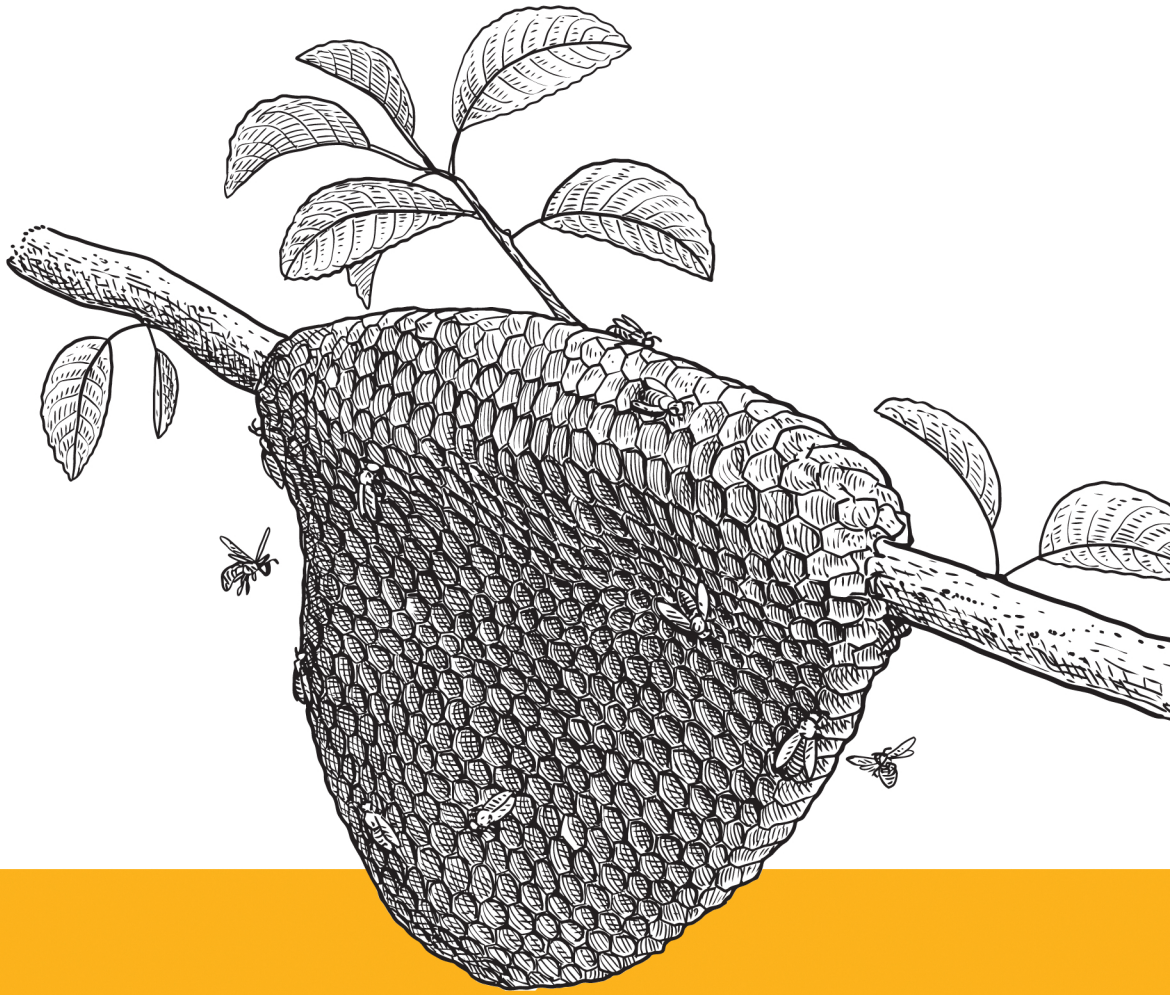
БИБЛИОТЕКА ПРОГРАММИСТА

Кеннеди Берман



ОСНОВЫ PYTHON ДЛЯ DATA SCIENCE

ОСВОЙТЕ PYTHON И СРАЗУ ПРИСТУПАЙТЕ К РЕШЕНИЮ ПРИКЛАДНЫХ ЗАДАЧ, СВЯЗАННЫХ С АНАЛИЗОМ ДАННЫХ

Foundational Python for Data Science

Kennedy R. Behrman

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

ОСНОВЫ PYTHON ДЛЯ DATA SCIENCE

Кеннеди Берман

Санкт-Петербург • Москва • Минск
2023

ББК 32.973.2-018.1
УДК 004.43
Б50

Берман Кеннеди

Б50 Основы Python для Data Science. — СПб.: Питер, 2023. — 272 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2251-6

Python — язык программирования № 1 для машинного обучения и Data Science. Но как же сложно решить, с чего начать изучение Python, ведь у него огромный инструментарий! Кеннеди Берман фокусируется на тех навыках программирования, которые понадобятся вам для решения задач в области Data Science и машинного обучения.

Вы познакомитесь с блокнотами Jupyter — лучшей средой для профессиональной работы с данными. После этого перейдете к ключевым библиотекам, которые упрощают процесс математических вычислений, визуализации, решение задач машинного обучения и обработки естественного языка. После этого, овладев основами, вы перейдете к продвинутым техникам, позволяющим решать более сложные задачи.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-0136624356 англ.

ISBN 978-5-4461-2251-6

Authorized translation from the English language edition, entitled Foundational Python for Data Science, 1st Edition, by Kennedy Behrman, published by Pearson Education, Inc, publishing as Addison Wesley Professional

Краткое содержание

Оглавление

ЧАСТЬ II
БИБЛИОТЕКИ DATA SCIENCE

Посвящается Татьяне, Итте и Мэйпл,
которая, кажется,
все еще прячется под моей кроватью.

Предисловие

Язык Python создан давно и применяется в самых различных сферах. Изначально он задумывался Гвидо ван Россумом в 1989 году в качестве инструмента для системного администрирования, как альтернатива Bash-скриптам и программам на C[1]. С момента выхода в 1991 году Python эволюционировал и стал использоваться в разных сферах: от веб-разработки, кино, управления и науки до бизнеса[2].

Впервые я познакомился с Python, работая в киноиндустрии. Мы применяли его в разных отделах для управления данными и их автоматизации. В последнее же десятилетие этот язык стал доминирующим инструментом для дата-сайентистов.

Этому поспособствовали две разработки: Jupyter Notebook и мощные сторонние библиотеки. В 2001 году Фернандо Перес создал проект IPython, интерактивную оболочку Python, вдохновленную блокнотами Maple и Mathematica[3]. К 2014 году часть проекта, посвященная им, была выделена в проект Jupyter. Эти блокноты отлично подходят для научной и статистической работы.

В это же время под Python создавались сторонние библиотеки для научных и статистических вычислений. Увеличение числа приложений значительно повысило функциональные возможности, доступные программисту Python. Благодаря специализированным пакетам для решения разных задач, от открытия веб-сокетов до обработки исходных текстов, начинающему разработчику доступно даже больше, чем нужно.

Этот проект был детищем Ноя Гифта[4]. Работая преподавателем, он обнаружил, что у студентов, записавшихся на курс Data Science, недостаточно ресурсов для

---

1 https://docs.python.org/3/faq/general.html#why-was-python-created-in-the-first-place.

2 https://www.python.org/success-stories/.

3 http://blog.fperez.org/2012/01/ipython-notebook-historical.html.

4 https://noahgift.com.

изучения тех частей Python, которые им нужны. Было много общих книг по Python и Data Science, но не было литературы для изучения основ Python именно в рамках Data Science. Это я и попытался реализовать здесь. Моя книга не будет обучать Python для настройки веб-страниц или выполнения системного администрирования. Она не предназначена и для обучения Data Science. Здесь будет рассказано об основах Python, необходимых для изучения науки о данных.

Я надеюсь, это руководство окажется для вас полезным и позволит вам расширить свои знания в области Data Science.

Примеры кода

Большую часть кода из примеров в этой книге можно найти на сайте GitHub по адресу https://github.com/kbehrman/foundation-python-for-data-science.

Благодарности

Идея этой книги принадлежит Ною Гифту. Именно он определил необходимость специализированного введения в Python для студентов, изучающих Data Science. Спасибо тебе, Ной. Отдельная благодарность Колину Эрдману, который, будучи научным редактором, обратил внимание на важные детали. Спасибо команде издательства Pearson, в том числе Малобике Чакраборти, которая взяла на себя руководство моим проектом, Марку Ренфроу, который помог реализовать его, и Лоре Левин, которая помогла его запустить.

Об авторе

Кеннеди Берман — опытный инженер-программист. Он внедрил Python в процессы управления цифровыми активами в индустрии визуальных эффектов и с тех пор активно использует его. Кеннеди — автор множества книг и программ по обучению Python. Сейчас он работает старшим специалистом по инженерии данных в Envestnet.

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

PYTHON В JUPYTER NOTEBOOK

1

Введение в Jupyter Notebook

Все животные равны. Но некоторые равнее других.

Джордж Оруэлл

В этой главе

- Выполнение операторов Python.
- Блокноты Jupyter.
- Блокноты в Google Colab.
- Ячейки с кодом и текстом.
- Загрузка файлов в Colab.
- Использование системного псевдонима для запуска команд из оболочки.
- Магические функции.

В этой главе описана среда Jupyter Notebook от Google Colab, отлично подходящая для новичков, начинающих разработку на Python. Здесь я рассмотрю традиционные способы программирования на этом языке.

Выполнение операторов в Python

Как правило, Python-код выполнялся либо в интерактивной оболочке Python, либо через предоставление текстовых файлов интерпретатору. Если этот язык уже установлен в системе, то можно открыть его встроенную интерактивную оболочку, набрав `python` в командной строке:

```
python
Python 3.9.1 (default, Mar 7 2021, 09:53:19)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

> **ПРИМЕЧАНИЕ**
>
> В этой книге полужирный шрифт применяется для оформления пользовательского ввода (код, который вы вводите), а не получаемого результата.

Затем вы можете ввести оператор Python и выполнить его, нажав Enter:

```
print("Hello")
Hello
```

Как мы видим, результат каждого оператора отображается сразу после его ввода, строкой ниже.

Когда команды Python хранятся в текстовом файле с расширением .py, можно запустить их в командной строке, набрав `python` перед именем файла. Например, если у вас есть файл `hello.py` с оператором `print("Hello")`, вы можете вызвать его в командной строке и просмотреть вывод следующим образом:

```
python hello.py
Hello
```

В стандартных программных проектах на Python интерактивная среда подходит для изучения синтаксиса и проведения простых экспериментов. Выполнение кода из файлов — тот способ, с помощью которого действительно выполняется разработка

и пишется программное обеспечение (ПО). Такие файлы можно распространять в любой среде, необходимой для запуска кода. Но для научных вычислений такие решения не подходят. Ученые хотели интерактивно взаимодействовать с данными, сохраняя возможность работать с ними в формате документа. Для восполнения этого пробела появилась разработка на основе блокнотов.

Блокноты Jupyter

IPython — это более функциональная версия интерактивной оболочки Python. Проект Jupyter возник из IPython. Он сочетает в себе интерактивный характер оболочки Python с постоянством формата, основанного на документах. Блокнот Jupyter — это исполняемый документ, где исполняемый код сочетается с отформатированным текстом. Он состоит из *ячеек*, содержащих код или текст. Когда ячейка кода выполняется, вывод отображается под ней. Любые изменения состояния, выполняемые кодовой ячейкой, используются всеми ячейками, выполняемыми впоследствии.

Это значит, что вы можете создавать свой код ячейка за ячейкой без необходимости повторного запуска всего документа при внесении изменений. Особенно полезным это может быть при исследовании данных и экспериментах с ними.

Блокноты Jupyter широко применяются для работы в data science. Вы можете запускать их со своего компьютера или из облачных сервисов (AWS, Kaggle, Databricks или Google).

Блокноты Google Colab

Colab (сокращение от Colaboratory) — облачный сервис блокнотов Google — это отличный инструмент для начала работы с Python. Вам не нужно ничего устанавливать, иметь дело с зависимостями библиотек или управлением средой. В этой книге для всех примеров используются блокноты Colab. Для использования этого сервиса войдите в учетную запись Google и перейдите на сайт https://colab.research.google.com. (рис. 1.1). Там вы можете создавать новые блокноты или открывать существующие. Последние могут включать в себя примеры, предоставленные Google, созданные вами ранее или скопированные на ваш Google Диск.

Новый блокнот откроется на отдельной вкладке браузера.

Первый созданный вами блокнот по умолчанию будет называться `Untitled0.ipynb`. Чтобы его переименовать, дважды щелкните на заголовке и введите новое имя (рис. 1.2).

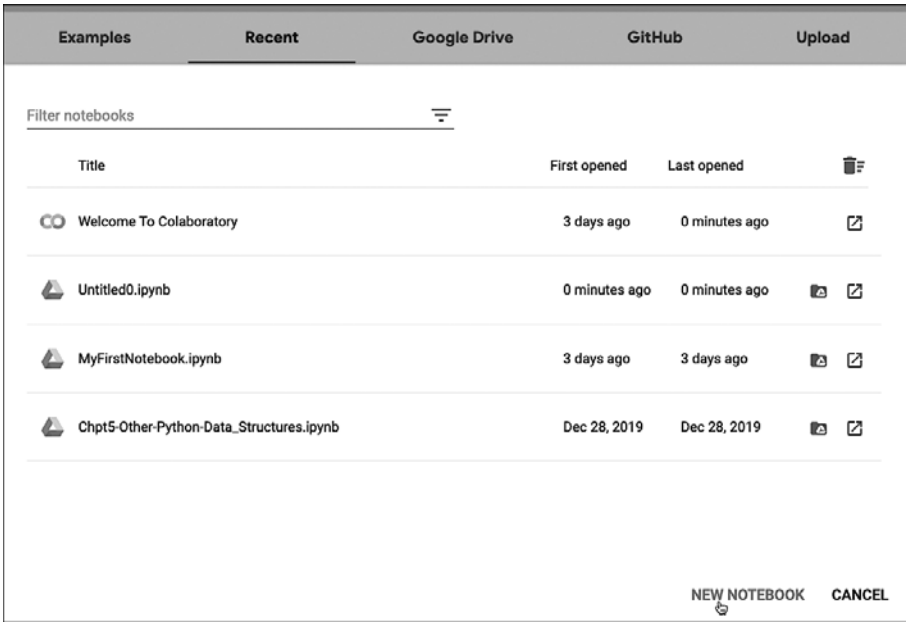


Рис. 1.1. Начало взаимодействия с Google Colab

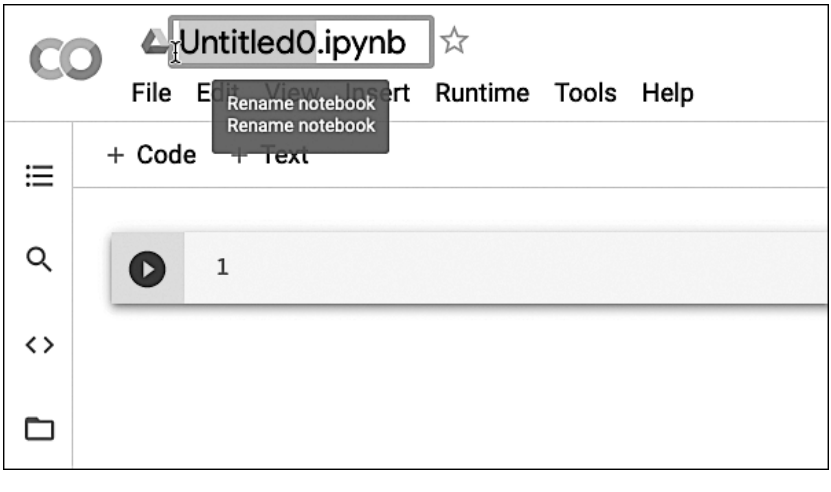


Рис. 1.2. Переименование блокнота в Google Colab

Colab автоматически сохраняет ваши блокноты на Google Диск, доступ к которому можно получить, перейдя на сайт Drive.Google.com. Расположением по умолчанию будет каталог Colab Notebooks (рис. 1.3).

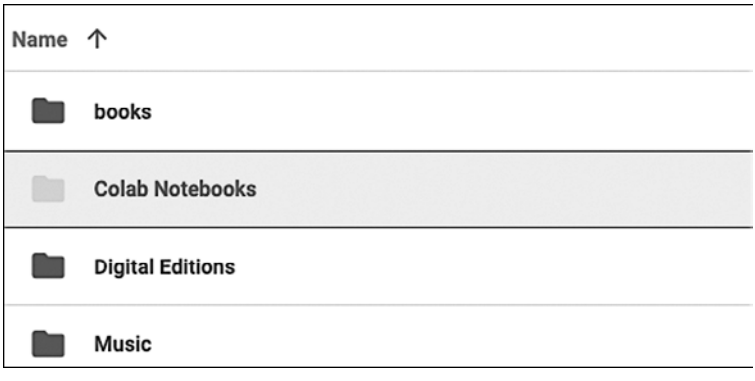


Рис. 1.3. Папка Colab Notebooks на Google Диске

Текстовые ячейки Colab

В новом блокноте Google Colab есть одна ячейка. Она может быть двух типов: текст или код. Добавлять новые ячейки можно с помощью кнопок +Code и +Text в левом верхнем углу интерфейса.

Для форматирования текстовых ячеек используется язык Markdown (см. https://colab.research.google.com/notebooks/markdown_guide.ipynb). Чтобы изменить ячейку, дважды щелкните на ней кнопкой мыши, после чего справа откроется область предварительного просмотра (рис. 1.4).

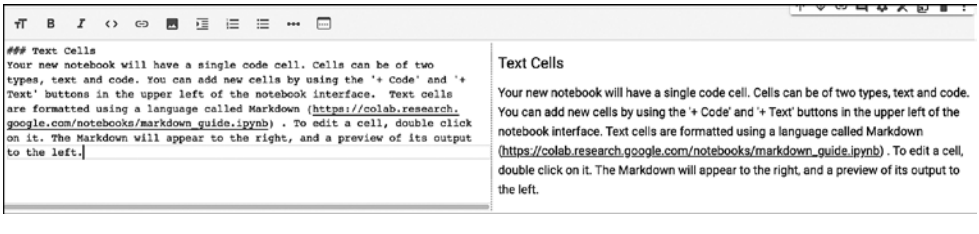


Рис. 1.4. Редактирование текстовых ячеек в блокноте Google Colab

Вы можете сделать текст в блокноте полужирным, курсивным, зачеркнутым и моноширинным (рис. 1.5).

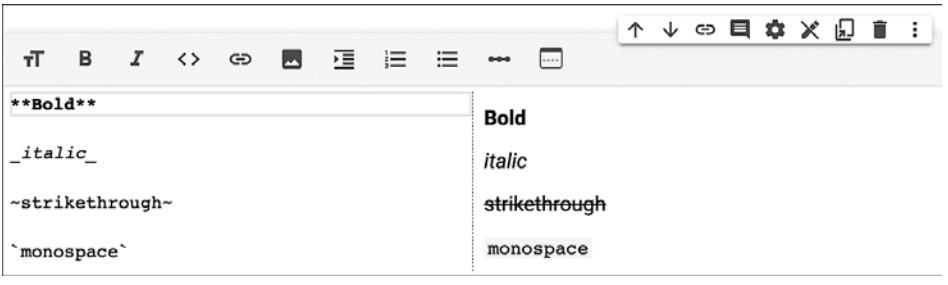


Рис. 1.5. Форматирование текста в блокноте Google Colab

Нумерованный список можно создать, указав числа перед элементами, а маркированный — поставив перед ними звездочки (рис. 1.6).

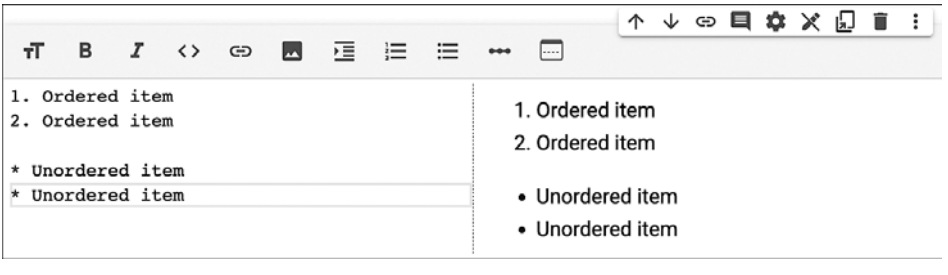


Рис. 1.6. Создание списков в блокноте Google Colab

Вы можете создавать заголовки, добавляя перед текстом символ решетки (#). Один символ решетки создает заголовок верхнего уровня, два — первого уровня и т. д. (рис. 1.7).

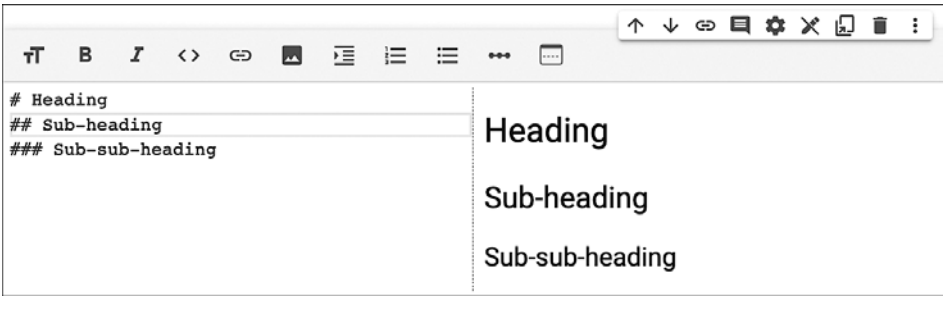


Рис. 1.7. Создание заголовков в блокноте Google Colab

Заголовок вверху ячейки определяет иерархию в документе. Посмотреть ее можно, открыв оглавление. Для этого нажмите кнопку Menu в левом верхнем углу интерфейса блокнота (рис. 1.8).

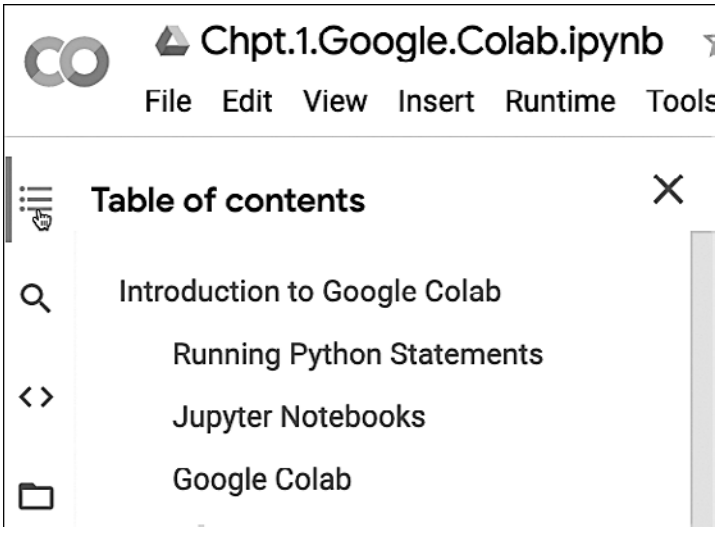


Рис. 1.8. Оглавление в блокноте Google Colab

С помощью оглавления вы можете перемещаться по документу, щелкая на отображаемых заголовках. У ячейки заголовка с дочерними ячейками есть треугольник рядом с текстом заголовка. Вы можете щелкнуть на нем, чтобы скрыть или просмотреть ячейки (рис. 1.9).

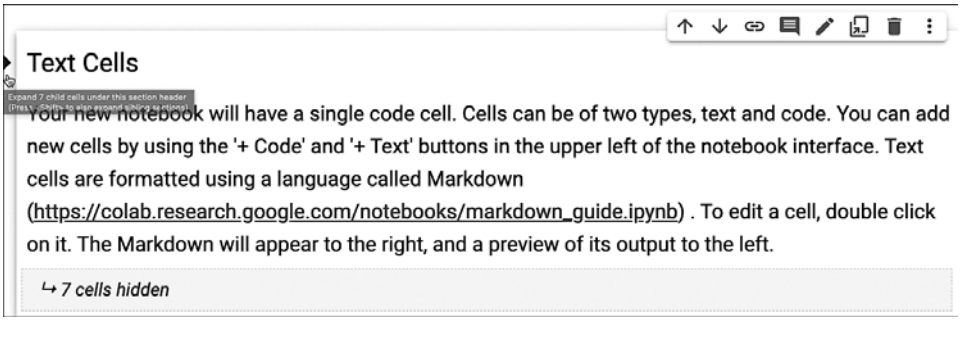


Рис. 1.9. Скрытие ячеек в блокноте Google Colab

LaTeX

Язык LaTeX (https://www.latex-project.org/about/), предназначенный для подготовки технических документов, применим и для представления математических выражений. LaTeX позволяет сосредоточиться на содержании, а не на внешнем виде. Вы можете вставить код LaTeX в текстовые ячейки блокнота Colab, окружив его знаками доллара (рис. 1.10).

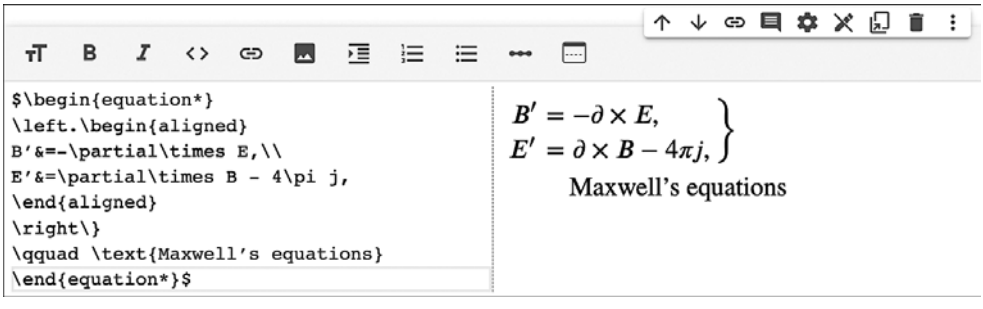


Рис. 1.10. LaTeX, встроенный в блокнот Google Colab

Ячейки кода Colab

В блокнотах Google Colab вы используете ячейки для написания и выполнения кода. Чтобы выполнить оператор Python, введите его в ячейку кода и либо нажмите Play слева от ячейки, либо используйте сочетание клавиш Shift+Enter. Последнее переместит вас к следующей ячейке или создаст новую при отсутствии следующих. Результат выполнения кода отображается под ячейкой:

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

В дальнейших примерах используются только ячейки кода для блокнотов Colab.

Файлы Colab

Для просмотра доступных в Colab файлов и папок нажмите Files в левой части интерфейса (рис. 1.11). По умолчанию у вас есть доступ к папке sample_data, предоставленной Google.

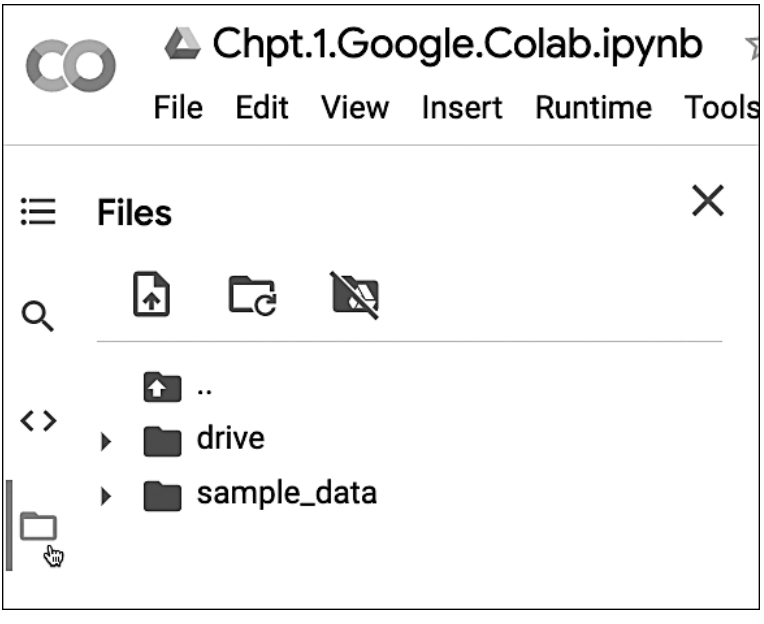


Рис. 1.11. Просмотр файлов в Google Colab

Вы можете нажать Upload, чтобы загрузить файлы в текущий сеанс (рис. 1.12).

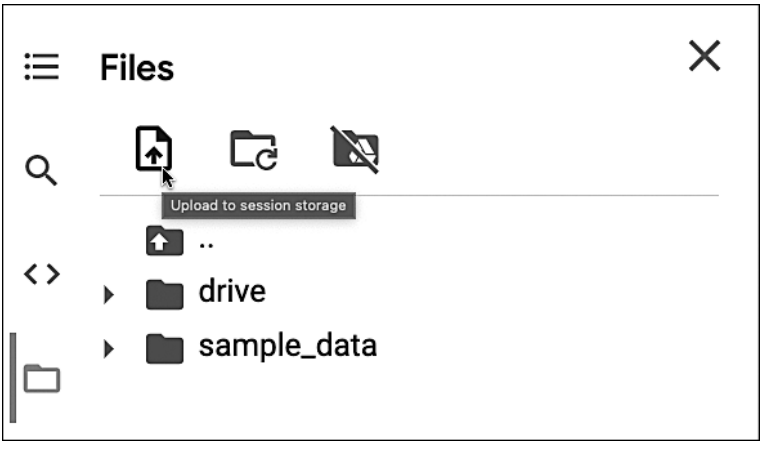


Рис. 1.12. Загрузка файлов в Google Colab

Загружаемые файлы доступны только в текущем сеансе вашего документа. Возвращаясь к нему позднее, вы должны будете загрузить их снова. Все файлы, доступные в Colab, имеют корневой путь `/content/`. Например, при загрузке файла с именем `heights.over.time.csv` его путь будет — `/content/heights.over.time.csv`.

Вы можете подключить свой Google Диск, нажав Mount Drive (рис. 1.13). Его содержимое имеет корневой путь `/content/drive`.

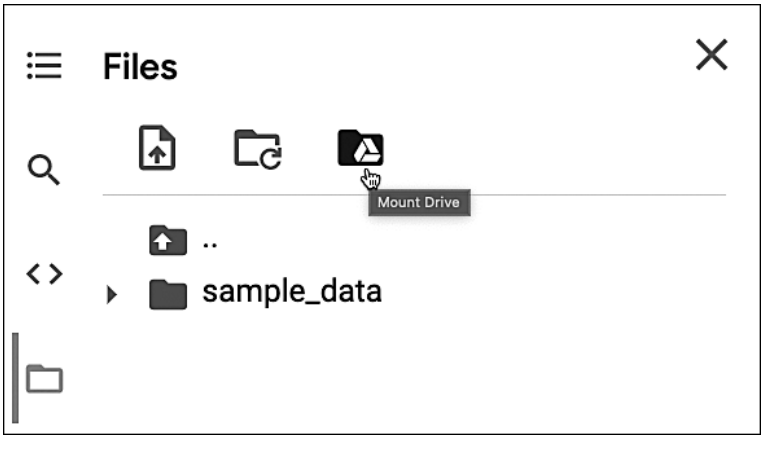


Рис. 1.13. Подключение Google Диска

Управление документами Colab

По умолчанию блокноты сохраняются на вашем Google Диске. В меню File можно выбрать другие варианты сохранения (на GitHub или в виде структур gist либо в виде отслеживаемых файлов или в формате блокнота Jupyter (с расширением `.ipynb`) и в виде файлов Python (с расширением `.py`)). Вы можете поделиться блокнотами, нажав Share в правом верхнем углу интерфейса.

Фрагменты кода Colab

С помощью Code Snippets в левой части интерфейса Colab можно искать и выбирать части кода (рис. 1.14). Вставить выбранные фрагменты можно, нажав Insert. Использование фрагментов кода — отличный способ узнать о возможностях Colab, в том числе о создании интерактивных форм, загрузке данных и использовании разных параметров визуализации.

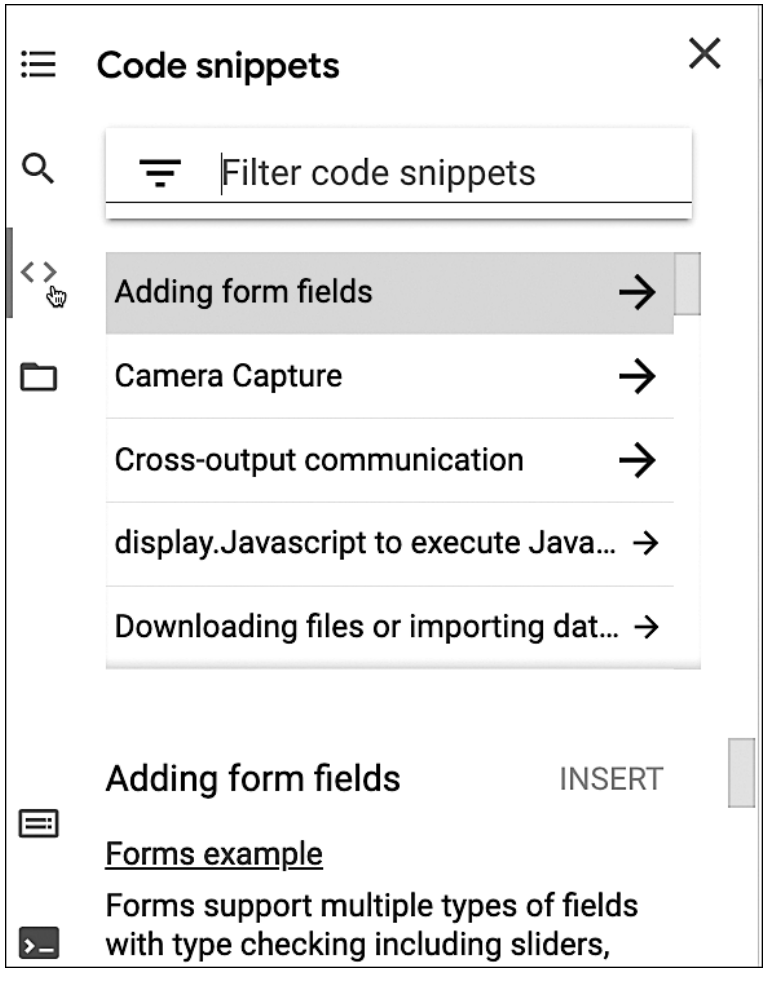


Рис. 1.14. Использование фрагментов кода в Google Colab

Существующие коллекции

Вы можете использовать блокноты Google Colab для объяснения и демонстрации методов, концепций и рабочих процессов. Работы по Data Science есть во многих коллекциях блокнотов в Интернете. У Kaggle (https://www.kaggle.com/code) и Google Seedbank (https://research.google.com/seedbank/) много таких.

Системные псевдонимы

Запустить команду оболочки из ячейки блокнота Colab можно, поставив перед ней восклицательный знак. Следующий пример выводит рабочий каталог:

```
!pwd
/content
```

Вы можете передать любой вывод команды оболочки в переменную Python, как показано здесь, и использовать ее в последующем коде:

```
var = !ls sample_data
print(var)
```

ПРИМЕЧАНИЕ

Пока не беспокойтесь о переменных. Мы поговорим о них в главе 2.

Магические функции

Магические функции изменяют способ выполнения кода в ячейке. Например, вы можете рассчитать время выполнения оператора Python с помощью магической функции `%timeit()`:

```
import time
%timeit(time.sleep(1))
```

Магическая функция `%%html` поможет запускать HTML-код с помощью ячейки:

```
%%html
<marquee style=''width: 30%; color: blue;'><b>Whee!</b></marquee>
```

ПРИМЕЧАНИЕ

Дополнительную информацию о магических функциях можно найти в примерах блокнотов Cell Magics, которые являются частью документации Jupyter, по ссылке https://nbviewer.jupyter.org/github/ipython/ipython/blob/1.x/examples/notebooks/Cell%20Magics.ipynb.

Резюме

Блокноты Jupyter — это документы, где форматированный текст сочетается с исполняемым кодом. Они приобрели популярность в научной среде, и многие примеры можно найти в Интернете. Google Colab предлагает готовые блокноты и содержит множество популярных библиотек, используемых в Data Science. Блокнот состоит из текстовых ячеек, отформатированных в Markdown, и кодовых ячеек, которые могут выполнять код Python. В следующих главах мы рассмотрим много примеров блокнотов Colab.

Вопросы для закрепления

1. Какие блокноты размещены в Google Colab?
2. Какие типы ячеек доступны в Google Colab?
3. Как вы подключаете свой Google Диск к Colab?
4. Какой язык используется в ячейках кода Google Colab?

2

Основы Python

Все модели неверны, но некоторые полезны.

Джордж Бокс

В этой главе

- Встроенные типы Python.
- Знакомство с операторами.
- Выражения.
- Утверждения (для отладки кода).
- Операторы присваивания и переменные.
- Операторы импорта.
- Вывод данных.
- Базовые математические операции.
- Точечная нотация.

Здесь рассматриваются важные сведения, которые пригодятся вам для создания программ на Python. Глава знакомит с основными встроенными типами данных — целыми числами и строками. Вы узнаете о простых операторах, которые можно использовать для управления действиями вашего компьютера. В главе 2 рассматриваются операторы, присваивающие значения переменным, а также те, что помогут убедиться, что код выполняется как положено. Еще мы обсудим, как импортировать модули для расширения функциональности, доступной в вашем коде. К концу этой главы у вас будет достаточно знаний для написания программы, выполняющей простые математические операции над сохраненными значениями.

Основные типы в Python

В биологии принята классификация живых существ в иерархии от домена и царства до рода и вида. Чем ниже вы спускаетесь по этой иерархии, тем больше похожи формы жизни, которые входят в группу. Подобная иерархия есть и в Data Science.

Синтаксический анализатор — это программа, которая принимает ваш код в качестве входных данных и переводит его в инструкции для компьютера. Синтаксический анализатор Python разбивает ваш код на токены (опознавательные знаки), имеющие особое значение, определенное для языка Python. Полезно классифицировать эти токены по общему поведению и атрибутам, как это делают биологи с живыми организмами.Такие группы в Python называются *коллекциями* и *типами*. Есть типы, встроенные в сам язык или определенные разработчиками вне его ядра. На высоком уровне документация Python (https://docs.python.org/3/library/stdtypes.html) определяет основные встроенные типы как числа, последовательности (см. главу 3), сопоставления (см. главу 4), классы (см. главу 14), экземпляры (см. главу 14) и исключения. На низком уровне есть такие основные встроенные типы:

- *числовые*: логические значения, числа с плавающей точкой, целые и мнимые числа;
- *последовательности*: строки и двоичные строки.

В простейшем случае целые числа (`int`, от англ. integer — «целое число») представляются в коде как обычные цифры. Числа с плавающей точкой представлены в виде группы цифр, включая разделитель — точку. Вы можете использовать функцию `type`, чтобы увидеть тип целого числа и числа с плавающей точкой:

```
type(13)
int

type(4.1)
float
```

Если вы хотите, чтобы число было с плавающей точкой, убедитесь, что у него есть точка и число справа; но если число справа равно нулю, его можно не писать, оставив лишь точку:

```
type(1.0)
float
```

Логические (или булевы) значения представлены двумя константами, `True` и `False`, обе из которых возвращают логический тип, «за кулисами» являющийся специализированной формой `int`:

```
type(True)
bool

type(False)
bool
```

Строка состоит из символов, заключенных в кавычки. Вы можете использовать строки для представления разного текста с множеством различных применений. Например:

```
type("Hello")
str
```

ПРИМЕЧАНИЕ

Вы узнаете гораздо больше о строках в главе 3.

Специальный тип `NoneType` имеет только одно значение `None`. Он используется для представления чего-то, у чего нет значения:

```
type(None)
NoneType
```

Высокоуровневые и низкоуровневые языки

Написание программного обеспечения (ПО) подразумевает просто передачу компьютеру инструкций. Хитрость в том, чтобы перевести действия, понятные человеку, в форму, понятную компьютеру. Языки программирования варьируются от очень близких к тому, как компьютер понимает логику (низкоуровневые), до более близких к человеческому языку (высокоуровневые). Машинный код и язык ассемблера — примеры *языков низкого уровня*. С ними у вас есть полный контроль над тем, что именно делает процессор вашего компьютера, но писать код на них сложно и долго.

Языки высокого уровня объединяют группы инструкций в более крупные функциональные блоки. У них есть ряд преимуществ. Например, язык C, находящийся на нижней ступени высокоуровневых языков, позволяет напрямую управлять использованием памяти программой и писать высокооптимизированное ПО, необходимое для встраиваемых систем. Python же находится на верхней ступени. Вы не можете прямо сказать, сколько памяти использовать для сохранения ваших данных или освободить по окончании. Синтаксис Python очень близок к логике человеческого языка. Код на нем проще понять и написать, чем на низкоуровневых языках. Перевод действий с человеческого языка на Python — быстрый и интуитивно понятный процесс.

Операторы

Программа Python состоит из операторов. Каждый можно рассматривать как действие, которое должен выполнить компьютер. Если представить программу как рецепт из поваренной книги, то оператор — это отдельная инструкция, например «взбить яичные желтки, пока они не побелеют» или «выпекать 15 минут».

В самом простом случае оператор Python — это одна строка кода, конец которой означает конец оператора. Простой оператор может, например, вызвать одну функцию, как здесь:

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

Оператор может быть и более сложным. Например, следующий оператор, который оценивает условия и присваивает переменную на основе этой оценки:

```
x,y = 5,6
bar = x**2 if (x < y) and (y or z) else x//2
```

Python допускает как простые, так и сложные операторы. Простые включают в себя выражения (expressions), утверждения (assert), присваивания (assign), передачу (pass), удаление (delete), возврат (return), остановку генератора (yield), вызов исключения (raise), прерывание (break), продолжение (continue), импорт (import) а также предстоящие (future), глобальные (global) и нелокальные (nonlocal) операторы. В этой главе рассматриваются некоторые из них, а в последующих главах — большинство остальных. Главы 5 и 6 посвящены сложным операторам.

Множественные операторы

Хотя для объявления программы достаточно использовать одиночный оператор, наиболее полезные программы состоят из множественных. Результаты одного оператора могут использоваться последующими операторами, создавая

функциональность через объединение действий. Например, вы можете задействовать следующий оператор, чтобы присвоить переменной результат целочисленного деления, использовать этот результат для вычисления значения другой переменной и применить обе переменные в третьем операторе в качестве входных данных для `print`:

```
x = 23//3
y = x**2
print(f"x is {x}, y is {y}")
x is 7, y is 49
```

Выражения

Выражение в Python — это фрагмент кода, результат которого — определенное значение (или `None`). Оно может быть, среди прочего, математическим выражением или вызовом функции/метода. Это простой оператор, который содержит выражение, но не фиксирует его вывод для дальнейшего использования. Выражения обычно полезны только в таких интерактивных средах, как оболочка IPython, где результат выражения отображается пользователю после его выполнения. Таким образом, если, находясь в оболочке, вы захотите узнать, что возвращает функция или чему равно 12 344, разделенное на 12, вы можете увидеть результат без написания кода для его вывода.

Вы можете использовать выражение и для просмотра значения переменной (см. пример ниже) или просто для отображения значения любого типа. Вот несколько простых выражений и вывод каждого из них:

```
23 * 42
966

"Hello"
'Hello'

import os
os.getcwd()
'/content'
```

В этой книге вы найдете несколько выражений, раскрывающих функционал Python. В каждом случае сначала будет идти выражение, а в следующей строке — его результат.

Утверждения

Утверждения принимают выражение в качестве аргумента и гарантируют, что результат оценивается как `True`. Выражения, возвращающие `False`, `None`, ноль, пустые

контейнеры и пустые строки оцениваются как `False`. Остальные значения оцениваются как `True` (подробнее о контейнерах см. в главах 3 и 4). Утверждение выдает ошибку, если выражение оценивается как `False`:

```
assert(False)
------------------------------------------------------------
AssertionError            Traceback (most recent call last)
<ipython-input-5-8808c4021c9c> in <module>()
----> 1 assert(False)
```

В противном случае утверждение вызывает выражение и переходит к следующему оператору:

```
assert(True)
```

Вы можете использовать утверждения при отладке. Это позволяет убедиться, что некоторые условия, которые, как вы считаете, должны быть истинными, действительно такими являются. Но эти операторы влияют на производительность. Поэтому если вы часто их используете при разработке, то можете отключить их для кода в реальной среде. Для этого при запуске своего кода из командной строки добавьте флаг `-o` (optimize):

```
python -o my_script.py
```

Операторы присваивания

Переменная — это имя, указывающее на некоторую часть данных. Важно понимать, что в операторе присваивания она указывает на данные, но не является ими. Одна переменная может указывать на разные данные, даже если они разных типов.

Вы можете поменять данные, на которые указывает переменная, не изменяя ее саму. Как и ранее в примерах, переменной присваивается значение с помощью оператора присваивания (один знак равенства).

Имя переменной отображается слева от оператора, а значение — справа. Ниже показано, как присвоить значение `12` переменной `x` и строку `Hello` переменной `y`:

```
x = 12
y = 'Hello'
```

После этого можно использовать имена переменных вместо значений. Так, вы можете выполнять математические операции с помощью переменной `x` или использовать переменную `y` для построения большего фрагмента текста:

```
answer = x - 3
print(f"{y} Jeff, the answer is {answer}")
Hello Jeff, the answer is 9
```

Здесь значения `x` и `y` используются там, где были вставлены переменные. Вы можете присвоить несколько значений нескольким переменным в одном операторе, разделив имена и значения запятыми:

```
x, y, z = 1,'a',3.0
```

Здесь переменной `x` присваивается значение `1`, `y` — значение `'a'`, а `z` — `3.0`.

Рекомендуется присваивать переменным имена, которые помогают объяснить их использование. Например, использование `x` для значения на оси *X* графика — это нормально, но `x` для обозначения имени клиента сбивает с толку — `first_name` было бы гораздо понятнее.

Оператор pass

Оператор `pass` — это заполнитель. Сам он ничего не делает, но, когда есть код, требующий синтаксической корректности оператора, можно использовать этот оператор. Он состоит только из слова `pass` и обычно используется для заглушки функций и классов при проектировании кода (для ввода имен без функциональности). Вы узнаете больше о функциях в главе 6, а о классах — в главе 14.

Оператор del

Такой оператор удаляет что-то из работающей программы. Он состоит из слова `del`, за которым следует удаляемый элемент в скобках. После удаления элемента на него нельзя будет ссылаться снова, если только он не будет переобъявлен. Ниже показано, как присваивается и удаляется значение для переменной:

```
polly = 'parrot'
del(polly)
print(polly)
---------------------------------------------------------
NameError                 Traceback (most recent call last)
<ipython-input-6-c0525896ade9> in <module>()
      1 polly = 'parrot'
      2 del(polly)
----> 3 print(polly)

NameError: name 'polly' is not defined
```

Здесь, при попытке получить доступ к переменной с помощью функции `print` возникает ошибка.

ПРИМЕЧАНИЕ

У Python есть своя система «сборки мусора». Как правило, вам не нужно удалять объекты, чтобы освободить память, но бывают случаи, когда вы захотите их удалить принудительно.

Оператор return

`return` определяет возвращаемое значение функции. Вы увидите, как писать функции с использованием операторов `return` в главе 6.

Оператор yield

`yield` используется при написании функций-генераторов, обеспечивающих оптимизацию производительности и использования памяти. Мы рассмотрим генераторы в главе 13.

Оператор raise

В некоторых примерах этой главы показан код, вызывающий ошибки. Такие ошибки, возникающие во время выполнения программы (в отличие от тех, что возникают в синтаксисе и препятствуют ее запуску), называются исключениями. Они прерывают нормальное выполнение программы и, если они не обработаны, приводят к ее аварийному завершению.

`raise` используется как для повторного вызова найденного *исключения*, так и для инициализации встроенного или разработанного специально для вашей программы. У Python много встроенных исключений, охватывающих множество разных вариантов использования (https://docs.python.org/3/library/exceptions.html#bltin-exceptions). Для вызова одного из них можно использовать оператор `raise`. Он состоит из слова `raise`, за которым следует исключение. Например, `NotImplementedError` — это ошибка, используемая в иерархиях классов для указания того, что дочерний класс должен реализовать метод (см. главу 14).

В следующем примере эта ошибка вызывается оператором `raise`:

```
raise NotImplementedError
------------------------------------------------------------
NotImplementedError       Traceback (most recent call last)
<ipython-input-1-91639a24e592> in <module>()
----> 1 raise NotImplementedError
```

Оператор break

Используйте `break` для завершения цикла до выполнения его нормального условия. Циклы и оператор `break` рассматриваются в главе 5.

Оператор continue

Используйте `continue`, чтобы пропустить одну итерацию цикла. Этот оператор тоже рассматривается в главе 5.

Операторы import

Одна из самых мощных функций написания ПО — это возможность повторного использования фрагментов кода в разных контекстах. Код Python можно сохранять в файлах (с расширением `.py`). Те из них, что предназначены для многократного использования, называются *модулями*.

При запуске Python в интерактивном сеансе или как отдельной программы некоторые функции доступны в составе языка. Это значит, что вы можете использовать их напрямую, без дополнительной настройки. После установки Python становятся доступны его основные функции и стандартная библиотека. Последняя представляет собой серию модулей, которые вы можете добавить в сеанс Python для расширения функционала. Для доступа к одному из этих модулей в вашем коде используется оператор, состоящий из слова `import` и имени импортируемого модуля. Ниже показано, как импортировать модуль `os`, применяемый для взаимодействия с операционной системой:

```
import os
```

После импорта `os` вы можете использовать его функциональность так же, как если бы он был встроенным. В этом модуле есть функция `listdir`, которая выводит список содержимого текущего каталога:

```
os.listdir()
['.config', 'sample_data']
```

Когда модули или группы модулей подходят для более широкого применения, они называются *пакетами*. Один из привлекательных аспектов Python, особенно для Data Science, — большая экосистема сторонних пакетов. Они могут быть локальными для вас и вашей организации, но большинство общедоступных пакетов размещены в индексе пакетов Python — pypi.org. Чтобы использовать один из них, сначала установите его с помощью `pip` — стандартного менеджера пакетов Python.

Например, чтобы установить известную полезную библиотеку Pandas для локального использования, вы запускаете в командной строке следующее:

```
pip install pandas
```

Затем импортируете это в свой код:

```
import pandas
```

Вы можете присвоить модулю псевдоним во время импорта. Например, Pandas принято импортировать как `pd`:

```
import pandas as pd
```

Далее можно ссылаться на модуль, используя его псевдоним, а не имя:

```
pd.read_excel('/some_excel_file.xls')
```

Вы можете импортировать определенные части модуля с помощью `from` в строке импорта:

```
from os import path
path
<module 'posixpath' from '/usr/lib/python3.6/posixpath.py'>
```

Здесь путь к подмодулю импортируется из модуля `os`. Теперь вы можете использовать `path` в своей программе, как если бы он был определен вашим кодом.

Оператор __future__

`__future__` позволяет использовать определенные модули, являющиеся частью будущей версии Python. В моей книге этот оператор не рассматривается, так как редко используется дата-сайентистами.

Операторы global

Область видимости в программе относится к среде, разделяющей объявления имен и значений. Вы уже видели, что при объявлении переменной в операторе присваивания она сохраняет свое имя и значение для будущих операторов. Эти операторы *глобальны* — имеют широкий охват.

Когда вы начнете писать функции (в главе 6) и классы (в главе 14), то столкнетесь с неразделяемыми областями видимости. Применение `global` — способ совместного использования переменных в разных областях (подробнее о глобальных операторах см. в главе 13).

Оператор nonlocal

Применение `nonlocal` — еще один способ совместного использования переменных в пределах области видимости. В то время как `global` — общая переменная для всего модуля, `nonlocal` охватывает текущую область. Такие операторы полезны только с несколькими вложенными областями видимости. Они используются редко, поэтому в этой книге не рассматриваются.

Оператор print

Работая в такой интерактивной среде, как оболочка Python, IPython или блокнот Colab, вы можете использовать операторы выражения, чтобы увидеть значение любого выражения Python (выражение — это фрагмент кода, результатом которого будет значение).

Иногда вам может понадобиться вывести текст другими способами. Например, при запуске программы в командной строке или облачной функции. Тогда самый простой способ отобразить вывод — использовать оператор `print`. По умолчанию он выводит текст в поток стандартного вывода. Вы можете передать любой из встроенных типов или большинство других объектов как аргументы `print`. Рассмотрим примеры:

```
print(1)
1

print('a')
a
```

Можно передать несколько аргументов, и они будут выведены в одной строке:

```
print(1,'b')
1 b
```

Можете использовать необязательный аргумент для установки разделителя между элементами, когда указано несколько аргументов:

```
print(1,'b',sep='->')
1->b
```

Можно даже вывести саму функцию `print`:

```
print(print)
<built-in function print>
```

Выполнение базовых математических операций

Python можно использовать как калькулятор. Базовые математические операции встроены в его функционал. Вы можете заниматься математикой в интерактивной оболочке или использовать результаты вычислений в программе. Ниже приведены примеры сложения, вычитания, умножения, деления и возведения в степень в Python:

```
2 + 3
5

5 – 6
-1

3*4
12

9/3
3.0

2**3
8
```

Обратите внимание, что деление возвращает число с плавающей точкой, даже если используются целые числа. Чтобы ограничить результат деления целыми числами, можно использовать двойную косую черту[1]:

```
5//2
2
```

Еще один удобный оператор — «остаток от деления». Он возвращает остаток от деления. Для выполнения этой операции используйте знак процента:

```
5%2
1
```

С помощью остатка от деления удобно определять, является ли одно число делителем другого (в этом случае результат должен быть равен нулю). В примере ниже используется `is`, чтобы проверить, равен ли остаток от деления нулю:

```
14 % 7 is 0
True
```

Больше математических операций мы рассмотрим в части II.

---

[1] Оператор целочисленного деления, возвращающий только целое число без остатка от деления. — *Примеч. ред.*

Классы, объекты и точечная нотация

В главе 14 вы узнаете, как объявлять собственные классы и объекты. Сейчас вы можете представлять объект как объединение функций с данными. Большинство функций Python имеют атрибуты или методы. Для доступа к ним используется точечная нотация. Доступ к атрибуту можно получить, используя точку после имени объекта, а затем имя атрибута.

В следующем примере показано, как получить доступ к атрибуту числителя (numerator) целого числа:

```
a_number = 2
a_number.numerator
```

Получить доступ к методам объекта можно точно так же, но с последующими круглыми скобками. В примере ниже используется метод `to_bytes()` для того же целого числа, что конвертирует целое число в виде байтов:

```
a_number.to_bytes(8, 'little')
b'\x02\x00\x00\x00\x00\x00\x00\x00'
```

Резюме

Языки программирования помогают перевести человеческий язык в инструкции компьютеру. Python использует для этого разные типы операторов, причем каждый описывает определенное действие. Вы можете комбинировать операторы для создания ПО. Данные, над которыми выполняются действия, представлены в Python разными типами, включая как встроенные типы, так и определенные разработчиками и третьими сторонами. У этих типов есть свои характеристики, атрибуты и зачастую методы, доступ к которым можно получить с помощью точечной нотации.

Вопросы для закрепления

1. Что будет выведено командой `type(12)` в Python?
2. Как влияет использование в Python `assert(True)` на операторы, следующие за ним?
3. Как бы вы использовали Python для вызова исключения `LastParamError`?
4. Как бы вы использовали Python для вывода строки `Hello`?
5. Как с помощью Python возвести число 2 в степень 3?

3

Последовательности

Ошибок при использовании неадекватных данных
намного меньше, чем при отсутствии данных вообще.

Чарльз Бэббидж

В этой главе

- Общие операции с последовательностями.
- Списки и кортежи.
- Строки и строковые методы.
- Диапазоны.

В главе 2 вы узнали о типах коллекций. Здесь же мы познакомимся с группой встроенных типов, называемых *последовательностями*. Последовательность — это упорядоченная конечная коллекция. Вы можете представить ее как полку в библиотеке, где у каждой книги есть свое место, доступ к которой можно легко получить,

если вы знаете это место. Книги упорядочены, у каждой (кроме стоящих в конце) есть стоящие перед ней и после нее. Вы можете положить книгу на полку, взять ее оттуда, но эта полка может быть и пустой. Встроенные типы, составляющие последовательность, — это списки, кортежи, строки, бинарные строки и диапазоны. В этой главе рассмотрены их общие характеристики и особенности.

Общие операции

У семейства последовательностей есть довольно много общих функций. В частности, есть способы использования последовательностей, применимых для большинства членов группы. Есть операции, относящиеся к последовательностям конечной длины, для доступа к элементам в последовательности и создания новой последовательности на основе ее содержимого.

Проверка принадлежности

Проверить, принадлежит ли элемент последовательности, можно с помощью операции `in`. Она возвращает значение `True`, если последовательность содержит элемент, который оценивается как равный тому, что в запросе, и `False` — в противном случае. Ниже приведены примеры использования `in` с разными типами последовательностей:

```
'first' in ['first', 'second', 'third']
True

23 in (23,)
True

'b' in 'cat'
False

b'a' in b'ieojjza'
True
```

Можно использовать `not` в сочетании с `in`, чтобы проверить отсутствие чего-либо в последовательности:

```
'b' not in 'cat'
True
```

Два самых распространенных случая использования сочетания `in` и `not` — это интерактивная сессия для изучения данных и как часть оператора `if` (см. главу 5).

Индексирование

Поскольку последовательность — это упорядоченная серия элементов, вы можете получить доступ к элементу в ней, используя его местоположение, или *индекс*. Индексы начинаются с нуля и увеличиваются до числа, на единицу меньшего количества элементов. Например, в последовательности из восьми элементов у первого будет индекс ноль, а у последнего — семь.

Чтобы с помощью индекса получить доступ к элементу, заключите номер индекса в квадратные скобки. В следующем примере объявляется строка и осуществляется доступ к ее первой и последней подстрокам с помощью индексных номеров:

```
name = "Ignatius"
name[0]
'I'

name[7]
's'
```

С помощью отрицательных индексов вы можете индексировать обратный отсчет с конца последовательности:

```
name[-1]
's'

name[-2]
'u'
```

Слайсинг

Вы можете использовать индексы для создания новых последовательностей, которые будут представлять собой подпоследовательности оригинала. В квадратных скобках укажите начальный и конечный индексные номера последовательности, разделив их двоеточием, — так будет возвращена новая последовательность:

```
name = "Ignatius"
name[2:5]
'nat'
```

Возвращаемая подпоследовательность содержит элементы от первого индекса и до последнего, не включая его. Если вы пропустите начальный индекс, подпоследовательность начнется с родительской последовательности. А если последний, то подпоследовательность перейдет в конец последовательности:

```
name[:5]
'Ignat'
```

```
name[4:]
'tius'
```

Отрицательные индексные номера можно использовать для создания срезов (слайсов)[1], начинающих отсчет с конца последовательности. Ниже показано, как захватить последние три буквы строки:

```
name[-3:]
'ius'
```

Если вы хотите, чтобы срез пропускал элементы, предоставьте третий аргумент, указывающий на то, как вести подсчет. Так, если у вас есть список последовательности целых чисел, вы можете создать срез, просто используя начальный и конечный индексные номера:

```
scores = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
scores[3:15]
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Но вы можете указать и размер шага. Например, считать по три:

```
scores[3:15:3]
[3, 6, 9, 12]
```

Для обратного отсчета используйте отрицательный шаг:

```
scores[18:0:-4]
[18, 14, 10, 6, 2]
```

Сбор информации

Вы можете выполнять общие операции над последовательностями, чтобы собрать информацию о них. Последовательность конечна, и у нее есть длина, которую можно узнать с помощью функции `len`:

```
name = "Ignatius"
len(name)
8
```

Используйте `min` и `max`, чтобы найти минимальные и максимальные элементы:

[1] Срез (слайс, slice) — извлечение из данной строки одного символа или некоторого фрагмента подстроки или подпоследовательности. — *Примеч. пер.*

```
scores = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
min(scores)
0

max(name)
'u'
```

Эти методы предполагают, что содержимое последовательности можно сравнить способом, подразумевающим упорядочивание. В последовательностях, допускающих смешанные типы элементов, могут возникнуть ошибки, если содержимое нельзя будет сравнить:

```
max(['Free', 2, 'b'])
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-d8babe38f9d9> in <module>()
----> 1 max(['Free', 2, 'b'])
TypeError: '>' not supported between instances of 'int' and 'str'
```

Выяснить, сколько раз появляется элемент в последовательности, можно с помощью `count`:

```
name.count('a')
1
```

Получить индекс элемента в последовательности можно с помощью `index`:

```
name.index('s')
7
```

Можно использовать метод `index` для создания среза до элемента. Например, до буквы в строке:

```
name[:name.index('u')]
'Ignati'
```

Математические операции

Вы можете выполнять операции сложения и умножения с последовательностями одного типа. При этом эти операции проводятся именно с последовательностями, а не с их содержимым. Так, в результате сложения списка [1] и [2] получится [1,2], а не [3]. Ниже приведен пример использования оператора плюс (`+`) для создания новой строки из трех отдельных строк:

```
"prefix" + "-" + "postfix"
'prefix-postfix'
```

Оператор умножения (*) работает, выполняя многократное сложение всей последовательности, а не ее содержимого:

```
[0,2] * 4
[0, 2, 0, 2, 0, 2, 0, 2]
```

Это полезный способ настройки последовательности со значениями по умолчанию. Допустим, вы хотите отследить баллы для заданного количества участников в списке. Вы можете инициализировать список с помощью умножения так, что в нем будут исходные баллы для каждого участника:

```
num_participants = 10
scores = [0] * num_participants
scores
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Списки и кортежи

Списки и кортежи — последовательности, которые могут содержать объекты любого типа. Их содержимое может быть смешанным, поэтому у вас в одном списке могут быть строки, целые числа, экземпляры, числа с плавающей точкой и другие объекты. Разделители в списках и кортежах — запятые. Элементы в списке заключены в квадратные скобки, а в кортеже — в круглые. Основное различие между списками и кортежами в том, что первые изменяемы, а вторые — нет. Это означает, что вы можете изменить содержимое списка, но как только создан кортеж, изменить его уже невозможно. Чтобы изменить содержимое кортежа, нужно создать новый на основе текущего. Благодаря своей способности изменяться списки более функциональны, но они используют больше памяти.

Создание списков и кортежей

Вы создаете список с помощью функции — конструктора списков `list()` или синтаксиса квадратных скобок. Например, чтобы создать список исходных значений, просто укажите значения в квадратных скобках:

```
some_list = [1,2,3]
some_list
[1, 2, 3]
```

Кортежи создаются с помощью функции — конструктора кортежей `tuple()` или круглых скобок. Чтобы создать кортеж с одним элементом, поставьте за ним запятую, иначе Python воспримет круглые скобки не как обозначение кортежа, а как логическую группировку. Вы можете создать кортеж и без круглых скобок — просто поставьте запятую после элемента. Ознакомьтесь с примерами создания кортежа в листинге 3.1.

Листинг 3.1. Создание кортежа

```
tup = (1,2)
tup
(1,2)

tup = (1,)
tup
(1,)

tup = 1,2,
tup
(1,2)
```

ОБРАТИТЕ ВНИМАНИЕ

Распространенная, но незначительная ошибка случается, когда вы оставляете висящую запятую после аргумента функции. Это превращает аргумент в кортеж, содержащий исходный аргумент. Поэтому вторым аргументом функции my_function(1, 2,) будет (2,), а не 2.

Вы можете применять конструкторы списка и кортежа для последовательности в качестве аргумента. В следующем примере используется строка и создается список элементов в ней:

```
name = "Ignatius"
letters = list(name)
letters
['I', 'g', 'n', 'a', 't', 'i', 'u', 's']
```

Добавление и удаление элементов списка

Вы можете добавлять элементы в список и удалять их из него. Чтобы понять, как это работает, представьте, что список — это стопка книг. Самый эффективный способ добавить элемент в список — использовать метод `append`. Он добавляет элемент в конец списка, как если бы вы положили книги наверх стопки. Чтобы добавить элемент в другую позицию списка, используйте метод `insert`, указав в качестве аргумента номер индекса, в котором вы хотите разместить новый элемент. Это менее эффективно, чем применение `append`, так как может потребоваться переместить другие элементы в списке, чтобы

освободить место для нового. Но это является проблемой обычно только при работе с очень большими списками. В листинге 3.2 показаны примеры добавления и вставки.

Листинг 3.2. Добавление и вставка элементов списка

```
flavours = ['Chocolate', 'Vanilla']
flavours
['Chocolate', 'Vanilla']

flavours.append('SuperFudgeNutPretzelTwist')
flavours
['Chocolate', 'Vanilla', 'SuperFudgeNutPretzelTwist']

flavours.insert(0,"sourMash")
flavours
['sourMash', 'Chocolate', 'Vanilla', 'SuperFudgeNutPretzelTwist']
```

Для удаления элемента из списка используйте `pop`. При отсутствии аргументов этот метод удаляет последний элемент. Но, используя необязательный аргумент индекса, вы можете указать определенный. В обоих случаях элемент удаляется из списка и возвращается.

В следующем примере из списка извлекается последний элемент, а затем элемент с индексом 0. Вы можете видеть, что оба элемента возвращаются при извлечении, а затем исчезают из списка:

```
flavours.pop()
'SuperFudgeNutPretzelTwist'

flavours.pop(0)
'sourMash'

flavours
['Chocolate', 'Vanilla']
```

Чтобы добавить содержимое из одного списка в другой, используйте `extend`:

```
desserts = ['Cookies', 'Water Melon']
desserts
['Cookies', 'Water Melon']

desserts.extend(flavours)
desserts
['Cookies', 'Water Melon', 'Chocolate', 'Vanilla']
```

Этот метод изменяет первый список так, что теперь к его содержимому добавляется содержимое второго.

ИНИЦИАЛИЗАЦИЯ ВЛОЖЕННЫХ СПИСКОВ

Есть хитрая ошибка, с которой сталкиваются начинающие разработчики Python. Она включает в себя сочетание изменяемости списка с характером умножения последовательностей. Чтобы инициализировать список с четырьмя подсписками, вы, возможно, попробуете умножить один так:

```
lists = [[]] * 4
lists
[[], [], [], []]
```

Все идет хорошо, пока вы не меняете один из подсписков:

```
lists[-1].append(4)
lists
[[4], [4], [4], [4]]
```

Все ваши подсписки оказались изменены! Это связано с тем, что умножение инициализирует только один список и ссылается на него четыре раза. Ссылки выглядят независимыми, пока вы не попробуете изменить одну из них. Решение — использовать списковые включения (см. главу 13).

```
lists = [[] for _ in range(4)]
lists[-1].append(4)
lists
 [[], [], [], [4]]
```

Распаковка

Вы приписываете значения множественным переменным из списка или кортежа в одной строке:

```
a, b, c = (1,3,4)
a
1

b
3

c
4
```

Или, если вы хотите присвоить множественные значения одной переменной, одновременно присваивая единственное значение другим, используйте символ * перед

переменной, которая будет принимать множество значений. Затем она присоединит все элементы, не присвоенные другим переменным:

```
*first, middle, last = ['horse', 'carrot', 'swan', 'burrito', 'fly']
first
['horse', 'carrot', 'swan']

last
'fly'

middle
'burrito'
```

Сортировка списков

Для списков можно использовать встроенные методы `sort` и `reverse`, которые могут изменить порядок содержимого. Подобно `min` и `max`, эти методы работают, только если содержимое сопоставимо, как в следующих примерах:

```
name = "Ignatius"
letters = list(name)
letters
['I', 'g', 'n', 'a', 't', 'i', 'u', 's']

letters.sort()
letters
['I', 'a', 'g', 'i', 'n', 's', 't', 'u']

letters.reverse()
letters
['u', 't', 's', 'n', 'i', 'g', 'a', 'I']
```

Строки

Строка — это последовательность символов. В Python строки по умолчанию используют Юникод, и любой его символ может быть частью строки. Строки представлены как символы в кавычках. Работают как одинарные, так и двойные кавычки, и созданные с их помощью строки одинаковы:

```
'Here is a string'
'Here is a string'

"Here is a string" == 'Here is a string'
True
```

Чтобы заключить в кавычки слово или слова внутри строки, используйте один тип кавычек, одиночные или двойные, чтобы заключить в них это слово или слова, а другой — для заключения целой строки. В следующем примере слово *is* заключено в двойные кавычки, а целая строка — в одиночные:

```
'Here "is" a string'
'Here "is" a string'
```

Длинные строки вы заключаете в три набора двойных кавычек:

```
a_very_large_phrase = """
Wikipedia is hosted by the Wikimedia Foundation,
a non-profit organization that also hosts a range of other projects.
"""
```

В строках Python вы можете использовать специальные символы, каждому из которых предшествует обратный слеш. Специальные символы включают `\t` для табуляции, `\r` для возврата каретки[1] и `\n` для новой строки. Во время вывода они интерпретируются со специальным значением. Хотя эти символы, как правило, полезны, они могут быть неудобны, если вы представляете путь Windows:

```
windows_path = "c:\row\the\boat\now"
print(windows_path)

ow heoat
      ow
```

В таких случаях вы можете использовать необработанный строковый тип Python, интерпретирующий все символы буквально. Это можно сделать, добавляя к строке префикс `r`:

```
windows_path = r"c:\row\the\boat\now"
print(windows_path)
c:\row\the\boat\now
```

Есть ряд вспомогательных функций для строк, позволяющих работать с разными заглавными буквами (см. листинг 3.3).

Листинг 3.3. Вспомогательные функции строки

```
captain = "Patrick Tayluer"
captain
'Patrick Tayluer'
```

[1] Некорректно работает в Python IDLE. — *Примеч. ред.*

```
captain.capitalize()
'Patrick tayluer'

captain.lower()
'patrick tayluer'

captain.upper()
'PATRICK TAYLUER'

captain.swapcase()
'pATRICK tAYLUER'

captain = 'patrick tayluer'
captain.title()
'Patrick Tayluer'
```

В Python 3.6 представлены строки формата, или f-строки. Вы можете вставить значения в них во время выполнения с помощью полей замены, разделенных фигурными скобками. В поле замены можно вставить любое выражение, включая переменные. У f-строки есть префикс `F` или `f`:

```
strings_count = 5
frets_count = 24
f"Noam Pikelny's banjo has {strings_count} strings and {frets_count} frets"
'Noam Pikelny's banjo has 5 strings and 24 frets'
```

Следующий пример показывает, как вставить математическое выражение в поле замены:

```
a = 12
b = 32
f"{a} times {b} equals {a*b}"
'12 times 32 equals 384'
```

Ниже приведен пример того, как вставить элементы из списка в поле замены:

```
players = ["Tony Trischka", "Bill Evans", "Alan Munde"]
f"Performances will be held by {players[1]}, {players[0]}, and {players[2]}"
'Performances will be held by Bill Evans, Tony Trischka, and Alan Munde'
```

Диапазоны

Использование объектов диапазона — эффективный способ представления ряда чисел, упорядоченных по значению. Он широко применяется для указания того, сколько раз

должен выполняться цикл (см. главу 5). Объекты диапазона могут принимать следующие аргументы: начало (необязателен), конец (обязателен) и шаг (необязателен).

Как и в случае со слайсингом, начало включено в диапазон, а конец — нет. И точно так же можно использовать отрицательные шаги для обратного отсчета. Диапазоны производят вычисления по запросу, поэтому им не нужно выделять больше памяти. Листинг 3.4 демонстрирует, как создавать диапазоны с необязательными аргументами и без них. В этом листинге создаются списки из диапазонов так, чтобы вы могли увидеть все содержимое, которое будет предоставлено диапазоном.

Листинг 3.4. Создание диапазонов

```
range(10)
range(0, 10)

list(range(1, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]

list(range(0,10,2))
[0, 2, 4, 6, 8]

list(range(10, 0, -2))
[10, 8, 6, 4, 2]
```

Резюме

В этой главе рассмотрена группа типов импорта, известная как последовательности. Последовательность — это упорядоченный конечный набор элементов.

Содержимое списков и коретежей может быть смешанным с точки зрения типов. Первые можно изменить после создания, а вторые — нет.

Строки — это последовательности текста. Объекты диапазона используются для описания диапазона чисел. Списки, строки и диапазоны являются наиболее распространенными типами в Python.

Вопросы для закрепления

1. Как вы проверите наличие `a` в списке `my_list`?
2. Как выяснить, сколько раз `b` появляется в строке `my_string`?
3. Как вы добавите `a` в конец списка `my_list`?
4. Равны ли строки `'superior'` и `"superior"`?
5. Как бы вы создали диапазон от 3 до 13?

4

Прочие структуры данных

Статистическое мышление однажды станет
столь же необходимым для истинного гражданина,
как умение читать и писать.

Сэмюэл Стенли Уилкс

В этой главе

- Создание словарей.
- Обновление словарей и получение доступа к их содержимому.
- Создание множеств.
- Операции над множествами.

Помимо порядкового представления данных, есть и другие. Словари и множества — это структуры данных, которые не зависят от их порядка, — мощные модели, встроенные в инструментарий Python.

Словари

Представьте, что вы проводите исследование, чтобы определить, существует ли зависимость между ростом студента и его средним баллом (GPA, grade point average, средний балл оценки). Вам нужна структура для представления данных отдельного студента (его имя, рост и средний балл). Можно сохранить информацию в списке или кортеже. Но вам придется отслеживать, какой фрагмент данных представляет тот или иной индекс.

Лучше всего пометить данные так, чтобы вам не пришлось отслеживать перевод из индекса в атрибут. Вы можете использовать словари для хранения данных в виде пар ключ/значение. Доступ к каждому элементу или значению в словаре осуществляется с помощью ключа. Такой поиск очень эффективен и выполняется намного быстрее, чем поиск по длинной последовательности.

При использовании пары ключ/значение ключ и значения разделяются двоеточием. Можно передавать множество пар ключ/значение, разделенных запятыми и заключенных в фигурные скобки.

Итак, словарь для записи данных студента может выглядеть так:

```
{ 'name': 'Betty', 'height': 62,'gpa': 3.6 }
```

Ключами для этого словаря служат строки `'name'`, `'height'` и `'gpa'`. Каждый указывает на фрагмент данных: `'name'` указывает на строку `'Betty'`, `'height'` — на целое число `62`, а `'gpa'` — на число с плавающей точкой `3.6`. Значения могут быть любого типа, но есть некоторые ограничения для типа ключа. Об этом мы с вами поговорим чуть позже.

Создание словарей

Создавать словари можно с исходными данными или без них. Вы можете создать пустой словарь с помощью метода-конструктора `dict()` или просто используя фигурные скобки:

```
dictionary = dict()
dictionary
{}

dictionary = {}
dictionary
{}
```

В первом примере создается пустой словарь с помощью метода-конструктора `dict()`. Он приписывается переменной с именем `dictionary`. Во втором примере создается пустой словарь с помощью фигурных скобок и приписывается той же переменной. В обоих случаях создан пустой словарь, представленный пустыми фигурными скобками.

Вы можете создавать словари, сразу инициализированные данными. Один из вариантов — передать ключи и значения в качестве именованных параметров:

```
subject_1 = dict(name='Paula', height=64, gpa=3.8, ranking=1)
```

Альтернативный вариант — передать пары ключ/значение конструктору в виде списка/кортежа списков/кортежей (возможен любой вариант из четырех. — *Примеч. ред.*), где каждый подсписок/подкортеж представляет пару ключ/значение:

```
subject_2 = dict([['name','Paula'],['height',64],['gpa',3.8]],['ranking',1])
```

Третий вариант — создать словарь с помощью фигурных скобок, при этом ключи и значения соединяются попарно двоеточием и разделяются запятыми:

```
subject_3 = {'name':'Paula', 'height':64, 'gpa':3.8, 'ranking':1}
```

Все эти три метода создают словари, которые расцениваются одинаково, пока используются одни и те же ключи и значения:

```
subject_1 == subject_2 == subject_3
True
```

Доступ, добавление и обновление с помощью ключей

Ключи словаря обеспечивают средства для доступа к данным и их изменения. Обычно вы получаете доступ к данным с помощью соответствующего ключа в квадратных скобках так же, как получаете доступ к индексам в последовательности:

```
student_record = {'name':'Paula', 'height':64, 'gpa':3.8}
student_record['name']
'Paula'

student_record['height']
64

student_record['gpa']
3.8
```

Чтобы добавить пару ключ/значение в существующий словарь, можно присвоить значение слоту с помощью того же синтаксиса:

```
student_record['applied'] = '2019-10-31'
student_record
{'name':'Paula',
 'height':64,
 'gpa':3.8,
 'applied': '2019-10-31'}
```

Новая пара ключ/значение теперь содержится в исходном словаре.

Чтобы обновить значение для существующего ключа, можно точно так же использовать синтаксис квадратных скобок:

```
student_record['gpa'] = 3.0
student_record['gpa']
3.0
```

Удобный способ увеличения числовых данных — использование оператора `+=`, который является сокращением для обновления значения путем добавления к нему:

```
student_record['gpa'] += 1.0
student_record['gpa']
4.0
```

Удаление элементов из словарей

Иногда возникает необходимость в удалении данных, например когда в словаре есть личная информация. Допустим, ваши данные включают студенческий ID, но он не относится к конкретному исследованию. Чтобы сохранить конфиденциальность студента, вы можете обновить значение для ID на `None`:

```
student_record = {'advisor': 'Pickerson',
                  'first': 'Julia',
                  'gpa': 4.0,
                  'last': 'Brown',
                  'major': 'Data Science',
                  'minor': 'Math'}
student_record['id'] = None
student_record
{'advisor': 'Pickerson',
 'first': 'Julia',
 'gpa': 4.0,
```

```
 'id': None,
 'last': 'Brown',
 'major': 'Data Science',
 'minor': 'Math'}
```

Это предотвратит использование ID посторонними.

Другой способ — удаление пары ключ/значение целиком с помощью функции `del()`. Она принимает словарь с ключом в квадратных скобках в качестве аргумента и удаляет подходящую пару ключ/значение:

```
del(student_record['id'])
student_record
{'advisor': 'Pickerson',
 'first': 'Julia',
 'gpa': 4.0,
 'last': 'Brown',
 'major': 'Data Science',
 'minor': 'Math'}
```

ПРИМЕЧАНИЕ

Конечно, чтобы действительно защитить личность человека, вы захотите удалить его имя и любую другую персональную информацию.

Представления словаря

Представления словаря — это объекты, предлагающие информацию о словаре. Их всего три: `dict_keys`, `dict_values` и `dict_items`. Каждый тип представления позволяет взглянуть на словарь с определенного ракурса.

У словарей есть метод `keys()`, возвращающий объект `dict_keys`. Этот объект дает вам доступ к текущим ключам словаря:

```
keys = subject_1.keys()
keys
dict_keys(['name', 'height', 'gpa', 'ranking'])
```

Метод `values()` возвращает объект `dict_values`, который дает вам доступ к значениям в словаре:

```
values = subject_1.values()
values
dict_values(['Paula', 64, 4.0, 1])
```

Метод `items()` возвращает объект `dict_items`, представляющий пары ключ/значение в словаре:

```
items = subject_1.items()
items
dict_items([('name', 'Paula'), ('height', 64), ('gpa', 4.0), ('ranking', 1)])
```

Проверить принадлежность любого из этих представлений можно с помощью оператора `in`. Следующий пример показывает, как проверить, используется ли в словаре ключ `'ranking'`:

```
'ranking' in keys
True
```

Пример ниже показывает, как проверить, является ли целое число 1 одним из значений словаря:

```
1 in values
True
```

Проверить, равно ли сопоставление пары ключ/значение с `'ranking'` числу 1, можно так:

```
('ranking',1) in items
True
```

Начиная с версии Python 3.8, представления словарей стали динамическими. Это значит, что представление, после получения которого были внесены изменения в словарь, отразит эти изменения. Представьте, что хотите удалить пару ключ/значение из словаря, к чьим представлениям получен доступ выше:

```
del(subject_1['ranking'])
subject_1
{'name': 'Paula', 'height': 64, 'gpa': 4.0}
```

Пара ключ/значение удаляется и из объектов представления:

```
'ranking' in keys
False

1 in values
False

('ranking',1) in items
False
```

У каждого типа представления словаря есть длина, доступ к которой можно получить с помощью той же функции `len`, используемой с последовательностями:

```
len(keys)
3

len(values)
3

len(items)
3
```

Начиная с версии Python 3.8, вы можете использовать функцию `reversed` для представления `dict_key`, чтобы получить его в обратном порядке:

```
keys
dict_keys(['name', 'height', 'gpa'])

list(reversed(keys))
['gpa', 'height', 'name']
```

Представления `dict_key` подобны множествам. Это значит, что многие операции над множествами будут работать и с ними. Следующий пример показывает, как создать два словаря:

```
admission_record = {'first':'Julia',
                    'last':'Brown',
                    'id': 'ax012E4',
                    'admitted': '2020-03-14'}
student_record = {'first':'Julia',
                  'last':'Brown',
                  'id': 'ax012E4',
                  'gpa':3.8,
                  'major':'Data Science',
                  'minor': 'Math',
                  'advisor':'Pickerson'}
```

Вы можете проверить равенство ключей:

```
admission_record.keys() == student_record.keys()
False
```

Или поискать симметрическую разность:

```
admission_record.keys() ^ student_record.keys()
{'admitted', 'advisor', 'gpa', 'major', 'minor'}
```

Так можно найти пересечение:

```
admission_record.keys() & student_record.keys()
{'first', 'id', 'last'}
```

А так — разность:

```
admission_record.keys() - student_record.keys()
{'admitted'}
```

Таким способом можно вернуть объединение представлений:

```
admission_record.keys() | student_record.keys()
{'admitted', 'advisor', 'first', 'gpa', 'id', 'last', 'major', 'minor'}
```

ПРИМЕЧАНИЕ

Подробнее о множествах и операциях над ними вы узнаете в следующем разделе.

Наиболее распространенное использование представлений `key_item` — итерация словаря и выполнение операций с каждой парой ключ/значение. В следующем примере используется цикл `for` (см. главу 5) для вывода каждой пары:

```
for k,v in student_record.items():
    print(f"{k} => {v}")
first => Julia
last => Brown
gpa => 4.0
major => Data Science
minor => Math
advisor => Pickerson
```

При необходимости можно выполнить похожие циклы с `dict_keys` или `dict_values`.

Проверка наличия ключа в словаре

Чтобы проверить, используется ли ключ в словаре, можно использовать `dict_key` и оператор `in`:

```
'last' in student_record.keys()
True
```

Для сокращения кода можно проверить ключ без явного вызова представления `dict_key`. Вместо этого просто используйте `in` прямо со словарем:

```
'last' in student_record
True
```

Это работает и при итерации по ключам словаря. Вам не нужен прямой доступ к представлению `dict_key`:

```
for key in student_record:
    print(f"key: {key}")
key: first
key: last
key: gpa
key: major
key: minor
key: advisor
```

Метод get

Попытка получить доступ к ключу, которого нет в словаре, с помощью синтаксиса квадратных скобок приведет к ошибке:

```
student_record['name']
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-18-962c04650d3e> in <module>()
----> 1 student_record['name']
   KeyError: 'name'
```

Такой тип ошибки прерывает выполнение программы, запущенной за пределами блокнота. Избежать ее можно, проверив, есть ли ключ в словаре, до совершения запроса на доступ к нему:

```
if 'name' in student_record:
    student_record['name']
```

В этом примере используется оператор `if`, получающий доступ к ключу `'name'`, только если он есть в словаре (подробнее об операторе `if` вы узнаете в главе 5).

Для удобства в словарях есть метод `get()`. Он создан для безопасного доступа к отсутствующим ключам. Если по умолчанию ключа нет, этот метод возвращает константу `None`:

```
print( student_record.get('name') )
None
```

Вы можете предоставить второй аргумент, который будет возвращаемым значением в случае отсутствия ключа:

```
student_record.get('name', 'no-name')
'no-name'
```

Можно также объединить в цепочку множество операторов `get`:

```
student_record.get('name', admission_record.get('first', 'no-name'))
'Julia'
```

Здесь совершается попытка получить значение для ключа `'name'` из словаря `student_record`, а при его отсутствии — значение для ключа `'first'` из словаря `admission_record`. Если и этого ключа нет, возвращается значение по умолчанию `'no-name'`.

Допустимые типы ключей

Значение некоторых объектов можно изменить, но значения других могут быть статичны. Объекты, чьи значения можно изменить, называются *изменяемыми*. Как вы уже видели, списки тоже входят в их число. Значения же неизменяемых объектов изменить нельзя. К таковым относятся целые числа, строки, объекты диапазона, двоичные строки и кортежи. Неизменяемые объекты, кроме определенных кортежей, можно использовать в качестве ключей в словарях:

```
{ 1         : 'an integer',
  'string'  : 'a string',
  ('item',) : 'a tuple',
  range(12) : 'a range',
  b'binary' : 'a binary string' }
```

Изменяемые объекты, такие как списки, напротив, нельзя использовать как ключи для словарей. Если вы попробуете это сделать, то столкнетесь с ошибкой:

```
{('item',): 'a tuple',
1: 'an integer',
b'binary': 'a binary string',
range(0, 12): 'a range',
'string': 'a string',
['a', 'list'] : 'a list key' }
   ---------------------------------------------------------------------------
  TypeError                                 Traceback (most recent call last)
  <ipython-input-31-1b0e555de2b5> in <module>()
  ----> 1 { ['a', 'list'] : 'a list key' }
  TypeError: unhashable type: 'list'
```

Кортеж с неизменяемым содержимым можно использовать как ключ словаря. Поэтому кортежи чисел, строк и другие — допустимые ключи:

```
tuple_key = (1, 'one', 1.0, ('uno',))
{ tuple_key: 'some value' }
{(1, 'one', 1.0, ('uno',)): 'some value'}
```

Если в кортеже есть изменяемый объект, например список, то в качестве ключа его использовать нельзя:

```
bad_tuple = ([1, 2], 3)
{ bad_tuple: 'some value' }
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-b2cddfdda91e> in <module>()
      1 bad_tuple = ([1, 2], 3)
----> 2 { bad_tuple: 'some value' }
TypeError: unhashable type: 'list'
```

Метод hash

Словарь можно представить как хранилище значений в индексированной спископодобной структуре с методом, быстро и надежно сопоставляющим ключевые объекты с индексными номерами. Этот метод известен как хэш-функция. Его можно найти в таких неизменяемых объектах Python, как метод `_hash_()`. Он создан для скрытого использования, но может быть вызван и напрямую:

```
a_string = 'a string'
a_string.__hash__()
4815474858255585337

a_tuple = 'a','b',
a_tuple.__hash__()
7273358294597481374

a_number = 13
a_number.__hash__()
13
```

Функция `hash` использует значение объекта для получения согласованного вывода. Поэтому для изменяемого объекта невозможно создать согласованное хэширование. Вы не сможете получить хэш такого изменяемого объекта, как список:

```
a_list = ['a','b']
a_list.__hash__()
```

```
---------------------------------------------------------------------------
TypeError Traceback (most recent call last) <ipython-input-40-c4f99d4ea902> in
<module>()
      1 a_list = ['a','b']
----> 2 a_list.__hash__()
TypeError: 'NoneType' object is not callable
```

Словари и списки — наиболее часто используемые структуры данных в Python. Они предоставляют способы структурирования данных для содержательного и быстрого их просмотра.

ПРИМЕЧАНИЕ

Хотя механизм просмотра пар ключ/значение не зависит от порядка данных, начиная с Python 3.7, порядок ключей отражает порядок их вставки.

Множества

Структура данных «множества» в Python — это реализация множеств, с которыми вы, возможно, знакомы с курса математики. *Множество* — это неупорядоченная совокупность уникальных элементов. Это что-то вроде волшебного чемодана, не допускающего хранения в себе одинаковых объектов. Элементы во множествах могут быть любого хэшируемого типа.

Множество представлено в Python в виде списка с элементами, разделенными запятыми и заключенными в фигурные скобки:

```
{ 1, 'a', 4.0 }
```

Создать множество можно либо с помощью функции-конструктора `set()`, либо напрямую с применением фигурных скобок. Но, используя пустые фигурные скобки, вы создаете пустой словарь, а не пустое множество. Чтобы создать пустое множество, нужно использовать `set()`:

```
empty_set = set()
empty_set
set()

empty_set = {}
empty_set
{}
```

Создать множество с исходными значениями можно с помощью конструктора или фигурных скобок.

Вы можете предоставить любой тип последовательности в качестве аргумента, а множество будет возвращено на основе уникальных элементов этой последовательности:

```
letters = 'a', 'a', 'a', 'b', 'c'
unique_letters = set(letters)
unique_letters
{'a', 'b', 'c'}

unique_chars = set('mississippi')
unique_chars
{'i', 'm', 'p', 's'}

unique_num = {1, 1, 2, 3, 4, 5, 5}
unique_num
{1, 2, 3, 4, 5}
```

Содержимое множества должно быть хэшируемым и неизменяемым. Потому что, как и ключи словаря, множества хэшируют свое содержимое, чтобы определить его уникальность. Поэтому список не может быть элементом множества:

```
bad_set = { ['a','b'], 'c' }
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
    <ipython-input-12-1179bc4af8b8> in <module>()
----> 1 bad_set = { ['a','b'], 'c' }
TypeError: unhashable type: 'list'
```

Добавить элементы в множество можно с помощью метода `add()`:

```
unique_num.add(6)
unique_num
{1, 2, 3, 4, 5, 6}
```

Используйте оператор `in` для проверки принадлежности к множеству:

```
3 in unique_num
True

3 not in unique_num
False
```

Используйте функцию `len()`, чтобы увидеть количество элементов в множестве:

```
len(unique_num)
6
```

Как и со списками, вы можете удалять элемент из множества и возвращать его с помощью метода `pop()`:

```
unique_num.pop()
unique_num
{2, 3, 4, 5, 6}
```

Но, в отличие от списков, в множествах вы не можете полагаться на метод `pop()` при удалении элементов в определенном порядке. Чтобы удалить определенный элемент, используйте метод `remove()`:

```
students = {'Karl', 'Max', 'Tik'}
students.remove('Karl')
students
{'Max', 'Tik'}
```

Этот метод не возвращает удаленный элемент. Попытавшись удалить элемент, которого нет во множестве, вы получите ошибку:

```
students.remove('Barb')
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
    <ipython-input-3-a36a5744ac05> in <module>()
----> 1 students.remove('Barb')
KeyError: 'Barb'
```

Можно написать код, чтобы проверить наличие элемента во множестве, прежде чем удалять его. Но есть удобная функция `discard()`, которая не выдает ошибку при попытке удалить отсутствующий элемент:

```
students.discard('Barb')
students.discard('Tik')
students
{'Max'}
```

Удалить все содержимое множества можно с помощью `clear()`:

```
students.clear()
students
set()
```

Помните, что множества неупорядочены, поэтому не поддерживают индексацию:

```
unique_num[3]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-fecab0cd5f95> in <module>()
----> 1 unique_num[3]
TypeError: 'set' object does not support indexing
```

Проверить равенство можно с помощью операторов равенства (==) и неравенства (!=), рассмотренных в главе 5. Множества неупорядочены, поэтому те, что созданы из последовательностей с одинаковыми элементами, расположенными в разном порядке, равны:

```
first = {'a','b','c','d'}
second = {'d','c','b','a'}
first == second
True

first != second
False
```

Операции над множествами

Над множествами можно выполнить ряд операций. Многие из них предлагаются и как методы для объектов множеств, и как отдельные операторы (<, <=, >, >=, &, | и ^). Методы множества можно использовать для выполнения операций между множеством и другими множествами или другими итерируемыми объектами (типами данных, которые можно итерировать). Операторы множества работают только между множеством и другими множествами (или замороженными множествами).

Непересекающиеся множества

Два множества являются непересекающимися, если у них нет общих элементов. Чтобы это проверить, с множествами Python можно использовать метод `disjoint()`. При сравнении множества четных и нечетных чисел общих у них не найдется, поэтому результат `disjoint()` будет равен `True`:

```
even = set(range(0,10,2))
even
{0, 2, 4, 6, 8}

odd = set(range(1,11,2))
odd
{1, 3, 5, 7, 9}

even.isdisjoint(odd)
True
```

Подмножество

Если все элементы множества B можно найти в множестве A, то B — это подмножество A. Метод `subset()` проверяет, является ли текущее множество подмножеством другого. В следующем примере проверяется, является ли множество положительных чисел, кратных 3 и меньших 21, подмножеством положительных целых чисел, которые меньше 21:

```
nums = set(range(21))
nums
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}

threes = set(range(3,21,3))
threes
{3, 6, 9, 12, 15, 18}

threes.issubset(nums)
       True
```

Оператор `<=` позволит проверить, является ли множество слева подмножеством множества справа:

```
threes <= nums
True
```

Как уже упоминалось, версия-метод этого оператора работает с аргументами, не являющимися множествами. В следующем примере проверяется, принадлежит ли множество чисел, кратных 3, диапазону от 0 до 20:

```
threes.issubset(range(21))
True
```

Оператор не работает с объектами, не являющимися множеством:

```
threes <= range(21)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
    <ipython-input-30-dbd51effe302> in <module>()
    ----> 1 threes <= range(21)
TypeError: '<=' not supported between instances of 'set' and 'range'
```

Строгие подмножества

Если все элементы первого множества есть во втором, но не все элементы второго есть в первом, то первое множество — строгое подмножество второго. Иначе говоря, первое множество — это подмножество второго, но они не равны. Для проверки на строгое подмножество используйте оператор <:

```
threes < nums
True

threes < {'3','6','9','12','15','18'}
False
```

Надмножества и строгие надмножества

Надмножество обратно подмножеству: если в первом множестве есть все элементы второго, оно будет его надмножеством. И если одно множество служит надмножеством другого и они не равны, то оно будет строгим надмножеством. В Python есть метод `issuperset()`, принимающий другое множество или любой другой итерируемый объект в качестве аргумента:

```
nums.issuperset(threes)
True

nums.issuperset([1,2,3,4])
True
```

Вы используете оператор => для проверки на надмножество и оператор > для проверки на строгое надмножество:

```
nums >= threes
True

nums > threes
True

nums >= nums
True

nums > nums
False
```

Объединение

Результат объединения двух множеств — множество, содержащее все их элементы. Для множеств Python можно использовать метод `union()`. Он работает с множествами и другими итерируемыми объектами. Можно также использовать автономный оператор |, возвращающий объединение двух множеств:

```
odds = set(range(0,12,2))
odds
{0, 2, 4, 6, 8, 10}

evens = set(range(1,13,2))
evens
{1, 3, 5, 7, 9, 11}

odds.union(evens)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

odds.union(range(0,12))
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

odds | evens
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

Пересечение

Пересечение двух множеств — множество со всеми общими для обоих множеств элементами. Для выполнения пересечений используйте метод `intersection()` или оператор &:

```
under_ten = set(range(10))
odds = set(range(1,21,2))
under_ten.intersection(odds)
{1, 3, 5, 7, 9}

under_ten & odds
{1, 3, 5, 7, 9}
```

Разность

Разность между двумя множествами — это все элементы первого множества, которых нет во втором. Для вычисления разницы множеств используйте метод `difference()` или оператор –:

```
odds.difference(under_ten)
{11, 13, 15, 17, 19}
```

```
odds - under_ten
{11, 13, 15, 17, 19}
```

Симметрическая разность

Симметрическая разность двух множеств — это множество с элементами, находящимися либо в одном, либо в другом множестве, но не в их пересечении. Для вычисления симметрической разности во множествах Python есть метод `symmetric_difference()` и оператор ^:

```
under_ten = set(range(10))
over_five = set(range(5, 15))
under_ten.symmetric_difference(over_five)
{0, 1, 2, 3, 4, 10, 11, 12, 13, 14}

under_ten ^ over_five
{0, 1, 2, 3, 4, 10, 11, 12, 13, 14}
```

Обновление множеств

В Python есть несколько готовых способов обновления содержимого множества. В дополнение к методу `update()`, добавляющему содержимое во множество, можно использовать варианты, которые обновляются на основе разных операций с множествами.

Следующий пример показывает, как выполнить обновление из другого множества:

```
unique_num = {0, 1, 2}
unique_num.update( {3, 4, 5, 7} )
unique_num
{0, 1, 2, 3, 4, 5, 7}
```

Пример ниже показывает, как выполнить обновление из списка:

```
unique_num.update( [8, 9, 10] )
unique_num
{0, 1, 2, 3, 4, 5, 7, 8, 9, 10}
```

Этот пример демонстрирует, как обновить разницу из диапазона:

```
unique_num.difference_update( range(0,12,2) )
unique_num
{1, 3, 5, 7, 9}
```

В следующем примере показано, как обновить пересечение:

```
unique_num.intersection_update( { 2, 3, 4, 5 } )
unique num
{3, 5}
```

Ниже показано, как обновить симметрическую разность:

```
unique_num.symmetric_difference_update( {5, 6, 7 } )
unique_num
{3, 6, 7}
```

Из этого примера видно, как обновить оператор объединения:

```
unique_letters = set("mississippi")
unique_letters
{'i', 'm', 'p', 's'}

unique_letters |= set("Arkansas")
unique_letters
{'A', 'a', 'i', 'k', 'm', 'n', 'p', 'r', 's'}
```

Следующий пример демонстрирует, как обновить оператор разницы:

```
unique_letters -= set('Arkansas')
unique_letters
{'i', 'm', 'p'}
```

В этом примере показано, как обновить оператор пересечения:

```
unique_letters
{'m', 'p'}

unique_letters ^= set('mud') 2 unique_letters
{'d', 'p', 'u'}
```

Замороженные множества

Множества изменяемы, поэтому их нельзя использовать как ключи словаря или элементы в других множествах. В Python замороженные множества — это подобные множествам неизменяемые объекты. Их можно применять вместо множеств для любых операций, содержимое которых не нужно изменять:

```
froze = frozenset(range(10))
froze
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})

froze < set(range(21))
True

froze & set(range(5, 15))
frozenset({5, 6, 7, 8, 9})

froze ^ set(range(5, 15))
frozenset({0, 1, 2, 3, 4, 10, 11, 12, 13, 14})

froze | set(range(5,15))
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14})
```

Резюме

Встроенные структуры данных Python предлагают много способов представления и организации ваших данных. Словари и множества — дополнение к типам последовательностей. Первые эффективно сопоставляют ключи со значениями. Вторые реализуют математические операции над множествами как структуры данных. И словари, и множества — хороший выбор, когда порядок не важен.

Вопросы для закрепления

1. Каковы три способа создания словаря со следующими парами ключ/значение: `{'name': 'Smuah', 'height':62}`?
2. Как вы обновите значение, связанное с ключом `gpa` в словаре `student`, до `'4.0'`?
3. Если в словаре заданы определенные данные, как безопасно получить доступ к значению для ключа `settings` при его возможном отсутствии?
4. В чем разница между изменяемым и неизменяемым объектом?
5. Как создать множество из строки `"lost and lost again"`?

5

Управление выполнением

Приблизительный ответ на правильный вопрос
стоит намного больше, чем точный ответ
на неправильный.

Джон Тьюки

В этой главе

- Знакомство с составными операторами.
- Операции проверки на равенство.
- Операции сравнения.
- Булевы операции.
- Операторы `if`.
- Циклы `while`.
- Циклы `for`.

Вплоть до этой главы вы рассматривали операторы как отдельные единицы, выполняющие последовательно одну строку за раз. Программирование становится намного интереснее и полезнее, когда вы можете группировать операторы, чтобы они выполнялись как единый модуль. Простые операторы, соединенные вместе, могут выполнять более сложные действия.

Составные операторы

Глава 2 знакомит вас с простыми операторами, выполняющими отдельные действия. Здесь же рассматриваются составные, позволяющие управлять выполнением группы операторов. Такое выполнение может произойти, только если условие истинно. Составные операторы, рассматриваемые в этой главе, включают циклы `for`, `while` и операторы `if`, `try` и `with`.

Структура составных операторов

Составные операторы состоят из управляющего (-их) оператора (-ов) и группы управляемых операторов. Они начинаются с ключа, указывающего на тип оператора, далее следует специфичное для этого типа выражение и двоеточие:

```
<ключ> <выражение>:
```

Управляемые операторы можно объединить двумя способами. Первый, более распространенный способ — объединение их в виде *блока кода* — группы операторов, выполняемых вместе. В Python блоки кода объявляются с помощью отступа. Группа операторов с одинаковым отступом объединяется в один блок кода. Группа заканчивается, когда есть оператор без отступа, как у остальных. Конечный оператор не входит в блок кода и будет выполняться независимо от управляющего оператора.

Вот как выглядит блок кода:

```
<управляющий оператор>:
    <управляемый оператор 1>
    <управляемый оператор 2>
    <управляемый оператор 3>
<конечный оператор>
```

Использование отступа для объявления блоков кода — отличительная черта Python. Большинство других популярных языков программирования используют другие алгоритмы для группировки кода, например фигурные скобки.

Еще один способ группировки управляемых операторов — выводить их после управляющего оператора и отделять с помощью точки с запятой:

```
<управляющий оператор>:<управляемый оператор 1>; <управляемый оператор 2>;
```

Второй способ лучше использовать, только если у вас очень мало управляемых операторов и вы чувствуете, что ограничение составного оператора одной строки не ухудшит читаемость программы.

Оценка True или False

Оператор `if`, циклы `while` и `for` — это составные операторы, зависящие от управляющего выражения, которое должно оцениваться как `True` или `False`. К счастью, Python почти все оценивает как равное одному из этих значений. Четыре наиболее часто используемых встроенных выражения, которые применяются для управления составными операторами, — это операции проверки на равенство, сравнения, булевы операторы и оценка объектов.

Операции проверки на равенство

В Python есть операторы равенства `==`, неравенства `!=` и тождественный оператор `is`. Операторы равенства и неравенства сравнивают значение двух объектов и возвращают одну из констант — `True` или `False`. В листинге 5.1 двум переменным присваивается целое число `1`, а другой переменной — `2`. Далее используется оператор равенства, чтобы показать, что первые две переменные равны, а третья нет. То же делается с оператором неравенства, результаты которого противоположны результатам оператора равенства с теми же входными данными.

Листинг 5.1. Операции проверки на равенство

```
# Присваиваем значения переменным
a, b, c = 1, 1, 2
# Проверяем, равно ли значение
a == b
True

a == c
False

a != b
False

a != c
True
```

С помощью операторов равенства и неравенства можно сравнивать разные типы объектов. Для числовых типов (числа с плавающей точкой и целые числа) значения сравниваются. Например, при сравнении целого числа `1` и числа с плавающей точкой `1.0` они оцениваются как равные:

```
1 == 1.0
True
```

Большинство других межтиповых сравнений возвращают `False`, вне зависимости от значения. Сравнение строки и целого числа всегда будет возвращать `False`:

```
'1' == 1
False
```

Веб-формы часто сообщают обо всех входных данных пользователя в виде строк. Распространенная проблема возникает при попытке сравнить эти данные из онлайн-форм, представляющих число, но в виде строки, с фактическим числом. Входные данные строки при сравнении с числом всегда оцениваются как `False`, даже если они — строковая версия того же значения.

Операции сравнения

Операторы сравнения используются для сравнения порядка объектов. Какой именно это порядок, зависит от типа сравниваемых объектов. Для чисел сравнение — это порядок числовой строки, для строк — значение используемого символа Юникода. Операторы сравнения — это меньше (`<`), меньше или равно (`<=`), больше (`>`) и больше или равно (`>=`). Листинг 5.2 показывает поведение разных операторов сравнения.

Листинг 5.2. Операции сравнения

```
a, b, c = 1, 1, 2
a < b
False

a < c
True

a <= b
True

a > b
False

a >= b
True
```

Есть случаи, когда можно использовать операции сравнения для двух объектов разных типов, таких как числовые, но большинство межтиповых сравнений недопустимы. При использовании оператора сравнения для несравнимых типов, таких как строки и списки, произойдет ошибка.

Булевы операции

В основе булевых операторов лежит булева алгебра, с которой вас, возможно, знакомили на курсах математики. Эти операции были впервые сформулированы ученым Джорджем Булем в XIX веке. В Python булевы операторы — это `and`, `or` и `not`. Операторы `and` и `or` принимают два аргумента, а `not` — лишь один.

Оператор `and` оценивается как `True`, если оба его аргумента оцениваются как `True`; в противном случае — как `False`. Оператор `or` оценивается как `True`, если любой из его аргументов оценивается как `True`; в противном случае — как `False`. Оператор `not` возвращает `True`, если его аргумент оценивается как `False`, и наоборот (листинг 5.3).

Листинг 5.3. Булевы операции

```
True and True
True

True and False
False

True or False
True

False or False
False

not False
True

not True
False
```

Оба оператора, `and` и `or`, шунтирующие. Это значит, что они оценивают свои входные выражения лишь в той степени, в какой это нужно для определения выходных данных. Например, у вас есть два метода, `returns_false()` и `returns_true()`, и вы используете их в качестве входных данных для оператора `and` так:

```
returns_false() and returns_true()
```

Если `returns_false()` возвращает `False`, то `returns_true()` не будет вызван, так как результат операции `and` уже определен. Точно так же, допустим, вы используете их как аргументы для операций `or`, например:

```
returns_true() or returns_false()
```

Здесь второй метод не будет вызван, если первый возвращает `True`.

Оператор `not` всегда возвращает одну из булевых констант — `True` или `False`. Другие два возвращают результат последнего оцененного выражения. Это очень полезно при оценке объектов.

Оценка объектов

Все объекты в Python оцениваются как `True` или `False`. Значит, вы можете использовать их как аргументы для булевых операций. Объекты, которые оцениваются как `False`, — это константы `None` и `False`, любое числовое значение, равное нулю, или что-то с нулевой длиной. Сюда входят пустые последовательности, такие как пустая строка (`" "`) или пустой список (`[ ]`). Почти все остальное оценивается как `True`.

Поскольку оператор `or` возвращает последнее оцененное им выражение, его можно использовать для создания значения по умолчанию, когда переменная оценивается как `False`:

```
a = ''
b = a or 'default value'
b
'default value'
```

Поскольку здесь первая переменная присваивается пустой строке с нулевой длиной, эта переменная оценивается как `False`. Оператор `or` оценивает это, а затем вычисляет и возвращает второе выражение.

Операторы if

Оператор `if` — это составной оператор, позволяющий разветвлять поведение вашего кода в зависимости от текущего состояния. Вы можете использовать его, только если выполняется выбранное условие, или применить более сложный оператор для выбора из нескольких действий в зависимости от множества условий. Управляемые операторы начинаются с ключевого слова `if`, за которым следует выражение (которое оценивается как `True` или `False`) и двоеточие.

Управляемые операторы либо следуют друг за другом в одной строке, разделяемые точкой с запятой:

```
if True:message="It's True!";print(message)
It's True!
```

либо выглядят как блок кода с отступами, разделенный новыми строками:

```
if True:
    message="It's True"
    print(message)
It's True
```

В обоих примерах управляющее выражение — это просто зарезервированная константа `True`, которая всегда оценивается как `True`. Есть два управляемых оператора: первый присваивает строку переменной `message`, а второй выводит значение этой переменной. Чаще всего лучше использовать синтаксис блока, как во втором примере, так как код становится более читаемым.

Если управляющее выражение оценивается как `False`, программа продолжает выполняться, пропуская управляемый (-е) оператор (-ы):

```
if False:
    message="It's True"
    print(message)
```

МОРЖОВЫЙ ОПЕРАТОР

Когда вы присваиваете значение переменной, Python не возвращает его. Распространенная ситуация — присвоение переменной с последующей проверкой ее значения. Например, вы можете присвоить переменной возвращенное функцией значение и, если оно не `None`, использовать возвращенный объект.

Метод `search` Python модуля `re` (см. главу 15) возвращает объект `match`, если находит его в строке, в противном случае он возвращает `None`. Поэтому, чтобы использовать объект `match`, нужно сначала убедиться, что он не принимает значение `None`:

```
import re
s = '2020-12-14'
match = re.search(r'(\d\d\d\d)-(\d\d)-(\d\d)', s)
if match:
    print(f"Matched items: {match.groups(1)}")
else:
    print(f"No match found in {s}")
```

В Python 3.8 появился новый оператор — оператор присваивания (`:=`). Его называют *моржовым оператором* из-за его сходства с головой моржа. Он присваивает значение переменной и возвращает его. Вы можете переписать пример с объектом `match`, используя этот оператор:

```
import re
s = '2020-12-14'
if match := re.search(r'(\d\d\d\d)-(\d\d)-(\d\d)', s):
    print(f"Matched items: {match.groups(1)}")
else:
    print(f"No match found in {s}")
```

Моржовый оператор создает менее сложный и более читабельный код.

Ниже приведен пример, где в качестве управляющего выражения используется тест принадлежности:

```
snack = 'apple'
fruit = {'orange', 'apple', 'pear'}
if snack in fruit:
    print(f"Yeah, {snack} is good!"
Yeah, apple is good!
```

Здесь проверяется, есть ли значение переменной `snack` во множестве `fruit`. Если это так, то выводится поощряющее сообщение.

Чтобы запустить альтернативный блок кода, когда управляющее выражение принимает значение `False`, используйте оператор `else`. Он состоит из ключевого слова `else`, за которым следует двоеточие и блок кода, который будет выполняться, только если предшествующее ему управляющее выражение оценивается как `False`. Это позволяет вам разветвлять логику вашего кода. Думайте об этом как о выборе действий в зависимости от текущего состояния. В листинге 5.4 показан `else`, добавленный к оператору `if`, связанному с переменной `snack`. Второй вывод оператора выполняется, только если управляющее выражением `snack` в `fruit` принимает значение `False`.

Листинг 5.4. Оператор else

```
snack = 'cake'
fruit = {'orange', 'apple', 'pear'}
if snack in fruit:
    print(f"Yeah, {snack} is good!")
else:
    print(f"{snack}!? You should have some fruit")
cake!? You should have some fruit
```

Если вы хотите несколько ветвей в своем коде, можете вложить операторы `if` и `else`, как в листинге 5.5. В этом примере сделано три выбора: первый — если баланс положительный, второй — если он на нуле, и последний — если отрицательный.

Листинг 5.5. Вложенные операторы else

```
balance = 2000.32
account_status = None

if balance > 0:
    account_status = 'Positive'
else:
    if balance == 0:
        account_status = 'Empty'
    else:
        account_status = 'Overdrawn'
print(account_status)
Positive
```

Хотя этот код правильный и будет работать как положено, читать его трудно. Чтобы сделать ветвление более лаконичным, можно использовать оператор `elif`, который добавляется после исходного оператора `if`. У него есть свое управляющее выражение, которое будет оценено, только если предыдущее выражение оператора было оценено как `False`. Листинг 5.6 выполняет ту же логику, что и листинг 5.5, но вложенные операторы `else` и `if` заменены в нем на `elif`.

Листинг 5.6. Оператор elif

```
balance = 2000.32
account_status = None

if balance > 0:
    account_status = 'Positive'
elif balance == 0:
    account_status = 'Empty'
else:
    account_status = 'Overdrawn'

print(account_status)
Positive
```

За счет сцепления множества операторов `elif` с оператором `if`, как в листинге 5.7, вы можете выполнять сложные выборы. Обычно `else` добавляется в конце, чтобы поймать момент, когда все управляющие выражения принимают значение `False`.

Листинг 5.7. Сцепление операторов elif

```
fav_num = 13

if fav_num in (3,7):
    print(f"{fav_num} is lucky")
elif fav_num == 0:
    print(f"{fav_num} is evocative")
elif fav_num > 20:
    print(f"{fav_num} is large")
elif fav_num == 13:
    print(f"{fav_num} is my favorite number too")
else:
    print(f"I have no opinion about {fav_num}")
is my favorite number too
```

Циклы while

Цикл `while` состоит из ключевого слова `while`, за которым следует управляющее выражение, двоеточие и управляющий блок кода. Управляемый оператор в цикле `while` выполняется, только если управляемый оценивается как `True`. Здесь он похож на `if`, но, в отличие от него, цикл `while` повторно продолжает выполнять управляющий блок, пока его управляющий оператор остается равным `True`. Ниже представлен цикл `while`, который выполняется, пока счетчик переменной меньше пяти:

```
counter = 0
while counter < 5:
    print(f"I've counted {counter} so far, I hope there aren't more")
    counter += 1
```

Заметьте, что переменная увеличивается с каждой итерацией. Это дает гарантию того, что цикл будет завершен. Вот как выглядят выходные данные после выполнения этого цикла:

```
I've counted 0 so far, I hope there aren't more
I've counted 1 so far, I hope there aren't more
I've counted 2 so far, I hope there aren't more
I've counted 3 so far, I hope there aren't more
I've counted 4 so far, I hope there aren't more
```

Цикл запускался пять раз, каждый раз увеличивая переменную.

ПРИМЕЧАНИЕ

Важно обеспечить условие выхода, или ваш цикл будет повторяться бесконечно.

Циклы for

Циклы `for` используются для итерации некоторой группы объектов. Эта группа может быть последовательностью, генератором, функцией или другим итерируемым объектом. Итерируемый объект — это любой объект, возвращающий серию элементов по одному за раз. Обычно циклы `for` используются для выполнения блока кода установленное количество раз или действия с каждым членом последовательности. Управляющий оператор цикла состоит из ключевого слова `for`, переменной, ключевого слова `in` и итерируемого, за которым следует двоеточие:

`for <переменная> in <итерируемое>:`

Переменная присваивается первому значению из итерируемого, управляющий блок выполняется с ним, и затем переменная присваивается следующему значению. Так продолжается, пока у итерируемого есть значения для возврата.

Общепринятый способ запуска блока кода установленное количество раз — использование цикла `for` с объектом диапазона в качестве итерируемого:

```
for i in range(6):
    j = i + 1
    print(j)
       1
       2
       3
       4
       5
       6
```

Здесь значения `0`, `1`, `2`, `3`, `4` и `5` присваиваются переменной `i`, запускающей блок кода для каждого из них.

Ниже приведен пример использования списка в качестве итерируемого:

```
colors = ["Green", "Red", "Blue"]
for color in colors:
    print(f"My favorite color is {color}")
    print("No, wait...")
My favorite color is Green
No, wait...
My favorite color is Red
No, wait...
My favorite color is Blue
No, wait...
```

Каждый элемент списка используется в блоке кода, и когда они заканчиваются, цикл завершается.

Операторы break и continue

Оператор `break` позволяет вам раньше завершить цикл `for` или `while`. Когда он оценен, текущий блок прекращает выполняться и цикл завершается. Обычно это используется в связке с вложенным оператором `if`. В листинге 5.8 показан цикл, чье управляющее выражение всегда принимает значение `True`. Вложенный оператор `if` вызывает `break`, когда выполняются его условия, завершая цикл в этой точке.

Листинг 5.8. Оператор break

```
fish = ['mackerel', 'salmon', 'pike']
beasts = ['salmon', 'pike', 'bear', 'mackerel']
i = 0

while True:
    beast = beasts[i]
    if beast not in fish:
        print(f"Oh no! It's not a fish, it's a {beast}")
        break
    print(f"I caught a {beast} with my fishing net")
    i += 1
I caught a salmon with my fishing net
I caught a pike with my fishing net
Oh no! It's not a fish, it's a bear
```

Оператор `continue` пропускает одну (текущую) итерацию цикла при вызове. Он тоже обычно используется в связке с вложенным оператором `if`. Ниже показано использование оператора `continue` для пропуска вывода имен, которые не начинаются на букву *b*.

Листинг 5.9. Оператор continue

```
for name in ['bob', 'billy', 'bonzo', 'fred', 'baxter']:
    if not name.startswith('b'):
            continue
    print(f"Fine fellow that {name}")
Fine fellow that bob
Fine fellow that billy
Fine fellow that bonzo
Fine fellow that baxter
```

Резюме

Такие составные операторы, как `if`, циклы `while` и `for`, — основополагающая часть кода, выходящая за рамки простых скриптов. Умение разветвлять и повторять код позволит вам формировать блоки действий, описывающие сложное составное поведение. Теперь у вас есть инструмент для построения более сложного ПО.

Вопросы для закрепления

1. Что выводит этот код, если переменная `a` присвоена пустому списку?

```
if a:
        print(f"Hiya {a}")
else:
        print(f"Biya {a}")
```

2. Что выводит предыдущий код, если переменной `a` присвоена строка `Henry`?
3. Напишите цикл `for`, выводящий числа от 0 до 9, пропуская 3, 5 и 7.

6

Функции

В нашей страсти к измерению
мы часто измеряем то, что можем, а не то, что хотим...
и забываем, что есть разница.

Джордж Удни Юл

В этой главе

- Объявление функции.
- Строки документации.
- Позиционные и ключевые параметры.
- Параметры подстановочного знака.
- Операторы возврата.
- Область видимости.
- Декораторы.
- Анонимные функции.

Последний и самый мощный составной оператор, который мы обсудим, — это функция. Она позволяет вам присвоить имя блоку кода, заключенному в оболочку объекта. Этот код затем можно будет вызвать, используя его имя, причем многократно и во множестве мест.

Объявление функций

Объявление функции задает ее объект, который обертывает исполняемый блок. Оно не запускает блок кода, а просто описывает, как функция может быть названа, какие параметры могут быть ей переданы и что будет выполняться при ее вызове. К составляющим функции относятся: управляющий оператор, опциональные строки документации, управляемый блок кода и оператор возврата.

Управляющий оператор

Первая строка объявления функции — это управляющий оператор следующего вида:

```
def <имя функции> (<параметры>):
```

Ключевое слово `def` указывает на объявление функции, *<имя функции>* — это место, где будет объявлено имя для вызова функции, *<параметры>* — место, где объявляются любые аргументы, которые могут быть переданы функции.

Следующая функция объявлена с именем `do_nothing` и единственным параметром `not_used`:

```
def do_nothing(not_used):
    pass
```

Блок кода здесь состоит из единственного оператора `pass`, который ничего не делает. В руководстве по оформлению кода Python PEP8 есть правила для именования функций (https://www.python.org/dev/peps/ pep-0008/#function-and-variable-names).

Строки документации

Следующая часть объявления функции — строка документации, или *docstring*, содержащая документацию для функции. Ее можно опустить, и компилятор Python не будет возражать. Рекомендуется указывать строки документации для всех методов, кроме самых очевидных.

Строка документации сообщает о ваших намерениях в написании функции, о том, что она делает и как ее вызывать. В PEP8 есть руководство по содержанию строк документации (https://www.python.org/dev/peps/pep-0008/#documentation-strings). Строка документации состоит из однострочной или многострочной строки, заключенной в три пары двойных кавычек, которая следует сразу за управляющим оператором:

```
def do_nothing(not_used):
    """This function does nothing."""
    pass
```

Для однострочной строки документации кавычки находятся на одной строке с текстом. Для многострочной они, как правило, расположены над или под текстом, как в листинге 6.1.

Листинг 6.1. Многострочная строка документации

```
def do_nothing(not_used):
    """
    This function does nothing.
    This function uses a pass statement to
    avoid doing anything.
    Parameters:
        not_used - a parameter of any type,
                   which is not used.
    """
    pass
```

Первая строка *docstring* должна быть утверждением, обобщающим действия функции. При более подробном объяснении после первого утверждения оставляют пустую строку. Есть много разных правил, касающихся содержания после первой строки *docstring*. Но обычно там объясняется, что делает функция, какие параметры принимает и что она должна возвращать. Строка документации полезна как для того, кто читает ваш код, так и для разных утилит, которые читают и отображают первую строку или полностью весь *docstring*. Например, при вызове функции `help()` для `do_nothing()` строка документации отображается как в листинге 6.2.

Листинг 6.2. Строка документации для help

```
help(do_nothing)
Help on function do_nothing in module __main__:
do_nothing(not_used)

This function does nothing.

This function uses a pass statement to avoid doing anything.
```

```
Parameters:
    not_used - a parameter of any type,
                which is not used.
```

Параметры

Параметры позволяют передавать значения в функцию, которые можно использовать в блоке ее кода. Они подобны переменным, присваиваемым функциям при их вызове. Но параметр может быть разным при каждом вызове функции.

Функция не обязана принимать любой параметр. Для такой функции оставьте круглые скобки после ее имени пустыми:

```
def no_params():
    print("I don't listen to nobody")
```

При вызове функции вы передаете значения для параметров в круглых скобках, которые следуют за именем функции. Значения можно установить на основе позиции, в которой они передаются, или ключевых слов. Функции могут быть объявлены так, чтобы их параметры передавались одним из способов или их комбинацией. Значения, передаваемые функции, присоединяются к переменным с именами, заданными в ее объявлении. В листинге 6.3 объявлено три параметра: `first`, `second` и `third`. Эти переменные доступны следующему блоку кода, который выводит значения для каждого параметра.

Листинг 6.3. Параметры согласно их позиции или ключевому слову

```
def does_order(first, second, third):
    '''Prints parameters.'''
    print(f'First: {first}')
    print(f'Second: {second}')
    print(f'Third: {third}')

does_order(1, 2, 3)
First: 1
Second: 2
Third: 3
does_order(first=1, second=2, third=3)
First: 1
Second: 2
Third: 3
does_order(1, third=3, second=2)
First: 1
Second: 2
Third: 3
```

В листинге 6.3 объявлена функция `does_order()`, которая будет вызываться трижды. Первый раз используется позиция аргументов `(1, 2, 3)` для присвоения значений переменных. Первое значение присваивается первому параметру, `first`, второе — второму, `second`, и третье — третьему, `third`.

Вызывая функцию `does_order()` во второй раз, листинг использует присвоение по ключевому слову, явно присваивая значения с использованием имен параметров (`first=1, second=2, third=3`). Во время третьего вызова первый параметр присваивается согласно его позиции, а другие два — с помощью присвоения по ключевому слову. Заметьте, что во всех трех случаях параметрам присваиваются одни и те же значения.

Присвоение по ключевым словам не зависит от их позиции. Можно без проблем присвоить `third=3` в позицию перед `second=2`. Но нельзя использовать присвоение по ключевому слову слева от позиционного присваивания:

```
does_order(second=2, 1, 3)
File "<ipython-input-9-eed80203e699>", line 1
    does_order(second=2, 1, 3)
                         ^
SyntaxError: positional argument follows keyword argument
```

Вы можете потребовать, чтобы параметр вызывался только с использованием метода ключевого слова. Для этого достаточно поставить * слева от него в объявлении функции. Все параметры справа от звездочки могут быть вызваны только с помощью ключевых слов. В листинге 6.4 показано, как сделать `third` параметром с обязательным ключевым словом, а затем вызвать его с помощью синтаксиса этого слова.

Листинг 6.4. Параметры с обязательным ключевым словом

```
def does_keyword(first, second, *, third):
    '''Prints parameters.'''
    print(f'First: {first}')
    print(f'Second: {second}')
    print(f'Third: {third}')

does_keyword(1, 2, third=3)
First: 1
Second: 2
Third: 3
```

При попытке вызова параметра с обязательным ключевым словом с помощью позиционного синтаксиса вы получите сообщение об ошибке:

```
does_keyword(1, 2, 3)
---------------------------------------------------------------------------
```

```
    TypeError Traceback (most recent call last)
<ipython-input-15-88b97f8a6c32> in <module>
----> 1 does_keyword(1, 2, 3)

TypeError: does_keyword() takes 2 positional arguments but 3 were given
```

Параметр можно сделать необязательным, присвоив ему значение по умолчанию в объявлении функции. Это значение будет использоваться, если во время вызова функции для параметра не предусмотрено значение. В листинге 6.5 объявлена функция `does_defaults()`, у третьего параметра которой значение по умолчанию `3`. Затем функция вызывается дважды: первый раз с помощью позиционного присваивания для всех трех параметров и второй — с помощью значения по умолчанию для третьего.

Листинг 6.5. Параметр со значением по умолчанию

```
def does_defaults(first, second, third=3):
    '''Prints parameters.'''
    print(f'First: {first}')
    print(f'Second: {second}')
    print(f'Third: {third}')

does_defaults(1, 2, 3)
First: 1
Second: 2
Third: 3
does_defaults(1, 2)
First: 1
Second: 2
Third: 3
```

Как и в случае ограничения порядка ключевых слов и позиционных аргументов во время вызова функции, вы не можете объявить функцию с помощью параметра со значением по умолчанию слева от параметра без него:

```
def does_defaults(first=1, second, third=3):
    '''Prints parameters.'''
    print(f'First: {first}')
    print(f'Second: {second}')
    print(f'Third: {third}')

File "<ipython-input-19-a015eaeb01be>", line 1
       def does_defaults(first=1, second, third=3):
                         ^
SyntaxError: non-default argument follows default argument
```

Значения по умолчанию устанавливаются в объявлении функции, а не при ее вызове. Значит, если вы используете изменяемый объект (список или словарь) в качестве этого значения, он будет создан для функции лишь единожды. Каждый раз при вызове функции с помощью этого значения по умолчанию будет использоваться тот же список или объект словаря. Если это не предусмотреть, можно столкнуться с трудноуловимыми сложностями. В листинге 6.6 объявлена функция со списком в качестве аргумента по умолчанию. Блок кода добавляет `1` в список. Заметьте, что каждый раз при вызове функции список удерживает значения из прошлых вызовов.

Листинг 6.6. Изменяемый объект в качестве параметра по умолчанию

```
def does_list_default(my_list=[]):
    '''Uses list as default.'''
    my_list.append(1)
    print(my_list)

does_list_default()
[1]

does_list_default()
[1, 1]

does_list_default()
[1, 1, 1]
```

Лучше избегать использования изменяемых объектов в качестве параметров по умолчанию, чтобы не сталкиваться со сложными в отслеживании ошибками и сбоями. В листинге 6.7 показан общий шаблон для обработки значений по умолчанию для изменяемых типов параметров.

Значение по умолчанию в объявлении функции равно `None`. Блок кода проверяет наличие у параметра присвоенного значения. Если его нет, создается новый список и присваивается переменной. Поскольку список создается в блоке кода, это происходит при каждом вызове функции без значения для параметра.

Листинг 6.7. Шаблон по умолчанию в блоке кода

```
def does_list_param(my_list=None):
    '''Assigns default in code to avoid confusion.'''
    my_list = my_list or []
    my_list.append(1)
    print(my_list)

does_list_param()
[1]

does_list_param()
```

```
[1]

does_list_param()
[1]
```

Начиная с Python 3.8, вы можете ограничивать параметры только позиционным присваиванием, ставя их слева от косой черты (`/`) в объявлении функции. В листинге 6.8 функция `does_positional` объявлена так, что первый ее параметр (`first`) только позиционный.

Листинг 6.8. Только позиционные параметры (версия Python 3.8 и выше)

```
def does_positional(first, /, second, third):
    '''Demonstrates a positional parameter.'''
    print(f'First: {first}')
    print(f'Second: {second}')
    print(f'Third: {third}')

does_positional(1, 2, 3)
First: 1
Second: 2
Third: 3
```

При попытке вызвать `does_positional` с помощью присвоения по ключевому слову для `first` вы получите сообщение об ошибке:

```
does_positional(first=1, second=2, third=3)
---------------------------------------------------------------------------
TypeError Traceback (most recent call las t)
<ipython-input-24-7b1f45f64358> in <module>
----> 1 does_positional(first=1, second=2, third=3)
TypeError: does_positional() got some positional-only arguments passed as
keyword arguments: 'first'
```

В листинге 6.9 функция `does_positional` изменена так, чтобы использовать одновременно только позиционные и только ключевые параметры. Параметр `first` исключительно позиционный, `second` может быть задан с помощью позиционного или ключевого присвоения, а последний, `third`, только с помощью ключевого.

Листинг 6.9. Только позиционные и только ключевые параметры

```
def does_positional(first, /, second, *, third):
    '''Demonstrates a positional and keyword parameters.'''
    print(f'First: {first}')
    print(f'Second: {second}')
    print(f'Third: {third}')
```

```
does_positional(1, 2, third=3)
First: 1
Second: 2
Third: 3
```

Вы можете использовать подстановочные знаки в объявлении функции, чтобы принимать неустановленное количество позиционных или ключевых аргументов. Так часто делается, когда функция вызывает функцию из внешнего программного интерфейса приложения (API). Функция может передавать аргументы без запроса объявления всех параметров внешнего API.

Чтобы использовать подстановочный знак для позиционных параметров, используйте символ *. Листинг 6.10 показывает объявление функции с позиционным подстановочным знаком *args. Блок кода получает любые позиционные аргументы, заданные в вызове функции, как элементы в списке с именем args. Это функция просматривает список, выводя каждый элемент. Далее вызывается функция с аргументами 'Donkey', 3 и ['a']. Каждый из них доступен из списка и выводится на экран.

Листинг 6.10. Позиционные параметры подстановочного знака

```
def does_wildcard_positions(*args):
    '''Demonstrates wildcard for positional parameters.'''
    for item in args:
        print(item)

does_wildcard_positions('Donkey', 3, ['a'])
Donkey
3
['a']
```

Для объявления функции с ключевыми параметрами через подстановочный знак установите параметр, начиная с **. В листинге 6.11 объявлена функция does_wildcard_keywords с параметром **kwargs. В блоке кода ключевые параметры доступны в виде ключей и значений в словаре kwargs.

Листинг 6.11. Ключевые параметры через подстановочный знак

```
def does_wildcard_keywords(**kwargs):
    '''Demonstrates wildcard for keyword parameters.'''
    for key, value in kwargs.items():
        print(f'{key} : {value}')

does_wildcard_keywords(one=1, name='Martha')
one : 1
name : Martha
```

Можно использовать и позиционные, и ключевые параметры подстановочных знаков в одной функции: просто первыми определите позиционные, а вторыми — ключевые параметры. В листинге 6.12 показана функция, использующая оба типа параметров.

Листинг 6.12. Позиционные и ключевые параметры подстановочных знаков

```
def does_wildcards(*args, **kwargs):
    '''Demonstrates wildcard parameters.'''
    print(f'Positional: {args}')
    print(f'Keyword: {kwargs}')

does_wildcards(1, 2, a='a', b=3)
Positional: (1, 2)
Keyword: {'a': 'a', 'b': 3}
```

Операторы возврата

Эти операторы определяют, какое значение вычисляет функция при вызове. Оператор возврата состоит из ключевого слова `return`, за ним следует выражение, которое может быть простым значением, более сложным вычислением или вызовом другой функции. В листинге 6.13 объявлена функция, принимающая число в качестве аргумента и возвращающая его плюс 1.

Листинг 6.13. Возвращаемое значение

```
def adds_one(some_number):
    '''Demonstrates return statement.'''
    return some_number + 1

adds_one(1)
2
```

У каждой функции Python есть возвращаемое значение. Если вы не определили оператор `return` явно, функция вернет значение `None`.

```
def returns_none():
    '''Demonstrates default return value.'''
    pass

returns_none() == None
True
```

В этом примере оператор `return` опускается, а затем проверяется, что возвращаемое значение равно `None`.

Область видимости в функциях

Область видимости — это доступность объектов, объявленных в коде. Переменная, объявленная в глобальной области видимости, доступна во всем вашем коде, а та, что объявлена в локальной области видимости, доступна только там. В листинге 6.14 объявлены переменные `outer` и `inner`. Обе доступны в блоке кода функции `shows_scope` после их вывода на экран.

Листинг 6.14. Локальная и глобальная области видимости

```
outer = 'Global scope'

def shows_scope():
    '''Demonstrates local variable.'''
    inner = 'Local scope'
    print(outer)
    print(inner)

shows_scope()
Global scope
Local scope
```

Переменная `inner` — локальная для функции. Она объявлена в ее блоке кода. Если вы попробуете вызвать `inner` извне, она не будет определена:

```
print(inner)
---------------------------------------------------------------------------
NameError Traceback (most recent call last)
<ipython-input-39-9504624e1153> in <module>
----> 1 print(inner)
NameError: name 'inner' is not defined
```

Понимание области видимости пригодится при использовании декораторов, описанных далее.

Декораторы

Декоратор позволяет вам создавать функции, изменяющие другие функции. Обычно они используются для настройки логирования журнала с помощью набора соглашений или сторонних библиотек. Возможно, вам и не понадобится писать свои декораторы, но полезно понимать принцип их работы. В этом разделе мы поговорим об основных концепциях.

В Python все является объектами, включая функции. Это значит, что в переменной вы можете ссылаться на функцию. В листинге 6.15 объявлена функция `add_one(n)`,

которая принимает число и добавляет к нему 1. Далее создается переменная `my_func` с функцией `add_one()` в качестве значения.

> **ПРИМЕЧАНИЕ**
>
> Если вы не вызываете функцию, не используйте круглые скобки при присвоении переменной. Опуская круглые скобки, вы ссылаетесь на объект функции, а не на возвращаемое значение. Это можно увидеть в листинге 6.15, где выводится my_func, являющаяся объектом функции. Позже вы можете вызвать функцию, добавив круглые скобки и аргумент в my_func, которая возвращает аргумент плюс 1.

Листинг 6.15. Функция как значение переменной

```
def add_one(n):
    '''Adds one to a number.'''
    return n + 1

my_func = add_one
print(my_func)
<function add_one at 0x1075953a0>

my_func(2)
3
```

Поскольку функции — это объекты, их можно использовать вместе с такими структурами данных, как словари или списки. В листинге 6.16 объявлены две функции, помещенные в список, на который указывает переменная `my_functions`. Далее выполняется итерация через список. Каждая функция во время ее итерации присваивается переменной `my_func` и вызывается во время блока кода цикла `for`.

Листинг 6.16. Вызов списка функций

```
def add_one(n):
    '''Adds one to a number.'''
    return n + 1

def add_two(n):
    '''Adds two to a number.'''
    return n + 2

my_functions = [add_one, add_two]

for my_func in my_functions:
    print(my_func(1))
2
3
```

Python позволяет объявить одну функцию как часть блока кода другой. Такая функция называется *вложенной*. В листинге 6.17 объявлена функция `nested()` в блоке кода функции `called_nested()`. Эта вложенная функция затем используется как возвращаемое значение для внешней.

Листинг 6.17. Вложенные функции

```
def call_nested():
    '''Calls a nested function.'''
    print('outer')

    def nested():
        '''Prints a message.'''
        print('nested')

    return nested

my_func = call_nested()
outer
my_func()
nested
```

Можно обернуть одну функцию другой, добавив функциональность до или после. В листинге 6.18 `add_one(number)` оборачивается функцией `wrapper(number)`. Обертывающая функция принимает параметр `number`, который потом передает обернутой функции. У нее есть операторы до и после вызова `add_one(number)`. Порядок операторов вывода можно увидеть при вызове `wrapper(1)` и заметить, что он возвращает ожидаемые значения из `add_one`: `1` и `2`.

Листинг 6.18. Обертывающие функции

```
def add_one(number):
    '''Adds to a number.'''
    print('Adding 1')
    return number + 1

def wrapper(number):
    '''Wraps another function.'''
    print('Before calling function')
    retval = add_one(number)
    print('After calling function')
    return retval

wrapper(1)
Before calling function
Adding 1
After calling function
2
```

Можно пойти дальше и использовать функцию в качестве параметра. Вы можете передать ее в качестве значения функции, у которой есть вложенное определение, обертывающее переданную функцию.

Взглянем на листинг 6.19. Сначала здесь объявляется функция `add_one(number)`, как и раньше. Но теперь объявляется функция `wrapper(number)`, вложенная в блок кода новой функции `do_wrapping(some_func)`. Эта новая функция принимает функцию в качестве аргумента и затем использует ее в объявлении `wrapper(number)`. Далее возвращается вновь объявленная версия `wrapper(number)`. Присвоив этот результат переменной и вызвав ее, вы можете увидеть обернутые результаты.

Листинг 6.19. Вложенная обертывающая функция

```
def add_one(number):
    '''Adds to a number.'''
    print('Adding 1')
    return number + 1

def do_wrapping(some_func):
    '''Returns a wrapped function.'''
    print('wrapping function')

    def wrapper(number):
        '''Wraps another function.'''
        print('Before calling function')
        retval = some_func(number)
        print('After calling function')
        return retval

    return wrapper

my_func = do_wrapping(add_one)
wrapping function

my_func(1)
Before calling function
Adding 1
After calling function
2
```

`do_wrapping(some_func)` можно использовать для обертывания любой функции. Например, если у вас есть функция `add_two(number)`, можете передать ее в качестве аргумента, как вы сделали с функцией `add_one(number)`:

```
my_func = do_wrapping(add_two)
my_func(1)
wrapping function
Before calling function
Adding 2
After calling function
3
```

Декораторы предоставляют синтаксис, который может упростить этот тип обертывания функции. Вместо вызова `do_wrapping(some_func)`, присваивания ее переменной и затем вызова ее из этой переменной можно просто поместить `@do_wrapping` в начало объявления функции. Потом `add_one(number)` можно будет вызвать напрямую, а обертывание произойдет в скрытом режиме.

В листинге 6.20 видно, что `add_one(number)` обернуто так же, как в листинге 6.18, но синтаксис декоратора в нем проще.

Листинг 6.20. Синтаксис декоратора

```
def do_wrapping(some_func):
    '''Returns a wrapped function.'''
    print('wrapping function')

    def wrapper(number):
        '''Wraps another function.'''
        print('Before calling function')
        retval = some_func(number)
        print('After calling function')
        return retval

    return wrapper

@do_wrapping
def add_one(number):
    '''Adds to a number.'''
    print('Adding 1')
    return number + 1
wrapping function

add_one(1)
Before calling function
Adding 1
After calling function
2
```

Анонимные функции

В большинстве случаев при объявлении функций вы захотите использовать синтаксис для именованных функций, который вы видели до сих пор. Но есть альтернатива: неименованные, анонимные функции. В Python они известны как лямбда-функции, и синтаксис их выглядит так:

```
lambda <параметр>: <оператор>
```

где `lambda` — это ключевое слово, обозначающее лямбда-функцию, *<параметр>* — это входной параметр, а *<оператор>* — это оператор для выполнения с помощью параметра. Результат *<оператор>* — возвращаемое значение.

Вот так объявляется лямбда-функция, добавляющая один к входному значению:

```
lambda x: x +1
```

В целом без лямбда-функций ваш код будет легче читать, использовать и отлаживать. Они полезны лишь в случае, когда одна простая функция применяется в качестве аргумента к другой. В листинге 6.21 объявлена функция `apply_to_list(data, my_func)`, принимающая список и функцию в качестве аргументов. Лямбда-функция отлично подойдет для вызова этой функции с целью добавления 1 к каждому элементу списка.

Листинг 6.21. Лямбда-функция

```
def apply_to_list(data, my_func):
    '''Applies a function to items in a list.'''
    for item in data:
        print(f'{my_func(item)}')

apply_to_list([1, 2, 3], lambda x: x + 1)
2
3
4
```

Резюме

Функции, являющиеся важными элементами при создании сложных программ, — это многократно используемые блоки кода. Функции документируются с помощью строк документации (docstring) и могут принимать параметры разными способами. Оператор возврата используется функцией для передачи значения при завершении ее выполнения. Декораторы — это специальные функции-обертки для других функций. Лямбда-функции — это анонимные, неименованные функции.

Вопросы для закрепления

Для ответов на вопросы 1–3 обратитесь к листингу 6.22.

Листинг 6.22. Функции для вопросов 1–3

```
def add_prefix(word, prefix='before-'):
    '''Prepend a word.'''
    return f'{prefix}{word}'3

def return_one():
    return 1

    def wrapper():
        print('a')
        retval = return_one()
        print('b')
        print(retval)
```

1. Каковы выходные данные для следующего вызова:
   ```
   add_prefix('nighttime', 'after-')
   ```
2. Каковы выходные данные для следующего вызова:
   ```
   add_prefix('nighttime')
   ```
3. Каковы выходные данные для следующего вызова:
   ```
   add_prefix()
   ```
4. Какую строку нужно разместить над объявлением функции, чтобы декорировать ее с помощью `standard_logging`?
   ```
   *standard_logging
   **standard_logging
   @standard_logging
   [standard_logging]
   ```
5. Что будет выведено на экран при следующем вызове:
   ```
   wrapper()
   ```

ЧАСТЬ II

БИБЛИОТЕКИ DATA SCIENCE

7

Библиотека NumPy

Все должно быть настолько просто,
насколько это возможно, но не проще.

Роджер Сешнс (перефр. Эйнштейна)

В этой главе

- Знакомство со сторонними библиотеками.
- Создание массивов NumPy.
- Индексация и слайсинг массивов.
- Фильтрация данных массива.
- Методы массива.
- Бродкастинг.

Это первая глава данной книги, посвященная библиотекам Data Science. Функциональные возможности Python, рассмотренные до этого момента, делают его мощным базовым языком. Библиотеки, о которых вы узнаете здесь, делают его доминирующим в data science. Первой мы рассмотрим NumPy. Эта библиотека служит основой для большинства других библиотек data science. В текущей главе вы узнаете о массиве NumPy — эффективной многомерной структуре данных Python.

СТОРОННИЕ БИБЛИОТЕКИ

Код Python организован в виде библиотек. Все функциональные возможности, о которых вы прочли, доступны в стандартной библиотеке Python, которая загружается при установке любой версии этого языка.

Сторонние библиотеки могут предоставить гораздо больше возможностей. Они разрабатываются и обслуживаются командами за пределами организации, поддерживающей сам Python. Существование этих команд и библиотек создает динамичную экосистему, сохраняющую доминирующее положение Python в мире программирования. Большинство этих библиотек доступны в среде Colab, и вы легко можете импортировать их в файл. Если вы работаете за пределами Colab-среды, вам может понадобиться установить их. Обычно это можно сделать с помощью пакетного менеджера Python `pip`.

Установка и импорт NumPy

NumPy уже установлен в среде Colab — вам нужно лишь импортировать его. Если вы работаете за пределами этой среды, есть несколько способов его установки (перечисленные по ссылке https://scipy.org/install.html). Наиболее распространенный из них — использование `pip`:

```
pip install numpy
```

После установки NumPy вы можете его импортировать. При импорте любой библиотеки можно изменить ее название в вашей среде с помощью ключевого слова `as`. NumPy обычно переименовывают в `np`:

```
import numpy as np
```

После установки и импорта библиотеки можно получить доступ ко всему функционалу NumPy через объект `np`.

Создание массивов

Массив NumPy — это структура данных, созданная для эффективной обработки операций с большими наборами данных. Эти наборы могут быть разной размерности и содержать многочисленные типы данных, но не в одном и том же объекте. Массивы NumPy используются как входные и выходные данные для других библиотек. Они применяются и в качестве основы других структур данных, важных для Data Science, таких как Pandas и SciPy.

Вы можете создавать массивы из других структур данных или инициализировать их с заданными значениями. В листинге 7.1 показаны разные способы создания одномерного массива. Здесь видно, что объект массива отображается как имеющий в качестве данных внутренний список. На самом деле данные не хранятся в списках, но такое отображение облегчает чтение массива.

Листинг 7.1. Создание массива

```
np.array([1,2,3])     # Массив из списка
array([1, 2, 3])

np.zeros(3)           # Массив нулей
array([0., 0., 0.])

np.ones(3)            # Массив единиц
array([1., 1., 1.])

np.empty(3)           # Массив произвольных данных
array([1., 1., 1.])

np.arange(3)          # Массив из диапазона чисел
array([0, 1, 2])

np.arange(0, 12, 3)   # Массив из диапазона чисел
array([0, 3, 6, 9])

np.linspace(0, 21, 7) # Массив интервалов
array([ 0. , 3.5, 7. , 10.5, 14. , 17.5, 21. ])
```

У массивов есть размерность (количество измерений). У одномерного массива измерение только одно и представляет количество элементов. В случае метода `np.array` размер измерения (-ий) совпадает с размером списка (-ов), используемого (-ых) в качестве входных данных. Для `np.zeros`, `np.ones` и `np.empty` измерение задается в качестве явного аргумента.

Метод `np.range` создает массив, подобно последовательности диапазонов. Полученные размерности и значения совпадают с теми, что созданы с помощью диапазона. Вы можете устанавливать значения начала и конца, а также размер шага.

`np.linspace` создает равномерно распределенные числа внутри интервала. Первые два аргумента определяют интервал, а третий — количество элементов.

Метод `np.empty` полезен при создании больших массивов. Помните, что, поскольку данные произвольные, вам нужно его использовать только при случаях, требующих замены всех исходных данных.

В листинге 7.2 показаны некоторые атрибуты массива.

Листинг 7.2. Характеристики массива

```
oned = np.arange(21)
oned
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
        11, 12, 13, 14, 15, 16, 17, 18, 19, 20 ])

oned.dtype     # Тип данных
dtype('int64')

oned.size      # Количество элементов
21

oned.nbytes    # Объем памяти (в байтах), используемый элементами массива
168

oned.shape     # Количество элементов в каждом измерении
(21,)

oned.ndim      # Количество измерений
1
```

При проверке типа данных для массива можно увидеть, что это `numpy.ndarray` (псевдоним `np` в этом случае не отображается):

```
type(oned)
numpy.ndarray
```

ПРИМЕЧАНИЕ

ndarray — это сокращенное название n-мерного массива (n-dimensional array).

Как уже упоминалось, вы можете создавать массивы с несколькими измерениями. Двумерные используются как матрицы. В листинге 7.3 создан такой массив из списка трех трехэлементных списков. Как видите, у полученного массива форма 3 × 3 и два измерения.

Листинг 7.3. Матрица из списков

```
list_o_lists = [[1,2,3],
```

```
              [4,5,6],
              [7,8,9]]

twod = np.array(list_o_lists)
twod
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

twod.shape
(3, 3)

twod.ndim
2
```

С помощью метода `reshape` можно создать массив с теми же элементами, но с другой размерностью. Этот метод принимает новую форму в качестве аргумента. В листинге 7.4 показано использование одномерного массива для создания двумерного, а затем создания из последнего одномерного и трехмерного массивов.

Листинг 7.4. Применение reshape

```
oned = np.arange(12)
oned
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

twod = oned.reshape(3,4)
twod
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])

twod.reshape(12)
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

twod.reshape(2,2,3)
array([[[ 0, 1, 2],
        [ 3, 4, 5]],
       [[ 6, 7, 8],
       [ 9, 10, 11]]])
```

Предоставляемая массиву форма должна соответствовать числу элементов в нем. Например, если вы возьмете массив из 12 элементов `twod` и попытаетесь задать его размерность по форме, не включающей в себя 12 элементов, то получите ошибку:

```
twod.reshape(2,3)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

```
<ipython-input-295-0b0517f762ed> in <module>
----> 1 twod.reshape(2,3)

ValueError: cannot reshape array of size 12 into shape (2,3)
```

Обычно изменение формы применяется для методов `np.zeros`, `np.ones` и `np.empty`, чтобы создать многомерные массивы с заданными по умолчанию значениями. Например, можно создать трехмерный массив единиц:

```
np.ones(12).reshape(2,3,2)
array([[[1., 1.],
        [1., 1.],
        [1., 1.]],
       [[1., 1.],
        [1., 1.],
        [1., 1.]]])
```

Индексация и слайсинг

Доступ к данным массивов можно получить через индексацию и слайсинг. В листинге 7.5 видно, что индексация и слайсинг одномерного массива выполняются по аналогии со списком. Можно проиндексировать отдельные элементы с начала или конца массива, указав номер индекса или несколько элементов с помощью среза.

Листинг 7.5. Индексация и слайсинг одномерного массива

```
oned = np.arange(21)
oned
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
          11, 12, 13, 14, 15, 16, 17, 18, 19, 20 ])

oned[3]
3

oned[-1]
20

oned[3:9]
array([3, 4, 5, 6, 7, 8])
```

Для многомерных массивов вы можете указать один аргумент для каждого измерения. Если вы опустите аргумент для измерения, то же по умолчанию произойдет и со всеми другими элементами этого измерения. Поэтому, если для двумерного массива в качестве аргумента предоставлено одно число, оно будет указывать

на возвращаемый ряд. При предоставлении как аргументов единственного числа для всех измерений вам вернется один элемент.

Можно также предоставить срез для каждого измерения. Так вы получите подмассив элементов, измерения которых определены длиной ваших срезов. В листинге 7.6 показаны разные варианты индексации и слайсинга двумерного массива.

Листинг 7.6. Индексация и слайсинг двумерного массива

```
twod = np.arange(21).reshape(3,7)
twod
array([[ 0, 1, 2, 3, 4, 5, 6],
       [ 7, 8, 9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20]])

twod[2]           # Доступ к строке 2
array([14, 15, 16, 17, 18, 19, 20])

twod[2, 3]        # Доступ к элементу в строке 2, столбец 3
17

twod[0:2]         # Доступ к строкам 0 и 1
array([[ 0, 1, 2, 3, 4, 5, 6],
       [ 7, 8, 9, 10, 11, 12, 13]])

twod[:, 3]        # Доступ к столбцу 3 всех строк
array([ 3, 10, 17])

twod[0:2, -3:]    # Доступ к последним трем столбцам строк 0 и 1
array([[ 4, 5, 6],
       [11, 12, 13]])
```

Присвоить новые значения существующему массиву можно по аналогии со списком, используя индексацию и слайсинг. При присвоении значения срезу он весь обновляется новым значением. В листинге 7.7 показано, как обновить отдельный элемент и срез в двумерном массиве.

Листинг 7.7. Изменение значений в массиве

```
twod = np.arange(21).reshape(3,7)
twod
array([[ 0, 1, 2, 3, 4, 5, 6],
       [ 7, 8, 9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20]])

twod[0,0] = 33
twod
array([[33, 1, 2, 3, 4, 5, 6],
```

```
       [ 7, 8, 9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20]])

twod[1:,:3] = 0
array([[33, 1, 2, 3, 4, 5, 6],
      [ 0, 0, 0, 10, 11, 12, 13],
      [ 0, 0, 0, 17, 18, 19, 20]])
```

Поэлементные операции

Массив — это не последовательность. По некоторым характеристикам массивы похожи на списки, и на каком-то уровне данные в массиве легко представить как список списков. Но между массивами и последовательностями есть много различий. Одно из них касается выполнения операций между элементами в двух массивах или последовательностях.

Помните, что при выполнении таких операций, как умножение с последовательностью, операция выполняется именно с ней, а не с ее содержимым. Поэтому при умножении списка на ноль вы получите список длиной, равной нулю:

```
[1, 2, 3]*0
[]
```

Вы не можете перемножить два списка, даже если они одинаковой длины:

```
[1, 2, 3]*[4, 5, 6]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-325-f525a1e96937> in <module>
----> 1 [1, 2, 3]*[4, 5, 6]

TypeError: can't multiply sequence by non-int of type 'list'
```

Можно написать код для выполнения операций между элементами списка. Листинг 7.8 показывает выполнение цикла по двум спискам для создания третьего, содержащего результат множественных пар элементов. Функция `zip()` используется для объединения двух списков в список кортежей, где каждый кортеж содержит элементы из каждого исходного списка.

Листинг 7.8. Поэлементные операции со списками

```
L1 = list(range(10))
L2 = list(range(10, 0, -1))
```

```
L1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

L2
 [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

L3 = []
for i, j in zip(L1, L2):
L3.append(i*j)
L3
[0, 9, 16, 21, 24, 25, 24, 21, 16, 9]
```

Хотя можно использовать циклы для выполнения поэлементных операций над списками, намного проще использовать для таких операций массивы NumPy. Они выполняют поэлементные операции по умолчанию. В листинге 7.9 показаны операции умножения, сложения и деления между двумя массивами. Заметьте, что в каждом случае операция выполняется между элементами массивов.

Листинг 7.9. Поэлементные операции с массивами

```
array1 = np.array(L1)
array2 = np.array(L2)
array1*array2
array([ 0,  9, 16, 21, 24, 25, 24, 21, 16, 9])

array1 + array2
array([10, 10, 10, 10, 10, 10, 10, 10, 10, 10])

array1 / array2
array([0.        , 0.11111111, 0.25      , 0.42857143, 0.66666667,
       1.        , 1.5       , 2.33333333, 4.        , 9.        ])
```

Фильтрация значений

Одно из самых используемых свойств массивов NumPy и структур данных, построенных поверх них, — способность фильтровать значения на основе выбранных условий. Так вы можете использовать массив для ответа на вопросы о ваших данных.

Листинг 7.10 показывает двумерный массив целых чисел `twod`. У второго массива `mask` размерность та же, что и у `twod`, но он содержит булевы значения. Массив `mask` определяет, какие элементы из `twod` возвращать. Итоговый массив содержит элементы из `twod`, соответствующие позиции которых в `mask` принимают значение `True`.

Листинг 7.10. Фильтрация с помощью булевых значений

```
twod = np.arange(21).reshape(3,7)
twod
```

```
array([[ 0, 1, 2, 3, 4, 5, 6],
       [ 7, 8, 9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20]])

mask = np.array([[ True, False, True, True, False, True, False],
                 [ True, False, True, True, False, True, False],
                 [ True, False, True, True, False, True, False]])
twod[mask]
array([ 0, 2, 3, 5, 7, 9, 10, 12, 14, 16, 17, 19])
```

Операторы сравнения, которые мы видим, возвращают единичное булево значение до возврата массива при использовании вместе с массивами. Поэтому если вы используете оператор < напротив массива `twod` так, как указано ниже, то в результате получите массив со значением `True` для каждого элемента меньше пяти и `False` для остальных:

```
twod < 5
```

Этот результат можно использовать как маску, чтобы получить только значения, равные `True` при сравнении. В листинге 7.11 создана маска и возвращены только значения массива `twod` меньше 5.

Листинг 7.11. Фильтрация с помощью сравнения

```
mask = twod < 5
mask
array([[ True, True, True, True],
       [ True, False, False, False],
       [False, False, False, False]])

twod[mask]
array([0, 1, 2, 3, 4])
```

Как видите, использование операторов сравнения и порядка поможет вам с легкостью извлечь сведения из данных. Также можно сочетать эти сравнения для создания более сложных масок. В листинге 7.12 используется символ `&` для объединения двух условий, чтобы создать маску, которая оценивается как `True` только для элементов, удовлетворяющих обоим.

Листинг 7.12. Фильтрация с помощью множественных сравнений

```
mask = (twod < 5) & (twod%2 == 0)
mask
array([[ True, False, True, False],
       [ True, False, False, False],
       [False, False, False, False]])

twod[mask]
array([0, 2, 4])
```

ПРИМЕЧАНИЕ

Фильтрация с помощью маски — процесс, который вы будете выполнять снова и снова, особенно с датафреймами Pandas, построенными поверх массивов NumPy. О датафреймах вы узнаете подробнее в главе 9.

Представления и копии

Массивы NumPy созданы для эффективной работы с большими наборами данных. В этом вам поможет использование представлений. Возвращенный после слайсинга или фильтрации массив становится, когда это возможно, представлением, а не копией. Представление позволяет вам посмотреть на те же данные по-новому.

Важно понимать, что память и вычислительная мощность не используются для создания копий данных каждый раз при выполнении слайсинга или фильтрации. Меняя значение в представлении массива, вы меняете это значение в исходном массиве и в любых других представлениях, отображающих этот элемент. В листинге 7.13 взяли срез из массива `data1` и назвали его `data2`. Затем значение `11` в `data2` заменили на `-1`. Когда вы вернетесь в `data1`, то увидите, что элемент, значение которого было `11`, теперь принял значение `-1`.

Листинг 7.13. Изменение значений в представлениях

```
data1 = np.arange(24).reshape(4,6)
data1
array([[ 0, 1, 2, 3, 4, 5],
       [ 6, 7, 8, 9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])

data2 = data1[:2,3:]
data2
array([[ 3, 4, 5],
       [ 9, 10, 11]])

data2[1,2] = -1
data2
array([[ 3, 4, 5],
       [ 9, 10, -1]])

data1
array([[ 0, 1, 2, 3, 4, 5],
       [ 6, 7, 8, 9, 10, -1],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

Такое поведение может привести к сбоям и ошибкам в вычислении. Но, разобравшись в нем, вы можете получить весомые преимущества при работе с большими наборами данных. Чтобы поменять данные из среза или после фильтрации без изменения исходного массива, можно сделать копию. Например, в листинге 7.14 при изменении элемента в копии исходный массив остается неизменным.

Листинг 7.14. Изменение значений в копии

```
data1 = np.arange(24).reshape(4,6)
data1
array([[ 0, 1, 2, 3, 4, 5],
       [ 6, 7, 8, 9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])

data2 = data1[:2,3:].copy()
data2
array([[ 3, 4, 5],
       [ 9, 10, 11]])

data2[1,2] = -1
data2
array([[ 3, 4, 5],
       [ 9, 10, -1]])

data1
array([[ 0, 1, 2, 3, 4, 5],
       [ 6, 7, 8, 9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

Методы массива

В массивах NumPy есть встроенные методы как для получения статистических сводных данных, так и для выполнения операций над матрицами. Листинг 7.15 показывает методы для выполнения сводной статистики. Они бывают нескольких видов: для получения максимума, минимума, суммы, среднего значения и стандартного отклонения. Если ось не задана, эти методы дают результаты по всему массиву. Если же указано значение оси, равное `1`, выполняется массив с результатами для каждой строки, а при значении оси, равном `0`, — для каждого столбца.

Листинг 7.15. Интроспекция

```
data = np.arange(12).reshape(3,4)
data
array([[ 0, 1, 2, 3],
```

```
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])

data.max()         # Максимальное значение
11

data.min()         # Минимальное значение
0

data.sum()         # Сумма всех значений
66

data.mean()        # Среднее значение
5.5

data.std()         # Стандартное отклонение
3.452052529534663

data.sum(axis=1)   # Сумма каждой строки
array([ 6, 22, 38])

data.sum(axis=0)   # Сумма каждого столбца
array([12, 15, 18, 21])

data.std(axis=0)   # Стандартное отклонение каждой строки
array([3.26598632, 3.26598632, 3.26598632, 3.26598632])

data.std(axis=1))  # Стандартное отклонение каждого столбца
array([1.11803399, 1.11803399, 1.11803399])
```

В листинге 7.16 показаны некоторые матричные операции с массивами: возврат транспонирования, возврат произведения матриц и возврат диагонали. Помните, что для поэлементного умножения можно использовать оператор * между двумя массивами. Чтобы вычислить скалярное произведение двух матриц, используйте оператор `@` или метод `.dot()`.

Листинг 7.16. Матричные операции

```
A1 = np.arange(9).reshape(3,3)
A1
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

A1.T             # Транспонирование
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
```

```
A2 = np.ones(9).reshape(3,3)
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])

A1 @ A2          # Произведение матриц
array([[ 3., 3., 3.],
       [12., 12., 12.],
       [21., 21., 21.]])

A1.dot(A2)       # Скалярное произведение
array([[ 3., 3., 3.],
       [12., 12., 12.],
       [21., 21., 21.]])

A1.diagonal()    # Диагональ матрицы
array([0, 4, 8])
```

В отличие от большинства типов последовательностей, массив может содержать только один тип данных. У вас не может быть массива, содержащего строки и целые числа. Если тип данных не указан, NumPy угадывает его.

Когда вы начинаете с целых чисел, NumPy устанавливает тип данных `int64` (листинг 7.17). Проверив атрибут `nbytes`, можно увидеть, что данные для этого массива занимают 800 байтов памяти.

Листинг 7.17. Автоматическая установка типа

```
darray = np.arange(100)
darray
array([  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
        34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
        51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
        68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
        85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])

darray.dtype
dtype('int64')

darray.nbytes
800
```

Объем используемой памяти для больших наборов данных можно контролировать, явно устанавливая тип данных. Тип `int8` может представлять числа от –128 до 127, поэтому он подойдет для набора данных 1–99. Установить тип данных массива можно с помощью параметра `dtype`. В листинге 7.18 это делается для уменьшения объема данных до 100 байт.

Листинг 7.18. Явная установка типа

```
darray = np.arange(100, dtype=np.int8)
darray
array([  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
        34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
        51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
        68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
        85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99],
        dtype=int8)

darray.nbytes
100
```

ПРИМЕЧАНИЕ

Больше доступных типов данных NumPy можно найти по ссылке https://numpy.org/devdocs/user/basics.types.html.

Поскольку массив может хранить лишь один тип данных, вы не можете вставить данные, которые не могут быть приведены к этому типу. Например, если вы попробуете добавить строку в массив `int8`, то получите сообщение об ошибке:

```
darray[14] = 'a'
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-335-17df5782f85b> in <module>
----> 1 darray[14] = 'a'

ValueError: invalid literal for int() with base 10: 'a'
```

Трудноуловимая ошибка с типом массива возникает, если вы добавляете в сам массив данные с более высокой степенью детализации, чем тип данных массива. Это может привести к потере данных. Допустим, вы добавляете число с плавающей точкой `0.5` в массив `int8`:

```
darray[14] = 0.5
```

Число с плавающей точкой `0.5` приводится к `int`, которое оставляет значение `0`:

```
darray[14]
0
```

Как видите, важно понимать данные, чтобы выбрать лучший тип.

Бродкастинг

Вы можете выполнять операции между массивами с разным количеством измерений, когда эти измерения одинаковы или одна из них равна другой как минимум в одном из массивов. В листинге 7.19 к каждому элементу массива `A1` тремя разными способами добавляется 1: с помощью массива единиц с одинаковыми измерениями (3, 3), с помощью массива с одним измерением, равным единице (1, 3), и с помощью целого числа 1.

Листинг 7.19. Бродкастинг

```
A1 = np.array([[1,2,3],
               [4,5,6],
               [7,8,9]])

A2 = np.array([[1,1,1],
               [1,1,1],
               [1,1,1]])

A1 + A2
array([[ 2, 3, 4],
       [ 5, 6, 7],
       [ 8, 9, 10]])

A2 = np.array([1,1,1])
A1 + A2
array([[ 2, 3, 4],
       [ 5, 6, 7],
       [ 8, 9, 10]])

A1 + 1
array([[ 2, 3, 4],
       [ 5, 6, 7],
       [ 8, 9, 10]])
```

Во всех трех случаях результат одинаковый: массив размерностью (3, 3). Это называется *бродкастинг* — измерение, равное единице, расширяется для соответствия более крупному измерению. Поэтому, если вы выполняете операции (1, 3, 4, 4) и (5, 3, 4, 1), у итогового массива будут измерения (5, 3, 4, 4). Бродкастинг не работает с измерениями, размер которых отличен от единицы.

В листинге 7.20 выполнены операции над массивами с измерениями (2, 1, 5) и (2, 7, 1). Размерность итогового массива — (2, 7, 5).

Листинг 7.20. Расширение измерений

```
A4 = np.arange(10).reshape(2,1,5)
A4
```

```
array([[[0, 1, 2, 3, 4]],
       [[5, 6, 7, 8, 9]]])

A5 = np.arange(14).reshape(2,7,1)
A5
array([[[ 0],
        [ 1],
        [ 2],
        [ 3],
        [ 4],
        [ 5],
        [ 6]],
        [[ 7],
        [ 8],
        [ 9],
        [10],
        [11],
        [12],
        [13]]])

A6 = A4 - A5
A6
array([[[ 0, 1, 2, 3, 4],
        [-1, 0, 1, 2, 3],
        [-2, -1, 0, 1, 2],
        [-3, -2, -1, 0, 1],
        [-4, -3, -2, -1, 0],
        [-5, -4, -3, -2, -1],
        [-6, -5, -4, -3, -2]],

        [[-2, -1, 0, 1, 2],
        [-3, -2, -1, 0, 1],
        [-4, -3, -2, -1, 0],
        [-5, -4, -3, -2, -1],
        [-6, -5, -4, -3, -2],
        [-7, -6, -5, -4, -3],
        [-8, -7, -6, -5, -4]]])
A6.shape
(2, 7, 5)
```

Математические функции NumPy

В дополнение к массивам библиотека NumPy предлагает множество математических функций: тригонометрических, логарифмических и арифметических. Они предназначены для выполнения с массивами NumPy и часто используются вместе с типами данных в других библиотеках. В этом разделе кратко рассмотрены многочлены NumPy.

NumPy предлагает класс `poly1d` для моделирования одномерных полиномов (многочленов). Чтобы использовать `poly1d`, нужно импортировать его из NumPy:

```
[1]       1 from numpy import poly1d
```

Теперь создайте полиномиальный объект, указав коэффициенты в качестве аргумента:

```
poly1d((4,5))
poly1d([4, 5])
```

При выводе объекта `poly1d` он отображает представление полинома:

```
c = poly1d([4,3,2,1])
print(c)
    3     2
4 x + 3 x + 2 x + 1
9780136624356_
```

Если для второго аргумента задано значение `True`, первый интерпретируется как корни, а не как коэффициенты. В следующем примере моделируется полином, полученный в результате вычисления $(x - 4)(x - 3)(x - 2)(x - 1)$:

```
r = poly1d([4,3,2,1], True)
print(r)
    4      3      2
1 x - 10 x + 35 x - 50 x + 24
```

Вычислить полином можно, указав значение коммандом в качестве аргумента для самого объекта. Например, вы можете вычислить предыдущий полином для значения коммандом, равного 5:

```
r(5)
24.0
```

Класс `poly1d` позволяет вам производить такие операции между полиномами, как сложение и умножение. Он предлагает такие полиномиальные функциональные возможности, как специальные методы класса. В листинге 7.21 показано использование этого класса с полиномами.

Листинг 7.21. Полиномы

```
p1 = poly1d((2,3))
print(p1)
2 x + 3
```

```
p2 = poly1d((1,2,3))
print(p2)
    2
1 x + 2 x + 3

print(p2*p1)            # Умножение полиномов
    3     2
2 x + 7 x + 12 x + 9

print(p2.deriv())       # Использование производной
2 x + 2

print(p2.integ())       # Возврат антипроизводных
         3     2
0.3333 x + 1 x + 3 x
```

Класс `poly1d` — это лишь один из многих специализированных математических инструментов в арсенале NumPy. Все они используются вместе с другими специализированными инструментами, о которых вы узнаете в следующих главах.

Резюме

Сторонние библиотеки NumPy — рабочая лошадка для изучения data science в Python. Даже не используя напрямую массивы NumPy, вы столкнетесь с ними, ведь они лежат в основе многих других библиотек. Такие библиотеки, как SciPy и Pandas, строятся именно на массивах NumPy.

Массивы NumPy могут быть созданы во многих измерениях и типах данных. Вы можете настроить их, чтобы регулировать потребление памяти, управляя их типом данных. Эти массивы предназначены для эффективной работы с большими наборами данных.

Вопросы для закрепления

1. Назовите три различия между массивами NumPy и списками Python.
2. Вам дан следующий код. Каким будет конечное значение `d2`?

```
d1 = np.array([[0, 1, 3],
               [4, 2, 9]])
d2 = d1[:, 1:]
```

3. Вам дан следующий код. Каким будет конечное значение `d1[0, 2]`?

```
d1 = np.array([[0, 1, 3],
               [4, 2, 9]])
d2 = d1[:, 1:]
d2[0,1] = 0
```

4. При сложении двух массивов с измерениями (1, 2, 3) и (5, 2, 1) каковы будут измерения итогового массива?

5. Используйте класс `poly1d` для моделирования следующего полинома:

$$6x^4 + 2x^3 + 5x^2 + x - 10$$

8

Библиотека SciPy

Многие используют статистику так же,
как пьяный — фонарные столбы:
для поддержки, а не для освещения.

Эндрю Лэнг

В этой главе

- Математика с NumPy.
- Знакомство с SciPy.
- Подмодуль `scipy.misc`.
- Подмодуль `scipy.special`.
- Подмодуль `scipy.stats`.

В главе 7 мы познакомились с массивами NumPy, на которых основаны многие биб-лиотеки, связанные с data science. Эта глава знакомит с SciPy — библиотекой для математики, науки и инженерии.

Обзор SciPy

Библиотека SciPy — это набор основанных на NumPy пакетов, предоставляющих инструменты для научных расчетов через компьютер. Она включает подмодули, которые занимаются оптимизацией, преобразованиями Фурье, обработкой сигналов и изображений, линейной алгеброй и, среди прочего, статистикой. Эта глава затрагивает три подмодуля: `scipy.misc`, `scipy.special` и `scipy.stats` — наиболее полезный для Data Science.

В некоторых примерах главы используется библиотека `matplotlib`. Она позволяет визуализировать разные типы графиков, диаграмм и изображений. По соглашению для импорта библиотеки построения графиков ее нужно импортировать с именем `plt`:

```
import matplotlib.pyplot as plt
```

Подмодуль scipy.misc

Подмодуль `scipy.misc` содержит функции, которым больше нигде нет места. Одна из забавных функций этого модуля — `scipy.misc.face()`, которая может быть запущена с помощью следующего кода:

```
from scipy import misc
import matplotlib.pyplot as plt
face = misc.face()
plt.imshow(face)
plt.show()
```

Вы можете попробовать это самостоятельно, чтобы сгенерировать вывод.

Функция `ascent` возвращает изображение в градациях серого, доступное для использования и демонстрации. При вызове `ascent()` результатом будет двумерный массив NumPy:

```
a = misc.ascent()
print(a)
 [[ 83  83  83 ... 117 117 117]
  [ 82  82  83 ... 117 117 117]
  [ 80  81  83 ... 117 117 117]
             ...
  [178 178 178 ...  57  59  57]
  [178 178 178 ...  56  57  57]
  [178 178 178 ...  57  57  58]]
```

Если вы передадите этот массив объекту графика `matplotlib`, то увидите изображение, как на рис. 8.1:

```
plt.imshow(a)
plt.show()
```

Рис. 8.1. Демонстрационное изображение из подмодуля scipy.misc

Здесь видно, что метод `plt.imshow()` используется для визуализации изображений.

Подмодуль scipy.special

Подмодуль `scipy.special` содержит утилиты для математической физики: функции Эйри, эллиптические функции, функции Бесселя, Струве и многие другие. Большинство из них поддерживают бродкастинг и совместимы с массивами NumPy. Чтобы использовать эти функции, просто импортируйте `scipy.special` из SciPy и вызовите их напрямую. Например, с помощью функции `special.factorial()` можно вычислить факториал числа:

```
from scipy import special
special.factorial(3)
6.0
```

Рассчитать количество сочетаний и перестановок можно так:

```
special.comb(10, 2)
45.0
special.perm(10,2)
90.0
```

Здесь показаны десять элементов и выбор двух из них за раз.

ПРИМЕЧАНИЕ

У scipy.special есть некоторые низкоуровневые функции из подмодуля scipy.stats, но они не предназначены для прямого использования. scipy.stats применяется в статистических целях. Вы познакомитесь с ним в следующем разделе.

Подмодуль scipy.stats

`scipy.stats` предлагает распределение вероятностей и статистические функции. Ниже рассматриваются несколько распределений, предлагаемых этим подмодулем.

Дискретные распределения

SciPy предлагает несколько дискретных распределений, применяющих общие методы. Они показаны в листинге 8.1, где используется биномиальное распределение — некоторое количество проб, результат каждой из которых либо успешен, либо неудачен.

Листинг 8.1. Биномиальное распределение

```
from scipy import stats
B = stats.binom(20, 0.3) # Определение биномиального распределения,
                         # состоящего из 20 испытаний и 30 % шансов на успех

B.pmf(2) # Вероятностная функция (вероятность того, что выборка равна 2)
0.02784587252426866

B.cdf(4) # Кумулятивная функция распределения (вероятность того, что
         # выборка меньше 4)
0.2375077788776017

B.mean # Среднее значение распределения
6.0

B.var() # Дисперсия распределения
4.199999999999999
```

```
B.std() # Стандартное отклонение распределения
2.0493901531919194

B.rvs() # Получение случайной выборки из распределения
5

B.rvs(15) # Получите 15 случайных выборок
array([ 2, 8, 6, 3, 5, 5, 10, 7, 5, 10, 5, 5, 5, 2, 6])
```

Если вы возьмете достаточно большую случайную выборку распределения

```
rvs = B.rvs(size=100000)
rvs
array([11, 4, 4, ..., 7, 6, 8])
```

то можете использовать `matplotlib` для построения графика и получения представления о его форме (см. рис. 8.2):

```
import matplotlib.pyplot as plt
plt.hist(rvs)
plt.show()
```

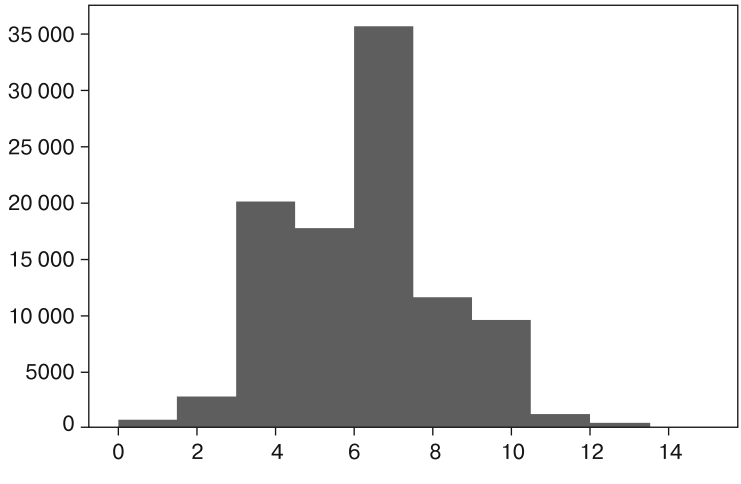


Рис. 8.2. Биномиальное распределение

Числа в нижней части распределения на рис. 8.2 — это количество успешных операций в 20-пробном эксперименте. Вы можете видеть, что 6 из 20 — наиболее распространенный результат, соответствующий 30 % вероятности успеха.

Другое распределение в подмодуле `scipy.stats` — распределение Пуассона. Оно моделирует вероятность определенного числа отдельных событий за некоторый промежуток времени. Форма распределения управляется его средним значением, которое вы можете установить с помощью ключевого слова `mu`. Например, более низкое значение, такое как 3, сдвинет распределение влево (рис. 8.3):

```
P = stats.poisson(mu=3)
rvs = P.rvs(size=10000)
rvs
array([4, 4, 2, ..., 1, 0, 2])

plt.hist(rvs)
plt.show()
```

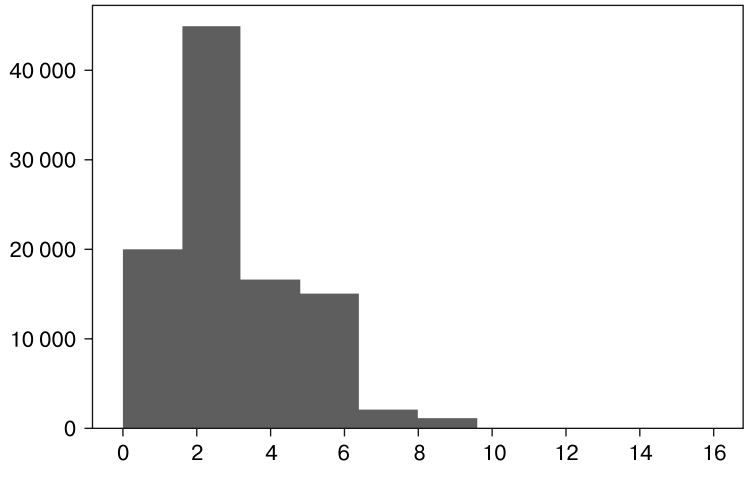


Рис. 8.3. Распределение Пуассона со сдвигом влево

Более высокое среднее значение, такое как 15, передвинет распределение вправо (см. рис. 8.4):

```
P = stats.poisson(mu=15)
rvs = P.rvs(size=100000)
plt.hist(rvs)
plt.show()
```

Другие дискретные распределения в подмодуле `scipy.stats`: бета-биномиальное распределение, распределение Больцмана (усеченное Планка), распределение Планка (дискретный экспоненциал), геометрическое, гипергеометрическое, логарифмическое и распределение Юла-Саймона. На момент написания этой книги есть всего 14 распределений, моделируемых в подмодуле `scipy.stats`.

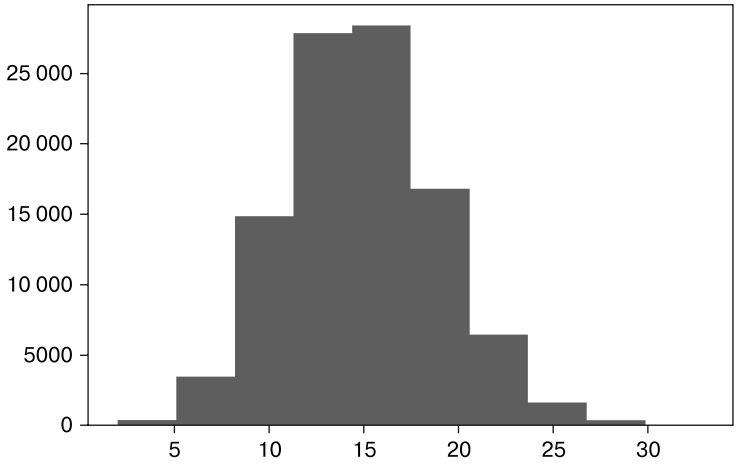


Рис. 8.4. Распределение Пуассона со сдвигом вправо

Непрерывные распределения

Подмодуль `scipy.stats` включает гораздо больше непрерывных распределений, чем дискретных. На момент написания книги их было 87. Все эти распределения принимают аргументы для местоположения (`loc`) и масштаба (`scale`). Все по умолчанию имеют местоположение `0` и масштаб `1.0`.

К непрерывным распределениям относится нормальное, которое вы можете знать как колокол, или гауссову кривую. В этом симметричном распределении одна половина данных находится слева от среднего значения, а другая — справа.

Вот так можно создать нормальное распределение с помощью местоположения и масштаба по умолчанию:

```
N = stats. norm()
rvs = N.rvs(size=100000)
plt.hist(rvs, bins=1000)
plt.show()
```

На рис. 8.5 показан график этого распределения.

Вы можете видеть, что распределение сосредоточено на 0 и находится примерно между –4 и 4. Рисунок 8.6 показывает результат создания второго нормального распределения — на этот раз установлено значение местоположения 30 и масштаб 50:

```
N = stats. norm()
rvs = N.rvs(size=100000)
plt.hist(rvs, bins=1000)
plt.show()
```

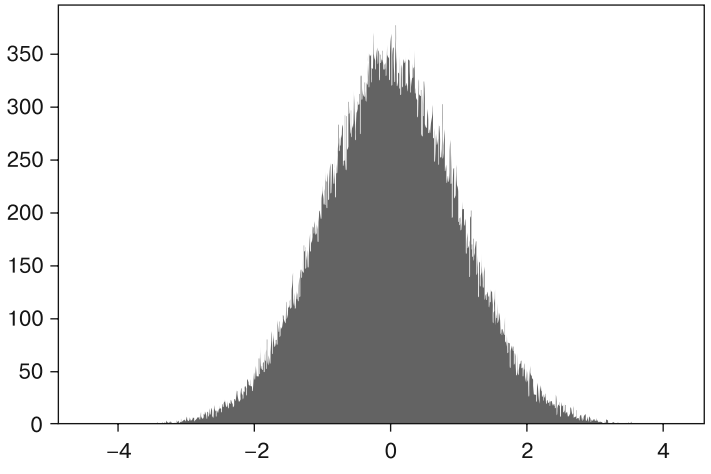


Рис. 8.5. Гауссова кривая

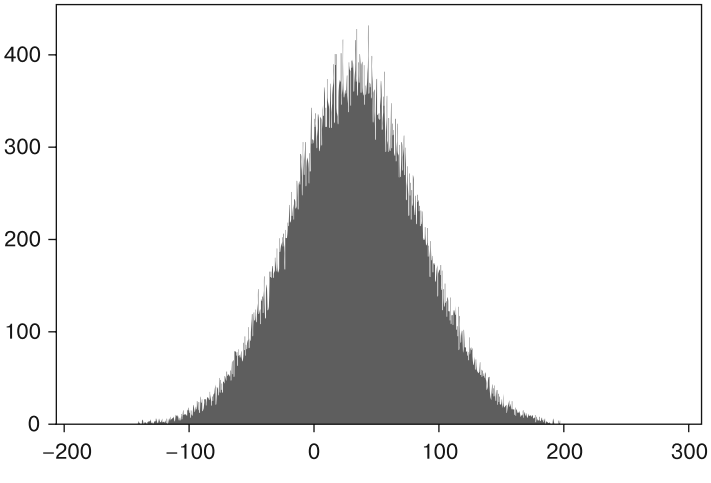


Рис. 8.6. Смещение гауссовой кривой

Обратите внимание, что распределение теперь сосредоточено вокруг 30 и охватывает куда более широкий диапазон чисел. У непрерывных распределений есть

несколько общих функций, которые смоделированы в листинге 8.2. Заметьте, что здесь использовано второе нормальное распределение со смещенным местоположением и большим стандартным отклонением.

Листинг 8.2. Нормальное распределение

```
N1 = stats.norm(loc=30, scale=50)
N1.mean() # Среднее значение распределения, которое соответствует значению loc
30.0

N1.pdf(4) # Функция плотности вероятности
0.006969850255179491

N1.cdf(2) # Кумулятивная функция распределения
0.28773971884902705

N1.rvs() # Случайная выборка
171.55168607574785

N1.var() # Дисперсия
2500.0

N1.median() # Медиана
30.0

N1.std() # Стандартное отклонение
50.0
```

ПРИМЕЧАНИЕ

Если вы попробуете воспроизвести показанные здесь примеры, некоторые из ваших значений могут отличаться из-за генерации случайных чисел.

Следующее — экспоненциально изменяющееся непрерывное распределение — характеризуется экспоненциально изменяющейся кривой — либо вверх, либо вниз (рис. 8.7):

```
E = stats.expon()
rvs = E.rvs(size=100000)
plt.hist(rvs, bins=1000)
plt.show()
```

Рисунок 8.7 отображает кривую, которую и следовало ожидать от экспоненциальной функции. Ниже приведено равномерное распределение с постоянной вероятностью, известное как прямоугольное распределение:

```
U = stats.uniform()
rvs = U.rvs(size=10000)
rvs
```

```
array([8.24645026e-01, 5.02358065e-01, 4.95390940e-01, ...,
       8.63031657e-01, 1.05270200e-04, 1.03627699e-01])

plt.hist(rvs, bins=1000)
plt.show()
```

Оно дает равномерно распределенную вероятность в заданном диапазоне, график которой показан на рис. 8.8.

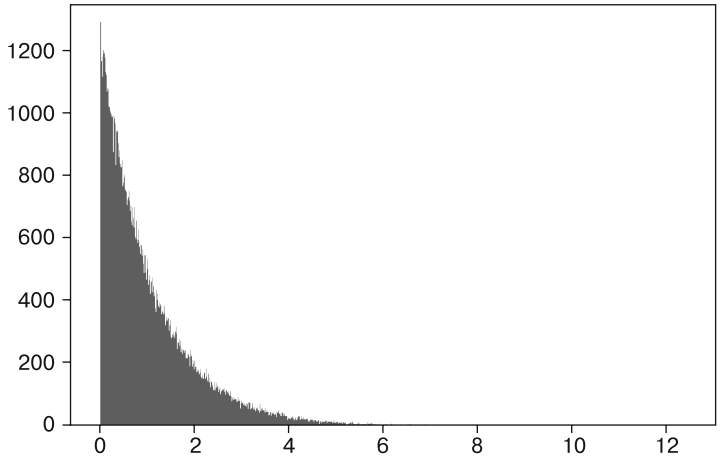


Рис. 8.7. Экспоненциально изменяющееся распределение

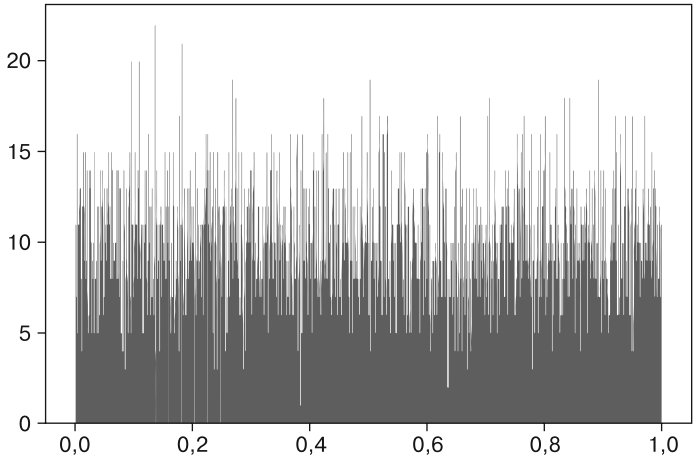


Рис. 8.8. Равномерное распределение

Резюме

Обе библиотеки, NumPy и SciPy, предлагают утилиты для решения сложных математических задач. Они охватывают множество ресурсов, а их применению посвящены целые книги. Вы познакомились лишь с некоторыми из множества их возможностей. Эти библиотеки — первое, на что вам стоит обратить внимание, когда вы приступаете к решению или моделированию сложных математических задач.

Вопросы для закрепления

1. Используйте подмодуль `scipy.stats` для моделирования нормального распределения со средним значением 15.
2. Сгенерируйте 25 случайных выборок из распределения, смоделированного в вопросе 1.
3. У какого подмодуля SciPy есть утилиты, разработанные для математической физики?
4. Какой метод предоставляется дискретным распределением для вычисления его стандартного отклонения?

9

Библиотека Pandas

Чтобы уточнить, *добавьте* данные.

Эдвард Рольф Тафти

В этой главе

- Знакомство с датафреймами Pandas.
- Создание датафреймов.
- Интроспекция датафреймов.
- Получение доступа к данным.
- Управление датафреймами.
- Управление данными датафреймов.

Датафреймы Pandas (структуры данных DataFrame), построенные поверх массива NumPy, — наиболее часто используемая структура данных. Они похожи на высокопроизводительные электронные таблицы в коде. Это один из основных

инструментов в data science. В этой главе рассматривается создание датафреймов и управление ими, получение доступа к данным в них и управление этими данными.

Структура датафреймов

Датафреймы библиотеки Pandas, как и электронные таблицы, состоят из столбцов и строк. Каждый столбец — это объект `pandas.Series`. В каком-то смысле датафрейм похож на двумерный массив NumPy с метками для столбцов и индексов. Но, в отличие от него, датафрейм может содержать разные типы данных.

Объект `pandas.Series` можно представить как одномерный массив NumPy с метками. Он, как и массив NumPy, может содержать лишь один тип данных. Объект `pandas.Series` может использовать многие из методов, которые вы видели у массивов: `min()`, `max()`, `mean()` и `medium()`.

Как правило, пакет Pandas импортируют под псевдонимом `pd`:

```
import pandas as pd
```

Создание датафреймов

Создать датафрейм с данными из многих источников, включая словари и списки, можно путем считывания файлов. Сформируйте пустой датафрейм с помощью конструктора `DataFrame`:

```
df = pd.DataFrame()
print(df)
Empty DataFrame
Columns: []
Index: []
```

Но лучше инициализировать датафрейм уже с данными.

Создание датафреймов из словаря

Вы можете создавать датафрейм из одного словаря или списка словарей, где каждый ключ — это метка столбца со значениями для этого ключа, удерживающего данные столбца. В листинге 9.1 показано, как создать датафрейм, сгенерировав список данных для каждого столбца, а после — словарь с именами столбцов в качестве

ключей и этими списками в качестве значений. После этого нужно передать словарь конструктору `DataFrame`.

Листинг 9.1. Создание датафрейма из словаря

```
first_names = ['shanda', 'rolly', 'molly', 'frank',
               'rip', 'steven', 'gwen', 'arthur']

last_names = ['smith', 'brocker', 'stein', 'bach',
              'spencer', 'de wilde', 'mason', 'davis']

ages = [43, 23, 78, 56, 26, 14, 46, 92]
data = {'first':first_names,
        'last':last_names,
        'ages':ages}

participants = pd.DataFrame(data)
```

Итоговый датафрейм `participants` выглядит в блокноте Colab или Jupyter так:

	first	last	ages
0	shanda	smith	43
1	rolly	brocker	23
2	molly	stein	78
3	frank	bach	56
4	rip	spencer	26
5	steven	de wilde	14
6	gwen	mason	46
7	arthur	davis	92

ПРИМЕЧАНИЕ

В этой главе датафреймы, полученные из примера кода, будут представлены в виде таблицы после кода.

Вы можете видеть метки столбцов вверху, метки индексов — слева, а данные — в каждой строке.

Создание датафреймов из списка списков

Вы можете создать список списков, каждый подсписок которого будет содержать данные для одной строки в порядке следования столбцов:

```
data = [["shanda", "smith", 43],
        ["rolly", "brocker", 23],
        ["molly", "stein", 78],
        ["frank", "bach", 56],
        ["rip", "spencer", 26],
        ["steven", "de wilde", 14],
        ["gwen", "mason", 46],
        ["arthur", "davis", 92]]
```

Затем вы можете использовать это в качестве аргумента данных:

```
participants = pd.DataFrame(data)
participants
```

Полученные данные будут те же, что и при создании датафрейма из словаря:

	0	1	2
0	shanda	smith	43
1	rolly	brocker	23
2	molly	stein	78
3	frank	bach	56
4	rip	spencer	26
5	steven	de wilde	14
6	gwen	mason	46
7	arthur	davis	92

Обратите внимание, что итоговый датафрейм был создан с целочисленными именами столбцов. Это происходит по умолчанию, если имена столбцов не указаны.

Вы можете явно указать имена столбцов в виде списка строк:

```
column_names = ['first', 'last', 'ages']
```

Точно так же можно указать метки индексов в виде списка:

```
index_labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Далее метки применяются во время инициализации с помощью параметров `column` и `index`:

```
participants = pd.DataFrame(data,
                            columns=column_names,
                            index=index_labels)
```

	first	last	ages
a	shanda	smith	43
b	rolly	brocker	23
c	molly	stein	78
d	frank	bach	56
e	rip	spencer	26
f	steven	de wilde	14
g	gwen	mason	46
h	arthur	davis	92

Создание датафрейма из файла

Несмотря на то что датафрейм можно формировать из словарей и списков, большую часть времени вы будете создавать их из существующих источников данных. Файлы — самые распространенные среди них. Pandas поддерживает функции для создания датафреймов из многих распространенных типов файлов, включая CSV, Excel, HTML, JSON и подключения к базе данных SQL.

Допустим, вы хотите открыть файл формата CSV с сайта FiveThirtyEight, https://data.fivethirtyeight.com, в наборе данных `college_majors`. После распаковки и загрузки файла в блокнот Colab вы сможете его открыть, просто указав его путь в функции Pandas `read_csv`:

```
college_majors = pd.read_csv('/content/all-ages.csv')
college_majors
```

	Major	Major_category	Total	Unemployment_rate
0	GENERAL AGRICULTURE	Agriculture & Natural Resources	128148	0.026147
1	AGRICULTURE PRODUCTION AND MANAGEMENT	Agriculture & Natural Resources	95326	0.028636
2	AGRICULTURAL ECONOMICS	Agriculture & Natural Resources	33955	0.030248
...	...	...	...	...
170	MISCELLANEOUS BUSINESS & MEDICAL ADMINISTRATION	Business	102753	0.052679
171	HISTORY	Humanities & Liberal Arts	712509	0.065851
172	UNITED STATES HISTORY	Humanities & Liberal Arts	17746	0.073500

Pandas использует данные в CSV-файле для определения меток и типов столбцов.

Взаимодействие с данными датафреймов

Как только вы загрузили данные в датафрейм, взгляните на них. Pandas предлагает множество способов получения к ним доступа. Вы можете просматривать данные по строкам, столбцам, отдельным ячейкам или сочетая эти способы. Также можно извлекать данные на основе их значений.

ПРИМЕЧАНИЕ

После загрузки незнакомых данных я начинаю с просмотра нескольких верхних строк и проверки сводной статистики. Просмотр верхних строк в датафрейме дает мне представление о новых данных и позволяет убедиться, что они соответствуют моим ожиданиям.

Головы и хвосты

Чтобы увидеть верхние строки датафрейма, используйте метод `head`, который возвращает верхние пять строк:

```
college_majors.head()
```

	Major	Major_category	Total	Unemployment_rate
0	GENERAL AGRICULTURE	Agriculture & Natural Resources	128148	0.026147
1	AGRICULTURE PRODUCTION AND MANAGEMENT	Agriculture & Natural Resources	95326	0.028636
2	AGRICULTURAL ECONOMICS	Agriculture & Natural Resources	33955	0.030248
3	ANIMAL SCIENCES	Agriculture & Natural Resources	103549	0.042679
4	FOOD SCIENCE	Agriculture & Natural Resources	24280	0.049188

Метод `head` принимает необязательный аргумент, указывающий число возвращаемых строк. Три верхние строки можно указать так:

```
college_majors.head(3)
```

	Major	Major_category	Total	Unemployment_rate
0	GENERAL AGRICULTURE	Agriculture & Natural Resources	128148	0.026147
1	AGRICULTURE PRODUCTION AND MANAGEMENT	Agriculture & Natural Resources	95326	0.028636
2	AGRICULTURAL ECONOMICS	Agriculture & Natural Resources	33955	0.030248

Метод `tail` работает аналогично `head`, но возвращает строки из конца таблицы. Он тоже принимает необязательный аргумент, указывающий число возвращаемых строк:

```
college_majors.tail()
```

	Major	Major_category	Total	Unemployment_rate
168	HOSPITALITY MANAGEMENT	Business	200854	0.051447
169	MANAGEMENT INFORMATION SYSTEMS AND STATISTICS	Business	156673	0.043977
170	MISCELLANEOUS BUSINESS & MEDICAL ADMINISTRATION	Business	102753	0.052679
171	HISTORY	Humanities & Liberal Arts	712509	0.065851
172	UNITED STATES HISTORY	Humanities & Liberal Arts	17746	0.073500

Описательная статистика

После просмотра некоторых строк датафрейма я хочу получить представление о форме данных. В этом может помочь метод `describe`. Он создает разную описательную статистику о данных. Вы можете вызвать `describe` без аргументов:

```
college_majors.describe()
```

	Total	Unemployment_rate
count	1.730000e+02	173.000000
mean	2.302566e+05	0.057355
std	4.220685e+05	0.019177
min	2.396000e+03	0.000000
25%	2.428000e+04	0.046261
50%	7.579100e+04	0.054719
75%	2.057630e+05	0.069043
max	3.123510e+06	0.156147

Этот метод вычисляет количество, среднее и стандартное отклонение, минимум, максимум и квантили для столбцов с числовыми данными. Он принимает необязательный аргумент для управления обрабатываемыми типами данных и диапазонами всех возвращаемых квантилей. Чтобы изменить квантили, используйте аргумент `percentiles`:

```
college_majors.describe(percentiles=[0.1, 0.9])
```

	Total	**Unemployment_rate**
count	1.730000e+02	173.000000
mean	2.302566e+05	0.057355
std	4.220685e+05	0.019177
min	2.396000e+03	0.000000
10%	9.775600e+03	0.037053
50%	7.579100e+04	0.054719
90%	6.739758e+05	0.080062
max	3.123510e+06	0.156147

В этом примере указаны процентили для 10 и 90 % вместо установленных по умолчанию 25 и 75 %. Обратите внимание, что 50 % вставляется вне зависимости от аргумента.

Чтобы просмотреть статистику, рассчитанную из нечисловых столбцов, укажите типы данных для обработки. Это можно сделать с помощью ключевого слова `include`. Передаваемое этому слову значение должно быть последовательностью типа данных, которые могут быть типом данных NumPy, таким как `np.object`.

В Pandas строки представляют тип `object`, поэтому ниже приведены столбцы со строковыми типами данных:

```
import numpy as np college_majors.describe(include=[np.object])
```

Так вы сможете найти и строковое имя типа данных, которое в случае с `np.object` будет `object`. Следующий код возвращает соответствующую типу статистику:

```
college_majors.describe(include=['object'])
```

Итак, для строк вы получаете количество, число уникальных значений, верхнее значение и его частоту:

	Major	**Major_category**
count	173	173
unique	173	16
top	GEOSCIENCES	Engineering
freq	1	29

Чтобы получить статистику для всех столбцов, передайте строку `all` вместо списка типов данных:

```
college_majors.describe(include='all')
```

	Major	**Major_category**	**Total**	**Unemployment_ rate**
count	173	173	1.730000e+02	173.000000
unique	173	16	NaN	NaN
top	GEOSCIENCES	Engineering	NaN	NaN
freq	1	29	NaN	NaN
mean	NaN	NaN	2.302566e+05	0.057355
std	NaN	NaN	4.220685e+05	0.019177
min	NaN	NaN	2.396000e+03	0.000000
25%	NaN	NaN	2.428000e+04	0.046261
50%	NaN	NaN	7.579100e+04	0.054719
75%	NaN	NaN	2.057630e+05	0.069043
max	NaN	NaN	3.123510e+06	0.156147

ПРИМЕЧАНИЕ

Там, где применение статистики не подходит для типа данных, например в случае стандартного отклонения для строки, вставляется нечисленное значение NAN.

Чтобы исключить определенные типы данных, а не указать, какие из них включить, Pandas предоставляет аргумент `exclude`, принимающий те же типы аргументов, что и `include`:

```
college_majors.describe(exclude=['int'])
```

Доступ к данным

После первичного просмотра фрейма с помощью `head` и `tail` и получения представления о форме данных с помощью `describe` можно начать просматривать данные и отдельные столбцы, строки или ячейки.

Ниже представлен рассмотренный ранее датафрейм `participants`:

```
participants
```

	first	last	ages
a	shanda	smith	43
b	rolly	brocker	23
c	molly	stein	78
d	frank	bach	56
e	rip	spencer	26
f	steven	de wilde	14
g	gwen	mason	46
h	arthur	davis	92

Синтаксис со скобками

Для получения доступа к столбцам и строкам датафрейма Pandas используйте синтаксис со скобками. Он отлично подходит для интерактивных сеансов, когда вы изучаете данные и экспериментируете с ними.

Чтобы получить доступ к отдельному столбцу, укажите имя столбца в качестве аргумента в скобках, как если вы бы использовали ключ словаря:

```
participants['first']
a    shanda
b     rolly
```

```
c     molly
d     frank
e       rip
f    steven
g      gwen
h    arthur
Name: first, dtype: object
```

Это возвращает данные из столбца вместе с индексом, меткой и типом данных.

Если столбец не совпадает с существующим атрибутом датафрейма и в его имени нет длинного тире или специальных символов, вы можете получить доступ к столбцу как к атрибуту.

Например, так вы получаете доступ к столбцу `ages`:

```
participants.ages
a    43
b    23
c    78
d    56
e    26
f    14
g    46
h    92
Name: ages, dtype: int64
```

Это не будет работать для столбцов `first` и `last`, так как они уже существуют как атрибуты датафрейма.

Чтобы получить доступ к множеству столбцов, укажите метку столбца как список:

```
participants[['last', 'first']]
```

	last	**first**
a	smith	shanda
b	brocker	rolly
c	stein	molly
d	bach	frank
e	spencer	rip
f	de wilde	steven
g	mason	gwen

Команда возвращает датафрейм только с запрошенными столбцами.

Синтаксис скобок перегружен, чтобы вы смогли захватить как строки, так и столбцы. Для указания строк используйте срез в качестве аргумента. Если он использует целые числа, то они представляют возвращаемые номера строк.

Например, для возврата строк 3, 4 и 5 из датафрейма `participants` вы можете использовать срез `3:6`:

```
participants[3:6]
```

	first	last	ages
d	frank	bach	56
e	rip	spencer	26
f	steven	de wilde	14

Срез можно производить и с помощью меток индексов. При их использовании будет включено последнее значение. Поэтому, чтобы получить строки `a`, `b` и `c`, выполните срез `a:c`:

```
participants['a':'c']
```

	first	last	ages
a	shanda	smith	43
b	rolly	brocker	23
c	molly	stein	78

С помощью булева списка вы можете указать возвращаемые строки. В списке должно быть одно булево значение для строки: `True` для нужных строк и `False` для остальных. В следующем примере возвращаются вторая, третья и шестая строки:

```
mask = [False, True, True, False, False, True, False, False]
participants[mask]
```

	first	last	ages
b	rolly	brocker	23
c	molly	stein	78
f	steven	de wilde	14

Синтаксис скобок обеспечивает очень удобный и легко читабельный доступ к данным. Он часто применяется в интерактивных сессиях при исследовании датафреймов и экспериментах с ними, но не оптимизирован для работы с большими наборами данных. Рекомендованный способ индексирования датафрейма в продакшене или для больших наборов данных — использование индексаторов `loc` и `iloc`. Они используют синтаксис скобок, похожий на то, что вы видели здесь. `loc` индексирует с помощью меток, а `iloc` использует позиции индекса.

Оптимизированный доступ по метке

С помощью индексатора `loc` можно указать отдельную метку строки, для которой будут возвращены значения. Чтобы получить значения из строки с меткой `c`, можно просто указать `c` в качестве аргумента:

```
participants.loc['c']
first    molly
last     stein
ages        78
Name: c, dtype: object
```

Можете указать часть меток срезом (опять же — включая последнюю):

```
participants.loc['c':'f']
```

	first	last	ages
c	molly	stein	78
d	frank	bach	56
e	rip	spencer	26
f	steven	de wilde	14

Или можно указать булеву последовательность:

```
mask = [False, True, True, False, False, True, False, False]
participants.loc[mask]
```

	first	last	ages
b	rolly	brocker	23
c	molly	stein	78
f	steven	de wilde	14

Второй необязательный аргумент может указывать, какой столбец возвращать. Например, чтобы вернуть все строки в столбце `first`, укажите все строки с помощью среза, запятой и метки столбца:

```
participants.loc[:, 'first']
a    shanda
b     rolly
c     molly
d     frank
e       rip
f    steven
g      gwen
h    arthur
Name: first, dtype: object
```

Можно указать список меток столбцов:

```
participants.loc[:'c', ['ages', 'last']]
```

	ages	last
a	43	smith
b	23	brocker
c	78	stein

Или предоставить булев список:

```
participants.loc[:'c', [False, True, True]]
```

	last	ages
a	smith	43
b	brocker	23
c	stein	78

Оптимизированный доступ по индексу

Индексатор `iloc` позволяет использовать позиции индекса для выбора строк и столбцов. Как и в примере со скобками, вы можете использовать только одно значение для указания одной строки:

```
participants.iloc[3]
first    frank
last      bach
ages        56
Name: d, dtype: object
```

Или можно указать множество строк с помощью среза:

```
participants.iloc[1:4]
```

	first	last	ages
b	rolly	brocker	23
c	molly	stein	78
d	frank	bach	56

Используя второй срез, можно указать, какой столбец возвращать:

```
participants.iloc[1:4, :2]
```

	first	last
b	rolly	brocker
c	molly	stein
d	frank	bach

Маски и фильтрация

Полезная особенность датафрейма — способность выбирать данные на основе значений. Вы можете использовать операторы сравнения со столбцами, чтобы увидеть, какие значения удовлетворяют определенным условиям. Например, чтобы увидеть, у какой из строк датафрейма `college_majors` есть значение `Humanities & Liberal Arts` в качестве основной категории, можно использовать оператор равенства (`==`):

```
college_majors.Major_category == 'Humanities & Liberal Arts'
0      False
1      False
2      False
3      False
       ...
169    False
170    False
171    True
172    True
Name: Major_category, Length: 173, dtype: bool
```

Здесь создается объект `pandas.Series` со значением `True` для каждой строки, удовлетворяющей условию. Конечно, можно воспользоваться серией булевых значений, но настоящую мощь вы ощутите при совмещении ее с индексатором для фильтрации результатов.

Помните, что индексатор `loc` возвращает строки для каждого значения `True` входной последовательности. Вы можете создать условие на основе оператора сравнения и строки, как показано ниже для оператора `>` и строки `Total`:

```
total_mask = college_majors.loc[:, 'Total'] > 1200000
```

Можете использовать результат как маску, чтобы выбрать только удовлетворяющие этим условиям строки:

```
top_majors = college_majors.loc[total_mask]
top_majors
```

	Major	Major_category	Total	Unemployment_rate
25	GENERAL EDUCATION	Education	1438867	0.043904
28	ELEMENTARY EDUCATION	Education	1446701	0.038359

	Major	Major_category	Total	Unemployment_rate
114	PSYCHOLOGY	Psychology & Social Work	1484075	0.069667
153	NURSING	Health	1769892	0.026797
158	GENERAL BUSINESS	Business	2148712	0.051378
159	ACCOUNTING	Business	1779219	0.053415
161	BUSINESS MANAGEMENT AND ADMINISTRATION	Business	3123510	0.058865

Используйте метод `min()`, чтобы проверить, удовлетворяет ли итоговый датафрейм условиям:

```
top_majors.Total.min()
1438867
```

Теперь, допустим, вы хотите узнать, у какой из основных категорий самые низкие показатели безработицы (unemployment rate). Здесь можно применить метод `describe` как для одного столбца, так и для всего датафрейма. Если вы используете `describe` на столбце `Unemployment_rate`, то можете увидеть, что верхний показатель для нижнего процентиля равен 0,046261:

```
college_majors.Unemployment_rate.describe()
count    173.000000
mean       0.057355
std        0.019177
min        0.000000
25%        0.046261
50%        0.054719
75%        0.069043
max        0.156147
Name: Unemployment_rate, dtype: float64
```

Можете создать маску для всех строк с показателями безработицы, меньше или равными этому:

```
employ_rate_mask = college_majors.loc[:, 'Unemployment_rate'] <= 0.046261
```

Можно использовать эту маску для создания датафрейма, содержащего только эти строки:

```
employ_rate_majors = college_majors.loc[employ_rate_mask]
```

Чтобы увидеть, какая из основных категорий есть в итоговом датафрейме, можно использовать уникальный метод объекта `pandas.Series`:

```
employ_rate_majors.Major_category.unique()
array(['Agriculture & Natural Resources', 'Education', 'Engineering',
       'Biology & Life Science', 'Computers & Mathematics',
       'Humanities & Liberal Arts', 'Physical Sciences', 'Health',
       'Business'], dtype=object)
```

У всех этих категорий есть как минимум одна строка с уровнем занятости, удовлетворяющим условиям.

Булевы операторы библиотеки Pandas

Вы можете использовать булевы операторы И (`&`), ИЛИ (`|`) и НЕ (`~`) с результатами ваших условий. Операторы `&` или `|` применяются для объединения и создания более сложных условий, а `~` — для создания маски, противоположной вашему условию.

Оператор AND можно использовать, например, для создания новой маски на основе предыдущих, чтобы увидеть, у каких основных категорий наиболее популярных специализаций самый низкий уровень безработицы. Для этого добавьте оператор `&` между существующими масками с целью создания новой:

```
total_rate_mask = employ_rate_mask & total_mask
total_rate_mask
0      False
1      False
2      False
3      False
4      False
      ...
168    False
169    False
170    False
171    False
172    False
Length: 173, dtype: bool
```

Взглянув на финальный датафрейм, можно увидеть, у каких самых популярных специализаций наименьший уровень безработицы:

```
college_majors.loc[total_rate_mask]
```

	Major	Major_category	Total	Unemployment_rate
25	GENERAL EDUCATION	Education	1438867	0.043904
28	ELEMENTARY EDUCATION	Education	1446701	0.038359
153	NURSING	Health	1769892	0.026797

Можете использовать оператор ~ с маской уровня занятости для создания датафрейма, у всех строк которого уровень занятости будет выше нижнего процентиля:

```
lower_rate_mask = ~employ_rate_mask
lower_rate_majors = college_majors.loc[lower_rate_mask]
```

Эту работу можно проверить с помощью метода `min` на столбце `Unemployment_rate`, чтобы увидеть, что он находится выше верхнего уровня нижнего процентиля:

```
lower_rate_majors.Unemployment_rate.min()
0.046261360999999994
```

Чтобы выбрать все строки, удовлетворяющие условию либо популярных специализаций, либо уровня занятости, используйте оператор |:

```
college_majors.loc[total_mask | employ_rate_mask]
```

В итоговом датафрейме есть все строки, соответствующие одному из двух условий.

Управление датафреймами

После получения необходимых данных в датафрейме вы можете захотеть изменить его. Вы можете переименовать столбцы или индексы, добавить новые столбцы и строки или удалить их.

Изменить метку столбца легко с помощью метода `rename`. Вот так вы можете использовать атрибут столбцов датафрейма, чтобы просмотреть текущие имена столбцов:

```
participants.columns
Index(['first', 'last', 'ages'], dtype='object')
```

Далее можете переименовать выбранные столбцы, предоставив словарь, сопоставляющий каждое старое имя с новым. Так, можно изменить метку столбца `ages` на `Age`:

```
participants.rename(columns={'ages': 'Age'})
```

	first	last	Age
a	shanda	smith	43
b	rolly	brocker	23
c	molly	stein	78
d	frank	bach	56
e	rip	spencer	26
f	steven	de wilde	14
g	gwen	mason	46
h	arthur	davis	92

Метод `rename` по умолчанию возвращает новый датафрейм, используя новые метки столбцов. Если вы снова проверите имена столбцов изначального датафрейма, то увидите старое имя столбца:

```
participants.columns
Index(['first', 'last', 'ages'], dtype='object')
```

Так работает большинство методов `DataFrame` (сохраняя исходное состояние). Многие из них предлагают необязательный аргумент `inplace`, который, если ему присвоено значение `True`, изменяет исходный датафрейм:

```
participants.rename(columns={'ages':'Age'}, inplace=True)
participants.columns
Index(['first', 'last', 'Age'], dtype='object')
```

Для создания нового столбца используйте синтаксис индексаторов. Нужно получить доступ к столбцу, как если бы он уже существовал, используя индексатор и приведенные значения:

```
participants['Zip Code'] = [94702, 97402, 94223, 94705,
                            97503, 94705, 94111, 95333]
participants
```

	first	last	Age	Zip Code
a	shanda	smith	43	94702
b	rolly	brocker	23	97402
c	molly	stein	78	94223
d	frank	bach	99	94705
e	rip	spencer	26	97503
f	steven	de wilde	14	94705
g	gwen	mason	46	94111
h	arthur	davis	92	95333

Для создания значения в новом столбце используйте такие операции между столбцами, как добавление строки. Чтобы добавить столбец с полными именами участников (participants), создайте значения с помощью существующих столбцов для их имен и фамилий:

```
participants['Full Name'] = ( participants.loc[:, 'first'] +
                              participants.loc[:, 'last'] )
participants
```

	first	last	Age	Zip Code	Full Name
a	shanda	smith	43	94702	shandasmith
b	rolly	brocker	23	97402	rollybrocker
c	molly	stein	78	94223	mollystein
d	frank	bach	99	94705	frankbach
e	rip	spencer	26	97503	ripspencer
f	steven	de wilde	14	94705	stevende wilde
g	gwen	mason	46	94111	gwenmason
h	arthur	davis	92	95333	arthurdavis

Обновить столбец можно с помощью того же синтаксиса. Например, если вы решили, что между значениями имени и фамилии в столбце полных имен должен быть пробел, можете присвоить новые значения, используя то же имя столбца:

```
participants['Full Name'] = ( participants.loc[:, 'first'] +
                              ' ' +
                              participants.loc[:, 'last'] )
participants
```

	first	last	Age	Zip Code	Full Name
a	shanda	smith	43	94702	shanda smith
b	rolly	brocker	23	97402	rolly brocker
c	molly	stein	78	94223	molly stein
d	frank	bach	99	94705	frank bach
e	rip	spencer	26	97503	rip spencer
f	steven	de wilde	14	94705	steven de wilde
g	gwen	mason	46	94111	gwen mason
h	arthur	davis	92	95333	arthur davis

Управление данными

Pandas позволяет вам изменять данные в датафрейме несколькими способами. Вы можете устанавливать значения с помощью тех же индексаторов, что использовали ранее. У вас есть возможность выполнять операции над всем датафреймом или его отдельными столбцами. Кроме того, вы можете применять функции для изменения элементов в столбце или создания новых значений для множественных строк или столбцов.

Чтобы изменить данные с помощью индексатора, выберите местоположение, куда хотите разместить новые данные, так же, как выбирали данные для просмотра, а затем присвойте новое значение.

Чтобы изменить `arthur` в столбце `h` на `Paul`, используйте индексатор `loc`:

```
participants.loc['h', 'first'] = 'Paul'
participants
```

	first	last	Age	Zip Code	Full Name
a	shanda	smith	43	94702	shanda smith
b	rolly	brocker	23	97402	rolly brocker
c	molly	stein	78	94223	molly stein
d	frank	bach	99	94705	frank bach
e	rip	spencer	26	97503	rip spencer
f	steven	de wilde	14	94705	steven de wilde
g	gwen	mason	46	94111	gwen mason
h	paul	davis	92	95333	arthur davis

В качестве альтернативы можно использовать `iloc`, чтобы установить в строке значение возраста Molly равным `78`:

```
participants.iloc[3, 2] = 78
participants
```

	first	last	Age	Zip Code	Full Name
a	shanda	smith	43	94702	shanda smith
b	rolly	brocker	23	97402	rolly brocker
c	molly	stein	78	94223	molly stein
d	frank	bach	99	94705	frank bach
e	rip	spencer	26	97503	rip spencer
f	steven	de wilde	14	94705	steven de wilde
g	gwen	mason	46	94111	gwen mason
h	paul	davis	92	95333	arthur davis

Этот процесс может показаться интуитивным, если вы рассматриваете результат как разновидность индексированного присваивания, которое вы использовали со списками и словарями.

Ранее в этой главе вы использовали операции между столбцами, чтобы создать значения для нового столбца. Для изменения этих значений можно использовать замещающие операторы, такие как `+=` и `/=`.

Например, чтобы вычесть `1` из возраста каждого участника, используйте оператор `-=`:

```
participants.Age -= 1
participants
```

	first	last	Age	Zip Code	Full Name
a	shanda	smith	42	94702	shanda smith
b	rolly	brocker	22	97402	rolly brocker
c	molly	stein	77	94223	molly stein
d	frank	bach	98	94705	frank bach
e	rip	spencer	25	97503	rip spencer
f	steven	de wilde	13	94705	steven de wilde
g	gwen	mason	45	94111	gwen mason
h	paul	davis	91	95333	arthur davis

Метод replace

Метод `replace` находит значения в датафрейме и замещает их. Его можно использовать, например, чтобы заменить имя `rolly` на `Smiley`:

```
participants.replace('rolly', 'Smiley')
```

	first	last	Age	Zip Code	Full Name
a	shanda	smith	42	94702	shanda smith
b	smiley	brocker	22	97402	rolly brocker
c	molly	stein	77	94223	molly stein
d	frank	bach	98	94705	frank bach
e	rip	spencer	25	97503	rip spencer
f	steven	de wilde	13	94705	steven de wilde

Метод работает и с регулярными выражениями.

Так, можно создать регулярное выражение, которое подбирает слова, начинающиеся на `s`, заменяя в них `s` на `S`:

```
participants.replace(r'(s)([a-z]+)', r'S\2', regex=True)
```

	first	last	Age	Zip Code	Full Name
a	shanda	smith	42	94702	Shanda Smith
b	rolly	brocker	22	97402	rolly brocker
c	molly	stein	77	94223	molly Stein
d	frank	bach	98	94705	frank bach
e	rip	spencer	25	97503	rip Spencer
f	steven	de wilde	13	94705	Steven de wilde
g	gwen	mason	45	94111	gwen mason
h	paul	davis	91	95333	arthur davis

И у датафрейма, и у объекта `pandas.Series` есть метод `apply()`, способный вызвать функцию для значений. В случае с объектом `pandas.Series` он вызывает выбранную вами функцию индивидуально для каждого значения в объекте.

Допустим, вы объявляете функцию, которая возвращает любую переданную ей строку с заглавной буквы:

```
def cap_word(w):
    return w.capitalize()
```

Затем, при передаче ее в качестве аргумента для `apply()` в столбце `first`, каждое имя будет написано с заглавной буквы:

```
participants.loc[:, 'first'].apply(cap_word)
a    Shanda
b     Rolly
```

```
c     Molly
d     Frank
e       Rip
f    Steven
g      Gwen
h      Paul
Name: first, dtype: object
```

В случае с датафреймом `apply` принимает строку в качестве аргумента, позволяя создавать новые значения из ее столбцов. Допустим, вы объявляете функцию, использующую значения из столбцов `first` и `Age`:

```
def say_hello(row):
    return f'{row["first"]} is {row["Age"]} years old.'
```

Затем можно применить функцию ко всему датафрейму:

```
participants.apply(say_hello, axis=1)
a    shanda is 42 years old.
b     rolly is 22 years old.
c     molly is 77 years old.
d     frank is 98 years old.
e       rip is 25 years old.
f    steven is 13 years old.
g      gwen is 45 years old.
h      paul is 91 years old.
dtype: object
```

Этот метод для вызова функции можно применять по строкам или столбцам. Чтобы указать, что именно должна ожидать ваша функция, используйте аргумент `axis`.

Интерактивный дисплей

Если вы работаете с датафреймами в среде Colab, попробуйте запустить этот фрагмент:

```
%load_ext google.colab.data_table
```

Это сделает выходные данные ваших датафреймов интерактивными и позволит вам фильтровать и выбирать в интерактивном режиме.

Резюме

Датафреймы Pandas — мощный инструмент для работы с данными в среде, похожей на электронную таблицу. Создавать их можно из многих ресурсов, но создание из файла — самое распространенное. Вы можете расширить датафрейм новыми столбцами и строками.

Вы можете получить доступ к данным с помощью индексаторов, которые допускается использовать и чтобы задавать эти самые данные. Датафрейм предоставляет отличный способ для изучения данных и управления ими.

Вопросы для закрепления

Для ответов на вопросы используйте эту таблицу:

Sample Size (mg)	%P	%Q
0.24	40	60
2.34	34	66
0.0234	12	88

1. Создайте датафрейм, представляющий эту таблицу.
2. Добавьте новый столбец с меткой `Total Q` с количеством `Q` (в миллиграммах) для каждого образца.
3. Разделите столбцы `%P` и `%Q` на 100.

10

Библиотеки визуализации данных

Величайшая ценность картины в том,
что она заставляет нас заметить то,
что мы никогда не ожидали увидеть.

Джон Тьюки

В этой главе

- Создание и оформление графиков с помощью инструмента `matplotlib`.
- Построение графиков с помощью библиотеки Seaborn и ее тем.
- Построение графиков с помощью библиотек Plotly и Bokeh.

Визуализация данных важна для их изучения и представления. Выражение «картина стоит тысячи слов» определенно применимо к пониманию данных. Из визуализаций часто можно получить информацию, которая не очевидна из сводной

статистики. Фрэнк Энском создал четыре набора данных, чья сводная статистика была практически идентична, но при построении графика они сильно различались.

Как правило, данные проще объяснить, опираясь на визуальное отображение. Подумайте, насколько информативнее может быть презентация с диаграммами и графиками. К счастью, в Python есть достаточно библиотек для визуализации.

Библиотека matplotlib

`matplotlib` — базовый инструмент для создания готового к публикации графика. Он широко применяется сам по себе и в качестве основы для других библиотек с целью построения графиков. Он — часть экосистемы SciPy, наряду с NumPy и Pandas. Это очень крупный проект с широкими возможностями, но как раз из-за его размера им может быть сложно пользоваться.

Есть много интерфейсов для применения `matplotlib`. Один из них — `pylab`, который обычно импортируется так:

```
from matplotlib.pylab import *
```

Хотя эти старые примеры и могут быть полезны, использование `pylab` больше не приветствуется. Изначально он был предназначен для моделирования среды, аналогичной `MATPLOT`, являющейся математическим инструментом построения графиков, не принадлежал Python. Но импорт всего содержимого модуля, что и делает `import*`, считается плохой практикой в Python. Вместо этого предлагается импортировать только то, что вы собираетесь использовать.

Рекомендованный интерфейс для `matplotlib` — `pyplot`, который по соглашению для удобства носит псевдоним `plt`:

```
import matplotlib.pyplot as plt
```

Два основных понятия в `matplotlib` — это фигуры и оси. Первые используются для отображения данных в виде графика. Вторые — это области, на которых точки могут быть заданы с помощью координат. *Оси* отображаются с помощью *фигур*. У одной фигуры может быть множество осей, но ось может быть прикреплена только к одной фигуре.

Для создания фигур и осей `matplotlib` предлагает два подхода: явный и неявный. Следующие примеры демонстрируют неявный подход.

Есть несколько методов построения графиков, таких как `plt.plot` и `plt.hist`, которые строят график для текущих осей и фигуры. Они создают ось и родительскую фигуру, если та еще не существует.

`plt.plot` создает линейный график на основании значений x и y, как на рис. 10.1:

```
[X = [0, 1, 2, 3, 4, 5, 7, 8, 9, 10]
Y = [20, 25, 35, 50, 10, 12, 20, 40, 70, 110]
plt.plot(X, Y)
```

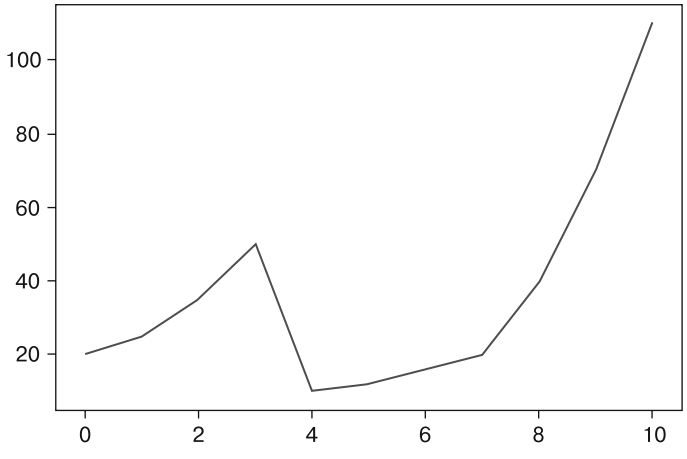


Рис. 10.1. Линейный график на основании значений x и y

Оформление графиков

Управлять стилем графика можно с помощью двух разных механизмов. Первый — применение свойств класса `matplotlib.Line2D`. Они управляют маркерами, используемыми в графике, стилем линии и цветом. Полный список свойств `matplotlib.Line2D` можно найти в документации `matplotlib` (https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.lines.Line2D.html).

Эти свойства можно применять как аргументы ключевого слова для `plt.plot`. В этом разделе показано использование свойств `marker`, `linestyle` и `color`.

Доступны следующие типы маркеров:

```
.       точка
,       пиксель
o       круг
v       triangle_down
^       triangle_up
<       triangle_left
>       triangle_right
```

```
1       tri_down
2       tri_up
3       tri_left
4       tri_right
s       квадрат
p       пентагон
*       звезда
h       шестиугольник 1
H       шестиугольник 2
+       плюс
x       x
D       бриллиант
d       thin_diamond
|       vline
_       hline
```

Можно указать тип маркера с помощью ключевого слова `marker`. Ниже представлен пример установки маркеров в виде квадратов (рис. 10.2):

```
X = [0, 1, 2, 3, 4, 5, 7, 8, 9, 10]
Y = [20, 25, 35, 50, 10, 12, 20, 40, 70, 110]
plt.plot(X, Y, marker='s')
```

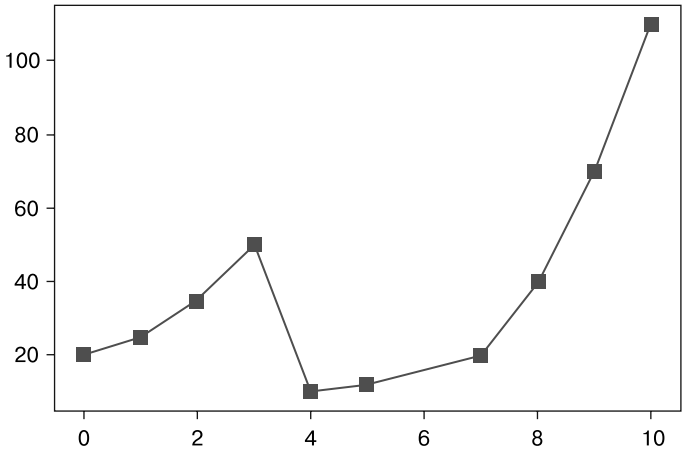


Рис. 10.2. График с маркерами в виде квадратов

Доступны следующие стили линий:

```
-       сплошной стиль (solid)
--      штриховой стиль (dashed)
```

```
-.      стиль штрих-пунктир (dash-dot)
:       пунктирный стиль (dotted)
```

Чтобы задать стиль линии, используйте ключевое слово `linestyle` (рис. 10.3):

```
X = [0, 1, 2, 3, 4, 5, 7, 8, 9, 10]
Y = [20, 25, 35, 50, 10, 12, 20, 40, 70, 110]
plt.plot(X, Y, marker='s', linestyle=':')
```

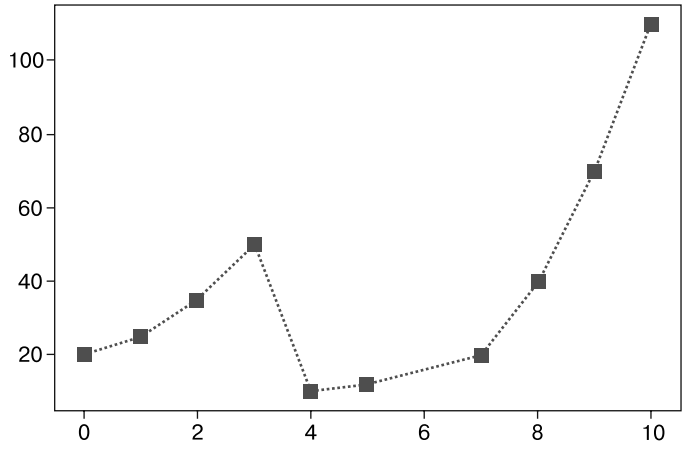


Рис. 10.3. Параметр, заданный с помощью ключевого слова linestyle

Ниже представлены доступные цвета:

```
b       синий
g       зеленый
r       красный
c       голубой
m       пурпурный
y       желтый
k       черный
w       белый
```

Задать цвет можно с помощью ключевого слова `color`. Если вы попробуете выполнить этот пример, то увидите график как на рис. 10.3, но в цвете:

```
X = [0, 1, 2, 3, 4, 5, 7, 8, 9, 10]
Y = [20, 25, 35, 50, 10, 12, 20, 40, 70, 110]
plt.plot(X, Y, marker='s', linestyle=':', color='m')
```

Другой способ настройки свойств стиля — использование аргумента `fmt`. Это позиционный параметр, который отображается справа от параметра `Y`. Он состоит из форматирующей строки, использующей условное обозначение для настроек маркера, стиля линии и цвета. Форматирующая строка имеет вид `[marker][linestyle][color]`, причем все разделы опциональны. Например, для графика на рис. 10.4 вы можете задать квадратные маркеры, пунктирный стиль линии и красный цвет с помощью форматирующей строки `'s-.r'`:

```
X = [0, 1, 2, 3, 4, 5, 7, 8, 9, 10]
Y = [20, 25, 35, 50, 10, 12, 20, 40, 70, 110]
fmt = 's-.r'
plt.plot(X, Y, fmt)
```

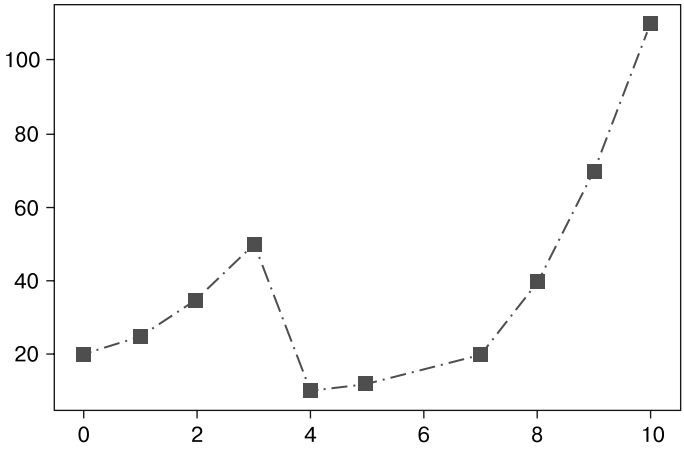


Рис. 10.4. Линейный график с использованием форматирующей строки 's-.r'

Можно использовать форматирующую строку и аргумент ключевого слова вместе. Например, график на рис. 10.5 совмещает форматирующую строку `'s-.r'` и ключевой аргумент `linewidth`:

```
X = [0, 1, 2, 3, 4, 5, 7, 8, 9, 10]
Y = [20, 25, 35, 50, 10, 12, 20, 40, 70, 110]
fmt = 's-.r'
plt.plot(X, Y, fmt, linewidth=4.3)
```

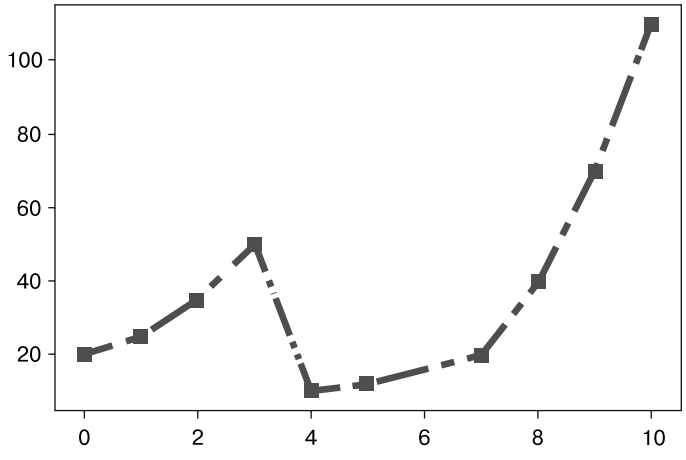


Рис. 10.5. Линейный график, отформатированный с помощью ключевого слова linewidth

Маркировка данных

Функции построения графиков `matplotlib` могут использовать маркированные данные. Это касается датафреймов Pandas, словарей и любых структур, где доступ к данным осуществляется с помощью синтаксиса скобок. Вместо указания последовательности значений для `x` и `y` вы указываете соответствующие метки.

Вот как можно создать датафрейм, содержащий средний рост американских мужчин и женщин старше 16 лет, основанный на данных Центров по контролю и профилактике заболеваний (https://www.cdc.gov/nchs/data/nhsr/nhsr122-508.pdf):

```
import pandas as pd

data = {"Years": ["2000", "2002", "2004", "2006", "2008",
                  "2010", "2012", "2014", "2016"],
        "Men": [189.1, 191.8, 193.5, 196.0, 194.7,
                196.3, 194.4, 197.0, 197.8],
        "Women": [175.7, 176.4, 176.5, 176.2, 175.9,
                  175.9, 175.7, 175.8, 175.3]}
heights_df = pd.DataFrame(data)
```

Вы можете создать линейный график роста женщин, указав метки столбцов, которые будут использоваться для `x` и `y`, а также датафрейм, из которого будете извлекать данные (см. рис. 10.6):

```
plt.plot('Years', 'Women', data=heights_df)
```

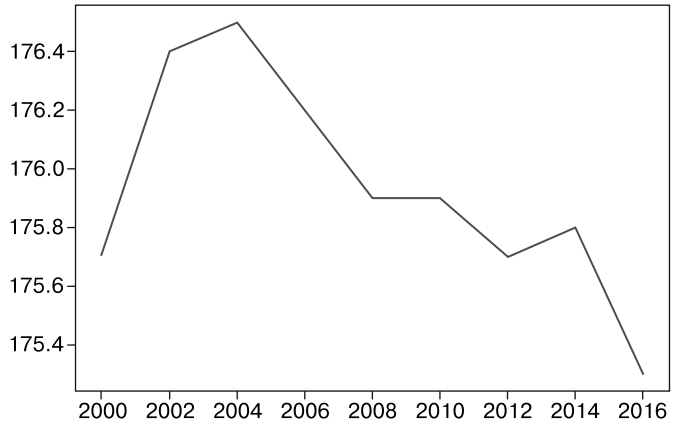


Рис. 10.6. Линейный график с заданными метками x и y

Построение графиков для множества наборов данных

Есть три подхода к отображению множества наборов данных на одном графике. Первый — вызов функции построения графиков множество раз:

```
X = [0, 1, 2, 3, 4, 5, 7, 8, 9, 10]
Y = [20, 25, 35, 50, 10, 12, 20, 40, 70, 110]
fmt = 's-.r'

X1 = [0, 1, 2, 3, 4, 5, 7, 8, 9, 10]
Y2 = [90, 89, 87, 82, 72, 60, 45, 28, 10, 0]
fmt2 = '^k:'
plt.plot(X, Y, fmt)
plt.plot(X1, Y2, fmt2)
```

Помните, что `plt.plot` использует текущие оси и фигуру. Это значит, что множественные вызовы будут продолжать использовать одни и те же фигуру и график. На рис. 10.7 можно увидеть много графиков на одной фигуре.

Второй способ — передача множества наборов данных напрямую в функцию построения графиков:

```
plt.plot(X, Y, fmt, X1, Y2, fmt2)
```

Для маркированных данных можно передать множественные метки, а каждый столбец будет добавлен в график (рис. 10.8):

```
plt.plot('Years', 'Women', 'Men', data=heights_df)
```

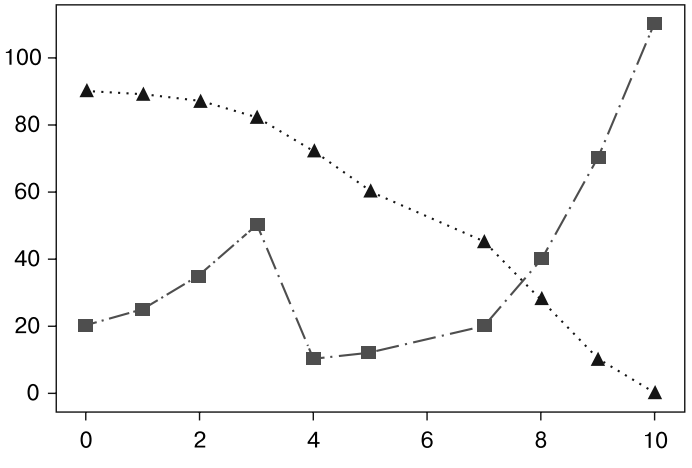


Рис. 10.7. Много графиков на одной фигуре

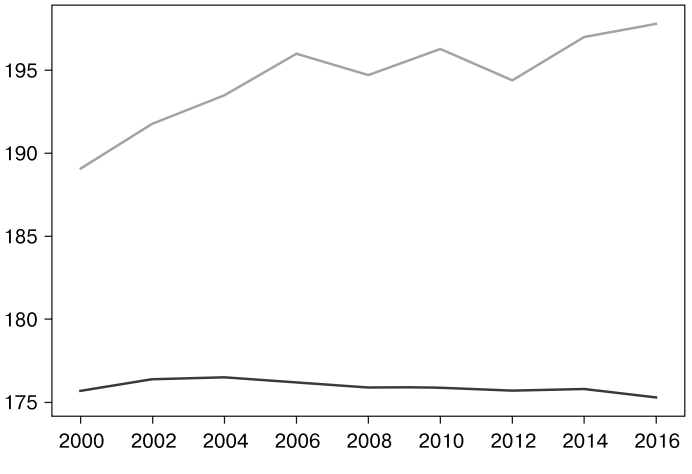


Рис. 10.8. Линейный график с множеством наборов данных

`matplotlib` предлагает удобные функции для добавления меток, названия и легенды графика. Можно создать маркированную версию графика с рис. 10.8 (рис. 10.9):

```
plt.plot('Years', 'Women', 'Men', data=heights_df)
plt.xlabel('Year')
plt.ylabel('Height (Inches)')
plt.title("Heights over time")
plt.legend(['Women', 'Men'])
```

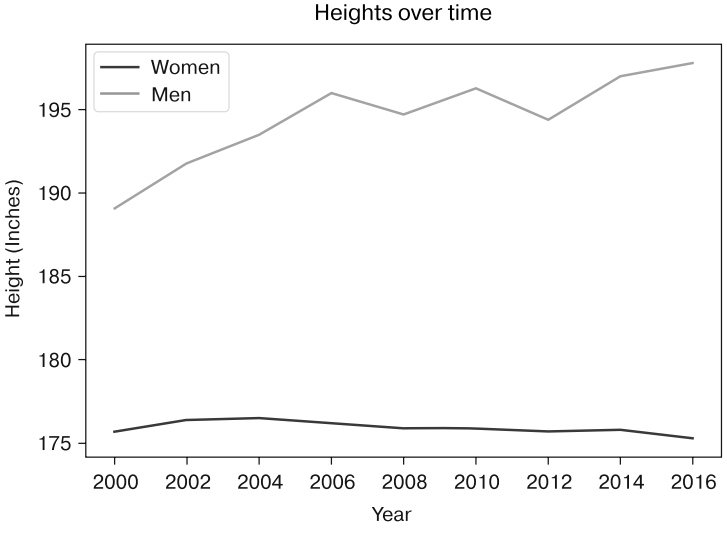


Рис. 10.9. Добавление множества меток в график

Объектно-ориентированный стиль

Неявный способ работы с фигурами и осями, который мы с вами разобрали, удобен для изучения данных, особенно в интерактивной среде. `matplotlib` тоже позволяет вам работать напрямую с фигурами и осями, что дает вам больше контроля. Функция `plt.subplots()` возвращает фигуру и указанное количество осей. Теперь можно строить графики по осям почти так же, как при использовании неявного построения:

```
fig, ax = plt.subplots()
ax.plot('Years', 'Women', 'Men', data=heights_df)
ax.set_xlabel('Year')
ax.set_ylabel('Height (Inches)')
```

```
ax.set_title("Heights over time")
ax.legend(['Women', 'Men'])
```

Результаты будут совпадать с полученными с помощью тех же данных, что и на рис. 10.9.

Для построения множества графиков на одной фигуре задайте множество осей, как в листинге 10.1. Первый аргумент определяет число строк, второй задает число столбцов. На рис. 10.10 показан пример создания двух осей в фигуре.

Листинг 10.1. Создание множества осей

```
fig, (ax1, ax2) = plt.subplots(1, 2).       # Создание одной фигуры и двух осей

ax1.plot('Years', 'Women', data=heights_df) # График женщины/год на первой оси
ax1.set_xlabel('Year')                      # Метка оси x первой оси
ax1.set_ylabel('Height (Inches)')           # Метка оси y первой оси
ax1.set_title("Women")                      # Установка заголовка первой оси
ax1.legend(['Women'])                       # Установка легенды первой оси

ax2.plot('Years', 'Men', data=heights_df )  # Построение второй оси
ax2.set_xlabel('Year')                      # Установка метки x для второй оси
ax2.set_title("Men")                        # Установка заголовка для второй оси
ax2.legend(['Men'])                         # Установка легенды для второй оси

fig.autofmt_xdate(rotation=65)              # Поворот меток даты
```

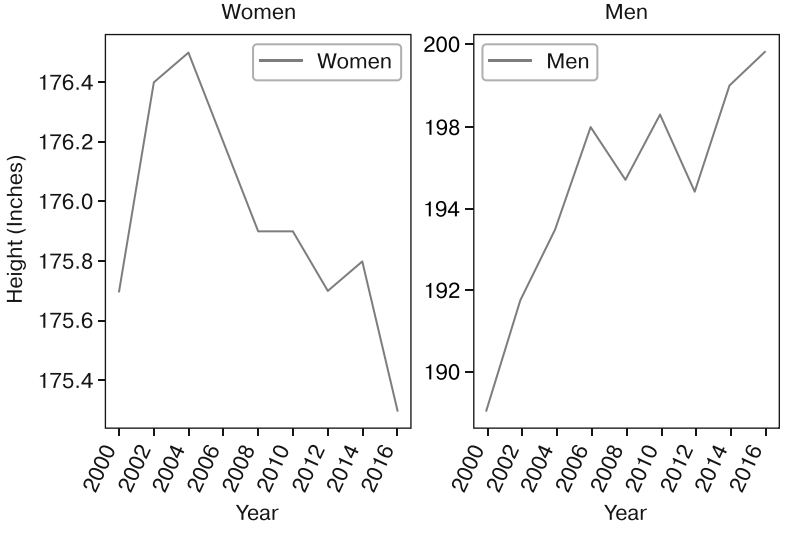


Рис. 10.10. Построение графика двух осей в фигуре

Неявный стиль построения графиков отлично подходит для интерактивного изучения данных. Явный стиль дает вам гораздо больше контроля и рекомендован для построения графика в продакшен-коде.

Библиотека Seaborn

Seaborn — это библиотека для создания статистических графиков, которая строится поверх `matplotlib`. Она разработана для упрощения создания красивых статистических графиков и известна наличием стиля по умолчанию, который выглядит лучше, чем предлагаемые другими библиотеками.

По соглашению эта библиотека импортируется как `sns`:

```
import seaborn as sns
```

Seaborn содержит наборы выборочных данных, используемых в предоставленной документации и руководствах. Эти наборы данных также удобный источник для изучения свойств Seaborn. Их можно загрузить как датафреймы Pandas с помощью функции `sns.load_dataset()` с именем набора данных в качестве аргумента. Доступные наборы данных можно найти по ссылке: https://github.com/mwaskom/seaborn-data.

Ниже показано, как загрузить набор данных автомобильных аварий и выбрать столбцы для работы:

```
car_crashes = sns.load_dataset('car_crashes')
car_crashes = car_crashes[['total', 'not_distracted', 'alcohol']]
```

В этом примере функция Seaborn `sns.relplot()` используется для построения графика, отображающего зависимость между двумя столбцами (рис. 10.11):

```
sns.relplot(data=car_crashes,
            x='total',
            y='not_distracted')
```

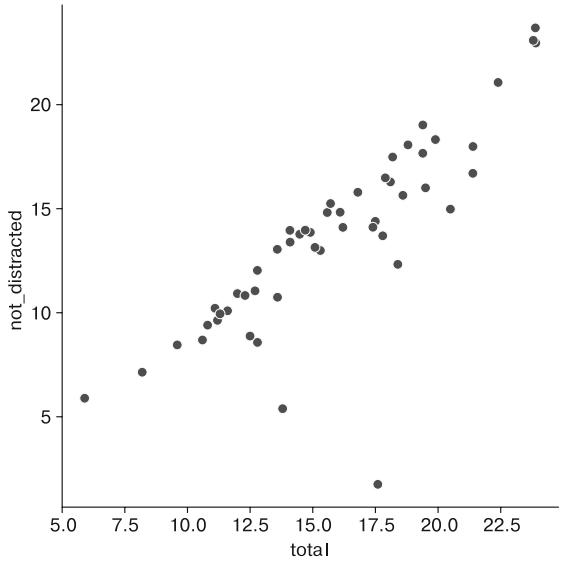


Рис. 10.11. Использование sns.relplot() для построения зависимости между двумя столбцами

Темы Seaborn

Использование тем Seaborn — простой способ контролировать внешний вид диаграмм. Применить одну из тем по умолчанию можно с помощью следующей функции:

```
sns.set_theme()
```

Вы можете перестроить график, чтобы посмотреть новый вид, показанный на рис. 10.12:

```
sns.relplot(data=car_crashes,
            x='total',
            y='not_distracted')
```

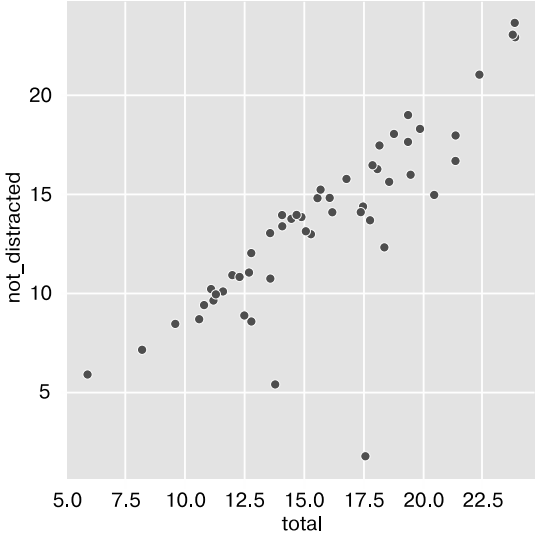


Рис. 10.12. Использование тем Seaborn для управления видом графика

После установки темы Seaborn она применяется к любым последующим графикам, даже созданным напрямую с помощью `matplotlib`. Seaborn объединяет параметры `matplotlib` в две группы: одна имеет дело с эстетическим видом графика, а вторая — с элементами масштаба.

В Seaborn доступны пять предустановленных тем: `darkgrid`, `whitegrid`, `dark`, `white`, и `ticks`. Стиль можно установить с помощью функции `sns.set_style()`. Например, стиль `dark` применяется следующим образом:

```
sns.set_style('dark')
sns.relplot(data=car_crashes,
            x='total',
            y='not_distracted')
```

Темы, доступные для настройки масштаба элементов фигуры, основаны на целевом представлении. К ним относятся `paper`, `notebook`, `talk` и `poster`.

Установить тему можно с помощью функции `sns.set_context()` (рис. 10.13):

```
sns.set_context('talk')
```

При перепостроении графика масштаб скорректируется (рис. 10.14).

```
sns.relplot(data=car_crashes,
            x='total',
            y='not_distracted')
```

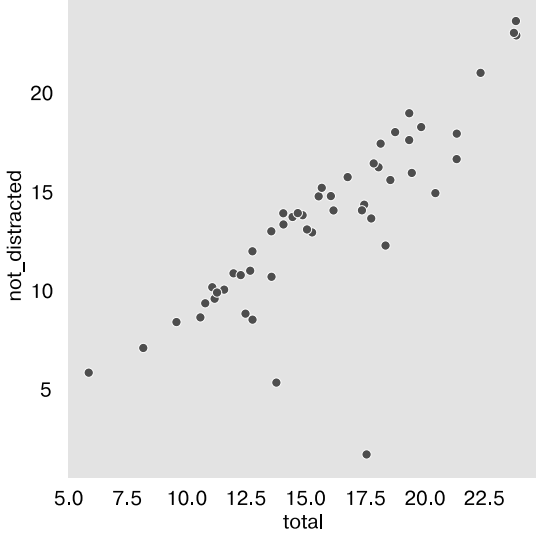


Рис. 10.13. Использование темы Dark Style

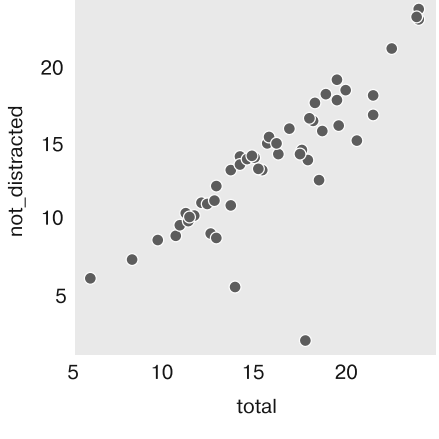


Рис. 10.14. Данные, перестроенные с помощью функции sns.set_context

Seaborn предлагает много типов графиков. Один из наиболее полезных типов для поиска корреляции в данных — `sns.pairplot()`. Он создает сетку осей, отображающих взаимосвязь между всеми столбцами датафрейма. Создать парный график с помощью набора данных `iris` можно так (рис 10.15):

```
df = sns.load_dataset('iris')
sns.pairplot(df, hue='species')
```

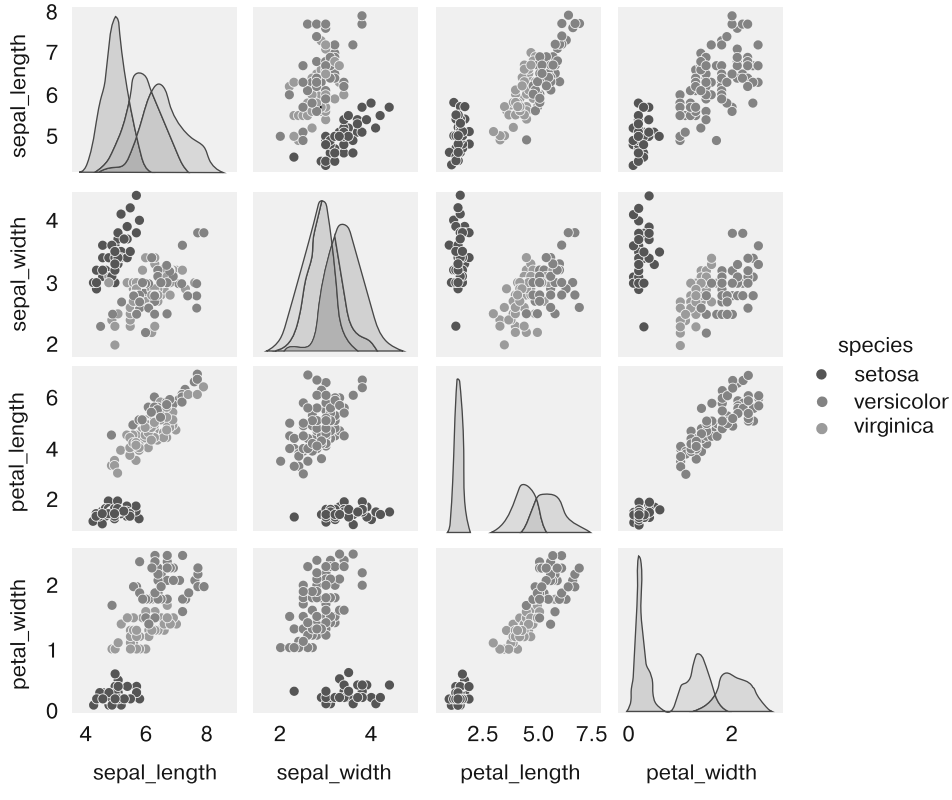


Рис. 10.15. Парный график с использованием набора данных Iris

Библиотека Plotly

`matplotlib` и Seaborn — отличные инструменты для создания готовых к публикации статичных графиков. Оба могут быть расширены для создания интерактивных

презентаций данных. Но библиотеки Plotly и Bokeh предназначены специально для создания высококачественных интерактивных графиков. Plotly предлагает много типов графиков, но его отличительная черта в том, что он упрощает построение 3D-графиков. Рисунок 10.16 показывает статическую версию динамического графика. Запустив код в блокноте, вы сможете поворачивать и масштабировать этот график:

```
import plotly.express as px
iris = px.data.iris()
fig = px.scatter_3d(iris,
                    x='sepal_length',
                    y='petal_width',
                    z='petal_length',
                    color='species')
fig.show()
```

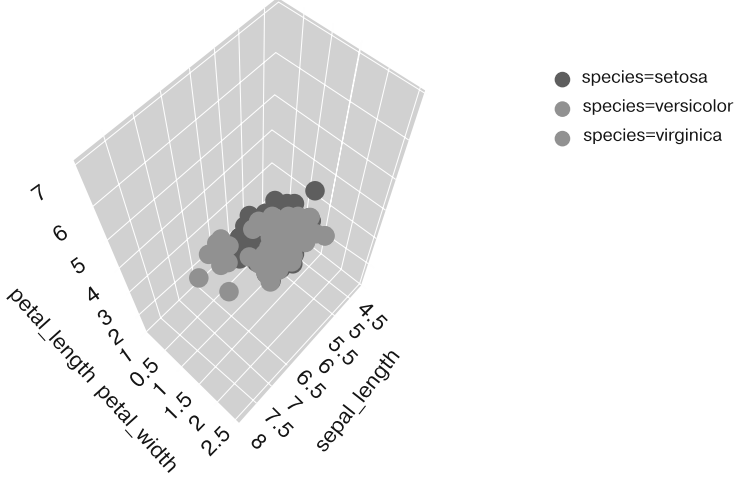


Рис. 10.16. Статическая версия динамического графика

Библиотека Bokeh

Bokeh — это альтернатива Plotly для простого создания интерактивных графиков. Одно из преимуществ Bokeh — в использовании специального объекта данных `ColumnDataSource`. Он обеспечивает повышенную производительность, позволяя обновлять и прикреплять данные без перезагрузки состояния. Также источник

данных может быть общим для фигур. Поэтому взаимодействие с данными в одной фигуре изменяет их в другой. В листинге 10.2 задано множество фигур с общим объектом данных, а рис. 10.17 отображает результат.

ПРИМЕЧАНИЕ

Имейте в виду, что Bokeh требует дополнительной установки в Colab. Этот пример дает вам представление о возможностях Bokeh, но не показывает, как заставить ее работать в Colab.

Листинг 10.2. Общие данные Bokeh

```
from bokeh.io import output_notebook
from bokeh.plotting import figure, show
from bokeh.models import ColumnDataSource
from bokeh.layouts import gridplot

Y = [x for x in range(0,200, 2)]
Y1 = [x**2 for x in Y]
X = [x for x in range(100)]
data={'x':X,
      'y':Y,
      'y1':Y1}

TOOLS = "box_select"                  # Выбор интерактивных инструментов
source = ColumnDataSource(data=data)  # Создание ColumnDataSource
left = figure(tools=TOOLS,
              title='Brushing')       # Создание рисунка выбранными инструментами

left.circle('x',
            'y',
            source=source)            # Создание кругового графика на первой фигуре

right = figure(tools=TOOLS,
               title='Brushing')      # Создание рисунка выбранными инструментами
right.circle('x',
             'y1',
            source=source)            # Создание кругового графика на второй фигуре

p = gridplot([[left, right]])         # Размещение фигур на сетке
show(p)                               # Вывод сетки
```

Выводимые фигуры позволяют осуществлять отбор поперечных осей с учетом выбранного инструмента. Это значит, что при выборе участка одного графика соответствующие точки во втором тоже будут выбраны.

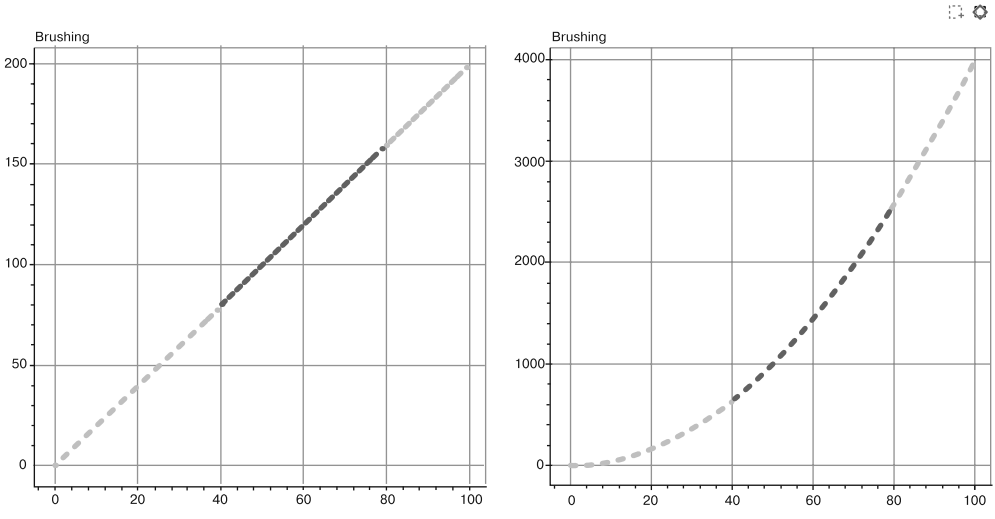


Рис. 10.17. Множество фигур с общим объектом данных

Другие библиотеки визуализации

Есть много разных отличных библиотек визуализации помимо описанных в этой главе. Ниже представлены некоторые из них, которые могут вас заинтересовать:

- `geoplotlib` — позволяет визуализировать карты и географические данные;
- `ggplot` — основанный на языке R пакет ggplot2;
- `pygal` — упрощает создание простых графиков;
- `folium` — позволяет создавать интерактивные карты;
- `missingno` — позволяет визуализировать недостающие данные.

Резюме

Визуализация — невероятно полезная часть исследования данных и важная часть их представления. Есть много библиотек, позволяющих визуализировать данные, у каждой из которых свои особенности и направленности. `matplotlib` — основа для многих из них. Она дает широкие возможности, но требует серьезного изучения. Seaborn — это библиотека статистической визуализации, построенная на базе `matplotlib`. Она упрощает работу над внешним видом графика и создание графиков для разных целевых сред. Plotly и Bokeh предназначены для работы с интерактивными графиками и дашбордами.

Вопросы для закрепления

Для ответа на вопросы используйте следующий код:

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

data = {'X'  : [x for x in range(50)],
        'Y'  : [y for y in range(50, 0, -1)],
        'Y1' : [y**2 for y in range(25, 75)]}

df = pd.DataFrame(data)
```

1. Используйте `matplotlib` для построения графика зависимости между столбцами `X` и `Y`.
2. Используйте `matplotlib` для добавления зависимости между столбцами `X` и `Y1` в том же графике.
3. Используйте `matplotlib` для построения графика зависимостей из вопросов 1 и 2 на отдельных осях одной фигуры.
4. Используйте Seaborn для изменения темы на `darkgrid` и повторите графики из вопроса 3.

11

Библиотеки машинного обучения

Проконсультироваться со статистиком по завершении эксперимента часто означает попросить его провести посмертное обследование. Возможно, он скажет, от чего умер эксперимент.

Рональд Фишер

В этой главе

- Обзор популярных библиотек машинного обучения.
- Знакомство с библиотекой Scikit-learn.
- Знакомство с процессом машинного обучения.

Машинное обучение позволяет компьютеру найти способ решения задачи на основе данных. В этом отличие от традиционного программирования, где программист определяет в коде способ поиска решения, а не само решение. В этой главе вы

познакомитесь с некоторыми популярными библиотеками, используемыми для машинного обучения. Они реализуют алгоритмы для создания и подготовки моделей машинного обучения, которые могут быть использованы по-разному в зависимости от типа задачи. Например, одни модели полезны для прогнозирования значений, а другие — для классификации данных по группам или категориям.

Популярные библиотеки машинного обучения

Есть четыре наиболее популярные библиотеки — TensorFlow, Keras, PyTorch, и Scikit-learn.

- **TensorFlow.** Google разработал эту мощную библиотеку для внутреннего использования. Она применяется для решения задач с помощью глубокого обучения и включает в себя определение слоев, которые преобразуют данные и настраиваются по мере соответствия решения им.
- **Keras.** Эта библиотека с открытым исходным кодом была создана для работы с TensorFlow и теперь включена в библиотеку TensorFlow (https://www.tensorflow.org/guide).
- **PyTorch.** Это вклад Facebook в создание пригодных для производства библиотек машинного обучения. Она основана на библиотеке Torch, использующей графические процессоры для решения задач глубокого обучения (https://pytorch.org/docs/stable/index.html).
- **Scikit-learn.** Эта популярная библиотека для запуска машинного обучения построена на базе библиотек NumPy и SciPy. В ней есть классы для большинства традиционных алгоритмов. Вы узнаете больше о ней в следующем разделе.

Принцип работы машинного обучения

Алгоритмы машинного обучения можно разделить на два типа: относящиеся к неконтролируемому и контролируемому обучению. Неконтролируемое обучение — это получение информации о данных без ранее существовавших результатов для проверки. Обычно это означает выявление закономерностей на основе характеристик данных без участия человека. Контролируемое обучение — это использование известных данных для обучения и тестирования модели.

Шаги по обучению контролируемой модели следующие.

1. Преобразование данных.
2. Выделение данных для проверки.

3. Обучение модели.
4. Проверка точности.

У Scikit-learn есть инструменты для упрощения этих шагов, о чем мы с вами поговорим далее.

Преобразования

Для некоторых алгоритмов выгодно преобразовать данные перед началом обучения модели. Например, вы можете захотеть превратить непрерывную переменную, такую как возраст, в дискретную категорию, такую как возрастные диапазоны.

У Scikit-learn есть много типов преобразователей, включая те, что предназначены для очистки, выделения признаков, сокращения и расширения. Они представлены как классы, обычно использующие метод `.fit()` для определения преобразования и `.transform()` для изменения данных с его помощью. В листинге 11.1 использован преобразователь `MinMaxScaler`, который масштабирует значения, чтобы они соответствовали определенному диапазону — от 0 до 1 по умолчанию.

Листинг 11.1. Преобразование с помощью MinMaxScaler

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler

data = np.array([[100, 34, 4],
                 90, 2, 0],
                 78, -12, 16],
                 23, 45, 4]]) # Массив с диапазоном данных от -12 до 100

data
array([[100, 34,   4],
       [ 90,  2,   0],
       [ 78, -12, 16],
       [ 23, 45, 4]])
minMax = MinMaxScaler()        # Создание объекта преобразования
scaler = minMax.fit(data)      # Подгонка объекта преобразования под данные

scaler.transform(data)         # Масштабирование в диапазоне от 0 до 1
array([[1.        , 0.80701754, 0.25 ],
       [0.87012987, 0.24561404, 0.   ],
       [0.71428571, 0.        , 1.   ],
       [0.        , 1.        , 0.25 ]])
```

Бывают ситуации, когда вы хотите разделить ваши данные перед подгонкой преобразователя. Это никак не повлияет на настройки преобразователя. Подгонка и преобразование требуют отдельных методов. Легко подогнать тренировочные данные и использовать их для преобразования тестовых.

Разделение тестовых и тренировочных данных

Одна из серьезных ошибок при обучении модели — переобучение. Так случается, когда модель идеально прогнозирует данные, использованные для ее обучения, но в случае с новыми данными эта способность ухудшается. Проще говоря, чтобы избежать переобучения, не тестируйте модель с помощью данных, использованных при ее обучении. Scikit-learn предлагает вспомогательные методы, облегчающие разделение данных.

Прежде чем рассмотреть пример разделения данных, можете загрузить простой пример. Как и ряд других библиотек Data Science, Scikit-learn поставляется с образцами наборов данных.

В листинге 11.2 загружен набор данных `iris`. Обратите внимание, что функции `load_iris()` загружают два массива данных NumPy: исходные данные (характеристики, которые будут использованы для прогнозов) и целевая характеристика для прогнозирования. В случае с `iris` у исходных данных есть 150 образцов четырех параметров и 150 целевых объектов, предоставляющих типы `iris`.

Листинг 11.2. Загрузка образцов наборов данных

```
from sklearn import datasets # Загрузка образцов наборов данных
source, target = datasets.load_iris(return_X_y=True) # Загрузка источника
                                                     # и целевых объектов

print(type(source))
<class 'numpy.ndarray'>
print(source.shape)
(150, 4)

print(type(target))
<class 'numpy.ndarray'>
print(target.shape)
(150,)
```

В листинге 11.3 функция `train_test_split()` Scikit-learn используется для разделения предоставленного библиотекой набора данных `iris` на обучающие и тестовые наборы данных. Здесь видно, что образцы разделены так, что 112 из них входят в обучающий набор, а 38 — в тестовый.

Листинг 11.3. Разделение набора данных

```
from sklearn.model_selection import train_test_split

train_s, test_s, train_t, test_t = train_test_split(source, target)
train_s.shape
(112, 4)

train_t.shape
(112,)

test_s.shape
(38, 4)

test_t.shape
(38,)
```

Обучение и тестирование

Scikit-learn предлагает множество классов, предоставляющих разные алгоритмы машинного обучения. Они называются *оценщиками*. Многие из них можно настроить с помощью параметров во время создания экземпляра.

У каждого оценщика есть метод `.fit()`, обучающий модель. Большинство этих методов принимают два аргумента: первый — обучающие данные (*образцы*), второй — результаты или цели для этих образцов. Оба аргумента должны быть объектами, подобными массиву, например массивом NumPy. По завершении обучения модель может прогнозировать результаты с помощью ее метода `.predict()`. Точность этого прогноза можно проверить с помощью функций из модуля метода.

В листинге 11.4 показан простой пример использования оценщика `KNeighborsClassifier`. Метод *k*-ближайших соседей — это алгоритм, группирующий образцы на основании расстояния между характеристиками. Его прогнозы основаны на сравнении нового образца с существующими, находящимися ближе всего к нему.

Алгоритм можно настроить, выбрав количество соседей, которые будут сравниваться с новым образцом. После обучения модели можно делать прогнозы с помощью тестовых данных и проверять их точность.

Листинг 11.4. Обучение модели

```
from sklearn.neighbors import KNeighborsClassifier  # Импорт класса оценщика
from sklearn import metrics  # Импорт модуля метрик для проверки точности
knn = KNeighborsClassifier(n_neighbors=3)  # Создание класса оценки с 3 соседями
knn.fit(train_s, train_t)          # Обучение модели с использованием обучающих данных
test_prediction = knn.predict(test_s)      # Прогнозы на основе исходных данных
```

```
metrics.accuracy_score(test_t, test_prediction)    # Точность по сравнению
                                                   # с тестовыми данными
0.8947368421052632
```

Подробнее о Scikit-learn

Эта глава лишь поверхностно описывает возможности Scikit-learn. У этой библиотеки есть множество полезных функций и инструментов. Например, инструменты для перекрестной проверки, когда набор данных разбивается много раз во избежание переобучения тестовых данных, и конвейеры, объединяющие преобразователи, оценщики и перекрестную проверку. За подробной информацией о Scikit-learn и дополнительными обучающими материалами обращайтесь по адресу https://scikit-learn.org/stable/.

Резюме

Многие алгоритмы для создания моделей машинного обучения представлены в основных библиотеках Python, специализирующихся на этом. TensorFlow — это библиотека глубокого обучения от Google. PyTorch — библиотека, построенная на базе Torch компанией Facebook. Scikit-learn — популярная библиотека для начала работы с машинным обучением. В ней есть модули и функции для создания и анализа модели.

Вопросы для закрепления

1. На каком этапе обучения контролируемого оценщика будет полезен преобразователь Scikit-learn?
2. Почему важно разделять обучающие и тестовые данные в машинном обучении?
3. Что нужно сделать после преобразования данных и обучения модели?

12

Инструментарий естественного языка (NLTK)

Во введении в статистику первым делом изучают тот факт,
что корреляция — это не причинно-следственная связь.
Это также и то, о чем сразу забывают.

Томас Соуэлл

В этой главе

- Знакомство с пакетом NLTK.
- Доступ к образцам текста и их загрузка.
- Использование частотного распределения.
- Текстовые объекты.
- Классификация текста.

Использовать компьютер для получения понимания текста очень полезно. Подмножество data science, которое занимается получением представления о тексте, называется *обработкой естественного языка*. Инструментарий естественного языка (NLTK) — это пакет Python для всех видов языковой обработки, который мы кратко рассмотрим в этой главе.

Образцы текстов NLTK

Пакет NLTK предлагает образцы текстов из многих источников, которые вы можете загрузить и использовать для изучения языковой обработки. «Гутенберг» — это проект, который выкладывает копии книг онлайн (см. http://www.gutenberg.org). Он состоит в основном из книг, являющихся общественным достоянием. Подмножество этой коллекции доступно для загрузки и использования с NLTK. Используйте функцию `nltk.download()` для загрузки данных в папку nltk_data/corpora в вашем домашнем каталоге.

```
import nltk
nltk.download('gutenberg')
[nltk_data]   Downloading package gutenberg to
[nltk_data]   /Users/kbehrman/nltk_data...
[nltk_data]   Unzipping corpora/gutenberg.zip.
True
```

Затем можно импортировать данные в вашу сессию Python в качестве объекта читателя корпусов

ПРИМЕЧАНИЕ

Каждый читатель корпусов предназначен для чтения определенного набора текстов, предоставленных NLTK.

```
from nltk.corpus import gutenberg
gutenberg
<PlaintextCorpusReader in '/Users/kbehrman/nltk_data/corpora/gutenberg'>
```

Есть читатели корпусов для разных типов текстовых источников. В этом примере используется `PlaintextCorpusReader`, который создан для простого текста. Список отдельных текстов можно сформировать с помощью метода `fileids()`, составляющего список имен файлов, которые могут быть использованы для загрузки текстов:

```
gutenberg.fileids()
['austen-emma.txt',
 'austen-persuasion.txt',
 'austen-sense.txt',
 'bible-kjv.txt',
 'blake-poems.txt',
 'bryant-stories.txt',
 'burgess-busterbrown.txt',
 'carroll-alice.txt',
 'chesterton-ball.txt',
 'chesterton-brown.txt',
 'chesterton-thursday.txt',
 'edgeworth-parents.txt',
 'melville-moby_dick.txt',
 'milton-paradise.txt',
 'shakespeare-caesar.txt',
 'shakespeare-hamlet.txt',
 'shakespeare-macbeth.txt',
 'whitman-leaves.txt']
```

У читателя корпусов есть много методов чтения текста. Вы можете загрузить текст, разбитый на отдельные слова, предложения или абзацы. В листинге 12.1 загружен текст трагедии Уильяма Шекспира «Юлий Цезарь» во всех трех форматах.

Листинг 12.1. Загрузка текста

```
caesar_w = gutenberg.words('shakespeare-caesar.txt') # Список слов caesar_w
['[', 'The',  'Tragedie',  'of',  'Julius',  'Caesar', ...]
nltk.download('punkt') # Загрузка токенизатора, используемого
                       # для определения окончаний предложений
[nltk_data] Downloading package punkt to /Users/kbehrman/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.
True

caesar_s = gutenberg.sents('shakespeare-caesar.txt') # Список предложений caesar_s
[['[', 'The', 'Tragedie', 'of', 'Julius', 'Caesar', 'by', 'William',
    'Shakespeare', '1599', ']'], ['Actus', 'Primus', '.'], ...]

caesar_p = gutenberg.paras('shakespeare-caesar.txt') # Список абзацев caesar_p
[[['[', 'The', 'Tragedie', 'of', 'Julius', 'Caesar', 'by', 'William',
     'Shakespeare','1599', ']']], [['Actus', 'Primus', '.'],
['Scoena', 'Prima', '.']], ...]
```

Обратите внимание: прежде чем разбирать текст на предложения, нужно загрузить токенизатор Punkt. *Токенизатор* используется для разбиения (токенизации) фрагмента текста. Punkt используется для разделения текста на предложения и поддерживает тексты на многих языках.

В листинге 12.2 показано, как просмотреть подкаталог NLTK вашего домашнего каталога с помощью команды оболочки `ls`, составляющей список объектов и каталогов. Здесь видно, что есть каталоги для корпусов и для токенизаторов. В первом можно увидеть загруженную коллекцию, а во втором — загруженный токенизатор. В подкаталоге punkt находятся файлы для каждого доступного языка.

Листинг 12.2. Каталог данных

```
!ls /root/nltk_data
corpora    tokenizers

!ls /root/nltk_data/corpora
gutenberg     gutenberg.zip

!ls /root/nltk_data/tokenizers
punkt     punkt.zip

!ls /root/nltk_data/tokenizers/punkt
PY3               english.pickle     greek.pickle      russian.pickle
README            estonian.pickle    italian.pickle    slovene.pickle
czech.pickle      finnish.pickle     norwegian.pickle  spanish.pickle
danish.pickle     french.pickle      polish.pickle     swedish.pickle
dutch.pickle      german.pickle      portuguese.pickle turkish.pickle
```

Частотное распределение

Число появлений каждого слова в тексте можно посчитать с помощью класса `nltk.FreqDist`. У него есть методы, позволяющие увидеть количество отдельных слов в тексте и какие слова встречаются чаще всего (здесь термин *слово* относится к любому фрагменту текста, который не является пробелом).

`FreqDist` выделяет в тексте знаки препинания как отдельные слова. Ниже он использован для поиска наиболее часто встречающихся слов в тексте:

```
caesar_dist = nltk.FreqDist(caesar_w)
caesar_dist.most_common(15)
 [(',', 2204),
  ('.', 1296),
  ('I', 531),
  ('the', 502),
  (':', 499),
  ('and', 409),
  ("""", 384),
  ('to', 370),
```

```
 ('you', 342),
 ('of', 336),
 ('?', 296),
 ('not', 249),
 ('a', 240),
 ('is', 230),
 ('And', 218)]
```

Чтобы увидеть наиболее распространенные слова без знаков препинания, вы можете отфильтровать эти знаки. В модуле `string` стандартной библиотеки Python есть атрибут пунктуации, который можно использовать для этого.

Листинг 12.3 перебирает исходные слова текста. Он проверяет, является ли каждый элемент знаком препинания, и если нет, то добавляет его в новый список в переменную `caesar_r`. Сравнивая длину исходного и отфильтрованного файлов, этот листинг находит в тексте 4960 знаков препинания. Далее создается новое частотное распределение, чтобы отобразить наиболее распространенные слова без знаков препинания.

Листинг 12.3. Удаление пунктуации

```
import string
string.punctuation                    # Взгляд на строку с пунктуацией
'!"#$%&\'()*+,-./:;<=>?@[\\]^_'{|}~'

caesar_r = []
for word in caesar_w:
     if word not in string.punctuation:
         caesar_r.append(word)        # Добавление слов без пунктуации

len(caesar_w) - len(caesar_r)         # Число знаков пунктуации
4960

caesar_dist = nltk.FreqDist(caesar_r)
caesar_dist.most_common(15)
[('I', 531),
 ('the', 502),
 ('and', 409),
 ('to', 370),
 ('you', 342),
 ('of', 336),
 ('not', 249),
 ('a', 240),
 ('is', 230),
 ('And', 218),
 ('d', 215),
 ('in', 204),
```

```
 ('that', 200),
 ('Caesar', 189),
 ('my', 188)]
```

В листинге 12.3 вы можете увидеть, что слово Caesar встречается в тексте 189 раз. Другие распространенные слова не дают особого понимания текста. Можно отфильтровать такие слова, как *the* и *is*. Для этого используйте корпус NLTK под названием `stopwords`. В листинге 12.4 показано, как его загрузить и отфильтровать эти слова, прежде чем создать новое частотное распределение.

Листинг 12.4. Фильтрация стоп-слов

```
nltk.download('stopwords')                        # Скачиваем корпус
from nltk.corpus import stopwords
[nltk_data] Downloading package stopwords to
[nltk_data] /Users/kbehrman/nltk_data...
[nltk_data] Unzipping corpora/stopwords.zip.

english_stopwords = stopwords.words('english')   # Загрузка английских стоп-слов
english_stopwords[:10]
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're"]

caesar_r = []
for word in caesar_w:
     if word not in string.punctuation:
         if word.lower() not in english_stopwords:
             caesar_r.append(word)   # Не пунктуация и не стоп-слова

len(caesar_w) - len(caesar_r)
14706

caesar_dist = nltk.FreqDist(caesar_r)
caesar_dist.most_common(15)
[('Caesar', 189),
 ('Brutus', 161),
 ('Bru', 153),
 ('haue', 128),
 ('shall', 107),
 ('Cassi', 107),
 ('thou', 100),
 ('Cassius', 85),
 ('Antony', 75),
 ('know', 66),
 ('Enter', 63),
 ('men', 62),
 ('vs', 62),
 ('man', 58),
 ('thee', 55)]
```

Список наиболее распространенных слов теперь дает вам больше информации о тексте: вы можете увидеть, какие символы упоминаются чаще всего. Неудивительно, что во главе списка Caesar и Brutus.

В листинге 12.5 рассмотрены некоторые методы класса `FreqDist`.

Листинг 12.5. Класс FreqDist

```
caesar_dist.max()          # Получаем слово с наибольшим количеством появлений
'Caesar'

caesar_dist['Cassi']       # Подсчитываем количество появлений определенного слова
107

caesar_dist.freq('Cassi') # Количество появлений слова, поделенное на общее количество
0.009616248764267098

caesar_dist.N()            # Получаем количество слов
11127

caesar_dist.tabulate(10)  # Выводим количество 10 самых встречаемых слов
Caesar  Brutus     Bru    haue   shall   Cassi   thou   Cassius   Antony   know
  189     161     153     128    107     107     100     85        75       66
```

В `FreqDist` есть встроенный метод построения графиков. В следующем примере отображен график наиболее часто встречающихся слов (рис. 12.1):

```
caesar_dist.plot(10)
```

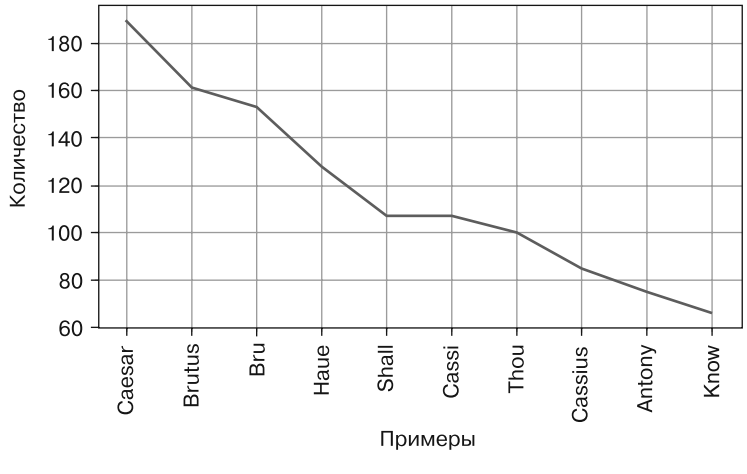


Рис. 12.1. Десять наиболее часто встречающихся слов

Текстовые объекты

Библиотека NLTK предлагает класс `Text`, предоставляющий функциональность, полезную на начальном этапе исследования нового текста. `Text` принимает список слов в качестве аргумента во время инициализации:

```
from nltk.text import Text
caesar_t = Text(caesar_w)
type(caesar_t)
nltk.text.Text
```

Метод `Text.concordance()` показывает контекст вокруг слова. Здесь он показывает пять примеров слова Antony в контексте:

```
caesar_t.concordance('Antony', lines=5)
Displaying 5 of 75 matches:
efulnesse . Exeunt . Enter Caesar , Antony for the Course , Calphurnia , Porti
Of that quicke Spirit that is in Antony : Let me not hinder Cassius your de
He loues no Playes , As thou dost Antony : he heares no Musicke ; Seldome he
r ' d him the Crowne ? Cask . Why Antony Bru . Tell vs the manner of it , ge
I did not marke it . I sawe Marke Antony offer him a Crowne , yet ' twas not
```

Метод `Text.collocations()` отображает слова, наиболее часто встречающиеся вместе:

```
caesar_t.collocations(num=4)
Mark Antony; Marke Antony; Good morrow; Caius Ligarius
```

Метод `Text.similar()` находит слова, появляющиеся в контекстах, похожих на данное слово:

```
caesar_t.similar('Caesar')
me it brutus you he rome that cassius this if men worke him vs feare world thee
```

Метод `Text.findall()` выводит текст, совпадающий с регулярным выражением для поиска текста. Определить шаблон соответствия регулярным выражениям можно с помощью < и > для определения границ слов и `.*` в качестве подстановочного знака, который может соответствовать чему угодно. Затем готовый шаблон сопоставляет все появления слова `O`, за которым следует любое слово, начинающееся с `C`:

```
caesar_t.findall(r'<O><C.*>')
O Cicero; O Cassius; O Conspiracie; O Caesar; O Caesar; O Caesar; O
Constancie; O Caesar; O Caesar; O Caesar; O Cassius; O Cassius; O
Cassius; O Coward; O Cassius; O Clitus
```

Метод `Text.dispersion_plot()` позволяет проверить, где в тексте встречаются заданные слова (рис. 12.2):

```
caesar_t.dispersion_plot(['Caesar', 'Antony', 'Brutus', 'Cassi'])
```

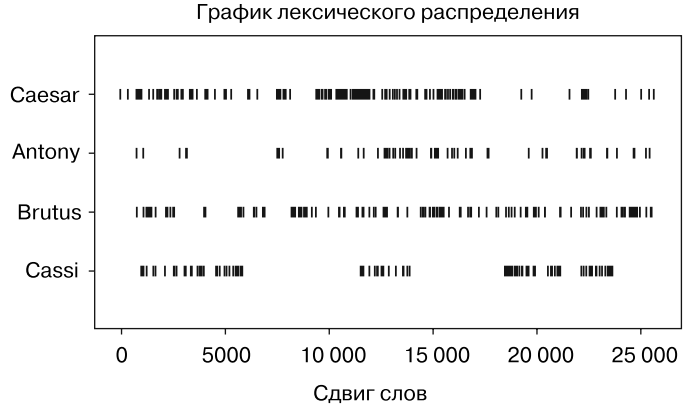


Рис. 12.2. Результаты метода Text. dispersion_plot()

Классификация текста

В NLTK есть классы-классификаторы, применяющие разные алгоритмы для работы с маркированием текстовых данных. Обычно, чтобы создать модель для классификации текста, нужно подготовить набор признаков в паре с категорией или меткой. В этом разделе рассматривается простой пример, использующий корпус Brown, доступный в NLTK (http://korpus.uib.no/icame/brown/bcm.html). У этого корпуса есть предварительно категоризированные тексты.

Допустим, вы считаете, что можете пометить абзац одного из этих текстов либо как редакционный, либо как художественный, основываясь на появлении определенных слов, на которые указывает переменная `tell_words`:

```
tell_words = ['american', 'city', 'congress', 'country', 'county',
              'editor', 'fact', 'government', 'national', 'nuclear',
              'party', 'peace', 'political', 'power', 'president',
              'public', 'state', 'states', 'united', 'war',
              'washington', 'world', 'big', 'church', 'every', 'eyes',
              'face', 'felt', 'found', 'god', 'hand', 'head', 'home',
              'house', 'knew', 'moment', 'night', 'room', 'seemed',
              'stood', 'think', 'though', 'thought', 'told', 'voice']
```

В листинге 12.6 показано, как загрузить корпусы, которые вы будете использовать, и найти абзацы для редакционной и художественной категорий.

Листинг 12.6. Загрузка совокупностей текстов

```
nltk.download('brown')                # Загружаем корпус
[nltk_data] Downloading package brown to /Users/kbehrman/nltk_data...
[nltk_data] Unzipping corpora/brown.zip.

from nltk.corpus import brown
nltk.download('stopwords')
from nltk.corpus import stopwords
english_stopwords = stopwords.words('english')

ed_p = brown.paras(categories='editorial') # Загрузка абзацев редакционного текста

fic_p = brown.paras(categories='fiction') # Загрузка абзацев художественного текста

print(len(ed_p))
1003

print(len(fic_p))
1043
```

Формат предоставленных абзацев — это списки списков, с подсписками, представляющими предложения. Предположим, вам нужен набор слов для каждого абзаца. В листинге 12.7 сначала объявлен метод сглаживания, а после в каждом наборе данных сглаживаются абзацы.

Листинг 12.7. Сглаживание вложенных списков

```
def flatten(paragraph):
    output = set([]) # Используем множество, так как требуется
                     # только одно появление слова
    for item in paragraph:
        if isinstance(item, (list, tuple)): # Добавляемый элемент — список или кортеж
            output.update(item)
        else:
            output.add(item)              # Добавление элемента
    return output

ed_flat = []
for paragraph in ed_p:
    ed_flat.append(flatten(paragraph))    # Сглаживание абзацев редакционного текста
fic_flat = []
for paragraph in fic_p:
    fic_flat.append(flatten(paragraph))   # Сглаживание абзацев художественного текста
```

Далее попарно соедините каждый абзац с меткой на основании его исходной категории. В листинге 12.8 это делается как для редакционных, так и для художественных текстов, а затем порядок перетасовывается с помощью метода `shuffle` из модуля `random`, чтобы убедиться, что он не повлияет на классификатор.

Листинг 12.8. Маркирование данных

```
labeled_data = []
for paragraph in ed_flat:
    labeled_data.append((paragraph, 'editorial'))

for paragraph in fic_flat:
    labeled_data.append((paragraph, 'fiction'))

from random import shuffle
shuffle(labeled_data)
```

Классификатор не использует исходные абзацы, а скорее ожидает набор признаков. Этот набор признаков будет представлен в виде словаря, сопоставляющего признаки со значениями. В листинге 12.9 объявлена функция для создания словаря признаков, чьи значения установлены как `True`, если названное слово найдено в абзаце, и как `False` — если нет. Та же функция применяется и для составления списка парных признаков и меток. Эта информация разбивается на обучающие и тестовые данные, чтобы вы могли обучить ваш классификатор.

Листинг 12.9. Определение признаков

```
def define_features(paragraph):
    features = {}
    for tell_word in tell_words:
        features[tell_word] = tell_word in paragraph
    return features

feature_data = []
for labeled_paragraph in labeled_data:
    paragraph, label = labeled_paragraph
    feature_data.append((define_features(paragraph), label,))

train_data = feature_data[:1400]
test_data = feature_data[1400:]
```

В листинге 12.10 показано, как обучать модель с помощью класса `nltk.NaiveBayesClassifier`, а после этого использовать ее для классификации отдельного набора признаков, проверить, какие из названных слов сильнее всего повлияли на обучение, а затем проверить точность с помощью тестовых данных.

Листинг 12.10. Обучение и проверка модели

```
bayes = nltk.NaiveBayesClassifier.train(train_data) # Обучение модели
bayes.classify(train_data[0][0]) # Классифицируйте один из абзацев обучающего набора
'fictio'

bayes.show_most_informative_features()
Most Informative Features
                    knew = True             fictio : editor =     22.3 : 1.0
                  editor = True             editor : fictio =     16.6 : 1.0
                   stood = True             fictio : editor =     16.0 : 1.0
               political = True             editor : fictio =     14.5 : 1.0
                 nuclear = True             editor : fictio =     12.4 : 1.0
              government = True             editor : fictio =     10.8 : 1.0
                 thought = True             fictio : editor =     10.2 : 1.0
                  seemed = True             fictio : editor =      7.0 : 1.0
                national = True             editor : fictio =      6.6 : 1.0
                  public = True             editor : fictio =      6.5 : 1.0

nltk.classify.accuracy(bayes, test_data)  # Проверка точности
0.6842105263157895
```

Вы можете видеть, что точность модели в прогнозировании меток для тестовых данных примерно 66 % — выше, чем при подбрасывании монеты.

Этот пример должен дать вам представление об использовании классификатора NLTK, но у него есть гораздо больше функций, чем было рассмотрено в этой главе. Чтобы подробнее узнать об обработке естественного языка с помощью NLTK, почитайте книгу «Обработка естественного языка с Python», авторы которой являются создателями библиотеки NLTK (http://www.nltk.org/book).

Резюме

В библиотеке NLTK есть инструменты для обработки текста, и она поставляется вместе с образцами текста, которые вы можете загрузить. Класс `FreqDist` позволяет получить представление о частоте появления разных слов. Класс `Text` предоставляет удобный способ для изучения нового текста. В NLTK есть встроенные классы-классификаторы, которые можно использовать для категоризации текста на основе обучающих данных.

Упражнения

1. Загрузите текст романа «Эмма» Джейн Остин как слова, предложения и абзацы.
2. Подсчитайте частоту появления слова *Алиса* в «Алисе в Стране Чудес» Льюиса Кэрролла.
3. Используйте `tabulate` для отображения топ-10 самых распространенных слов в «Алисе в Стране Чудес», исключая знаки препинания и стоп-слова.
4. Найдите слова, похожие на слово *кролик,* в «Алисе в Стране Чудес».
5. Используйте корпус `names`, чтобы найти десять наиболее часто встречающихся имен в «Гамлете».

ЧАСТЬ III

СРЕДНИЙ УРОВЕНЬ PYTHON

13

Функциональное программирование

Управление сложностью —
суть компьютерного программирования.

Брайан Кернига́н

В этой главе

- Знакомство с функциональным программированием.
- Состояние и область видимости.
- Функциональные функции.
- Списковые включения.
- Генераторы.

Как вы уже знаете, программа Python состоит из серии операторов, которые могут быть простыми или составными. Способы их организации бывают разными и влияют на производительность, читабельность и простоту модификации. Самые широко распространенные из них: процедурное, функциональное и объектно-ориентированное программирование. В этой главе вы познакомитесь с некоторыми понятиями функционального программирования, в том числе с включениями и генераторами, которые заимствованы из истинно функциональных языков.

Знакомство с функциональным программированием

Функциональное программирование основано на математическом определении функций. Функция в этом смысле преобразует входные данные в выходные. Для любого ввода может быть только один вывод. Иначе говоря, выходные данные для отдельных входных всегда одинаковы. Некоторые языки программирования (Haskell и Erlang) строго придерживаются этого ограничения. Python же достаточно гибок, чтобы адаптировать функциональные понятия без подобных ограничений. Функциональное программирование в Python иногда называется *функциональным лайт-программированием*.

Область видимости и состояние

Состояние программы включает имена, объявления и значения, которые есть в конкретное время в этой программе, в том числе объявления функций, импортируемые модули и присваиваемые переменным значения.

У состояния есть так называемая *область видимости* — часть программы, для которой удерживается состояние. Области иерархичны. Когда вы делаете отступ в блоке кода, у него есть вложенная область видимости. Она наследует область от окружающего кода без отступов, но не меняет напрямую внешнюю область видимости.

В листинге 13.1 установлены значения для переменных `a` и `b` во внешней области видимости. Далее блок кода функции устанавливает для `a` другое значение и выводит обе переменные. Можно увидеть, что, когда функция вызвана, она использует собственное объявление переменной `a`, но наследует объявление для `b` из внешней области видимости. Во внешней области значение, присвоенное `a`, игнорируется, так как оно находится вне области видимости.

Листинг 13.1. Наследование области видимости

```
a = 'a outer'
b = 'b outer'
```

```
def scoped_function():
    a = 'a inner'
    print(a)
    print(b)

scoped_function()
a inner
b outer

print(a)
a outer

print(b)
b outer
```

Зависимость от глобального состояния

До текущего момента код в этой книге был представлен в основном с помощью процедурного подхода. В этом подходе текущее состояние определяется операторами, которые выполнялись в строках, предшествующих текущей. Это состояние передается через программу и изменяется на всем ее протяжении. Значит, у функции, использующей состояние для определения своих выходных данных, могут быть другие выходные данные с теми же входными. Рассмотрим несколько примеров, сравнивающих процедурный подход с функциональным.

В листинге 13.2 создана функция `describe_the_wind()`, которая возвращает предложение с помощью переменной `wind`, объявленной во внешней области видимости. Здесь видно, что выходные данные этой функции будут различаться в зависимости от этой переменной.

Листинг 13.2. Зависимость от внешней области видимости

```
wind = 'Southeast'

def describe_the_wind():
    return f'The wind blows from the {wind}'

describe_the_wind()
'The wind blows from the Southeast'

wind = 'North'
describe_the_wind()
'The wind blows from the North'
```

Более функциональный подход — передать переменную в качестве аргумента. Тогда функция будет возвращать одно и то же значение для переданного ей, вне зависимости от внешней области видимости:

```
def describe_the_wind(wind):
    return f'The wind blows from the {wind}'

describe_the_wind('Northeast')
'The wind blows from the Northeast'
```

Изменение состояния

Помимо того, что функциональная функция не должна полагаться на внешнее состояние, она не должна напрямую изменять его. Листинг 13.3 показывает программу, изменяющую внешнее состояние переменной `WIND` внутри функции `change_wind()`. Обратите внимание на использование ключевого слова `global`, указывающего на изменение переменной внешнего состояния, а не на объявление новой переменной во внутреннем.

Листинг 13.3. Изменение внешней области видимости

```
WINDS = ['Northeast', 'Northwest', 'Southeast', 'Southwest']
WIND = WINDS[0]

def change_wind():
    global WIND
    WIND = WINDS[(WINDS.index(WIND) + 1)%3]

WIND
'Northeast'

change_wind()
WIND
'Northwest'

for _ in WINDS:
    print(WIND)
    change_wind()
Northwest
Southeast
Northeast
Northwest
```

Более функциональный подход к получению того же вывода состоит в том, чтобы переместить переменную `winds` во внутреннее состояние и заставить функцию `change_wind()` принимать аргумент для определения вывода:

Листинг 13.4. Неизменение внешней области видимости

```
def change_wind(wind_index):
    winds = ['Northeast', 'Northwest', 'Southeast', 'Southwest']
    return winds[wind_index]

print( change_wind(0) )
Northeast

print( change_wind(1) )
Northwest

print( change_wind(2) )
Southeast

print( change_wind(3) )
Southwest
```

Изменение изменяемых данных

Более изысканный способ изменения внешнего состояния — передача изменяемых объектов. Напомню, что это такие объекты, как списки и словари, чье содержимое можно изменить. Если вы установите переменную во внешнем состоянии, передадите ее в качестве аргумента в функцию, а затем измените ее значение во внутреннем состоянии функции, версия внешнего состояния переменной сохранит свое исходное значение:

```
b = 1

def foo(a):
    a = 2

foo(b)
print(b)
1
```

Но если вы передадите в функцию изменяемый объект, такой как словарь, в качестве аргумента, любое изменение этого объекта в ней будет отражаться и во внешнем состоянии. В следующем примере объявляется функция, которая принимает словарь в качестве аргумента и изменяет одно из его значений:

```
d = {"vehicle": "ship", "owner": "Joseph Bruce Ismay"}

def change_mutable_data(data):
    '''A function which changes mutable data.'''
    data['owner'] = 'White Star Line'

change_mutable_data(d)
print(d)
{'vehicle': 'ship', 'owner': 'White Star Line'}
```

Как видите, словарь `d`, переданный этой функции, изменил свое значение во внешнем состоянии.

Такое изменение внешней области видимости изменяемых объектов может привести к трудноуловимым ошибкам. Один из способов избежать этого, если ваша структура данных не слишком объемна, — сделать копию во внутренней области видимости, и управлять уже ею:

```
d = {"vehicle": "ship", "owner": "Joseph Bruce Ismay"}

def change_owner(data):
    new_data = data.copy()
    new_data['owner'] = 'White Star Line'
    return new_data

changed = change_owner(d)
changed
{'owner': 'White Star Line', 'vehicle': 'ship'}
```

При работе с копией гораздо проще увидеть, где изменяются значения.

Функции функционального программирования

Три из встроенных в Python функций пришли из функционального программирования: `filter()`, `map()` и `reduce()`.

`map()` применяется к последовательности значений и возвращает объект `map`. Входная последовательность может быть любого итерируемого типа, то есть любой объект, что может быть итерируемым, например последовательность Python. Возвращенный объект `map` тоже итерируемый, поэтому его можно перебирать или привести в форму списка для отображения результатов:

```
def grow_flowers(d):
    return d * "❀"
```

```
gardens = map(grow_flowers, [0,1,2,3,4,5])
type(gardens)
map

list(gardens)
['', '❀', '❀❀', '❀❀❀', '❀❀❀❀', '❀❀❀❀❀']
```

Можно сделать `map()` функцией, принимающей множество аргументов и предоставляющей множество последовательностей входных значений:

```
l1 = [0,1,2,3,4]
l2 = [11,10,9,8,7,6]

def multi(d1, d2):
    return d1 * d2

result = map(multi, l1, l2)
print( list(result) )
 [0, 10, 18, 24, 28]
```

Обратите внимание, что здесь одна из последовательностей входных значений длиннее другой. Функция `map()` останавливается по достижении конца более короткой входной последовательности.

Функция `reduce()` тоже принимает функцию и итерируемый объект в качестве аргументов. Она использует функцию для возврата одного значения на основе входных данных. Например, вычесть сумму из баланса бюджета можно с помощью цикла `for`:

```
initial_balance = 10000
debits = [20, 40, 300, 3000, 1, 234]

balance = initial_balance

for debit in debits:
    balance -= debit

balance
6405
```

Того же результата можно достичь с помощью функции `reduce()`:

```
from functools import reduce

inital_balance = 10000
debits = [20, 40, 300, 3000, 1, 234]
```

```
def minus(a, b):
    return a - b

balance = reduce(minus, debits, initial_balance)
balance
6405
```

Модуль `operator` предоставляет все стандартные операторы в виде функций, включая функции для стандартных математических операций. Вы можете использовать функцию `operator.sub()` в качестве аргумента для `reduce()` в виде замены функции `minus()`:

```
from functools import reduce
import operator

initial_balance = 10000
debits = [20, 40, 300, 3000, 1, 234]

reduce(operator.sub, debits, initial_balance)
6405
```

`filter()` принимает функцию и итерируемый объект в качестве аргументов. В зависимости от каждого элемента функция должна возвращать `True` или `False`. В результате мы получаем итерируемый объект, содержащий только входные значения, для которых функция возвращает `True`. Например, чтобы получить из строки только заглавные буквы, вы можете объявить функцию, которая проверяет, является ли символ заглавным, и передать его и строку в `filter()`:

```
charles = 'ChArlesTheBald'

def is_cap(a):
    return a.isupper()

retval = filter(is_cap, charles)
list(retval)
['C', 'A', 'T', 'B']
```

Один из немногих случаев, когда я действительно рекомендую использовать лямбда-функции, — при применении `map()`, `filter()`и `reduce()`. При проведении простых сравнений, например для всех чисел меньше 10 и больше 3, используйте лямбда-функцию и `range()` понятным и простым для чтения способом:

```
nums = filter(lambda x: x > 3, range(10))
list(nums)
 [4, 5, 6, 7, 8, 9]
```

Списковые включения

Списковые включения — это синтаксис, заимствованный из функционального языка программирования Haskell (https://docs.python.org/3/howto/functional.html). Haskell — полностью функциональный язык программирования, реализованный с синтаксисом, подходящим для чисто функционального подхода. Вы можете рассматривать списковое включение как однострочный цикл `for`, возвращающий список. Хотя источник списковых включений — в функциональном программировании, их использование стало стандартным во всех подходах Python.

Базовый синтаксис списковых включений

Базовый синтаксис для спискового включения следующий:

```
[ \<возвращаемый элемент\> for \<исходный элемент\> in \<итерируемое\> ]
```

Например, для списка имен, где вы хотите изменить первые буквы имен на заглавные, используйте `x.title()` в качестве возвращаемого элемента, а каждое имя — в качестве исходного:

```
names = ['tim', 'tiger', 'tabassum', 'theodora', 'tanya']
capd = [x.title() for x in names]
capd
['Tim', 'Tiger', 'Tabassum', 'Theodora', 'Tanya']
```

Ниже представлен эквивалентный процесс с использованием цикла `for`:

```
names = ['tim', 'tiger', 'tabassum', 'theodora', 'tanya']
capd = []

for name in names:
    capd.append(name.title())

capd
['Tim', 'Tiger', 'Tabassum', 'Theodora', 'Tanya']
```

Замена map и filter

Вы можете использовать списковые включения для замены функций `map()` и `filter()`. Например, следующий код сопоставляет числа от 0 до 5 с функцией, вставляющей их в строку:

```
def count_flower_petals(d):
    return f"{d} petals counted so far"

counts = map(count_flower_petals, range(6))

list(counts)
['0 petals counted so far',
 '1 petals counted so far',
 '2 petals counted so far',
 '3 petals counted so far',
 '4 petals counted so far',
 '5 petals counted so far']
```

Этот код можно заменить следующим, более простым, списковым включением:

```
[f"{x} petals counted so far" for x in range(6)]
['0 petals counted so far',
 '1 petals counted so far',
 '2 petals counted so far',
 '3 petals counted so far',
 '4 petals counted so far',
 '5 petals counted so far']
```

Можете добавить условный оператор в списковое включение, используя следующий синтаксис:

```
[ \<возвращаемый элемент \> for \<исходный элемент\> in \<итерируемое\> if \<условие\> ]
```

С ним вы можете легко воспроизвести функциональные возможности `filter()`. Например, следующий образец `filter()` возвращает только заглавные буквы:

```
characters = ['C', 'b', 'c', 'A', 'b', 'P', 'g', 'S']
def cap(a):
    return a.isupper()

retval = filter(cap, characters)

list(retval)
['C', 'A', 'P', 'S']
```

Эту функцию можно заменить следующим списковым включением, использующим условие:

```
characters = ['C', 'b', 'c', 'A', 'b','P', 'g', 'S']
[x for x in characters if x.isupper()]
['C', 'A', 'P', 'S']
```

Множественные переменные

Если элементы в исходном итерируемом объекте являются последовательностями, можете распаковать их с помощью множественных переменных:

```
points = [(12, 3), (-1, 33), (12, 0)]

[ f'x: {x} y: {y}' for x, y in points ]
['x: 12 y: 3', 'x: -1 y: 33', 'x: 12 y: 0']
```

Вы можете выполнить эквивалент вложенных циклов `for` с помощью множества операторов `for` в одном списковом включении:

```
list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]

[x for y in list_of_lists for x in y]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Словарные включения

Словарные включения используют синтаксис, аналогичный синтаксису списковых включений. Но, в отличие от добавления единичного значения в список, в словарь вы добавляете пару ключ/значение. В этом примере используются значения в двух списках для построения словаря:

```
names = ['James', 'Jokubus', 'Shaemus']
scores = [12, 33, 23]

{ name:score for name in names for score in scores}
{'James': 23, 'Jokubus': 23, 'Shaemus': 23}
```

Генераторы

Одно из главных преимуществ использования объекта `range` по сравнению с объектом `list` при работе с большими числовыми диапазонами в том, что `range` вычисляет результаты по запросу. Это значит, что объем его памяти неизменно мал. Генераторы позволяют вам использовать собственные вычисления для создания значений по запросу, работая при этом аналогично объектам диапазона.

Выражения-генераторы

Один из способов создания генератора — применение выражений-генераторов, использующих тот же синтаксис, что и списковые включения, за исключением того, что квадратные скобки заменяются круглыми. В этом примере показано, как создать список и генератор, основанные на одном и том же вычислении, и вывести их:

```
l_ten = [x**3 for x in range(10)]
g_ten = (x**3 for x in range(10))

print(f"l_ten is a {type(l_ten)}")
l_ten is a <class 'list'>

print(f"l_ten prints as: {l_ten}")
l_ten prints as: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

print(f"g_ten is a {type(g_ten)}")
g_ten is a <class 'generator'>

print(f"g_ten prints as: {g_ten}")
g_ten prints as: <generator object <genexpr> at 0x7f3704d52f68>
```

При выводе списка можно увидеть его содержимое. В случае с генератором это не так. Чтобы получить значение из него, нужно запросить следующее. Это можно сделать с помощью функции `next()`:

```
next(g_ten)
0
```

Или, что более распространено, можно выполнить итерацию генератора в цикле `for`:

```
for x in g_ten:
    print(x)
1
8
27
64
125
216
343
512
729
```

Поскольку генераторы формируют значения только по запросу, их нельзя индексировать или срезать:

```
g_ten[3]
---------------------------------------------------------------------------
                                  TypeError Traceback (most recent call last)
<ipython-input-6-e7b8f961aa33> in <module>()
      1
----> 2 g_ten[3]

TypeError: 'generator' object is not subscriptable
```

Одно из важных преимуществ генераторов по сравнению со списками — объем занимаемой памяти. Ниже используется функция `sys.getsizeof()` для сравнения размеров списка и генератора:

```
import sys
x = 100000000
l_big = [x for x in range(x)]
g_big = (x for x in range(x))

print( f"l_big is {sys.getsizeof(l_big)} bytes")
l_big is 859724472 bytes

print( f"g_big is {sys.getsizeof(g_big)} bytes")
g_big is 88 bytes
```

Функции-генераторы

Функции-генераторы можно использовать для создания сложных генераторов. Эти функции кажутся обычными, но в них оператор `return` заменен на `yield`. Генератор сохраняет свое внутреннее состояние, возвращая значения по запросу:

```
def square_them(numbers):
    for number in numbers:
        yield number * number

s = square_them(range(10000))

print(next(s))
0

print(next(s))
1
```

```
print(next(s))
4

print(next(s))
9
```

Еще одно преимущество генераторов перед списками — их возможность создавать бесконечный генератор. Он возвращает столько значений, сколько было запрошено. Например, можно создать генератор, который инкрементирует число столько раз, сколько вы хотите:

```
def counter(d):
    while True:
        d += 1
        yield d

c = counter(10)

print(next(c))
11

print(next(c))
12

print(next(c))
13
```

В листинге 13.5 объединены четыре генератора. Это полезный способ сделать каждый из них понятным, используя их своевременные вычисления.

Листинг 13.5. Конвейер генераторов

```
evens = (x*2 for x in range(5000000))
three_factors = (x//3 for x in evens if x%3 == 0)
titles = (f"this number is {x}" for x in three_factors)
capped = (x.title() for x in titles)

print(f"The first call to capped: {next(capped)}")
The first call to capped: This Number Is 0

print(f"The second call to capped: {next(capped)}")
The second call to capped: This Number Is 2

print(f"The third call to capped: {next(capped)}")
The third call to capped: This Number Is 4
```

Использование генератора — отличный способ повысить производительность вашего кода. Старайтесь применять его всякий раз, когда производите итерацию длинной последовательности вычисляемых значений.

Резюме

Функциональное программирование — это подход к организации программ, полезный для разработки ПО, которое может выполняться параллельно. В его основе — идея о том, что внутреннее состояние функции должно меняться внешним или само изменять внешнее состояние вызывающего ее кода. Функция всегда должна возвращать одинаковое значение для заданного входного значения. Три встроенные в Python функции из мира функционального программирования — `map()`, `filter()` и `reduce()`. Использование списковых включений и генераторов — типичные для Python способы создания последовательностей значений. Использование генераторов рекомендовано при итерации любого большого количества значений или когда вы не знаете, сколько значений вам нужно.

Вопросы для закрепления

1. Что выведет следующий код?

```
a = 1
b = 2

def do_something(c):
    c = 3
    a = 4
    print(a)
    return c

b = do_something(b)
print(a + b)
```

2. Используйте функцию `map()`, чтобы принять строку `omni` и вернуть список `['oo','mm', 'nn', 'ii']`.
3. Используйте функцию `sum()`, суммирующую содержимое последовательности, со списковым включением, чтобы найти сумму положительных четных чисел меньше 100.
4. Напишите выражение-генератор, которое возвращает числа в кубе до 1000.
5. Последовательность Фибоначчи начинается с 0 и 1, а каждое последующее число — сумма предыдущих двух. Напишите функцию-генератор, которая вычисляет последовательность Фибоначчи.

14

Объектно-ориентированное программирование

Любой дурак может написать код,
понятный компьютеру.
Хороший программист напишет код,
который поймет человек.

Мартин Фаулер

В этой главе

- Связывание состояния и функции.
- Классы и объекты.
- Специальные функции.
- Наследование класса.

Объектно-ориентированный подход к программированию — один из наиболее популярных. Он заключается в попытках моделировать объекты и их взаимосвязи, сочетая функциональные возможности и данные.

Допустим, вы захотите смоделировать в коде машину. В объектно-ориентированном подходе как раз есть методы, выполняющие такие действия, как ускорение и торможение, и содержащие такие данные, как количество топлива в бензобаке, прикрепленные к одному и тому же объекту. Другие подходы хранили бы данные (например, количество бензина) отдельно от объявлений функции, передавая их в качестве аргументов. Огромное преимущество объектно-ориентированного подхода — способность создавать понятные для человека представления сложных систем.

Связывание состояния и функции

В отличие от функционального подхода, в объектно-ориентированном данные и функциональность объединяются в комплекты, известные как *объекты*. В Python все является объектом, даже у базовых типов есть как методы, так и данные. Например, объект `int` содержит не только значение, но и методы. Один из них — метод `to_bytes()`, преобразующий значения в их байтовые представления:

```
my_num = 13
my_num.to_bytes(8, 'little')
b'\r\x00\x00\x00\x00\x00\x00\x00'
```

Более сложные типы данных, такие как списки, строки, словари и датафреймы Pandas, все сочетают в себе данные и функционал. Присоединенная к объекту функция в Python называется *методом*. Сила объектно-ориентированных возможностей Python в том, что вы можете использовать объекты из предоставленных библиотек и создавать собственные.

Классы и экземпляры

Объекты объявляются классами. Воспринимайте класс как шаблон для объекта. При создании экземпляра класса вы получаете объект этого типа класса. Синтаксис для создания базового объявления класса следующий:

```
class <class name>():
    <statement>
```

Используйте оператор `pass` для объявления простого класса, который ничего не делает:

```
class DoNothing():
    pass
```

Синтаксис для создания экземпляра класса следующий:

```
<class name>()
```

Чтобы создать экземпляр с именем `do_nothing` из класса `DoNothing`, создайте экземпляр объекта:

```
do_nothing = DoNothing()
```

Если вы проверите тип объекта

```
type(do_nothing)
__main__.DoNothing
```

то увидите, что он нового типа, определенного `DoNothing`. Это можно подтвердить с помощью встроенной функции `isinstance()`, проверяющей, является ли объект экземпляром определенного класса:

```
isinstance(do_nothing, DoNothing)
True
```

Наиболее распространенный способ объявления метода, присоединенного к классу, — отступ в объявлении функции во внутренней области видимости класса с помощью следующего синтаксиса:

```
class <CLASS NAME>():
    def <FUNCTION NAME>():
        <STATEMENT>
```

Первый аргумент функции — экземпляр класса, из которого она вызывается (`self`). В следующем примере объявлен класс `DoNothing` с методом `return_self()`, который возвращает `self`, создает экземпляр класса и показывает, что возвращаемое значение `return_self()` на самом деле и есть сам экземпляр класса:

```
class DoSomething():
    def return_self(self):
        return self
```

```
do_something = DoSomething()

do_something == do_something.return_self()
True
```

ПРИМЕЧАНИЕ

Хотя необходимо иметь self в качестве параметра в объявлении метода, когда вы вызываете метод, вы не указываете этот параметр, так как он передается автоматически «за кулисами».

За пределами параметра `self` можно объявить методы так же, как и при работе с другими функциями. Вы также можете использовать `self` для создания объектных переменных и доступа к ним внутри объявления класса с помощью следующего синтаксиса:

```
self.<VARIABLE NAME>
```

Точно так же методы и атрибуты могут быть присоединены к объекту, экземпляр которого создан из класса:

```
class AddAttribute():
    def add_score(self):
        self.score = 14

add_attribute = AddAttribute()
add_attribute.add_score()

add_attribute.score
14
```

Для вызова одного метода из другого в одном и том же классе используйте следующий синтаксис:

```
self.<METHOD NAME>
```

В листинге 14.1 показано, как вызвать один метод из другого для одного класса.

Листинг 14.1. Внутренний вызов методов

```
class InternalMethodCaller():
    def method_one(self):
        print('Calling method one')

    def method_two(self, n):
        print(f'Method two calling method one {n} times')
        for _ in range(n):
            self.method_one()
```

```
internal_method_caller = InternalMethodCaller()
internal_method_caller.method_one()
Calling method one

internal_method_caller.method_two(2)
Method two calling method one 2 times
Calling method one
Calling method one
```

Закрытые методы и переменные

Наличие доступа к объекту означает доступ к его методам и переменным. Методы и переменные, которые вы видели ранее, известны как *публичные* — они предоставляют данные и функционал для прямого использования. Иногда при объявлении класса нужно выделить переменные или методы, которые вы не хотите использовать напрямую. Они известны как *приватные* атрибуты, и детали их реализации могут изменяться по мере развития класса. Приватные атрибуты используются публичными методами внутренне. У Python нет механизма для предотвращения доступа к ним, но имя приватного атрибута обычно начинается с нижнего подчеркивания, как в следующем примере:

```
class PrivatePublic():
    def _private_method(self):
        print('private')

    def public_method(self):
        # Вызов приватного метода
        self._private_method()
        # ... Другие команды
```

Переменные класса

Переменные, которые объявляются с помощью синтаксиса `self.<ИМЯ ПЕРЕМЕННОЙ>`, называются *переменными экземпляра*. Они связаны с отдельными экземплярами класса, у каждого из которых могут быть разные значения для его переменных. Переменные можно также связать с классом. *Переменные класса* общие для всех экземпляров этого класса.

В листинге 14.2 показан класс, содержащий как переменную класса, так и переменную экземпляра. Два экземпляра этого класса разделяют между собой данные переменной класса, но у каждого из них уникальные значения для переменной экземпляра. Обратите внимание, что переменная класса не привязана к объекту экземпляра `self`.

Листинг 14.2. Переменные класса и экземпляра

```
class ClassyVariables():
    class_variable = 'Yellow'

    def __init__(self, color):
        self.instance_variable = color

red = ClassyVariables('Red')
blue = ClassyVariables('Blue')

red.instance_variable
'Red'

red.class_variable
'Yellow'

blue.class_variable
'Yellow'

blue.instance_variable
'Blue'
```

Специальные методы

В Python некоторые специальные имена методов зарезервированы для определенных задач. К ним относятся методы для функциональности оператора и контейнера, а также инициализации объекта. Чаще всего из них используется метод `__init__()` — он вызывается каждый раз при создании экземпляра объекта и используется для установки исходных значений атрибута для него.

В листинге 14.3 объявлен класс `Initialized` с методом `__init__()`, принимающим дополнительный параметр `n`. После создания экземпляра этого класса укажите значение для этого параметра, после чего оно будет присвоено переменной `count`. К этой переменной можно будет получить доступ с помощью других методов класса (например, `self.count`) или из созданного экземпляра объекта (например, `<объект>.<атрибут>`).

Листинг 14.3. Метод __init__

```
class Initialized():
    def __init__(self, n):
        self.count = n

    def increment_count(self):
        self.count += 1
```

```
initialized = Initialized(2)
initialized.count
2

initialized.increment_count()
initialized.count
3
```

Методы представления

Методы `__repr__()` и `__str__()` используются для управления представлением объекта. `__repr__()` дает его техническое описание. В идеале оно включает информацию, необходимую для пересоздания объекта. Такое представление вы можете увидеть при использовании объекта в качестве оператора.

`__str__()` предназначен для определения менее строгого, но более удобного для пользователя представления. Это выходные данные при преобразовании объекта в строку, как это автоматически выполняется функцией `print()`. В листинге 14.4 показано использование методов `__repr__()` и `__str__()`.

Листинг 14.4. Методы __repr__() и __str__()

```
class Represented():
    def __init__(self, n):
        self.n = n

    def __repr__(self):
        return f'Represented({self.n})'

    def __str__(self):
        return 'Object demonstrating __str__ and __repr__'

represented = Represented(13)

represented
Represented(13)

r = eval(represented.__repr__())
type(r)
__main__.Represented

r.n
13

str(represented)
'Object demonstrating __str__ and __repr__'
```

```
print(represented)
Object demonstrating __str__ and __repr__
```

Расширенные методы сравнения

Расширенные методы сравнения применяются для определения поведения объекта, когда он используется встроенными в Python операторами. В листинге 14.5 показано, как объявить методы для разных операторов сравнения. Класс `CompareMe` использует переменную `score` для сравнений и возвращается к `time` только при необходимости.

Листинг 14.5. Методы сравнения

```
class CompareMe():
    def __init__(self, score, time):
        self.score = score
        self.time = time

def __lt__(self, O):
    """ Less than"""
    print('called __lt__')
    if self.score == O.score:
        return self.time > O.time
    return self.score < O.score

    def __le__(self, O):
        """Less than or equal"""
        print('called __le__')
        return self.score <= O.score

    def __eq__(self, O):
        """Equal"""
        print('called __eq__')
        return (self.score, self.time) == (O.score, O.time)

    def __ne__(self, O):
        """Not Equal"""
        print('called __ne__')
        return (self.score, self.time) != (O.score, O.time)

    def __gt__(self, O):
        """Greater Than"""
        print('called __gt__')
        if self.score == O.score:
            return self.time < O.time
        return self.score > O.score
```

```
    def __ge__(self, O):
        """Greater Than or Equal"""
        print('called __ge__')
        return self.score >= O.score
```

Ниже создается экземпляр класса `CompareMe` с разными значениями, а затем тестируются некоторые операторы сравнения.

Листинг 14.6. Испытание операторов

```
high_score = CompareMe(100, 100)
mid_score = CompareMe(50, 50)
mid_score_1 = CompareMe(50, 50)
low_time = CompareMe(100, 25)

high_score > mid_score
called __gt__
True

high_score >= mid_score_1
called __ge__
True

high_score == low_time
called __eq__
False

mid_score == mid_score_1
called __eq__
True

low_time > high_score
called __gt__
True
```

Также возможно объявить сравнения, сопоставляющие атрибут с объектом. В листинге 14.7 создан класс, напрямую сравнивающий свой атрибут `score` с другим объектом. Это позволяет сравнивать объект с любым типом, сопоставимым с `int` (другими словами, этот список реализует только методы «меньше» и «равно»).

Листинг 14.7. Сравнение с объектом

```
class ScoreMatters():

    def __init__(self, score):
        self.score = score

    def __lt__(self, O):
        return self.score < O
```

```
    def __eq__(self, O):
        return self.score == O

my_score = ScoreMatters(14)
my_score == 14.0
True

my_score < 15
True
```

Важно не объявлять запутанные или нелогичные сравнения в коде Python. Учитывайте в этих объявлениях конечного пользователя. Например, в листинге 14.8 объявлен класс, который при сравнении всегда больше чего угодно, даже самого себя. Это может сбить с толку конечного пользователя класса.

Листинг 14.8. Приводящий в замешательство класс

```
class ImAlwaysBigger():
    def __gt__(self, O):
        return True

    def __ge__(self, O):
        return True

i_am_bigger = ImAlwaysBigger()
no_i_am_bigger = ImAlwaysBigger()

i_am_bigger > "Anything"
True

i_am_bigger > no_i_am_bigger
True

no_i_am_bigger > i_am_bigger
True

i_am_bigger > i_am_bigger
True
```

Методы математических операторов

В Python есть специальные методы для математических операций. В листинге 14.9 объявлен класс, который реализует методы для операторов `+`, `-` и `*`. Он возвращает новые объекты на основе их переменной `value`.

Листинг 14.9. Выбранные математические операции

```
class MathMe():
    def __init__(self, value):
        self.value = value

    def __add__(self, O):
        return MathMe(self.value + O.value)

    def __sub__(self, O):
        return MathMe(self.value - O.value)

    def __mul__(self, O):
        return MathMe(self.value * O.value)

m1 = MathMe(3)
m2 = MathMe(4)
m3 = m1 + m2
m3.value
7

m4 = m1 - m3
m4.value
-4

m5 = m1 * m3
m5.value
21
```

Есть гораздо больше специальных методов, включая методы для побитовых операций и объявления контейнероподобных объектов, поддерживающих слайсинг. Их полный список вы найдете по ссылке https://docs.python.org/3/reference/datamodel.html#special-method-names.

Наследование

Наследование — один из самых важных и мощных концептов объектно-ориентированного программирования. При наследовании один класс объявляет другой класс (классы) родителем (-ями). Дочерний класс может использовать методы и переменные от своих родителей, как если бы они были описаны в его объявлении. В листинге 14.10 объявлен класс `Person`, а затем он становится родительским для другого класса `Student`.

Листинг 14.10. Базовое наследование

```
class Person():
```

```
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

class Student(Person):
    def introduce_yourself(self):
        print(f'Hello, my name is {self.first_name}')

barb = Student('Barb', 'Shilala')
barb.first_name
'Barb'

barb.introduce_yourself()
Hello, my name is Barb
```

Обратите внимание, что метод `Student.introduce_yourself()` использует переменную `Person.first_name`, как если бы она была объявлена частью класса `Student`. При проверке экземпляра класса вы увидите, что это `Student`:

```
type(barb)
__main__.Student
```

Помните: используя функцию `isinstance()`, вы можете увидеть, что экземпляр будет как экземпляром класса `Student`

```
isinstance(barb, Student)
True
```

так и экземпляром класса `Person`

```
isinstance(barb, Person)
True
```

Наследование полезно при написании кода, ожидающего некоторое общее поведение среди классов. Например, при реализации системы оркестратора бесконечных задач вы можете ожидать, что у каждого типа задания будет метод `run()`. Вместо тестирования всех возможных типов задач можно просто объявить родительский класс с помощью `run()`. Для любой задачи, наследующей от родительского класса и являющейся его экземпляром, будет объявлен метод `run()`, как в листинге 14.11.

Листинг 14.11. Тестирование на основной класс

```
class Job():
    def run(self):
        print("I'm running")
```

```
class ExtractJob(Job):
    def extract(self, data):
        print('Extracting')

class TransformJob(Job):
    def transform(self, data):
        print('Transforming')

job_1 = ExtractJob()
job_2 = TransformJob()
for job in [job_1, job_2]:
    if isinstance(job, Job):
        job.run()
I'm running
I'm running
```

Если дочерний класс объявляет переменную или метод с тем же именем, что и в его родительском классе, экземпляры дочернего будут использовать дочернее объявление. Допустим, вы объявили родительский класс с помощью `run()`:

```
class Parent():
    def run(self):
        print('I am a parent running carefully')
```

И установили дочерний класс, переобъявляющий метод:

```
class Child(Parent):
    def run(self):
        print('I am a child running wild')
```

Далее экземпляры дочернего класса будут использовать его объявления:

```
chile = Child()
chile.run()
I am a child running wild
```

Иногда полезно явно вызвать метод родительского класса. Например, нет ничего необычного в том, чтобы вызвать метод родительского класса `__init__()` изнутри метода дочернего класса `__init__()`. У функции `super()` есть доступ к родительскому классу и его атрибутам.

В следующем примере использована функция `super()` для вызова `Person.__init__()` из дочернего класса `Student`:

```
class person():
    def __init__(self, first_name, last_name):
```

```
        self.first_name = first_name
        self.last_name = last_name

class student(person):
    def __init__(self, school_name, first_name, last_name):
        self.school_name = school_name
        super().__init__(first_name, last_name)

lydia = student('boxford', 'lydia', 'smith')
lydia.last_name
'smith'
```

Наследование не ограничено одним родительским классом или уровнем. Класс может наследовать от другого класса, который сам наследует от еще одного:

```
class A():
    pass

class B(A):
    pass

class C(B):
    pass

c = C()
isinstance(c, B)
True

isinstance(c, A)
True
```

Класс может наследовать и от множества родителей:

```
class A():
    def a_method(self):
        print(A's method)

class B():
    def b_method(self):
        print(B's method)

class C(A, B):
    pass
```

```
c = C()
c.a_method()
A's method

c.b_method()
B's method
```

ПРИМЕЧАНИЕ

Я советую по возможности не строить слишком сложные деревья наследования. В сложном наследовании труднее исправлять ошибки, так как вы отслеживаете взаимодействие между переменными и методами, объявленными по всему дереву.

ПРИМЕЧАНИЕ

Есть много информации о дизайне объектно-ориентированного кода. Во избежание ошибок изучите его подробнее, прежде чем начинать крупный объектно-ориентированный проект.

Резюме

Объектно-ориентированное программирование включает группировку данных и функций в объектах, объявленных классами. Специальные методы позволяют объявлять те классы, которые будут работать с операторами Python, и те, которые реализуют поведение контейнера. Классы могут наследовать объявления от других классов.

Вопросы для закрепления

1. Что означает переменная `self` в объявлении класса?
2. Когда вызывается специальный метод `__init__()`?
3. Дано следующее объявление класса:

```
class Confuzed():
    def __init__(self, n):
        self.n = n

    def __add__(self, O):
        return self.n – O
```

Какой результат вы ожидаете от следующего кода?

```
c = Confuzed(12)
c + 12
```

4. Каковы выходные данные следующего кода?

```
class A():
    def say_hello(self):
        print('Hello from A')

    def say_goodbye(self):
        print('Goodbye from A')

class B(A):
    def say_goodbye(self):
        print('Goodbye from B')

b = B()
b.say_hello()
b.say_goodbye()
```

15

Прочие вопросы

Важнейшее свойство программы в том,
выполняет ли она намерение пользователя.

Чарльз Энтони Ричард Хоар

В этой главе

- Сортировка списков.
- Чтение и запись файлов.
- Объекты `datetime`.
- Регулярные выражения.

В этой главе рассматриваются некоторые компоненты стандартной библиотеки Python, которые являются мощными инструментами как для data science, так и для общего использования.

Сначала вы познакомитесь с разными способами сортировки данных, а затем перейдем к чтению и записи файлов с помощью контекстных менеджеров. После этого мы разберем представление времени с помощью объектов `datetime`. Наконец, в этой главе рассматривается поиск текста с использованием мощной библиотеки регулярных выражений.

Важно иметь хотя бы общее представление об этих темах, ведь все они широко используются в производственном программировании. Эта глава должна дать вам достаточное представление о них, чтобы вы могли использовать эту информацию, когда она вам понадобится.

Сортировка

У некоторых структур данных Python, таких как списки, массивы NumPy и датафреймы Pandas, есть встроенные возможности сортировки. Эти структуры данных можно использовать в готовом виде или настроить самостоятельно при помощи своих функций сортировки.

Списки

Для списков Python используйте встроенный метод `sort()`, сортирующий списки на месте. Допустим, вы объявляете список строк, представляющих китов:

```
whales = [ 'Blue', 'Killer', 'Sperm', 'Humpback', 'Beluga', 'Bowhead' ]
```

Используя метод `sort()` для этого списка следующим образом

```
whales.sort()
```

вы увидите, что список отсортирован в алфавитном порядке:

```
whales
['Beluga', 'Blue', 'Bowhead', 'Humpback', 'Killer', 'Sperm']
```

Этот метод не возвращает копию списка. Если вы захватите возвращаемое значение, то увидите, что оно равно `None`:

```
return_value = whales.sort()
print(return_value)
None
```

Чтобы создать отсортированную копию списка, используйте встроенную функцию Python `sorted()`, которая возвращает отсортированный список.

```
sorted(whales)
['Beluga', 'Blue', 'Bowhead', 'Humpback', 'Killer', 'Sperm']
```

`sorted()` можно использовать для любого итерируемого объекта, включая списки, строки, множества, кортежи и словари. Функция возвращает отсортированный список вне зависимости от типа объекта. Если вы вызываете ее в строке, она возвращает отсортированный список символов строки:

```
sorted("Moby Dick")
[' ', 'D', 'M', 'b', 'c', 'i', 'k', 'o', 'y']
```

И метод `list.sort()`, и функция `sorted()` принимают необязательный параметр `reverse`, который по умолчанию `False`:

```
sorted(whales, reverse=True)
['Sperm', 'Killer', 'Humpback', 'Bowhead', 'Blue', 'Beluga']
```

И `list.sort()`, и `sorted()` принимают необязательный аргумент `key`, который объявляет способ сортировки. Например, чтобы отсортировать китов по длине строки, можно объявить лямбду, которая возвращает длину строки, и передать ее в качестве ключа:

```
sorted(whales, key=lambda x: len(x))
['Blue', 'Sperm', 'Beluga', 'Killer', 'Bowhead', 'Humpback']
```

Вы можете определить и более сложные функции `key`. Следующий пример показывает, как определить функцию, которая возвращает длину строки, пока это не `'Beluga'`, — в этом случае она возвращает `1`. Это значит, что пока длина других строк больше 1, функция `key` будет сортировать список по длине строки, кроме значения `'Beluga'`, которое помещается первым:

```
def beluga_first(item):
    if item == 'Beluga':
        return 1
    return len(item)

sorted(whales, key=beluga_first)
['Beluga', 'Blue', 'Sperm', 'Killer', 'Bowhead', 'Humpback']
```

Вы можете использовать функцию `sorted()` с классами, которые объявляете. В листинге 15.1 объявлен класс `Food` и созданы четыре его экземпляра. Затем экземпляры класса сортируются с помощью атрибута `rating` в качестве ключа:

Листинг 15.1. Сортировка объектов с помощью лямбды

```
class Food():
    def __init__(self, rating, name):
        self.rating = rating
        self.name = name

    def __repr__(self):
        return f'Food({self.rating}, {self.name})'

foods = [Food(3, 'Banana'),
         Food(9, 'Orange'),
         Food(2, 'Tomato'),
         Food(1, 'Olive')]

foods
[Food(3, Banana), Food(9, Orange), Food(2, Tomato), Food(1, Olive)]

sorted(foods, key=lambda x: x.rating)
[Food(1, Olive), Food(2, Tomato), Food(3, Banana), Food(9, Orange)]
```

При вызове функции `sorted()` для словаря она вернет отсортированный список его ключевых имен. Начиная с версии Python 3.7 (https://docs.python.org/3/whatsnew/3.7.html), ключи словаря появляются в том порядке, в котором они были вставлены в словарь.

В листинге 15.2 создан словарь веса китов на основе данных с сайта https://www.whalefacts.org/how-big-are-whales/. Ключи словаря выводятся, чтобы показать, что они сохраняют порядок, в котором их вставляли. Далее используйте функцию `sorted()`, чтобы получить список ключевых имен, отсортированных по алфавиту, и вывести названия и вес китов по порядку.

Листинг 15.2. Сортировка ключей словаря

```
weights = {'Blue': 300000,
           'Killer': 12000,
           'Sperm': 100000,
           'Humpback': 78000,
           'Beluga': 3500,
           'Bowhead': 200000 }
```

```
for key in weights:
    print(key)
Blue
Killer
Sperm
Humpback
Beluga
Bowhead

sorted(weights)
['Beluga', 'Blue', 'Bowhead', 'Humpback', 'Killer', 'Sperm']

for key in sorted(weights):
    print(f'{key} {weights[key]}')
Beluga 3500
Blue 300000
Bowhead 200000
Humpback 78000
Killer 12000
Sperm 100000
```

В датафреймах Pandas есть метод сортировки `.sort_values()`, принимающий список имен столбцов, который может быть отсортирован (листинг 15.3).

Листинг 15.3. Сортировка датафреймов Pandas

```
import pandas as pd
data = {'first': ['Dan', 'Barb', Bob'],
        'last': ['Huerando', 'Pousin', 'Smith'],
        'score': [0, 143, 99]}

df = pd.DataFrame(data)
df

        first    last        score
0       Dan     Huerando        0
1       Bob     Pousin        143
2       Bob     Smith          99

df.sort_values(by=['last','first'])
        first    last        score
0       Bob     Pousin        143
1       Bob     Smith          99
2       Dan     Huerando        0
```

Чтение и запись файлов

Вы уже видели, что Pandas может читать разные файлы прямо в датафреймах. Но бывают случаи, когда нужно читать и записывать данные файла без использования Pandas. В Python есть встроенная функция `open()`, которая при заданном пути возвращает открытый файловый объект.

Следующий пример показывает, как я открываю конфигурационный файл из домашнего каталога (вы можете использовать любой путь файла тем же способом):

```
read_me = open('/Users/kbehrman/.vimrc')
read_me
<_io.TextIOWrapper name='/Users/kbehrman/.vimrc' mode='r' encoding='UTF-8'>
```

Прочесть одну строку из файлового объекта можно с помощью метода `.readline()`:

```
read_me.readline()
'set nocompatible\n'
```

Файловый объект отслеживает ваше местоположение в файле. С каждым вызовом метода `.readline()` следующая строка возвращается в таком виде:

```
read_me.readline()
'filetype off\n'
```

Важно закрыть подключение к файлу по окончании работы, иначе это может помешать открыть файл снова. Как вариант, сделать это с помощью функции `close()`:

```
read_me.close()
```

Контекстные менеджеры

Составной оператор контекстного менеджера позволяет автоматически закрывать файлы. Этот тип оператора начинается с ключевого слова `with` и закрывает файл, когда тот выходит из своего локального состояния.

В следующем примере файл открывается с помощью контекстного менеджера и читается с помощью метода `readlines()`:

```
with open('/Users/kbehrman/.vimrc') as open_file:
    data = open_file.readlines()

data[0]
'set nocompatible\n'
```

Содержимое файла считывается в виде списка строк и присваивается именованным данным переменной. Затем контекст завершается и файловый объект автоматически закрывается.

При открытии файла файловый объект по умолчанию готов к чтению в виде текста. Можно указать и другие режимы, например двоичное чтение (`'rb'`), запись (`'w'`) и двоичную запись (`'wb'`).

В следующем примере для записи нового файла использован аргумент `'w'`:

```
text = 'My intriguing story'

with open('/Users/kbehrman/my_new_file.txt', 'w') as open_file:
    open_file.write(text)
```

Так вы можете проверить, действительно ли был создан файл:

```
!ls /Users/kbehrman
Applications    Downloads     Movies     Public
Desktop         Google Drive  Music      my_new_file.txt
Documents       Library       Pictures   sample.json
```

JSON — это общепринятый формат для передачи и хранения данных. Стандартная библиотека Python включает модуль для перевода в JSON и из него. Этот модуль может осуществлять перевод между JSON и типами Python.

Этот пример показывает, как открыть и прочесть файл формата JSON:

```
import json

with open('/Users/kbehrman/sample.json') as open_file:
    data = json.load(open_file)
```

Объекты datetime

Данные, моделирующие значения во времени, называются *данными временных рядов* и обычно используются для задач data science. Для применения этого типа данных нужен способ представления времени. Один из них — использование строк. Если вам нужен более широкий функционал, например возможность легко добавить и удалить или извлечь данные по году, месяцу и дню, то потребуется что-то посложнее.

Библиотека `Datetime` предлагает разные способы моделирования времени наряду с полезными функциями для управления временными значениями. Класс `datetime.datetime()` представляет момент с точностью до микросекунды.

В листинге 15.4 показано, как создать объект `datetime` и получить доступ к некоторым его значениям.

Листинг 15.4. Атрибуты datetime

```
from datetime import datetime

dt = datetime(2022, 10, 1, 13, 59, 33, 10000)
dt
datetime.datetime(2022, 10, 1, 13, 59, 33, 10000)

dt.year
2022

dt.month
10

dt.day
1

dt.hour
13

dt.minute
59

dt.second
33

dt.microsecond
10000
```

С помощью функции `datetime.now()` можно получить объект для текущего времени:

```
datetime.now()
datetime.datetime(2021, 3, 7, 13, 25, 22, 984991)
```

Вы можете перевести строки в объекты `datetime` и наоборот с помощью функций `datetime.strptime()` и `datetime.strftime()`. Они обе полагаются на коды форматов, определяющих способ обработки строки. Эти форматы можно найти в документации Python по ссылке https://docs.python.org/3/library/datetime. html#strftime-strptime-behavior.

Чтобы создать `datetime` для строки, в листинге 15.5 использован код формата `%Y` для четырехзначного года, `%m` для двузначного месяца и `%d` для двузначного дня. Можно использовать `%y`, представляющий двузначный год, чтобы создать новую версию строки.

Листинг 15.5. Объекты datetime в строки и наоборот

```
dt = datetime.strptime('1968-06-20', '%Y-%m-%d')
dt
datetime.datetime(1968, 6, 20, 0, 0)

dt.strftime('%m/%d/%y')
'06/20/68'
```

Вы можете использовать класс `datetime.timedelta` для создания нового `datetime` относительно существующего:

```
from datetime import timedelta
delta = timedelta(days=3)

dt - delta
datetime.datetime(1968, 6, 17, 0, 0)
```

В Python 3.9 представлен новый пакет `zoneinfo` для установки часовых поясов. С его помощью легко устанавливать временные зоны в `datetime`:

```
from zoneinfo import ZoneInfo

dt = datetime(2032, 10, 14, 23, tzinfo=ZoneInfo("America/Jujuy"))
dt.tzname()
'-03'
```

ПРИМЕЧАНИЕ

На момент написания книги Colab все еще работает с Python 3.7, поэтому у вас пока может не быть доступа к zoneinfo.

Библиотека `datetime` включает и класс `datetime.date`:

```
from datetime import date

date.today()
datetime.date(2021, 3, 7)
```

Он похож на `datetime.datetime`, но отслеживает только дату, а не время дня.

Регулярные выражения

Последний пакет, рассматриваемый в этой главе, — библиотека регулярных выражений `re`. Регулярные выражения (regex) предоставляют собой замысловатый язык для поиска в тексте. Можно объявить схему поиска в виде строки и использовать ее для поиска по заданному тексту. На простейшем уровне шаблоном поиска может быть именно тот текст, который вы хотите сопоставить.

В следующем примере объявлен текст с именами капитанов кораблей и их электронными адресами. Далее с помощью функции `re.match()`, которая возвращает объект `match`, производится поиск по этому тексту:

```
captains = '''Ahab: ahab@pequod.com
              Peleg: peleg@pequod.com
              Ishmael: ishmael@pequod.com
              Herman: herman@acushnet.io
              Pollard: pollard@essex.me'''

import re
re.match("Ahab:", captains )
<re.Match object; span=(0, 5), match='Ahab:'>
```

Можете использовать результат этого сопоставления с оператором `if`, чей блок кода будет выполняться, только если текст сопоставим:

```
if re.match("Ahab:", captains ):
    print("We found Ahab")
We found Ahab
```

Функция `re.match()` выполняет сопоставления с начала строки. Если вы позже попытаетесь найти соответствие подстроки в исходной строке, у вас не получится:

```
if re.match("Peleg", captains):
    print("We found Peleg")
else:
    print("No Peleg found!")
No Peleg found!
```

Чтобы найти соответствие любой подстроки в тексте, используйте функцию `re.search()`:

```
re.search("Peleg", captains)
<re.Match object; span=(22, 27), match='Peleg'>
```

Наборы символов

Наборы символов обеспечивают синтаксис для определения более обобщенных соответствий. Синтаксис для набора символов — это некая группа символов, заключенная в квадратные скобки. Для поиска первого вхождения 0 или 1 используйте следующий набор символов:

```
"[01]"
```

Для поиска первого вхождения гласной буквы с последующим знаком препинания можете использовать этот набор символов:

```
"[aeiou][!,?.;]"
```

С помощью дефиса можно установить диапазон символов в наборе. Для любой цифры будет использован синтаксис `[0-9]`, для любой заглавной буквы — `[A-Z]`, а для любой строчной — `[a-z]`. Вы можете сопроводить набор символов знаком `+` для сопоставления одного и более экземпляров или числом в фигурных скобках для сопоставления с точным числом вхождений в строке. В листинге 15.6 показано использование наборов символов.

Листинг 15.6. Наборы символов

```
re.search("[A-Z][a-z]", captains)
<re.Match object; span=(0, 2), match='Ah'>

re.search("[A-Za-z]+", captains)
<re.Match object; span=(0, 4), match='Ahab'>

re.search("[A-Za-z]{7}", captains)
<re.Match object; span=(46, 53), match='Ishmael'>

re.search("[a-z]+\@[a-z]+\.[a-z]+", captains)
<re.Match object; span=(6, 21), match='ahab@pequod.com'>
```

Классы символов

Классы символов — это предообъявленные группы символов для упрощения сопоставления. Весь список классов можно найти в документации `re` (https://docs.python.org/3/library/re.html). К некоторым распространенным классам символов относятся `\d` для цифровых символов, `\s` для символов пробела и `\w` для символов слов. Последние соответствуют любым символам, которые обычно используются в словах, цифровым знакам и символам подчеркивания.

Для поиска первого вхождения цифры, окруженной символами слова, используйте `"\w\d\w"`:

```
re.search("\w\d\w", "His panic over Y2K was overwhelming.")
<re.Match object; span=(15, 18), match='Y2K'>
```

Используйте `+` или фигурные скобки, чтобы указать несколько последовательных вхождений класса символов так же, как вы это делаете с наборами символов:

```
re.search("\w+\@\w+\.\w+", captains)
<re.Match object; span=(6, 21), match='ahab@pequod.com'>
```

Группы

При заключении частей шаблона регулярного выражения в круглые скобки они становятся группой. Доступ к группам можно получить в объекте сопоставления с помощью метода `group()`. Группы пронумерованы, и группа 0 означает совпадение целиком:

```
m = re.search("(\w+)\@(\w+)\.(\w+)", captains)

print(f'Group 0 is {m.group(0)}')
Group 0 is ahab@pequod.com

print(f'Group 1 is {m.group(1)}')
Group 1 is ahab

print(f'Group 2 is {m.group(2)}')
Group 2 is pequod

print(f'Group 3 is {m.group(3)}')
Group 3 is com
```

Именованные группы

Иногда полезно ссылаться на группы по имени, а не с помощью номера. Синтаксис для определения именованной группы следующий:

```
(?P<GROUP_NAME>PATTERN)
```

Далее можно найти группы с помощью их имен:

```
m = re.search("(?P<name>\w+)\@(?P<SLD>\w+)\.(?P<TLD>\w+)", captains)
```

```
print(f'''
Email address: {m.group()}
Name: {m.group("name")}
Secondary level domain: {m.group("SLD")}
Top level Domain: {m.group("TLD")}''')
Email address: ahab@pequod.com
Name: ahab
Secondary level domain: pequod
Top level Domain: com
```

Найти все

До сих пор вы могли находить только первое появление в совпадении. Функция `re.findall()` поможет при сопоставлении всех появлений. Она возвращает каждое совпадение в виде строки:

```
re.findall("\w+\@\w+\.\w+", captains)
['ahab@pequod.com',
 'peleg@pequod.com',
 'ishmael@pequod.com',
 'herman@acushnet.io',
 'pollard@essex.me']
```

Если группы определены, `re.findall()` возвращает каждое совпадение в виде кортежа строк, где каждая строка начинает сопоставление для группы:

```
re.findall("(?P<name>\w+)\@(?P<SLD>\w+)\.(?P<TLD>\w+)", captains)
[('ahab', 'pequod', 'com'),
 ('peleg', 'pequod', 'com'),
 ('ishmael', 'pequod', 'com'),
 ('herman', 'acushnet', 'io'),
 ('pollard', 'essex', 'me')]
```

Найти итератор

Если вы ищете все совпадения в большом тексте, используйте `re.finditer()`. Эта функция возвращает итератор, возвращающий каждое последующее совпадение с каждой итерацией:

```
iterator = re.finditer("\w+\@\w+\.\w+", captains)

print(f"An {type(iterator)} object is returned by finditer" )
An <class 'callable_iterator'> object is returned by finditer
```

```
m = next(iterator)
f"""The first match, {m.group()} is processed
without processing the rest of the text"""
'The first match, ahab@pequod.com is processed
without processing the rest of the text'
```

Замена

Регулярные выражения можно использовать как для сопоставления, так и для замены. Функция `re.sub()` принимает шаблон соответствия, строку замены и исходный текст:

```
re.sub("\d", "#", "Your secret pin is 12345")
      'Your secret pin is #####'
```

Замена с использованием именованных групп

Ссылаться на именованные группы в строке замены можно с помощью следующего синтаксиса:

```
\g<GROUP_NAME>
```

Чтобы реверсировать (развернуть наоборот) адреса электронной почты в тексте с капитанами, используйте замену так:

```
new_text = re.sub("(?P<name>\w+)\@(?P<SLD>\w+)\.(?P<TLD>\w+)",
                  "\g<TLD>.\g<SLD>.\g<name>", captains)

print(new_text)
Ahab: com.pequod.ahab
Peleg: com.pequod.peleg
Ishmael: com.pequod.ishmael
Herman: io.acushnet.herman
Pollard: me.essex.pollard
```

Компиляция регулярных выражений

Компиляция шаблонов регулярных выражений требует определенных затрат. Если вы используете одно и то же регулярное выражение много раз, эффективнее будет скомпилировать его один раз. Это можно сделать с помощью функции `re.compile()`.

Она возвращает скомпилированный объект регулярного выражения на основе шаблона совпадения:

```
regex = re.compile("\w+: (?P<name>\w+)\@(?P<SLD>\w+)\.(?P<TLD>\w+)")
regex
re.compile(r'\w+: (?P<name>\w+)\@(?P<SLD>\w+)\.(?P<TLD>\w+)', re.UNICODE)
```

У этого объекта есть методы, соответствующие многим функциям `re`, например `match()`, `search()`, `findall()`, `finditer()` и `sub ()` (листинг 15.7).

Листинг 15.7. Скомпилированные регулярные выражения

```
regex.match(captains)
<re.Match object; span=(0, 21), match='Ahab: ahab@pequod.com'>

regex.search(captains)
<re.Match object; span=(0, 21), match='Ahab: ahab@pequod.com'>

regex.findall(captains)
[('ahab', 'pequod', 'com'),
 ('peleg', 'pequod', 'com'),
 ('ishmael', 'pequod', 'com'),
 ('herman', 'acushnet', 'io'),
 ('pollard', 'essex', 'me')]

new_text = regex.sub("Ahoy \g<name>!", captains)
print(new_text)
Ahoy ahab!
Ahoy peleg!
Ahoy ishmael!
Ahoy herman!
Ahoy pollard!
```

Резюме

В этой главе вы познакомились с сортировкой данных, файловыми объектами, библиотеками `Datetime` и `re`. Любому разработчику на Python важно иметь хотя бы поверхностные знания по этим темам. Выполнить сортировку помогут либо функция `sorted()`, либо объектные методы `sort()`, такие как тот, что прикреплен к объектам списка. С помощью функции `open()` можно открыть файл и, пока он открыт, производить чтение или запись. Библиотека `Datetime` моделирует время и особенно полезна при работе с данными временных рядов. Наконец, вы можете использовать библиотеку `re` для объявления сложных поисков по тексту.

Вопросы для закрепления

1. Каково конечное значение `sorted_names` в следующем примере?

```
names = ['Rolly', 'Polly', 'Molly']
sorted_names = names.sort()
```

2. Как отсортировать список `nums = [0, 4, 3, 2, 5]` в порядке убывания?
3. Контекстный менеджер может открывать и читать файловые объекты. Какую операцию он может выполнить для проведения «уборки» после своей работы?
4. Как создать объект `datetime` из следующих переменных:

```
year = 2022
month = 10
day = 14
hour = 12
minute = 59
second = 11
microsecond = 100
```

5. Что обозначает `\d` в шаблоне регулярного выражения?

Приложение А.
Ответы к вопросам в конце глав

Здесь вы найдете ответы на вопросы в конце каждой главы.

Глава 1

1. Блокноты Jupiter.
2. Текст и код.
3. Использовать кнопку Mount Drive в части Files левого интерфейса.
4. Python в Google Colab.

Глава 2

1. `int`
2. Они будут выполняться как обычно.
3. `raise LastParamError`
4. `print("Hello")`
5. `2**3`

Глава 3

1. `'a' in my_list`
2. `my_string.count('b')`
3. `my_list.append('a')`
4. Да.
5. `range(3, 14)`

Глава 4

1. `dict(name='Smuah', height=62)`

   или

   `{'name':'Smuah', 'height':62}`

   или

   `dict([['name','Smuah'],['height',62]])`

2. `student['gpa'] = 4.0`
3. `data.get('settings')`
4. У изменяемого объекта есть данные, которые могут быть изменены. Изменить данные после создания неизменяемого объекта нельзя.
5. `set("lost and lost again")`.

Глава 5

1. `Biya[]`
2. `Hiya Henry`
3.
```
for x in range(9):
    if x not in (3, 5, 7):
        print(x)
```

Глава 6

1. `'after-nighttime'`
2. `'before-nighttime'`
3. Ошибка.
4. `@standard_logging`
5. `a`

   `b`

   `1`

Глава 7

1. Массивы NumPy содержат только один тип данных.
   Массивы NumPy осуществляют поэлементные операции.
   У массивов NumPy есть методы матричной математики.
2. `array([[1, 3],`
   `[2, 9]])`
3. `array([[0, 1, 0],`
   `[4, 2, 9]])`
4. `5, 2, 3`
5. `poly1d((6,2,5,1,-10))`

Глава 8

1. `stats.norm(loc=15)`
2. `nrm.rvs(25)`
3. `scipy.special`
4. `std()`

Глава 9

1. ```
   df = pd.DataFrame({'Sample Size(mg)':[ 0.24, 2.34, 0.0234 ],
                      '%P':              [40, 34, 12 ],
                      '%Q':              [60, 66, 88 ]})
   ```
   или
   ```
   df = pd.DataFrame([[ 0.24, 40, 60 ],
                      [ 2.34, 34, 66],
                      [0.0234, 12, 88 ]],
              columns=[ 'Sample Size(mg) ', '%P', '%Q' ])
   ```
2. ```
   df['Total Q'] = df['%Q']/df['Sample Size(mg)']
   ```
   или
   ```
   df['Total Q'] = df.loc[:,'%Q']/df.loc[:,'Sample Size(mg)']
   ```
   или
   ```
   df['Total Q'] = df.iloc[:,2]/df.iloc[:,0]
   ```
3. `df.loc[:, ['%P', '%Q']] / 100`

Глава 10

1. `plt.plot(data['X'], data['Y']`
2. `plt.plot(data['X'], data['Y'])`
3. ```
   fig, (ax1, ax2) = plt.subplots(1, 2)
   ax1.plot(data['X'], data['Y'])
   ax2.plot(data['X'], data['Y1'])
   fig.show()
   ```

   или

   ```
   fig, (ax1, ax2) = plt.subplots(1, 2)
   ax1.plot('X','Y', data=data)
   ax2.plot('X','Y1', data=data)
   fig.show()
   ```

Глава 11

1. Преобразовать данные.
2. Избегать переобучения.
3. Проверить точность модели.

Глава 12

1. ```
   gutenberg.words('austen-emma.txt')
   gutenberg.sents('austen-emma.txt')
   gutenberg.paras('austen-emma.txt')
   ```
2. ```
   alice = gutenberg.words('carroll-alice.txt')
   alice['Alice']
   ```
3. ```
   alice = gutenberg.words('carroll-alice.txt')
   alice_r = []
   for word in alice_w:
   ```

```
    if word not in string.punctuation:
        if word.lower() not in english_stopwords:
            alice_r.append(word)
alice_dist = nltk.FreqDist(alice_r)
alice_dist.tabulate(10)
```

4.
```
alice = Text(gutenberg.words('carroll-alice.txt'))
alice.similar('rabbit')
```

5.
```
nltk.download('names')
names = nltk.corpus.names
all_names = names.words('male.txt')
all_names.extend( names.words('female.txt') )
hamlet_w = gutenberg.words('shakespeare-hamlet.txt')
hamlet_names = []
for word in hamlet_w:
    if word in all_names:
        hamlet_names.append(word)

hamlet_dist = nltk.FreqDist(hamlet_names)
hamlet_dist.most_common(5)
```

Глава 13

1.
```
4
```
4

2.
```
list(map(lambda x: f'{x}'*2, 'omni'))
```
или
```
list(map(lambda x: f'{x}{x}', 'omni'))
```

3.
```
sum([x for x in range(100, 2)])
```

4.
```
(x**2 for x in range(1000))
```

5.
```
def fib():
    f0 = 0
```

```
f1 = 1
while True:
    yield f0
    f0, f1 = f1, f0 + f1
```

Глава 14

1. Текущий экземпляр класса.
2. Когда создается экземпляр объекта.
3. `0`
4. `Hello from A`
   `Goodbye from B`

Глава 15

1. `None`
2. `nums.sort(reverse=True)`
3. Закрытие файлового объекта.
4. `datetime(year, month, day, hour, minute, second, microsecond)`
5. Цифру.

Кеннеди Берман

Основы Python для Data Science

Перевел с английского С.Черников

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*А. Котов*
Литературный редактор	*Т. Сажина*
Художественный редактор	*В. Мостипан*
Корректоры	*Л. Галаганова, М. Лауконен*
Верстка	*Е. Цыцен*

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: 01.2023.
Наименование: книжная продукция.
Срок годности: не ограничен.

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

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

Подписано в печать 10.11.22. Формат 70×100/16. Бумага офсетная.
Усл. п. л. 21,930. Тираж 1000. Заказ 0000.

Дэвид Бизли

PYTHON. ИСЧЕРПЫВАЮЩЕЕ РУКОВОДСТВО

Разнообразие возможностей современного Python становится испытанием для разработчиков всех уровней. Как программисту на старте карьеры понять, с чего начать, чтобы это испытание не стало для него непосильным? Как опытному разработчику Python понять, эффективен или нет его стиль программирования? Как перейти от изучения отдельных возможностей к мышлению на Python на более глубоком уровне? «Python. Исчерпывающее руководство» отвечает на эти, а также на многие другие актуальные вопросы. Эта книга делает акцент на основополагающих возможностях Python (3.6 и выше), а примеры кода демонстрируют «механику» языка и учат структурировать программы, чтобы их было проще читать, тестировать и отлаживать. Дэвид Бизли знакомит нас со своим уникальным взглядом на то, как на самом деле работает этот язык программирования. Перед вами практическое руководство, в котором компактно изложены такие фундаментальные темы программирования, как абстракции данных, управление программной логикой, структура программ, функции, объекты и модули, лежащие в основе проектов Python любого масштаба.

Франсуа Шолле

ГЛУБОКОЕ ОБУЧЕНИЕ НА PYTHON

2-е межд. издание

Глубокое обучение динамично развивается, открывая все новые и новые возможности создания ПО. Это не только автоматический перевод текстов с одного языка на другой, распознавание изображений, но и многое другое. Глубокое обучение превратилось в важный навык, необходимый каждому разработчику. Keras и TensorFlow облегчают жизнь разработчикам и позволяют легко работать даже тем, кто не имеет фундаментальных знаний в области математики или науки о данных. Настала пора познакомиться с глубоким обучением и мощной библиотекой Keras! В этом расширенном и дополненном издании создатель библиотеки Keras — Франсуа Шолле — делится знаниями и с новичками, и с опытными специалистами. Иллюстрации и наглядные примеры помогут вам разобраться с самыми сложными вопросами и концепциями. Вы быстро приобретете навыки, необходимые для разработки приложений глубокого обучения.

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР»

предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург
м. «Выборгская», Б. Сампсониевский пр., д. 29а;
тел. (812) 703-73-73, доб. 6282; e-mail: dudina@piter.com

Москва
м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж;
тел./факс (495) 234-38-15; e-mail: reception@piter.com

БЕЛАРУСЬ

Минск
ул. Харьковская, д. 90, пом. 18
тел./факс: +37 (517)348-60-01, 374-43-25, 272-76-56
e-mail: dudik@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: тел./факс (812) 703-73-73, доб. 6282; e-mail: sales@piter.com

---

Заказ книг для вузов и библиотек:
тел./факс (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

Заказ книг в интернет-магазине: на сайте www.piter.com;
тел. (812) 703-73-74, доб. 6216; e-mail: books@piter.com

Вопросы по продаже электронных книг: тел. (812) 703-73-74, доб. 6217;
e-mail: kuznetsov@piter.com

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ЗАКАЗ И ДОСТАВКА КНИГ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ

- **на нашем сайте:** www.piter.com
- **по электронной почте:** books@piter.com
- **по телефону:** (812) 703-73-74 или 8(800) 500-42-17

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ

Наложенным платежом с оплатой при получении в ближайшем почтовом отделении, пункте выдачи заказов (ПВЗ) или курьеру.

С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.

Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, WebMoney и Qiwi-кошелек.

В любом банке, распечатав квитанцию, которая формируется автоматически после оформления вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ

- курьерская доставка до дома или офиса
- на пункт выдачи заказов выбранной вами транспортной компании
- в отделение «Почты России»

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ

- фамилию, имя, отчество, телефон, e-mail
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру
- название книги, автора, количество заказываемых экземпляров