании всей информации, собранной к настоящему моменту, можно построить простую карту мира. И хотя первая версия будет выглядеть убого, нужно убедиться в том, что информация отображается правильно, прежде чем сосредоточиться на стиле и визуальном оформлении. Исходная карта выглядит так:

eq_world_map.py

```
  import json

❶ from plotly.graph_objs import Scattergeo, Layout
  from plotly import offline

  ...
  for eq_dict in all_eq_dicts:
      ...

  # Нанесение данных на карту.
❷ data = [Scattergeo(lon=lons, lat=lats)]
❸ my_layout = Layout(title='Global Earthquakes')

❹ fig = {'data': data, 'layout': my_layout}
  offline.plot(fig, filename='global_earthquakes.html')
```

Программа импортирует тип диаграммы `Scattergeo` и класс `Layout`, а затем модуль `offline` для вывода карты ❶. Как и при построении гистограммы, определяется список с именем `data`. Объект `Scattergeo` создается в списке ❷, потому что в любой построенной визуализации можно нанести более одного набора данных. Тип диаграммы `Scattergeo` позволяет наложить на карту диаграмму разброса географических данных. В простейшем варианте использования этого типа диаграммы достаточно передать список долгот и широт.

Мы предоставляем заголовок диаграммы ❸ и создаем словарь с именем `fig`, содержащий данные и макет ❹. Наконец, `fig` передается функции `plot()` с содержательным именем файла для вывода данных.

При выполнении этого файла должна открыться карта, примерный вид которой показан на рис. 16.7. Землетрясения обычно происходят поблизости от границ тектонических плит, что соответствует тому, что мы видим на диаграмме.

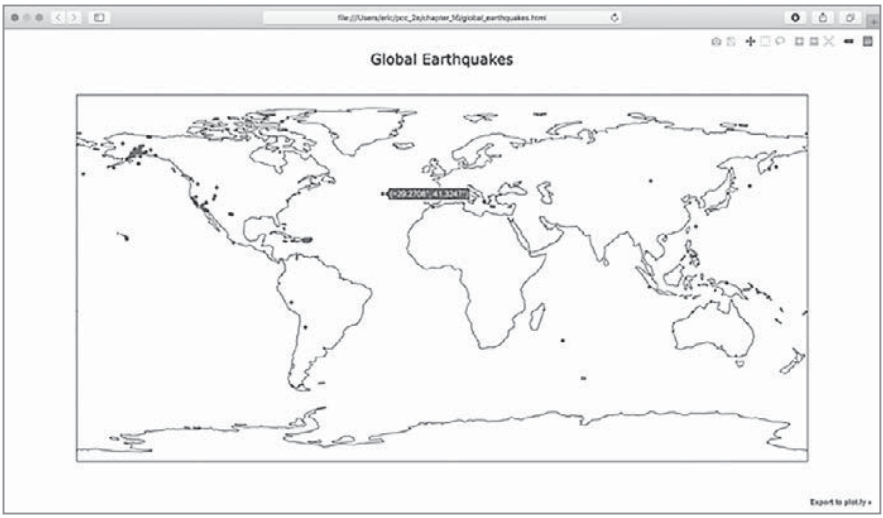

Рис. 16.7. Простая карта с информацией о землетрясениях, произошедших за последние 24 часа

В диаграмму можно внести множество изменений, которые сделают карту более информативной и удобочитаемой. Внесем некоторые из этих изменений.

Другой способ определения данных для диаграммы

Прежде чем переходить к настройке диаграммы, рассмотрим другой способ определения данных для диаграмм Plotly. Для текущей диаграммы список данных определяется в одной строке:

```
data = [Scattergeo(lon=lons, lat=lats)]
```

Это один из простейших способов определения данных диаграмм в Plotly. Тем не менее это не идеальный вариант настройки визуализации. Эквивалентный способ определения данных для текущей диаграммы выглядит так:

```
data = [{
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
}]
```

В этом варианте вся информация о данных структурируется в форме пар «ключ-значение» в словаре. Включив этот код в eq_plot.py, вы увидите ту же диаграмму, которая была сгенерирована ранее. С этим форматом настройки задаются проще, чем с предыдущим форматом.

Настройка размера маркера

Разбираясь с тем, как улучшить оформление карты, следует сосредоточиться на тех аспектах карты, которые вы бы хотели передать более четко. На текущей карте показано местоположение каждого землетрясения, но она не передает силу каждого землетрясения. Пользователь должен сразу видеть, где в мире происходят самые разрушительные землетрясения.

Для этого мы будем изменять размер маркеров в зависимости от магнитуды каждого землетрясения.

eq_world_map.py

```
  import json
  ...
  # Нанесение данных на карту.
  data = [{
      'type': 'scattergeo',
      'lon': lons,
      'lat': lats,
❶     'marker': {
❷         'size': [5*mag for mag in mags],
      },
  }]
  my_layout = Layout(title='Global Earthquakes')
  ...
```

Plotly предоставляет широкие возможности настройки рядов данных, каждый элемент которых может быть представлен в форме «ключ-значение». В данном случае мы используем ключ `'marker'` для определения величины каждого маркера на карте ❶. Мы используем вложенный словарь как значение, связанное с `'marker'`, потому что вы можете задать ряд настроек для всех маркеров ряда.

Мы хотим, чтобы размер маркера соответствовал магнитуде каждого землетрясения. Но если передать только список `mags`, то маркеры получатся слишком мелкими

и вам будет трудно разглядеть различия в размерах. Чтобы получить подходящий размер маркера, необходимо умножить магнитуду на масштабный коэффициент. Для моего экрана хорошо подходит значение 5; возможно, для вашей карты лучше подойдет чуть большее или меньшее значение. Мы используем генератор списка, который сгенерирует правильный размер маркера для каждого значения в списке `mags` ❷.

Примерный вид карты, которая будет построена при выполнении этого кода, показан на рис. 16.8. Карта выглядит значительно лучше, но это еще не все.

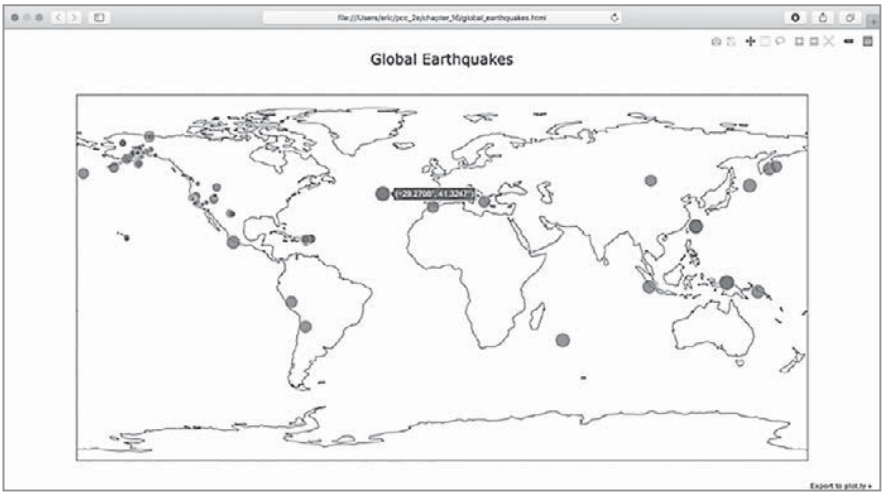

Рис. 16.8. На карте обозначены магнитуды всех землетрясений

Настройка цвета маркеров

Также можно изменить цвет каждого маркера, чтобы обозначить разрушительную силу каждого землетрясения. Для этого мы воспользуемся *цветовыми шкалами* Plotty. Прежде чем вносить изменения, скопируйте файл eq_data_30_day_m1.json в каталог data. Этот файл содержит данные землетрясений за 30-дневный период, и с расширенным набором данных карта будет выглядеть намного интереснее.

Пример использования цветовой шкалы для представления магнитуды каждого землетрясения:

eq_world_map.py

```
  ...
❶ filename = 'data/eq_data_30_day_m1.json'
  ...
  # Нанесение данных на карту.
  data = [{
      ...
      'marker': {
```

```
            'size': [5*mag for mag in mags],
❷           'color': mags,
❸           'colorscale': 'Viridis',
❹           'reversescale': True,
❺           'colorbar': {'title': 'Magnitude'},
        },
    }]
    ...
```

Не забудьте обновить имя файла, чтобы использовать 30-дневный набор данных ❶. Все значимые изменения происходят в словаре `'marker'`, потому что мы изменяем только внешний вид маркеров. Настройка `'color'` сообщает Plotly, какое значение должно использоваться для определения местоположения каждого маркера на цветовой шкале ❷. Для определения цвета будет использоваться список `mags`. Настройка `'colorscale'` указывает Plotly, какой диапазон цветов должен использоваться: цветовая шкала `'Viridis'` лежит в диапазоне от темно-синего до светло-желтого и хорошо подходит для набора данных ❸. Мы присваиваем `'reversescale'` значение `True`, потому что светло-желтый цвет должен использоваться для самых слабых, а темно-синий — для самых разрушительных землетрясений ❹. Настройка `'colorbar'` управляет внешним видом цветной полосы, которая выводится сбоку от карты. Здесь цветовой шкале присваивается заголовок `'Magnitude'`, чтобы было сразу ясно, что представляют разные цвета ❺.

Если запустить программу в этой версии, карта будет выглядеть намного симпатичнее. На рис. 16.9 цветовая шкала обозначает разрушительность отдельных землетрясений. При большом количестве точек данных становится ясно видно, где проходят границы тектонических плит!

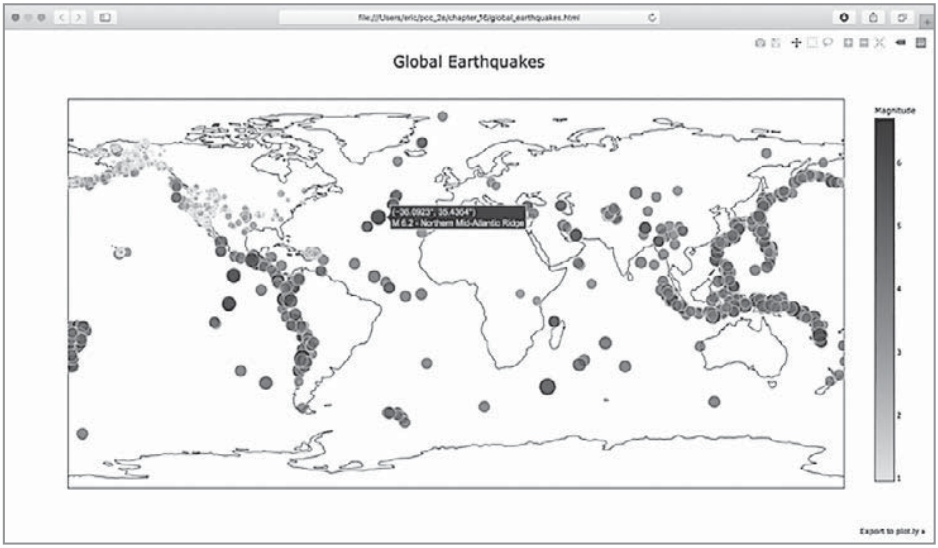

Рис. 16.9. Цвета и размеры маркеров представляют магнитуду землетрясений

Другие цветовые шкалы

Также для оформления диаграммы возможно выбрать другую цветовую шкалу. Чтобы просмотреть доступные варианты цветовых шкал, сохраните следующую короткую программу под именем show_color_scales.py:

show_color_scales.py

```
from plotly import colors

for key in colors.PLOTLY_SCALES.keys():
    print(key)
```

Plotly хранит цветовые шкалы в модели `colors`. Цветовые шкалы определяются в словаре `PLOTLY_SCALES`, а имена цветовых шкал служат ключами в словаре. Результат выполнения с перечнем доступных цветовых шкал:

```
Greys
YlGnBu
Greens
...
Viridis
```

Поэкспериментируйте с этими цветовыми шкалами; помните, что любую шкалу можно «перевернуть» при помощи настройки `reversescale`.

ПРИМЕЧАНИЕ Выведите содержимое словаря PLOTLY_SCALES, и вы увидите, как определены цветовые шкалы. У каждой шкалы имеется начальный и конечный цвет, а некоторые шкалы также определяют один или несколько промежуточных цветов. Plotly интерполирует оттенки между этими цветами.

Добавление подсказки

Чтобы закончить построение карты, мы добавим подсказку, которая будет появляться при наведении указателя мыши на маркер, представляющий землетрясение. Кроме значений долготы и широты, которые должны выводиться по умолчанию, мы выведем магнитуду и описание приблизительного местоположения. Для этого нужно извлечь из файла еще немного данных, а также добавить их в словарь в `data`:

eq_world_map.py

```
  ...
❶ mags, lons, lats, hover_texts = [], [], [], []
  for eq_dict in all_eq_dicts:
      ...
      lat = eq_dict['geometry']['coordinates'][1]
❷     title = eq_dict['properties']['title']
      mags.append(mag)
      lons.append(lon)
      lats.append(lat)
      hover_texts.append(title)
```

```
...

# Нанесение данных на карту.
data = [{
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
❸   'text': hover_texts,
    'marker': {
        ...
    },
}]
...
```

Сначала мы создаем список с именем `hover_texts` для хранения меток, которые будут использоваться для разных маркеров ❶. Секция `title` данных землетрясений содержит текстовое описание магнитуды и местоположения каждого землетрясения в дополнение к его долготе и широте. В точке ❷ мы извлекаем эту информацию, присваиваем ее переменной `title`, а затем присоединяем к списку `hover_texts`.

Если объект `data` содержит ключ `'text'`, Plotly выводит это значение в подсказке маркера, когда пользователь задерживает над ним указатель мыши. Когда вы передаете список, соответствующий количеству маркеров, Plotly извлекает отдельную метку для каждого генерируемого маркера ❸. Запустите программу; при наведении указателя мыши на любой маркер должна появиться подсказка с описанием того, где произошло землетрясение, и его точной магнитудой.

Впечатляет! Приблизительно в 40 строках кода мы создали привлекательную и содержательную карту глобальной сейсмической активности, которая к тому же демонстрирует геологическую структуру планеты. Plotly предоставляет многочисленные средства настройки оформления и поведения ваших визуализаций. С их помощью вы сможете строить диаграммы и карты, содержащие именно ту информацию, которая вам нужна.

УПРАЖНЕНИЯ

16.6. Рефакторинг: в цикле, извлекающем данные из `all_eq_dicts`, используются переменные для сохранения магнитуды, долготы, широты и заголовка каждого землетрясения перед присоединением этих значений к соответствующим спискам. Такой подход был выбран для того, чтобы процесс извлечения данных из файла JSON был более понятным, но в вашем коде он необязателен. Вместо использования временных переменных извлеките каждое значение из `eq_dict` и присоедините его к соответствующему списку в одной строке. В результате тело цикла сократится до четырех строк.

16.7. Автоматизированный заголовок: в этом разделе заголовок задавался вручную при определении `my_layout`; это означает, что вы должны обновлять заголовок при каждом изменении исходного файла. Вместо этого можно воспользоваться заголовком набора данных из метаданных файла JSON. Извлеките это значение, присвойте его переменной и используйте для заголовка карты при определении `my_layout`.

16.8. Недавние землетрясения: в интернете доступны файлы данных с информацией о последних землетрясениях за одночасовой, однодневный и 30-дневный период. Откройте

страницу https://earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php и найдите список ссылок на наборы данных за разные периоды времени. Загрузите один из этих наборов данных и создайте визуализацию последней сейсмической активности.

16.9. Пожары: в ресурсах этой главы присутствует файл world_fires_1_day.csv. Он содержит информацию о пожарах по всему миру, включая долготу, широту и площадь каждого пожара. Используя процедуру обработки данных из первой части этой главы и картографические средства из этого раздела, постройте карту с информацией о том, какие части мира страдают от пожаров.

Обновленные версии этих данных можно загрузить по адресу https://earthdata.nasa.gov/earth-observation-data/near-real-time/irms/active-fire-data/. Ссылки на данные в формате CSV находятся в разделе TXT.

Итоги

В этой главе вы научились работать с реальными наборами данных. Вы узнали, как обрабатывать файлы CSV и JSON и как извлечь данные, на которых вы хотите сосредоточиться. Используя реальные погодные данные, вы освоили новые возможности работы с библиотекой Matplotlib, включая использование модуля `datetime` и возможность нанесения нескольких наборов данных на одну диаграмму. Вы узнали, как нанести данные на карту мира с использованием Plotly и как изменить оформление карт и диаграмм Plotly.

С накоплением опыта работы с файлами CSV и JSON вы сможете обрабатывать практически любые данные, которые вам потребуется проанализировать. Многие сетевые наборы данных могут загружаться хотя бы в одном из этих форматов. После работы с этими форматами вам также будет проще усвоить другие форматы данных.

В следующей главе вы напишете программы для автоматического сбора данных из сетевых источников, а затем создадите визуализации этих данных. Это занятие весьма интересное, если вы рассматриваете программирование как увлечение, и абсолютно необходимое, если вы занимаетесь программированием профессионально.

17 Работа с API

В этой главе вы научитесь писать специализированные программы для построения визуализаций на основании загруженных программами данных. Ваша программа будет использовать *программный интерфейс* (API) веб-приложения для автоматического запроса конкретной информации с сайта (вместо целых страниц). Полученная информация будет использоваться для построения визуализации. Так как программы, написанные по такой схеме, всегда используют самые свежие данные для построения визуализации, даже при быстро изменяющихся данных полученная диаграмма всегда будет оставаться актуальной.

Использование API веб-приложений

API веб-приложения представляет собой часть веб-сайта, предназначенную для взаимодействия с программами, которые используют особым образом построенные URL-адреса для запроса информации. Подобные запросы называются *вызовами API*. Запрашиваемые данные возвращаются в удобном формате (например, JSON или CSV). Многие приложения, зависящие от внешних источников данных (как приложения, интегрирующиеся с сайтами социальных сетей), используют вызовы API.

Git и GitHub

Наша визуализация будет построена на базе информации с GitHub — сайта, организующего совместную работу программистов над проектами. Мы воспользуемся API GitHub для запроса информации о проектах Python и последующего построения интерактивной визуализации относительной популярности этих проектов в Plotly.

Имя GitHub (https://github.com/) происходит от Git — распределенной системы контроля версий, которая позволяет программистам совместно трудиться над проектами. Пользователи Git управляют своим индивидуальным вкладом в проект, чтобы изменения, вносимые одним человеком, не конфликтовали с изменениями, вносимыми другими людьми. Когда вы реализуете новую возможность в проекте, Git отслеживает изменения, внесенные в каждый файл. Если новый код успешно работает, вы *закрепляете* внесенные изменения, и Git записывает новое состояние

проекта. Если же вы допустили ошибку и захотите отменить внесенные изменения, Git позволяет легко вернуться к любому из предыдущих рабочих состояний. (За дополнительной информацией об управлении версиями с использованием Git обращайтесь к приложению Г.) Проекты GitHub хранятся в *репозиториях*, содержащих все ресурсы, связанные с проектом: код, информацию о других участниках, все проблемы или отчеты об ошибках и т. д.

Если проект нравится пользователям GitHub, то пользователи могут «поставить звезду», чтобы продемонстрировать свою поддержку и следить за проектами, которые могут им пригодиться. В этой главе мы напишем программу для автоматической загрузки информации о проектах Python с наибольшим количеством звезд на GitHub, а затем построим содержательную визуализацию таких проектов.

Запрос данных с использованием вызовов API

GitHub поддерживает API (программный интерфейс) для запроса разнообразной информации посредством вызовов API. Чтобы понять, как выглядит вызов API, введите следующий адрес в адресной строке своего браузера и нажмите Enter:

```
https://api.github.com/search/repositories?q=language:python&sort=stars
```

Этот вызов возвращает количество проектов Python, размещенных на GitHub в настоящее время, а также информацию о самых популярных репозиториях Python. Рассмотрим вызов подробнее: первая часть `https://api.github.com/` передает запрос части сайта GitHub, отвечающей на вызовы API. Следующая часть, `search/repositories`, приказывает API провести поиск по всем репозиториям в GitHub.

Вопросительный знак после `repositories` означает, что мы собираемся передать аргумент. Символ `q` обозначает запрос (Query), а знак равенства начинает определение запроса (`q=`). Выражение `language:python` указывает, что запрашивается информация только по репозиториям, для которых основным языком указан Python. Завершающая часть, `&sort=stars`, сортирует проекты по количеству присвоенных им звезд.

В следующем фрагменте приведены несколько начальных строк ответа.

```
  {
❶     "total_count": 3494012,
❷     "incomplete_results": false,
❸     "items": [
        {
          "id": 21289110,
          "node_id": "MDEwOlJlcG9zaXRvcnkyMTI4OTExMA==",
          "name": "awesome-python",
          "full_name": "vinta/awesome-python",
          ...
```

Вероятно, по виду ответа вы уже поняли, что этот URL-адрес не предназначен для обычных пользователей, потому что ответ закодирован в формате, рассчитанном

на машинную обработку. На момент написания книги на GitHub было найдено 3 494 012 проектов Python ❶. Значение `"incomplete_results"` равно `false`, а значит, запрос был обработан успешно (информация не является неполной) ❷. Если бы у GitHub возникли проблемы с полной обработкой запроса API, то в этом поле было бы возвращено значение `true`. Возвращаемые данные отображаются в списке `"items"` с информацией о самых популярных проектах Python на GitHub ❸.

Установка пакета requests

Пакет `Requests` предоставляет удобные средства для запроса информации с сайтов из программ Python и анализа полученных ответов. Для установки `requests` используется `pip`:

```
$ python -m pip install --user requests
```

Эта команда приказывает Python запустить модуль `pip` и включить пакет `Requests` в установку Python текущего пользователя. Если для запуска программ или установки пакетов вы используете `python3` или другую команду, проследите за тем, чтобы здесь использовалась та же команда.

ПРИМЕЧАНИЕ Если эта команда не работает в macOS, попробуйте снова выполнить команду без флага `--user`.

Обработка ответа API

Теперь мы напишем программу, которая выдает вызов API для поиска на Github проектов Python с наибольшим количеством звезд:

python_repos.py

```
❶ import requests

  # Создание вызова API и сохранение ответа.
❷ url = 'https://api.github.com/search/repositories?q=language:python&sort=stars'
❸ headers = {'Accept': 'application/vnd.github.v3+json'}
❹ r = requests.get(url, headers=headers)
❺ print(f"Status code: {r.status_code}")

  # Сохранение ответа API в переменной.
❻ response_dict = r.json()

  # Обработка результатов.
  print(response_dict.keys())
```

В точке ❶ импортируется модуль `requests`. В точке ❷ URL-адрес вызова API сохраняется в переменной `url`. В настоящее время GitHub использует третью версию API, поэтому для вызова API определяются заголовки ❸, которые явно требуют использовать эту версию API. После этого модуль `requests` используется для вызова ❹.

Мы вызываем метод `get()` и передаем ему URL и заголовок, а объект ответа сохраняется в переменной `r`. Объект ответа содержит атрибут `status_code`, в котором хранится признак успешного выполнения запроса. (Код 200 — признак успешного ответа.) В точке ❺ программа выводит значение `status_code`, чтобы вы могли убедиться в том, что вызов был обработан успешно.

API возвращает информацию в формате JSON, поэтому в программе используется метод `json()` ❻ для преобразования информации в словарь Python. Полученный словарь сохраняется в переменной `response_dict`.

Наконец, программа выводит ключи словаря `response_dict`, и мы видим следующее:

```
Status code: 200
dict_keys(['total_count', 'incomplete_results', 'items'])
```

Код статуса 200 означает, что запрос был обработан успешно. Словарь ответа содержит всего три ключа: `'total_count'`,`'incomplete_results'` и `'items'`. Присмотримся повнимательнее к словарю ответа.

ПРИМЕЧАНИЕ Подобные простые вызовы должны возвращать полный набор результатов, поэтому значение, связанное с `'incomplete_results'`, можно достаточно безопасно игнорировать. Но если ваша программа выдает более сложные вызовы API, обязательно проверяйте это значение.

Работа со словарем ответа

Итак, полученная при вызове API информация хранится в словаре, и мы можем заняться работой с данными. Для начала построим сводку с обобщенными сведениями — это позволит убедиться в том, что вызов вернул ожидаемую информацию, и перейти к анализу интересующих данных.

python_repos.py

```
  import requests

  # Создание вызова API и сохранение ответа.
  ...
  # Сохранение ответа API в переменной.
  response_dict = r.json()
❶ print(f"Total repositories: {response_dict['total_count']}")

  # Анализ информации о репозиториях.
❷ repo_dicts = response_dict['items']
  print(f"Repositories returned: {len(repo_dicts)}")

  # Анализ первого репозитория.
❸ repo_dict = repo_dicts[0]
❹ pprint(f"\nKeys: {len(repo_dict)}")
❺ for key in sorted(repo_dict.keys()):
      print(key)
```

В точке ❶ выводится значение, связанное с `'total_count'`, которое представляет общее количество репозиториев Python в GitHub.

Значение, связанное с `'items'`, представляет собой список со словарями, каждый из которых содержит данные об одном репозитории Python. В точке ❷ этот список словарей сохраняется в `repo_dicts`. Затем программа выводит длину `repo_dicts`, чтобы пользователь видел, по какому количеству репозиториев имеется информация.

Чтобы получить первое представление об информации, возвращенной по каждому репозиторию, программа извлекает первый элемент из `repo_dicts` и сохраняет его в `repo_dict` ❸. Затем программа выводит количество ключей в словаре — это значение определяет объем доступной информации ❹. В точке ❺ выводятся все ключи словаря; по ним можно понять, какая информация включена в ответ.

Из сводки начинает вырисовываться более четкая картина полученных данных:

```
Status code: 200
Total repositories: 3494030
Repositories returned: 30

❶ Keys: 73
  archive_url
  archived
  assignees_url
  ...
  url
  watchers
  watchers_count
```

API GitHub возвращает подробную информацию о каждом репозитории: в `repo_dict` 73 ключа ❶. Просмотр ключей дает представление о том, какую информацию можно извлечь о проекте. (Чтобы узнать, какую информацию можно получить через API, следует либо прочитать документацию, либо проанализировать информацию в коде, как мы и поступаем.)

Прочитаем значения некоторых ключей `repo_dict`:

python_repos.py

```
  ...
  # Анализ информации о репозиториях.
  repo_dicts = response_dict['items']
  print(f"Repositories returned: {len(repo_dicts)}")

  # Анализ первого репозитория.
  repo_dict = repo_dicts[0]

  print("\nSelected information about first repository:")
❶ print(f"Name: {repo_dict['name']}")
❷ print(f"Owner: {repo_dict['owner']['login']}")
❸ print(f"Stars: {repo_dict['stargazers_count']}")
  print(f"Repository: {repo_dict['html_url']}")
```

```
❹ print(f"Created: {repo_dict['created_at']}")
❺ print(f"Updated: {repo_dict['updated_at']}")
  print(f"Description: {repo_dict['description']}")
```

В программе выводятся значения, связанные с некоторыми ключами словаря первого репозитория. В точке ❶ выводится имя проекта. Владельца проекта представляет целый словарь, поэтому в точке ❷ ключ `owner` используется для обращения к словарю, представляющему владельца, после чего ключ `login` используется для получения регистрационного имени владельца. В точке ❸ выводится количество звезд, заработанных проектом, и URL репозитория GitHub проекта. Далее выводится дата создания ❹ и последнего обновления репозитория ❺. В завершение выводится описание репозитория; вывод должен выглядеть примерно так:

```
Status code: 200
Total repositories: 3494032
Repositories returned: 30

Selected information about first repository:
Name: awesome-python
Owner: vinta
Stars: 61549
Repository: https://github.com/vinta/awesome-python
Created: 2014-06-27T21:00:06Z
Updated: 2019-02-17T04:30:00Z
Description: A curated list of awesome Python frameworks, libraries, software
    and resources
```

Из вывода видно, что на момент написания книги самым «звездным» проектом Python на GitHub был проект *awesome-python*, владельцем которого является пользователь *vinta*, и звезды этот проект получил более чем от 60 000 пользователей GitHub. Мы видим URL репозитория проекта, дату создания (июнь 2014 г.) и то, что проект недавно обновлялся. Наконец, из описания следует, что `awesome-python` содержит список самых популярных ресурсов Python.

Сводка самых популярных репозиториев

При построении визуализации этих данных на диаграмму необходимо нанести более одного репозитория. Напишем цикл для вывода информации о каждом репозитории, возвращаемом вызовом API, чтобы все эти репозитории можно было включить в визуализацию:

python_repos.py

```
  ...
  # Анализ информации о репозиториях.
  repo_dicts = response_dict['items']
  print(f"Repositories returned: {len(repo_dicts)}")

❶ print("\nSelected information about each repository:")
❷ for repo_dict in repo_dicts:
```

```
        print(f"\nName: {repo_dict['name']}")
        print(f"Owner: {repo_dict['owner']['login']}")
        print(f"Stars: {repo_dict['stargazers_count']}")
        print(f"Repository: {repo_dict['html_url']}")
        print(f"Description: {repo_dict['description']}")
```

В точке ❶ выводится приветственное сообщение. В точке ❷ перебираются все словари в `repo_dicts`. Внутри цикла выводится имя каждого проекта, его владелец, количество звезд, URL на GitHub и краткое описание проекта:

```
Status code: 200
Total repositories: 3494040
Repositories returned: 30

Selected information about each repository:

Name: awesome-python
Owner: vinta
Stars: 61549
Repository: https://github.com/vinta/awesome-python
Description: A curated list of awesome Python frameworks, libraries, software
  and resources

Name: system-design-primer
Owner: donnemartin
Stars: 57256
Repository: https://github.com/donnemartin/system-design-primer
Description: Learn how to design large-scale systems. Prep for the system
  design interview.  Includes Anki flashcards.
...

Name: python-patterns
Owner: faif
Stars: 19058
Repository: https://github.com/faif/python-patterns
Description: A collection of design patterns/idioms in Python
```

В этих результатах встречаются интересные проекты; возможно, вам стоит присмотреться к некоторым из них... Но не увлекайтесь, потому что мы собираемся создать визуализацию, которая существенно упростит чтение результатов.

Проверка ограничений частоты обращений API

Многие API ограничивают частоту обращений; иначе говоря, существует предел для количества запросов в определенный промежуток времени. Чтобы узнать, не приближаетесь ли вы к ограничениям GitHub, введите в браузере адрес https://api.github.com/rate_limit. Вы получите ответ, который выглядит примерно так:

```
{
  "resources": {
    "core": {
      "limit": 60,
      "remaining": 58,
```

```
        "reset": 1550385312
      },
❶     "search": {
❷       "limit": 10,
❸       "remaining": 8,
❹       "reset": 1550381772
      }
    },
  ...
```

В этих данных нас интересует частота обращений для поискового API ❶. В точке ❷ видно, что предельная частота составляет 10 запросов в минуту и что на текущую минуту осталось еще 8 запросов ❸. Значение `reset` представляет *Unix-время*, или *эпохальное время* (число секунд, прошедших с полуночи 1 января 1970 г.) момента, когда произойдет сброс квоты ❹. При достижении предельного количества обращений вы получите короткий ответ, уведомляющий о достижении предела API. Если это произойдет, просто подождите, пока квота будет сброшена.

ПРИМЕЧАНИЕ Многие API требуют регистрации и получения ключа API для совершения вызовов. На момент написания для GitHub такого требования не было, но если вы получите ключ API, предельная частота обращений для ваших программ значительно увеличится.

Визуализация репозиториев с использованием Plotly

Теперь, с появлением интересных данных, мы построим визуализацию, демонстрирующую относительную популярность проектов Python в GitHub. Мы построим интерактивную столбцовую диаграмму: высота каждого столбца будет представлять количество звезд у проекта. Щелчок на столбце будет открывать домашнюю страницу проекта на GitHub. Сохраните копию программы, над которой вы работаете, под именем python_repos_visual.py, а затем приведите ее к следующему виду:

python_repos_visual.py

```
import requests

❶ from plotly.graph_objs import Bar
  from plotly import offline

❷  # Создание вызова API и сохранение ответа.
  url = 'https://api.github.com/search/repositories?q=language:python&sort=stars'
  headers = {'Accept': 'application/vnd.github.v3+json'}
  r = requests.get(url, headers=headers)
  print(f"Status code: {r.status_code}")

  # Обработка результатов.
  response_dict = r.json()
  repo_dicts = response_dict['items']
❸ repo_names, stars = [], []
  for repo_dict in repo_dicts:
```

```
        repo_names.append(repo_dict['name'])
        stars.append(repo_dict['stargazers_count'])

    # Построение визуализации.
❹ data = [{
        'type': 'bar',
        'x': repo_names,
        'y': stars,
    }]
❺ my_layout = {
        'title': 'Most-Starred Python Projects on GitHub',
        'xaxis': {'title': 'Repository'},
        'yaxis': {'title': 'Stars'},
    }

    fig = {'data': data, 'layout': my_layout}
    offline.plot(fig, filename='python_repos.html')
```

Мы импортируем класс `Bar` и модуль `offline` из `plotly` ❶. Импортировать класс `Layout` не нужно, потому что для определения макета будет использоваться словарь, как это делалось со списком `data` в проекте с землетрясениями из главы 16. Затем выводится статус ответа на вызов API, чтобы мы сразу узнали о возможной проблеме с вызовом API ❷. Часть кода обработки ответа API удалена, потому что фаза исследования данных осталась позади; мы знаем, что получены именно те данные, которые нам нужны.

В точке ❸ создаются два пустых списка для хранения данных, включаемых в диаграмму. Нам понадобится имя каждого проекта (для пометки столбцов) и количество звезд, определяющее высоту столбцов. В цикле имя каждого проекта и количество звезд присоединяются к соответствующему списку.

Затем определяется список `data` ❹. Он содержит словарь вроде того, который использовался в главе 16: он определяет тип диаграммы и содержит значения по осям `x` и `y`. По оси `x` размещаются названия проектов, а по оси `y` — количество звезд, назначенное каждому проекту.

В точке ❺ макет диаграммы определяется при помощи словаря. Вместо того чтобы создавать экземпляр класса `Layout`, мы строим словарь с нужными спецификациями макета. Далее назначается заголовок для диаграммы в целом, а также метки каждой оси.

Полученная диаграмма изображена на рис. 17.1. Мы видим, что несколько первых проектов существенно популярнее остальных, но все эти проекты занимают важное место в экосистеме Python.

Доработка диаграмм Plotly

Немного доработаем стилевое оформление диаграммы. Как было показано в главе 16, все директивы стилевого оформления включаются в виде пар «ключ-значение» в словари `data` и `my_layout`.

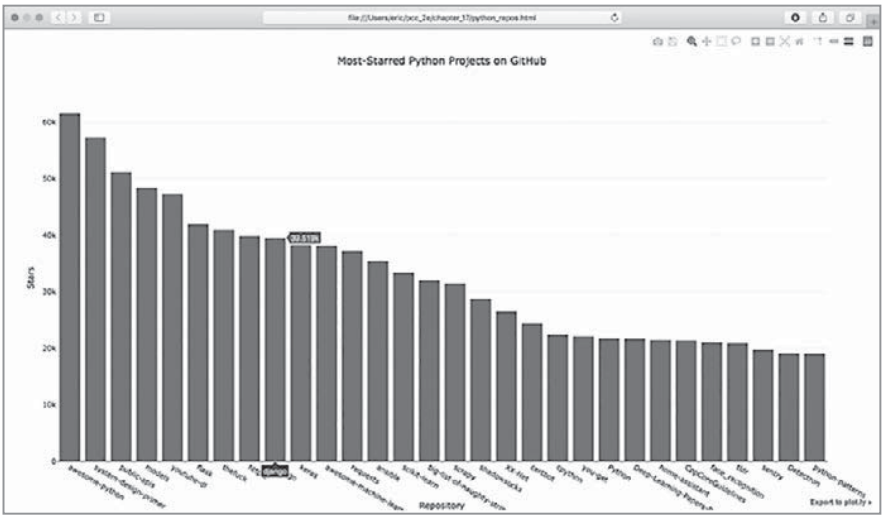

Рис. 17.1. Проекты Python на GitHub с наибольшим количеством звезд

Изменения в объекте `data` влияют на вид столбцов. Ниже приведена обновленная версия объекта `data` для диаграммы, которая назначает конкретный цвет и хорошо заметную границу для каждого столбца:

python_repos_visuals.py

```
...
data = [{
    'type': 'bar',
    'x': repo_names,
    'y': stars,
    'marker': {
        'color': 'rgb(60, 100, 150)',
        'line': {'width': 1.5, 'color': 'rgb(25, 25, 25)'}
    },
    'opacity': 0.6,
}]
...
```

Настройки `marker` влияют на внешний вид столбцов. Мы назначаем столбцам синий цвет и указываем, что они должны иметь серую границу толщиной 1,5 пиксела. Также для столбцов устанавливается прозрачность `0.6`, чтобы изображение казалось немного размытым.

Затем мы внесем изменения в `my_layout`:

python_repos_visual.py

```
  ...
  my_layout = {
      'title': 'Most-Starred Python Projects on GitHub',
```

```
❶     'titlefont': {'size': 28},
❷     'xaxis': {
          'title': 'Repository',
          'titlefont': {'size': 24},
          'tickfont': {'size': 14},
      },
❸     'yaxis': {
          'title': 'Stars',
          'titlefont': {'size': 24},
          'tickfont': {'size': 14},
      },
  }
  ...
```

Ключ `'titlefont'` добавлен для определения размера шрифта общего заголовка диаграммы ❶. В словарь `'xaxis'` добавляются настройки для управления размером шрифта заголовка оси `x` (`'titlefont'`) и меток делений (`'tickfont'`) ❷. Так как речь идет об отдельных вложенных словарях, вы можете включить ключи для цвета и семейства шрифтов заголовков осей и меток делений. В точке ❸ аналогичные настройки определяются для оси `y`.

На рис. 17.2 изображена диаграмма с измененным оформлением.

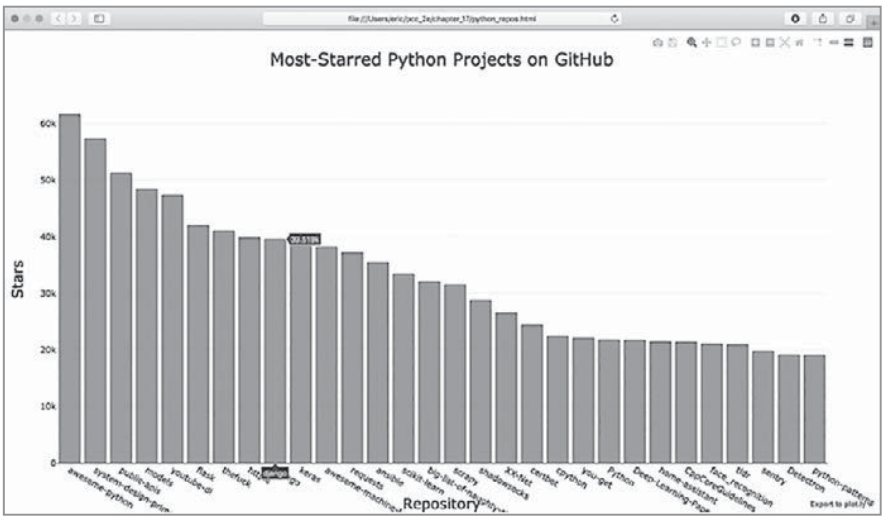

Рис. 17.2. Диаграмма с улучшенным оформлением

Добавление подсказок

В Plotly при наведении указателя мыши на отдельный столбец отображается информация, которую этот столбец представляет. В текущей версии экранная *подсказка* (tooltip) отображает количество звезд проекта. Давайте создадим нестандартную подсказку, которая также будет выводить описание каждого проекта.

Чтобы сгенерировать подсказки и изменить объект `data`, необходимо извлечь дополнительные данные:

python_repos_visual.py

```
  ...
  # Обработка результатов.
  response_dict = r.json()
  repo_dicts = response_dict['items']
❶ repo_names, stars, labels = [], [], []
  for repo_dict in repo_dicts:
      repo_names.append(repo_dict['name'])
      stars.append(repo_dict['stargazers_count'])

❷     owner = repo_dict['owner']['login']
      description = repo_dict['description']
❸     label = f"{owner}<br />{description}"
      labels.append(label)


  # Построение визуализации.
  data = [{
      'type': 'bar',
      'x': repo_names,
      'y': stars,
❹     'hovertext': labels,
      'marker': {
          'color': 'rgb(60, 100, 150)',
          'line': {'width': 1.5, 'color': 'rgb(25, 25, 25)'}
      },
      'opacity': 0.6,
  }]
  ...
```

Сначала определяется новый пустой список `labels` для хранения текста, который должен выводиться для каждого проекта ❶. В цикле, где происходит обработка данных, мы извлекаем владельца и описание для каждого проекта ❷. Plotly позволяет использовать разметку HTML в текстовых элементах, поэтому мы сгенерируем для метки текст с разрывом строки (`<br />`) между именем владельца проекта и описанием ❸. Затем метка сохраняется в списке `labels`.

В словарь `data` добавляется запись с ключом `'hovertext'`, которой присваивается только что созданный список ❹. При создании каждого столбца Plotly извлекает метки из списка и выводит их только в тот момент, когда пользователь задерживает указатель мыши над столбцом.

Полученная диаграмма изображена на рис. 17.3.

Добавление активных ссылок на диаграмму

Так как Plotly позволяет использовать HTML в текстовых элементах, диаграмму можно легко дополнить ссылками. Используем метки оси `x` для того, чтобы пользо-

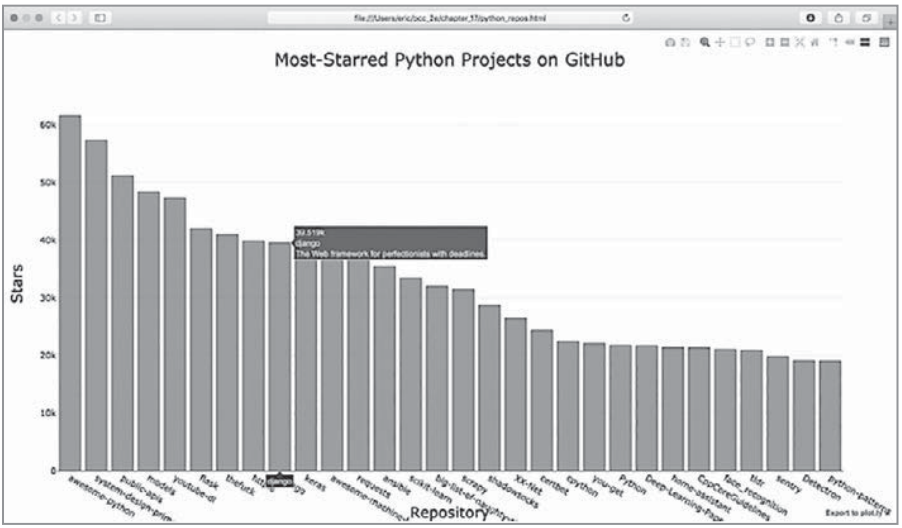

Рис. 17.3. При наведении указателя мыши на столбец выводится информация о владельце и описание проекта

ватель мог открыть домашнюю страницу проекта на GitHub. Необходимо извлечь URL-адреса из данных и использовать их при генерировании меток для оси x:

python_repos_visual.py

```
  ...
  # Обработка результатов.
  response_dict = r.json()
  repo_dicts = response_dict['items']
❶ repo_links, stars, labels = [], [], []
  for repo_dict in repo_dicts:
      repo_name = repo_dict['name']
❷     repo_url = repo_dict['html_url']
❸     repo_link = f"<a href='{repo_url}'>{repo_name}</a>"
      repo_links.append(repo_link)

      stars.append(repo_dict['stargazers_count'])
      ...

  # Построение визуализации.
  data = [{
      'type': 'bar',
❹     'x': repo_links,
      'y': stars,
      ...
  }]
  ...
```

Список, создаваемый на базе `repo_names`, переименован в `repo_links`, чтобы он более точно отражал свойства информации, объединяемой для диаграммы ❶. Затем мы извлекаем URL-адрес проекта из `repo_dict` и присваиваем его временной переменной `repo_url` ❷. В точке ❸ генерируется ссылка на проект. Для этого используется якорный тег HTML вида `<a href='URL'>`*текст ссылки*`</a>`. Затем ссылка присоединяется к списку `repo_links`.

В точке ❹ этот список используется для получения значений оси `x` диаграммы. Результат выглядит так же, как прежде, но теперь пользователь может щелкнуть на любом из имен проектов в нижней части диаграммы, чтобы посетить домашнюю страницу этого проекта на GitHub. Нам удалось построить интерактивную, содержательную визуализацию данных, полученных через API!

Подробнее о Plotly и GitHub API

Если вам захочется больше узнать о работе с диаграммами Plotly, в интернете есть пара хороших мест. Руководство пользователя Plotly находится по адресу https://plot.ly/python/user-guide/. Этот ресурс поможет лучше понять, как Plotly использует ваши данные для построения визуализации и почему выбран именно такой подход к определению визуализаций данных.

На странице https://plot.ly/python/reference/ перечислены все параметры, используемые для настройки визуализаций Plotly. Здесь приведены все возможные типы диаграмм, а также все атрибуты, которые могут устанавливаться для всех параметров конфигурации.

За дополнительной информацией о GitHub API обращайтесь к документации по адресу https://developer.github.com/v3/. Из документации вы узнаете, как извлекать разнообразную информацию из GitHub. Если у вас уже имеется учетная запись GitHub, вы можете работать как со своими данными, так и с общедоступными данными из репозиториев других пользователей.

API Hacker News

Чтобы познакомиться с использованием вызовов API для других сайтов, мы обратимся к сайту Hacker News (http://news.ycombinator.com/). На этом сайте пользователи делятся друг с другом статьями, посвященными программированию и технологиям, а также активно обсуждают эти статьи. API сайта Hacker News предоставляет доступ ко всем статьям и комментариям на сайте, а для его использования не требуется регистрация с получением ключа.

Следующий вызов возвращает информацию о текущей самой популярной статье (на момент написания книги):

```
https://hacker-news.firebaseio.com/v0/item/19155826.json
```

Если ввести этот URL в браузере, вы увидите, что текст на странице заключен в фигурные скобки; это означает, что перед вами словарь. Но в ответе трудно разобраться без дополнительного форматирования. Обработаем URL методом `json.dump()`, как было сделано в проекте с землетрясениями из главы 16, чтобы нам было удобнее изучать возвращенную информацию:

hn_article.py

```
import requests
import json

# Вызов API и сохранение ответа.
url = 'https://hacker-news.firebaseio.com/v0/item/19155826.json'
r = requests.get(url)
print(f"Status code: {r.status_code}")

# Анализ структуры данных.
response_dict = r.json()
readable_file = 'data/readable_hn_data.json'
with open(readable_file, 'w') as f:
    json.dump(response_dict, f, indent=4)
```

В этой программе все должно быть вам знакомо, потому что все эти средства использовались в двух предшествующих главах. Ответ представляет собой словарь с информацией о статье с идентификатором 19155826:

readable_hn_data.json

```
  {
      "by": "jimktrains2",
❶     "descendants": 220,
      "id": 19155826,
❷     "kids": [
          19156572,
          19158857,
          ...
      ],
      "score": 722,
      "time": 1550085414,
❸     "title": "Nasa's Mars Rover Opportunity Concludes a 15-Year Mission",
      "type": "story",
❹     "url": "https://www.nytimes.com/.../mars-opportunity-rover-dead.html"
  }
```

Словарь содержит ряд ключей, которые могут нам пригодиться. Ключ `'descendants'` содержит количество комментариев, полученных статьей ❶. Ключ `'kids'` предоставляет идентификаторы всех комментариев, сделанных непосредственно в ответ на эту статью ❷. У каждого из этих комментариев могут быть свои дочерние комментарии, так что количество потомков у статьи может быть больше количества дочерних комментариев. Также в данных виден заголовок обсуждаемой статьи ❸ и ее URL-адрес ❹.

Следующий URL возвращает простой список всех идентификаторов текущих популярных статей на сайте Hacker News:

```
https://hacker-news.firebaseio.com/v0/topstories.json
```

При помощи этого вызова можно узнать, какие статьи находятся на домашней странице, а затем сгенерировать серию вызовов API, аналогичных приведенному выше. Это позволит нам вывести сводку всех статей, находящихся на главной странице Hacker News в данный момент:

hn_submissions.py

```
  from operator import itemgetter

  import requests

  # Создание вызова API и сохранение ответа.
❶ url = 'https://hacker-news.firebaseio.com/v0/topstories.json'
  r = requests.get(url)
  print(f"Status code: {r.status_code}")

  # Обработка информации о каждой статье.
❷ submission_ids = r.json()
❸ submission_dicts = []
  for submission_id in submission_ids[:30]:
      # Создание отдельного вызова API для каждой статьи.
❹     url = f"https://hacker-news.firebaseio.com/v0/item/{submission_id}.json"
      r = requests.get(url)
      print(f"id: {submission_id}\tstatus: {r.status_code}")
      response_dict = r.json()

      # Построение словаря для каждой статьи.
❺     submission_dict = {
          'title': response_dict['title'],
          'hn_link': f"http://news.ycombinator.com/item?id={submission_id}",
          'comments': response_dict['descendants'],
      }
❻     submission_dicts.append(submission_dict)

❼ submission_dicts = sorted(submission_dicts, key=itemgetter('comments'),
                            reverse=True)

❽ for submission_dict in submission_dicts:
      print(f"\nTitle: {submission_dict['title']}")
      print(f"Discussion link: {submission_dict['hn_link']}")
      print(f"Comments: {submission_dict['comments']}")
```

Сначала программа создает вызов API и выводит статус ответа ❶. Этот вызов API возвращает список идентификаторов 500 самых популярных статей на Hacker News на момент выдачи вызова. Текст ответа преобразуется в список Python ❷, который сохраняется в переменной `submission_ids`. Идентификаторы будут использованы для построения набора словарей, каждый из которых содержит информацию об одной из текущих статей.

В точке ❸ создается пустой список с именем `submission_dicts` для хранения словарей. Далее программа перебирает идентификаторы 30 самых популярных статей и выдает новый вызов API для каждой статьи, генерируя URL с текущим значением `submission_id` ❹. Также выводится статус каждого запроса, чтобы мы могли проверить, успешно ли он был обработан.

В точке ❺ создается словарь для текущей обрабатываемой статьи, в котором сохраняется заголовок статьи, ссылка на страницу с ее обсуждением и количество комментариев к статье. Затем каждый элемент `submission_dict` присоединяется к списку `submission_dicts` ❻.

Статьи Hacker News ранжируются по общей системе, основанной на нескольких факторах: сколько раз за статью голосовали, сколько комментариев она получила и давно ли была опубликована. Требуется отсортировать список словарей по количеству комментариев. Для этого мы используем функцию `itemgetter()` ❼ из модуля `operator`. Мы передаем этой функции ключ `'comments'`, а она извлекает значение, связанное с данным ключом, из каждого словаря в списке. Функция `sorted()` затем использует это значение для сортировки списка. Мы сортируем список в обратном порядке, чтобы публикации с наибольшим количеством комментариев оказались на первом месте.

После того как список будет отсортирован, мы перебираем элементы ❽ и выводим для каждой из самых популярных статей три атрибута: заголовок, ссылку на страницу обсуждения и текущее количество комментариев:

```
Status code: 200
id: 19155826    status: 200
id: 19180181    status: 200
id: 19181473    status: 200
...

Title: Nasa's Mars Rover Opportunity Concludes a 15-Year Mission
Discussion link: http://news.ycombinator.com/item?id=19155826
Comments: 220

Title: Ask HN: Is it practical to create a software-controlled model rocket?
Discussion link: http://news.ycombinator.com/item?id=19180181
Comments: 72

Title: Making My Own USB Keyboard from Scratch
Discussion link: http://news.ycombinator.com/item?id=19181473
Comments: 62
...
```

Аналогичный процесс применяется для обращения и анализа информации из любого API. С такими данными вы сможете построить визуализацию, показывающую, какие публикации вызывали наиболее активные обсуждения за последнее время. Этот метод также лежит в основе приложений, предоставляющих специализированные средства чтения таких сайтов, как Hacker News. Чтобы больше узнать об информации, доступной через Hacker News API, посетите страницу документации по адресу https://github.com/HackerNews/API/.

УПРАЖНЕНИЯ

17.1. Другие языки: измените вызов API в программе python_repos.py так, чтобы на диаграмме отображались самые популярные проекты на других языках. Попробуйте такие языки, как JavaScript, Ruby, C, Java, Perl, Haskell и Go.

17.2. Активные обсуждения: на основании данных из hn_submissions.py постройте столбцовую диаграмму самых активных обсуждений, проходящих на Hacker News. Высота каждого столбца должна соответствовать количеству комментариев к каждой статье. Метка столбца должна включать заголовок статьи, а сам столбец должен служить ссылкой на страницу обсуждения этой публикации.

17.3. Тестирование python_repos.py: в python_repos.py для проверки успешности вызова API выводится значение `status_code`. Напишите программу test_python_repos.py, которая использует модуль `unittest` для проверки того, что значение `status_code` равно 200. Придумайте другие условия, которые могут проверяться при тестировании, — например, что количество возвращаемых элементов совпадает с ожидаемым, а общее количество репозиториев превышает некоторый порог.

17.4. Дальнейшие исследования: ознакомьтесь с документацией Plotly и GitHub API (или Hacker News API). Используйте полученную информацию для настройки стилевого оформления уже построенных диаграмм или загрузите другие данные и постройте собственные визуализации.

Итоги

В этой главе вы узнали, как использовать API для написания программ, автоматически собирающих необходимые данные и использующих полученную информацию для создания визуализации. Мы использовали GitHub API для получения информации о самых популярных проектах Python на GitHub, а также в общих чертах рассмотрели API Hacker News. Вы узнали, как с помощью пакета `Requests` автоматически выдать вызов API к GitHub и как обработать результаты этого вызова. Также были описаны некоторые средства конфигурации Plotly, позволяющие выполнить дополнительную настройку внешнего вида создаваемых диаграмм.

В последнем проекте мы используем Django для построения веб-приложения, которое станет нашим последним проектом в этой книге.

Проект 3
Веб-приложения

18 Знакомство с Django

Современные веб-сайты в действительности представляют собой многофункциональные приложения, достаточно близкие к полноценным приложениям для настольных систем. Python содержит мощный инструментарий для построения веб-приложений, который называется Django. Django представляет собой *веб-фреймворк* — набор средств, упрощающих построение интерактивных веб-сайтов. В этой главе вы научитесь использовать Django (http://djangoproject.com/) для построения проекта Learning Log — сетевой журнальной системы для отслеживания информации, полученной вами по определенной теме.

Мы напишем спецификацию для этого проекта, а затем определим модели для данных, с которыми будет работать приложение. Мы воспользуемся административной системой Django для ввода некоторых начальных данных, а затем научимся писать представления и шаблоны, на базе которых Django будет строить страницы нашего сайта.

Django может реагировать на запросы страниц, упрощает чтение и запись информации в базы данных, управление пользователями и многие другие операции. В главах 19 и 20 мы доработаем проект Learning Log, а затем развернем его на сервере, чтобы вы (и ваши друзья) могли использовать его.

Подготовка к созданию проекта

В начале работы над проектом необходимо описать проект в *спецификации*. Затем вы создадите виртуальную среду для построения проекта.

Написание спецификации

В полной спецификации описываются цели проекта, его функциональность, а также внешний вид и интерфейс пользователя. Как и любой хороший проект или бизнес-план, спецификация должна сосредоточиться на самых важных аспектах и обеспечивать планомерную разработку проекта. Здесь мы не будем писать полную спецификацию, а сформулируем несколько четких целей, которые будут задавать направление процесса разработки. Вот как выглядит спецификация:

Мы напишем веб-приложение с именем Learning Log, при помощи которого пользователь сможет вести журнал интересующих его тем и создавать записи в журнале во время изучения каждой темы. Домашняя страница Learning Log содержит описание сайта и приглашает пользователя зарегистрироваться либо ввести свои учетные данные. После успешного входа пользователь получает возможность создавать новые темы, добавлять новые записи, читать и редактировать существующие записи.

Во время изучения нового материала бывает полезно вести журнал того, что вы узнали, — записи пригодятся для контроля и возвращения к необходимой информации. Хорошее приложение повышает эффективность этого процесса.

Создание виртуальной среды

Для работы с Django необходимо сначала создать виртуальную среду для работы. *Виртуальная среда* представляет собой подраздел системы, в котором вы можете устанавливать пакеты в изоляции от всех остальных пакетов Python. Отделение библиотек одного проекта от других проектов принесет пользу при развертывании Learning Log на сервере в главе 20.

Создайте для проекта новый каталог с именем learning_log, перейдите в этот каталог в терминальном режиме и создайте виртуальную среду следующими командами:

```
learning_log$ python -m venv ll_env
learning_log$
```

Команда запускает модуль виртуальной среды `venv` и использует его для создания виртуальной среды с именем `ll_env` (обратите внимание: в имени `ll_env` две буквы `l`, а не одна). Если для запуска программ или установки пакетов используется другая команда (например, `python3`), подставьте ее на место `python`.

Активизация виртуальной среды

После того как виртуальная среда будет создана, ее необходимо активизировать следующей командой:

```
  learning_log$ source ll_env/bin/activate
❶ (ll_env)learning_log$
```

Команда запускает сценарий activate из каталога ll_env/bin. Когда среда активизируется, ее имя выводится в круглых скобках ❶; теперь вы можете устанавливать пакеты в среде и использовать те пакеты, что были установлены ранее. Пакеты, установленные в ll_env, будут доступны только в то время, пока среда остается активной.

ПРИМЕЧАНИЕ Если вы работаете в системе Windows, используйте команду `ll_env\Scripts\activate` (без слова `source`) для активизации виртуальной среды. Если вы используете PowerShell, слово Activate должно начинаться с буквы верхнего регистра.

Чтобы завершить использование виртуальной среды, введите команду `deactivate`:

```
(ll_env)learning_log$ deactivate
learning_log$
```

Среда также становится неактивной при закрытии терминального окна, в котором она работает.

Установка Django

После того как вы создали свою виртуальную среду и активизировали ее, установите Django:

```
(ll_env)learning_log$ pip install django
Collecting django
...
Installing collected packages: pytz, django
Successfully installed django-2.2.0 pytz-2018.9 sqlparse-0.2.4
(ll_env)learning_log$
```

Так как вы работаете в виртуальной среде, эта команда выглядит одинаково во всех системах. Использовать флаг `--user` не нужно, как и использовать более длинные команды вида `python -m pip install` *имя_пакета*.

Помните, что с Django можно работать только в то время, пока среда остается активной.

ПРИМЕЧАНИЕ Новая версия Django выходит приблизительно раз в восемь месяцев; возможно, при установке Django будет выведен новый номер версии. Скорее всего, проект будет работать в том виде, в каком он здесь приведен, даже в новых версиях Django. Если вы хотите использовать ту же версию Django, которая используется здесь, введите команду `pip install django==2.2.*`. Команда установит последний выпуск Django 2.2. Если у вас возникнут проблемы, связанные с версией, обращайтесь к электронным ресурсам книги по адресу https://nostarch.com/pythoncrashcourse2e/.

Создание проекта в Django

Не выходя из активной виртуальной среды (пока `ll_env` выводится в круглых скобках), введите следующие команды для создания нового проекта:

```
❶ (ll_env)learning_log$ django-admin.py startproject learning_log .
❷ (ll_env)learning_log$ ls
  learning_log ll_env manage.py
❸ (ll_env)learning_log$ ls learning_log
  __init__.py  settings.py  urls.py  wsgi.py
```

Команда ❶ приказывает Django создать новый проект с именем `learning_log`. Точка в конце команды создает новый проект со структурой каталогов, которая упрощает развертывание приложения на сервере после завершения разработки.

ПРИМЕЧАНИЕ Не забывайте про точку, иначе у вас могут возникнуть проблемы с конфигурацией при развертывании приложения. А если вы все же забыли, удалите созданные файлы и папки (кроме `ll_env`) и снова выполните команду.

Команда ls (dir в Windows) ❷ показывает, что Django создает новый каталог с именем learning_log. Также создается файл manage.py — короткая программа, которая получает команды и передает их соответствующей части Django для выполнения. Мы используем эти команды для управления такими задачами, как работа с базами данных и запуск серверов.

В каталоге learning_log находятся четыре файла ❸, важнейшими из которых являются файлы settings.py, urls.py и wsgi.py. Файл settings.py определяет то, как Django взаимодействует с вашей системой и управляет вашим проектом. Мы изменим некоторые из существующих настроек и добавим несколько новых настроек в ходе разработки проекта. Файл urls.py сообщает Django, какие страницы следует строить в ответ на запросы браузера. Файл wsgi.py помогает Django предоставлять созданные файлы (имя файла является сокращением от «Web Server Gateway Interface»).

Создание базы данных

Так как Django хранит большую часть информации в базе данных, относящейся к проекту, на следующем этапе необходимо создать базу данных, с которой Django сможет работать. Введите следующую команду (все еще не покидая активную среду):

```
  (ll_env)learning_log$ python manage.py migrate
❶ Operations to perform:
    Apply all migrations: admin, auth, contenttypes, sessions
  Running migrations:
    Applying contenttypes.0001_initial... OK
    Applying auth.0001_initial... OK
    ...
    Applying sessions.0001_initial... OK
❷ (ll_env)learning_log$ ls
  db.sqlite3  learning_log  ll_env  manage.py
```

Каждое изменение базы данных называется *миграцией*. Первое выполнение команды `migrate` приказывает Django проверить, что база данных соответствует текущему состоянию проекта. Когда мы впервые выполняем эту команду в новом проекте с использованием SQLite (вскоре мы расскажем о SQLite более подробно), Django создает новую базу данных за нас. В точке ❶ Django сообщает о создании и подготовке базы данных к хранению информации, необходимой для выполнения административных операций и аутентификации.

Выполнение команды ls показывает, что Django создает другой файл с именем db.sqlite3 ❷. SQLite — база данных, работающая с одним файлом; она идеально

подходит для написания простых приложений, потому что вам не нужно особенно следить за управлением базой данных.

ПРИМЕЧАНИЕ В активной виртуальной среде для выполнения команд manage.py используется команда python, даже если для запуска других программ вы используете другую команду (например, python3). В виртуальной среде команда python относится к версии Python, создавшей виртуальную среду.

Просмотр проекта

Убедимся в том, что проект был создан правильно. Введите команду `runserver` для просмотра текущего состояния проекта:

```
(ll_env)learning_log$ python manage.py runserver
Watchman unavailable: pywatchman not installed.
Watching for file changes with StatReloader
Performing system checks...

❶ System check identified no issues (0 silenced).
  February 18, 2019 - 16:26:07
❷ Django version 2.2.0, using settings 'learning_log.settings'
❸ Starting development server at http://127.0.0.1:8000/
  Quit the server with CONTROL-C.
```

Django запускает сервер, называемый *сервером разработки*, чтобы вы могли просмотреть проект в своей системе и проверить, как он работает. Когда вы запрашиваете страницу, вводя URL в браузере, сервер Django отвечает на запрос; для этого он строит соответствующую страницу и отправляет страницу браузеру.

В точке ❶ Django проверяет правильность созданного проекта; в точке ❷ выводится версия Django и имя используемого файла настроек; в точке ❸ возвращается URL-адрес, по которому доступен проект. URL http://127.0.0.1:8000/ означает, что проект ведет прослушивание запросов на порте 8000 локального хоста (localhost), то есть вашего компьютера. Термином «локальный хост» обозначается сервер, который обрабатывает только запросы вашей системы; он не позволяет никому другому просмотреть разрабатываемые страницы.

Теперь откройте браузер и введите URL http://localhost:8000/ — или http://127.0.0.1:8000/, если первый адрес не работает. Вы увидите нечто похожее на рис. 18.1 — страницу, которую создает Django, чтобы сообщить вам, что все пока работает правильно. Пока не завершайте работу сервера (но когда вы захотите прервать ее, это можно сделать нажатием клавиш Ctrl+C в терминале, в котором была введена команда `runserver`).

ПРИМЕЧАНИЕ Если вы получаете сообщение об ошибке «Порт уже используется», прикажите Django использовать другой порт; для этого введите команду `python manage.py runserver 8001` и продолжайте перебирать номера портов по возрастанию, пока не найдете открытый порт.

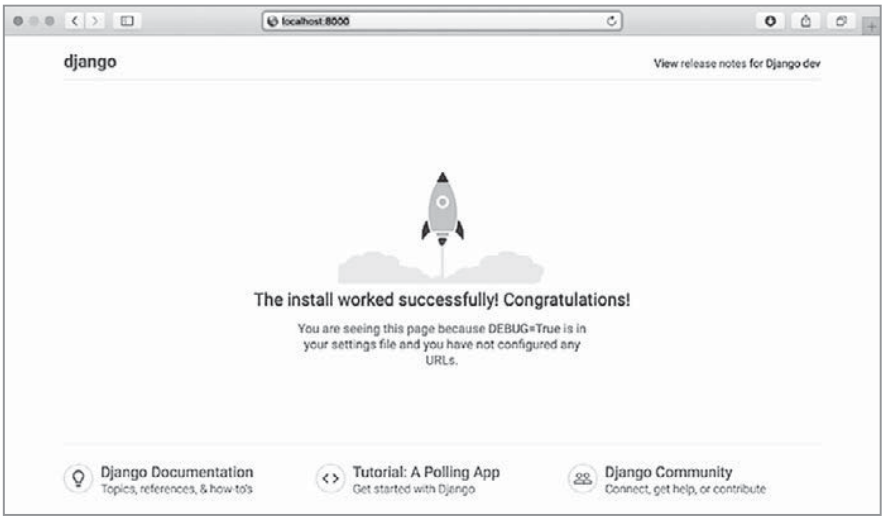

Рис. 18.1. Пока все работает правильно

УПРАЖНЕНИЯ

18.1. Новые проекты: чтобы лучше понять, что делает Django, постройте пару пустых проектов и посмотрите, что произойдет. Создайте новый каталог с простым именем типа `snap_gram` или `insta_chat` (за пределами каталога `learning_log`), перейдите в этот каталог в терминальном окне и создайте виртуальную среду. Установите Django и выполните команду `django-admin.py startproject snap_gram .` (обратите внимание на точку в конце команды).

Просмотрите файлы и каталоги, созданные командой, и сравните их с файлами и каталогами Learning Log. Проделайте это несколько раз, пока не начнете хорошо понимать, что именно делает Django при создании нового проекта, а затем удалите каталоги проектов.

Начало работы над приложением

Проект Django представляет собой группу отдельных *приложений*, совместная работа которых обеспечивает работу проекта в целом. Пока мы создадим одно приложение, которое будет выполнять большую часть работы в нашем проекте. Другое приложение для управления учетными записями пользователей будет добавлено в главе 19.

Оставьте сервер разработки выполняться в терминальном окне, открытом ранее. Откройте новое терминальное окно (или вкладку) и перейдите в каталог, содержащий manage.py. Активизируйте виртуальную среду и выполните команду `startapp`:

```
  learning_log$ source ll_env/bin/activate
  (ll_env)learning_log$ python manage.py startapp learning_logs
❶ (ll_env)learning_log$ ls
```

```
  db.sqlite3  learning_log  learning_logs  ll_env  manage.py
❷ (ll_env)learning_log$ ls learning_logs/
  admin.py  __init__.py  migrations  models.py  tests.py  views.py
```

Команда `startapp` *имя_приложения* приказывает Django создать инфраструктуру, необходимую для построения приложения. Заглянув сейчас в каталог проекта, вы найдете в нем новый подкаталог с именем learning_logs ❶. Откройте этот каталог, чтобы увидеть, какие файлы были созданы Django ❷. Самые важные файлы в этом каталоге — models.py, admin.py и views.py. Файл models.py будет использоваться для определения данных, которыми нужно управлять в нашем приложении. К файлам admin.py и views.py мы вернемся позднее.

Определение моделей

Подумаем, какие данные нам понадобятся. Каждый пользователь создает набор тем в своем журнале. Каждая запись, которую он сделает, будет привязана к определенной теме, а записи будут выводиться в текстовом виде. Также необходимо хранить временную метку каждой записи, чтобы пользователь знал, когда эта запись была создана.

Откройте файл models.py и просмотрите его текущее содержимое:

models.py

```
from django.db import models

# Создайте здесь свои модели.
```

Модуль с именем `models` импортируется автоматически, и нам предлагается создать свои модели. *Модель* сообщает Django, как работать с данными, которые будут храниться в приложении. С точки зрения кода модель представляет собой обычный класс; она содержит атрибуты и методы, как и все остальные классы, рассматривавшиеся нами ранее. Вот как выглядит модель тем обсуждения, которые будут сохраняться пользователями:

```
  from django.db import models

  class Topic(models.Model):
      """Тема, которую изучает пользователь"""
❶     text = models.CharField(max_length=200)
❷     date_added = models.DateTimeField(auto_now_add=True)

❸     def __str__(self):
          """Возвращает строковое представление модели."""
          return self.text
```

Мы создали класс с именем `Topic`, наследующий от `Model` — родительского класса, включенного в Django и определяющего базовую функциональность модели. В класс `Topic` добавляются два атрибута: `text` и `date_added`.

Атрибут `text` содержит данные `CharField` — блок данных, состоящий из символов, то есть текст ❶. Атрибуты `CharField` могут использоваться для хранения небольших объемов текста: имен, заголовков, названий городов и т. д. При определении атрибута `CharField` необходимо сообщить Django, сколько места нужно зарезервировать для него в базе данных. В данном случае задается максимальная длина `max_length`, равная 200 символам; этого должно быть достаточно для хранения большинства имен тем.

Атрибут `date_added` содержит данные `DateTimeField` — блок данных для хранения даты и времени ❷. Аргумент `auto_add_now=True` приказывает Django автоматически присвоить этому атрибуту текущую дату и время каждый раз, когда пользователь создает новую тему.

ПРИМЕЧАНИЕ Полный список всех полей, которые могут использоваться в модели, приведены в документе Django Model Field Reference на https://docs.djangoproject.com/en/2.2/ref/models/fields/. Возможно, вся эта информация вам сейчас не понадобится, но она будет в высшей степени полезной, когда вы начнете разрабатывать собственные приложения.

Необходимо сообщить Django, какой атрибут должен использоваться по умолчанию при вводе информации о теме. Django вызывает метод `__str__()` для вывода простого представления модели. Мы написали реализацию `__str__()`, которая возвращает строку, хранящуюся в атрибуте `text` ❸.

Активизация моделей

Чтобы использовать модели, необходимо приказать Django включить приложение в общий проект. Откройте файл settings.py (из каталога learning_log/learning_log) и найдите в нем раздел, который сообщает Django, какие приложения установлены в проекте:

settings.py

```
...
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)
...
```

Добавьте наше приложение в этот кортеж; измените содержимое `INSTALLED_APPS`, чтобы оно выглядело так:

```
...
INSTALLED_APPS = (
    # Мои приложения
```

```
    'learning_logs',

    # Приложения django по умолчанию.
    'django.contrib.admin',
    ...
]
...
```

Группировка приложений в проекте упрощает управление ими по мере того, как проект растет, а количество приложений увеличивается. Здесь мы создаем раздел, который пока содержит только приложение `learning_logs`. Очень важно разместить свои приложения перед приложениями по умолчанию на случай, если вам понадобится переопределить поведение таких приложений.

Затем необходимо приказать Django изменить базу данных для хранения информации, относящейся к модели `Topic`. В терминальном окне введите следующую команду:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
  learning_logs/migrations/0001_initial.py
    - Create model Topic
(ll_env)learning_log$
```

По команде `makemigrations` Django определяет, как изменить базу данных для хранения информации, связанной с новыми моделями. Из результатов видно, что Django создает файл миграции с именем 0001_initial.py. Эта миграция создает в базе данных таблицу для модели `Topic`.

Теперь применим миграцию для автоматического изменения базы данных:

```
  (ll_env)learning_log$ python manage.py migrate
  Operations to perform:
    Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
  Running migrations:
❶   Applying learning_logs.0001_initial... OK
```

Большая часть вывода этой команды совпадает с выводом, полученным при первом выполнении команды `migrate`. Обратить внимание следует на строку ❶; здесь Django подтверждает, что применение миграции для `learning_logs` прошло успешно.

Каждый раз, когда вы захотите изменить данные, которыми управляет Learning Log, выполните эти три действия: внесите изменения в models.py, вызовите makemigrations для `learning_logs` и прикажите Django выполнить миграцию проекта (`migrate`).

Административный сайт Django

Django позволяет легко работать с моделями, определенными для приложения, через *административный сайт*. Этот сайт используется администраторами сайта,

а не рядовыми пользователями. В этом разделе мы создадим административный сайт и используем его для добавления некоторых тем через модель `Topic`.

Создание суперпользователя

Django позволяет создать пользователя, обладающего полным набором привилегий на сайте; такой пользователь называется *суперпользователем*. *Привилегии* управляют действиями, которые разрешено выполнять пользователю. На самом жестком уровне привилегий пользователь может только читать общедоступную информацию на сайте. Зарегистрированным пользователям обычно предоставляется привилегия чтения своих приватных данных, а также избранной информации, доступной только для участников сообщества. Для эффективного администрирования веб-приложения владельцу сайта обычно должна быть доступна вся информация, хранящаяся на сайте. Хороший администратор внимательно относится к конфиденциальной информации пользователя, потому что пользователи доверяют тем приложениям, с которыми они работают.

Чтобы создать суперпользователя в Django, введите следующую команду и ответьте на запросы:

```
  (ll_env)learning_log$ python manage.py createsuperuser
❶ Username (leave blank to use 'eric'): ll_admin
❷ Email address:
❸ Password:
  Password (again):
  Superuser created successfully.
  (ll_env)learning_log$
```

При получении команды `createsuperuser` Django предлагает ввести имя пользователя, который является суперпользователем ❶. Здесь мы вводим имя `ll_admin`, но вы можете ввести любое имя на свое усмотрение. Также можно ввести адрес электронной почты или оставить это поле пустым ❷. После этого следует дважды ввести пароль ❸.

ПРИМЕЧАНИЕ Часть конфиденциальной информации может быть скрыта от администраторов сайта. Например, Django на самом деле не сохраняет введенный пароль; вместо этого сохраняется хеш — специальная строка, построенная на основе пароля. И когда в будущем вы вводите пароль, Django снова хеширует введенные данные и сравнивает результат с хранимым хешем. Если два хеша совпадают, то проверка пройдена. Если же хакер в результате атаки получит доступ к базе данных сайта, то он сможет прочитать только хранящийся в базе хеш, но не пароли. При правильной настройке сайта восстановить исходные пароли из хешей почти невозможно.

Регистрация модели на административном сайте

Django добавляет некоторые модели (например, `User` и `Group`) на административный сайт автоматически, но модели, которые мы создали, придется регистрировать вручную.

При запуске приложения `learning_logs` Django создает файл admin.py в одном каталоге с models.py. Откройте файл admin.py:

admin.py

```
from django.contrib import admin

# Зарегистрируйте здесь ваши модели.
```

Чтобы зарегистрировать `Topic` на административном сайте, введите следующую команду:

```
  from django.contrib import admin
❶ from .models import Topic

❷ admin.site.register(Topic)
```

Этот код импортирует регистрируемую модель `Topic` ❶. Точка перед `models` сообщает Django, что файл models.py следует искать в одном каталоге с admin.py. Вызов `admin.site.register()` сообщает Django, что управление моделью должно осуществляться через административный сайт ❷.

Теперь используйте учетную запись суперпользователя для входа на административный сайт. Введите адрес http://localhost:8000/admin/, затем имя пользователя и пароль для только что созданного суперпользователя, и вы увидите экран наподобие изображенного на рис. 18.2. На этой странице можно добавлять новых пользователей и группы, а также вносить изменения в уже существующие настройки. Также можно работать с данными, связанными с только что определенной моделью `Topic`.

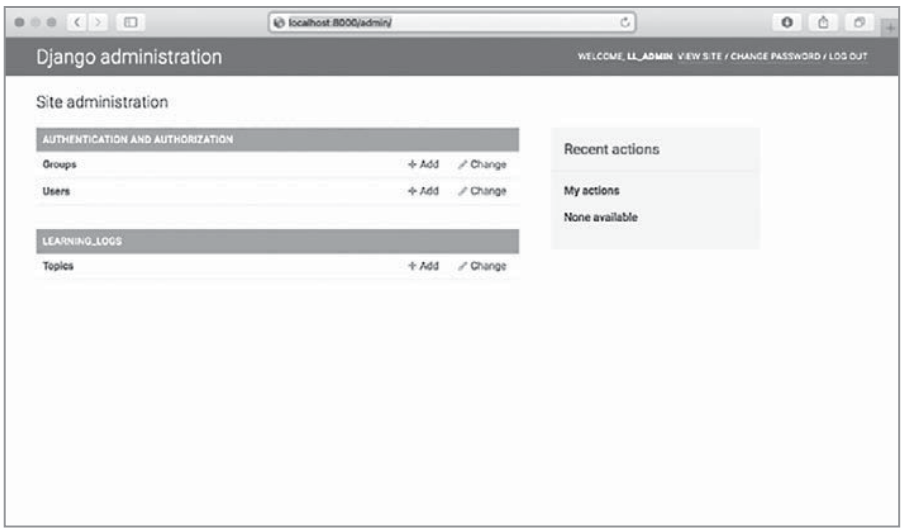

Рис. 18.2. Административный сайт с включением модели Topic

ПРИМЕЧАНИЕ Если в браузере появляется сообщение о недоступности веб-страницы, убедитесь в том, что сервер Django работает в терминальном окне. Если сервер не работает, активизируйте виртуальную среду и снова введите команду `python manage.py runserver`. Если у вас возникнут проблемы с просмотром проекта в любой момент в процессе разработки, закройте все открытые терминалы и снова введите команду runserver; это станет хорошим первым шагом в процессе диагностики.

Добавление тем

Когда модель `Topic` была зарегистрирована на административном сайте, добавим первую тему. Щелкните на ссылке Topics, чтобы перейти к странице Topics; страница практически пуста, потому что еще нет ни одной темы для выполнения операций. Щелкните на ссылке Add Topic; открывается форма для добавления новой темы. Введите в первом поле текст Chess и щелкните на ссылке Save. Вы возвращаетесь к административной странице Topics, на которой появляется только что созданная тема.

Создадим вторую тему, чтобы у вас было больше данных для работы. Снова щелкните на ссылке Add Topic и создайте вторую тему Rock Climbing. Ссылка Save снова возвращает вас к основной странице Topics, где отображаются обе темы, Chess и Rock Climbing.

Определение модели Entry

Чтобы сохранить информацию о том, что вы узнали по этим двум темам, необходимо определить модель для записей, которые пользователь делает в своих журналах. Каждая запись должна ассоциироваться с конкретной темой. Такое отношение называется *отношением «многие к одному»*, поскольку многие записи могут быть связаны с одной темой.

Код модели `Entry` (из файла models.py) выглядит так:

models.py

```
  from django.db import models

  class Topic(models.Model):
  ...

❶ class Entry(models.Model):
      """Информация, изученная пользователем по теме"""
❷     topic = models.ForeignKey(Topic, on_delete=models.CASCADE)
❸     text = models.TextField()
      date_added = models.DateTimeField(auto_now_add=True)

❹     class Meta:
          verbose_name_plural = 'entries'

      def __str__(self):
          """Возвращает строковое представление модели."""
❺         return f"{self.text[:50]}..."
```

Класс `Entry` наследует от базового класса `Model`, как и рассмотренный ранее класс `Topic` ❶. Первый атрибут, `topic`, является экземпляром `ForeignKey` ❷. Термин *внешний ключ* (foreign key) происходит из теории баз данных; внешний ключ содержит ссылку на другую запись в базе данных. Таким образом каждая запись связывается с конкретной темой. Каждой теме при создании присваивается ключ, или идентификатор. Если потребуется установить связь между двумя записями данных, Django использует ключ, связанный с каждым блоком информации. Вскоре мы используем такие связи для получения всех записей, связанных с заданной темой. Аргумент `on_delete=models.CASCADE` сообщает Django, что при удалении темы все записи, связанные с этой темой, также должны быть удалены (это называется *каскадным удалением*).

Затем идет атрибут с именем `text`, который является экземпляром `TextField` ❸. Полю такого типа ограничение размера не требуется, потому что размер отдельных записей не ограничивается. Атрибут `date_added` позволяет отображать записи в порядке их создания и снабжать каждую запись временной меткой.

В точке ❹ класс `Meta` вкладывается в класс `Entry`. Класс `Meta` хранит дополнительную информацию по управлению моделью; в данном случае он позволяет задать специальный атрибут, который приказывает Django использовать форму множественного числа `Entries` при обращении более чем к одной записи. (Без этого Django будет использовать неправильную форму `Entrys`.)

Метод `__str__()` сообщает Django, какая информация должна отображаться при обращении к отдельным записям. Так как запись может быть достаточно длинным блоком текста, мы приказываем Django выводить только первые 50 символов ❺. Также добавляется многоточие — признак вывода неполного текста.

Миграция модели Entry

Так как мы добавили новую модель, миграцию базы данных необходимо провести снова. Вскоре вы привыкнете к этому процессу: вы изменяете models.py, выполняете команду `python manage.py makemigrations` *имя_приложения*, а затем команду `python manage.py migrate`.

Проведите миграцию базы данных и проверьте вывод:

```
  (ll_env)learning_log$ python manage.py makemigrations learning_logs
  Migrations for 'learning_logs':
❶   learning_logs/migrations/0002_entry.py
      - Create model Entry
  (ll_env)learning_log$ python manage.py migrate
  Operations to perform:
    ...
❷   Applying learning_logs.0002_entry... OK
```

Команда генерирует новую миграцию с именем 0002_entry.py, которая сообщает Django, как изменить базу данных для хранения информации, связанной с моделью

Entry ❶. При выдаче команды `migrate` Django подтверждает, что применение миграции прошло успешно ❷.

Регистрация Entry на административном сайте

Модель `Entry` тоже необходимо зарегистрировать. Файл admin.py должен выглядеть так:

admin.py

```
from django.contrib import admin

from .models import Topic, Entry

admin.site.register(Topic)
admin.site.register(Entry)
```

Вернитесь на страницу http://localhost/admin/, и вы увидите раздел `Entries` в категории `learning_logs`. Щелкните на ссылке Add для Entries или щелкните на Entries и выберите вариант Add. На экране появляется раскрывающийся список для выбора темы, для которой создается запись, и текстовое поле для ввода записи. Выберите в раскрывающемся списке вариант Chess и добавьте запись.

При выборе ссылки Save вы вернетесь к основной административной странице. Здесь проявляются преимущества использования формата `text[:50]` в качестве строкового представления каждой записи; работать с несколькими записями в административном интерфейсе намного удобнее, если вы видите только часть записи вместо ее полного текста.

Создайте вторую запись для темы Chess и одну запись для темы Rock Climbing, чтобы у нас были исходные данные для дальнейшей разработки Learning Log.

Интерактивная оболочка Django

Введенные данные можно проанализировать на программном уровне в интерактивном терминальном сеансе. Эта интерактивная среда, называемая *оболочкой* (shell) Django, прекрасно подходит для тестирования и диагностики проекта. Пример сеанса в интерактивной оболочке:

```
  (ll_env)learning_log$ python manage.py shell
❶ >>> from learning_logs.models import Topic
  >>> Topic.objects.all()
  <QuerySet [<Topic: Chess>, <Topic: Rock Climbing>]>
```

Команда `python manage.py shell` (выполняемая в активной виртуальной среде) запускает интерпретатор Python, который может использоваться для работы с информацией в базе данных проекта. В данном случае мы импортируем модель `Topic` из модуля `learning_logs.models` ❶. Затем метод `Topic.objects.all()` используется

для получения всех экземпляров модели `Topic`; возвращаемый список называется *итоговым набором* (queryset).

Содержимое итогового набора перебирается точно так же, как и содержимое списка. Например, просмотр идентификаторов, назначенных каждому объекту темы, выполняется так:

```
>>> topics = Topic.objects.all()
>>> for topic in topics:
...   print(topic.id, topic)
...
1 Chess
2 Rock Climbing
```

Итоговый набор сохраняется в `topics`, после чего выводится атрибут `id` каждого объекта `topic` и его строковое представление. Мы видим, что теме `Chess` присвоен идентификатор 1, а `Rock Climbing` присвоен идентификатор 2.

Зная идентификатор конкретного объекта, можно при помощи метода `Topic.objects.get()` получить этот объект и проанализировать содержащиеся в нем атрибуты. Просмотрим значения `text` и `date_added` для темы `Chess`:

```
>>> t = Topic.objects.get(id=1)
>>> t.text
'Chess'
>>> t.date_added
datetime.datetime(2019, 2, 19, 1, 55, 31, 98500, tzinfo=<UTC>)
```

Также можно просмотреть записи, относящиеся к конкретной теме. Ранее мы определили атрибут `topic` для модели `Entry`. Он был экземпляром `ForeighKey`, представляющим связь между записью и темой. Django может использовать эту связь для получения всех записей, относящихся к некоторой теме:

```
❶ >>> t.entry_set.all()
  <QuerySet [<Entry: The opening is the first part of the game, roughly...>,
  <Entry:
  In the opening phase of the game, it's important t...>]>
```

Чтобы получить данные через отношение внешнего ключа, воспользуйтесь именем связанной модели, записанным в нижнем регистре, за которым следует символ подчеркивания и слово `set` ❶. Допустим, у вас имеются модели `Pizza` и `Topping` и модель `Topping` связана с `Pizza` через внешний ключ. Если ваш объект называется `my_pizza`, для получения всех связанных с ним экземпляров `Topping` используется выражение `my_pizza.topping_set.all()`.

Мы будем использовать такой синтаксис при переходе к программированию страниц, которые могут запрашиваться пользователями. Оболочка очень удобна тогда, когда вы хотите проверить, что ваш код получает нужные данные. Если в оболочке код работает так, как задумано, можно ожидать, что он будет правильно работать и в файлах, которые вы создаете в своем проекте. Если код выдает ошибки или не

загружает те данные, которые должен загружать, вам будет намного проще отладить его в простой оболочке, чем при работе с файлами, генерирующими веб-страницы. В книге мы не будем часто возвращаться к оболочке, но вам не стоит забывать о ней — это полезный инструмент, который поможет вам освоить синтаксис Django для работы с данными проекта.

ПРИМЕЧАНИЕ При каждом изменении модели необходимо перезапустить оболочку, чтобы увидеть результаты этих изменений. Чтобы завершить сеанс работы с оболочкой, нажмите Ctrl+D; в Windows нажмите Ctrl+Z, а затем Enter.

УПРАЖНЕНИЯ

18.2. Короткие записи: метод `__str__()` в модели Entry в настоящее время присоединяет многоточие к каждому экземпляру Entry, отображаемому Django на административном сайте или в оболочке. Добавьте в метод `__str__()` команду `if`, добавляющую многоточие только для записей, длина которых превышает 50 символов. Воспользуйтесь административным сайтом, чтобы ввести запись длиной менее 50 символов, и убедитесь в том, что при ее просмотре многоточие не отображается.

18.3. Django API: при написании кода для работы с данными проекта вы создаете запрос. Просмотрите документацию по созданию запросов к данным по адресу https://docs.djangoproject.com/en/2.2/topics/db/queries/. Многое из того, что вы увидите, покажется вам новым, но эта информация пригодится, когда вы начнете работать над собственными проектами.

18.4. Пиццерия: создайте новый проект с именем `pizzeria`, содержащий приложение `pizzas`. Определите модель `Pizza` с полем `name`, в котором хранятся названия видов пиццы (например, «Гавайская» или «Четыре сыра»). Определите модель `Topping` с полями `pizza` и `name`. Поле `pizza` должно содержать внешний ключ к модели `Pizza`, а поле name должно позволять хранить такие значения, как «ананас» или «грибы».

Зарегистрируйте обе модели на административном сайте. Используйте сайт для ввода названий пиццы и топпингов. Изучите введенные данные в интерактивной оболочке.

Создание страниц: домашняя страница Learning Log

Обычно процесс создания веб-страниц в Django состоит из трех стадий: определения URL, написания представлений и написания шаблонов. Сначала следует определить *схемы* (patterns) URL. Схема URL описывает структуру URL-адреса и сообщает Django, на какие компоненты следует обращать внимание при сопоставлении запроса браузера с URL-адресом на сайте, чтобы выбрать возвращаемую страницу.

Затем каждый URL-адрес связывается с конкретным *представлением* — функция представление читает и обрабатывает данные, необходимые странице. Функция представления часто вызывает *шаблон*, который строит страницу, подходящую для передачи браузеру. Чтобы вы лучше поняли, как работает этот механизм, создадим домашнюю страницу для приложения Learning Log. Мы определим URL-адрес домашней страницы, напишем для него функцию представления и создадим простой шаблон.

Так как мы сейчас всего лишь убеждаемся в том, что Learning Log работает как положено, страница пока останется простой. Когда приложение будет завершено, вы можете заниматься его оформлением сколько душе угодно; приложение, которое хорошо выглядит, но не работает, бессмысленно. Пока на домашней странице будет отображаться только заголовок и краткое описание.

Сопоставление URL

Пользователь запрашивает страницы, вводя URL-адреса в браузере и щелкая на ссылках, поэтому мы должны решить, какие URL-адреса понадобятся в нашем проекте. Начнем с URL домашней страницы: это базовый адрес, используемый для обращения к проекту. На данный момент базовый URL-адрес http://localhost:8000/ возвращает сайт, сгенерированный Django по умолчанию; он сообщает о том, что проект был создан успешно. Мы изменим домашнюю страницу, связав базовый URL-адрес с домашней страницей Learning Log.

В каталоге проекта `learning_log` откройте файл urls.py. Вы увидите в нем следующий код:

urls.py

```
❶ from django.contrib import admin
  from django.urls import path

❷ urlpatterns = [
❸     path('admin/', admin.site.urls),
  ]
```

Первые две строки импортируют функции и модули, управляющие URL-адресами проекта и административным сайтом ❶. В теле файла определяется переменная `urlpatterns` ❷. В файле urls.py, представляющем проект в целом, переменная `urlpatterns` включает наборы URL-адресов из приложений в проект. Код ❸ включает модуль `admin.site.urls`, определяющий все URL-адреса, которые могут запрашиваться с административного сайта.

Добавим в этот файл URL-адреса `learning_logs`:

```
  from django.contrib import admin
  from django.urls import path, include

  urlpatterns = [
      path('admin/', admin.site.urls),
❶     path('', include('learning_logs.urls')),
  ]
```

В точке ❶ добавляется строка включения модуля `learning_logs.urls`.

Файл urls.py по умолчанию находится в каталоге learning_log; теперь нужно создать второй файл urls.py в папке learning_logs. Создайте новый файл Python, сохраните его под именем urls.py в learning_logs и включите в него следующий код:

urls.py

```
❶ """Определяет схемы URL для learning_logs."""

❷ from django.urls import path

❸ from . import views

❹ app_name = 'learning_logs'
❺ urlpatterns = [
      # Домашняя страница
❻     path('', views.index, name='index'),
  ]
```

Чтобы было понятно, с какой версией urls.py мы работаем, в начало файла добавляется строка документации ❶. Затем импортируется функция `path`, она необходима для связывания URL с представлениями ❷. Также импортируется модуль `views` ❸; точка приказывает Python импортировать представления из каталога, в котором находится текущий модуль urls.py. Переменная `app_name` помогает Django отличить этот файл urls.py от одноименных файлов в других приложениях в проекте ❹. Переменная `urlpatterns` в этом модуле представляет собой список страниц, которые могут запрашиваться из приложения `learning_logs` ❺.

Схема URL представляет собой вызов функции `path()` с тремя аргументами ❻. Первый аргумент содержит строку, которая помогает Django правильно маршрутизировать текущий запрос. Django получает запрашиваемый URL и пытается отобразить его на представление. Для этого Django ищет среди всех определенных схем URL ту, которая соответствует текущему запросу. Базовый URL-адрес проекта (http://localhost:8000/) игнорируется, так что пустая строка совпадает с базовым URL-адресом. Любой другой URL-адрес не будет соответствовать этому выражению, и Django вернет страницу с ошибкой, если запрашиваемый URL не соответствует ни одной из существующих схем URL.

Второй аргумент `path()` ❻ определяет вызываемую функцию из views.py. Когда запрашиваемый URL-адрес соответствует регулярному выражению, Django вызывает `index()` из views.py (мы напишем эту функцию представления в следующем разделе). Третий аргумент определяет имя `index` для этой схемы URL, чтобы на нее можно было ссылаться в других частях кода. Каждый раз, когда потребуется предоставить ссылку на домашнюю страницу, мы будем использовать это имя вместо URL.

Написание представления

Функция представления получает информацию из запроса, подготавливает данные, необходимые для построения страницы, и возвращает данные браузеру — часто с использованием шаблона, определяющего внешний вид страницы.

Файл views.py в `learning_logs` был сгенерирован автоматически при выполнении команды `python manage.py startapp`. На данный момент его содержимое выглядит так:

views.py

```
from django.shortcuts import render

# Создайте здесь свои представления.
```

Сейчас файл только импортирует функцию `render()`, которая генерирует ответ на основании данных, полученных от представлений. Откройте файл представления и добавьте следующий код домашней страницы:

```
from django.shortcuts import render

def index(request):
    """Домашняя страница приложения Learning Log"""
    return render(request, 'learning_logs/index.html')
```

Если URL запроса совпадает с только что определенной схемой, Django ищет в файле views.py функцию с именем `index()`, после чего передает этой функции представления объект `request`. В нашем случае никакая обработка данных для страницы не нужна, поэтому код функции сводится к вызову `render()`. Функция `render()` использует два аргумента — исходный объект запроса и шаблон, используемый для построения страницы. Давайте напишем этот шаблон.

Написание шаблона

Шаблон определяет общий внешний вид страницы, а Django заполняет его соответствующими данными при каждом запросе страницы. Шаблон может обращаться к любым данным, полученным от представления. Так как наше представление домашней страницы никаких данных не предоставляет, шаблон получается относительно простым.

В каталоге `learning_logs` создайте новый каталог с именем templates. В каталоге templates создайте другой каталог с именем learning_logs. На первый взгляд такая структура кажется избыточной (каталог learning_logs в каталоге templates внутри каталога learning_logs), но созданная таким образом структура будет однозначно интерпретироваться Django даже в контексте большого проекта, состоящего из множества отдельных приложений. Во внутреннем каталоге learning_logs создайте новый файл с именем index.html (таким образом, полное имя файла имеет вид learning_log/learning_logs/templates/learning_logs/index.html). Включите в него следующий текст:

index.html

```
<p>Learning Log</p>

<p>Learning Log helps you keep track of your learning, for any topic you're
learning about.</p>
```

Это очень простой файл. Если вы не знакомы с синтаксисом HTML, теги `<p></p>` обозначают абзацы. Тег `<p>` открывает абзац, а тег `</p>` закрывает его. В нашей стра-

нице два абзаца: первый содержит заголовок, а второй описывает, что пользователь может сделать с помощью приложения Learning Log.

Теперь при запросе базового URL-адреса проекта http://localhost:8000/ вы увидите только что построенную страницу вместо страницы по умолчанию. Django берет запрошенный URL-адрес и видит, что он совпадает со схемой `''`; в этом случае Django вызывает функцию `views.index()`, что приводит к построению страницы с использованием шаблона, содержащегося в index.html. Полученная страница показана на рис. 18.3.

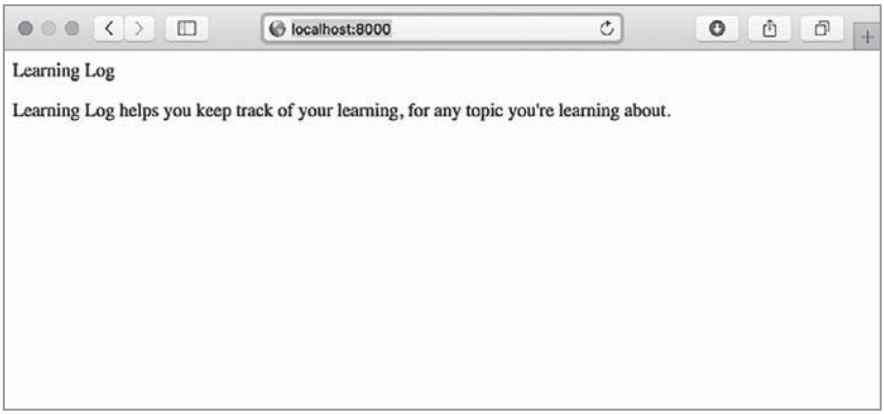

Рис. 18.3. Домашняя страница Learning Log

И хотя может показаться, что для одной страницы этот процесс слишком сложен, такое разделение URL-адресов, представлений и шаблонов работает хорошо. Оно позволяет сосредоточиться на отдельных аспектах проекта, а в более крупных проектах отдельные участники могут сосредоточиться на тех областях, в которых они наиболее сильны. Например, специалист по базам данных может заняться моделями, программист — кодом представления, а веб-дизайнер — шаблонами.

ПРИМЕЧАНИЕ Вы можете получить следующее сообщение об ошибке:

```
ModuleNotFoundError: No module named 'learning_logs.urls'
```

В таком случае остановите сервер разработки нажатием клавиш Ctrl+C в терминальном окне, в котором была введена команда runserver. Затем снова введите команду `python manage.py runserver`. Каждый раз, когда вы сталкиваетесь с подобными ошибками, попробуйте остановить и перезапустить сервер.

УПРАЖНЕНИЯ

18.5. План питания: представьте приложение для составления плана питания на неделю. Создайте новый каталог с именем `meal_planner`, а в этом каталоге — новый проект Django. Создайте новое приложение с именем `meal_plans`. Постройте простую домашнюю страницу для этого проекта.

18.6. Домашняя страница Pizzeria: добавьте домашнюю страницу в проект `Pizzeria`, который вы начали строить в упражнении 18.4 (с. 408).

Построение других страниц

Теперь, когда вы начали представлять процесс построения страниц, можно переходить к построению проекта Learning Log. Мы создадим две страницы для вывода данных: на одной будет выводиться список всех тем, а на другой — все записи по конкретной теме. Для каждой страницы мы создадим схему URL, напишем функцию представления и создадим шаблон. Но прежде чем переходить к работе, стоит создать базовый шаблон, от которого будут наследовать все шаблоны этого проекта.

Наследование шаблонов

При построении сайта некоторые элементы почти всегда повторяются на каждой странице. Вместо того чтобы встраивать эти элементы непосредственно в страницы, вы можете написать базовый шаблон с повторяющимися элементами; все страницы будут наследовать от этого шаблона. Такое решение позволит сосредоточиться на разработке уникальных аспектов каждой страницы и существенно упростит изменение общего оформления проекта в целом.

Родительский шаблон

Начнем с создания шаблона base.html в одном каталоге с файлом index.html. Этот файл будет содержать элементы, общие для всех страниц; все остальные шаблоны наследуют от base.html. Пока единственным элементом, который должен повторяться на каждой странице, остается заголовок в верхней части страницы. Так как шаблон будет включаться в каждую страницу, преобразуем заголовок в ссылку на домашнюю страницу:

```
  base.html
  <p>
❶   <a href="{% url 'learning_logs:index' %}">Learning Log</a>
  </p>

❷ {% block content %}{% endblock content %}
```

Первая часть файла создает абзац с именем проекта, который также работает как ссылка на домашнюю страницу. Для построения ссылки использовался *шаблонный тег*, обозначенный фигурными скобками и знаками процента `{% %}`. Шаблонный тег представляет собой блок кода, который генерирует информацию для вывода на странице. В данном примере шаблонный тег `{% url 'learning_logs:index' %}` генерирует URL-адрес, соответствующий схеме URL, определенной в файле learning_logs/urls.py с именем `'index'` ❶. В данном примере `learning_logs` — пространство имен, а `index` — схема URL с уникальным именем в этом пространстве имен.

Пространство имен определяется значением, присвоенным `app_name` в файле learning_logs/urls.py.

В этой простой странице HTML ссылка заключается в *якорный* тег:

```
<a href="url_ссылки">текст ссылки</a>
```

Генерирование URL-адреса шаблонным тегом существенно упрощает актуализацию ссылок. Чтобы изменить URL-адрес в проекте, достаточно изменить схему URL в urls.py, а Django автоматически вставит обновленный URL-адрес при следующем запросе страницы. Каждая страница в проекте будет наследовать от base.html, так что в дальнейшем на каждой странице будет содержаться ссылка на домашнюю страницу.

В точке ❷ вставляется пара тегов `block`. Блок с именем `content` резервирует место; информация, попадающая в блок `content`, будет определяться дочерним шаблоном.

Дочерний шаблон не обязан определять каждый блок в своем родителе, так что в родительских шаблонах можно зарезервировать место для любого количества блоков, а дочерний шаблон будет использовать столько из них, сколько потребуется.

ПРИМЕЧАНИЕ В коде Python почти всегда используются отступы в четыре пробела. Файлы шаблонов обычно имеют больший уровень вложенности, чем файлы Python, поэтому каждый уровень отступа обычно обозначается двумя пробелами. Будьте внимательны и действуйте последовательно.

Дочерний шаблон

Теперь нужно переписать файл index.html так, чтобы он наследовал от base.html. Обновленный файл index.html выглядит так:

index.html

```
❶ {% extends "learning_logs/base.html" %}

❷ {% block content %}
    <p>Learning Log helps you keep track of your learning, for any topic you're
    learning about.</p>
❸ {% endblock content %}
```

Сравнивая этот файл с исходной версией index.html, мы видим, что заголовок Learning Log заменен кодом наследования от родительского шаблона ❶. В первой строке дочернего шаблона должен находиться тег `{% extends %}`, который сообщает Django, от какого родительского шаблона он наследует. Файл base.html является частью `learning_logs`, поэтому `learning_logs` включается в путь к родительскому шаблону. Эта строка извлекает все содержимое из шаблона base.html и позволяет index.html определить, что должно попасть в пространство, зарезервированное блоком `content`.

Блок `content` определяется в точке ❷ вставкой тега `{% block %}` с именем `content`. Все, что не наследуется от родительского шаблона, попадает в блок `content`. В данном случае это абзац с описанием проекта Learning Log. В точке ❸ мы сообщаем о том, что определение `content` завершено, при помощи тега `{% endblock content %}`. Наличие имени у `{% endblock %}` не обязательно, но если шаблон увеличится и станет включать несколько блоков, будет полезно сразу видеть, какой именно блок завершается.

Вероятно, вы уже начинаете понимать преимущества наследования шаблонов: в дочерний шаблон достаточно включить информацию, уникальную для этой страницы. Такой подход не только упрощает каждый шаблон, но и значительно упрощает изменение сайта. Чтобы изменить элемент, общий для многих страниц, достаточно изменить элемент в родительском шаблоне. Внесенные изменения будут автоматически перенесены на каждую страницу, наследующую от этого шаблона. В проекте из десятков и сотен страниц такая структура значительно упрощает и ускоряет доработку сайта.

ПРИМЕЧАНИЕ В больших проектах часто создается один родительский шаблон base.html для всего сайта и родительские шаблоны для каждого крупного раздела сайта. Все шаблоны разделов наследуют от base.html, и каждая страница сайта наследует от шаблона раздела. При такой структуре вы сможете легко изменять оформление и поведение сайта в целом, любого его раздела или отдельной страницы. Данная конфигурация сильно повышает эффективность работы и стимулирует разработчика к дальнейшему совершенствованию сайта.

Страница со списком тем

Разобравшись с тем, как эффективно организовать построение страниц, мы можем сосредоточиться на следующих двух страницах: списке всех тем и списке записей по одной теме. На странице тем выводится перечень всех тем, созданных пользователями, и это первая страница, на которой нам придется работать с данными.

Схема URL для тем

Сначала нужно определить URL для страницы тем. Обычно в таких случаях выбирается простой фрагмент URL, который отражает суть информации, представленной на странице. Мы воспользуемся словом *topics*, так что для получения страницы будет использоваться URL http://localhost:8000/topics/. А вот какие изменения следует внести в learning_logs/urls.py:

urls.py

```
  """Определяет схемы URL для learning_logs."""
  ...
  urlpatterns = [
      # Домашняя страница
      path('', views.index, name='index'),
      # Страница со списком всех тем.
❶     path('topics/', views.topics, name='topics'),
  ]
```

Мы просто добавили `topics/` в аргумент регулярного выражения, используемый с URL-адресом домашней страницы ❶. Когда Django проверяет запрашиваемый URL-адрес, эта схема совпадет с любым URL-адресом, который состоит из базового URL-адреса и слова `topics`. Слеш в конце можно включить, а можно не включать, но после слова `topics` ничего быть не должно, иначе схема не совпадет. Любой запрос с URL-адресом, соответствующим этой схеме, будет передан функции `topics()` в views.py.

Представление topics

Функция `topics()` должна получать данные из базы данных и отправлять их шаблону. Обновленная версия views.py выглядит так:

views.py

```
  from django.shortcuts import render
❶ from .models import Topic

  def index(request):
  ...

❷ def topics(request):
      """Выводит список тем."""
❸     topics = Topic.objects.order_by('date_added')
❹     context = {'topics': topics}
❹     return render(request, 'learning_logs/topics.html', context)
```

Сначала импортируется модель, связанная с нужными данными ❶. Функции `topics()` необходим один параметр: объект `request`, полученный Django от сервера ❷. В точке ❸ выдается запрос к базе данных на получение объектов `Topic`, отсортированных по атрибуту `date_added`. Полученный итоговый набор сохраняется в `topics`.

В точке ❹ определяется контекст, который будет передаваться шаблону. *Контекст* представляет собой словарь, в котором ключами являются имена, используемые в шаблоне для обращения к данным, а значениями — данные, которые должны передаваться шаблону. В данном случае существует всего одна пара «ключ-значение», которая содержит набор тем, отображаемых на странице. При построении страницы, использующей данные, функции `render()` передается переменная `context`, а также объект `request` и путь к шаблону ❺.

Шаблон topics

Шаблон страницы со списком тем получает словарь `context`, чтобы шаблон мог использовать данные, предоставленные `topics()`. Создайте файл с именем topics.html в одном каталоге с index.html. Вывод списка тем в шаблоне осуществляется следующим образом:

topics.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p>Topics</p>

❶ <ul>
❷   {% for topic in topics %}
❸     <li>{{ topic }}</li>
❹   {% empty %}
      <li>No topics have been added yet.</li>
❺   {% endfor %}
❻ </ul>

{% endblock content %}
```

Сначала тег `{% extends %}` объявляет о наследовании от base.html, как и в случае с шаблоном `index`, после чего открывается блок `content`. Тело страницы содержит маркированный (bulleted) список введенных тем. В стандартном языке HTML маркированный список называется *неупорядоченным списком* и обозначается тегами `<ul></ul>`. Список тем начинается в точке ❶.

В точке ❷ находится другой шаблонный тег, эквивалентный циклу `for`, для перебора списка `topics` из словаря `context`. Код, используемый в шаблоне, отличается от Python некоторыми важными особенностями. Python использует отступы для обозначения строк, входящих в тело цикла. В шаблоне каждый цикл `for` должен снабжаться явным тегом `{% endfor %}`, обозначающим конец цикла. Таким образом, в шаблонах часто встречаются циклы следующего вида:

```
{% for элемент in список %}
    действия для каждого элемента
{% endfor %}
```

В цикле каждая тема должна быть преобразована в элемент маркированного списка. Чтобы вывести значение переменной в шаблоне, заключите ее имя в двойные фигурные скобки. Фигурные скобки на странице не появятся; они всего лишь сообщают Django об использовании шаблонной переменной. Код `{{ topic }}` в точке ❸ будет заменен значением `topic` при каждом проходе цикла. Тег HTML `<li></li>` обозначает *элемент списка*. Все, что находится между тегами, в паре тегов `<ul></ul>`, будет отображаться как элемент маркированного списка.

В точке ❹ находится шаблонный тег `{% empty %}`, который сообщает Django, что делать при отсутствии элементов в списке. В нашем примере выводится сообщение о том, что темы еще не созданы. Последние две строки завершают цикл `for` ❺ и маркированный список ❻.

Затем необходимо изменить базовый шаблон и включить ссылку на страницу с темами. Добавьте следующий код в base.html:

base.html

```
    <p>
❶     <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
❷     <a href="{% url 'learning_logs:topics' %}">Topics</a>
    </p>

    {% block content %}{% endblock content %}
```

После ссылки на домашнюю страницу ❶ добавляется дефис, после которого вставляется ссылка на страницу тем, которая также представлена шаблонным тегом `{% url %}` ❷. Эта строка приказывает Django сгенерировать ссылку, соответствующую схеме URL с именем `'topics'` в learning_logs/urls.py.

Обновив домашнюю страницу в браузере, вы увидите ссылку Topics. Щелчок на этой ссылке открывает страницу, похожую на рис. 18.4.

Рис. 18.4. Страница со списком тем

Страницы отдельных тем

Следующим шагом станет создание страницы для вывода информации по одной теме, с названием темы и всеми записями по этой теме. Мы снова определим новую схему URL, напишем представление и создадим шаблон. Кроме того, на странице со списком тем каждый элемент маркированного списка будет преобразован в ссылку на соответствующую страницу отдельной темы.

Схема URL для отдельных тем

Схема URL для страницы отдельной темы немного отличается от других схем URL, которые встречались нам ранее, потому что в ней используется атрибут `id` темы

для обозначения запрашиваемой темы. Например, если пользователь хочет просмотреть страницу с подробной информацией по теме `Chess (id=1)`, эта страница будет иметь URL-адрес http://localhost:8000/topics/1/. Вот как выглядит схема для этого URL-адреса из learning_logs/urls.py:

urls.py

```
...
urlpatterns = [
    ...
    # Страница с подробной информацией по отдельной теме
    path('topics/<int:topic_id>/', views.topic, name='topic'),
]
```

Рассмотрим строку `'topics/<int:topic_id>/'` в этой схеме URL. Первая часть строки сообщает Django, что искать следует URL-адреса, у которых за базовым адресом идет слово *topics*. Вторая часть строки, `/<int:topic_id>/`, описывает целое число, заключенное между двумя слешами; это целое число сохраняется в аргументе `topic_id`.

Когда Django находит URL-адрес, соответствующий этой схеме, вызывается функция представления `topic()`, в аргументе которой передается значение, хранящееся в `topic_id`. Значение `topic_id` используется для получения нужной темы внутри функции.

Представление отдельной темы

Функция `topic()` должна получить тему и все связанные с ней записи из базы данных:

views.py

```
  ...
❶ def topic(request, topic_id):
      """Выводит одну тему и все ее записи."""
❷     topic = Topic.objects.get(id=topic_id)
❸     entries = topic.entry_set.order_by('-date_added')
❹     context = {'topic': topic, 'entries': entries}
❺     return render(request, 'learning_logs/topic.html', context)
```

Это первая функция представления, которой требуется параметр, отличный от объекта `request`. Функция получает значение, совпавшее с выражением `/<int:topic_id>/`, и сохраняет его в `topic_id` ❶. В точке ❷ функция `get()` используется для получения темы (по аналогии с тем, как мы это делали в оболочке Django). В точке ❸ загружаются записи, связанные с данной темой, и они упорядочиваются по значению `date_added`: знак «минус» перед `date_added` сортирует результаты в обратном порядке, то есть самые последние записи будут находиться на первых местах. Тема и записи сохраняются в словаре `context` ❹, который передается шаблону topic.html ❺.

ПРИМЕЧАНИЕ Выражения в строках ❷ и ❸, обращающиеся к базе данных за конкретной информацией, называются запросами. Когда вы пишете подобные запросы для своих проектов, сначала опробуйте их в оболочке Django. Вы сможете проверить результат намного быстрее, чем если напишете представление и шаблон, а затем проверите результаты в браузере.

Шаблон отдельной темы

В шаблоне должно отображаться название темы и текст записей. Также необходимо сообщить пользователю, если по теме еще не было сделано ни одной записи:

topic.html

```
  {% extends 'learning_logs/base.html' %}

  {% block content %}

❶   <p>Topic: {{ topic }}</p>

    <p>Entries:</p>
❷   <ul>
❸   {% for entry in entries %}
      <li>
❹       <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
❺       <p>{{ entry.text|linebreaks }}</p>
      </li>
❻   {% empty %}
      <li>There are no entries for this topic yet.</li>
    {% endfor %}
    </ul>

  {% endblock content %}
```

Шаблон расширяет base.html, как и для всех страниц проекта. Затем выводится текущая тема ❶ из шаблонной переменной `{{ topic }}`. Переменная `topic` доступна, потому что она включена в словарь `context`. Затем создается маркированный список со всеми записями по теме ❷; перебор записей осуществляется так же, как это делалось ранее для тем ❸.

С каждым элементом списка связываются два значения: временная метка и полный текст каждой записи. Для временной метки ❹ выводится значение атрибута `date_added`. В шаблонах Django вертикальная черта (`|`) представляет *фильтр* — функцию, изменяющую значение шаблонной переменной. Фильтр `date:'M d, Y H:i'` выводит временные метки в формате *January 1, 2018 23:00*. Следующая строка выводит полное значение `text` (вместо первых 50 символов каждой записи). Фильтр `linebreaks` ❺ следит за тем, чтобы длинный текст содержал разрывы строк в формате, поддерживаемом браузером (вместо блока непрерывного текста). В точке ❻ шаблонный тег `{% empty %}` используется для вывода сообщения об отсутствии записей.

Ссылки на странице

Прежде чем просматривать страницу отдельной темы в браузере, необходимо изменить шаблон списка тем, чтобы каждая тема вела на соответствующую страницу. Внесите следующие изменения в topics.html:

topics.html

```
...
    {% for topic in topics %}
      <li>
        <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
      </li>
    {% empty %}
...
```

Шаблонный тег URL используется для генерирования ссылки на основании схемы URL из `learning_logs` с именем `'topic'`. Этой схеме URL необходим аргумент `topic_id`, поэтому в шаблонный тег URL добавляется атрибут `topic.id`. Теперь каждая тема в списке представляет собой ссылку на страницу темы, например http://localhost:8000/topics/1/.

Если теперь обновить страницу тем и щелкнуть на теме, открывается страница, изображенная на рис. 18.5.

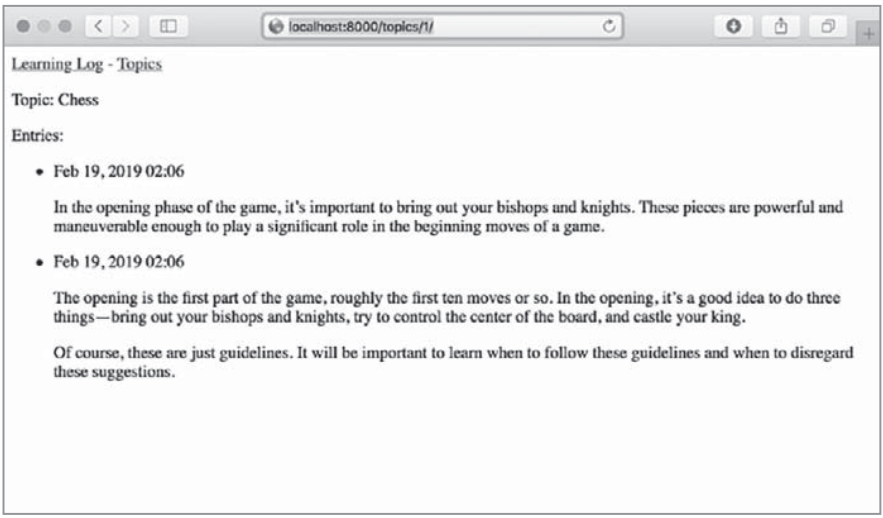

Рис. 18.5. Страница со списком всех записей по отдельной теме

ПРИМЕЧАНИЕ Между `topic.id` и `topic_id` существует неочевидное, но важное различие. Выражение `topic.id` проверяет тему и получает значение соответствующего идентификатора. Переменная `topic_id` содержит ссылку на этот идентификатор в коде. Если вы столкнетесь с ошибками при работе с идентификаторами, убедитесь в том, что эти выражения используются правильно.

УПРАЖНЕНИЯ

18.7. Документация шаблонов: просмотрите документацию по шаблонам Django по адресу https://docs.djangoproject.com/en/2.2/ref/templates/. Используйте ее в работе над собственными проектами.

18.8. Страницы Pizzeria: добавьте страницу в проект `Pizzeria` из упражнения 18.6 (с. 413) с названиями видов пиццы. Свяжите каждое название пиццы со страницей, на которой выводится список дополнений к этой пицце. Обязательно примените наследование шаблонов, чтобы повысить эффективность построения страниц.

Итоги

В этой главе вы начали осваивать построение веб-приложений с использованием инфраструктуры Django. Вы написали короткую спецификацию проекта, установили Django в виртуальной среде, узнали, как настроить проект, и проверили правильность настройки. Вы узнали, как создать приложение и как определить модели для представления данных в вашем приложении. Также были рассмотрены базы данных и вы узнали, как Django упрощает миграцию баз данных после внесения изменений в модель. Вы научились создавать суперпользователей для административного сайта, а также использовали административный сайт для ввода исходных данных.

Также в этой главе была представлена оболочка Django, позволяющая работать с данными проекта в терминальном сеансе. Вы научились определять URL-адреса, создавать функции представления и писать шаблоны для построения страниц сайта. Наконец, вы применили механизм наследования шаблонов, который упрощает структуру отдельных шаблонов и модификацию сайта по мере развития проекта.

В главе 19 мы создадим интуитивно понятные, удобные страницы, на которых пользователи смогут добавлять новые темы и записи, а также редактировать существующие записи без участия административного сайта. Также будет добавлена система регистрации пользователей, чтобы любой пользователь мог создать учетную запись и вести свой журнал. Собственно, в этом и заключается сущность веб-приложения — создание функциональности, с которой может взаимодействовать любое количество пользователей.

19 Учетные записи пользователей

Что является самым главным для веб-приложения? Что любой пользователь, живущий в любой стране мира, сможет создать учетную запись в вашем приложении и начать работать с ним. В этой главе мы построим формы, на которых пользователи смогут вводить свои темы и записи, а также редактировать существующие данные. Также вы узнаете, как Django защищает приложения от распространенных атак на страницы с формами, чтобы вам не приходилось тратить много времени на продумывание средств защиты вашего приложения.

Затем будет реализована система проверки пользователей. Мы создадим страницу регистрации, на которой пользователи смогут создавать учетные записи, и ограничим доступ к некоторым страницам для анонимных пользователей. Затем некоторые функции представления будут изменены так, чтобы пользователь мог видеть только свои собственные данные. Вы узнаете, как обеспечить безопасность и конфиденциальность данных пользователей.

Редактирование данных

Прежде чем строить систему аутентификации пользователей для создания учетных записей, сначала мы добавим несколько страниц, на которых пользователи смогут вводить собственные данные. У пользователей появится возможность создавать новые темы, добавлять новые записи и редактировать записи, сделанные ранее.

В настоящее время данные могут вводиться только суперпользователем на административном сайте. Однако разрешать пользователям работу на административном сайте явно нежелательно, поэтому мы воспользуемся средствами построения форм Django для создания страниц, на которых пользователи смогут вводить данные.

Добавление новых тем

Начнем с возможности создания новых тем. Страницы на базе форм добавляются практически так же, как и те страницы, которые мы уже строили ранее: вы определяете URL, пишете функцию представления и создаете шаблон. Принципиальное отличие — добавление нового модуля forms.py, содержащего функциональность форм.

Объект ModelForm

Любая страница, на которой пользователь может вводить и отправлять информацию, является *формой*, даже если на первый взгляд она на форму не похожа. Когда пользователь вводит информацию, необходимо *проверить*, что он ввел корректные данные, а не вредоносный код (например, код для нарушения работы сервера). Затем проверенная информация обрабатывается и сохраняется в нужном месте базы данных. Django автоматизирует большую часть этой работы.

Простейший способ построения форм в Django основан на использовании класса `ModelForm`, который автоматически строит форму на основании моделей, определенных в главе 18. Ваша первая форма будет создана в файле forms.py, который должен находиться в одном каталоге с models.py:

forms.py

```
  from django import forms

  from .models import Topic

❶ class TopicForm(forms.ModelForm):
      class Meta:
❷         model = Topic
❸         fields = ['text']
❹         labels = {'text': ''}
```

Сначала импортируется модуль `forms` и модель, с которой мы будем работать: `Topic`. В точке ❶ определяется класс с именем `TopicForm`, наследующий от `forms.ModelForm`.

Простейшая версия `ModelForm` состоит из вложенного класса `Meta`, который сообщает Django, на какой модели должна базироваться форма и какие поля на ней должны находиться. В точке ❷ форма создается на базе модели `Topic`, а на ней размещается только поле `text` ❸. Код ❹ приказывает Django не генерировать подпись для текстового поля.

URL-адрес для new_topic

URL-адрес новой страницы должен быть простым и содержательным, поэтому после того, как пользователь выбрал команду создания новой темы, он направляется по адресу http://localhost:8000/new_topic/. Ниже приведена схема URL для страницы `new_topic`, которая добавляется в learning_logs/urls.py:

urls.py

```
...
urlpatterns = [
    ...
    # Страница для добавления новой темы
    path('new_topic/', views.new_topic, name='new_topic'),
]
```

Эта схема URL будет отправлять запросы функции представления `new_topic()`, которую мы сейчас напишем.

Функция представления new_topic()

Функция `new_topic()` должна обрабатывать две разные ситуации: исходные запросы страницы `new_topic` (в этом случае должна отображаться пустая форма) и обработка данных, отправленных на форме. Затем она должна перенаправить пользователя обратно на страницу `topics`:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic
from .forms import TopicForm

...
def new_topic(request):
    """Определяет новую тему."""
❶   if request.method != 'POST':
        # Данные не отправлялись; создается пустая форма.
❷       form = TopicForm()
    else:
        # Отправлены данные POST; обработать данные.
❸       form = TopicForm(data=request.POST)
❹       if form.is_valid():
❺           form.save()
❻           return redirect('learning_logs:topics')

    # Вывести пустую или недействительную форму.
❼   context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
```

Мы импортируем класс `HttpResponseRedirect`, который будет использоваться для перенаправления пользователя к странице `topics` после отправки введенной темы. Функция `reverse()` определяет URL по заданной схеме URL (то есть Django сгенерирует URL при запросе страницы). Также импортируется только что написанная форма `TopicForm`.

Запросы GET и POST

При построении веб-приложений используются два основных типа запросов — GET и POST. Запросы *GET* используются для страниц, которые только читают данные с сервера, а запросы *POST* обычно используются в тех случаях, когда пользователь должен отправить информацию на форме. Для обработки всех наших форм будет использоваться метод POST (существуют и другие разновидности запросов, но в нашем проекте они не используются).

Функция `new_topic()` получает в параметре объект запроса. Когда пользователь впервые запрашивает эту страницу, его браузер отправляет запрос GET. Когда

пользователь уже заполнил и отправил форму, его браузер отправляет запрос POST. В зависимости от типа запроса мы определяем, запросил ли пользователь пустую форму (запрос GET) или предлагает обработать заполненную форму (запрос POST).

Метод запроса — GET или POST — проверяется в точке ❶. Если метод запроса отличен от POST, вероятно, используется запрос GET, поэтому необходимо вернуть пустую форму (даже если это запрос другого типа, это все равно безопасно). Мы создаем экземпляр `TopicForm` ❷, сохраняем его в переменной `form` и отправляем форму шаблону в словаре `context` ❼. Так как при создании `TopicForm` аргументы не передавались, Django создает пустую форму, которая заполняется пользователем.

Если используется метод запроса POST, выполняется блок `else`, который обрабатывает данные, отправленные в форме. Мы создаем экземпляр `TopicForm` ❸ и передаем ему данные, введенные пользователем, хранящиеся в `request.POST`. Возвращаемый объект `form` содержит информацию, отправленную пользователем.

Отправленную информацию нельзя сохранять в базе данных до тех пор, пока она не будет проверена ❹. Функция `is_valid()` проверяет, что все обязательные поля были заполнены (все поля формы по умолчанию являются обязательными), а введенные данные соответствуют типам полей — например, что длина текста меньше 200 символов, как было указано в файле models.py в главе 18. Автоматическая проверка избавляет нас от большого объема работы. Если все данные действительны, можно вызвать метод `save()` ❺, который записывает данные из формы в базу данных.

После того как данные будут сохранены, страницу можно покинуть. Мы используем вызов `redirect()` ❻ для перенаправления браузера на страницу `topics`, на которой пользователь увидит только что введенную им тему в общем списке тем.

Переменная `context` определяется в конце функции представления ❼, а страница строится на базе шаблона new_topic.html, который будет создан на следующем шаге. Код размещается за пределами любых блоков `if`; он выполняется при создании пустой формы, а также при определении того, что отправленная форма была недействительной. Недействительная форма включает стандартные сообщения об ошибках, чтобы помочь пользователю передать действительные данные.

Шаблон new_topic

Теперь создадим новый шаблон с именем new_topic.html для отображения только что созданной формы:

new_topic.html

```
{% extends "learning_logs/base.html" %}

{% block content %}
```

```
    <p>Add a new topic:</p>

❶   <form action="{% url 'learning_logs:new_topic' %}" method='post'>
❷     {% csrf_token %}
❸     {{ form.as_p }}
❹     <button name="submit">add topic</button>
    </form>


  {% endblock content %}
```

Этот шаблон расширяет base.html, поэтому он имеет такую же базовую структуру, как и остальные страницы Learning Log. В точке ❶ определяется форма HTML. Аргумент `action` сообщает серверу, куда передавать данные, отправленные формой; в данном случае данные возвращаются функции представления `new_topic()`. Аргумент `method` приказывает браузеру отправить данные в запросе типа POST.

Django использует шаблонный тег `{% csrf_token %}` ❷ для предотвращения попыток получения несанкционированного доступа к серверу (атаки такого рода называются *межсайтовой подделкой запросов*). В точке ❸ отображается форма; это наглядный пример того, как легко в Django выполняются такие стандартные операции, как отображение формы. Чтобы автоматически создать все поля, необходимые для отображения формы, достаточно включить шаблонную переменную `{{ form.as_p }}`. Модификатор `as_p` приказывает Django отобразить все элементы формы в формате абзацев — это простой способ аккуратного отображения формы.

Django не создает кнопку отправки данных для форм, поэтому мы определяем ее в точке ❹.

Создание ссылки на страницу new_topic

Далее ссылка на страницу `new_topic` создается на странице `topics`:

topics.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p>Topics</p>

  <ul>
    ...
  </ul>

  <a href="{% url 'learning_logs:new_topic' %}">Add a new topic:</a>

{% endblock content %}
```

Разместите ссылку после списка существующих тем. Полученная форма изображена на рис. 19.1. Воспользуйтесь ею и добавьте несколько своих тем.

Рис. 19.1. Страница для добавления новой темы

Добавление новых записей

Теперь, когда пользователь может добавлять новые темы, он также захочет добавлять новые записи. Мы снова определим URL, напишем новую функцию и шаблон и создадим ссылку на страницу. Но сначала нужно добавить в forms.py еще один класс.

Класс EntryForm

Мы должны создать форму, связанную с моделью `Entry`, но более специализированную по сравнению с `TopicForm`:

forms.py

```
from django import forms

from .models import Topic, Entry

class TopicForm(forms.ModelForm):
    ...

class EntryForm(forms.ModelForm):
    class Meta:
        model = Entry
        fields = ['text']
❶       labels = {'text': 'Entry:'}
❷       widgets = {'text': forms.Textarea(attrs={'cols': 80})}
```

Сначала в команду `import` к `Topic` добавляется `Entry`. Новый класс `EntryForm` наследует от `forms.ModelForm` и содержит вложенный класс `Meta` с указанием модели,

на которой он базируется, и поле, включаемое на форму. Полю `'text'` снова назначается пустая надпись ❶.

В точке ❷ включается атрибут `widgets`. *Виджет* (widget) представляет собой элемент формы HTML: однострочное или многострочное текстовое поле, раскрывающийся список и т. д. Включая атрибут `widgets`, вы можете переопределить виджеты, выбранные Django по умолчанию. Приказывая Django использовать элемент `forms.Textarea`, мы настраиваем виджет ввода для поля `'text'`, чтобы ширина текстовой области составляла 80 столбцов вместо значения по умолчанию 40. У пользователя будет достаточно места для создания содержательных записей.

URL-адрес для new_entry

Необходимо включить аргумент `topic_id` в URL-адрес для создания новой записи, потому что запись должна ассоциироваться с конкретной темой. Вот как выглядит URL, который мы добавляем в learning_logs/urls.py:

urls.py

```
...
urlpatterns = [
    ...
    # Страница для добавления новой записи
    path('new_entry/<int:topic_id>/', views.new_entry, name='new_entry'),
]
```

Эта схема URL соответствует любому URL-адресу в форме http://localhost:8000/new_entry/id/, где *id* — число, равное идентификатору темы. Код `<int:topic_id>` захватывает числовое значение и сохраняет его в переменной `topic_id`. При запросе URL-адреса, соответствующего этой схеме, Django передает запрос и идентификатор темы функции представления `new_entry()`.

Функция представления new_entry()

Функция представления `new_entry` очень похожа на функцию добавления новой темы. Включите следующий код в файл views.py:

views.py

```
  from django.shortcuts import render, redirect

  from .models import Topic
  from .forms import TopicForm, EntryForm

  ...
  def new_entry(request, topic_id):
      """Добавляет новую запись по конкретной теме."""
❶     topic = Topic.objects.get(id=topic_id)
```

```
❷     if request.method != 'POST':
          # Данные не отправлялись; создается пустая форма.
❸         form = EntryForm()
      else:
          # Отправлены данные POST; обработать данные.
❹         form = EntryForm(data=request.POST)
          if form.is_valid():
❺             new_entry = form.save(commit=False)
❻             new_entry.topic = topic
              new_entry.save()
❼             return redirect('learning_logs:topic', topic_id=topic_id)

      # Вывести пустую или недействительную форму.
      context = {'topic': topic, 'form': form}
      return render(request, 'learning_logs/new_entry.html', context)
```

Мы обновляем команду `import` и включаем в нее только что созданный класс `EntryForm`. Определение `new_entry()` содержит параметр `topic_id` для сохранения полученного значения из URL. Идентификатор темы понадобится для отображения страницы и обработки данных формы, поэтому мы используем `topic_id` для получения правильного объекта темы ❶.

В точке ❷ проверяется метод запроса: POST или GET. Блок `if` выполняется для запроса GET, и мы создаем пустой экземпляр `EntryForm` ❸.

Для метода запроса POST мы обрабатываем данные, создавая экземпляр `EntryForm`, заполненный данными POST из объекта `request` ❹. Затем проверяется корректность данных формы. Если данные корректны, необходимо задать атрибут `topic` объекта записи перед сохранением его в базе данных. При вызове `save()` мы включаем аргумент `commit=False` ❺ для того, чтобы создать новый объект записи и сохранить его в `new_entry`, не сохраняя пока в базе данных. Мы присваиваем атрибуту `topic` объекта `new_entry` тему, прочитанную из базы данных в начале функции ❻, после чего вызываем `save()` без аргументов. В результате запись сохраняется в базе данных с правильно ассоциированной темой.

Вызов `redirect()` в точке ❼ получает два аргумента — имя представления, которому передается управление, и аргумент для функции представления. В данном случае происходит перенаправление функции `topic()`, которой должен передаваться аргумент `topic_id`. Вызов перенаправляет пользователя на страницу темы, для которой была создана запись, и пользователь видит новую запись в списке записей.

В конце функции создается словарь `context`, а страница строится на базе шаблона new_entry.html. Этот код выполняется для пустой формы или для отправленной формы, которая была определена как недействительная.

Шаблон new_entry

Как видно из следующего кода, шаблон `new_entry` похож на шаблон `new_topic`:

new_entry.html

```
  {% extends "learning_logs/base.html" %}

  {% block content %}

❶   <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

    <p>Add a new entry:</p>
❷   <form action="{% url 'learning_logs:new_entry' topic.id %}" method='post'>
      {% csrf_token %}
      {{ form.as_p }}
      <button name='submit'>add entry</button>
    </form>

  {% endblock content %}
```

В начале страницы выводится тема ❶, чтобы пользователь мог видеть, в какую тему добавляется новая запись. Тема также служит ссылкой на главную страницу этой темы.

Аргумент `action` формы включает значение `topic_id` из URL, чтобы функция представления могла связать новую запись с правильной темой ❷. В остальном этот шаблон почти не отличается от new_topic.html.

Создание ссылки на страницу new_entry

Затем необходимо создать ссылку на страницу `new_entry` на каждой странице темы:

topic.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p>Topic: {{ topic }}</p>

  <p>Entries:</p>
  <p>
    <a href="{% url 'learning_logs:new_entry' topic.id %}">add new entry</a>
  </p>

  <ul>
  ...
  </ul>

{% endblock content %}
```

Ссылка добавляется перед выводом записей, потому что добавление новой записи является самым частым действием на этой странице. На рис. 19.2 изображена страница `new_entry`. Теперь пользователь может добавить сколько угодно новых тем и новых записей по каждой теме. Опробуйте страницу `new_entry`, добавив несколько записей для каждой из созданных вами тем.

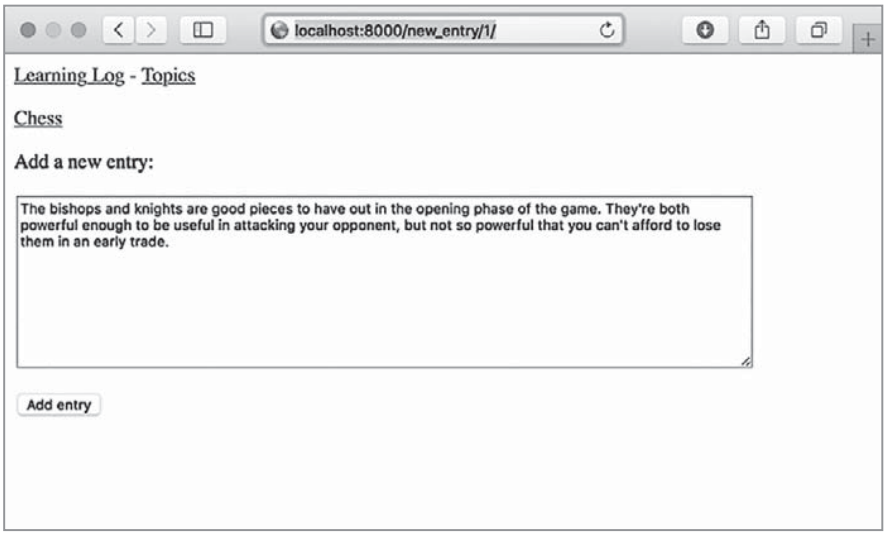

Рис. 19.2. Страница new_entry

Редактирование записей

А теперь мы создадим страницу, на которой пользователи смогут редактировать ранее добавленные записи.

URL-адрес для edit_entry

В URL-адресе страницы должен передаваться идентификатор редактируемой записи. В файл learning_logs/urls.py для этого вносятся следующие изменения:

urls.py

```
...
urlpatterns = [
    ...
    # Страница для редактирования записи
    path('edit_entry/<int:entry_id>/', views.edit_entry, name='edit_entry'),
]
```

Идентификатор, переданный в URL (например, http://localhost:8000/edit_entry/1/), сохраняется в параметре `entry_id`. Схема URL отправляет запросы, соответствующие этому формату, функции представления `edit_entry()`.

Функция представления edit_entry()

Когда страница `edit_entry` получает запрос GET, `edit_entry()` возвращает форму для редактирования записи. При получении запроса POST с отредактированной записью страница сохраняет измененный текст в базе данных:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic, Entry
from .forms import TopicForm, EntryForm
...

def edit_entry(request, entry_id):
    """Редактирует существующую запись."""
❶   entry = Entry.objects.get(id=entry_id)
    topic = entry.topic

    if request.method != 'POST':
        # Исходный запрос; форма заполняется данными текущей записи.
❷       form = EntryForm(instance=entry)
    else:
        # Отправка данных POST; обработать данные.
❸       form = EntryForm(instance=entry, data=request.POST)
        if form.is_valid():
❹           form.save()
❺           return redirect('learning_logs:topic', topic_id=topic.id)

    context = {'entry': entry, 'topic': topic, 'form': form}
    return render(request, 'learning_logs/edit_entry.html', context)
```

Сначала необходимо импортировать модель `Entry`. В точке ❶ мы получаем объект записи, который пользователь хочет изменить, и тему, связанную с этой записью. В блоке `if`, который выполняется для запроса GET, создается экземпляр `EntryForm` с аргументом `instance=entry` ❷. Этот аргумент приказывает Django создать форму, заранее заполненную информацией из существующего объекта записи. Пользователь видит свои существующие данные и может отредактировать их.

При обработке запроса POST передаются аргументы `instance=entry` и `data=request.POST` ❸. Они приказывают Django создать экземпляр формы на основании информации существующего объекта записи, обновленный данными из `request.POST`. Затем проверяется корректность данных формы. Если данные корректны, следует вызов `save()` без аргументов ❹. Далее происходит перенаправление на страницу темы ❺, и пользователь видит обновленную версию отредактированной им записи.

Если отображается исходная форма для редактирования записи или если отправленная форма недействительна, создается словарь `context`, а страница строится на базе шаблона edit_entry.html.

Шаблон edit_entry

Шаблон edit_entry.html очень похож на new_entry.html:

edit_entry.html

```
{% extends "learning_logs/base.html" %}

{% block content %}
```

```
    <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

    <p>Edit entry:</p>

❶   <form action="{% url 'learning_logs:edit_entry' entry.id %}" method='post'>
      {% csrf_token %}
      {{ form.as_p }}
❷     <button name="submit">save changes</button>
    </form>

  {% endblock content %}
```

В точке ❶ аргумент `action` отправляет форму функции `edit_entry()` для обработки. Идентификатор записи включается как аргумент в тег `{% url %}` , чтобы функция представления могла изменить правильный объект записи. Кнопка отправки данных создается с текстом, который напоминает пользователю, что он сохраняет изменения, а не создает новую запись ❷.

Создание ссылки на страницу edit_entry

Теперь необходимо включить ссылку на страницу `edit_entry` в каждую тему на странице со списком тем:

topic.html

```
...
  {% for entry in entries %}
    <li>
      <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
      <p>{{ entry.text|linebreaks }}</p>
      <p>
        <a href="{% url 'learning_logs:edit_entry' entry.id %}">Edit entry</a>
      </p>
    </li>
...
```

После даты и текста каждой записи включается ссылка редактирования. Мы используем шаблонный тег `{% url %}` для определения схемы URL из именованной схемы `edit_entry` и идентификатора текущей записи в цикле (`entry.id`). Текст ссылки `"edit entry"` выводится после каждой записи на странице. На рис. 19.3 показано, как выглядит страница со списком тем с этими ссылками.

Приложение Learning Log уже сейчас содержит большую часть необходимой функциональности. Пользователи могут добавлять темы и записи, а также читать любые записи по своему усмотрению. В этом разделе мы реализуем систему регистрации пользователей, чтобы любой желающий мог создать свою учетную запись в Learning Log и ввести собственный набор тем и записей.

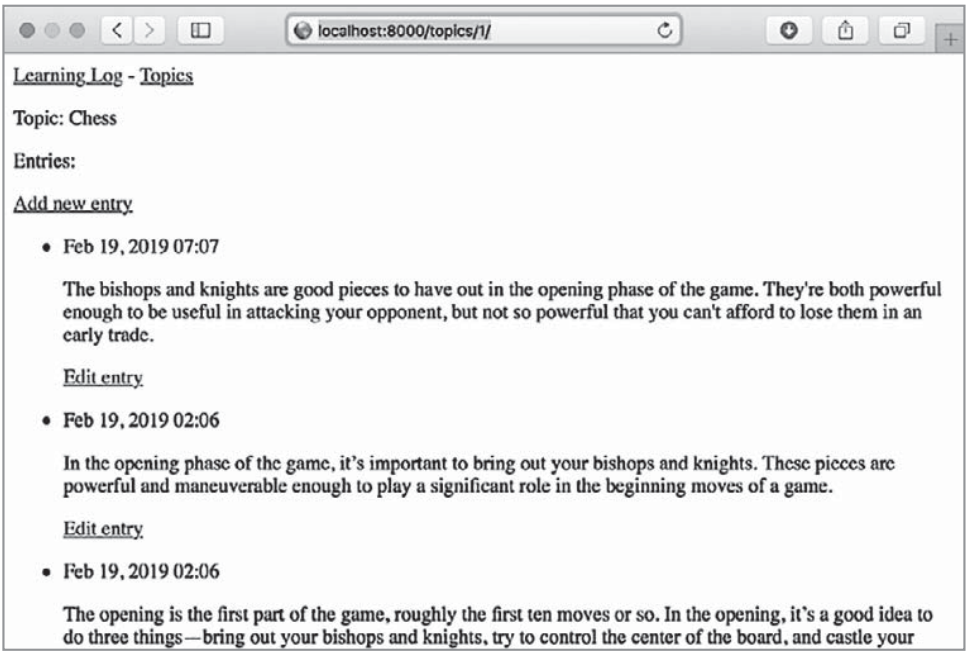

Рис. 19.3. Каждая запись снабжается ссылкой для редактирования этой записи

УПРАЖНЕНИЯ

19.1. Блог: создайте новый проект Django с именем Blog. Создайте в проекте приложение с именем blogs и моделью `BlogPost`. Модель должна содержать такие поля, как `title`, `text` и `date_added`. Создайте суперпользователя для проекта и воспользуйтесь административным сайтом для создания пары коротких сообщений. Создайте домашнюю страницу, на которой выводятся все сообщения в хронологическом порядке.

Сделайте одну форму для создания новых сообщений, а другую форму для редактирования существующих сообщений. Заполните формы и убедитесь в том, что они работают.

Создание учетных записей пользователей

В этом разделе мы создадим систему регистрации и авторизации пользователей, чтобы люди могли создать учетную запись, начать и завершать сеанс работы с приложением. Для всей функциональности, относящейся к работе с пользователями, будет создано отдельное приложение. Мы также слегка изменим модель `Topic`, чтобы каждая тема была связана с конкретным пользователем.

Приложение users

Начнем с создания нового приложения `users` командой `startapp`:

```
  (ll_env)learning_log$ python manage.py startapp users
  (ll_env)learning_log$ ls
❶ db.sqlite3  learning_log  learning_logs  ll_env  manage.py  users
  (ll_env)learning_log$ ls users
❷ __init__.py admin.py apps.py migrations models.py tests.py views.py
```

Эта команда создает новый каталог с именем users ❶, структура которого повторяет структуру каталогов приложения learning_logs ❷.

Добавление пользователей в settings.py

Новое приложение необходимо добавить в settings.py:

settings.py

```
...
INSTALLED_APPS = [
    # Мои приложения
    'learning_logs',
    'users',

    # Приложения django по умолчанию.
    ...
]
...
```

Django включает приложение `users` в общий проект.

Включение URL-адресов из users

Затем необходимо изменить корневой файл urls.py, чтобы он включал URL-адреса, написанные для приложения `users`:

urls.py

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('users/', include('users.urls')),
    path('', include('learning_logs.urls')),
]
```

Добавим строку для включения файла urls.py из `users`. Эта строка будет соответствовать любому URL-адресу, начинающемуся со слова `users`, например http://localhost:8000/users/login/.

Страница входа

Начнем с реализации страницы входа. Мы воспользуемся стандартным представлением `login`, которое предоставляет Django, так что шаблон URL выглядит немного иначе. Создайте новый файл urls.py в каталоге learning_log/users/ и добавьте в него следующий код:

urls.py

```
  """Определяет схемы URL для пользователей"""

  from django.urls import path, include

❶ app_name = 'users'
  urlpatterns = [
      # Включить URL авторизации по умолчанию.
❷     path('', include('django.contrib.auth.urls')),
  ]
```

Сначала импортируется функция `path`, а затем функция `include` для включения аутентификационных URL-адресов по умолчанию, определенных Django. Эти URL-адреса по умолчанию включают именованные схемы, такие как `'login'` и `'logout'`. Переменной `app_name` присваивается значение `'users'`, чтобы инфраструктура Django могла отличить эти URL-адреса от URL-адресов, принадлежащих другим приложениям ❶. Даже URL-адреса по умолчанию, предоставляемые Djano, при включении в файл urls.py приложения `users` будут доступны через пространство имен `users`.

Схема страницы входа соответствует URL http://localhost:8000/users/login/ ❷. Когда Django читает этот URL-адрес, слово `users` указывает, что следует обратиться к users/urls.py, а `login` сообщает о том, что запросы должны отправляться представлению `login` по умолчанию.

Шаблон login

Когда пользователь запрашивает страницу входа, Django использует свое представление `login` по умолчанию, но мы все равно должны предоставить шаблон для этой страницы. Аутентификационные представления по умолчанию ищут шаблоны в каталоге с именем registration, поэтому вы должны создать этот каталог. В каталоге learning_log/users/ создайте каталог с именем templates, а внутри него — еще один каталог с именем registration. Вот как выглядит шаблон login.html, который должен находиться в learning_log/users/templates/registration/:

login.html

```
  {% extends "learning_logs/base.html" %}

  {% block content %}

❶   {% if form.errors %}
```

```
      <p>Your username and password didn't match. Please try again.</p>
    {% endif %}

❷   <form method="post" action="{% url 'users:login' %}">
      {% csrf_token %}
❸     {{ form.as_p }}

❹     <button name="submit">log in</button>
❺     <input type="hidden" name="next"
        value="{% url 'learning_logs:index' %}" />
    </form>

  {% endblock content %}
```

Шаблон расширяет base.html, чтобы страница входа по оформлению и поведению была похожа на другие страницы сайта. Обратите внимание: шаблон в одном приложении может расширять шаблон из другого приложения.

Если у формы установлен атрибут `errors`, выводится сообщение об ошибке ❶. В нем говорится, что комбинация имени пользователя и пароля не соответствует информации, хранящейся в базе данных.

Мы хотим, чтобы представление обработало форму, поэтому аргументу `action` присваивается URL страницы входа ❷. Представление отправляет форму шаблону, мы должны вывести форму ❸ и добавить кнопку отправки данных ❹. В точке ❺ включается скрытый элемент формы `'next'`; аргумент `value` сообщает Django, куда перенаправить пользователя после успешно выполненного входа. В нашем случае пользователь возвращается обратно на домашнюю страницу.

Создание ссылки на страницу входа

Добавим ссылку на страницу входа в base.html, чтобы она присутствовала на каждой странице. Ссылка не должна отображаться, если пользователь уже прошел процедуру входа, поэтому она вкладывается в тег `{% if %}`:

base.html

```
  <p>
    <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
    <a href="{% url 'learning_logs:topics' %}">Topics</a> -
❶   {% if user.is_authenticated %}
❷     Hello, {{ user.username }}.
    {% else %}
❸     <a href="{% url 'users:login' %}">log in</a>
    {% endif %}
  </p>

  {% block content %}{% endblock content %}
```

В системе аутентификации Django в каждом шаблоне доступна переменная `user`, которая всегда имеет атрибут `is_authenticated`: атрибут равен `True`, если пользо-

ватель прошел проверку, и `False` в противном случае. Это позволяет вам выводить разные сообщения для проверенных и непроверенных пользователей.

В данном случае мы выводим приветствие для пользователей, выполнивших вход ❶. У проверенных пользователей устанавливается дополнительный атрибут `username`, который обеспечит личную настройку приветствия и напомнит пользователю о том, что вход был выполнен ❷. В точке ❸ выводится ссылка на страницу входа для пользователей, которые еще не прошли проверку.

Использование страницы входа

Учетная запись пользователя уже создана; попробуем ввести данные и посмотрим, работает ли страница. Откройте страницу http://localhost:8000/admin/. Если вы все еще работаете с правами администратора, найдите ссылку выхода в заголовке и щелкните на ней.

После выхода перейдите по адресу http://localhost:8000/users/login/. На экране должна появиться страница входа, похожая на рис. 19.4. Введите имя пользователя и пароль, заданные ранее, и вы снова должны оказаться на странице со списком. В заголовке страницы должно выводиться сообщение с указанием имени пользователя.

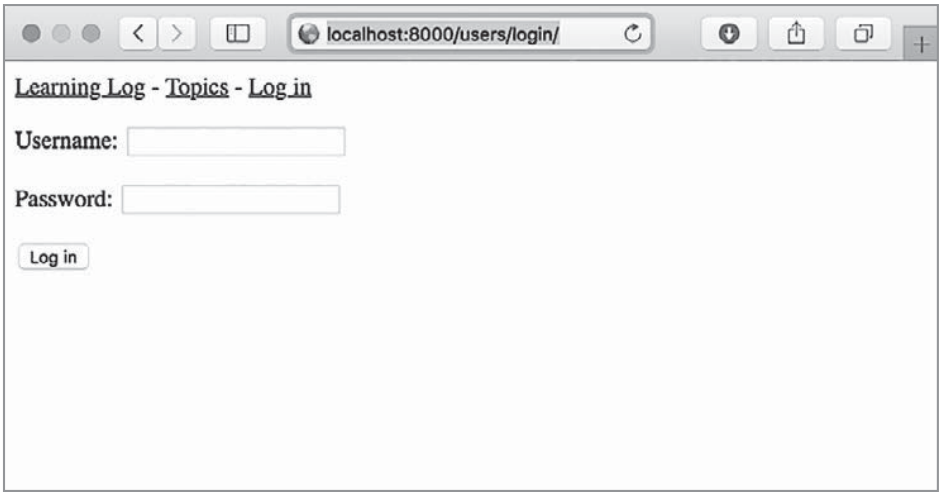

Рис. 19.4. Страница входа

Выход

Теперь необходимо предоставить пользователям возможность выхода из приложения. Мы включим в base.html ссылку для выхода пользователя; при щелчке на этой ссылке открывается страница, подтверждающая, что выход был выполнен успешно.

Добавление ссылки для выхода

Теперь нужно создать ссылку для выхода. Мы добавим ее в файл base.html, чтобы она была доступна на каждой странице, и включим в секцию `{% if user.is_authenticated %}`, чтобы ссылка была видна только пользователям, уже выполнившим вход:

base.html

```
...
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">log out</a>
  {% else %}
    ...
```

По умолчанию схеме URL для выхода назначается имя `'logout'`.

Страница подтверждения выхода

Пользователь должен знать, что выход прошел успешно, поэтому представление по умолчанию для выхода строит страницу на базе шаблона logged_out.html, который мы сейчас создадим. Он представляет простую страницу с уведомлением о том, что пользователь вышел из сеанса работы с приложением. Сохраните файл в каталоге templates/registration — в том же каталоге, в котором был сохранен файл login.html:

logged_out.html

```
{% extends "learning_logs/base.html" %}

{% block content %}
  <p>You have been logged out. Thank you for visiting!</p>
{% endblock content %}
```

Рис. 19.5. Страница выхода подтверждает, что выход был выполнен успешно

Ничего другого на этой странице быть не должно, потому что base.html предоставляет ссылки на домашнюю страницу и страницу входа на случай, если пользователь захочет вернуться к какой-либо из этих страниц.

На рис. 19.5 изображена страница выхода так, как ее видит пользователь, выполнивший вход. Оформление страницы минимально, потому что сейчас нас в первую очередь интересует работа сайта. Когда необходимые функции заработают, можно переходить к стилевому оформлению сайта и приданию ему более профессионального вида.

Страница регистрации

Теперь мы построим страницу для регистрации новых пользователей. Для этой цели мы используем класс Django `UserCreationForm`, но напишем собственную функцию представления и шаблон.

URL-адрес регистрации

Следующий код предоставляет шаблон URL для страницы регистрации — также в файле users/urls.py:

urls.py

```
""" Определяет схемы URL для пользователей. """

from django.urls import path, include

from . import views

app_name = 'users'
urlpatterns = [
    # Включить URL авторизации по умолчанию.
    path('', include('django.contrib.auth.urls')),
    # Страница регистрации.
    path('register/', views.register, name='register'),
]
```

Мы импортируем модуль `views` из `users`; этот модуль необходим, потому что мы пишем собственное представление для страницы регрессии. Шаблон соответствует URL http://localhost:8000/users/register/ и отправляет запросы функции `register()`, которую мы сейчас напишем.

Функция представления register()

Функция представления `register()` должна вывести пустую форму регистрации при первом запросе страницы регистрации, а затем обработать заполненную форму регистрации при отправке данных. Если регистрация прошла успешно, функция также должна выполнить вход для нового пользователя. Включите следующий код в users/views.py:

views.py

```
  from django.shortcuts import render, redirect
  from django.contrib.auth import login
  from django.contrib.auth.forms import UserCreationForm

  def register(request):
      """Регистрирует нового пользователя."""
      if request.method != 'POST':
          # Выводит пустую форму регистрации.
❶         form = UserCreationForm()
      else:
          # Обработка заполненной формы.
❷         form = UserCreationForm(data=request.POST)

❸         if form.is_valid():
❹             new_user = form.save()
              # Выполнение входа и перенаправление на домашнюю страницу.
❺             login(request, new_user)
❻             return redirect('learning_logs:index')

      # Вывести пустую или недействительную форму.
      context = {'form': form}
      return render(request, 'users/register.html', context)
```

Сначала импортируются функции `render()` и `redirect()`. Затем мы импортируем функцию `login()` для выполнения входа пользователя, если регистрационная информация верна. Также импортируется класс `UserCreationForm` по умолчанию. В функции `register()` мы проверяем, отвечает ли функция на запрос POST. Если нет, создается экземпляр `UserCreationForm`, не содержащий исходных данных ❶.

В случае ответа на запрос POST создается экземпляр `UserCreationForm`, основанный на отправленных данных ❷. Мы проверяем, что данные верны ❸; в данном случае что имя пользователя содержит правильные символы, пароли совпадают, а пользователь не пытается вставить вредоносные конструкции в отправленные данные.

Если отправленные данные верны, мы вызываем метод `save()` формы для сохранения имени пользователя и хеша пароля в базе данных ❹. Метод `save()` возвращает только что созданный объект пользователя, который сохраняется в `new_user`. После того как информация пользователя будет сохранена, мы выполняем вход; этот процесс состоит из двух шагов: сначала вызывается функция `login()` с объектами `request` и `new_user` ❺, которая создает действительный сеанс для нового пользователя. Наконец, пользователь перенаправляется на домашнюю страницу ❻, где приветствие в заголовке сообщает о том, что регистрация прошла успешно.

В конце функции строится страница, которая будет либо пустой формой, либо отправленной формой, содержащей недействительные данные.

Шаблон регистрации

Шаблон страницы регистрации похож на шаблон страницы входа. Проследите за тем, чтобы он был сохранен в одном каталоге с login.html:

register.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <form method="post" action="{% url 'users:register' %}">
    {% csrf_token %}
    {{ form.as_p }}

    <button name="submit">register</button>
    <input type="hidden" name="next" value="{% url 'learning_logs:index' %}" />
  </form>

{% endblock content %}
```

Мы снова используем метод as_p, чтобы инфраструктура Django могла правильно отобразить все поля формы, включая все сообщения об ошибках, если форма была заполнена неправильно.

Создание ссылки на страницу регистрации

Следующий шаг — добавление кода для вывода ссылки на страницу регистрации для любого пользователя, еще не выполнившего вход:

base.html

```
...
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">log out</a>
{% else %}
    <a href="{% url 'users:register' %}">Register</a> -
    <a href="{% url 'users:login' %}">log in</a>
  {% endif %}
...
```

Теперь пользователи, выполнившие вход, получат персональное приветствие и ссылку для выхода. Другие пользователи видят ссылку на страницу регистрации и ссылку для входа. Проверьте страницу регистрации, создав несколько учетных записей с разными именами пользователей.

В следующем разделе доступ к некоторым страницам будет ограничен, чтобы страницы были доступны только для зарегистрированных пользователей. Также необходимо позаботиться о том, чтобы каждая тема принадлежала конкретному пользователю.

ПРИМЕЧАНИЕ Такая система регистрации позволяет любому пользователю создать сколько угодно учетных записей Learning Log. Однако некоторые системы требуют, чтобы пользователь подтвердил свою заявку, отправляя сообщение электронной почты, на которое пользователь должен ответить. При таком подходе в системе будет создано меньше спамерских учетных записей, чем в простейшей системе из нашего примера. Но пока вы только учитесь строить приложения, вполне нормально тренироваться на упрощенной системе регистрации вроде используемой нами.

УПРАЖНЕНИЯ

19.2. Учетные записи в блоге: добавьте систему аутентификации и регистрации в проект Blog, работа над которым началась в упражнении 19.1 (с. 435). Проследите за тем, чтобы пользователь, выполнивший вход, видел свое имя где-то на экране, а незарегистрированные пользователи видели ссылку на страницу регистрации.

Редактирование данных

Пользователь должен иметь возможность вводить данные, принадлежащие только ему лично. Мы создадим систему, которая будет определять, какому пользователю принадлежат те или иные данные, и будет ограничивать доступ к страницам, чтобы пользователь мог работать только с принадлежащими ему данными.

В этом разделе мы изменим модель `Topic`, чтобы каждая тема принадлежала конкретному пользователю. При этом также автоматически решается проблема с записями, так как каждая запись принадлежит конкретной теме. Начнем с ограничения доступа к страницам.

Ограничение доступа с использованием @login_required

Django позволяет легко ограничить доступ к определенным страницам для пользователей, выполнивших вход, с помощью декоратора `@login_required`. *Декоратор* (decorator) представляет собой директиву, размещенную непосредственно перед определением функции, применяемую к функции перед ее выполнением и влияющую на поведение кода. Рассмотрим пример.

Ограничение доступа к страницам тем

Каждая тема будет принадлежать пользователю, поэтому только зарегистрированные пользователи смогут запрашивать страницы тем. Добавьте следующий код в learning_logs/views.py:

views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required

from .models import Topic, Entry
```

```
...
@login_required
def topics(request):
    """Выводит все темы."""
    ...
```

Сначала импортируется функция `login_required()`. Мы применяем `login_required()` как декоратор для функции представления `topics()`, для чего перед именем `login_required()` ставится знак `@`; он сообщает Python, что этот код должен выполняться перед кодом `topics()`.

Код `login_required()` проверяет, вошел ли пользователь в систему, и Django запускает код `topics()` только при выполнении этого условия. Если же пользователь не выполнил вход, он перенаправляется на страницу входа.

Чтобы перенаправление работало, необходимо внести изменения settings.py и сообщить Django, где искать страницу входа. Добавьте следующий фрагмент в самый конец settings.py:

settings.py

```
...
# Мои настройки
LOGIN_URL = '/users/login/'
```

Когда пользователь, не прошедший проверку, запрашивает страницу, защищенную декоратором `@login_required`, Django отправляет пользователя на URL-адрес, определяемый `LOGIN_URL` в settings.py.

Чтобы протестировать эту возможность, завершите сеанс в любой из своих учетных записей и вернитесь на домашнюю страницу. Щелкните на ссылке Topics, которая должна направить вас на страницу входа. Выполните вход с любой из своих учетных записей, на домашней странице снова щелкните на ссылке Topics. На этот раз вы получите доступ к странице со списком тем.

Ограничение доступа в Learning Log

Django упрощает ограничение доступа к страницам, но вы должны решить, какие страницы следует защищать. Лучше сначала подумать, к каким страницам можно разрешить неограниченный доступ, а затем ограничить его для всех остальных страниц. Снять излишние ограничения несложно, причем это куда менее рискованно, чем оставлять действительно важные страницы без ограничения доступа.

В приложении Learning Log мы оставим неограниченный доступ к домашней странице, странице регистрации и странице выхода. Доступ ко всем остальным страницам будет ограничен.

Вот как выглядит файл learning_logs/views.py с декораторами `@login_required`, примененными к каждому представлению, кроме `index()`:

views.py

```
...
@login_required
def topics(request):
    ...

@login_required
def topic(request, topic_id):
    ...

@login_required
def new_topic(request):
    ...

@login_required
def new_entry(request, topic_id):
    ...

@login_required
def edit_entry(request, entry_id):
    ...
```

Попробуйте обратиться к любой из этих страниц без выполнения входа: вы будете перенаправлены обратно на страницу входа. Кроме того, вы не сможете щелкать на ссылках на такие страницы, как `new_topic`. Но если ввести URL http://localhost:8000/new_topic/, вы будете перенаправлены на страницу входа. Ограничьте доступ ко всем URL-адресам, связанным с личными данными пользователей.

Связывание данных с конкретными пользователями

Теперь данные, отправленные пользователем, необходимо связать с тем пользователем, который их отправил. Связь достаточно установить только с данными, находящимися на высшем уровне иерархии, а низкоуровневые данные последуют за ними автоматически. Например, в приложении Learning Log на высшем уровне находятся темы, а каждая запись связывается с некоторой темой. Если каждая тема принадлежит конкретному пользователю, мы сможем отследить владельца каждой записи в базе данных.

Изменим модель `Topic` и добавим отношение внешнего ключа к пользователю. После этого необходимо провести миграцию базы данных. Наконец, необходимо изменить некоторые представления, чтобы в них отображались только данные, связанные с текущим пользователем.

Изменение модели Topic

В файле models.py изменяются всего две строки:

models.py

```
from django.db import models
from django.contrib.auth.models import User
```

```
class Topic(models.Model):
    """Тема, которую изучает пользователь"""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(User, on_delete=models.CASCADE)

def __str__(self):
    """Возвращает строковое представление модели."""
    return self.text

class Entry(models.Model):
    ...
```

Сначала модель `User` импортируется из `django.contrib.auth`. Затем в `Topic` добавляется поле `owner`, используемое в отношении внешнего ключа к модели `User`. Если пользователь удаляется, все темы, связанные с этим пользователем, также будут удалены.

Идентификация существующих пользователей

При проведении миграции Django модифицирует базу данных, чтобы в ней хранилась связь между каждой темой и пользователем. Для выполнения миграции Django необходимо знать, с каким пользователем должна быть связана каждая существующая тема. Проще всего связать все существующие темы с одним пользователем, например суперпользователем. Но для этого сначала необходимо узнать идентификатор этого пользователя.

Просмотрим идентификаторы всех пользователей, созданных до настоящего момента. Запустите сеанс оболочки Django и введите следующие команды:

```
  (ll_env)learning_log$ python manage.py shell
❶ >>> from django.contrib.auth.models import User
❷ >>> User.objects.all()
  <QuerySet [<User: ll_admin>, <User: eric>, <User: willie>]>
❸ >>> for user in User.objects.all():
  ...     print(user.username, user.id)
  ...
  ll_admin 1
  eric 2
  willie 3
  >>>
```

В точке ❶ в сеанс оболочки импортируется модель `User`. После этого просматриваются все пользователи, созданные до настоящего момента ❷. В выходных данных перечислены три пользователя: *ll_admin*, *eric* и *willie*.

В точке ❸ перебирается список пользователей, и для каждого пользователя выводится его имя и идентификатор. Когда Django спросит, с каким пользователем связать существующие темы, мы используем один из этих идентификаторов.

Миграция базы данных

Зная значение идентификатора, можно провести миграцию базы данных. Когда вы это делаете, Python предлагает связать модель `Topic` с конкретным владельцем временно или добавить в models.py значение по умолчанию, которое сообщит, как следует поступить. Выберите вариант 1:

```
❶ (ll_env)learning_log$ python manage.py makemigrations learning_logs
❷ You are trying to add a non-nullable field 'owner' to topic without a default;
  we can't do that (the database needs something to populate existing rows).
❸ Please select a fix:
   1) Provide a one-off default now (will be set on all existing rows with a
      null value for this column)
   2) Quit, and let me add a default in models.py
❹ Select an option: 1
❺ Please enter the default value now, as valid Python
  The datetime and django.utils.timezone modules are available, so you can do
  e.g. timezone.now
  Type 'exit' to exit this prompt
❻ >>> 1
  Migrations for 'learning_logs':
    learning_logs/migrations/0003_topic_owner.py
  - Add field owner to topic
  (ll_env)learning_log$
```

Сначала выдается команда `makemigrations` ❶. В ее выходных данных ❷ Django сообщает, что мы пытаемся добавить обязательное поле (значения которого отличны от null) в существующую модель (`topic`) без указания значения по умолчанию. Django предоставляет два варианта ❸: мы можем либо указать значение по умолчанию прямо сейчас, либо завершить выполнение программы и добавить значение по умолчанию в models.py. В точке ❹ выбирается первый вариант. Тогда Django запрашивает значение по умолчанию ❺.

Чтобы связать все существующие темы с исходным административным пользователем `ll_admin`, я ввел в точке ❺ идентификатор пользователя 1. Вы можете использовать идентификатор любого из созданных пользователей; он не обязан быть суперпользователем. Django проводит миграцию базы данных, используя это значение, и создает файл миграции 0003_topic_owner.py, добавляющий поле `owner` в модель `Topic`.

Теперь можно провести миграцию. Введите следующую команду в активной виртуальной среде:

```
(ll_env)learning_log$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
❶   Applying learning_logs.0003_topic_owner... OK
(ll_env)learning_log$
```

Django применяет новую миграцию с результатом `OK` ❶.

Чтобы убедиться в том, что миграция сработала так, как и ожидалось, можно воспользоваться интерактивной оболочкой:

```
❶ >>> from learning_logs.models import Topic
❷ >>> for topic in Topic.objects.all():
...     print(topic, topic.owner)
...
Chess ll_admin
Rock Climbing ll_admin
>>>
```

После импортирования `Topic` из `learning_logs.models` ❶ мы перебираем все существующие темы, выводим каждую тему и имя пользователя, которому она принадлежит ❷. Как видите, сейчас каждая тема принадлежит пользователю `ll_admin`. (Если при выполнении кода произойдет ошибка, попробуйте выйти из оболочки и запустить ее заново.)

ПРИМЕЧАНИЕ Вместо миграции можно просто сбросить содержимое базы данных, но это приведет к потере всех существующих данных. Полезно научиться выполнять миграцию базы данных без нарушения целостности данных пользователей. Если вы хотите начать с новой базы данных, используйте команду `python manage.py flush` для повторного построения структуры базы данных. Вам придется создать нового суперпользователя, а все данные будут потеряны.

Ограничение доступа к темам

В настоящее время пользователь, выполнивший вход, будет видеть все темы независимо от того, под какой учетной записью он вошел. Сейчас мы изменим приложение, чтобы каждый пользователь видел только принадлежащие ему темы.

Внесите следующее изменение в функцию `topics()` в файле views.py:

views.py

```
...
@login_required
def topics(request):
    """Выводит список тем."""
    topics = Topic.objects.filter(owner=request.user).order_by('date_added')
    context = {'topics': topics}
    return render(request, 'learning_logs/topics.html', context)
...
```

Если пользователь выполнил вход, в объекте запроса устанавливается атрибут `request.user` с информацией о пользователе. Фрагмент кода `Topic.objects.filter(owner=request.user)` приказывает Django извлечь из базы данных только те объекты `Topic`, у которых атрибут `owner` соответствует текущему пользователю. Так как способ отображения не изменяется, изменять шаблон для страницы тем вообще не нужно.

Чтобы увидеть, как работает этот способ, выполните вход в качестве пользователя, с которым связаны все существующие темы, и перейдите к странице со списком тем. На ней должны отображаться все темы. Теперь завершите сеанс и войдите снова с другой учетной записью. На этот раз страница должна быть пустой.

Защита тем пользователя

Никаких реальных ограничений на доступ к страницам еще не существует, поэтому любой зарегистрированный пользователь может опробовать разные URL (например, http://localhost:8000/topics/1/) и просмотреть страницы тем, которые ему удастся подобрать.

Попробуйте сделать это. После входа с учетной записью суперпользователя скопируйте URL или запишите идентификатор в URL темы, после чего завершите сеанс и войдите снова от имени другого пользователя. Введите URL этой темы. Вам удастся прочитать все записи, хотя сейчас вы вошли под именем другого пользователя.

Чтобы решить эту проблему, мы будем выполнять проверку перед получением запрошенных данных в функции представления `topic()`:

views.py

```
  from django.shortcuts import render, redirect
  from django.contrib.auth.decorators import login_required
❶ from django.http import Http404

  ...
  @login_required
  def topic(request, topic_id):
      """Выводит одну тему и все ее записи."""
      topic = Topic.objects.get(id=topic_id)
      # Проверка того, что тема принадлежит текущему пользователю.
❷     if topic.owner != request.user:
          raise Http404

      entries = topic.entry_set.order_by('-date_added')
      context = {'topic': topic, 'entries': entries}
      return render(request, 'learning_logs/topic.html', context)
  ...
```

Код 404 — стандартное сообщение об ошибке, которое возвращается в тех случаях, когда запрошенный ресурс не существует на сервере. В данном случае мы импортируем исключение `Http404` ❶, которое будет выдаваться программой при запросе пользователем темы, которую ему видеть не положено. Получив запрос темы, перед отображением страницы мы убеждаемся в том, что пользователь этой темы является текущим пользователем приложения. Если тема не принадлежит текущему пользователю, выдается исключение `Http404` ❷, а Django возвращает страницу с ошибкой 404.

Пока при попытке просмотреть записи другого пользователя вы получите от Django сообщение «Страница не найдена». В главе 20 проект будет настроен так, чтобы пользователь видел полноценную страницу ошибки.

Защита страницы edit_entry

Страницы `edit_entry` используют URL-адреса в форме http://localhost:8000/edit_entry/entry_id/, где entry_id — число. Защитим эту страницу, чтобы никто не мог подобрать URL для получения доступа к чужим записям:

views.py

```
...
@login_required
def edit_entry(request, entry_id):
    """Редактирует существующую запись."""
    entry = Entry.objects.get(id=entry_id)
    topic = entry.topic
    if topic.owner != request.user:
        raise Http404

    if request.method != 'POST':
        ...
```

Программа читает запись и тему, связанную с этой записью. Затем мы проверяем, совпадает ли владелец темы с текущим пользователем; при несовпадении выдается исключение `Http404`.

Связывание новых тем с текущим пользователем

В настоящее время страница добавления новых тем не совершенна, потому что она не связывает новые темы с конкретным пользователем. При попытке добавить новую тему выдается сообщение об ошибке `IntegrityError` с уточнением `NOT NULL constraint failed: learning_logs_topic.owner_id`. Django говорит, что при создании новой темы обязательно должно быть задано значение поля `owner`.

Проблема легко решается, потому что мы можем получить доступ к информации текущего пользователя через объект `request`. Добавьте следующий код, связывающий новую тему с текущим пользователем:

views.py

```
...
@login_required
def new_topic(request):
    """Определяет новую тему."""
    if request.method != 'POST':
        # Данные не отправлялись; создается пустая форма.
        form = TopicForm()
    else:
```

```
            # Отправлены данные POST; обработать данные.
            form = TopicForm(data=request.POST)
            if form.is_valid():
❶               new_topic = form.save(commit=False)
❷               new_topic.owner = request.user
❸               new_topic.save()
                return redirect('learning_logs:topics')

        # Вывести пустую или недействительную форму.
        context = {'form': form}
        return render(request, 'learning_logs/new_topic.html', context)
    ...
```

При первом вызове `form.save()` передается аргумент `commit=False`, потому что новая тема должна быть изменена перед сохранением в базе данных ❶. Атрибуту `owner` новой темы присваивается текущий пользователь ❷. Наконец, мы вызываем `save()` для только что определенного экземпляра темы ❸. Теперь тема содержит все обязательные данные и ее сохранение пройдет успешно.

Вы сможете добавить сколько угодно новых тем для любого количества разных пользователей. Каждому пользователю будут доступны только его собственные данные, какие бы операции он ни пытался выполнять — просмотр данных, ввод новых или изменение существующих данных.

УПРАЖНЕНИЯ

19.3. Рефакторинг: в views.py есть два места, в которых программа проверяет, что пользователь, связанный с темой, является текущим пользователем. Поместите код этой проверки в функцию с именем `check_topic_owner()` и вызовите эту функцию при необходимости.

19.4. Защита new_entry: пользователь может попытаться добавить новую запись в журнал другого пользователя, вводя URL-адрес с идентификатором темы, принадлежащей другому пользователю. Чтобы предотвратить подобные атаки, перед сохранением новой записи проверьте, что текущий пользователь является владельцем темы, к которой относится запись.

19.5. Защищенный блог: в проекте Blog примите меры к тому, чтобы каждое сообщение в блоге было связано с конкретным пользователем. Убедитесь в том, что чтение всех сообщений доступно всем пользователям, но только зарегистрированные пользователи могут создавать новые и редактировать существующие сообщения. В представлении, в котором пользователи редактируют сообщения, перед обработкой формы убедитесь в том, что редактируемое сообщение принадлежит именно этому пользователю.

Итоги

В этой главе вы научились использовать формы для создания новых тем и записей, а также редактирования существующих данных. Далее мы перешли к реализации системы учетных записей. Вы предоставили существующим пользователям возможность начинать и завершать сеанс работы с приложением, а также научились использовать класс Django `UserCreationForm` для создания новых учетных записей.

После создания простой системы аутентификации и регистрации пользователей вы ограничили доступ пользователей к некоторым страницам; для этого использовался декоратор `@login_required`. Затем данные были связаны с конкретными пользователями при помощи отношения внешнего ключа. Вы также узнали, как выполнить миграцию базы данных, когда миграция требует ввести данные по умолчанию.

В последней части главы вы узнали, как ограничить состав данных, просматриваемых пользователем, с использованием функций представления. Для чтения соответствующих данных использовался метод `filter()`, а владелец запрашиваемых данных сравнивался с текущим пользователем.

Не всегда бывает сразу понятно, какие данные должны быть доступны всем пользователям, а какие данные следует защищать, но этот навык приходит с практикой. Решения, принятые нами в этой главе для защиты данных пользователей, наглядно показывают, почему при построении проекта желательно работать в команде: если кто-то просматривает код вашего проекта, это повышает вероятность выявления плохо защищенных областей.

К настоящему моменту мы построили полностью функциональный проект, работающий на локальной машине. В последней главе мы доработаем оформление приложения Learning Log, чтобы оно выглядело более привлекательно. Также проект будет развернут на сервере, чтобы любой пользователь с доступом к интернету мог зарегистрироваться и создать учетную запись.

20 Оформление и развертывание приложения

Приложение Learning Log уже вполне работоспособно, но оно не имеет стилевого оформления и работает только на локальной машине. В этой главе мы определим для проекта простое, но профессиональное оформление, а затем развернем его на сервере, чтобы любой желающий мог создать учетную запись.

Для стилевого оформления будет использоваться библиотека Bootstrap — набор инструментов для оформления веб-приложений, с которыми они будут выглядеть профессионально на любых современных устройствах, от большого монитора с плоским экраном до смартфона. Для этого мы воспользуемся приложением `django-bootstrap4`, а вы заодно потренируетесь в использовании приложений, созданных другими разработчиками Django.

Для развертывания Learning Log будет использоваться Heroku — сайт, позволяющий загрузить ваш проект на один из его серверов, чтобы сделать его доступным для любого пользователя с подключением к интернету. Также мы начнем пользоваться системой контроля версий Git для отслеживания изменений в проекте.

Когда работа с Learning Log будет завершена, вы будете уметь разрабатывать простые веб-приложения, придавать им качественный внешний вид и развертывать их на работающих серверах. Также по мере накопления опыта вы научитесь пользоваться ресурсами с материалами более высокого уровня.

Оформление приложения Learning Log

До сих пор мы намеренно игнорировали оформление приложения, чтобы сосредоточиться на его функциональности. И это вполне разумный подход к разработке, потому что приложение приносит пользу только в том случае, если оно работает. Конечно, когда приложение начинает работать, оформление выходит на первый план, чтобы пользователи захотели работать с ним.

В этом разделе я кратко опишу приложение `django-bootstrap4` и покажу, как интегрировать его в проект и подготовить к развертыванию.

Приложение django-bootstrap4

Для интеграции Bootstrap в наш проект будет использоваться приложение `django-bootstrap4`. Это приложение загружает необходимые файлы Bootstrap, размещает их в правильных каталогах проекта и предоставляет доступ к стилевым директивам в шаблонах проекта.

Чтобы установить `django-bootstrap4`, введите следующую команду в активной виртуальной среде:

```
(ll_env)learning_log$ pip install django-bootstrap4
...
Successfully installed django-bootstrap4-0.0.7
```

Затем необходимо добавить следующий код для включения `django-bootstrap4` в список `INSTALLED_APPS` в файле settings.py:

settings.py

```
...
INSTALLED_APPS = [
    # Мои приложения
    'learning_logs',
    'users',

    # Сторонние приложения
    'bootstrap4',

    # Приложения django по умолчанию.
    'django.contrib.admin',
    ...
```

Создайте новую секцию для приложений, созданных другими разработчиками, и включите в нее запись `'bootstrap4'`. Проследите за тем, чтобы секция располагалась после секции `# Мои приложения`, но перед секцией, содержащей приложения Django по умолчанию.

Использование Bootstrap для оформления Learning Log

По сути, Bootstrap представляет собой большой набор инструментов стилевого оформления. Также библиотека содержит ряд шаблонов, которые можно применить к проекту для формирования общего стиля. Пользоваться этими шаблонами намного проще, чем отдельными инструментами оформления. Чтобы просмотреть шаблоны, предоставляемые Bootstrap, перейдите по ссылке http://getbootstrap.com/, щелкните на ссылке Examples и найдите раздел Navbars. Мы воспользуемся шаблоном Navbar static, который предоставляет простую панель навигации и контейнер для содержимого страницы.

На рис. 20.1 показано, как будет выглядеть домашняя страница после применения шаблона Bootstrap к base.html и незначительного изменения index.html.

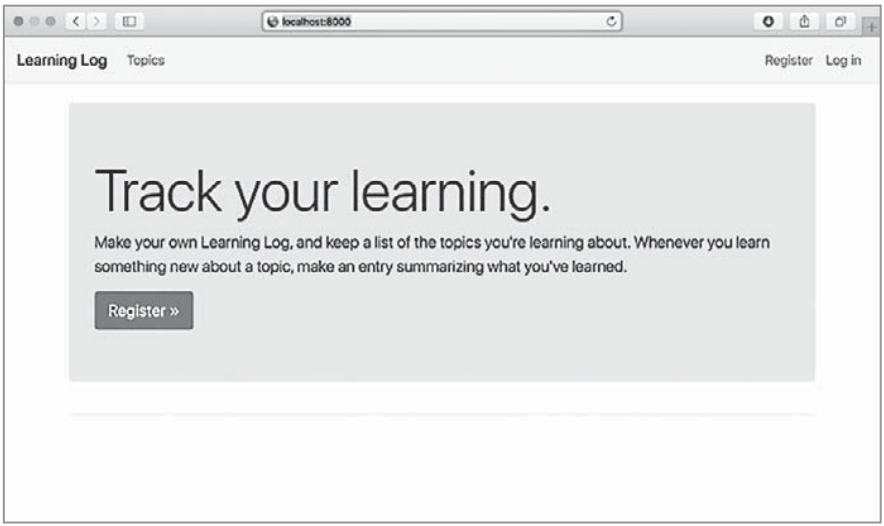

Рис. 20.1. Домашняя страница Learning Log с использованием Bootstrap

Изменение base.html

Шаблон base.html необходимо изменить так, чтобы в нем был задействован шаблон Bootstrap. Новая версия base.html будет представлена в несколько этапов.

Определение заголовков HTML

Первое изменение в base.html: заголовки HTML определяются в файле, чтобы при открытии страницы Learning Log в строке заголовка браузера выводилось имя сайта. Также будут добавлены некоторые требования для использования Bootstrap в шаблонах. Удалите все содержимое base.html и замените его следующим кодом:

base.html

```
❶ {% load bootstrap4 %}

❷ <!doctype html>
❸ <html lang="en">
❹  <head>
     <meta charset="utf-8">
     <meta name="viewport" content="width=device-width, initial-scale=1,
       shrink-to-fit=no">
❺    <title>Learning Log</title>

❻    {% bootstrap_css %}
     {% bootstrap_javascript jquery='full' %}

❼   </head>
```

В точке ❶ загружается коллекция шаблонных тегов из `django-bootstrap4`. Затем файл объявляется как документ HTML ❷, написанный на английском языке ❸. Файл HTML состоит из двух основных частей, заголовка и тела; заголовок файла начинается в точке ❹. Заголовок файла HTML не содержит контента: он всего лишь передает браузеру информацию, необходимую для правильного отображения страницы. В точке ❺ включается элемент `title` страницы; его содержимое будет выводиться в строке заголовка браузера при открытии Learning Log.

В точке ❻ используется один из шаблонных тегов `django-bootstrap4`, который приказывает Django включить все стилевые файлы Bootstrap. Следующий тег активизирует все интерактивное поведение, которое может использоваться на странице, например раздвижные навигационные панели. В точке ❼ располагается закрывающий тег `</head>`.

Определение навигационной панели

Код, определяющий навигационную панель в верхней части страницы, получается довольно длинным, потому что он должен хорошо работать как на узких экранах смартфонов, так и на широких экранах мониторов настольных компьютеров. Мы рассмотрим код навигационной панели по частям.

Первая часть навигационной панели выглядит так:

base.html

```
    ...
    </head>
❶   <body>

❷     <nav class="navbar navbar-expand-md navbar-light bg-light mb-4 border">

❸       <a class="navbar-brand" href="{% url 'learning_logs:index'%}">
            Learning Log</a>

❹       <button class="navbar-toggler" type="button" data-toggle="collapse"
            data-target="#navbarCollapse" aria-controls="navbarCollapse"
            aria-expanded="false" aria-label="Toggle navigation">
          <span class="navbar-toggler-icon"></span></button>
```

Первый элемент — открывающий тег `<body>` ❶. Тело файла HTML содержит контент, который будет виден пользователям на странице. В точке ❷ элемент `<nav>` обозначает раздел навигационных ссылок на странице. Весь контент внутри этого элемента оформляется по правилам Bootstrap, определяемым селекторами `navbar`, `navbar-expand-md` и другим перечисленным здесь. *Селектор* определяет, к каким элементам страницы должно применяться стилевое правило. Селекторы `navbar-light` и `bg-light` оформляют навигационную панель темой со светлым фоном. Сокращение `mb` в `mb-4` происходит от «margin-bottom», то есть «нижнее поле»; этот селектор гарантирует, что между навигационной панелью и остальным контентом страницы остается свободное место. Селектор `border` создает тонкую рамку вокруг светлого фона, чтобы немного отделить его от остального контента страницы.

В точке ❸ задается имя проекта, которое выводится у левого края навигационной панели, и создается ссылка на домашнюю страницу; она будет отображаться на каждой странице проекта. Селектор `navbar-brand` оформляет эту ссылку так, чтобы она выделялась на фоне остальных ссылок; это оформление становится одной из составляющих фирменной символики сайта.

В точке ❹ шаблон определяет кнопку, которая будет отображаться, если ширины окна браузера не хватает для горизонтального отображения всей навигационной панели. Когда пользователь нажимает кнопку, навигационные элементы выводятся в раскрывающемся списке. Атрибут `collapse` сворачивает навигационную панель при уменьшении размеров окна браузера и при отображении сайта на мобильных устройствах с малыми экранами.

Следующая часть кода, определяющего навигационную панель:

base.html

```
        ...
        <span class="navbar-toggler-icon"></span></button>
❶     <div class="collapse navbar-collapse" id="navbarCollapse">
❷       <ul class="navbar-nav mr-auto">
❸         <li class="nav-item">
            <a class="nav-link" href="{% url 'learning_logs:topics'%}">
                Topics</a></li>
        </ul>
```

В точке ❶ открывается новая секция навигационной панели. В процессе построения веб-страницы разработчик делит ее на секции и определяет стили и правила поведения, применяемые к текущей секции. Все стилевые директивы и правила поведения, определяемые в открывающем теге `div`, продолжают действовать до следующего, закрывающего тега `div`, который записывается в виде `</div>`. Это начало той части навигационной панели, которая будет сворачиваться на узких экранах и окнах.

В точке ❷ определяется новый набор ссылок. Bootstrap определяет навигационные элементы как элементы неупорядоченного списка со стилевым оформлением, с которым они совершенно не похожи на список. Каждая ссылка или элемент, который должен отображаться на панели, включается как элемент одного из этих списков. В данном случае единственным элементом списка является ссылка на страницу `Topics` ❸.

Следующая часть навигационной панели:

base.html

```
          ...
      </ul>
❶     <ul class="navbar-nav ml-auto">
❷       {% if user.is_authenticated %}
          <li class="nav-item">
❸           <span class="navbar-text"}">Hello, {{ user.username }}.</span>
          </li>
```

```
          <li class="nav-item">
            <a class="nav-link" href="{% url 'users:logout' %}">Log out</a>
          </li>
        {% else %}
          <li class="nav-item">
            <a class="nav-link" href="{% url 'users:register' %}">Register</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="{% url 'users:login' %}">Log in</a></li>
        {% endif %}
      </ul>
❹   </div>

  </nav>
```

В точке ❶ новый набор ссылок начинается при помощи другого открывающего тега `<ul>`. На странице можно создать столько групп ссылок, сколько вам понадобится. Эта группа содержит ссылки, связанные со входом и регистрацией, расположенные в правой части навигационной панели. Имя селектора `ml-auto` означает «margin-left-automatic», то есть «автоматическое левое поле»; этот селектор анализирует другие элементы на навигационной панели и определяет величину левого поля, которое сдвигает эту группу ссылок к правому краю экрана.

Блок `if` в точке ❷ уже использовался ранее для вывода сообщений для пользователей в зависимости от того, выполнили они вход или нет. На этот раз блок стал немного длиннее, потому что некоторые стилевые правила находятся внутри условных тегов. В точке ❸ расположен элемент `<span>`. Элемент `span` оформляет фрагменты текста или элементы страницы, которые являются частью более длинной строки. Если элементы `div` создают собственный раздел страницы, элементы `span` непрерывно располагаются внутри большего раздела. На первый взгляд такая структура кажется запутанной, потому что многие страницы содержат элементы `div` с большой вложенностью. Здесь элемент `span` используется для оформления текста на навигационной панели — например, имени пользователя, выполнившего вход. Эта информация должна отличаться по внешнему виду от ссылок, чтобы у пользователей не возникало желания щелкать на них.

В точке ❹ закрывается элемент `div` с частями навигационной панели, которые сворачиваются на узких экранах, а в конце секции закрывается навигационная панель в целом. Если вы захотите добавить больше ссылок на навигационную панель, включите другой элемент `<li>` в любую из групп `<ul>`, определенных на навигационной панели; используйте стилевые директивы, идентичные приведенным выше.

Работа с файлом base.html еще не закончена. Необходимо определить два блока, которые могут использоваться отдельными страницами для размещения контента, относящегося к этим страницам.

Определение основного раздела страницы

Оставшаяся часть base.html содержит основной контент страницы:

base.html

```
        ...
    </nav>

❶   <main role="main" class="container">
❷     <div class="pb-2 mb-2 border-bottom">
        {% block page_header %}{% endblock page_header %}
      </div>
❸     <div>
        {% block content %}{% endblock content %}
      </div>
    </main>

  </body>

  </html>
```

В точке ❶ открывается тег `<main>`. Элемент `main` используется для основного контента в теле страницы. В данном случае назначается селектор `container`, что является простым способом группировки элементов на странице. В этот контейнер будут включены два элемента `div`.

Первый элемент `div` ❷ содержит блок `page_header`. Мы будем использовать этот блок для отображения заголовка большинства страниц. Чтобы секция выделялась на фоне других частей страницы, мы разместим отступы под заголовком. *Отступ* (padding) представляет собой пространство между контентом элемента и его границей. Селектор `pb-2` — директива Bootstrap, создающая отступы умеренной величины в нижней части оформляемого элемента. *Полем* (margin) называется пространство между границей элемента и другими элементами страницы. В нашем приложении граница нужна только в нижней части страницы, поэтому мы используем селектор `border-bottom`, создающий тонкую границу в нижней части блока `page_header`.

В точке ❸ определяется еще один элемент `div`, содержащий блок `content`. К этому блоку не применяется никакой конкретный стиль, поэтому контент любой страницы можно оформить так, как вы считаете нужным для этой страницы. Файл base.html завершается закрывающими тегами для элементов `main`, `body` и `html`.

Загрузив домашнюю страницу Learning Log в браузере, вы увидите профессиональную навигационную панель, изображенную на рис. 20.1. Попробуйте изменить размеры окна, заметно уменьшив его ширину; навигационная панель должна превратиться в кнопку. Щелкните на кнопке, и все ссылки появятся в раскрывающемся списке.

Оформление домашней страницы с использованием табло

Изменим домашнюю страницу при помощи нового блока `header` и другого элемента Bootstrap, так называемого *джамботронома* — большого блока, который выдается на общем фоне страницы и может содержать любую информацию на ваше усмотре-

ние. Обычно этот элемент используется на домашних страницах для размещения краткого описания проекта.

Обновленный файл index.html выглядит так:

index.html

```
  {% extends "learning_logs/base.html" %}

❶ {% block page_header %}
❷   <div class='jumbotron'>
❸     <h1 class="display-3">Track your learning.</h1>

❹     <p class="lead">Make your own Learning Log, and keep a list of the
          topics you're learning about. Whenever you learn something new
          about a topic, make an entry summarizing what you've learned.</p>

❺     <a class="btn btn-lg btn-primary" href="{% url 'users:register' %}"
          role="button">Register &raquo;</a>
    </div>
❻ {% endblock page_header %}
```

В точке ❶ мы сообщаем Django о том, что далее следует определение содержимого блока `page_header`. Элемент `jumbotron` ❷ представляет собой обычный элемент `div`, к которому применяется набор стилевых директив. Селектор `jumbotron` применяет эту группу стилевых директив из библиотеки Bootstrap к элементу.

Внутри элемента `jumbotron` содержатся три элемента. Первый — короткое сообщение `Track your learning`, которое дает новым посетителям представление о том, что делает Learning Log. Класс `h1` является заголовком первого уровня, а с селектором `display-3` заголовок становится более тонким и высоким ❸. В точке ❹ включается более длинное сообщение с дополнительной информацией о том, что пользователь может сделать со своим дневником.

Вместо простой текстовой ссылки мы создаем кнопку ❺, которая предлагает пользователю зарегистрировать свою учетную запись Learning Log. Это та же ссылка, что и в заголовке, но кнопка выделяется на фоне страницы и показывает пользователю, что необходимо сделать для того, чтобы приступить к работе над проектом. В результате применения селекторов получается крупная кнопка, представляющая желаемое действие. Код `&raquo;` является *сущностью HTML*, представляющей две правые угловые скобки (>>). В точке ❻ блок `page_header` закрывается. Контент на эту страницу добавляться не будет, поэтому определять блок `content` на этой странице не нужно.

Теперь страница выглядит так, как на рис. 20.1. Она смотрится намного лучше по сравнению с проектом без применения оформления.

Оформление страницы входа

Мы усовершенствовали внешний вид страницы входа, но формы входа изменения пока не коснулись. Приведем внешний вид формы в соответствие с остальными элементами страницы:

login.html

```
  {% extends "learning_logs/base.html" %}
❶ {% load bootstrap4 %}

❷ {% block page_header %}
    <h2>Log in to your account.</h2>
  {% endblock page_header %}

  {% block content %}
❸   <form method="post" action="{% url 'users:login' %}" class="form">
      {% csrf_token %}
❹     {% bootstrap_form form %}
❺     {% buttons %}
        <button name="submit" class="btn btn-primary">Log in</button>
      {% endbuttons %}

      <input type="hidden" name="next"
        value="{% url 'learning_logs:index' %}" />
    </form>

  {% endblock content %}
```

В точке ❶ в шаблон загружаются шаблонные теги `bootstrap4`. В точке ❷ определяется блок `page_header`, который описывает, для чего нужна страница. Обратите внимание: блок `{% if form.errors %}` удален из шаблона; `django-bootstrap4` управляет ошибками формы автоматически.

В точке ❸ добавляется атрибут `class="form"`, после чего при отображении формы ❹ используется шаблонный тег `{% bootstrap_form %}`; он заменяет тег `{{ form.as_p }}`, используемый в главе 19. Шаблонный тег `{% booststrap_form %}` вставляет правила в стиле Bootstrap в отдельные элементы формы при ее построении. В точ-

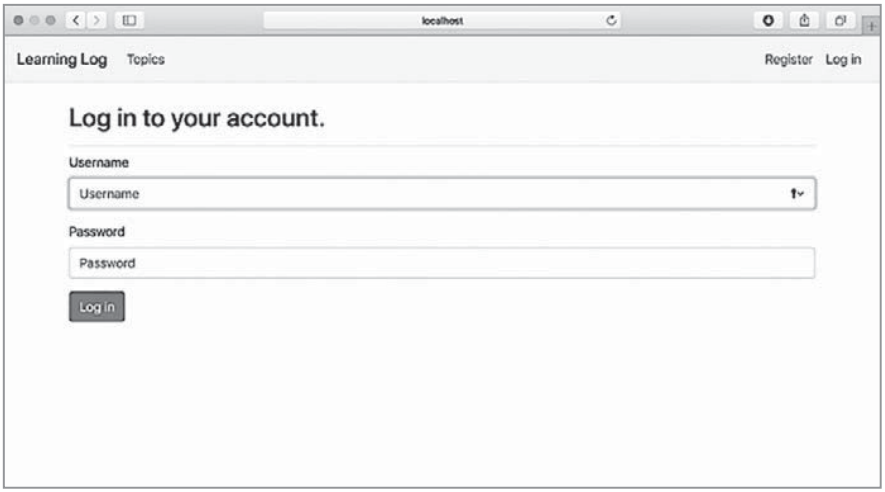

Рис. 20.2. Страница входа, оформленная с использованием Bootstrap

ке ❺ открывается шаблонный тег `bootstrap4 {% buttons %}`, который добавляет стилевое оформление Bootstrap к кнопкам.

На рис. 20.2 показана форма входа так, как она выглядит сейчас. Страница стала намного чище, ее оформление последовательно, а предназначение предельно ясно. Попробуйте выполнить вход с неверным именем пользователя или паролем; вы увидите, что даже сообщения об ошибках следуют тому же стилю оформления и хорошо интегрируются с сайтом в целом.

Оформление страницы со списком тем

А теперь позаботимся о том, чтобы страницы для просмотра информации также были выдержаны в том же стиле. Начнем со страницы со списком тем:

topics.html

```
  {% extends "learning_logs/base.html" %}

❶ {% block page_header %}
    <h1>Topics</h1>
  {% endblock page_header %}

  {% block content %}
    <ul>
      {% for topic in topics %}
❷      <li><h3>
          <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
        </h3></li>
      {% empty %}
        <li><h3>No topics have been added yet.</h3></li>
      {% endfor %}
    </ul>

❸   <h3><a href="{% url 'learning_logs:new_topic' %}">Add new topic</h3>
  {% endblock content %}
```

Тег `{% load bootstrap4 %}` не нужен, потому что в этом файле не используются никакие шаблонные теги `bootstrap4`. Заголовок `Topics` перемещается в блок `page_header`, и вместо простого абзацного тега ему назначается оформление заголовка ❶. Каждая тема оформляется как элемент `<h3>`, чтобы она выводилась чуть более крупным шрифтом на странице ❷; то же самое делается со ссылкой добавления новой темы ❸.

Оформление записей на странице темы

Страница темы содержит больше контента, чем большинство страниц, поэтому над ней придется потрудиться. Чтобы записи визуально выделялись, мы воспользуемся панелями Bootstrap. *Панель* (card) представляет собой элемент `div` с заранее определенным гибким стилем и идеально подходит для отображения записей темы:

topic.html

```
  {% extends 'learning_logs/base.html' %}

❶ {% block page_header %}
    <h3>{{ topic }}</h3>
  {% endblock page_header %}

  {% block content %}
    <p>
  <a href="{% url 'learning_logs:new_entry' topic.id %}">Add new entry</a>
    </p>

    {% for entry in entries %}
❷     <div class="card mb-3">
❸       <h4 class="card-header">
          {{ entry.date_added|date:'M d, Y H:i' }}
❹         <small><a href="{% url 'learning_logs:edit_entry' entry.id %}">
              edit entry</a></small>
        </h4>
❺       <div class="card-body">
          {{ entry.text|linebreaks }}
        </div>
      </div>
    {% empty %}
      <p>There are no entries for this topic yet.</p>
    {% endfor %}

  {% endblock content %}
```

Сначала тема размещается в блоке `page_header` ❶. Затем удаляется структура неупорядоченного списка, использовавшаяся ранее в этом шаблоне. Вместо того чтобы превращать каждую запись в элемент списка, мы создаем в точке ❷ элемент с селектором `card`. Он имеет два вложенных элемента: первый предназначен для хранения временной метки и ссылки для редактирования, а второй — для хранения тела записи.

Первый элемент в панели представляет собой заголовок — элемент `<h4>` с селектором `card-header` ❸. Заголовок панели содержит дату создания записи и ссылку для ее редактирования. Ссылка `edit_entry` заключается в тег `<small>`, чтобы она была чуть меньше временной метки ❹. Второй элемент представляет собой `div` с селектором `card-body` ❺, который размещает текст записи в простом поле на карте. Обратите внимание: код Django для включения информации на страницу не изменился; изменились только элементы, влияющие на внешний вид страницы.

На рис. 20.3 изображена страница темы с новым оформлением. Функциональность приложения Learning Log не изменилась, но приложение выглядит более привлекательно и заманчиво для пользователя.

ПРИМЕЧАНИЕ Если вы хотите использовать другой шаблон Bootstrap, действуйте в той же последовательности, которая уже использовалась в этой главе. Скопируйте шаблон в base.html и измените элементы, содержащие контент, чтобы шаблон отображал информацию вашего проекта. Затем воспользуйтесь средствами индивидуального стилевого оформления Bootstrap для оформления содержимого каждой страницы.

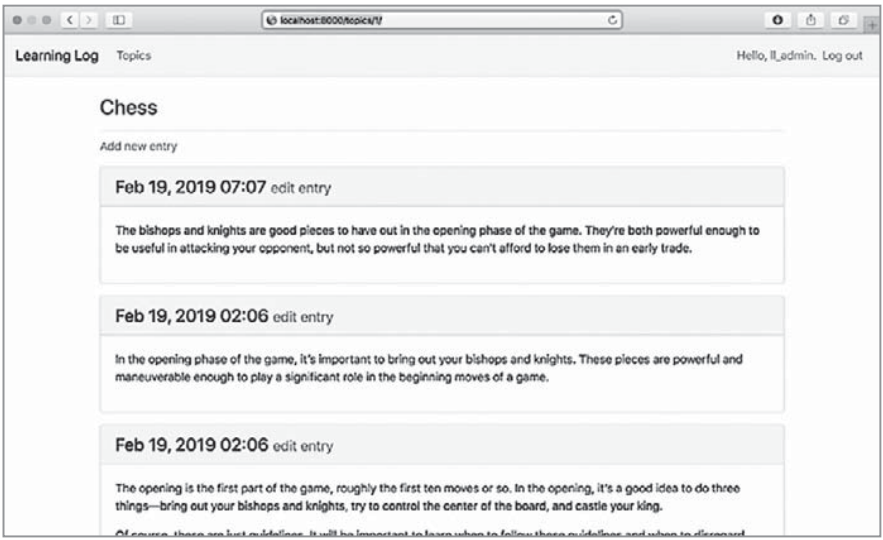

Рис. 20.3. Страница темы с оформлением Bootstrap

УПРАЖНЕНИЯ

20.1. Другие формы: мы применили стили Bootstrap к странице `login`. Внесите аналогичные изменения в другие страницы на базе форм: `new_topic`, `new_entry`, `edit_entry` и `register`.

20.2. Стилевое оформление Blog: используйте Bootstrap для стилевого оформления проекта Blog из главы 19.

Развертывание Learning Log

После того как проекту был придан профессиональный вид, мы развернем его на реальном сервере, чтобы любой пользователь с подключением к интернету мог работать с приложением. Мы воспользуемся Heroku — веб-платформой, позволяющей управлять развертыванием веб-приложений.

Создание учетной записи Heroku

Чтобы создать учетную запись, откройте сайт https://heroku.com/ и щелкните на одной из регистрационных ссылок. Учетные записи создаются бесплатно, и Heroku предоставляет бесплатный уровень для тестирования проектов в реальных условиях.

ПРИМЕЧАНИЕ На бесплатном уровне Heroku существуют свои ограничения (например, количество приложений, которые можно развернуть, и частота посещения приложения пользователями). Впрочем, эти ограничения достаточно либеральны, чтобы вы могли потренироваться в развертывании приложений без каких-либо затрат.

Установка инструментария Heroku CLI

Чтобы развернуть проект на серверах Heroku и управлять им, вам понадобятся инструменты из пакета Heroku Command Line Interface (CLI). Чтобы установить новейшую версию Heroku CLI, откройте сайт https://devcenter.heroku.com/articles/heroku-cli/ и выполните указания для своей операционной системы. В них содержится либо однострочная терминальная команда, либо программа установки, которую вы можете загрузить и запустить.

Установка необходимых пакетов

Вам также придется установить три пакета, упрощающих работу проектов Django на реальных серверах. В активной виртуальной среде введите следующие команды:

```
(ll_env)learning_log$ pip install psycopg2==2.7.*
(ll_env)learning_log$ pip install django-heroku
(ll_env)learning_log$ pip install gunicorn
```

Пакет `psycopg2` необходим для управления базой данных, используемой Heroku. Пакет `django-heroku` почти полностью управляет конфигурацией, необходимой нашему приложению для нормальной работы на серверах Heroku. К этой категории относится управление базой данных и хранение статических файлов, при котором они будут правильно предоставляться по запросу. Пакет `gunicorn` — сервер, способный предоставлять доступ к приложениям в реальной среде.

Создание файла requirements.txt

Heroku необходимо знать, от каких пакетов зависит наш проект, поэтому мы воспользуемся pip для построения файла со списком. Оставаясь в активной виртуальной среде, введите следующую команду:

```
(ll_env)learning_log$ pip freeze > requirements.txt
```

Команда `freeze` приказывает pip записать имена всех пакетов, в настоящее время установленных в системе, в файл requirements.txt. Откройте файл requirements.txt и просмотрите пакеты и номера версий, установленных в вашей системе:

requirements.txt

```
dj-database-url==0.5.0
Django==2.2.0
django-bootstrap4==0.0.7
django-heroku==0.3.1
gunicorn==19.9.0
psycopg2==2.7.7
pytz==2018.9
sqlparse==0.2.4
whitenoise==4.1.2
```

Приложение Learning Log уже зависит от восьми разных пакетов с конкретными номерами версий, поэтому для его правильной работы требуется конкретная конфигурация среды. (Мы установили четыре из этих пакетов вручную, а четыре были установлены автоматически как зависимости этих пакетов.)

При развертывании Learning Log Heroku устанавливает все пакеты, перечисленные в requirements.txt, и создает среду с теми же пакетами, которые мы используем локально. По этой причине разработчик может быть уверен в том, что развернутый проект будет работать точно так же, как в его локальной системе. Вы поймете, насколько это полезно, когда начнете строить и вести в своей системе несколько разных проектов.

ПРИМЕЧАНИЕ Если пакет включен в список в вашей системе, но номер версии отличается от показанного, используйте версию, установленную в вашей системе.

Назначение исполнительной среды Python

Если вы не укажете версию Python, то Heroku будет использовать собственную версию Python по умолчанию. Убедитесь в том, что Heroku использует ту же версию Python, которая установлена у вас. В активной виртуальной среде введите команду `python --version`:

```
(ll_env)learning_log$ python --version
Python 3.7.2
```

В этом примере я использую Python 3.7.2. Создайте новый файл с именем runtime.txt в одном каталоге с файлом manage.py и введите следующую команду:

runtime.txt

```
python-3.7.2
```

Этот файл должен содержать одну строку с версией Python, заданной точно в показанном формате: `python` в нижнем регистре, затем дефис и номер версии из трех частей.

ПРИМЕЧАНИЕ Если вы получите сообщение об ошибке, в котором сказано, что запрашиваемая исполнительная среда Python недоступна, откройте страницу https://devcenter.heroku.com/categories/languagesupport/ и найдите ссылку Specifying a Python Runtime. Просмотрите текст статьи, найдите доступные варианты исполнительной среды и выберите тот вариант, который ближе всего к вашей версии Python.

Изменение файла settings.py для Heroku

Затем в конец файла settings.py необходимо добавить раздел для определения настроек, предназначенных конкретно для среды Heroku:

settings.py

```
...
# Мои настройки
LOGIN_URL = 'users:login'

# Настройки Heroku
import django_heroku
django_heroku.settings(locals())
```

Здесь импортируется модуль `django_heroku` и вызывается функция `settings()`. Эта функция изменяет некоторые настройки, которые должны иметь определенные значения для среды Heroku.

Создание файла Procfile для запуска процессов

Файл *Procfile* сообщает Heroku, какие процессы должны запускаться для правильной работы проекта. Это однострочный файл, который должен быть сохранен под именем Procfile (символ `P` верхнего регистра, без расширения) в одном каталоге с файлом manage.py.

Содержимое Procfile выглядит так:

Procfile

```
web: gunicorn learning_log.wsgi --log-file -
```

Эта строка приказывает Heroku использовать для приложения сервер `gunicorn`, а при запуске приложения загрузить настройки из файла learning_log/wsgi.py. Флаг log-file сообщает Heroku, какие события должны регистрироваться в журнале.

Использование Git для управления файлами проекта

Как упоминалось в главе 17, Git — программа контроля версий, которая позволяет создать «мгновенный снимок» состояния кода проекта при реализации каждой новой функции. Это позволяет легко вернуться к последнему работоспособному состоянию проекта при возникновении каких-либо проблем (например, если в ходе работы над новой функцией была случайно внесена ошибка).

Если в вашем проекте используется Git, это означает, что вы можете работать над новой функциональностью, не беспокоясь о том, чтобы ничего не нарушить. Развертывая проект на сервере, вы должны убедиться в том, что вы развертываете работоспособную версию проекта. Если вы захотите больше узнать о Git и контроле версий, обращайтесь к приложению Г.

Установка Git

Возможно, пакет Git уже установлен в вашей системе. Чтобы проверить это, откройте новое терминальное окно и введите команду `git --version`:

```
(ll_env)learning_log$ git --version
git version 2.17.0
```

Если вы получите сообщение об ошибке, обращайтесь к инструкциям по установке Git в приложении Г.

Настройка Git

Git следит за тем, кто внес изменения в проект, даже в том случае, если над проектом работает только один человек. Для этого Git необходимо знать ваше имя пользователя и адрес электронной почты. Имя пользователя ввести обязательно, но ничто не мешает вам ввести вымышленный адрес электронной почты для учебных проектов:

```
(ll_env)learning_log$ git config --global user.name "ehmatthes"
(ll_env)learning_log$ git config --global user.email "eric@example.com"
```

Если вы забудете об этом шаге, Git запросит у вас необходимую информацию при первом закреплении.

Игнорирование файлов

Нам не нужно, чтобы система Git отслеживала все файлы в проекте, поэтому мы прикажем Git игнорировать некоторые файлы. Создайте файл с именем .gitignore в папке с файлом manage.py. Обратите внимание: имя файла начинается с точки, а файл не имеет расширения. Содержимое `.gitignore` выглядит так:

.gitignore

```
ll_env/
__pycache__/
*.sqlite3
```

Мы приказываем Git игнорировать весь каталог ll_env, потому что можем автоматически воссоздать его в любой момент. Также в системе контроля не отслеживается каталог __pycache__ с файлами .pyc, которые создаются автоматически, когда Django выполняет файлы .py. Мы не отслеживаем изменения в локальной базе данных, потому что так поступать вообще нежелательно: если на сервере будет использоваться SQLite, вы можете случайно переписать «живую» базу данных локальной тестовой базой данных при отправке проекта на сервер. Звездочка в выражении `*.sqlite3` приказывает Git игнорировать любые файлы с расширением `.sqlite3`.

ПРИМЕЧАНИЕ Если вы используете macOS, добавьте `.DS_Store` в файл .gitignore. В этом файле хранится информация о настройках папок в macOS, и он не имеет никакого отношения к этому проекту.

Отображение скрытых файлов

Во многих операционных системах файлы и папки, имена которых начинаются с точки (например, .gitignore), скрываются. Когда вы откроете программу просмотра файлов или попытаетесь открыть файл из такого приложения, как Sublime Text, по умолчанию эти файлы не отображаются. Тем не менее вам как программисту нужно их видеть. Ниже описаны способы отображения скрытых файлов в зависимости от операционной системы:

- В системе Windows откройте Проводник Windows и папку (например, Рабочий стол). Щелкните на вкладке Вид и проверьте состояние параметров Показывать скрытые файлы, папки и диски и Скрывать расширения для зарегистрированных типов файлов.
- В macOS нажмите you can press CommandShift. (точка) в любой программе просмотра файлов, чтобы увидеть скрытые файлы и папки.
- В системах Linux (таких, как Ubuntu) нажмите Ctrl+H в любой программе просмотра файлов, чтобы увидеть скрытые файлы и папки. Чтобы этот режим был включен постоянно, откройте программу просмотра файлов (например, Nautilus) и щелкните на вкладке параметров (обозначается тремя линиями). Установите флажок Показывать скрытые файлы.

Закрепление состояния проекта

Чтобы инициализировать репозиторий Git для Learning Log, добавьте все необходимые файлы в репозиторий и закрепите исходное состояние проекта. Вот как это делается:

```
❶ (ll_env)learning_log$ git init
  Initialized empty Git repository in /home/ehmatthes/pcc/learning_log/.git/
❷ (ll_env)learning_log$ git add .
❸ (ll_env)learning_log$ git commit -am "Ready for deployment to heroku."
  [master (root-commit) 79fef72] Ready for deployment to heroku.
   45 files changed, 712 insertions(+)
   create mode 100644 .gitignore
   create mode 100644 Procfile
   ...
   create mode 100644 users/views.py
❹ (ll_env)learning_log$ git status
  On branch master
  nothing to commit, working tree clean
  (ll_env)learning_log$
```

В точке ❶ вводится команда `git init`, которая инициализирует пустой репозиторий в каталоге, содержащем Learning Log. В точке ❷ команда `git add .` добавляет все файлы (кроме игнорируемых) в репозиторий (не забудьте точку). В точке ❸ вводится команда `git commit -am` *сообщение*: флаг `-a` приказывает Git включить все измененные файлы в закрепленное состояние, а флаг `-m` приказывает Git сохранить сообщение в журнале.

Вывод команды `git status` ❹ сообщает, что текущей является главная ветвь, а рабочий каталог пуст. Этот статус должен выводиться каждый раз, когда вы отправляете свой проект на Heroku.

Отправка проекта

Наконец-то все готово для отправки проекта на сервер Heroku. В активном терминальном сеансе введите следующие команды:

```
❶ (ll_env)learning_log$ heroku login
  heroku: Press any key to open up the browser to login or q to exit:
  Logging in... done
  Logged in as eric@example.com
❷ (ll_env)learning_log$ heroku create
  Creating app... done, * secret-lowlands-82594
  https://secret-lowlands-82594.herokuapp.com/ |
      https://git.heroku.com/secret-lowlands-82594.git
❸ (ll_env)learning_log$ git push heroku master
  ...
  remote: -----> Launching...
  remote:        Released v5
❹ remote:        https://secret-lowlands-82594.herokuapp.com/ deployed to Heroku
  remote: Verifying deploy... done.
  To https://git.heroku.com/secret-lowlands-82594.git
   * [new branch]      master -> master
  (ll_env)learning_log$
```

Сначала войдите на сервер Heroku в терминальном сеансе с именем пользователя и паролем, использованными при создании учетной записи ❶. Затем прикажите Heroku построить пустой проект ❷. Heroku генерирует имя, состоящее из двух слов и числа; позднее вы сможете его изменить. Затем вводится команда `git push heroku master` ❸, которая приказывает Git отправить главную ветвь проекта в репозиторий, только что созданный Heroku. Затем Heroku строит проект на своих серверах с использованием этих файлов. В точке ❹ указывается URL, который будет использоваться для обращения к развернутому проекту.

После ввода этих команд проект развернут, но еще не настроен полностью. Чтобы проверить, что серверный процесс был запущен правильно, введите команду `heroku ps`:

```
  (ll_env)learning_log$ heroku ps
❶ Free dyno hours quota remaining this month: 450h 44m (81%)
  Free dyno usage for this app: 0h 0m (0%)
  For more information on dyno sleeping and how to upgrade, see:
  https://devcenter.heroku.com/articles/dyno-sleeping
❷ === web (Free): gunicorn learning_log.wsgi --log-file - (1)
  web.1: up 2019/02/19 23:40:12 -0900 (~ 10m ago)
  (ll_env)learning_log$
```

В выходных данных указано, сколько еще времени проект может оставаться активным в следующем месяце ❶. На момент написания книги Heroku позволяет

бесплатно развернутым проектам оставаться активными до 550 часов за месяц. При нарушении этого лимита отображается стандартная серверная страница ошибки; вскоре мы изменим ее. В точке ❷ запускается процесс, определенный в Procfile.

Теперь мы можем открыть приложение в браузере командой `heroku open`:

```
(ll_env)learning_log$ heroku open
(ll_env)learning_log$
```

Команда избавляет вас от необходимости открывать браузер и вводить URL, который вы получили от Heroku, но при желании сайт можно открыть и так. Вы увидите домашнюю страницу Learning Log с правильным оформлением. Впрочем, с приложением еще работать нельзя, потому что база данных не подготовлена.

ПРИМЕЧАНИЕ Процесс развертывания приложений на серверах Heroku время от времени изменяется. Если у вас возникнут неразрешимые проблемы, обращайтесь к документации Heroku за помощью. Откройте страницу https://devcenter.heroku.com/ и щелкните на ссылке Python, затем найдите ссылку Get Started with Python или Deploying Python and Django Apps on Heroku. Если вы не понимаете то, что там написано, обратитесь к рекомендациям в приложении B.

Подготовка базы данных в Heroku

Вы должны выполнить команду `migrate`, чтобы подготовить базу данных и применить все миграции, сгенерированные в ходе разработки. Для выполнения команд Django и Python в проектах Heroku используется команда `heroku run`. Пример выполнения команды `migrate` в среде разработки Heroku:

```
❶ (ll_env)learning_log$ heroku run python manage.py migrate
❷ Running 'python manage.py migrate' on * secret-lowlands-82594... up, run.3060
    ...
❸ Running migrations:
    ...
    Applying learning_logs.0001_initial... OK
    Applying learning_logs.0002_entry... OK
    Applying learning_logs.0003_topic_owner... OK
    Applying sessions.0001_initial... OK
  (ll_env)learning_log$
```

Сначала мы вводим команду `heroku run python manage.py migrate` ❶. Heroku создает терминальный сеанс для выполнения команды `migrate` ❷. В точке ❸ Django применяет миграции по умолчанию, а также миграции, сгенерированные в ходе разработки Learning Log.

Теперь при обращении к развернутому приложению вы сможете использовать его так же, как это делалось в локальной системе. Однако при этом вы не увидите никаких данных, введенных при локальном развертывании, потому что мы не скопировали данные на сервер. Это обычная практика: локальные данные почти никогда не копируются на сервер, потому что они чаще всего являются тестовыми.

Если вы перешлете ссылку на Heroku, получатель сможет работать с вашей версией приложения Learning Log. В следующем разделе мы выполним еще несколько операций, чтобы завершить процесс развертывания и подготовиться к дальнейшей разработке Learning Log.

Доработка развернутого приложения

В этом разделе мы доработаем развернутое приложение и создадим суперпользователя (так же, как это было сделано в локальной версии). Заодно мы повысим уровень защиты проекта, переведя настройку отладочного режима `DEBUG` в состояние `False`, чтобы пользователи не получали в сообщениях об ошибке дополнительной информации, которая может использоваться для проведения атак на сервер.

Создание суперпользователя в Heroku

Вы уже видели, что для выполнения одиночных команд может использоваться команда `heroku run`. Однако команды также можно выполнять, открыв терминальный сеанс Bash при подключении к серверу Heroku командой `heroku run bash`. Bash — язык, который работает во многих терминалах Linux. Мы используем терминальный сеанс Bash для создания суперпользователя, чтобы иметь возможность обратиться к административному сайту в развернутом приложении:

```
  (ll_env)learning_log$ heroku run bash
  Running 'bash' on * secret-lowlands-82594... up, run.9858
❶ ~ $ ls
  learning_log learning_logs manage.py Procfile requirements.txt runtime.txt
  staticfiles users
❷ ~ $ python manage.py createsuperuser
  Username (leave blank to use 'u47318'): ll_admin
  Email address:
  Password:
  Password (again):
  Superuser created successfully.
❸ ~ $ exit
  exit
  (ll_env)learning_log$
```

В точке ❶ команда `ls` выводит информацию о файлах и каталогах, существующих на сервере; это те же файлы, которые присутствуют в нашей локальной системе. В этой файловой системе можно перемещаться так же, как и в любой другой.

ПРИМЕЧАНИЕ Пользователи Windows должны использовать те же команды (например, `ls` вместо `dir`), потому что они работают с терминалом Linux через удаленное подключение.

В точке ❷ выполняется команда создания суперпользователя. Она выдает тот же вывод, который был получен в локальной системе при создании суперпользователя в главе 18. После того как создание суперпользователя в терминальном сеансе

будет завершено, введите команду `exit` для возвращения к терминальному сеансу локальной системы ❸.

Теперь вы можете добавить /admin/ в конец URL-адреса развернутого приложения, чтобы войти на административный сайт. У меня этот URL-адрес имеет вид https://secret-lowlands-82594.herokuapp.com/admin/.

Если другие пользователи уже начали работать с вашим проектом, учтите, что вам будут доступны все их данные! Относитесь к конфиденциальности серьезно, и пользователи начнут доверять вам свои данные.

Создание удобного URL-адреса на Heroku

Вероятно, вы бы предпочли использовать более удобные и запоминающиеся URL-адреса, чем https://secret-lowlands-82594.herokuapp.com/. Чтобы переименовать приложение, достаточно одной команды:

```
(ll_env)learning_log$ heroku apps:rename learning-log
Renaming secret-lowlands-82594 to learning-log-2e... done
https://learning-log.herokuapp.com/ | https://git.heroku.com/learning-log.git
Git remote heroku updated
 *  Don't forget to update git remotes for all other local checkouts of the app.
(ll_env)learning_log$
```

Имя может содержать буквы, цифры и дефисы и выбирается произвольно (при условии, что никто другой еще не занял это имя). Развернутая версия теперь доступна по адресу https://learning-log.herokuapp.com/. По предыдущему URL-адресу проект теперь недоступен; команда `apps:rename` полностью перемещает проект на новый URL-адрес.

ПРИМЕЧАНИЕ При развертывании проекта на бесплатном сервисе Heroku переводит развернутое приложение в спящий режим, если оно не получало запросов в течение определенного периода времени или было слишком активным для бесплатного уровня. При первом обращении к сайту после перехода в спящий режим загрузка займет больше времени, но последующие запросы будут обрабатываться быстрее. Такой подход позволяет Heroku предоставлять бесплатное развертывание.

Безопасность проекта

В текущем варианте развертывания проекта существует одна очевидная проблема: настройка `DEBUG=True` в файле settings.py, включающая вывод отладочных сообщений при возникновении ошибок. Страницы ошибок Django предоставляют критическую отладочную информацию при разработке проекта, но они также дают слишком много информации хакерам, если оставить их включенными на рабочем сервере.

Для управления выводом отладочной информации на работающем сайте используется *переменная среды*, то есть значение, связанное с конкретной рабочей средой.

Это один из способов хранения конфиденциальной информации на сервере, при котором она отделяется от остального кода проекта.

Изменим файл settings.py, чтобы он проверял переменную среды при запуске проекта на Heroku:

settings.py

```
...
# Настройки Heroku
import django_heroku
django_heroku.settings(locals())

if os.environ.get('DEBUG') == 'TRUE':
    DEBUG = True
elif os.environ.get('DEBUG') == 'FALSE':
    DEBUG = False
```

Метод `os.environ.get()` читает значение, связанное с заданной переменной среды в любом окружении, в котором выполняется проект. Если запрашиваемая переменная задана, метод возвращает ее значение; если она не задана, метод возвращает `None`. Использование переменной среды для хранения логических значений может привести к путанице. В большинстве случаев переменные среды хранятся в строках, и вы должны действовать внимательно. Рассмотрим следующий фрагмент простого терминального сеанса Python:

```
>>> bool('False')
True
```

В логическом контексте строка `'False'` интерпретируется как `True`, потому что любая непустая строка интерпретируется как `True`. По этой причине мы использовали строки `'TRUE'` и `'FALSE'`, записанные в верхнем регистре, чтобы четко показать, что мы не сохраняем логические значения `True` и `False` языка Python. Когда Django читает переменную среды с ключом `'DEBUG'` в Heroku, мы присваиваем `DEBUG` значение `True`, если переменная содержит `'TRUE'`, или значение `False`, если переменная содержит `'FALSE'`.

Закрепление и отправка изменений

Теперь изменения, внесенные в settings.py, необходимо закрепить в репозитории Git, а затем отправить их на Heroku. Следующий терминальный сеанс показывает, как это делается:

```
❶ (ll_env)learning_log$ git commit -am "Set DEBUG based on environment variables."
  [master 3427244] Set DEBUG based on environment variables.
   1 file changed, 4 insertions(+)
❷ (ll_env)learning_log$ git status
  # On branch master
  nothing to commit, working directory clean
  (ll_env)learning_log$
```

Мы вводим команду `git commit` с коротким, но содержательным сообщением ❶. Напомню, что флаг `-am` обеспечивает закрепление всех изменившихся файлов и регистрацию сообщения в журнале. Git видит, что изменился один файл, и закрепляет изменение в репозитории.

В точке ❷ из статусной информации видно, что мы работаем с главной ветвью репозитория, а новые изменения для закрепления отсутствуют. Очень важно проверять статусную информацию перед отправкой на Heroku. Если вы не видите сообщение, значит, некоторые изменения не были закреплены и они не будут отправлены на сервер. Попробуйте снова ввести команду `commit`, а если вы не уверены в том, как решить проблему, — прочитайте приложение Г, чтобы лучше понять, как работать с Git.

Теперь отправим обновленный репозиторий на Heroku:

```
(ll_env)learning_log$ git push heroku master
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing requirements with pip
...
remote: -----> Launching...
remote:        Released v6
remote:        https://learning-log.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/learning-log.git
   144f020..d5075a1  master -> master
(ll_env)learning_log$
```

Heroku видит, что репозиторий обновился, и заново строит проект, чтобы все изменения были учтены. База данных при этом не воссоздается, поэтому выполнять `migrate` для этого обновления не придется.

Назначение переменных среды в Heroku

Теперь можно задать нужное значение `DEBUG` в файле settings.py через Heroku. Команда `heroku config:set` задает значение переменной за нас:

```
(ll_env)learning_log$ heroku config:set DEBUG='FALSE'
Setting DEBUG and restarting * learning-log... done, v7
DEBUG: FALSE
(ll_env)learning_log$
```

Каждый раз, когда вы задаете переменную среды в Heroku, проект автоматически перезапускается, чтобы переменная среды могла начать действовать.

Чтобы убедиться в том, что развернутое приложение стало более безопасным, введите URL проекта с путем, который не был определен. Например, попробуйте открыть страницу http://learning-log.herokuapp.com/letmein/. Должна появиться обобщенная страница, которая не сообщает никакой конкретной информации о проекте.

Если же вы попробуете применить тот же запрос с локальной версией Learning Log по адресу http://localhost:8000/letmein/, должна появиться полная страница ошибки Django. Получается именно то, что нужно: вы будете получать содержательные сообщения об ошибках в процессе разработки проекта, но оградите пользователей от критической информации о коде проекта.

Если вы только развертываете приложение и проводите диагностику исходного развертывания, выполните команду `heroku config:set DEBUG='TRUE'`, и вы временно сможете просматривать полный отчет об ошибке для работающего сайта. Только не забудьте вернуть переменной значение `'FALSE'`, завершив диагностику. Также постарайтесь не делать этого после того, как пользователи начнут регулярно обращаться к вашему сайту.

Создание специализированных страниц ошибок

В главе 19 мы настроили приложение Learning Log так, чтобы при запросе темы или записи, которая ему не принадлежит, пользователь получал ошибку 404. Вероятно, вы также сталкивались с примерами ошибок 500 (внутренние ошибки). Ошибка 404 обычно означает, что код Django правилен, но запрашиваемый объект не существует; ошибка 500 обычно означает, что в написанном вами коде существует ошибка (например, ошибка в функции из views.py). В настоящее время Django возвращает одну обобщенную страницу ошибки в обеих ситуациях, но мы можем написать собственные шаблоны страниц ошибок 404 и 500, которые соответствуют общему оформлению Learning Log. Эти шаблоны должны находиться в корневом каталоге шаблонов.

Создание пользовательских шаблонов

В папке learning_log/learning_log создайте новую папку с именем templates. Затем создайте новый файл с именем 404.html; полное имя файла имеет вид learning_log/templates/404.html. Код файла выглядит так:

404.html

```
{% extends "learning_logs/base.html" %}
{% block page_header %}
  <h2>The item you requested is not available. (404)</h2>
{% endblock page_header %}
```

Этот простой шаблон предоставляет ту же информацию, что и обобщенная страница ошибки 404, но его оформление соответствует остальным страницам сайта.

Создайте другой файл с именем 500.html:

500.html

```
{% extends "learning_logs/base.html" %}

{% block page_header %}
  <h2>There has been an internal error. (500)</h2>
{% endblock page_header %}
```

Новые файлы потребуют небольших изменений в settings.py.

settings.py

```
...
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        ...
    },
]
...
```

Это изменение приказывает Django искать шаблоны страниц ошибок в корневом каталоге шаблонов.

Локальный просмотр страниц ошибок

Если вы хотите посмотреть, как будут выглядеть страницы ошибок, в своей системе перед отправкой на сервер Heroku, вам сначала придется установить в локальных настройках режим `Debug=False`, чтобы подавить выдачу отладочных страниц Django по умолчанию. Для этого внесите следующие изменения в settings.py (убедитесь в том, что вы работаете в части settings.py, относящейся к локальной среде, а не той, которая относится к Heroku):

settings.py

```
...
# SECURITY WARNING: не включайте при реальной эксплуатации приложения!
DEBUG = False
...
```

Запросите тему или запись, которая вам не принадлежит, чтобы увидеть страницу ошибки 404. Затем запросите несуществующий URL-адрес (например, http://localhost:8000/topics/999/), чтобы увидеть страницу ошибки 500 (если вы еще не успели сгенерировать 999 тем).

Завершив проверку, верните `DEBUG` локальное значение `True`, чтобы продолжить разработку Learning Log. (Проследите за тем, чтобы режим работы с `DEBUG` не изменялся в разделе, относящемся к развертыванию в среде Heroku.)

ПРИМЕЧАНИЕ Страница ошибки 500 не содержит никакой информации о текущем пользователе, потому что Django не включает контекстную информацию в ответ при возникновении ошибки сервера.

Отправка изменений на Heroku

Теперь необходимо закрепить изменения в шаблоне и отправить их на Heroku:

```
❶ (ll_env)learning_log$ git add .
❷ (ll_env)learning_log$ git commit -am "Added custom 404 and 500 error pages."
   3 files changed, 15 insertions(+), 10 deletions(-)
   create mode 100644 learning_log/templates/404.html
   create mode 100644 learning_log/templates/500.html
❸ (ll_env)learning_log$ git push heroku master
  ...
  remote: Verifying deploy.... done.
  To https://git.heroku.com/learning-log.git
     d5075a1..4bd3b1c  master -> master
  (ll_env)learning_log$
```

В точке ❶ выдается команда `git add .`, потому что в проекте были созданы новые файлы и теперь нужно приказать Git начать отслеживание этих файлов. Затем мы закрепляем изменения ❷ и отправляем обновленный проект на Heroku❸. Теперь страницы ошибок имеют такое же оформление, как и остальные страницы сайта, а приложение выглядит более профессионально при возникновении ошибок.

Использование метода get_object_or_404()

На данный момент если пользователь вручную запрашивает несуществующую тему или запись, он получает ошибку сервера 500. Django пытается отобразить страницу, но не располагает достаточной информацией для этого, что приводит к ошибке 500. Такая ситуация более точно обрабатывается как ошибка 404, и это поведение можно реализовать при помощи вспомогательной функции Django `get_object_or_404()`. Эта функция пытается получить запрошенный объект из базы данных, а если этот объект не существует — инициирует исключение 404. Мы импортируем эту функцию в views.py и используем ее вместо `get()`:

views.py

```
from django.shortcuts import render, redirect, get_object_or_404
from django.contrib.auth.decorators import login_required
...
@login_required
def topic(request, topic_id):
    """Выводит одну тему и все ее записи."""
    topic = get_object_or_404(Topic, id=topic_id)
    # Проверка того, что тема принадлежит текущему пользователю.
    ...
```

Теперь при запросе несуществующей темы (например, http://localhost:8000/topics/999/) появляется страница ошибки 404. Чтобы развернуть это изменение, выполните закрепление и отправьте проект на Heroku.

Текущая разработка

Возможно, вы захотите продолжить разработку Learning Log после исходной отправки данных на сервер или создать и развернуть собственные проекты. Существует достаточно четко определенный процесс обновления проектов.

Сначала все необходимые изменения вносятся в локальный проект. Если изменения приводят к появлению новых файлов, добавьте эти файлы в репозиторий Git командой `git add .` (не забудьте точку в конце команды). Эта команда необходима для любого изменения, требующего миграции базы данных, потому что для каждой миграции генерируется новый файл миграции.

Затем закрепите изменения в репозитории командой `git commit -am "`*сообщение*`"`. Отправьте изменения на Heroku командой `git push heroku master`. Если вы провели локальную миграцию базы данных, также необходимо провести миграцию и для базы данных развернутого приложения. Либо используйте одноразовую команду `heroku run python manage.py migrate`, либо откройте удаленный терминальный сеанс командой `heroku run bash` и выполните команду `python manage.py migrate`. Затем посетите свой проект и убедитесь в том, что предполагаемые изменения вступили в силу.

В этом процессе легко допустить ошибку, так что не удивляйтесь, если что-то пойдет не так. Если код не работает, проанализируйте то, что было сделано, и попробуйте найти ошибку. Если найти ошибку не удается или вы не можете понять, как ее отменить, обращайтесь к рекомендациям в приложении B. Не стесняйтесь обращаться за помощью: все остальные учились строить проекты и задавали те же вопросы, которые возникнут и у вас, так что вы наверняка найдете кого-нибудь, кто согласится помочь. Решение всех возникающих проблем поможет развивать ваши навыки до того момента, когда вы начнете строить содержательные, надежные проекты и начнете отвечать на вопросы других людей.

Параметр SECRET_KEY

Django использует значение настройки `SECRET_KEY` из файла settings.py для реализации некоторых протоколов безопасности. В нашем проекте файл настроек был закреплен с включением настройки `SECRET_KEY`. Для учебного проекта этого достаточно, но на реальном сайте с настройкой `SECRET_KEY` следует действовать более осторожно. Если вы строите проект, предназначенный для реальной эксплуатации, обязательно изучите вопрос, как повысить безопасность настройки `SECRET_KEY`.

Удаление проекта с Heroku

Очень полезно многократно отработать процесс развертывания на одном проекте или серии малых проектов, чтобы получить представление о развертывании. Однако вы должны знать, как удалить проект после развертывания. Heroku также может ограничивать количество бесплатно развернутых проектов, и загромождать учетную запись учебными проектами нежелательно.

Войдите на веб-сайт Heroku (https://heroku.com/), и вы будете перенаправлены на страницу со списком проектов. Щелкните на проекте, который нужно удалить; открывается новая страница с информацией о проекте. Щелкните на ссылке Settings, прокрутите список и найдите ссылку для удаления проекта. Отменить удаление

не удастся, поэтому Heroku предложит подтвердить удаление, для чего вам нужно будет вручную ввести имя проекта.

Если вы предпочитаете работать в терминальном режиме, проект также можно удалить командой `destroy`:

```
(ll_env)learning_log$ heroku apps:destroy --app имя_приложения
```

Здесь *имя_приложения* — имя вашего проекта (например, `secret-lowlands-82594` или `learning-log`, если проект был переименован). Вам также будет предложено снова ввести имя проекта, чтобы подтвердить удаление.

ПРИМЕЧАНИЕ При удалении проекта на Heroku с локальной версией проекта ничего не происходит. Если никто не использовал ваш развернутый проект и вы просто отрабатываете процесс развертывания, ничто не мешает вам удалить свой проект на Heroku и развернуть его заново.

УПРАЖНЕНИЯ

20.3. Блог в интернете: разверните проект Blog, над которым вы работали ранее, на сервере Heroku. Проследите за тем, чтобы переменная `DEBUG` имела значение `False` и пользователи не видели полную страницу ошибки Django при возникновении каких-либо проблем.

20.4. Больше 404: функцию `get_object_or_404()` также следует применять в представлениях `new_entry()` и `edit_entry()`. Внесите изменения, протестируйте их на URL-адресе вида http://localhost:8000/new_entry/999/ и убедитесь в том, что при этом выводится ошибка 404.

20.5. Расширенное приложение Learning Log: добавьте простую функцию в Learning Log (например, вывод расширенной информации о проекте на домашней странице) и отправьте изменение в развернутую копию. Затем попробуйте внести более сложное изменение — например, чтобы пользователь мог назначить тему общедоступной. Для этого в модель Topic добавляется атрибут с именем public (по умолчанию он должен быть равен False), а на страницу `new_topic` — элемент формы, позволяющий превратить личную тему в общедоступную. После этого проведите миграцию проекта и переработайте файл views.py, чтобы любая общедоступная тема также была видимой и для пользователей, не прошедших аутентификацию. Не забудьте провести миграцию базы данных после отправки изменений на Heroku.

Итоги

В этой главе вы узнали, как придать вашему проекту простой, но профессиональный внешний вид при помощи библиотеки Bootstrap и приложения `django-bootstrap4`. При использовании Bootstrap выбранные вами стили будут работать одинаково практически на всех устройствах, используемых для работы с вашим проектом.

Вы узнали о шаблонах Bootstrap и использовали шаблон *Navbar static* для создания простого оформления Learning Log. Вы научились использовать элемент `jumbotron` для визуального выделения сообщений домашней страницы и узнали, как организовать единое стилевое оформление всех страниц на сайте.

В последней части проекта вы узнали, как развернуть проект на серверах Heroku, чтобы с ним мог работать любой желающий. Вы создали учетную запись Heroku и установили инструменты, упрощающие процесс развертывания. Вы использовали Git для закрепления рабочего проекта в репозитории и отправили репозиторий на серверы Heroku. Наконец, вы узнали, как защитить приложение, включив режим `DEBUG=False` на работающем сервере.

Итак, работа над Learning Log закончена, и вы можете взяться за построение собственных проектов. Начните с простых приложений и убедитесь в том, что приложение заработало, прежде чем наращивать сложность. Пусть ваше обучение будет интересным — и удачи с вашими собственными проектами!

Послесловие

Поздравляю! Вы изучили основы Python и применили полученные знания для построения реальных проектов. Вы построили компьютерную игру, провели визуализацию данных и создали веб-приложение. Теперь перед вами открыто несколько разных направлений, по которым вы можете повышать свою квалификацию программиста.

На первых порах продолжайте работу над осмысленными проектами, которые представляют для вас интерес. Программирование становится более интересным, когда вы решаете актуальные, важные задачи, а к этому моменту вы знаете достаточно для того, чтобы участвовать в самых разных проектах. Придумайте собственную игру или напишите свою версию классической аркадной игры. Попробуйте проанализировать данные, важные для вас, и создайте визуализации, которые выявляют интересные закономерности и связи. Создайте собственное веб-приложение или сымитируйте одно из своих любимых приложений.

По возможности предлагайте другим людям использовать ваши программы. Если вы написали игру — пусть кто-нибудь сыграет в нее. Если вы построили визуализацию, покажите ее другим людям и посмотрите, как они воспримут результат. Если вы создали веб-приложение, разверните его в интернете и предложите другим опробовать его. Прислушивайтесь к мнению пользователей и старайтесь учитывать полученную информацию в своих проектах; это поможет вам стать профессионалом более высокого класса.

В ходе работы над собственными проектами вы столкнетесь с проблемами, которые будет слишком трудно или даже невозможно решить самостоятельно. Поищите новые способы обратиться за помощью, найдите свое место в сообществе Python. Вступите в местную группу пользователей Python или присоединитесь к одному из сетевых сообществ. Рассмотрите возможность посещения ближайшей конференции PyCon.

Постарайтесь выдержать баланс между работой над проектами, интересующими вас, и развитием ваших общих навыков Python. В интернете можно найти много источников информации о Python, для программистов среднего уровня написано много книг. Теперь, когда вы овладели основами и знаете, как применять свои навыки, многие из этих ресурсов станут доступными для вас. Работа с учебниками

и книгами Python закрепит изученный материал, углубит ваше понимание программирования в целом и Python в частности. И когда вы после этого вернетесь к работе над проектами, вы сможете более эффективно решать более широкий круг задач.

Поздравляем — вы проделали долгий путь! Желаем удачи в дальнейшем изучении Python!

Приложения

A Установка Python и диагностика проблем

Python существует в нескольких разных версиях, с разными вариантами конфигурации в каждой операционной системе. Это приложение пригодится вам в том случае, если описание из главы 1 не сработало или вы захотите установить другую версию Python вместо той, которая поставлялась с вашей системой.

Python в Windows

Инструкции из главы 1 описывают установку Python при помощи официальной программы установки по адресу https://python.org/. Если вам не удастся запустить Python после использования программы установки, инструкции по диагностике из этого раздела помогут вам добиться работоспособности Python.

Поиск интерпретатора Python

Если при вводе простой команды `python` произойдет ошибка, скорее всего, вы забыли установить флажок включения Python в переменную `PATH` при запуске программы установки. В этом случае необходимо сообщить Windows, где следует искать интерпретатор Python. Чтобы найти его, откройте содержимое диска C и найдите папку, имя которой начинается с Python (попробуйте ввести слово python на панели поиска Проводника Windows). Откройте папку и найдите файл с именем python в нижнем регистре. Щелкните правой кнопкой мыши на этом файле и выберите команду Свойства (Properties); путь к этому файлу будет отображаться под заголовком Размещение (Location).

Чтобы сообщить Windows, где следует искать интерпретатор, откройте терминальное окно и введите путь, за которым следует команда `--version`:

```
$ C:\\Python37\python --version
Python 3.7.2
```

В вашей системе путь может выглядеть немного иначе, например C:\Users*пользователь*\Programs\Python37\python. При использовании этого пути Windows сможет запустить интерпретатор Python.

Включение Python в переменную PATH

Вводить полный путь каждый раз, когда вы захотите открыть терминал Python, довольно утомительно, поэтому мы добавим этот путь в системное окружение, чтобы вы могли использовать команду `python`. Откройте Панель управления своей системы, выберите категорию Система и безопасность (System and Security), затем выберите Система (System). Щелкните на ссылке Дополнительные параметры (Advanced System Settings). В открывшемся окне щелкните на кнопке Переменные среды (Environment Variables).

В поле Системные переменные (System variables) найдите переменную с именем Path. Щелкните на имени Path, затем на кнопке Изменить (Edit). Должен открыться список мест, которые просматриваются вашей системой при поиске программ. Щелкните на кнопке New и вставьте путь к вашему файлу python.exe в открывшемся текстовом поле. Если ваша система настроена так же, как и моя, результат будет выглядеть так:

```
C:\Python37
```

Обратите внимание: мы не включаем имя файла python.exe, а только сообщаем системе, где его следует искать.

Закройте терминальное окно и откройте новое. Новое содержимое переменной Path будет загружено в терминальном сеансе. Теперь при вводе команды `python --version` запускается версия Python, которая только что была задана в переменной `Path`. Теперь вы можете запустить терминальный сеанс Python, просто введя команду `python` в командной строке.

ПРИМЕЧАНИЕ Если вы работаете в более ранней версии Windows, при нажатии кнопки Изменить (Edit) может появиться поле Значение переменной (Variable Value). Прокрутите его до правого края при помощи клавиши →. Будьте внимательны — вы не должны заменить существующее значение; если это произойдет, щелкните на кнопке Отмена (Cancel) и повторите попытку. Добавьте точку с запятой и путь к файлу python.exe в существующую переменную:

```
%SystemRoot%\system32\...\System32\WindowsPowerShell\v1.0\;C:\Python37
```

Переустановка Python

Если вам все еще не удается запустить Python, нередко удаление Python и повторный запуск программы установки решают проблемы, возникшие при первой попытке.

Откройте панель управления в своей системе и выберите Programs and Features. Прокрутите список, найдите в нем только что установленную версию Python и выберите ее. Щелкните на кнопке Uninstall/Change, затем на кнопке Uninstall в открывшемся диалоговом окне. Снова запустите программу установки в соответствии с инструкциями из главы 1, но на этот раз проследите за тем, чтобы был установлен флажок Add Python to PATH; также проверьте другие настройки, относящиеся к вашей системе. Если вы все еще сталкиваетесь с проблемами и не знаете, куда обратиться за помощью, обращайтесь к рекомендациям в приложении B.

Python в macOS

В инструкциях по установке из главы 1 используется официальная программа установки Python по адресу https://python.org/, которую я рекомендую использовать, если только у вас нет веских причин сделать иначе. Другое возможное решение — Homebrew, инструмент, который может использоваться для установки разнообразных программ в macOS. Если вы уже применяете Homebrew и хотите воспользоваться этой программой для установки Python или если ваши коллеги работают с Homebrew и вы хотите работать по той же схеме, следуйте инструкции ниже.

Установка Homebrew

Homebrew зависит от некоторых средств командной строки пакета Apple Xcode, поэтому сначала необходимо установить инструментарий командной строки Xcode. Откройте терминальное окно и введите следующую команду:

```
$ xcode-select --install
```

Введите подтверждения в открывающихся диалоговых окнах (это может занять некоторое время в зависимости от скорости подключения). Затем установите Homebrew:

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Эту команду можно найти на основной странице сайта Homebrew по адресу http://brew.sh/. Обратите внимание на пробел между `curl -fsSL` и URL-адресом.

ПРИМЕЧАНИЕ Ключ `-e` в этой команде приказывает Ruby (язык программирования, на котором написана программа Homebrew) выполнить загружаемый код. Такие команды должны использоваться только с доверенными источниками.

Чтобы убедиться в том, что установка Homebrew прошла успешно, выполните следующую команду:

```
$ brew doctor
Your system is ready to brew.
```

Этот результат означает, что все готово для установки пакетов Python через Homebrew.

Установка Python

Чтобы установить новейшую версию Python, введите следующую команду:

```
$ brew install python
```

Для проверки установленной версии используется следующая команда:

```
$ python3 --version
Python 3.7.2
$
```

Теперь вы сможете запустить терминальный сеанс Python командой `python3`, а также воспользоваться командой `python3` для настройки текстового редактора, чтобы программы Python запускались в только что установленной версии Python вместо предшествующей версии, установленной в вашей системе. Если вам понадобится помощь по настройке Sublime Text для использования только что установленной версии, обращайтесь к инструкциям из главы 1.

Python в системе Linux

Python включается по умолчанию почти в каждую систему Linux. Но если по умолчанию в вашей системе используется версия ниже 3.6, вам стоит установить обновленную версию. Следующие инструкции должны работать в большинстве систем на базе apt.

Мы воспользуемся пакетом `deadsnakes`, упрощающим установку нескольких версий Python:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt-get update
$ sudo apt install python3.7
```

Эти команды устанавливают Python 3.7 в вашей системе.

Следующая команда запускает терминальный сеанс с Python 3.7:

```
$ python3.5
>>>
```

Эта команда также часто используется при настройке текстового редактора и при запуске программ из терминального окна.

Ключевые слова Python и встроенные функции

Python содержит целый набор ключевых слов и встроенных функций. Помните о них, выбирая имена переменных. Одна из типичных проблем программирования — выбор хороших имен переменных, которые должны быть достаточно короткими и содержательными. Однако в качестве имен нельзя использовать ключевые слова Python, а также имена встроенных функций Python, потому что это приведет к замещению функций.

В этом разделе перечислены ключевые слова Python и имена встроенных функций, чтобы вы знали, каких имен следует избегать.

Ключевые слова Python

Каждое ключевое слово из следующего списка имеет конкретный смысл в программах Python. При попытке использовать эти слова в качестве имен переменных произойдет ошибка.

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

Встроенные функции Python

Если вы используете любую из следующих встроенных функций в качестве имени переменной, это приведет не к ошибке, а к изменению поведения этой функции:

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Б Текстовые редакторы

Программисты проводят много времени за написанием, чтением и редактированием кода в текстовом редакторе, или *интегрированной среде разработки* (IDE, Integrated Development Environment). Очень важно, чтобы эта работа выполнялась по возможности эффективно. Эффективный редактор должен поддерживать некоторые простые возможности: например, выделять структуру кода, чтобы вы могли обнаружить типичные ошибки во время работы, — но не до такой степени, чтобы это отвлекало программиста от работы. Также редактор должен поддерживать автоматическую расстановку отступов, маркеры для обозначения длины строки и комбинации клавиш для часто выполняемых операций.

IDE представляет собой текстовый редактор, дополненный встроенными инструментами, например интерактивными отладчиками и интроспекцией. IDE анализирует ваш код в процессе ввода и пытается использовать информацию о создаваемом проекте. Например, когда вы начинаете вводить имя функции, IDE может вывести список всех аргументов, получаемых этой функцией. Такое поведение может быть очень полезно, когда среда работает идеально, а вы хорошо понимаете все происходящее. Но оно также может сбить с толку новичка и усложнить исправление ошибок, когда вы не понимаете, почему ваш код не работает в IDE.

Начинающему программисту рекомендуется использовать редактор, который поддерживает все эти функции, но не создает проблем на начальном этапе. Текстовые редакторы также создают существенно меньшую нагрузку на систему; таким образом, если вы работаете на старом компьютере или в системе с недостаточными ресурсами, текстовый редактор будет работать лучше, чем IDE. Но если вы уже знакомы с IDE или хотите работать в одних условиях с вашими коллегами, использующими IDE, — ничто не мешает вам попробовать.

На данный момент выбор инструмента не столь важен; лучше провести время за изучением языка и работой над интересующими вас проектами. А когда вы освоитесь с азами, вы начнете лучше понимать, какие инструменты подходят лично вам.

В этом приложении мы настроим текстовый редактор Sublime Text для более эффективной работы. Также будут кратко рассмотрены другие редакторы, заслуживающие вашего внимания.

Настройка Sublime Text

В главе 1 мы настроили Sublime Text для использования нужной версии Python при запуске программ. А теперь настроим некоторые функции, упомянутые в начале приложения.

Преобразование табуляций в пробелы

Чередование символов табуляции и пробелов в коде может создать коварные проблемы в программах. Чтобы этого не произошло, вы можете настроить редактор Sublime Text так, чтобы для создания отступов всегда использовались пробелы (даже при нажатии клавиши Tab). Выберите команду View ▸ Indentation и убедитесь в том, что флажок Indent Using Spaces установлен. Если флажок снят, установите его. Также убедитесь в том, что ширина позиции табуляции (Tab Width) установлена равной 4 пробелам.

Если в вашей программе используется смесь табуляций и пробелов, вы можете преобразовать все табуляции в пробелы командой View ▸ Indentation ▸ Convert Tabs to Spaces. Также можете обратиться к этим функциям, щелкнув на поле Spaces в правом нижнем углу окна Sublime Text. Теперь вы сможете использовать клавишу Tab для создания отступов в коде. Sublime Text автоматически вставляет пробелы для создания отступов в этих строках.

Выбор индикатора длины строки

В большинстве редакторов существует возможность назначения визуального признака (обычно вертикальной линии), обозначающего рекомендуемую длину строки. В сообществе Python по общепринятым соглашениям строки ограничиваются длиной в 79 символов и менее. Чтобы включить эту возможность, выберите команду View ▸ Ruler и щелкните на значении 80. Sublime Text устанавливает вертикальную черту в позиции 80-го символа, чтобы вам было проще ограничить строки необходимой длиной.

Расстановка и отмена отступов в блоках

Чтобы включить отступ в блоке, выделите код и выберите команду Edit ▸ Line ▸ Indent или нажмите Ctrl+] (Cmd+] в macOS). Чтобы убрать отступ в блоке, выберите команду Edit ▸ Line ▸ Unindent или нажмите Ctrl+[(или Cmd +]).

Преобразование блоков в комментарии

Чтобы временно исключить блок кода из программы, вы можете выделить блок и закрыть его комментарием, чтобы он игнорировался интерпретатором Python. Выберите команду Edit ▸ Comment ▸ Toggle Comment (Ctrl+/ или Cmd+/). Строки помечаются знаком комментария (#) на одном уровне отступа с кодом, указывающим на

то, что это не обычный комментарий. Когда вы захотите вернуть блок в программу, выделите его и выберите ту же команду.

Сохранение конфигурации

Некоторые из упомянутых настроек распространяются только на текущий файл, с которым вы работаете. Чтобы изменения действовали для всех файлов, открываемых в Sublime Text, необходимо внести их в пользовательскую конфигурацию. Выберите команду Sublime Text ▸ Preferences ▸ Settings и найдите файл Preferences.sublime-settings — User. Включите в этот файл следующий фрагмент:

```
{
    "rulers": [80],
    "translate_tabs_to_spaces": true
}
```

Сохраните файл, и ваши настройки линейки и табуляций будут действовать во всех файлах, с которыми вы работаете в Sublime Text. Если вы будете добавлять в этот файл другие строки, следите за тем, чтобы каждая строка, кроме последней, завершалась запятой. Просмотрите файлы настроек других пользователей в интернете и настройте свой редактор с параметрами, которые обеспечат вам наиболее эффективную работу.

Дальнейшая настройка

Многие аспекты Sublime Text можно настроить так, чтобы ваша работа была еще более эффективной. Изучая меню, обращайте внимание на комбинации клавиш для команд, которыми вы пользуетесь особенно часто. Каждый раз, когда вы используете комбинацию клавиш вместо того, чтобы тянуться к мыши или сенсорной панели, ваша работа становится чуть более эффективной. Не пытайтесь запомнить все сразу; постарайтесь эффективно выполнять наиболее часто выполняемые действия и обращайте внимание на другие средства, которые могут пригодиться в организации вашего рабочего процесса.

Другие текстовые редакторы и IDE

Многие разработчики пользуются другими текстовыми редакторами; наверняка вы не раз услышите о них. Как правило, такие редакторы настраиваются по тем же принципам, что и Sublime Text. Ниже приведена небольшая подборка текстовых редакторов, с которыми вы можете столкнуться.

IDLE

IDLE — текстовый редактор, включенный в поставку Python. В работе он менее интуитивен, чем Sublime Text, но он часто упоминается в других учебниках, предназначенных для начинающих, поэтому вам стоит познакомиться с ним.

Geany

Geany — простой текстовый редактор, который позволяет запускать почти все программы прямо из редактора. Кроме того, Geany выводит результаты в терминальном окне, что поможет вам освоить работу в терминале. Редактор Geany имеет очень простой интерфейс, но он достаточно мощен, поэтому многие опытные программисты продолжают пользоваться им.

Emacs и vim

Многие опытные программисты отдают предпочтение emacs или vim. Эти два популярных редактора спроектированы так, чтобы пользователю не приходилось отрывать руки от клавиатуры. Это означает, что опытный пользователь может читать, писать и редактировать код с очень высокой эффективностью. С другой стороны, для освоения этих редакторов придется основательно потрудиться. Vim входит в поставку большинства систем Linux и macOS, причем как Emacs, так и Vim могут выполняться полностью в режиме терминала. По этой причине они часто используются для написания кода на серверах через удаленный терминальный сеанс.

Программисты часто советуют хотя бы опробовать эти редакторы, но многие профессионалы забывают, как много всего нового узнает новичок. Знать о существовании этих редакторов полезно, но отложите знакомство с ними до того момента, когда вы начнете уверенно писать программы и работать с ними в более простых редакторах, которые позволяют сосредоточиться на изучении программирования, а не на работе с редактором.

Atom

Atom — текстовый редактор, обладающий рядом возможностей, которые обычно встречаются в IDE. Разработчик может открыть отдельный файл для работы или же открыть папку проекта и получить удобный доступ ко всем файлам в этом проекте. Atom интегрируется с Git и GitHub, и когда вы начнете пользоваться системами контроля версий, вы сможете работать с локальными и удаленными репозиториями прямо из редактора без запуска отдельного терминала.

Atom также позволяет устанавливать пакеты, а некоторые аспекты его поведения могут быть расширены. Многие пакеты расширения реализуют поведение, с которым Atom в большей степени напоминает IDE.

Visual Studio Code

Visual Studio Code, или VS Code, — другой редактор, более похожий на IDE. VS Code поддерживает эффективную работу с отладчиком, содержит встроенные средства контроля версий, а также предоставляет функциональность автоматического завершения кода.

PyCharm

Среда PyCharm популярна среди программистов Python, потому что она была построена специально для работы с Python. Полная версия требует платной подписки, но также существует бесплатная версия PyCharm Community Edition, которую многие разработчики считают полезной.

В PyCharm встроен *статический анализатор* кода, который проверяет, что ваш стиль программирования соответствует основным соглашениям Python, и предлагает рекомендации при отходе от стандартного форматирования кода Python. Также в PyCharm имеется интегрированный отладчик, который помогает эффективно находить ошибки, и специальные режимы для полноценной работы с рядом популярных библиотек Python.

Jupyter Notebook

Jupyter Notebook отличается от традиционных текстовых редакторов или IDE тем, что это веб-приложение, построенное из блоков. Блоки могут быть как программными, так и текстовыми. Текстовые блоки могут содержать разметку Markdown, что позволяет включать в них простое форматирование.

Документы Jupyter Notebook были разработаны для применения Python в научных приложениях, но с того времени они развивались и нашли применение в широком спектре ситуаций. Вместо того чтобы включать обычные комментарии в файл .py, вы включаете между частями кода понятный текст с несложным форматированием: заголовками, маркированными списками и гиперссылками. Каждый блок кода может выполняться независимо от других, что позволяет тестировать небольшие части программы; также можно выполнить все блоки одновременно. Каждый блок имеет собственную область вывода, причем эти области можно включать и отключать по мере надобности.

Документы Jupyter Notebook могут создавать некоторую путаницу из-за взаимодействия между ячейками. Функция, определенная в одной ячейке, становится доступной для других ячеек. В большинстве случаев это удобно, но также возможны недоразумения — например, в больших документах или если вы недостаточно хорошо понимаете, как работает среда Notebook.

Каждый разработчик, занимающийся научной работой или задачами обработки данных на Python, почти неизбежно столкнется с Jupyter Notebook в какой-то момент своей деятельности.

B Помощь и поддержка

Во время изучения программирования каждый из нас в какой-то момент оказывается в тупике. Один из важнейших навыков, которые должен освоить каждый программист, — умение быстро найти выход из тупика. В этом приложении описаны некоторые способы решения проблем, которые помогут вам выпутаться из сложной ситуации.

Первые шаги

Если у вас возникли трудности, сначала оцените ситуацию. Прежде чем обращаться за помощью, убедитесь в том, что вы можете четко ответить на следующие три вопроса:

- Что вы пытаетесь сделать?
- Что вы делали до настоящего момента?
- Какие результаты вы получили?

Ваши ответы должны быть как можно более конкретными. Например, в первом вопросе утверждение «Я пытаюсь установить последнюю версию Python на свою машину с Windows 10» достаточно информативно, чтобы другие пользователи сообщества Python могли вам помочь. Формулировки типа «Я пытаюсь установить Python» просто не содержат достаточной информации, чтобы кто-то мог предложить вам помощь.

Ответ на второй вопрос должен быть достаточно развернутым, чтобы вам не предлагали делать то, что уже было сделано: описание «Я открыл страницу https://python.org/downloads/ и выбрал кнопку Download для моей системы. Затем я запустил программу установки» более полезно, чем «Я зашел на сайт Python и что-то загрузил».

Что касается последнего вопроса, при поиске в интернете или обращении за помощью желательно знать точные сообщения об ошибках.

Иногда в процессе поиска ответов на эти три вопроса вы сами понимаете, где была допущена оплошность, и выходите из тупика самостоятельно. У программистов

даже имеется специальный термин для таких ситуаций: это называется «отладкой с резиновой уткой». Если вы четко объясните свою ситуацию резиновой утке (или любому другому неодушевленному объекту) и зададите конкретный вопрос, часто вы сможете ответить на него. Некоторые организации даже заводят настоящую резиновую утку, чтобы подтолкнуть своих программистов к «разговорам с уткой».

Попробуйте заново

Просто вернитесь к началу и попробуйте еще раз; часто этого оказывается достаточно для решения многих проблем. Допустим, вы пытаетесь написать цикл `for` на основе примера из книги. Возможно, вы пропустили что-то совсем простое — скажем, двоеточие в конце строки. Выполнение всех действий заново поможет избежать повторения ошибки.

Сделайте перерыв

Если вы уже долго бьетесь над какой-то проблемой, перерыв — едва ли не лучшее, что можно сделать. Когда мы трудимся над одной задачей в течение долгого времени, наш мозг начинает концентрироваться на единственном решении. Мы забываем о сделанных предположениях, а перерыв помогает взглянуть на проблему под новым углом. Перерыв даже не обязан быть долгим, просто нужно заняться чем-то, что выведет вас из текущего мысленного настроя. Если вы давно сидите на одном месте, переключитесь на какую-нибудь физическую нагрузку: походите или выйдите на улицу; может, стоит выпить стакан воды или съесть что-нибудь легкое и здоровое.

Если вы начинаете отчаиваться, попробуйте отложить работу на следующий день. Хороший сон почти всегда упрощает решение задачи.

Обратитесь к ресурсам этой книги

В список сетевых ресурсов этой книги (https://www.nostarch.com/pythoncrashcourse2e/) включен ряд полезных разделов, посвященных настройке системы и обзорам каждой главы. Просмотрите эти ресурсы — возможно, вы найдете в них то, что вам поможет.

Поиск в интернете

Вполне вероятно, что кто-то уже столкнулся с такой же проблемой и написал о ней в интернете. Хорошие навыки поиска и конкретные запросы помогут вам найти информацию для решения ваших проблем. Например, если у вас возникли трудности с установкой Python в Windows 10, поиск по условию *python windows 10* может привести вас к ответу.

Поиск по точным сообщениям об ошибках тоже может оказаться исключительно полезным. Допустим, при попытке запуска терминального сеанса Python произошла следующая ошибка:

```
> python
'python' is not recognized as an internal or external command,
operable program or batch file.
>
```

Вероятно, поиск по полному тексту сообщения принесет полезную информацию.

В результатах поиска, связанного с программированием, особенно часто встречаются некоторые сайты. Я опишу некоторые из них, чтобы вы знали, чего от них можно ждать.

Stack Overflow

Stack Overlow (http://stackoverlow.com/), один из самых популярных сайтов с вопросами и ответами для программистов, часто встречается на первой странице результатов поиска, связанного с Python. Пользователи публикуют вопросы по возникшим проблемам, а другие участники пытаются дать полезные ответы. Пользователи могут голосовать за ответы, которые, по их мнению, принесли наибольшую пользу, так что лучшими ответами обычно оказываются первые из найденных.

На сайте Stack Overflow можно найти ответы на многие основные вопросы по языку Python, потому что со временем они были хорошо проработаны. Пользователи также публикуют обновления, так что ответы остаются относительно актуальными. На момент написания книги на сайте Stack Overflow были опубликованы ответы более чем на 1 000 000 вопросов, связанных с Python.

Официальная документация Python

Официальная документация Python (http://docs.python.org/) уже не столь бесспорно полезна для новичков, потому что она написана для документирования языка, а не для разъяснений. Примеры в официальной документации должны работать, но возможно, что-то в них останется для вас непонятным. Тем не менее это полезный ресурс, к которому стоит обращаться при поиске, а по мере углубления вашего понимания Python он будет приносить еще больше пользы.

Официальная документация библиотек

Если вы используете конкретную библиотеку (например, Pygame, Matplotlib, Django и т. д.), в поиске часто будут встречаться ссылки на официальную документацию этого проекта — например, документация http://docs.djangoproject.com/ чрезвычайно полезна. Если вы собираетесь работать с любыми из этих библиотек, стоит ознакомиться с официальной документацией.

r/learnpython

Форум Reddit состоит из ряда *подфорумов* (subreddits). Подфорум r/learnpython (http://reddit.com/r/learnpython/) достаточно активен и благожелательно настроен. Здесь вы сможете прочитать вопросы других участников и опубликовать свои собственные.

Сообщения в блогах

Многие программисты ведут блоги и пишут об аспектах языка, с которыми они работают. Прежде чем брать на вооружение любой совет, просмотрите несколько первых комментариев к сообщению в блоге. Если комментариев нет, к сообщению следует относиться скептически. Вполне возможно, что никто другой не смог убедиться в полезности этого совета.

IRC (Internet Relay Chat)

Программисты общаются в реальном времени на каналах IRC. Если вы зашли в тупик, а поиск в интернете не принес результатов, обращение на канал IRC может оказаться лучшим вариантом. Многие люди, проводящие время на этих каналах, вежливы и доброжелательны, особенно если вы можете точно описать, что пытаетесь сделать, что уже было сделано и какие результаты вы получили.

Создание учетной записи IRC

Чтобы создать учетную запись на IRC, зайдите на сайт http://webchat.freenode.net/. Выберите имя пользователя, заполните поле с кодом проверки (captcha) и щелкните на кнопке Connect. Появляется сообщение, приветствующее вас на сервере IRC freenode. В поле в нижней части окна введите следующую команду:

```
/msg nickserv register пароль адрес_электронной_почты
```

Введите свой пароль и адрес электронной почты. Выберите простой пароль, который не используется для других учетных записей. На ваш адрес электронной почты придет сообщение с инструкцией по подтверждению учетной записи. В нем будет содержаться команда следующего вида:

```
/msg nickserv verify register имя код_проверки
```

Скопируйте эту строку на сайт IRC с именем, которое вы выбрали ранее, и полученным кодом проверки. Теперь все готово для присоединения к каналу.

Если в какой-то момент у вас возникнут трудности со входом под вашей учетной записью, попробуйте ввести следующую команду:

```
/msg nickserv identify имя пароль
```

Замените *имя* и *пароль* своими данными. После прохождения аутентификации в Сети вы сможете обратиться к каналу, требующему проверенного имени пользователя.

Выбор канала

Чтобы присоединиться к основному каналу Python, введите команду `/join #python`. На экране появляется подтверждение о том, что вы присоединились к каналу, а также общая информация о канале.

Канал `##learnpython` (с двумя символами `#`) обычно тоже активно работает. Этот канал связан с http://reddit.com/r/learnpython/, поэтому вы также будете видеть здесь информацию о сообщениях на `r/learnpython`. Если вы работаете над веб-приложениями, также можно присоединиться к каналу `#django`.

После присоединения к каналу вы сможете читать сообщения других пользователей, а также задавать вопросы.

Культура IRC

Чтобы помощь была эффективной, необходимо кое-что знать о культуре IRC. Если вы сосредоточитесь на трех вопросах, приведенных в начале приложения, это определенно станет значительным шагом на пути к успешному решению. Люди охотно помогут вам, если вы сможете точно объяснить, что вы пытаетесь сделать, что уже пробовали сделать и какие именно результаты при этом получили. Если вам нужно поделиться кодом или выходными данными, участники IRC используют внешние сайты, предназначенные для этой цели (например, https://bpaste.net/+python/). Тем самым предотвращается загромождение канала кодом и упрощается чтение опубликованного кода.

Терпение всегда повышает вероятность успеха. Четко сформулируйте свой вопрос и подождите ответа. Часто пользователи участвуют сразу в нескольких беседах, но скорее всего, кто-нибудь откликнется на вашу просьбу в разумные сроки. При небольшом количестве участников на получение ответа может потребоваться больше времени.

Slack

Slack — современное переосмысление технологии IRC. Каналы Slack часто используются для внутреннего общения в компаниях, но также существует много открытых групп, к которым вы можете присоединиться. Если вы захотите просмотреть Slack-группы Python, начните с https://pyslackers.com/. Чтобы получить приглашение, щелкните на ссылке Slack в верхней части страницы и введите свой адрес электронной почты.

Оказавшись в рабочем пространстве Python Developers, вы увидите список каналов. Щелкните на ссылке Channels и выберите тему, которая вас интересует. Возможно, вам стоит начать с каналов `#learning_python` и `#django channels`.

Discord

Discord — еще одна чат-среда с сообществом Python, в которой можно попросить о помощи и читать обсуждения, связанные с Python.

Чтобы поближе познакомиться с Discord, откройте страницу https://pythondiscord.com/ и щелкните на ссылке Chat Now. На экране должно появиться окно с автоматически сгенерированным приглашением; щелкните на ссылке Accept Invite. Если у вас уже имеется учетная запись Discord, вы можете выполнить вход с данными своей учетной записи. Если же учетной записи еще нет, введите имя пользователя и следуйте инструкциям для завершения процесса регистрации.

Если вы впервые посещаете Python-площадку Discord, то для полноценного участия необходимо принять правила сообщества. Когда это будет сделано, вы сможете присоединиться к любому из интересующих вас каналов. Если вам потребуется помощь, отправьте свой вопрос на один из каналов Python Help.

Г Git и контроль версий

Программы контроля версий позволяют создать «моментальный снимок» состояния программы, находящейся в рабочем состоянии. При внесении изменений в проект (например, при реализации новой возможности) существует возможность возврата к предыдущему работоспособному состоянию, если в новом состоянии проект работает не так, как ожидалось.

Благодаря программам контроля версий программист может работать над усовершенствованием и совершать ошибки, не опасаясь нарушить работоспособность проекта. Это особенно важно в больших проектах, но также может быть полезно и в меньших проектах — даже для программ, содержащихся в одном файле.

В этом приложении вы узнаете, как установить Git и использовать эту систему для управления версиями программ, над которыми вы работаете. В настоящее время Git является самой популярной программой контроля версий. Многие расширенные возможности Git предназначены для групп, совместно работающих над большими проектами, но базовые функции также хорошо подходят для разработчиков-одиночек. Контроль версий в Git основан на отслеживании изменений, вносимых в каждый файл проекта; если вы совершите ошибку, то сможете просто вернуться к ранее сохраненному состоянию.

Установка Git

Git работает во всех операционных системах, но способы установки в конкретных системах различаются. Ниже приведены инструкции для каждой операционной системы.

Установка Git в Windows

Программу установки Git можно загрузить по адресу https://git-scm.com/. Найдите ссылку для загрузки программы установки, соответствующей вашей системе.

Установка Git в macOS

Возможно, программа Git уже установлена в вашей системе; попробуйте ввести команду `git --version`. Если в выводе будет указан конкретный номер версии, Git

устанавливается в вашей системе. Если в сообщении будет предложено установить или обновить Git, просто выполните инструкции на экране.

Также есть другой способ: зайдите на сайт https://git-scm.com/ и выберите подходящую программу установки для вашей системы.

Установка Git в Linux

Чтобы установить Git в Linux, введите следующую команду:

```
$ sudo apt install git-all
```

Все, теперь вы можете использовать Git в своих проектах.

Настройка Git

Git следит за тем, кто вносит изменения в проект, даже если над проектом работает всего один человек. Для этого Git необходимо знать ваше имя пользователя и пароль. Имя пользователя должно быть указано обязательно, но ничто не мешает вам ввести фиктивный адрес электронной почты:

```
$ git config --global user.name "имя_пользователя"
$ git config --global user.email "username@example.com"
```

Если вы забудете это сделать, Git запросит информацию при первом закреплении.

Создание проекта

Для начала создадим проект для работы. Создайте в системе папку с именем git_practice. В этой папке разместите файл с простой программой Python:

hello_git.py

```
print("Hello Git world!")
```

Эта программа поможет нам изучить базовую функциональность Git.

Игнорирование файлов

Файлы с расширением .pyc автоматически строятся на основе файлов .py, и отслеживать их в Git не нужно. Эти файлы хранятся в каталоге с именем __pycache__. Чтобы приказать Git игнорировать этот каталог, создайте специальный файл с именем .gitignore (с точкой в начале имени и без расширения) и включите в него следующую строку:

.gitignore

```
__pycache__/
```

В результате Git будет игнорировать любые файлы, находящиеся в каталоге __pycache__. Файл .gitignore избавляет проект от излишнего «балласта» и упрощает работу с ним.

Возможно, для открытия файла .gitignore вам придется изменить настройки своего текстового редактора, чтобы в нем отображались скрытые файлы. Некоторые редакторы настроены на игнорирование имен файлов, начинающихся с точки.

Инициализация репозитория

Теперь, когда у вас имеется каталог с файлом Python и файлом .gitignore, можно переходить к инициализации репозитория Git. Откройте терминал, перейдите в каталог git_practice и выполните следующую команду:

```
git_practice$ git init
Initialized empty Git repository in git_practice/.git/
git_practice$
```

Из выходных данных видно, что Git инициализирует пустой репозиторий в каталоге git_practice. *Репозиторий* представляет собой набор файлов программы, который активно отслеживается системой Git. Все файлы, используемые Git для управления репозиторием, хранятся в скрытом каталоге .git/, с которым вам вообще не придется работать. Просто не удаляйте этот каталог, иначе вся история проекта будет потеряна.

Проверка статуса

Прежде чем делать что-либо, проверьте статус проекта:

```
  git_practice$ git status
❶ On branch master

  No commits yet

❷ Untracked files:
    (use "git add <file>..." to include in what will be committed)

    .gitignore
    hello_world.py


❸ nothing added to commit but untracked files present (use "git add" to track)
  git_practice$
```

В Git *ветвью* (branch) называется версия проекта, над которым вы работаете; из вывода видно, что текущей является ветвь с именем `master` ❶. Каждый раз, когда вы проверяете статус своего проекта, программа должна сообщать, что текущей является ветвь `master`. После этого мы видим, что система готова к исходному за-

креплению. *Закреплением* (commit) называется «моментальный снимок» проекта на определенный момент времени.

Git сообщает, что в проекте имеются неотслеживаемые файлы ❷, потому что мы еще не сообщили Git, какие файлы должны отслеживаться. Затем мы узнаем, что в текущее закрепление еще ничего не добавлено, но существуют неотслеживаемые файлы, которые следует добавить в репозиторий ❸.

Добавление файлов в репозиторий

Добавим в репозиторий два файла и снова проверим статус:

```
❶ git_practice$ git add .
❷ git_practice$ git status
  On branch master

  No commits yet

  Changes to be committed:
   (use "git rm --cached <file>..." to unstage)

❸   new file:   .gitignore
    new file:   hello_git.py

  git_practice$
```

Команда `git add .` добавляет в репозиторий все файлы проекта, которые еще не отслеживаются ❶. Закрепление при этом не выполняется; команда просто сообщает Git, что эти файлы нужно отслеживать. При проверке статуса проекта мы видим, что Git находит изменения, которые необходимо закрепить ❷. Метка *new file* означает, что эти файлы были только что добавлены в репозиторий ❸.

Закрепление

Выполним первое закрепление:

```
❶ git_practice$ git commit -m "Started project."
❷ [master (root-commit) ee76419] Started project.
❸  2 files changed, 4 insertions(+)
   create mode 100644 .gitignore
   create mode 100644 hello_git.py
❹ git_practice$ git status
  On branch master
  nothing to commit, working tree clean
  git_practice$
```

Команда `git commit -m "`*сообщение*`"` ❶ создает «моментальный снимок» состояния проекта. Флаг `-m` приказывает Git сохранить следующее сообщение (`"Started project."`) в журнале проекта. Из вывода следует, что текущей является ветвь `master` ❷, а в проекте изменились два файла ❸.

Проверка статуса показывает, что текущей является ветвь `master`, а рабочий каталог чист ❹. Это сообщение должно выводиться каждый раз, когда вы закрепляете рабочее состояние своего проекта. Если вы получите другое сообщение, внимательно прочитайте его; скорее всего, вы забыли добавить файл после закрепления.

Просмотр журнала

Git ведет журнал всех закреплений, выполненных в проекте. Проверим содержимое журнала:

```
git_practice$ git log
commit a9d74d87f1aa3b8f5b2688cb586eac1a908cfc7f (HEAD -> master)
Author: Eric Matthes <eric@example.com>
Date:   Mon Jan 21 21:24:28 2019 -0900

    Started project.
git_practice$
```

Каждый раз, когда вы выполняете закрепление, Git генерирует уникальный идентификатор из 40 символов. Сохраняется информация о том, кто выполнил изменение, когда оно было выполнено, а также передаваемое сообщение. Не вся эта информация вам понадобится, поэтому Git предоставляет возможность вывести упрощенный вариант записи журнала:

```
git_practice$ git log --pretty=oneline
ee76419954379819f3f2cacafd15103ea900ecb2 (HEAD -> master) Started project.
git_practice$
```

Флаг `--pretty=oneline` выводит два важнейших атрибута: идентификатор закрепления и сохраненное для этого закрепления сообщение.

Второе закрепление

Чтобы увидеть всю мощь системы контроля версий, следует внести в проект изменение и закрепить его. Добавим еще одну строку в hello_git.py:

hello_git.py

```
print("Hello Git world!")
print("Hello everyone.")
```

Проверяя статус проекта, мы видим, что изменение в файле было замечено Git:

```
  git_practice$ git status
❶ On branch master
  Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)
```

```
❷ modified:   hello_git.py

❸ no changes added to commit (use "git add" and/or "git commit -a")
  git_practice$
```

В результатах указывается текущая ветвь ❶, имя измененного файла ❷, а также то, что изменения не были закреплены ❸. Закрепим изменения и снова проверим статус:

```
❶ git_practice$ git commit -am "Extended greeting."
  [master 51f0fe5] Extended greeting.
   1 file changed, 1 insertion(+), 1 deletion(-)
❷ git_practice$ git status
  On branch master
  nothing to commit, working tree clean
❸ git_practice$ git log --pretty=oneline
  51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master) Extended greeting.
  ee76419954379819f3f2cacafd15103ea900ecb2 Started project.
  git_practice$
```

Команда `git commit` с флагом `-am` выполняет новое закрепление ❶. Флаг `-a` приказывает Git добавить все измененные файлы в репозитории в текущее закрепление. (Если вы создали новые файлы между закреплениями, просто снова выполните команду `git add .`, чтобы включить новые файлы в репозиторий.) Флаг `-m` приказывает Git сохранить сообщение в журнале.

При проверке статуса проекта рабочий каталог снова оказывается чистым ❷. Наконец, в журнале хранится информация о двух закреплениях ❸.

Отмена изменений

А теперь посмотрим, как отменить изменение и вернуться к предыдущему работоспособному состоянию. Сначала добавьте в файл hello_git.py новую строку:

hello_git.py

```
print("Hello Git world!")
print("Hello everyone.")

print("Oh no, I broke the project!")
```

Сохраните и запустите этот файл.

Из информации статуса видно, что Git видит изменения:

```
  git_practice$ git status
  On branch master
  Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)
```

```
❶     modified:   hello_git.py

  no changes added to commit (use "git add" and/or "git commit -a")
  git_practice$
```

Git понимает, что файл hello_git.py был изменен ❶, и при желании мы можем закрепить изменения. Но на этот раз вместо закрепления мы хотим вернуться к последней операции закрепления, в которой проект был заведомо работоспособным. С hello_git.py ничего делать не нужно — ни удалять строку, ни использовать функцию отмены в текстовом редакторе. Вместо этого введите следующие команды в терминальном сеансе:

```
git_practice$ git checkout .
git_practice$ git status
On branch master
nothing to commit, working tree clean
git_practice$
```

Команда `git checkout` позволяет работать с любым предшествующим закреплением. Команда `git checkout .` отменяет все изменения, внесенные с момента последнего закрепления, и восстанавливает проект в последнем закрепленном состоянии.

Вернувшись в текстовый редактор, вы увидите, что файл hello_git.py вернулся к следующему состоянию:

```
print("Hello Git world!")
print("Hello everyone.")
```

И хотя в таком простом проекте возврат к предыдущему состоянию может показаться тривиальным, если бы мы работали над большим проектом с десятками измененных файлов, все файлы, измененные с момента последнего закрепления, вернулись бы к предыдущему состоянию. Эта возможность невероятно полезна: можете вносить любые изменения в ходе реализации новой функции, а если они не сработают, вы просто отменяете их без вреда для проекта. Не нужно запоминать эти изменения и отменять их вручную; Git делает все это за вас.

ПРИМЕЧАНИЕ Возможно, вам придется обновить файл в редакторе, чтобы увидеть предыдущую версию.

Извлечение предыдущих закреплений

Git позволяет извлечь из журнала любое закрепление, не только последнее. Для этого в команде вместо точки указываются первые 6 символов идентификатора. Выполняя извлечение, вы можете просмотреть более раннее закрепление и либо вернуться к своему последнему закреплению, либо отказаться от всей последней работы и продолжить разработку с более раннего закрепления:

```
git_practice$ git log --pretty=oneline
51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master) Extended greeting.
ee76419954379819f3f2cacafd15103ea900ecb2 Started project.
git_practice$ git checkout ee7641
Note: checking out 'ee7641'.

❶ You are in 'detached HEAD' state. You can look around, make experimental
  changes and commit them, and you can discard any commits you make in this
  state without impacting any branches by performing another checkout.

  If you want to create a new branch to retain commits you create, you may
  do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b <new_branch_name>

  HEAD is now at ee7641... Started project.
  git_practice$
```

При извлечении предыдущего закрепления вы покидаете ветвь `master` и входите в состояние, которое Git называет *отсоединенным состоянием HEAD* ❶. HEAD — текущее состояние проекта; «отсоединенным» оно называется потому, что мы покинули именованную ветвь (`master` в данном случае).

Чтобы вернуться к ветви `master`, извлеките ее:

```
git_practice$ git checkout master
Previous HEAD position was ee76419 Started project.
Switched to branch 'master'
git_practice$
```

Вы снова возвращаетесь к ветви `master`. Если только вы не собираетесь использовать расширенные возможности Git, лучше не вносить изменения в проект при извлечении старого закрепления. Но если над проектом больше никто не работает и вы хотите отбросить все более поздние закрепления и вернуться к предыдущему состоянию, можете вернуть проект к предыдущему закреплению. Работая в ветви `master`, введите следующую команду:

```
❶ git_practice$ git status
  On branch master
  nothing to commit, working directory clean
❷ git_practice$ git log --pretty=oneline
  51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master) Extended greeting.
  ee76419954379819f3f2cacafd15103ea900ecb2 Started project.
❸ git_practice$ git reset --hard ee76419
  HEAD is now at ee76419 Started project.
❹ git_practice$ git status
  On branch master
  nothing to commit, working directory clean
❺ git_practice$ git log --pretty=oneline
  ee76419954379819f3f2cacafd15103ea900ecb2 (HEAD -> master) Started project.
  git_practice$
```

Сначала мы проверяем статус и убеждаемся в том, что текущей является ветвь master ❶. В журнале присутствуют оба закрепления ❷. Затем выдается команда `git reset --hard` с первыми шестью символами идентификатора того закрепления, к которому нужно вернуться ❸. Далее повторная проверка статуса показывает, что мы находимся в главной ветви, а закреплять нечего ❹. Повторное обращение к журналу показывает, что проект находится в том состоянии, к которому мы решили вернуться ❺.

Удаление репозитория

Случается, что история репозитория нарушается и вы не знаете, как ее восстановить. Если это произойдет, для начала обратитесь за помощью к методам, описанным в приложении В. Если исправить ошибку не удалось, а вы работаете над проектом в одиночку, продолжайте работать с файлами, но сотрите историю проекта, удалив каталог .git. Это не повлияет на текущее состояние файлов, но приведет к удалению всех закреплений, так что вы потеряете возможность извлекать другие состояния проекта.

Для этого либо откройте программу для работы с файлами и удалите репозиторий .git, либо сделайте это в командной строке. После этого необходимо создать новый репозиторий, чтобы снова отслеживать изменения в проекте. Вот как выглядит весь процесс в терминальном сеансе:

```
❶ git_practice$ git status
  On branch master
  nothing to commit, working directory clean
❷ git_practice$ rm -rf .git
❸ git_practice$ git status
  fatal: Not a git repository (or any of the parent directories): .git
❹ git_practice$ git init
  Initialized empty Git repository in git_practice/.git/
❺ git_practice$ git status
  On branch master

  No commits yet

  Untracked files:
    (use "git add <file>..." to include in what will be committed)

    .gitignore
    hello_git.py

  nothing added to commit but untracked files present (use "git add" to track)
❻ git_practice$ git add .
  git_practice$ git commit -m "Starting over."
  [master (root-commit) 6baf231] Starting over.
   2 files changed, 4 insertions(+)
   create mode 100644 .gitignore
   create mode 100644 hello_git.py
❼ git_practice$ git status
```

```
On branch master
nothing to commit, working tree clean
git_practice$
```

Сначала мы проверяем статус и видим, что рабочий каталог чист ❶. Затем команда `rm -rf .git` (`rmdir /s .git` в Windows) удаляет каталог .git ❷. При проверке статуса после удаления каталога .git выдается сообщение об отсутствии репозитория Git ❸. Вся информация, используемая Git для отслеживания репозитория, хранится в каталоге .git, поэтому его удаление приводит к уничтожению всего репозитория.

Тогда мы можем использовать команду `git init` для создания нового репозитория ❹. Проверка статуса показывает, что все вернулось к исходному состоянию, ожидая первого закрепления ❺. Мы добавляем файлы и выполняем первое закрепление ❻. Теперь проверка статуса показывает, что проект находится в новой ветви `master`, а закреплять пока нечего ❼.

Работа с системой контроля версий потребует некоторых усилий, но стоит немного освоиться, и вам уже не захочется работать без нее.

Эрик Мэтиз

Изучаем Python: программирование игр, визуализация данных, веб-приложения

3-е издание

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Художественный редактор	*В. Мостипан*
Литературный редактор	*М. Рогожин*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

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

Дата изготовления: 07.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные профессиональные, технические и научные.

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

Подписано в печать 08.07.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 41,280. Доп. тираж. Заказ 0000.