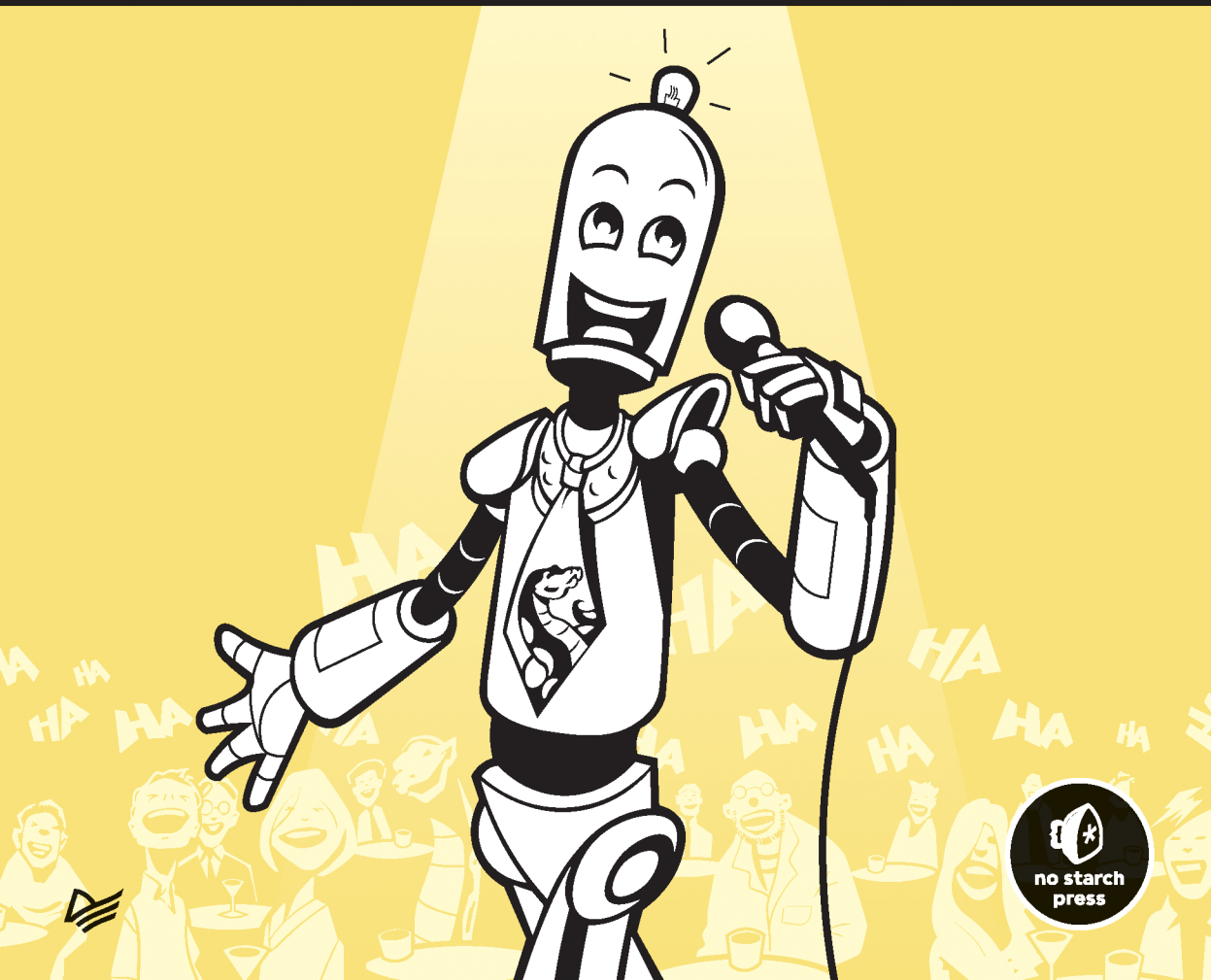
ОДНОСТРОЧНИКИ PYTHON

ЛАКОНИЧНЫЙ И СОДЕРЖАТЕЛЬНЫЙ КОД

КРИСТИАН МАЙЕР



PYTHON ONE-LINERS

Write Concise, Eloquent Python Like a Professional

by Christian Mayer

no starch
press

San Francisco

ОДНОСТРОЧНИКИ PYTHON

ЛАКОНИЧНЫЙ И СОДЕРЖАТЕЛЬНЫЙ КОД

КРИСТИАН МАЙЕР

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-018.1
УДК 004.43
М14

Майер Кристиан

М14 Однострочники Python: лаконичный и содержательный код. — СПб.: Питер, 2022. — 256 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2966-9

Краткость — сестра программиста. Эта книга научит вас читать и писать лаконичные и функциональные однострочники. Вы сможете системно разбирать и понимать код на Python, а также писать выразительно и компактно, как настоящий эксперт.

Здесь вы найдете приемы и хитрости написания кода, регулярные выражения, примеры использования однострочников в различных сферах, а также полезные алгоритмы. Подробные пояснения касаются в том числе и важнейших понятий computer science, что поможет вашему росту в программировании и аналитике.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1718500501 англ.

ISBN 978-5-4461-2966-9

Оглавление

Моей жене Анне

Об авторе

Кристиан Майер (Christian Mayer) — кандидат компьютерных наук (computer science) и создатель популярного сайта https://blog.finxter.com/, посвященного языку Python, автор почтовой рассылки о Python, насчитывающей более 20 000 активных подписчиков. Его обучающие материалы помогают десяткам тысяч студентов совершенствовать навыки написания кода, а также вести бизнес в интернете. Кроме того, Кристиан — автор серии самиздатовских книг Coffee Break Python.

О научном редакторе

Даниэль Зингаро (Daniel Zingaro) — доцент кафедры компьютерных наук и преподаватель Университета Торонто, удостоенный нескольких наград. Основная сфера научных интересов Даниэля — обучение компьютерным наукам. Он автор книги *Algorithmic Thinking* (издательство No Starch Press).

Благодарности

Для этого мира важнее не количество книг, а их качество. Я бесконечно благодарен сотрудникам издательства No Starch Press, которые во всем следуют данной философии. Эта книга — результат сотен часов их усердной работы, бесценных советов и конструктивных замечаний. Моя глубокая благодарность команде No Starch Press, сделавшей написание книг столь приятным занятием.

В частности, я хотел бы поблагодарить Билла Поллока (Bill Pollock), который предложил мне написать эту книгу: он вдохновил меня и рассказал все об издательском деле.

Я очень благодарен моему замечательному редактору Лиз Чедвик (Liz Chadwick), которая готовила рукопись к печати и искусно, настойчиво и изящно придала моим наброскам намного более удобочитаемый вид. Именно благодаря ее помощи книга достигла того уровня ясности изложения, который казался мне изначально недостижимым.

Хотел бы выразить признательность Алекс Фрид (Alex Freed) за неустанное стремление повысить качество текста. Для меня было честью работать с таким талантливым редактором.

Благодарю и моего выпускающего редактора Жанель Людовайз (Janelle Ludowise), которая с энтузиазмом и позитивным настроем доводила эту книгу до совершенства. Спасибо, Жанель. Огромное спасибо также Касси Андреадис (Kassie Andreadis).

Особая благодарность профессору Даниэлю Зингаро (Daniel Zingaro), не пожалевшему изрядной толики своего времени для устранения всех неточностей в этой книге. Кроме того, благодаря множеству его ценных замечаний изложение в ней обрело ясность. Без его усилий она не только была бы полна

ошибок, но и читалась бы намного тяжелее. В то же время все оставшиеся ошибки — исключительно на моей совести.

Мой научный руководитель профессор Розермель также косвенно поучаствовал в написании этой книги, посвятив немало своего времени, умений и усилий обучению меня компьютерным наукам. Я очень благодарен и признателен ему.

Огромное спасибо моей прекрасной жене Анне Альтимире, внимательно выслушивающей, поощряющей и поддерживающей даже самые сумасшедшие мои идеи. Я благодарен также моим детям, Амалии и Гэбриелу, вдохновляющим меня своей любознательностью.

Наконец, наибольшим источником мотивации для меня стали активные участники сообщества Finxter. Прежде всего книга предназначена для амбициозных программистов вроде вас, желающих улучшить навыки написания кода и решить какие-либо практические задачи в реальном мире. После долгих дней работы было так приятно получать письма от участников Finxter с благодарностями, вдохновлявшими меня на написание дальнейших частей книги.

Введение

Данная книга поможет вам стать настоящим экспертом по языку Python. Для этого мы сосредоточимся на написании *однострочников*: лаконичных, удобных программ, занимающих всего одну строку кода на Python. Благодаря им вы сможете писать и читать код на Python быстрее и лучше разберетесь в этом языке.

Кроме того, я насчитал еще пять причин, по которым однострочники Python помогут вам повысить эффективность работы и заслуживают изучения.

Во-первых, совершенствуя свои базовые навыки работы с языком Python, вы сможете побороть множество мешающих вам мелких недостатков, присущих многим программистам. Не имея досконального понимания основ, сложно двигаться дальше. Отдельные строки кода — основные «кирпичики» любой программы. Понимание этих «кирпичиков» помогает не растеряться и по-настоящему овладеть более сложными концепциями.

Во-вторых, вы научитесь использовать на все 100 % чрезвычайно популярные библиотеки Python, например, предназначенные для науки о данных (data science) и машинного обучения. Эта книга состоит из шести глав, посвященных различным аспектам Python, от регулярных выражений до машинного обучения. Подобный подход позволяет рассмотреть широкий диапазон возможных приложений Python, а также научить читателя использовать богатые возможности этих библиотек.

В-третьих, вы научитесь писать код в стиле Python. Начинающие программисты на Python, особенно работавшие ранее с другими языками программирования, зачастую пишут код в несвойственном Python стиле. Мы рассмотрим такие присущие Python понятия, как списковое включение, множественное присваивание и срезы. Вы научитесь писать удобочитаемый код, который сможете использовать совместно с другими программистами, работающими в этой сфере.

В-четвертых, изучение однострочников Python придает мысли ясность и лаконичность. Когда на счету каждый символ, нет места маловразумительному и бессистемному написанию кода.

В-пятых, новые навыки написания однострочников дадут вам возможность разобраться в переусложненных базах кода Python, а также произвести впечатление как на друзей, так и на будущих работодателей при собеседовании. Решать сложные программистские задачи с помощью одной строки кода интересно. И вы не одиноки: онлайн-сообщество энтузиастов Python постоянно соревнуется, кто создаст самые лаконичные и близкие стилю Python решения разнообразных практических (и не совсем практических) задач.

Пример однострочника Python

Главный посыл этой книги состоит в том, что изучение однострочников Python не только необходимо для понимания более сложных баз кода, но и позволит усовершенствовать навыки программирования. Прежде чем разбираться, что происходит в базе кода на многие тысячи строк, следует понять смысл отдельных строк кода.

Рассмотрим один однострочник Python. Не волнуйтесь, если не совсем понимаете, что в нем происходит. Мы расскажем о нем подробнее в главе 6.

```
q = lambda l: q( ❶[x for x in l[1:] if x <= l[0]]) + [l[0]] +
  q([x for x in l if x > l[0]]) if l else []
```

Этот однострочник — изящный лаконичный пример выражения известного алгоритма быстрой сортировки (Quicksort)[1], хотя начинающим и не слишком опытным программистам на Python будет непросто уловить его смысл.

В основе однострочников Python часто лежат другие однострочники, так что сложность рассматриваемых однострочников будет расти от главы к главе. Мы начнем с простых однострочников, которые далее станут основой для других, более сложных. Например, предыдущий однострочник для быстрой сортировки — сложный и длинный, но в его основе лежит более простая идея спискового включения ❶. Вот более простое списковое включение, предназначенное для создания списка квадратов чисел:

```
lst  = [x**2 for x in range(10)]
```

[1] Она же сортировка Хоара. — *Здесь и далее примечания переводчика.*

Можно разбить этот однострочник на еще более простые, чтобы познакомиться с важнейшими базовыми понятиями Python, например присваиванием переменных, математическими операторами, структурами данных, циклами `for`, операторами принадлежности и функцией `range`, — и все это в одной строке!

Учтите, что *базовые* не значит *тривиальные*. Все однострочники, о которых мы будем говорить, полезны, и каждая глава посвящена какой-либо отдельной сфере или дисциплине компьютерных наук, что позволяет продемонстрировать в книге весь спектр возможностей Python.

Примечание относительно удобочитаемости

«Дзен языка Python» состоит из 19 руководящих принципов программирования на языке Python. Можете прочитать их в командной оболочке Python с помощью команды `import this`:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
--обрезано--
```

Согласно «Дзену языка Python», «удобочитаемость важна». Однострочники — минималистичные программы для решения различных задач. Во многих случаях переписывание фрагмента кода Python в виде однострочника повышает его удобочитаемость и делает стиль кода ближе к стандартам Python. Один из примеров — использование *спискового включения* для свертки процедуры создания списка в одну строку кода. Взгляните на следующий пример:

```
# До
squares = []

for i in range(10):
    squares.append(i**2)

print(squares)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Здесь для создания списка из десяти квадратов первых чисел и вывода их в командную оболочку требуется пять строк кода. Однако однострочное решение намного проще, та же самая задача решается намного лаконичнее и удобнее для чтения:

```
# После
print([i**2 for i in range(10)])
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Выводимые результаты — такие же, но этот однострочник использует более свойственную Python идею спискового включения. Подобный код легче читать, и места он занимает меньше.

Однако понять, что делают некоторые однострочники Python, бывает непросто. В ряде случаев при написании решения в виде однострочника Python страдает удобочитаемость. Но подобно тому, как гроссмейстер должен знать все возможные ходы, прежде чем выбрать лучший, программисту необходимо знать все способы выражения в коде своих мыслей, чтобы выбрать лучший. Поиск *самого красивого* решения — не вопрос второстепенной важности, а сама суть экосистемы Python. Как учит нас «Дзен языка Python», «красивый код лучше, чем уродливый».

Для кого эта книга

Вы — начинающий программист на Python или уже достигли среднего уровня? Как и многие, вы наверняка чувствовали, что как программист застряли в своем развитии. Книга поможет вам выбраться из этой ситуации. Вы прочитали множество руководств по программированию в интернете. Вы уже писали собственный код и запускали в промышленную эксплуатацию небольшие проекты. Вы изучили курс основ программирования и прочитали один-два учебника по нему. Возможно, вы даже изучали основы компьютерных наук и программирования в университете.

Возможно, вас сдерживают определенные убеждения, например, что большинство программистов разбирается в исходном коде намного быстрее вас или что вам очень далеко до 10 % лучших программистов. Чтобы достичь продвинутого уровня написания кода и присоединиться к сообществу лучших экспертов, вам необходимо получить еще некоторые навыки.

Мне легко поставить себя на ваше место, поскольку когда я только начинал изучать computer science десять лет назад, я мучительно боролся с чувством,

что ничего не знаю о написании кода. А мои сверстники казались уже очень опытными и компетентными в этой сфере.

Цель книги — помочь вам преодолеть помеху в виде подобных субъективных убеждений и продвинуться на шаг дальше в овладении возможностями Python.

Чему книга вас научит

Ниже представлен общий перечень того, что вы узнаете из книги.

Глава 1 «Краткая памятка по Python» описывает основы Python в целях освежения ваших знаний.

Глава 2 «Трюки Python» содержит десять однострочных приемов, с помощью которых вы освоите основы языка: списковые включения, ввод данных из файлов, использование функций `lambda`, `map()` и `zip()`, квантификатора `all()`, срезов, а также простейшую арифметику списков. Вы также узнаете, как полноценно использовать различные структуры данных и выполнять операции над ними для решения разнообразных повседневных задач.

Глава 3 «Наука о данных» содержит десять однострочников для data science, основанных на библиотеке NumPy. Эта библиотека — центральный элемент обширной функциональности Python для машинного обучения и науки о данных. Вы изучите такие основные понятия NumPy, как массив, форма, оси координат, тип, транслирование, расширенная индексация, срезы, сортировка, поиск, агрегирование и сводные показатели.

Глава 4 «Машинное обучение» охватывает десять однострочников, предназначенных для машинного обучения с помощью библиотеки scikit-learn языка Python. Вы узнаете о регрессионных алгоритмах, предназначенных для предсказания значений. В их числе — линейная регрессия, метод k-ближайших соседей и нейронные сети. Кроме того, рассказывается об алгоритмах классификации, таких как логистическая регрессия, обучение с помощью деревьев принятия решений, метод опорных векторов и случайные леса. Более того, вы научитесь вычислять простейшие сводные показатели многомерных массивов данных и использовать алгоритм k-средних для обучения без учителя.

Эти алгоритмы и методы входят в число важнейших алгоритмов в сфере машинного обучения.

Глава 5 «Регулярные выражения» содержит десять однострочников, которые позволят вам еще профессиональнее использовать регулярные выражения. Вы узнаете о множестве различных простых регулярных выражений, которые можно сочетать для создания более сложных с помощью группировки и поименованных групп, негативного опережающего просмотра, экранированных символов, пробелов, наборов символов (в том числе негативных) и «жадных»/«нежадных» операторов.

Глава 6 «Алгоритмы» содержит десять однострочных алгоритмов, предназначенных для решения широкого спектра задач computer science, включающих анаграммы, палиндромы, надмножества, перестановки, факториалы, простые числа, числа Фибоначчи, обфускацию, поиск и сортировку. Многие из них составляют основу более продвинутых алгоритмов и включают элементы формального алгоритмического образования.

Послесловие завершает данную книгу, подытоживая ваш солидный багаж как совершенно новых, так и улучшенных старых навыков написания кода на языке Python.

Источники дополнительной информации в интернете

В дополнение к изложенному в данной книге материалу вы можете найти онлайн, на сайтах https://pythononeliners.com/ и http://www.nostarch.com/pythononeliners, и другие материалы. В числе этих интерактивных источников:

- *шпаргалки по Python* — можете скачать их в виде подходящих для распечатки PDF-файлов и повесить себе на стену. Они содержат описание основных возможностей языка Python; внимательно их изучив, можно освежить свои навыки Python и закрыть все пробелы в знаниях;
- *видеоуроки по однострочникам* — в качестве части моего курса Python, проводившегося по электронной почте, я записал немало уроков по однострочникам Python из данной книги, они находятся в свободном доступе для просмотра. Эти уроки помогут вам изучать однострочники, выступая в качестве мультимедийных обучающих материалов;

- *задачки по Python* — в этих онлайн-материалах можно найти немало интересных задач по Python, а с помощью бесплатного приложения Finxter.com можно экспериментировать и оттачивать навыки Python, оценивая прогресс обучения по мере чтения книги;
- *файлы кода и блокноты Jupyter* — закатайте рукава и начните работать с кодом, чтобы на высоком уровне овладеть Python. Не жалейте времени на эксперименты с различными значениями параметров и входных данных. Для вашего удобства я включил все однострочники Python в виде исполняемых файлов кода.

От издательства

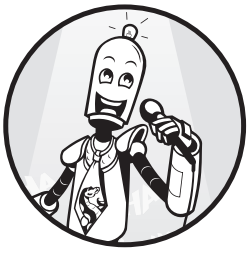
Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

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

1

Краткая памятка по Python



Задача главы — освежить в вашей памяти основные структуры данных, ключевые слова, операции управления потоком команд и прочие азы. Я писал эту книгу для программистов на Python среднего уровня, которые хотели бы достичь новых высот профессионализма. Чтобы стать настоящим специалистом, необходимо тщательно изучить все основы.

Разобравшись с основами, вы сможете охватить взглядом более широкую картину — это умение пригодится вам вне зависимости от того, кем вы хотите стать: техническим руководителем проекта в Google, доктором компьютерных наук или просто первоклассным программистом. Например, доктора компьютерных наук обычно досконально знают все основы своего предмета и могут аргументированно излагать свою позицию, а также находить пробелы, не охваченные исследованиями. Этих людей не ослепляют новейшие технологические достижения. В данной главе приведены важнейшие основы Python, которые станут фундаментом для более сложных вопросов, обсуждаемых далее в книге.

Основные структуры данных

Досконально понимание структур данных — один из важнейших навыков любого программиста. Оно пригодится вам в любом случае: разрабатываете

ли вы проекты машинного обучения, работаете ли с большими базами кода, создаете ли сайты и управляете ими или же пишете алгоритмы.

Числовые типы данных и структуры

Два важнейших числовых типа данных — integer (целое число) и float (число с плавающей точкой). *Integer* — положительное или отрицательное число без плавающей точки (например, 3). *Float* — положительное или отрицательное число, для которого задана определенная точность вычислений с плавающей точкой (например, 3.14159265359). Python предоставляет широкий выбор встроенных числовых операций, а также функциональность для преобразования между этими числовыми типами данных. Внимательно изучите примеры в листинге 1.1, чтобы освоить эти чрезвычайно важные числовые операции.

Листинг 1.1. Числовые типы данных

```
## Арифметические операции
x, y = 3, 2
print(x + y) # = 5
print(x - y) # = 1
print(x * y) # = 6
print(x / y) # = 1.5
print(x // y) # = 1
print(x % y) # = 1
print(-x) # = -3
print(abs(-x)) # = 3
print(int(3.9)) # = 3
print(float(x)) # = 3.0
print(x ** y) # = 9
```

Большинство этих операторов очевидны. Обратите внимание, что оператор `//` служит для целочисленного деления и возвращает округленное вниз целочисленное значение (например, `3 // 2 == 1`).

Булевы значения

Переменная типа *Boolean* может принимать только одно из двух значений — `False` и `True`.

В языке Python типы данных Boolean и integer тесно связаны: «под капотом» типа данных Boolean используются значения типа integer (по умолчанию булево значение `False` представлено целочисленным значением `0`, а булево

значение `True` — целочисленным значением `1`). В листинге 1.2 приведен пример этих двух ключевых слов Boolean.

Листинг 1.2. Булевы значения False и True

```
x = 1 > 2
print(x)
# False

y = 2 > 1
print(y)
# True
```

После вычисления значений указанных выражений переменная `x` будет ссылаться на булево значение `False`, а переменная `y` — на булево значение `True`.

Создать более сложные выражения на языке Python можно с помощью булевых значений, содержащих три следующих важных ключевых слова.

Ключевые слова: and, or, not

Булевы выражения соответствуют основным логическим операторам. Сочетание их всего с тремя следующими ключевыми словами позволяет создавать обширный спектр потенциально весьма сложных выражений:

- `and` — выражение `x and y` равно `True`, если `x` равно `True` *и* `y` равно `True`. Если же хотя бы одно из них равно `False`, то и все выражение в целом будет равно `False`;
- `or` — выражение `x or y` равно `True`, если `x` равно `True` *или* `y` равно `True`. Если же хотя бы одно из них равно `True`, то и все выражение в целом будет равно `True`;
- `not` — выражение `not x` равно `True`, если `x` равно `False`. В противном случае это выражение равно `False`.

Рассмотрим следующий фрагмент кода, представленный в листинге 1.3.

Листинг 1.3. Ключевые слова and, or и not

```
x, y = True, False

print((x or y) == True)
# True

print((x and y) == False)
```

```
# True

print((not y) == True)
# True
```

С помощью этих трех ключевых слов можно выразить любые логические выражения, которые только могут понадобиться.

Приоритет булевых операторов

Для понимания булевой логики очень важен порядок применения булевых операторов. Например, рассмотрим выражение на естественном языке `"идет дождь и холодно или ветрено"`. Его можно интерпретировать двояко:

- `"(идет дождь и холодно) или ветрено"` — в этом случае выражение будет истинным, если ветрено, — даже если дождь не идет;
- `"идет дождь и (холодно или ветрено)"` — в этом же случае выражение будет ложным, если дождь не идет, — неважно, холодно или дует ветер.

Порядок булевых операторов важен. Правильной будет первая интерпретация выражения, поскольку оператор логического И (`and`) обладает приоритетом перед оператором логического ИЛИ (`or`). Рассмотрим фрагмент кода, приведенный в листинге 1.4.

Листинг 1.4. Тип данных Boolean

```
## 1. Булевы операции
x, y = True, False

print(x and not y)
# True

print(not x and y or x)
# True

## 2. Если вычисление значения условия дает False
if None or 0 or 0.0 or '' or [] or {} or set():
    print("Dead code") # Не достигается
```

Этот код демонстрирует два важных нюанса. Во-первых, булевы операторы подчиняются правилам приоритета: наивысший приоритет у оператора `not`, далее следует оператор `and`, а затем оператор `or`. Во-вторых, следующие значения автоматически вычисляются как равные `False`: ключевое слово

None, целочисленное значение 0, значение с плавающей точкой 0.0, пустые строки и пустые объекты контейнерных типов.

Строковые значения

Строки Python представляют собой последовательности символов. Они неизменяемы, а после создания не могут быть модифицированы. Хотя существуют и другие способы создания строковых значений, чаще всего применяются следующие пять:

- *одинарные кавычки:* 'Yes';
- *двойные кавычки:* "Yes";
- *тройные кавычки для многострочных строковых значений:* '''Yes''' или """Yes""";
- *методы для работы со строками:* str(5) == '5' равно True;
- *конкатенация:* 'Py' + 'thon' равно 'Python'.

Нередко в строковых значениях приходится использовать *пробельные символы*. В число чаще всего применяемых пробельных символов входит символ новой строки \n, символ пробела \s и символ табуляции \t.

В листинге 1.5 приведены важнейшие методы для работы со строками.

Листинг 1.5. Строковые типы данных

```
## Важнейшие методы для работы со строками
y = "    This is lazy\t\n   "

print(y.strip())
# Удаляем пробелы: 'This is lazy'

print("DrDre".lower())
# В нижнем регистре: 'drdre'

print("attention".upper())
# В верхнем регистре: 'ATTENTION'

print("smartphone".startswith("smart"))
# Сопоставляет префикс строки с аргументом: True

print("smartphone".endswith("phone"))
# Сопоставляет суффикс строки с аргументом: True

print("another".find("other"))
```

```
# Индекс найденного вхождения: 2

print("cheat".replace("ch", "m"))
# Заменяет все вхождения первого аргумента на второй: meat

print(','.join(["F", "B", "I"]))
# Склеивает все элементы списка, используя строку-разделитель: F,B,I

print(len("Rumpelstiltskin"))
# Длина строки: 15

print("ear" in "earth")
# Содержится: True
```

Этот далеко не полный список методов для работы со строками демонстрирует широкие возможности типа данных string, позволяющие решать распространенные задачи работы со строками с помощью встроенной функциональности Python. Если вы не знаете, как решить какую-то задачу, связанную со строками, то загляните в онлайн-руководство, в котором перечислены все методы для работы со строками: https://docs.python.org/3/library/string.html#module-string.

Boolean, integer, float и string — важнейшие основные типы данных в языке Python. Но нередко приходится не просто создавать элементы данных, а и *структурировать* их. В подобных случаях вам пригодятся контейнерные типы данных. Но прежде чем рассмотреть их подробно, коротко обсудим важный специальный тип данных: None.

Ключевое слово None

Ключевое слово `None` представляет собой константу языка Python и означает *отсутствие значения*. В других языках программирования, например Java, вместо него используется значение `null`. Однако `null` часто вызывает путаницу, особенно у начинающих программистов, считающих, что оно равно целочисленному значению `0`. Поэтому в языке Python используется ключевое слово `None`, как показано в листинге 1.6, чтобы четко обозначить его отличие от числового значения `0`, пустого списка и пустой строки. Любопытно, что значение `None` — единственное значение в типе данных `NoneType`.

В этом коде приведено несколько примеров типа данных `None` (а также показано, чем он не является). Если не описать для функции возвращаемое значение, то по умолчанию будет возвращаться `None`.

Листинг 1.6. Ключевое слово None

```
def f():
   x = 2

# С ключевым словом 'is' я познакомлю вас ниже
print(f() is None)
# True

print("" == None)
# False

print(0 == None)
# False
```

Контейнерные структуры данных

Python включает *контейнерные типы данных*, позволяющие легко и эффективно осуществлять сложные операции.

Списки

Список (list) — это контейнерный тип данных, предназначенный для хранения последовательности элементов. В отличие от строк, списки *изменяемые* (mutable), их можно модифицировать во время выполнения. Что такое тип данных *список*, удобнее всего показать на примерах:

```
l = [1, 2, 2]
print(len(l))
# 3
```

Этот фрагмент кода демонстрирует создание списка и заполнение его тремя целочисленными элементами с помощью квадратных скобок. Как видите, в списке могут быть повторяющиеся элементы. Функция `len()` возвращает количество элементов в списке.

Ключевое слово: is

Ключевое слово `is` просто проверяет, ссылаются ли две переменные на один объект в памяти. Новичков, у которых нет опыта работы с Python, оно может привести в замешательство. В листинге 1.7 мы проверяем, ссылаются ли два целых числа и два списка на одни и те же объекты в памяти.

Листинг 1.7. Ключевое слово is

```
y = x = 3

print(x is y)
# True

print([3] is [3])
# False
```

Если создать два списка (даже состоящих из одних и тех же элементов), то они будут ссылаться на два различных объекта-списка в памяти. Модификация одного из этих объектов никак не повлияет на второй. Списки — *изменяемые*, поскольку их можно модифицировать после создания. Следовательно, если проверить, ссылаются ли два списка на один объект в памяти, результат будет равен `False`. А целочисленные значения — *неизменяемые*, так что нет риска случайно изменить через объект одной переменной значения всех остальных переменных. Объект `3` типа integer нельзя изменить, поскольку такая попытка просто приведет к созданию нового объекта integer, а старый останется неизменным.

Добавление элементов

Язык Python предлагает три основных способа добавления элементов в уже существующий список: *добавление в конец списка*, *вставка* и *конкатенация списков*.

```
# 1. Добавление в конец списка
l = [1, 2, 2]
l.append(4)
print(l)
# [1, 2, 2, 4]

# 2. Вставка
l = [1, 2, 4]
l.insert(2, 3)
print(l)
# [1, 2, 3, 4]

# 3. Конкатенация списков
print([1, 2, 2] + [4])
# [1, 2, 2, 4]
```

В результате всех трех операций получается один и тот же список `[1, 2, 2, 4]`. Но операция `append` выполняется быстрее всего, поскольку не требует ни обхода списка для вставки элементов в нужное место (как в случае `insert`), ни создания нового списка на основе двух подсписков (как в случае

конкатенации списков). Если не вдаваться в подробности, то операцию `insert` имеет смысл использовать только для добавления элемента в конкретное место списка, причем не в конце, а конкатенацию — для конкатенации двух списков произвольной длины. Обратите внимание, что существует и четвертый метод, `extend()`. Он служит для добавления в конец списка нескольких элементов сразу наиболее эффективным способом.

Удаление элементов

Элемент `x` можно легко удалить из списка с помощью метода `remove(x)` списка:

```
l = [1, 2, 2, 4]
l.remove(1)
print(l)
# [2, 2, 4]
```

Этот метод работает с самим объектом списка, а не создает новый, измененный список. В предыдущем примере кода мы создали объект списка `l` и удалили из него элемент. Такой подход не требует избыточных копий данных списка, позволяя экономить память.

Инвертирование списков

Порядок элементов списка можно инвертировать с помощью метода `list.reverse()`:

```
l = [1, 2, 2, 4]
l.reverse()
print(l)
# [4, 2, 2, 1]
```

Инвертирование списка приводит к модификации исходного объекта списка, а не просто создает новый объект списка.

Сортировка списков

Отсортировать элементы списка можно с помощью метода `list.sort()`:

```
l = [2, 1, 4, 2]
l.sort()
print(l)
# [1, 2, 2, 4]
```

Опять же, сортировка списка приводит к модификации исходного объекта списка. Полученный в результате список отсортирован в порядке возрастания. Содержащие строковые объекты списки сортируются в порядке лексикографического возрастания (от `'a'` до `'z'`). В общем случае функция сортировки предполагает, что объекты сравнимы. В общих чертах, если для объектов `a` и `b` произвольного типа данных можно вычислить `a > b`, то Python может отсортировать список `[a, b]`.

Индексация элементов списков

Узнать индекс заданного элемента списка `x` можно с помощью метода `list.index(x)`:

```
print([2, 2, 4].index(2))
# 0

print([2, 2, 4].index(2,1))
# 1
```

Метод `index(x)` ищет первое вхождение элемента `x` в списке и возвращает его индекс. Как и другие основные языки программирования, Python присваивает индекс `0` первому элементу, а индекс $i - 1$ — i-му элементу.

Стеки

Структура данных *стек* (stack) работает по интуитивно понятному принципу «последним поступил, первым обслуживается» (last in, first out, LIFO). Стек аналогичен стопке документов: новые помещаются сверху старых, и по мере работы с этой стопкой первым обрабатывается самый верхний документ. Стек — по-прежнему одна из базовых структур данных в computer science, используемая при управлении операционными системами в алгоритмах, при синтаксическом разборе и поиске с возвратом.

Списки Python могут вполне интуитивным образом играть роль стеков: при использовании операции `append()` для добавления элемента в стек, а `pop()` — для удаления последнего добавленного элемента.

```
stack = [3]
stack.append(42) # [3, 42]
stack.pop() # 42 (stack: [3])
stack.pop() # 3 (stack: [])
```

Благодаря эффективности реализации списков обычно можно обойтись без импорта внешних библиотек стеков.

Множества

Структура данных *множество* (set) — простейший тип коллекций в Python и многих других языках программирования. Многие популярные языки, предназначенные для распределенных вычислений (например, MapReduce и Apache Spark), практически исключительно сосредоточиваются на операциях с множествами как простыми типами данных. Что же такое множество? Множество — неупорядоченная коллекция уникальных элементов. Разобьем это определение на составные части.

Коллекция

Множество — коллекция элементов подобно спискам и кортежам. Коллекция состоит либо из элементов простых типов данных (целочисленных значений, значений с плавающей точкой, строковых значений), либо из более сложных элементов (объектов, кортежей). Однако все типы данных в множестве должны быть *хешируемыми*, то есть обладать соответствующим хеш-значением.

Хеш-значение объекта никогда не меняется и используется для его сравнения с другими объектами. Рассмотрим пример в листинге 1.8, где множество создается на основе трех строковых значений, после проверки их хеш-значений. Далее пробуем создать множество списков, но это нам не удается, поскольку списки нехешируемые.

Листинг 1.8. Множество требует хешируемых элементов

```
hero = "Harry"
guide = "Dumbledore"
enemy = "Lord V."
print(hash(hero))
# 6175908009919104006

print(hash(guide))
# -5197671124693729851

## Можно ли создать множество строковых значений?
characters = {hero, guide, enemy}
print(characters)
# {'Lord V.', 'Dumbledore', 'Harry'}
```

```
## Можно ли создать множество списков?
team_1 = [hero, guide]
team_2 = [enemy]
teams = {team_1, team_2}
# TypeError: unhashable type: 'list'
```

Множество строковых значений можно создать, поскольку строковые значения — *хешируемые*. А создать множество списков нельзя, поскольку списки *нехешируемые*. Дело в том, что хеш-значение зависит от содержимого элемента коллекции, а списки — *изменяемые*; если модифицировать данные в списке, то хеширование тоже должно измениться. А поскольку изменяемые типы данных нехешируемы, использовать их в множествах нельзя.

Неупорядоченность

В отличие от списков, у элементов множества нет четко заданного порядка. Вне зависимости от очередности помещения данных в множество, никогда нельзя быть уверенным, в каком порядке они будут храниться в множестве. Вот пример:

```
characters = {hero, guide, enemy}
print(characters)
# {'Lord V.', 'Dumbledore', 'Harry'}
```

Я вставил в множество сначала героя, но мой интерпретатор вывел первым антагониста (интерпретатор Python — явно на стороне зла). Учтите, что ваш интерпретатор может вывести элементы множества в отличном от моего порядке.

Уникальность

Все элементы множества должны быть уникальными. Строгое определение выглядит следующим образом: для всех пар значений `x, y` из множества при `x!=y` хеш-значения также отличаются: `hash(x)!=hash(y)`. А поскольку все значения в множестве различны, создать армию Гарри Поттеров для войны с лордом В. не получится:

```
clone_army = {hero, hero, hero, hero, hero, enemy}
print(clone_army)
# {'Lord V.', 'Harry'}
```

Неважно, сколько раз вставляется одно значение в одно и то же множество, все равно в нем будет сохранен только один экземпляр этого значения. Дело

в том, что у всех этих героев одно хеш-значение, а множество может содержать не более одного элемента с одинаковым хеш-значением. Существует расширение такой структуры данных , как множество, — *мультимножество*, в котором можно хранить несколько экземпляров одного значения. На практике, впрочем, оно используется редко. А обычные множества, напротив, встречаются практически в коде любого нетривиального проекта — например, для пересечения множества заказчиков и множества посетителей магазина, в результате чего будет возвращено новое множество заказчиков, которые также заходили в магазин.

Ассоциативные массивы

Ассоциативный массив — удобная структура данных для хранения пар (*ключ, значение*):

```
calories = {'apple' : 52, 'banana' : 89, 'choco' : 546}
```

Читать и записывать элементы можно путем указания ключа в квадратных скобках:

```
print(calories['apple'] < calories['choco'])
# True

calories['cappu'] = 74

print(calories['banana'] < calories['cappu'])
# False
```

Для доступа ко всем ключам и значениям ассоциативного массива служат функции `keys()` и `values()` соответственно:

```
print('apple' in calories.keys())
# True

print(52 in calories.values())
# True
```

Для доступа к парам (*ключ, значение*) ассоциативного массива служит метод `items()`:

```
for k, v in calories.items():
    print(k) if v > 500 else None
# 'choco'
```

Таким образом, можно легко проходить в цикле по всем ключам и значениям массива, не обращаясь к ним по отдельности.

Принадлежность

Для проверки того, содержит ли множество, список или ассоциативный массив определенный элемент, служит ключевое слово `in` (листинг 1.9).

Листинг 1.9. Ключевое слово in

```
❶ print(42 in [2, 39, 42])
  # True

❷ print("21" in {"2", "39", "42"})
   # False

   print("list" in {"list" : [1, 2, 3], "set" : {1,2,3}})
   # True
```

С помощью ключевого слова `in` можно выяснить, содержится ли целочисленное значение `42` ❶ в списке целочисленных значений, или проверить принадлежность строкового значения `"21"` множеству строковых значений ❷. Если x встречается в коллекции y, то мы будем говорить, что x — *элемент* коллекции y.

Проверка принадлежности элемента множеству выполняется быстрее, чем проверка принадлежности элемента списку: чтобы проверить наличие элемента x в списке y, необходимо обходить весь список до тех пор, пока не будет найден элемент x или не будут проверены все элементы. Множества же реализованы аналогично ассоциативным массивам: для проверки наличия элемента x в множестве y Python «под капотом» выполняет одну операцию `y[hash(x)]` и проверяет, не равно ли `None` возвращаемое значение.

Списковые включения и включения множеств

Списковое включение — популярная возможность Python, с помощью которой можно быстро создавать и модифицировать списки. Оно описывается простой формулой `[`*`выражение`* `+` *`контекст`*`]`:

- *`выражение`* указывает Python, что делать с каждым из элементов списка;
- *`контекст`* указывает Python, какие элементы списка брать. Контекст состоит из произвольного количества операторов `for` и `if`.

Например, в операторе спискового включения `[x for x in range(3)]` первая часть `x` — это выражение (идентификатор), а вторая часть `for x in range(3)` — контекст. Данный оператор создает список `[0, 1, 2]`. Функция `range()` при использовании с одним аргументом, как в нашем примере, возвращает диапазон последовательных целочисленных значений `0`, `1` и `2`. Ниже представлен еще один пример кода для спискового включения:

```
# (имя, $-доход)
customers = [("John", 240000),
             ("Alice", 120000),
             ("Ann", 1100000),
             ("Zach", 44000)]
# Ценные клиенты, зарабатывающие более $1 млн
whales = [x for x,y in customers if y>1000000]
print(whales)
# ['Ann']
```

Включения для множеств аналогичны списковым включениям, только создается множество, а не список.

Управляющие конструкции

Управляющие конструкции позволяют принимать в коде различные решения. Алгоритмы часто сравнивают с кулинарными рецептами, состоящими из последовательного списка команд: положить в кастрюлю рис, залить холодной водой, посолить, отварить, подать рис на стол. Без *условных операторов* выполнение последовательности команд заняло бы лишь несколько секунд, и рис, конечно, не сварился бы, поскольку вы бы, например, налили воду, посолили, положили рис, а потом сразу же подали его, не дожидаясь, пока вода закипит, а рис сварится.

В различных ситуациях необходимо реагировать по-разному: рис необходимо класть в кастрюлю, только *если* вода уже нагрелась, а подавать его, только *если* он уже мягкий. Практически невозможно писать программы так, чтобы предусмотреть все детерминированные события реального мира. Вместо этого необходимо писать программы, по-разному реагирующие на различные условия.

Ключевые слова if, else и elif

С помощью ключевых слов `if`, `else` и `elif` (листинг 1.10) можно производить условное выполнение различных ветвей кода.

Листинг 1.10. Использование ключевых слов if, else и elif

```
❶ x = int(input("your value: "))
❷ if x > 3:
      print("Big")
❸ elif x == 3:
      print("Medium")
❹ else:
      print("Small")
```

Получаем сначала вводимые пользователем данные, преобразуем их в целое число и сохраняем их в переменной `x` ❶. Затем проверяем, не превышает ли ❷, не равно ли ❸ или меньше `3` ❹ значение переменной. Другими словами, код различным образом реагирует на *непредсказуемые* реалистичные входные данные.

Циклы

Для повтора выполнения фрагментов кода в Python существует два типа циклов: цикл `for` и цикл `while`. С их помощью можно легко написать занимающую всего две строки кода программу, которая будет выполняться бесконечно. Реализовать подобный повтор выполнения иным способом будет непросто (в качестве альтернативы можно воспользоваться *рекурсией*).

В листинге 1.11 показаны в действии оба варианта циклов.

Листинг 1.11. Ключевые слова for и while

```
# Объявление цикла for
for i in [0, 1, 2]:
   print(i)

'''
0
1
2
'''

# Цикл while — аналогичная семантика
j = 0
while j < 3:
   print(j)
   j = j + 1

'''
0
1
2
'''
```

Оба варианта циклов выводят целые числа `0`, `1` и `2` в командную оболочку, но делают это по-разному.

Цикл `for` объявляет переменную цикла `i`, принимающую последовательно все значения из списка `[0, 1, 2]`. Его выполнение продолжается, пока значения не закончатся.

При использовании цикла `while` тело цикла выполняется до тех пор, пока не будет выполнено заданное условие — в данном случае, пока `j < 3`.

Существуют два основных способа выхода из цикла: можно задать условие цикла, которое рано или поздно станет равно `False`, либо воспользоваться ключевым словом `break` в конкретном месте тела цикла. Пример второго варианта приведен в листинге 1.12.

Листинг 1.12. Ключевое слово `break`

```
while True:
   break # цикл не бесконечный

print("hello world")
# hello world
```

Мы создали цикл `while` с условием, тождественно равным `True`. Так что, на первый взгляд, кажется, будто он будет выполняться бесконечно. Бесконечный цикл `while` — распространенная практика при, например, разработке веб-серверов, бесконечно повторяющих процедуру ожидания нового веб-запроса и его обработки.

Однако в некоторых случаях бывает нужно досрочно прервать выполнение цикла. В примере с веб-сервером имеет смысл приостановить выдачу файлов из соображений безопасности, если сервер подвергается атаке. В подобных случаях можно воспользоваться ключевым словом `break` для выхода из цикла и выполнения следующего за ним кода. В приведенном выше листинге 1.12 после досрочного завершения цикла выполняется `print("hello world")`.

Можно также приказать интерпретатору Python пропустить определенные части цикла, не прибегая к досрочному выходу из него. Например, вместо полного останова веб-сервера пропустить вредоносные веб-запросы. Реализовать это можно с помощью оператора `continue`, завершающего выполнение текущей итерации цикла и возвращающего выполнение обратно к условию цикла (листинг 1.13).

Листинг 1.13. Использование ключевого слова continue

```
while True:
  continue
  print("43") # недостижимый код
```

Данный код выполняется бесконечно, но оператор `print` не выполняется никогда. Дело в том, что оператор `continue` завершает текущую итерацию цикла и возвращает поток выполнения в его начало, поэтому `print` никогда не достигается. Подобный никогда не выполняемый код называется *недостижимым* (dead code). Поэтому оператор `continue` (как и оператор `break`) часто применяется при определенных условиях в среде условного выполнения `if-else`.

Функции

Функции позволяют при необходимости повторно использовать фрагменты кода: писать их один раз, а задействовать многократно. При описании функции указывается ключевое слово `def`, название функции и набор аргументов для параметризации выполнения ее тела. Различные наборы аргументов могут резко менять возвращаемые функцией результаты. Допустим, вы описали функцию `square(x)`, возвращающую квадрат входного аргумента `x`. Вызов `square(10)` возвращает `10 × 10 = 100`, а вызов `square(100)` возвращает `100 × 100 = 10000`.

Ключевое слово `return` завершает выполнение функции и передает поток выполнения вызывающей стороне. Можно также указать необязательное значение после ключевого слова `return`, содержащее возвращаемый функцией результат (листинг 1.14).

Листинг 1.14. Ключевое слово return

```
def appreciate(x, percentage):
   return x + x * percentage / 100

print(appreciate(10000, 5))
# 10500.0
```

Мы создали функцию `appreciate()` для вычисления роста вклада при заданной доходности. В приведенном коде мы вычисляем, насколько растет вклад 10 000 долларов за один год при ставке 5 %. В результате получается 10 500 долларов. С помощью ключевого слова `return` указываем, что результат функции должен равняться сумме исходного вклада и номинальной процентной ставки. Тип возвращаемого значения функции `appreciate()` — float.

Лямбда-функции

С помощью ключевого слова `lambda` в языке Python можно задавать *лямбда-функции* — анонимные функции, не описанные в пространстве имен. Если не вдаваться в подробности, то это функции без названия, предназначенные для однократного использования. Синтаксис их выглядит так:

```
lambda <аргументы> : <возвращаемое выражение>
```

У лямбда-функции может быть один или несколько аргументов, разделенных запятыми. После двоеточия (`:`) описывается возвращаемое выражение, в котором может использоваться (или не использоваться) описанный ранее аргумент. Роль возвращаемого выражения может играть любое выражение или даже другая функция.

Лямбда-функции играют важнейшую роль в Python. В коде реальных проектов они встречаются повсеместно: например, для сокращения кода или создания аргументов различных функций Python (например, `map()` или `reduce()`). Рассмотрим код в листинге 1.15.

Листинг 1.15. Использование ключевого слова lambda

```
print((lambda x: x + 3)(3))
# 6
```

Сначала мы создаем лямбда-функцию, принимающую на входе значение `x` и возвращающую результат выражения `x + 3`. Результат представляет собой объект-функцию, которую можно вызывать точно так же, как любую другую. В соответствии с ее семантикой эту функцию можно назвать *функцией-инкрементором*. Результат ее вызова с аргументом `x=3` — суффикс `(3)` в операторе `print` в листинге 1.15 — целочисленное значение `6`. В книге мы будем постоянно использовать лямбда-функции, поэтому убедитесь, что хорошо понимаете, как они работают (хотя у вас еще будет возможность тщательнее разобраться в них).

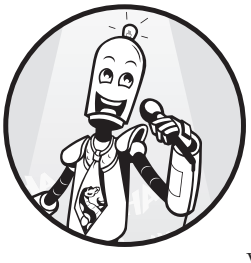
Итоги главы

В этой главе вы освежили свои знания Python благодаря приведенному краткому экспресс-курсу. Вы посмотрели на важнейшие структуры данных Python и их использование в примерах кода. Научились управлять потоком выполнения программы с помощью операторов `if-elif-else`, а также

циклов `for` и `while`. Вы повторили основные типы данных Python — Boolean, integer, float и string — и узнали, какие встраиваемые операции и функции часто применяются для работы с ними. В основе большинства фрагментов кода и нетривиальных алгоритмов на практике лежат обладающие бóльшими возможностями контейнерные типы данных, такие как списки, стеки, множества и ассоциативные массивы. Из приведенных примеров вы научились добавлять, удалять, вставлять и переупорядочивать элементы. Вы также узнали об операторах принадлежности и списковом включении: эффективном и обладающем большими возможностями методе программного создания списков в Python. Наконец, вы узнали о функциях и научились их описывать (включая анонимные лямбда-функции). Теперь вы готовы рассмотреть первые десять простейших однострочников Python.

2

Трюки Python



Трюками мы будем называть способы необычайно быстрого или легкого решения задач. В книге вы встретите массу различных трюков и методик повышения лаконичности кода, которые к тому же позволят ускорить его реализацию. Хотя приемы Python встретятся вам во всех технических главах данной книги, эта посвящена самому очевидному: трюкам, существенно ускоряющим написание кода, которые можно взять на вооружение быстро и без особых усилий.

Эта глава также играет роль фундамента для последующих, более продвинутых глав. Для понимания материала, который будет изложен далее, вам необходимо освоить навыки, заложенные в однострочниках из текущей главы. В частности, мы охватим широкий спектр простейшей функциональности Python, благодаря которой можно писать эффективный код, в том числе списковые включения, доступ к файлам, функции `map()` и `reduce()`, лямбда-функции, срезы, присваивание срезам, функции-генераторы и функцию `zip()`.

Если вы уже опытный программист, то можете пролистать эту главу и решить самостоятельно, какие вопросы хотите изучить подробнее, а в каких уже и так хорошо разбираетесь.

Поиск самых высокооплачиваемых работников с помощью спискового включения

В данном разделе вы познакомитесь с прекрасной, очень эффективной и полезной возможностью Python для создания списков: списковым включением (list comprehension). Оно пригодится нам во множестве однострочников далее в книге.

Общее описание

Представьте, что вы работаете в отделе кадров большой компании и вам нужно найти всех сотрудников, зарабатывающих по крайней мере 100 000 долларов в год. Выходные результаты должны представлять собой список кортежей, каждый из которых состоит из двух значений: имени сотрудника и его годовой зарплаты. Ниже представлен соответствующий код:

```
employees = {'Alice' : 100000,
             'Bob' : 99817,
             'Carol' : 122908,
             'Frank' : 88123,
             'Eve' : 93121}

top_earners = []
for key, val in employees.items():
    if val >= 100000:
        top_earners.append((key,val))

print(top_earners)
# [('Alice', 100000), ('Carol', 122908)]
```

И хотя код работает правильно, существует более простой и намного более лаконичный, а значит, и удобочитаемый способ получить тот же результат. При прочих равных условиях решение, занимающее *меньше строк*, будет понятнее для читающего код.

В Python существует замечательный способ создания новых списков: *списковое включение*. Оно описывается простой формулой:

```
[выражение + контекст]
```

Внешние квадратные скобки указывают, что результат представляет собой новый список. *Контекст* указывает, какие элементы списка необходимо

взять. *Выражение* описывает способ модификации элементов списка перед добавлением результата в список. Пример выглядит так:

```
[x * 2 for x in range(3)]
```

Выделенная жирным шрифтом часть, **`for x in range(3)`**, представляет собой контекст, а остальная часть, `x * 2`, — выражение. Выражение удваивает значения `0`, `1`, `2`, сгенерированные контекстом. Таким образом, результат спискового включения представляет собой следующий список:

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

Как выражение, так и контекст могут быть произвольной степени сложности. Выражение может представлять собой функцию от любой описанной в контексте переменной и выполнять любые вычисления — и даже вызывать внешние функции. Задача выражения — модифицировать каждый из элементов списка перед добавлением его в новый список.

Контекст может состоять из одной или нескольких переменных, описанных с помощью одного или нескольких вложенных циклов `for`. Можно также ограничить контекст, задействовав операторы `if`. В данном случае новое значение добавляется в список только при соблюдении заданного пользователем условия.

Списковое включение лучше всего пояснить на примере. Внимательно изучите следующие примеры, и вы поймете, что оно собой представляет:

```
print([❶x ❷for x in range(5)])
# [0, 1, 2, 3, 4]
```

Выражение ❶: тождественная функция (не меняет контекст переменной `x`).

Контекст ❷: переменная контекста `x` принимает все значения, возвращаемые функцией `range`: `0`, `1`, `2`, `3`, `4`.

```
print([❶(x, y) ❷for x in range(3) for y in range(3)])
# [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

Выражение ❶: создает новый кортеж из переменных контекста `x` и `y`.

Контекст ❷: переменная контекста `x` проходит в цикле по всем значениям, возвращаемым функцией `range (0, 1, 2)`; то же делает и переменная контекста `y`. Эти два цикла `for` — вложенные, вследствие чего переменная

контекста `y` повторяет итерации своего цикла для каждого из значений переменной контекста `x`. Таким образом, получается $3 \times 3 = 9$ сочетаний переменных контекста.

```
print([❶x ** 2 ❷for x in range(10) if x % 2 > 0])
# [1, 9, 25, 49, 81]
```

Выражение ❶: функция возведения в квадрат переменной контекста `x`.

Контекст ❷: переменная контекста `x` проходит в цикле по всем значениям, возвращаемым функцией `range` — `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, — но только нечетным, то есть когда `x % 2 > 0`.

```
print([❶x.lower() ❷for x in ['I', 'AM', 'NOT', 'SHOUTING']])
# ['i', 'am', 'not', 'shouting']
```

Выражение ❶: строковая функция приведения к нижнему регистру переменной контекста `x`.

Контекст ❷: переменная контекста `x` проходит в цикле по всем строковым значениям в списке: `'I'`, `'AM'`, `'NOT'`, `'SHOUTING'`.

Теперь вы сможете понять, что происходит во фрагменте кода, который будет показан ниже.

Код

Рассмотрим уже обсуждавшуюся ранее задачу с зарплатами сотрудников: по ассоциативному массиву со строковыми ключами и целочисленными значениями создать новый список из кортежей (ключ, значение), таких, что соответствующее ключу значение больше или равно 100 000. Соответствующий код приведен в листинге 2.1.

Листинг 2.1. Однострочное решение для спискового включения

```
## Данные
employees = {'Alice' : 100000,
             'Bob' : 99817,
             'Carol' : 122908,
             'Frank' : 88123,
             'Eve' : 93121}

## Однострочник
top_earners = [(k, v) for k, v in employees.items() if v >= 100000]
```

```
## Результат
print(top_earners)
```

Каковы же будут результаты выполнения этого фрагмента кода?

Принцип работы

Рассмотрим этот однострочник подробнее.

```
top_earners = [❶(k, v) ❷for k, v in employees.items() if v >= 100000]
```

Выражение ❶: создает простой кортеж (ключ, значение) для переменных контекста `k` и `v`.

Контекст ❷: метод ассоциативного массива `dict.items()` обеспечивает проход переменной контекста `k` в цикле по всем ключам ассоциативного массива, а переменной контекста `v` — в цикле по соответствующим переменной контекста `k` значениям, но только если значение переменной контекста `v` равно или больше 100 000, в соответствии с условием `if`.

Результат выполнения этого однострочника выглядит следующим образом:

```
print(top_earners)
# [('Alice', 100000), ('Carol', 122908)]
```

В этой простой однострочной программе мы познакомились с важным понятием *спискового включения*. Такие включения используются во многих местах данной книги, поэтому хорошо разберитесь с примерами в текущем разделе, прежде чем читать дальше.

Поиск информативных слов с помощью спискового включения

В этом однострочнике мы еще более углубимся в изучение обладающей большими возможностями функциональности списковых включений.

Общее описание

Поисковые системы ранжируют текстовую информацию по степени соответствия запросу пользователя. Для этого поисковые системы анализируют

содержимое текста, в котором необходимо произвести поиск. Любой текст состоит из слов. В одних содержится немало информации о содержимом текста, а в других — нет. Примеры первых слов — *white*, *whale*, *Captain*, *Ahab*[1] (узнали, откуда это?). Примеры слов второго типа — *is*, *to*, *as*, *the*, *a* и *how*, поскольку они содержатся в большинстве текстов. При реализации поисковых систем часто отфильтровывают слова, не несущие особого значения. Простейший эвристический подход — отфильтровывать все слова из трех или менее букв.

Код

Наша цель — решить следующую задачу: создать на основе многострочного строкового значения список списков, каждый из которых состоит из всех слов одной из строк, причем слова эти длиной три символа и более. В листинге 2.2 приведены данные и решение.

Листинг 2.2. Однострочное решение для поиска информативных слов

```
## Данные
text = '''
Call me Ishmael. Some years ago - never mind how long precisely - having
little or no money in my purse, and nothing particular to interest me
on shore, I thought I would sail about a little and see the watery part
of the world. It is a way I have of driving off the spleen, and regulating
the circulation. - Moby Dick'''

## Однострочник
w = [[x for x in line.split() if len(x)>3] for line in text.split('\n')]

## Результат
print(w)
```

Какими же будут результаты выполнения этого фрагмента кода?

Принцип работы

Данный однострочник создает список списков с помощью двух вложенных выражений для спискового включения:

- во внутреннем выражении для спискового включения `[x for x in line.split() if len(x)>3]` используется строковая функция `split()`

[1] Белый, кит, капитан, Ахав.

для разбиения заданной строки на последовательность слов. Мы проходим по всем словам `x` и добавляем в список те из них, длина которых не менее трех символов;

- во внешнем выражении для спискового включения создается строковое значение `line`, используемое в предыдущем операторе. Опять же, для разбиения текста по символам новой строки `'\n'` применяется функция `split()`.

Конечно, необходимо научиться думать на языке списковых включений, поэтому в первое время они могут показаться сложными. Но когда вы закончите читать данную книгу, списковые включения станут для вас обыденными и вы будете быстро писать код на языке Python в подобном стиле.

Чтение файла

В этом разделе мы прочитаем данные из файла и сохраним результат в виде списка строковых значений (по одному на строку). Мы также удалим из прочитанных строк все ведущие и хвостовые пробельные символы.

Общее описание

В Python чтение файла не представляет трудности, но требует обычно нескольких строк кода (и кое-какого поиска в Google). Ниже представлен один из стандартных способов чтения данных из файла в языке Python:

```
filename = "readFileDefault.py" # этот код

f = open(filename)
lines = []
for line in f:
    lines.append(line.strip())

print(lines)
"""
['filename = "readFileDefault.py" # этот код',
'',
'f = open(filename)',
'lines = []',
'for line in f:',
'lines.append(line.strip())',
'',
'print(lines)']
"""
```

Предполагается, что этот фрагмент кода сохранен в файле `readFileDefault.py` в текущем каталоге. Код открывает данный файл, создает пустой список, `lines`, и заполняет его строковыми значениями с помощью операции `append()` в теле цикла `for`, в котором проходит по всем строкам файла. Мы также воспользовались строковым методом `strip()` для удаления всех ведущих и хвостовых пробельных символов (в противном случае в строках бы оказались и символы новой строки `'\n'`).

Для доступа к файлам на компьютере необходимо знать, как их открывать и закрывать. Получить доступ к файлу данных можно только после его открытия. Если файл был закрыт, значит, все данные уже в него записаны. Python может создавать буфер и ожидать некоторое время, пока не запишет весь буфер в файл (рис. 2.1). Причина этого проста: доступ к файлам осуществляется довольно медленно. Из соображений эффективности Python не записывает биты по отдельности, а ждет, пока буфер наполнится достаточным количеством байтов, после чего сбрасывает весь буфер в файл целиком.

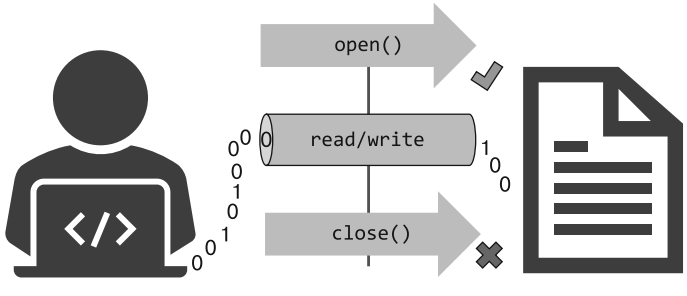


Рис. 2.1. Открытие и закрытие файла на языке Python

Именно поэтому рекомендуется с помощью команды `f.close()` закрывать файл после записи в него данных, чтобы гарантировать, что все данные записаны должным образом, а не остались во временной памяти. Однако существует несколько исключений, когда Python закрывает файл автоматически, в частности, когда счетчик ссылок уменьшается до нуля, как вы увидите в следующем коде.

Код

Наша задача: открыть файл, прочитать все строки, удалить ведущие и хвостовые пробельные символы и сохранить результаты в списке. Соответствующий однострочник приведен в листинге 2.3.

Листинг 2.3. Однострочное решение для построчного чтения файла

```
print([line.strip() for line in open("readFile.py")])
```

Попробуйте догадаться, какие результаты будут выведены при выполнении этого фрагмента кода, прежде чем читать следующий подраздел.

Принцип работы

Для вывода полученного списка в командную оболочку мы применили оператор `print()`. Этот список был создан с помощью спискового включения (см. раздел «Поиск самых высокооплачиваемых работников с помощью спискового включения» на с. 42). В части *выражение* спискового включения используется метод `strip()` строковых объектов.

Контекст спискового включения проходит в цикле по всем строкам файла.

В результате выполнения этого однострочника будет выведен просто он сам (поскольку он предназначен для чтения своего файла исходного кода на Python — `readFile.py`), обернутый в строковый объект и вставленный в список:

```
print([line.strip() for line in open("readFile.py")])
# ['print([line.strip() for line in open("readFile.py")])']
```

Этот подраздел демонстрирует, что повышение лаконичности кода делает его более удобочитаемым, не нанося какого-либо ущерба эффективности.

Лямбда-функции и функция map

В этом разделе я познакомлю вас с двумя важными возможностями Python: лямбда-функциями и функцией `map()` — ценными инструментами в наборе Python-разработчика. Мы воспользуемся ими для поиска конкретного значения в списке строковых значений.

Общее описание

Из главы 1 вы узнали, как описать новую функцию с помощью выражения `def x`, за которым следует тело функции. Однако это не единственный способ описания функции в языке Python. Можно также воспользоваться

лямбда-функциями для описания простой функции *с возвращаемым значением* (которое может быть любым объектом, в том числе кортежем, списком или множеством). Другими словами, лямбда-функция обязательно возвращает вызывающей стороне значение-объект. Учтите, что на практике это ограничивает сферу применения лямбда-функций, ведь в отличие от обычных функций они не предназначены для выполнения кода *без* возвращения объекта в вызывающую среду.

ПРИМЕЧАНИЕ

Мы уже рассматривали лямбда-функции в главе 1, но поскольку это очень важное понятие, используемое повсеместно в данной книге, мы рассмотрим его подробнее в этом разделе.

Благодаря лямбда-функциям можно описать новую функцию с помощью одной строки кода, указав ключевое слово `lambda`. Это удобно, если нужно быстро создать функцию для однократного использования и последующего немедленного сбора мусора. Посмотрим сначала на точный синтаксис лямбда-функций:

```
lambda аргументы : возвращаемое выражение
```

Описание такой функции начинается с ключевого слова `lambda`, за ним следует последовательность ее аргументов, которые необходимо будет указать при ее вызове. Далее указывается двоеточие (`:`) и *возвращаемое выражение*, в котором вычисляется возвращаемое значение на основе аргументов лямбда-функции. Возвращаемое выражение, в котором вычисляются выводимые функцией результаты, может быть любым выражением Python. В качестве примера рассмотрим следующее описание функции:

```
lambda x, y: x + y
```

У данной лямбда-функции два аргумента, `x` и `y`. Возвращаемое значение представляет собой просто их сумму, то есть `x + y`.

Лямбда-функции обычно задействуют, когда функцию нужно вызвать только один раз и ее можно легко описать с помощью одной строки кода. Распространенный пример: использование лямбда-функций с функцией `map`, принимающей в качестве входных аргументов функциональный объект `f` и последовательность `s`. Далее функция `map` применяет функцию `f` к каждому из элементов последовательности `s`. Конечно, *можно* описать для

функционального аргумента f полноценную поименованную функцию. Но зачастую это снижает удобочитаемость, особенно если функция короткая и понадобится только один раз, так что лучше воспользоваться лямбда-функцией.

Прежде чем показать вам однострочник, продемонстрирую еще один трюк Python, который сильно облегчит вам жизнь: проверку того, включает ли строковое значение x подстроку y, с помощью выражения y in x. Этот оператор возвращает True при наличии хотя бы одного вхождения строки символов y в строке x. Например, значение выражения '42' in 'The answer is 42' равно True, а выражения '21' in 'The answer is 42' равно False.

Теперь посмотрим на наш однострочник.

Код

Получая на входе список строковых значений, наш следующий однострочник (листинг 2.4) создает новый список кортежей, каждый из которых состоит из булева значения и исходной строки. Булево значение указывает, встречается ли в исходном строковом значении строка символов 'anonymous'! Мы назвали полученный в результате список mark, поскольку булевы значения *отмечают* (mark) строковые элементы в списке, содержащие строку символов 'anonymous'.

Листинг 2.4. Однострочное решение, помечающее строковые значения, содержащие строку символов 'anonymous'

```
## Данные
txt = ['lambda functions are anonymous functions.',
       'anonymous functions dont have a name.',
       'functions are objects in Python.']

## Однострочник
mark = map(lambda s: (True, s) if 'anonymous' in s else (False, s), txt)

## Результаты
print(list(mark))
```

Какими же будут результаты выполнения этого фрагмента кода?

Принцип работы

Функция map() добавляет к каждому строковому элементу исходного списка txt булево значение, равное True, если этот строковый элемент содержит

слово *anonymous*. Первый аргумент представляет собой анонимную лямбда-функцию, а второй — список строковых значений, которые мы хотим проверить на вхождение желаемой подстроки.

Для поиска подстроки `'anonymous'` используется возвращаемое выражение лямбда-функции `(True, s) if 'anonymous' in s else (False, s)`. Значение `s` представляет собой входной аргумент лямбда-функции, в этом примере — строковое значение. Если в данной строке встречается последовательность символов `'anonymous'`, то выражение возвращает кортеж `(True, s)`. В противном случае возвращается кортеж `(False, s)`.

Результат работы этого однострочника выглядит следующим образом:

```
## Результат
print(list(mark))
# [(True, 'lambda functions are anonymous functions.'),
# (True, 'anonymous functions dont have a name.'),
# (False, 'functions are objects in Python.')]
```

Булевы значения демонстрируют, что только первые два строковых значения в списке содержат подстроку `'anonymous'`.

Лямбда-функции очень пригодятся нам в следующих однострочниках. Кроме того, вы приблизитесь к достижению своей цели: пониманию всех строк кода Python, которые только встретятся вам на практике.

УПРАЖНЕНИЕ 2.1

Попробуйте получить те же результаты с помощью спискового включения вместо функции `map()`. (Решение можно найти в конце главы.)

Извлечение окружения вхождений подстрок с помощью срезов

В этом разделе вы узнаете о важности понятия *среза* (slicing) — процесса «вырезания» подпоследовательности из исходной полной последовательности — для обработки простых текстовых запросов. Мы поищем в каком-нибудь тексте конкретную строку символов, а затем выделим ее из текста вместе с некоторым количеством окружающих ее символов в качестве контекста.

Общее описание

Срезы играют очень важную роль во множестве понятий и навыков, связанных с Python, как простых, так и продвинутых, например, при использовании любых встроенных структур данных Python — списков, кортежей и строк. Срезы также лежат в основе большого количества продвинутых библиотек Python, например NumPy, Pandas, TensorFlow и scikit-learn. Доскональное изучение срезов окажет эффект домино (в положительном смысле) на вашей карьере как Python-разработчика.

Срезы позволяют извлекать из последовательностей подпоследовательности, например части символьных строк. Синтаксис очевиден. Пусть дана переменная `x`, ссылающаяся на строковое значение, список или кортеж. Извлечь из нее подпоследовательность можно с помощью следующей нотации:

```
x[начало:конец:шаг]
```

Полученная в результате подпоследовательность начинается на индексе *`начало`* (включительно) и заканчивается на индексе *`конец`* (не включая его). При желании можно включить необязательный третий аргумент *`шаг`*, определяющий, какие элементы извлекаются, позволяя выбрать только расположенные через *`шаг`* элементы последовательности. Например, применение к переменной `x = 'hello world'` операции среза `x[1:4:1]` дает в результате строку `'ell'`. А операция среза `x[1:4:2]` для той же переменной дает строку `'el'`, поскольку в итоговый срез выбирается только каждый второй элемент. Как вы помните из главы 1, первому элементу последовательности любого типа, например, строки или списка, в языке Python соответствует индекс 0.

Если не включить аргумент *`шаг`*, то Python по умолчанию считает его равным 1. Например, срез `x[1:4]` дает в результате строку `'ell'`.

Если опустить аргументы *`начало`* или *`конец`*, то Python предполагает, что вы хотите начать с начала или закончить в конце последовательности. Например, срез `x[:4]` дает строку `'hell'`, а срез `x[4:]` — строку `'o world'`.

Внимательно посмотрите на следующие примеры, чтобы лучше разобраться в этой функциональности.

```
s = 'Eat more fruits!'

print(s[0:3])
```

```
   # Eat

❶ print(s[3:0])
   # (empty string '')

   print(s[:5])
   # Eat m

   print(s[5:])
   # ore fruits!

❷ print(s[:100])
   # Eat more fruits!

   print(s[4:8:2])
   # mr

❸ print(s[::3])
   # E rfi!

❹ print(s[::-1])
   # !stiurf erom taE

   print(s[6:1:-1])
   # rom t
```

Эти варианты простейшего шаблона срезов Python `x[`*`начало`*`:`*`конец`*`:`*`шаг`*`]` демонстрируют множество интересных свойств данной методики:

- если *`начало`*`>=`*`конец`* с положительным значением *`шаг`*, то срез будет пустым ❶;
- если аргумент *`конец`* больше длины последовательности, то Python вырезает все элементы последовательности вплоть до крайнего справа элемента включительно ❷;
- если *`шаг`* больше нуля, то по умолчанию *`начало`* среза — крайний слева элемент, а *`конец`* — крайний справа (включительно) ❸;
- если *`шаг`* меньше нуля, то срез обходит последовательность в обратном порядке. При незаданных аргументах *`начало`* и *`конец`* срез начинается с крайнего справа элемента (включительно) и заканчивается крайним слева элементом (включительно) ❹. Обратите внимание, что если аргумент *`конец`* задан, то соответствующая позиция не включается в срез.

Далее мы воспользуемся срезами и методом `string.find(`*`значение`*`)` для поиска индекса строкового аргумента *`значение`* в заданной строке.

Код

Наша цель — найти конкретный текст внутри многострочного строкового значения. Затем необходимо вернуть его непосредственное окружение в виде 18 символов, окружающих найденное вхождение. Извлечение окружения вместе с искомым текстом позволяет увидеть текстовый контекст найденной строки — подобно тому, как Google выводит фрагменты текста, окружающие искомое ключевое слово. В листинге 2.5 мы ищем строку символов `'SQL'` в письме Amazon к акционерам — вместе с непосредственным ее окружением в виде 18 окружающих строку `'SQL'` символов.

Листинг 2.5. Однострочное решение для поиска в тексте последовательностей символов и их непосредственного окружения

```
## Данные
letters_amazon = '''
We spent several years building our own database engine,
Amazon Aurora, a fully-managed MySQL and PostgreSQL-compatible
service with the same or better durability and availability as
the commercial engines, but at one-tenth of the cost. We were
not surprised when this worked.
'''

## Однострочник
find = lambda x, q: x[x.find(q)-18:x.find(q)+18] if q in x else -1

## Результат
print(find(letters_amazon, 'SQL'))
```

Попробуйте догадаться, какими будут результаты выполнения этого фрагмента кода.

Принцип работы

Мы описали лямбда-функцию с двумя аргументами: строковым значением `x` и строкой запроса `q`, которую мы будем искать в тексте. Эту лямбда-функцию мы присваиваем переменной с именем `find`. Функция `find(x, q)` ищет строку запроса `q` в строке текста `x`.

Если поисковый запрос `q` не встречается в строковом значении `x`, мы возвращаем результат `-1`. В противном случае вырезаем из строки текста первое вхождение поискового запроса плюс 18 символов слева и 18 символов справа в качестве окружения запроса. Индекс первого вхождения `q` в `x` мы

находим с помощью строковой функции `x.find(q)`, которую вызываем дважды: чтобы определить индекс начала и индекс конца среза, но оба вызова возвращают одно значение, поскольку ни запрос `q`, ни строковое значение `x` не меняются.

И хотя этот код прекрасно работает, избыточный вызов функции требует дополнительных вычислительных ресурсов — недостаток, который можно легко исправить, прибегнув к добавлению вспомогательной переменной для хранения результата первого вызова этой функции и обращению к значению данной переменной в дальнейшем в целях повторного использования этого результата.

Это обсуждение проливает свет на немаловажный компромисс: ограничивая себя одной строкой кода, мы лишаемся возможности описать и повторно использовать вспомогательную переменную для хранения индекса первого вхождения поискового запроса. Вместо этого нам приходится выполнять одну и ту же функцию `find` для вычисления индекса начала среза (и уменьшить результат на 18 позиций индекса) и индекса конца среза (и увеличить результат на 18 позиций индекса). В главе 5 я расскажу вам о более эффективном способе поиска паттернов в строках текста (с помощью регулярных выражений), позволяющем решить эту проблему.

При поиске строки символов `'SQL'` в письме Amazon к акционерам мы находим следующее вхождение поискового запроса в тексте:

```
## Результат
print(find(letters_amazon, 'SQL'))
# a fully-managed MySQL and PostgreSQL
```

В результате получаем искомую строку символов и несколько слов около нее в качестве контекста. Срезы — важнейший элемент базового образования разработчика на языке Python. Давайте рассмотрим еще один пример однострочника.

Сочетание спискового включения и срезов

В этом разделе мы воспользуемся сочетанием спискового включения и срезов для выборки из двумерного набора данных. Наша цель — создать меньшую, но вполне репрезентативную выборку данных из неприемлемо большого набора данных.

Общее описание

Представьте, что работаете финансовым аналитиком в крупном банке и обучаете новую модель машинного обучения, предназначенную для прогнозов курсов акций. Однако набор данных огромен, и обучение модели занимает на вашем компьютере буквально вечность. Например, в машинном обучении часто контролируется точность предсказания модели для различных наборов ее параметров. В нашем приложении, скажем, приходится ждать часами завершения обучения модели (обучение очень сложных моделей на больших объемах данных на самом деле занимает часы). Для ускорения мы сократим набор данных вдвое, исключив из него каждую вторую точку данных курсов акций. Вряд ли эта модификация существенно снизит безошибочность модели.

В данном разделе мы обратимся к двум возможностям Python, о которых вы узнали ранее в этой главе: списковое включение и срезы. Списковое включение позволяет проходить в цикле по всем элементам списка, последовательно модифицируя их. Срезы помогают быстро выбрать из заданного списка каждый второй элемент, а потому прекрасно подходят для простых операций фильтрации. Посмотрим внимательнее на совместное использование этих двух возможностей.

Код

Наша цель — создать на основе имеющихся данных новую обучающую выборку: список списков, каждый из которых состоит из шести чисел с плавающей точкой. Для этого выберем из исходного набора данных каждое второе значение с плавающей точкой. Взгляните на листинг 2.6.

Листинг 2.6. Однострочное решение для выборки данных

```
## Данные (ежедневные котировки акций ($))
price = [[9.9, 9.8, 9.8, 9.4, 9.5, 9.7],
         [9.5, 9.4, 9.4, 9.3, 9.2, 9.1],
         [8.4, 7.9, 7.9, 8.1, 8.0, 8.0],
         [7.1, 5.9, 4.8, 4.8, 4.7, 3.9]]

## Однострочник
sample = [line[::2] for line in price]

## Результат
print(sample)
```

Как обычно, посмотрим, сможете ли вы догадаться, какими будут результаты выполнения этого фрагмента кода.

Принцип работы

В нашем решении используется двухэтапный подход. Во-первых, мы воспользуемся списковым включением для прохода в цикле по всем строкам исходного списка, `price`. Во-вторых, создадим новый список значений с плавающей точкой путем среза каждой из строк; для этого мы добавим `line[`*`начало`*`:`*`конец`*`:`*`шаг`*`]` с параметрами *`начало`* и *`конец`* по умолчанию и *`шаг`* = 2. Этот новый список чисел с плавающей точкой состоит всего из трех значений с плавающей точкой (вместо шести), в результате чего получается следующий массив:

```
## Результат
print(sample)
# [[9.9, 9.8, 9.5], [9.5, 9.4, 9.2], [8.4, 7.9, 8.0], [7.1, 4.8, 4.7]]
```

Это простой однострочник, использующий встроенную функциональность Python. Однако в главе 3 вы увидите еще более короткую версию, в которой для исследования данных применяется библиотека NumPy.

УПРАЖНЕНИЕ 2.2

Вернитесь к этому однострочнику после изучения главы 3 и попробуйте написать его более лаконичный вариант с помощью возможностей библиотеки NumPy. Подсказка: воспользуйтесь расширенными возможностями срезов NumPy.

Исправление испорченных списков с помощью присваивания срезам

В этом разделе вы познакомитесь с замечательной возможностью Python: присваиванием срезам. В процессе *присваивания срезам с левой стороны* используется нотация среза для модификации подпоследовательности исходной последовательности.

Общее описание

Представьте, что работаете в маленьком интернет-стартапе, отслеживающем, какие браузеры встречаются у его пользователей (Google Chrome, Firefox, Safari). Данные хранятся в БД. Для их анализа вы загружаете собранные данные о браузерах в большой список строковых значений, но из-за ошибки в алгоритме отслеживания каждая вторая строка ошибочна и требует замены на правильную.

Пусть ваш веб-сервер всегда перенаправляет первый веб-запрос пользователя на другой URL (распространенная практика в веб-разработке, известная под кодом HTML 301: *перемещено навсегда*). Из этого мы делаем вывод, что первое значение для браузера будет равно второму в большинстве случаев, поскольку при ожидании перенаправления браузер пользователя не меняется. Это означает, что можно легко восстановить исходные данные. Фактически нам нужно дублировать каждое второе строковое значение в списке, то есть превратить список `['Firefox', 'corrupted', 'Chrome', 'corrupted']` в список `['Firefox', 'Firefox', 'Chrome', 'Chrome']`.

Как сделать это быстро, удобочитаемо и эффективно (желательно в одной строке кода)? Сначала в голову приходит идея создать новый список, пройти в цикле по поврежденному списку и добавить каждое из неиспорченных значений для браузеров в новый список дважды. Но это требует двух списков в коде, каждый из которых может включать миллионы записей. Кроме того, данное решение требует нескольких строк кода, а значит, отрицательно скажется на лаконичности и удобочитаемости исходного кода.

К счастью, вы только что узнали о замечательной возможности Python: присваивании срезам. Оно позволяет выбирать и заменять *последовательности элементов*, расположенных между индексами `i` и `j`, с помощью нотации срезов вида `lst[i:j] = [00...0]`. Благодаря тому что срез `lst[i:j]` стоит *с левой стороны* операции присваивания (а не с правой, как мы видели раньше), эта возможность и называется *присваиванием* срезам.

Идея присваивания срезу проста и состоит в замене всех выбранных элементов исходной последовательности слева элементами справа.

Код

Наша цель — заменить каждое второе строковое значение на непосредственно предшествующее ему (листинг 2.7).

Листинг 2.7. Однострочное решение для замены всех испорченных строк

```
## Данные
visitors = ['Firefox', 'corrupted', 'Chrome', 'corrupted',
            'Safari', 'corrupted', 'Safari', 'corrupted',
            'Chrome', 'corrupted', 'Firefox', 'corrupted']

## Однострочник
visitors[1::2] = visitors[::2]

## Результат
print(visitors)
```

Какова же будет исправленная последовательность браузеров в результате выполнения этого кода?

Принцип работы

Наше однострочное решение заменяет «испорченные» строковые значения на строки с браузерами, которые предшествуют им в списке. Для доступа к испорченным элементам в списке `visitors` используется нотация присваивания срезам. Я выделил выбранные элементы в следующем фрагменте кода:

```
visitors = ['Firefox', 'corrupted', 'Chrome', 'corrupted',
            'Safari', 'corrupted', 'Safari', 'corrupted',
            'Chrome', 'corrupted', 'Firefox', 'corrupted']
```

Наш код заменяет эти выделенные элементы срезом справа от операции присваивания. Они выделены в следующем фрагменте кода:

```
visitors = ['Firefox', 'corrupted', 'Chrome', 'corrupted',
            'Safari', 'corrupted', 'Safari', 'corrupted',
            'Chrome', 'corrupted', 'Firefox', 'corrupted']
```

Элементы из первого указанного набора заменяются элементами из второго. Таким образом, итоговый список `visitors` выглядит так (жирным шрифтом выделены замененные элементы):

```
## Результат
print(visitors)
'''
```

```
['Firefox', 'Firefox', 'Chrome', 'Chrome',
'Safari', 'Safari', 'Safari', 'Safari',
'Chrome', 'Chrome', 'Firefox', 'Firefox']
'''
```

Результат представляет собой исходный список, в котором все строковые значения `'corrupted'` заменены предшествующими им строковыми значениями с указанием браузера. Таким образом, мы исправляем испорченный набор данных.

Присваивания срезам — простейший и наиболее эффективный способ решения нашей маленькой задачи. Обратите внимание, что статистика использования браузеров в очищенных данных неискаженная: браузер с долей на рынке 70 % в испорченных данных сохранит долю на рынке 70 % и в очищенных данных. Очищенные данные можно затем применять для дальнейшего анализа — например, чтобы узнать, лучшие ли покупатели пользователи Safari (в конце концов, они обычно тратят больше денег на аппаратное обеспечение).

Итак, вы освоили простую и лаконичную методику модификации списков программным образом, причем без создания дополнительных объектов.

Анализ данных о сердечной деятельности с помощью конкатенации списков

В этом разделе вы узнаете, как с помощью конкатенации списков многократно копировать маленькие списки и объединить их в один большой для генерации циклических данных.

Общее описание

На сей раз мы работаем над маленьким программным проектом для больницы. Наша задача — мониторинг и визуализация статистики состояния здоровья пациентов путем отслеживания их сердечных циклов. Благодаря построению графиков данных сердечных циклов мы помогаем докторам и пациентам отслеживать возможные отклонения от этого цикла. Например, при заданном ряде измерений отдельного сердечного цикла, хранящихся в списке `[62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62]`, необходимо получить визуализацию, подобную изображенной на рис. 2.2.

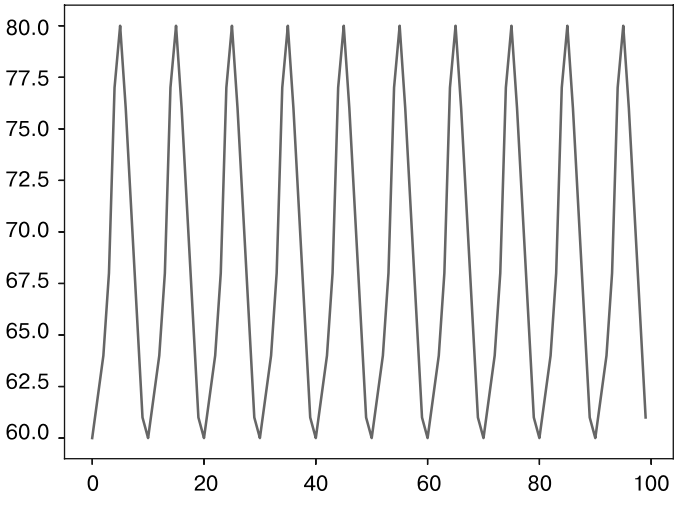


Рис. 2.2. Визуализация ожидаемых сердечных циклов путем копирования избранных значений из списка измеренных данных

Проблема состоит в том, что первое и второе значения данных в нашем списке избыточны: `[62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62]`. Это удобно при построении графика одного сердечного цикла в качестве указания на то, что был визуализирован один полный цикл. Однако необходимо избавиться от этих избыточных данных, чтобы наши ожидаемые сердечные циклы не выглядели так, как на рис. 2.3, при копировании одного и того же сердечного цикла.

Ясно, что нужно *очистить* исходный список, удалив из него избыточные два первых и два последних значения данных, то есть список `[62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62]` превращается в `[60, 62, 64, 68, 77, 80, 76, 71, 66, 61]`.

Можно воспользоваться срезами в сочетании с новой возможностью Python — *конкатенацией списков* (list concatenation), создающей новый список путем *конкатенации* (то есть *соединения*) существующих списков. Например, операция `[1, 2, 3] + [4, 5]` создает новый список `[1, 2, 3, 4, 5]`, не заменяя при этом существующие. При этом можно выполнить операцию * для многократной конкатенации *одного и того же* списка: например, операция `[1, 2, 3] * 3` создает новый список `[1, 2, 3, 1, 2, 3, 1, 2, 3]`.

Кроме того, можно использовать модуль `matplotlib.pyplot` для построения графика сгенерированных данных о сердечной деятельности. Функция

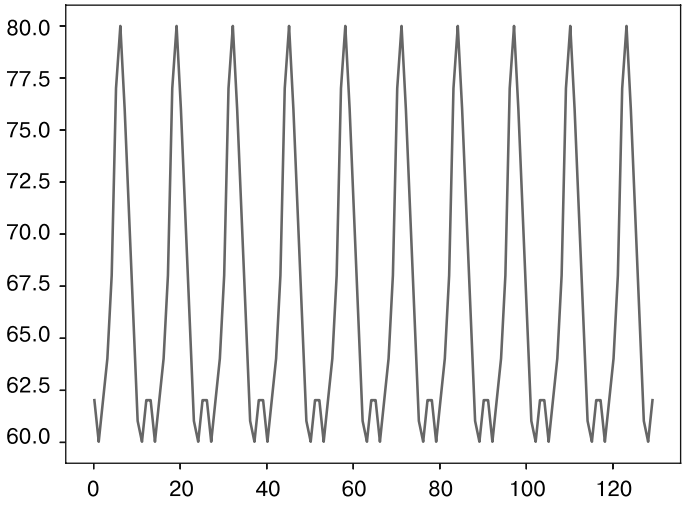


Рис. 2.3. Визуализация ожидаемых сердечных циклов путем копирования всех значений из списка измеренных данных (без фильтрации избыточных данных)

`plot(data)` библиотеки `matplotlib` ожидает на входе итерируемый аргумент `data` (*итерируемый* означает просто объект, по которому можно проходить в цикле, например список) и использует его в качестве значений `y` для последующих типов данных на двумерном графике. Рассмотрим этот пример подробнее.

Код

У нас есть список целых чисел, отражающий измерения сердечного цикла. Сначала мы хотим очистить данные, удалив два первых и два последних значения из этого списка. Далее создаем новый список с ожидаемыми будущими значениями частоты сердечных сокращений, копируя сердечный цикл в экземпляры для будущего времени. Код приведен в листинге 2.8.

Листинг 2.8. Однострочное решение для предсказания частоты сердечных сокращений в различные моменты времени

```
# Зависимости
import matplotlib.pyplot as plt

## Данные
cardiac_cycle = [62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62]
```

```
## Однострочник
expected_cycles = cardiac_cycle[1:-2] * 10

## Результат
plt.plot(expected_cycles)
plt.show()
```

Далее вы узнаете результаты выполнения этого фрагмента кода.

Принцип работы

Приведенный однострочник работает в два этапа. Во-первых, мы очищаем данные с помощью среза с отрицательным аргументом конца, равным `-2`, чтобы срез включал в себя все элементы до правого конца, за исключением двух последних избыточных значений. Во-вторых, мы выполняем конкатенацию полученных значений данных десять раз с помощью оператора повтора *. В результате получаем список из 10 × 10 = 100 целых чисел, состоящих из конкатенаций данных о сердечной деятельности. И на построенном графике результата мы видим желаемую картину, показанную ранее на рис. 2.2.

Поиск компаний, платящих меньше минимальной зарплаты, с помощью выражений-генераторов

В этом разделе вас ожидает комбинация некоторых уже изученных базовых возможностей Python, а также знакомство с удобной функцией `any()`.

Общее описание

Представьте, что работаете в инспекции Министерства труда США и хотите найти компании, которые платят меньше минимальной почасовой ставки, чтобы инициировать расследование. Подобно голодным псам у дверцы грузовика с мясом, ваши офицеры, следящие за соблюдением Закона о справедливых условиях труда (Fair Labor Standards Act, FLSA), ждут список компаний, нарушающих закон о минимальной зарплате. Можете ли вы предоставить им такой список?

Вот инструмент, который вам стоит взять на вооружение: функция `any()` языка Python, принимающая на входе итерируемый объект, например

список, и возвращающая `True`, если вычисление хотя бы одного элемента этого итерируемого объекта дает `True`. Например, выражение `any([True, False, False, False])` равно `True`, а выражение `any([2<1, 3+2>5+5, 3-2<0, 0])` равно `False`.

ПРИМЕЧАНИЕ

Создатель языка Python Гвидо ван Россум (Guido van Rossum), большой поклонник функции any(), даже предложил включить ее в качестве встроенной функции в Python 3. См. подробности в его сообщении в блоге от 2005 года, The Fate of reduce() in Python 3000 («Судьба функции reduce() в Python 3000»): https://www.artima.com/weblogs/viewpost.jsp?thread=98196.

Интересное расширение Python, обобщающее списковые включения, — *выражения-генераторы*. Они работают совершенно аналогично списковым включениям, только без создания в памяти собственно списка. Числа создаются по ходу дела, без сохранения их явным образом в списке. Например, вместо вычисления квадратов первых 20 натуральных чисел с помощью спискового включения, `sum([x*x for x in range(20)])`, можно воспользоваться выражением-генератором: `sum(x*x for x in range(20))`.

Код

Наши данные представляют собой ассоциативный массив ассоциативных массивов, в которых хранятся почасовые ставки работников компаний. Необходимо выделить из него список компаний, платящих по крайней мере одному сотруднику меньше установленной законом минимальной почасовой ставки (`< $9`) (листинг 2.9).

Листинг 2.9. Однострочное решение для поиска компаний, платящих меньше установленной законом минимальной почасовой ставки

```
## Данные
companies = {
    'CoolCompany' : {'Alice' : 33, 'Bob' : 28, 'Frank' : 29},
    'CheapCompany' : {'Ann' : 4, 'Lee' : 9, 'Chrisi' : 7},
    'SosoCompany' : {'Esther' : 38, 'Cole' : 8, 'Paris' : 18}}

## Однострочник
illegal = [x for x in companies if any(y<9 for y in companies[x].values())]
```

```
## Результат
print(illegal)
```

Деятельность каких компаний необходимо расследовать более подробно?

Принцип работы

В этом однострочнике используются два выражения-генератора.

Первое выражение, `y<9 for y in companies[x].values()`, генерирует входные данные для функции `any()`. Для каждого сотрудника оно проверяет, не платят ли ему по почасовой ставке ниже минимальной, `y<9`. Результат представляет собой итерируемый объект значений булева типа. Метод `values()` ассоциативного массива возвращает хранящуюся в нем коллекцию значений. Например, выражение `companies['CoolCompany'].values()` возвращает коллекцию почасовых ставок `dict_values([33, 28, 29])`. Если хотя бы одна из них ниже минимальной, то функция `any()` вернет `True`, а название компании будет сохранено в виде строкового значения в итоговом списке `illegal`, как описано ниже.

Второе выражение-генератор представляет собой списковое включение `[x for x in companies if any(...)]` и создает список названий компаний, для которых предыдущий вызов функции `any()` возвращает `True`. Это и есть компании, платящие своим сотрудникам меньше минимальной почасовой ставки. Обратите внимание, что выражение `x in companies` обходит все ключи ассоциативного массива: названия компаний `'CoolCompany'`, `'CheapCompany'` и `'SosoCompany'`.

Результат, соответственно, будет выглядеть следующим образом:

```
## Результат
print(illegal)
# ['CheapCompany', 'SosoCompany']
```

В двух из трех компаний нужно провести дальнейшее расследование, поскольку они платят слишком мало по крайней мере одному сотруднику. Похоже, вашим офицерам стоит поговорить с Энн, Криси и Коулом!

Форматирование баз данных с помощью функции zip()

В этом разделе вы узнаете, как задать названия столбцов базы данных для списка строк с помощью функции `zip()`.

Общее описание

Функция `zip()` принимает на входе итерируемые объекты `iter_1`, `iter_2`, `...`, `iter_n` и агрегирует их в один итерируемый объект путем выстраивания соответствующих *i*-х значений в один кортеж. В результате получается *итерируемый объект* из кортежей. Например, рассмотрим следующие два списка:

```
[1,2,3]
[4,5,6]
```

Если упаковать их вместе, после простого преобразования типов данных, как вы увидите чуть ниже, получится новый список:

```
[(1,4), (2,5), (3,6)]
```

Распаковка их обратно в исходные кортежи состоит из двух этапов. Во-первых, необходимо убрать внешние квадратные скобки результата, чтобы получить следующие три кортежа:

```
(1,4)
(2,5)
(3,6)
```

Далее, если упаковать их вместе, получится новый список:

```
[(1,2,3), (4,5,6)]
```

Мы опять получили оба исходных списка! Следующий фрагмент кода демонстрирует этот процесс полностью:

```
lst_1 = [1, 2, 3]
lst_2 = [4, 5, 6]

# Упаковка двух списков вместе
zipped = list(zip(lst_1, lst_2))
print(zipped)
# [(1, 4), (2, 5), (3, 6)]

# Обратная распаковка списков
lst_1_new, lst_2_new = zip(❶*zipped)
print(list(lst_1_new))
print(list(lst_2_new))
```

Оператор * служит для распаковки ❶ всех элементов списка. Этот оператор удаляет внешние квадратные скобки списка `zipped`, так что на вход

функции `zip()` попадают три итерируемых объекта (кортежи `(1, 4)`, `(2, 5)`, `(3, 6)`). Если упаковать эти итерируемые объекты вместе, то первые три значения кортежей `1`, `2` и `3` будут упакованы в один новый кортеж, а вторые три значения кортежей `4`, `5` и `6` — в другой новый кортеж. Вместе получатся итерируемые объекты `(1, 2, 3)` и `(4, 5, 6)`, то есть исходные (неупакованные) данные.

Теперь представьте, что работаете в IT-подразделении вашей компании. У вас есть база данных всех сотрудников с названиями столбцов `'name'`, `'salary'` и `'job'`. Однако ваши данные не маркированы, они представляют собой просто набор строк вида `('Bob', 99000, 'mid-level manager')`. Необходимо связать эти названия столбцов с элементами данных и привести их в удобочитаемый вид: `{'name': 'Bob', 'salary': 99000, 'job': 'mid-level manager'}`. Как это сделать?

Код

Наши данные состоят из названий столбцов и информации о сотрудниках в виде списка кортежей (строк). Связываем названия столбцов со строками, получая таким образом список ассоциативных массивов. Каждый из ассоциативных массивов связывает названия столбцов с соответствующими элементами данных (листинг 2.10).

Листинг 2.10. Однострочное решение для приведения списка кортежей в формат базы данных

```
## Данные
column_names = ['name', 'salary', 'job']
db_rows = [('Alice', 180000, 'data scientist'),
           ('Bob', 99000, 'mid-level manager'),
           ('Frank', 87000, 'CEO')]

## Однострочник
db = [dict(zip(column_names, row)) for row in db_rows]

## Результат
print(db)
```

В каком же формате будет выведена база данных `db`?

Принцип работы

Мы создали список с помощью спискового включения (подробности о том, что представляют собой выражение + контекст, см. в разделе «Поиск самых

высокооплачиваемых работников с помощью спискового включения» на с. 42). Контекст состоит из кортежей для всех строк в переменной `db_rows`. Выражение `zip(column_names, row)` упаковывает вместе схему и строки. Например, первым из созданных списковым включением элементов будет `zip(['name', 'salary', 'job'], ('Alice', 180000, 'data scientist'))`, объект, который после преобразования в список приобретает вид `[('name', 'Alice'), ('salary', 180000), ('job', 'data scientist')]`. Форма элементов — `(ключ, значение)`, поэтому можно преобразовать их в ассоциативный массив с помощью функции преобразования `dict()`, чтобы получить желаемый формат базы данных.

ПРИМЕЧАНИЕ

Для функции zip() не важно, что одно из входных значений — список, а другое — кортеж. Ей необходимо только, чтобы входные данные представляли собой итерируемые объекты (а и списки, и кортежи — итерируемые).

Результаты выполнения этого однострочного фрагмента кода таковы:

```
## Результат
print(db)
'''
[{'name': 'Alice', 'salary': 180000, 'job': 'data scientist'},
{'name': 'Bob', 'salary': 99000, 'job': 'mid-level manager'},
{'name': 'Frank', 'salary': 87000, 'job': 'CEO'}]
'''
```

Теперь всем элементам данных соответствуют названия в списке ассоциативных массивов. Вы научились эффективно использовать функцию `zip()`.

Итоги главы

В этой главе вы научились работать со списковыми включениями, вводом данных из файлов, лямбда-функциями, функциями `map()` и `zip()`, квантификатором `all()`, срезами и выполнять простейшие операции со списками. Вы также научились использовать структуры данных для решения разнообразных повседневных задач.

Легкое преобразование структур данных туда и обратно — навык, существенно влияющий на скорость написания кода. Можете не сомневаться — ваши темпы написания кода резко возрастут, как только вы научитесь

эффективнее выполнять различные операции с данными. Небольшие задачи по обработке данных наподобие тех, что встречались в главе, вносят немалый вклад в распространенную «смерть от тысячи порезов» — это огромный вред для общей производительности программиста. Благодаря приведенным в главе приемам, функциям и возможностям Python вы сможете эффективно защититься от этой «тысячи порезов». Образно говоря, эти новоприобретенные инструменты помогут вам гораздо быстрее оправиться от каждого из «порезов».

В следующей главе вы еще более усовершенствуете свои навыки работы с data science благодаря знакомству с новым набором инструментов, предоставляемых библиотекой NumPy для численных вычислений на языке Python.

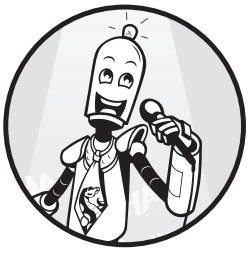
РЕШЕНИЕ УПРАЖНЕНИЯ 2.1

Вот как можно решить вышеупомянутую задачу фильтрации всех строк, содержащих строку символов 'anonymous', с помощью спискового включения вместо функции map(). В данном случае я даже рекомендую использовать именно списковое включение — это более быстрый и аккуратный способ.

```
mark = [(True, s) if 'anonymous' in s else (False, s) for s in txt]
```

3

Наука о данных



Умение анализировать реальные данные — один из наиболее востребованных навыков в XXI столетии. Благодаря мощному аппаратному обеспечению, алгоритмам и вездесущим датчикам, исследователи данных извлекают смысл из больших массивов необработанных данных о погоде, финансовых транзакциях, поведении покупателей и многом другом. Крупнейшие на сегодняшний день компании в мире — Google, Facebook, Apple и Amazon — по существу представляют собой гигантские заводы по обработке данных, и данные лежат в самой сердцевине их бизнес-моделей.

В этой главе вы научитесь обрабатывать и анализировать числовые данные с помощью библиотеки для численных вычислений языка Python *NumPy*. Я приведу десять реальных задач и объясню, как решить их одной строкой кода NumPy. А поскольку NumPy лежит в основе многих высокоуровневых библиотек для исследования данных и машинного обучения (Pandas, scikit-learn и TensorFlow, например), тщательное изучение данной главы повысит вашу рыночную стоимость в условиях нынешней, ориентированной на работу с данными экономики. Так что уделите мне все свое внимание!

Простейшие операции с двумерными массивами

В этом разделе с помощью одной строки кода вам предстоит решить повседневную бухгалтерскую задачу. Я познакомлю вас с функциональностью NumPy, важнейшей библиотеки языка Python для численных вычислений и исследования данных.

Общее описание

Важнейший элемент библиотеки NumPy — *массивы NumPy*, используемые для хранения данных, предназначенных для анализа, визуализации и различных операций. Многие более высокоуровневые библиотеки data science, например Pandas, напрямую или опосредованно основаны на массивах NumPy.

Массивы NumPy аналогичны спискам Python, но имеют некоторые дополнительные преимущества. Во-первых, массивы NumPy занимают меньше места в памяти и чаще всего отличаются бóльшим быстродействием. Во-вторых, массивы NumPy удобнее при обращении более чем к двум осям координат, то есть для *многомерных* данных (доступ к многомерным спискам и их модификация — непростые задачи). А поскольку массивы NumPy могут содержать несколько осей координат, мы будем рассматривать массивы через призму *измерений* (dimensions): включающий две оси координат массив — двумерный. В-третьих, функциональность доступа к массивам NumPy намного шире и включает транслирование, с которым вы познакомитесь ближе в этой главе.

В листинге 3.1 приведены примеры создания одномерного, двумерного и трехмерного массивов NumPy.

Листинг 3.1. Создание одномерного, двумерного и трехмерного массивов в NumPy

```
import numpy as np

# Создание одномерного массива из списка
a = np.array([1, 2, 3])
print(a)
"""
[1 2 3]
"""

# Создание двумерного массива из списка списков
```

```
b = np.array([[1, 2],
              [3, 4]])
print(b)
"""
[[1 2]
 [3 4]]
"""

# Создание трехмерного массива из списка списков списков
c = np.array([[[1, 2], [3, 4]],
              [[5, 6], [7, 8]]])
print(c)
"""
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
"""
```

Сначала мы импортировали библиотеку NumPy в наше пространство имен, указав практически стандартное название для нее: `np`. После импорта библиотеки мы создали массив NumPy, передав обычный список Python в качестве аргумента в функцию `np.array()`. Одномерный массив соответствует простому списку числовых значений (на самом деле массивы NumPy могут содержать и данные других типов, но здесь мы сосредоточим свое внимание на числах). Двумерный массив соответствует вложенному *списку списков* числовых значений, а трехмерный — вложенному *списку списков списков* числовых значений. Размерность массива NumPy определяется из количества открывающих и закрывающих скобок.

Массивы NumPy обладают более широкими возможностями, чем встроенные списки Python. Например, над двумя массивами NumPy можно выполнять простейшие арифметические операции `+`, `-`, `*` и `/`. Эти *поэлементные операции* над массивами (например, сложение их с помощью оператора `+`) заключаются в выполнении соответствующей операции над каждым из элементов массива `a` с соответствующим элементом массива `b`. Другими словами, поэлементная операция агрегирует два элемента, расположенных на одинаковых местах в массивах `a` и `b`. В листинге 3.2 приведены примеры простейших арифметических операций над двумерными массивами.

Листинг 3.2. Простейшие арифметические операции с массивами

```
import numpy as np

a = np.array([[1, 0, 0],
```

```
              [1, 1, 1],
              [2, 0, 0]])

b = np.array([[1, 1, 1],
              [1, 1, 2],
              [1, 1, 2]])

print(a + b)
"""
[[2 1 1]
 [2 2 3]
 [3 1 2]]
"""

print(a - b)
"""
[[ 0 -1 -1]
 [ 0  0 -1]
 [ 1 -1 -2]]
"""

print(a * b)
"""
[[1 0 0]
 [1 1 2]
 [2 0 0]]
"""

print(a / b)
"""
[[1.  0.  0. ]
 [1.  1.  0.5]
 [2.  0.  0. ]]
"""
```

ПРИМЕЧАНИЕ

При использовании операторов NumPy к массивам целых чисел библиотека пытается сгенерировать в качестве результата также массив целых чисел. Только при делении двух массивов целых чисел с помощью оператора деления, a / b, результат будет массивом чисел с плавающей точкой, на что указывают десятичные точки: *1.*, *0.* и *0.5*.

Если присмотреться, вы заметите, что все эти операции группируют два соответствующих массива NumPy поэлементно. При сложении двух массивов результат представляет собой новый массив: каждое новое значение

представляет собой сумму двух соответствующих значений из первого и второго массивов. То же самое относится и к вычитанию, умножению и делению.

NumPy предоставляет и многие другие средства для работы с массивами, включая функцию `np.max()`, вычисляющую *максимальное* из всех значений массива NumPy. Функция `np.min()` вычисляет *минимальное* из всех значений массива NumPy. Функция `np.average()` вычисляет *среднее* значение массива NumPy.

В листинге 3.3 приведены примеры этих трех операций.

Листинг 3.3. Вычисление максимального, минимального и среднего значений массива NumPy

```
import numpy as np

a = np.array([[1, 0, 0],
              [1, 1, 1],
              [2, 0, 0]])

print(np.max(a))
# 2

print(np.min(a))
# 0

print(np.average(a))
# 0.6666666666666666
```

Максимальное из всех значений данного массива NumPy равно `2`, минимальное — `0`, а среднее: `(1 + 0 + 0 + 1 + 1 + 1 + 2 + 0 + 0) / 9 = 2/3`. NumPy включает множество мощных инструментов, но и этих вполне достаточно для решения следующей задачи: как найти максимальный доход после уплаты налогов среди группы людей, если известны годовая зарплата и ставка налогообложения.

Код

Попробуем решить эту задачу на основе данных о зарплате Алисы, Боба и Тима. Похоже, что у Боба последние три года самая высокая зарплата из наших троих друзей. Но приносит ли он домой действительно больше всех денег с учетом индивидуальных ставок налогообложения? Посмотрим на листинг 3.4.

Листинг 3.4. Однострочное решение на основе простейших арифметических операций с массивами

```
## Зависимости
import numpy as np

## Данные: годовые зарплаты в тысячах долларов (за 2017, 2018 и 2019 гг.)
alice = [99, 101, 103]
bob = [110, 108, 105]
tim = [90, 88, 85]

salaries = np.array([alice, bob, tim])
taxation = np.array([[0.2, 0.25, 0.22],
                     [0.4, 0.5, 0.5],
                     [0.1, 0.2, 0.1]])

## Однострочник
max_income = np.max(salaries - salaries * taxation)

## Результат
print(max_income)
```

Попробуйте догадаться: каковы будут результаты выполнения этого кода?

Принцип работы

После импорта библиотеки NumPy мы помещаем данные в двумерный массив NumPy, содержащий три строки (по одной строке для каждого человека: Алисы, Боба и Тима) и три столбца (по одному столбцу для каждого года: 2017, 2018 и 2019). У нас два двумерных массива: `salaries` содержит годовой доход, а `taxation` — ставки налогообложения по всем людям и годам.

Для вычисления чистого дохода необходимо вычесть налоги (в долларах) из валового дохода, хранящегося в массиве `salaries`. Для этого мы воспользуемся перегруженными операторами NumPy `-` и `*`, выполняющими поэлементные вычисления с массивами NumPy.

Поэлементное произведение двух многомерных массивов называется *произведением Адамара*.

Листинг 3.5 демонстрирует, как выглядит массив NumPy после вычитания налогов из валового дохода.

Как видим из второй строки, высокий доход Боба существенно снизился после уплаты 40 и 50 % налогов.

Листинг 3.5. Простейшие арифметические операции над массивами

```
print(salaries - salaries * taxation)
"""
[[79.2  75.75 80.34]
 [66.   54.   52.5 ]
 [81.   70.4  76.5 ]]
"""
```

Предыдущий фрагмент кода выводит максимальное значение этого итогового массива. Функция `np.max()` просто находит в массиве максимальное значение, которое мы затем сохраняем в переменной `max_income`. Максимальным значением оказывается доход Тима, равный 90 000 долларов в 2017 году, облагаемый налогом всего 10 % — результат выполнения однострочника равен `81.` (опять же, точка в конце указывает на тип данных float).

С помощью простейших поэлементных арифметических операций над массивами NumPy мы проанализировали ставки налогов группы людей. Воспользуемся тем же набором данных для демонстрации более сложных понятий NumPy: срезов и транслирования.

Работа с массивами NumPy: срезы, транслирование и типы массивов

Данный однострочник демонстрирует мощь трех интересных возможностей NumPy: срезов, транслирования и типов массивов. Наши данные представляют собой массив, содержащий различные профессии и соответствующие зарплаты. Мы воспользуемся всеми этими тремя понятиями, чтобы каждые два года повышать зарплаты одних только исследователей данных на 10 %.

Общее описание

Основная загвоздка в нашей задаче — как поменять конкретные значения в многострочном массиве NumPy. Нам нужно поменять каждое второе значение для одной конкретной строки. Посмотрим на основные понятия, которые необходимы для решения этой задачи.

Срезы и доступ по индексу

Доступ по индексу и срезы в NumPy аналогичны доступу по индексу и срезам в Python (см. главу 2): к элементам одномерного массива можно обращаться

путем указания в квадратных скобках `[]` индекса или диапазона индексов. Например, операция доступа по индексу `x[3]` возвращает четвертый элемент массива NumPy `x` (поскольку индекс первого элемента — 0).

Можно также использовать доступ по индексу для многомерных массивов путем указания индекса для каждого измерения по отдельности либо разделенных запятыми индексов для доступа к различным измерениям. Например, операция доступа по индексу `y[0,1,2]` служит для обращения к первому элементу первой оси координат, второму элементу второй оси координат и третьему элементу третьей оси координат. Учтите, что такой синтаксис не подходит для многомерных списков Python.

Перейдем теперь к *срезам* в NumPy. Изучите примеры в листинге 3.6, чтобы разобраться с одномерными срезами в NumPy, и не стесняйтесь снова обратиться к главе 2, если у вас возникнут какие-либо проблемы с их пониманием.

Листинг 3.6. Примеры одномерных срезов

```
import numpy as np

a = np.array([55, 56, 57, 58, 59, 60, 61])
print(a)
# [55 56 57 58 59 60 61]

print(a[:])
# [55 56 57 58 59 60 61]

print(a[2:])
# [57 58 59 60 61]

print(a[1:4])
# [56 57 58]

print(a[2:-2])
# [57 58 59]

print(a[::2])
# [55 57 59 61]

print(a[1::2])
# [56 58 60]

print(a[::-1])
# [61 60 59 58 57 56 55]

print(a[:1:-2])
# [61 59 57]
```

```
print(a[-1:1:-2])
# [61 59 57]
```

Следующий этап — полностью разобраться с многомерными срезами. Доступ по индексу выполняется путем применения одномерных срезов по отдельности для каждой оси координат (разделенных запятыми), чтобы выбрать диапазоны элементов по этой оси. Не пожалейте времени, чтобы тщательно разобраться с примерами в листинге 3.7.

Листинг 3.7. Примеры многомерных срезов

```
import numpy as np

a = np.array([[0, 1, 2, 3],
              [4, 5, 6, 7],
              [8, 9, 10, 11],
              [12, 13, 14, 15]])

print(a[:, 2])
# Третий столбец: [ 2  6 10 14]

print(a[1, :])
# Вторая строка: [4 5 6 7]

print(a[1, ::2])
# Вторая строка, каждый второй элемент: [4 6]

print(a[:, :-1])
# Все столбцы, за исключением последнего:
# [[ 0  1  2]
# [ 4  5  6]
# [ 8  9 10]
# [12 13 14]]

print(a[:-2])
# Аналогично a[:-2, :]
# [[ 0  1  2  3]
# [ 4  5  6  7]]
```

Изучайте листинг 3.7 до тех пор, пока не будете уверены, что хорошо понимаете принципы многомерных срезов. Двумерный срез можно производить с помощью синтаксиса `a[`*`срез1, срез2`*`]`. Для дополнительных измерений необходимо добавить через запятую дополнительные операции срезов (с помощью операторов срезов *`начало`*`:`*`конец`* или *`начало`*`:`*`конец`*`:`*`шаг`*). Каждый срез служит для выбора отдельной подпоследовательности элементов соответствующего измерения. Понимания этой основной идеи вполне достаточно для перехода от одномерных к многомерным срезам.

Транслирование

Транслирование (broadcasting) означает автоматический процесс приведения двух массивов NumPy к одной форме для применения определенных поэлементных операций (см. подраздел «Срезы и доступ по индексу» на с. 77). Транслирование тесно связано с *атрибутом формы* массивов NumPy, который, в свою очередь, тесно связан с понятием осей координат. Так что займемся изучением осей координат, форм и транслирования.

Каждый массив охватывает несколько *осей координат*, по одной для каждого измерения (листинг 3.8).

Листинг 3.8. Оси координат и размерность трех массивов NumPy

```
import numpy as np

a = np.array([1, 2, 3, 4])
print(a.ndim)
# 1

b = np.array([[2, 1, 2], [3, 2, 3], [4, 3, 4]])
print(b.ndim)
# 2

c = np.array([[[1, 2, 3], [2, 3, 4], [3, 4, 5]],
              [[1, 2, 4], [2, 3, 5], [3, 4, 6]]])
print(c.ndim)
# 3
```

Здесь мы видим три массива: `a`, `b` и `c`. В атрибуте `ndim` массива хранится количество его осей координат. Мы просто вывели его в командную оболочку для каждого из массивов. Массив `a` — одномерный, массив `b` — двумерный, а массив `c` — трехмерный. У каждого из этих массивов есть атрибут, указывающий его форму: кортеж, содержащий количество элементов по каждой оси. У двумерного массива — два значения в этом кортеже: количество строк и количество столбцов. Для массивов большей размерности i-е значение кортежа задает количество элементов по i-й оси. Следовательно, количество элементов этого кортежа соответствует размерности массива NumPy.

ПРИМЕЧАНИЕ

При повышении размерности массива (например, при переходе от двумерного к трехмерному массиву) новая ось становится осью 0, а i-я ось массива более низкой размерности становится $(i + 1)$-й осью массива более высокой размерности.

В листинге 3.9 мы выводим атрибуты shape тех же самых массивов из листинга 3.8.

Листинг 3.9. Формы одномерного, двумерного и трехмерного массивов NumPy

```
import numpy as np

a = np.array([1, 2, 3, 4])
print(a)
"""
[1 2 3 4]
"""
print(a.shape)
# (4,)

b = np.array([[2, 1, 2], [3, 2, 3], [4, 3, 4]])
print(b)
"""
[[2 1 2]
 [3 2 3]
 [4 3 4]]
"""
print(b.shape)
# (3, 3)

c = np.array([[[1, 2, 3], [2, 3, 4], [3, 4, 5]],
              [[1, 2, 4], [2, 3, 5], [3, 4, 6]]])
print(c)
"""
[[[1 2 3]
  [2 3 4]
  [3 4 5]]

 [[1 2 4]
  [2 3 5]
  [3 4 6]]]
"""
print(c.shape)
# (2, 3, 3)
```

Как видите, атрибуты shape содержат намного больше информации, чем атрибуты ndim. Каждый атрибут shape представляет собой кортеж с числом элементов по каждой из осей координат:

- массив a — одномерный, так что его кортеж shape содержит один элемент, отражающий количество столбцов (четыре элемента);
- массив b — двумерный, поэтому его кортеж shape содержит два элемента, перечисляющих количество столбцов и строк;

- массив `c` — трехмерный, так что его кортеж `shape` содержит три элемента, по одному для каждой оси. Ось 0 содержит два элемента (каждый элемент — двумерный массив), ось 1 — три элемента (каждый из которых — одномерный массив), а ось 2 — три элемента (каждый из которых — целочисленное значение).

Теперь, когда вы разобрались с атрибутом `shape`, вам будет легче понять общую идею транслирования: приведение двух массивов к одной форме путем переупорядочивания их элементов. Посмотрим, как происходит транслирование. Оно автоматически исправляет поэлементные операции над массивами NumPy различной формы. Например, оператор умножения *, примененный к массивам NumPy, обычно выполняет поэлементное умножение. Но что произойдет, если данные слева и справа от него не совпадают по форме (скажем, операнд слева представляет собой массив NumPy, а справа — значение с плавающей точкой)? В этом случае NumPy не выдаст ошибку, а автоматически создаст новый массив из данных, расположенных справа. Размер и размерность этого нового массива — те же, что и у массива слева, он содержит те же значения с плавающей точкой.

Таким образом, транслирование представляет собой преобразование массива низкой размерности в массив более высокой размерности для осуществления поэлементных операций.

Однородные значения

Массивы NumPy — *однородны* (homogenous), то есть типы данных у всех значений массива одинаковы. Ниже представлен неполный список возможных типов данных массивов:

- `bool` — булев тип данных в Python (1 байт);
- `int` — целочисленный тип данных в Python (размер по умолчанию — 4 или 8 байт);
- `float` — тип данных с плавающей точкой в Python (размер по умолчанию — 8 байт);
- `complex` — тип данных для комплексных чисел в Python (размер по умолчанию — 16 байт);
- `np.int8` — целочисленный тип данных (1 байт);
- `np.int16` — целочисленный тип данных (2 байта);

- `np.int32` — целочисленный тип данных (4 байта);
- `np.int64` — целочисленный тип данных (8 байт);
- `np.float16` — тип данных с плавающей точкой (2 байта);
- `np.float32` — тип данных с плавающей точкой (4 байта);
- `np.float64` — тип данных с плавающей точкой (8 байт).

В листинге 3.10 показано создание массивов NumPy с различными типами.

Листинг 3.10. Массивы NumPy с различными типами

```
import numpy as np

a = np.array([1, 2, 3, 4], dtype=np.int16)
print(a) # [1 2 3 4]
print(a.dtype) # int16

b = np.array([1, 2, 3, 4], dtype=np.float64)
print(b) # [1. 2. 3. 4.]
print(b.dtype) # float64
```

В этом коде два массива, `a` и `b`. Тип данных первого из них — np.int16. Числа в этом массиве — типа integer (после числа нет десятичной точки). Если точнее, то результат вывода свойства `dtype` массива `a` — int16.

Тип данных второго массива — np.float64. Так что даже если создать массив, в основе которого лежит список целых чисел, NumPy все равно преобразует его тип в np.float64.

Из вышеизложенного можно сделать два важных вывода: NumPy позволяет управлять типом данных, причем типы данных массивов NumPy — однородны.

Код

Итак, имеются данные по многим специальностям, и нужно повышать зарплаты одних только исследователей данных на 10 % раз в два года. Соответствующий код приведен в листинге 3.11.

Задумайтесь на минуту о результатах выполнения этого фрагмента кода. Как вы думаете, что изменится? Какой тип данных будет у полученного в результате массива? Что выведет данный код?

Листинг 3.11. Однострочное решение, использующее срезы и присваивания срезам

```
## Зависимости
import numpy as np

## Данные: годовые зарплаты в тысячах долларов (за 2025, 2026 и 2027 гг.)
dataScientist =     [130, 132, 137]
productManager =    [127, 140, 145]
designer =          [118, 118, 127]
softwareEngineer =  [129, 131, 137]

employees = np.array([dataScientist,
                      productManager,
                      designer,
                      softwareEngineer])

## Однострочник
employees[0,::2] = employees[0,::2] * 1.1

## Результат
print(employees)
```

Принцип работы

В этом фрагменте кода вы попадаете в 2024 год. Во-первых, вы создаете массив NumPy, каждая строка которого содержит ожидаемую годовую зарплату одного специалиста (исследователя данных, руководителя по программному продукту, дизайнера или разработчика программного обеспечения). А каждый столбец отражает соответствующие годовые зарплаты за 2025, 2026 и 2027 годы. Полученный в результате массив NumPy включает четыре строки и три столбца.

Вы нашли средства, чтобы поощрить самых важных специалистов в компании. Вы верите в будущее data science, поэтому решили вознаградить незаметных героев вашей компании: исследователей данных. Вам нужно обновить массив NumPy так, чтобы только зарплаты исследователей данных возрастали на 10 % через год (без капитализации процентов) начиная с 2025 года.

Вы разработали следующий замечательный однострочник:

```
employees[0,::2] = employees[0,::2] * 1.1
```

Он выглядит просто и аккуратно, а результаты его работы — следующие:

```
[[143 132 150]
 [127 140 145]
```

```
[118 118 127]
[129 131 137]]
```

Несмотря на свою простоту, однострочник использует три интересных продвинутых понятия.

Срезы

Во-первых, мы воспользуемся *срезами* и *присваиванием срезам*. В этом примере с помощью среза мы извлечем каждое второе значение из первой строки массива NumPy `employees`. Далее выполним некоторые модификации и обновим каждое второе значение первой строки с помощью присваивания срезу. Синтаксис присваивания срезу не отличается от самого среза, за исключением одного важного нюанса: в этом случае срез указывается с левой стороны оператора присваивания. Соответствующие элементы будут заменены элементами, указанными справа от оператора присваивания. В представленном фрагменте кода мы заменяем содержимое первой строки массива NumPy обновленными данными о зарплатах.

Транслирование

Во-вторых, мы воспользуемся транслированием для автоматического исправления поэлементных операций над массивами NumPy различной формы. В нашем однострочнике левый операнд — массив NumPy, а правый — значение с плавающей точкой. Опять же, NumPy автоматически создает новый массив того же размера и размерности, что и массив слева от оператора присваивания, и заполняет его, по сути, копиями этого значения с плавающей точкой. Фактически NumPy производит вычисление наподобие следующего:

```
np.array([130 137]) * np.array([1.1, 1.1])
```

Типы массивов

В-третьих, наверное, вы поняли, что тип данных результата — не float, а integer, даже если вы выполняете операции над числами с плавающей точкой. При создании массива NumPy понимает, что тот содержит только целочисленные значения, поэтому полагает, что массив должен быть типа integer. Никакие операции над массивом типа integer не меняют типа данных, вследствие чего NumPy будет округлять значения до целочисленных. Опять же, узнать тип массива можно с помощью свойства `dtype`:

```
print(employees.dtype)
# int32
employees[0,::2] = employees[0,::2] * 1.1
print(employees.dtype)
# int32
```

Итак, вы узнали о срезах, присваивании срезам, транслировании и типах массивов NumPy — впечатляющее достижение для однострочного фрагмента кода. Закрепим достигнутый успех, решив небольшую, но практичную задачу исследования данных: выявление аномальных значений в измерениях загрязнений для различных городов.

Обнаружение аномальных значений с помощью условного поиска по массиву, фильтрации и транслирования

В этом однострочнике мы будем изучать данные о загрязненности воздуха в городах. А именно, по представленному двумерному массиву NumPy с данными измерений загрязнений (столбцы) для нескольких городов (строки) мы найдем города с загрязнением выше среднего. Полученные при чтении этого раздела навыки пригодятся вам при поиске аномальных значений в различных наборах данных.

Общее описание

Индекс качества воздуха (Air Quality Index, AQI) служит для оценки опасности вредного воздействия на здоровье и часто применяется для сравнения качества воздуха в различных городах. В следующем однострочнике мы будем исследовать AQI четырех городов: Гонконга, Нью-Йорка, Берлина и Монреаля.

Данный однострочник выявляет города, загрязненные выше среднего, то есть такие, максимальное значение AQI которых выше общего среднего значения по всем измерениям всех городов.

Важная составляющая нашего решения: поиск элементов в массиве NumPy, удовлетворяющих заданному условию. Это распространенная задача в data science, с которой вы будете сталкиваться очень часто.

Итак, разберемся, как найти элементы массива, удовлетворяющие определенному условию. NumPy предоставляет функцию `nonzero()` для поиска индексов элементов в массиве, которые не равны нулю. В листинге 3.12 приведен пример.

Листинг 3.12. Функция nonzero

```
import numpy as np

X = np.array([[1, 0, 0],
              [0, 2, 2],
              [3, 0, 0]])

print(np.nonzero(X))
```

Результат представляет собой кортеж из двух массивов NumPy:

```
(array([0, 1, 1, 2], dtype=int64), array([0, 1, 2, 0], dtype=int64))
```

Первый массив содержит индексы строк, а второй — индексы столбцов ненулевых элементов. В исходном двумерном массиве содержится четыре ненулевых элемента: 1, 2, 2 и 3, на позициях `X[0,0]`, `X[1,1]`, `X[1,2]` и `X[2,0]`.

Как же с помощью функции `nonzero()` найти в массиве элементы, удовлетворяющие определенному условию? Для этого обратимся еще к одной замечательной возможности NumPy: булевым операциям над массивами, выполняемым с помощью транслирования (листинг 3.13)!

Листинг 3.13. Транслирование и поэлементные булевы операторы в NumPy

```
import numpy as np

X = np.array([[1, 0, 0],
              [0, 2, 2],
              [3, 0, 0]])

print(X == 2)
"""
[[False False False]
 [False  True  True]
 [False False False]]
"""
```

Транслирование происходит при копировании (по существу) целочисленного значения 2 в новый массив той же формы, что и исходный. Далее NumPy

производит поэлементное сравнение всех целочисленных значений со значением 2 и возвращает полученный в результате булев массив.

В нашем основном коде для поиска элементов, удовлетворяющих определенному условию, мы воспользуемся сочетанием функции `nonzero()` и операций над булевыми массивами.

Код

В листинге 3.14 мы найдем в наборе данных города с максимумами загрязнения, превышающими среднее значение.

Листинг 3.14. Однострочное решение, использующее транслирование, булевы операторы и выборочный доступ по индексу

```
## Зависимости
import numpy as np

## Данные: измерения индекса качества воздуха, AQI (строка = город)
X = np.array(
    [[ 42, 40, 41, 43, 44, 43 ], # Гонконг
     [ 30, 31, 29, 29, 29, 30 ], # Нью-Йорк
     [ 8, 13, 31, 11, 11, 9 ], # Берлин
     [ 11, 11, 12, 13, 11, 12 ]]) # Монреаль

cities = np.array(["Hong Kong", "New York", "Berlin", "Montreal"])

## Однострочник
polluted = set(cities[np.nonzero(X > np.average(X))[0]])

## Результат
print(polluted)
```

Можете определить, какими будут результаты выполнения этого кода?

Принцип работы

Массив данных `X` содержит четыре строки (по одной для каждого города) и шесть столбцов (по одному для каждого отрезка измерения — в данном случае дня). Строковый массив `cities` содержит четыре названия городов в том порядке, в каком те встречаются в массиве с данными.

Вот однострочник для поиска городов, в которых наблюдается уровень AQI выше среднего:

```
## Однострочник
polluted = set(cities[np.nonzero(X > np.average(X))[0]])
```

Чтобы понять, как он работает в целом, необходимо сначала разобраться в каждой из его составных частей. Проанализируем его, начав изнутри. В его сердцевине находится операция над булевым массивом (листинг 3.15).

Листинг 3.15. Операция над булевым массивом с помощью транслирования

```
print(X > np.average(X))
"""
[[ True  True  True  True  True  True]
 [ True  True  True  True  True  True]
 [False False  True False False False]
 [False False False False False False]]
"""
```

Чтобы привести оба операнда к одной форме с помощью транслирования, мы воспользовались булевым выражением. Для вычисления среднего по всем элементам нашего массива NumPy значения AQI мы задействуем функцию `np.average()`. Далее булево выражение производит поэлементное сравнение, и получается булев массив, содержащий `True`, если соответствующее измерение превышает среднее значение AQI.

Благодаря генерации этого булева массива мы знаем в точности, какие элементы удовлетворяют условию «выше среднего», а какие — нет.

Напомним, что значение `True` языка Python представлено значением `1` типа integer, а `False` — `0`. На самом деле тип объектов `True` и `False` — `bool` — является подклассом `int`. Таким образом, каждое булево значение является также и целочисленным значением. Благодаря этому мы можем воспользоваться функцией `nonzero()` для поиска всех удовлетворяющих условию индексов строк и столбцов, вот так:

```
print(np.nonzero(X > np.average(X)))
"""
(array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2], dtype=int64),
array([0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 2], dtype=int64))
"""
```

Получаем два кортежа, первый — с индексами строк ненулевых элементов, а второй — с индексами соответствующих им столбцов.

Нам нужны только названия городов со значениями AQI выше среднего, и ничего больше, поэтому нас интересуют только индексы строк, которыми

мы можем воспользоваться для извлечения символьных названий городов из нашего строкового массива с помощью *расширенного доступа по индексу* (advanced indexing) — методики доступа по индексу, позволяющей описывать не непрерывную последовательность индексов массива.

Таким образом можно обращаться к произвольным элементам данного массива NumPy, указывая последовательность либо целых чисел (выбираемых индексов), либо булевых значений (для выбора тех индексов, для которых соответствующее булево значение равно `True`):

```
print(cities[np.nonzero(X > np.average(X))[0]])
"""
['Hong Kong' 'Hong Kong' 'Hong Kong' 'Hong Kong' 'Hong Kong' 'Hong Kong'
 'New York' 'New York' 'New York' 'New York' 'New York' 'New York' 'Berlin']
"""
```

Как видите, в полученной последовательности строковых значений немало повторов, поскольку в числе измерений AQI Гонконга и Нью-Йорка много значений выше среднего.

Осталось только убрать эти повторы. Для этого мы преобразуем нашу последовательность во множество Python, в котором по определению отсутствуют дублирующиеся значения, и получим краткую сводку названий всех городов, степень загрязнения воздуха в которых превышает средние значения AQI.

УПРАЖНЕНИЕ 3.1

Вернитесь к примеру с налогообложением в разделе «Простейшие операции с двумерными массивами» на с. 72 и извлеките из матрицы имя сотрудника с самой высокой зарплатой, применив вышеупомянутую методику выборочного булева доступа по индексу. Краткое резюме задачи: как найти человека с максимальным доходом после уплаты налогов в группе людей при заданных годовых зарплатах и ставках налогообложения?

Резюмируя: вы научились использовать булевы выражения для массивов NumPy (опять же, с помощью транслирования) и функцию `nonzero()` для поиска строк или столбцов, удовлетворяющих определенным условиям. Позанимавшись охраной окружающей среды в этом однострочнике, перейдем к влиятельным блогерам в социальных медиа.

Фильтрация двумерных массивов с помощью булева доступа по индексу

В этом разделе вы закрепите свои навыки доступа к массивам по индексу и транслирования на примере извлечения пользователей Instagram более чем со 100 миллионами подписчиков из небольшого набора данных. А именно, мы выясним имена всех известных блогеров более чем со 100 миллионами подписчиков по заданному двумерному массиву блогеров (строки), в котором первый столбец задает имя блогера в виде строкового значения, а второй — количество подписчиков этого блогера.

Общее описание

Массивы NumPy расширяют простой тип данных списка дополнительной функциональностью, например многомерными срезами и многомерным доступом по индексу. Взгляните на фрагмент кода в листинге 3.16.

Листинг 3.16. Выборочный (булев) доступ по индексу в NumPy

```
import numpy as np

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

indices = np.array([[False, False, True],
                    [False, False, False],
                    [True, True, False]])

print(a[indices])
# [3 7 8]
```

Мы создали два массива: `a`, содержащий двумерные числовые данные (можете считать его *массивом данных*), и `indices`, содержащий булевы значения (можете считать его *массивом индексов*). В NumPy замечательно то, что можно использовать булев массив для весьма выборочного доступа к массиву данных. Попросту говоря, мы создали новый массив, содержащий только те элементы массива данных `a`, для которых в соответствующих местах массива индексов `indices` содержатся значения `True`. Например, если `indices[i,j]==True`, то наш новый массив содержит значение `a[i,j]`. Аналогично, если `indices[i,j]==False`, то наш новый массив не содержит значения `a[i,j]`. Таким образом, итоговый массив содержит три значения: `3`, `7` и `8`.

В следующем однострочнике воспользуемся этой возможностью, чтобы провести небольшой анализ социальной сети.

Код

В листинге 3.17 мы найдем имена суперзвезд Instagram более чем со 100 миллионами подписчиков!

Листинг 3.17. Однострочное решение, использующее срезы, типы массивов и булевы операторы

```
## Зависимости
import numpy as np

## Данные: популярные учетные записи Instagram (миллионы подписчиков)
inst = np.array([[232, "@instagram"],
                 [133, "@selenagomez"],
                 [59,  "@victoriassecret"],
                 [120, "@cristiano"],
                 [111, "@beyonce"],
                 [76,  "@nike"]])

## Однострочник
superstars = inst[inst[:,0].astype(float) > 100, 1]

## Результат
print(superstars)
```

Как обычно, проверьте, сможете ли мысленно произвести вычисления этого однострочника, прежде чем читать пояснения.

Принцип работы

Наши данные состоят из двумерного массива, `inst`, каждая строка которого описывает известного блогера Instagram. Первый столбец отражает количество их подписчиков (в миллионах), а вторая — имена в Instagram. Из этих данных мы хотим извлечь имена известных блогеров Instagram, у которых более чем 100 миллионов подписчиков.

Существует несколько способов решить эту задачу в одной строке кода. Вот простейший подход:

```
## Однострочник
superstars = inst[inst[:,0].astype(float) > 100, 1]
```

Разберем этот однострочник по частям. Внутреннее выражение вычисляет булево значение, указывающее, превышает ли 100 миллионов количество подписчиков каждого из наших блогеров:

```
print(inst[:,0].astype(float) > 100)
# [ True True False True True False]
```

Количество подписчиков содержится в первом столбце, так что мы воспользуемся срезом для доступа к этим данным: `inst[:,0]` возвращает все строки, но только в первом столбце. Впрочем, поскольку массив данных содержит различные типы данных (целочисленные и строковые значения), NumPy автоматически выбирает для массива нечисловой тип данных. Дело в том, что числовой не подошел бы для строковых значений, поэтому NumPy преобразует данные в тип, способный отражать все данные из массива (строковые и целочисленные). Необходимо произвести числовое сравнение значений из первого столбца массива данных с числом 100, для чего мы сначала преобразуем полученный массив в тип данных с плавающей точкой с помощью `.astype(float)`[1].

Далее необходимо проверить, превышают ли значения в массиве NumPy с плавающей точкой целочисленное значение 100. В данном случае NumPy автоматически приводит два операнда в одинаковую форму с помощью транслирования, чтобы произвести поэлементное сравнение. Результат представляет собой массив булевых значений, из которого видно, что у четырех из наших влиятельных блогеров более 100 миллионов подписчиков.

Теперь можно на основе этого булева массива (называемого также *массивом индексов с маскированием* (mask index array)) выбрать блогеров более чем со 100 миллионами подписчиков (строки) с помощью булева доступа по индексу:

```
inst[inst[:,0].astype(float) > 100, 1]
```

А поскольку нам нужны только имена этих популярных блогеров, мы выбираем из результата второй столбец в качестве окончательного результата и сохраняем его в переменной `superstars`.

В итоге получаем следующий список блогеров более чем со 100 миллионами подписчиков из нашего набора данных:

```
# ['@instagram' '@selenagomez' '@cristiano' '@beyonce']
```

Подводя итог: мы воспользовались такими понятиями NumPy, как срезы, транслирование, булев доступ по индексу и преобразование типов данных,

[1] Возможно, лучше было привести к целочисленному типу, например int16.

чтобы решить небольшую задачу исследования данных в сфере анализа социальных медиа. Далее вам предстоит увидеть еще одну сферу приложения — интернет вещей.

Очистка каждого i-го элемента массива с помощью транслирования, присваивания срезу и изменения формы

Реальные данные редко бывают «чистыми» по многим причинам, включая поврежденные и сбойные датчики, они могут содержать ошибки и пропущенные значения. В этом разделе вы научитесь производить небольшие операции очистки для устранения ошибочных точек данных.

Общее описание

Представьте, что вы установили в саду датчик температуры. Каждое воскресенье вы приносите его домой для снятия показаний. А значит, отдаете себе отчет в неточности воскресных показаний, поскольку часть дня они отражают температуру в доме, а не в саду.

Вы хотели бы очистить данные, заменив все воскресные показания датчика средним значением за предыдущие семь дней (воскресное показание вы тоже учитываете, поскольку нельзя сказать, что оно совсем неточное). Прежде чем заняться кодом, посмотрим, что нам понадобится для работы.

Присваивание срезам

При использовании возможности присваивания срезам NumPy (см. раздел «Работа с массивами NumPy: срезы, транслирование и типы массивов» на с. 77) с левой стороны уравнения указываются значения, которые необходимо заменить, а справа — значения, которыми их заменяют. В листинге 3.18 приведен небольшой пример, который освежит вашу память.

Листинг 3.18. Простой пример создания списка и присваивания срезу

```
import numpy as np

a = np.array([4] * 16)
print(a)
# [4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4]
```

```
a[1::] = [42] * 15
print(a)
# [ 4 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42]
```

Этот фрагмент кода создает массив, в котором 16 раз повторяется значение 4. С помощью присваивания срезу мы заменяем последние 15 значений на 42. Напомним, что нотация a[*начало*:*конец*:*шаг*] служит для выбора последовательности, начинающейся с индекса *начало*, оканчивающейся на индексе *конец* (не включая его), причем состоящей только из элементов *шаг* последовательности. Если какие-либо из этих параметров не указаны, NumPy использует значения по умолчанию. Нотация a[1::] заменяет все элементы последовательности, кроме первого. В листинге 3.19 показано, как применять присваивание срезу вместе с уже неоднократно встречавшейся вам функциональной возможностью.

Листинг 3.19. Присваивание срезу в NumPy

```
import numpy as np

a = np.array([4] * 16)

a[1:8:2] = 16
print(a)
# [ 4 16  4 16  4 16  4 16  4  4  4  4  4  4  4  4]
```

Здесь мы заменяем каждое второе значение между индексами 1 и 8 (не включая последний). Как видите, для замены выбранных элементов достаточно указать единственное значение, 16, благодаря — да, вы правы! — *транслированию*. Правая сторона уравнения автоматически преобразуется в массив NumPy той же формы, что и массив слева.

Изменение формы

Прежде чем говорить о нашем однострочнике, вам нужно познакомиться с важной функцией NumPy — x.reshape((a,b)), — которая преобразует массив NumPy x в новый массив NumPy с a строк и b столбцов (то есть формы (a,b)). Пример выглядит так:

```
a = np.array([1, 2, 3, 4, 5, 6])
print(a.reshape((2, 3)))
'''
[[1 2 3]
 [4 5 6]]
'''
```

Если количество столбцов определяется однозначно, можно позволить NumPy сделать это автоматически. Пусть нам нужно изменить форму массива из шести элементов на двумерный массив из двух строк. NumPy может сама вычислить, исходя из равного 6 количества элементов исходного массива, что столбцов должно быть три. Пример таков:

```
a = np.array([1, 2, 3, 4, 5, 6])
print(a.reshape((2, -1)))
'''
[[1 2 3]
 [4 5 6]]
'''
```

Значение `-1` для соответствующего столбцам аргумента функции `reshape` указывает NumPy на необходимость заменить его правильным количеством столбцов (в данном случае три).

Аргумент axis

Наконец, рассмотрим следующий фрагмент кода, в котором вы познакомитесь с аргументом `axis`. Пусть дан массив `solar_x` с ежедневными котировками компании SolarX Илона Маска. Мы хотим вычислить средний курс акций утром, днем и вечером. Как это реализовать?

```
import numpy as np

# котировки акций по дням
# [утро, полдень, вечер]
solar_x = np.array(
    [[1, 2, 3], # сегодня
     [2, 2, 5]]) # вчера

# полдень — взвешенное среднее
print(np.average(solar_x, axis=0))
# [1.5 2.  4. ]
```

Массив `solar_x` состоит из курсов акций компании SolarX. Он включает две строки (по одной на день) и три столбца (по одному на каждый курс акций). Пусть нам нужно вычислить средний курс акций для утра, дня и вечера. В общих чертах мы хотим схлопнуть все значения в каждом столбце путем их усреднения. Другими словами, вычислить среднее значение по оси 0. Именно это и означает аргумент `axis=0`.

Код

Вот и все, что нужно знать для решения следующей задачи (листинг 3.20): заменить каждое седьмое значение в заданном массиве значений температуры средним значением за последние семь дней (включая значение этого седьмого дня).

Листинг 3.20. Однострочное решение, использующее операции усреднения и изменения формы, присваивания срезу и аргумент axis

```
## Зависимости
import numpy as np

## Данные
## Данные с датчиков (Пнд, Вт, Ср, Чт, Пт, Сб, Вс)
tmp = np.array([1, 2, 3, 4, 3, 4, 4,
                5, 3, 3, 4, 3, 4, 6,
                6, 5, 5, 5, 4, 5, 5])

## Однострочник
tmp[6::7] = np.average(tmp.reshape((-1,7)), axis=1)

## Результат
print(tmp)
```

Можете вычислить результат выполнения этого фрагмента кода?

Принцип работы

Исходные данные поступили в виде одномерного массива показаний датчиков.

Во-первых, мы создали массив `tmp`, содержащий одномерную последовательность показаний датчиков. В каждой строке описано семь значений измерений датчиков для семи дней недели.

Во-вторых, с помощью присваивания срезу мы заменяем все воскресные значения этого массива. Поскольку воскресенье — седьмой день, мы выбираем соответствующие воскресенью значения, начиная с седьмого элемента исходного массива `tmp`, с помощью выражения `tmp[6::7]`.

В-третьих, мы *изменяем форму* нашего одномерного массива показаний датчиков на двумерный массив, содержащий семь столбцов и три строки, что

упрощает вычисление средней температуры за неделю, для замены воскресных данных. Благодаря изменению формы мы можем теперь агрегировать все семь значений каждой строки в одно среднее значение. Для изменения формы передаем в `tmp.reshape()` кортеж значений `-1` и `7`, указывая тем самым NumPy, что количество строк *(ось координат 0)* необходимо вычислить автоматически. Проще говоря, мы указываем, что столбцов должно быть семь, а NumPy создает массив с таким количеством строк, которое бы удовлетворяло этому условию. В нашем случае после смены формы получается следующий массив:

```
print(tmp.reshape((-1,7)))
"""
[[1 2 3 4 3 4 4]
 [5 3 3 4 3 4 6]
 [6 5 5 5 4 5 5]]
"""
```

У нас теперь есть по строке для каждой недели и по столбцу для каждого дня недели.

Теперь можно вычислить среднее значение за семь дней, схлопнув каждую строку в одно число с помощью функции `np.average()` с аргументом `axis=1`, указывающим NumPy схлопнуть вторую ось в одно среднее значение. Обратите внимание, что значение за воскресенье также включено в вычисляемое среднее (см. постановку задачи в начале этого раздела). Ниже представлен результат правой половины уравнения:

```
print(np.average(tmp.reshape((-1,7)), axis=1))
# [3. 4. 5.]
```

Задача нашего однострочника состоит в замене трех воскресных значений температуры. Все остальные значения менять не нужно. Посмотрим, как можно добиться этого. После замены всех воскресных показаний датчиков окончательный результат работы нашего однострочника должен выглядеть следующим образом:

```
# [1 2 3 4 3 4 3 5 3 3 4 3 4 4 6 5 5 5 4 5 5]
```

Обратите внимание, что в итоге получается снова одномерный массив NumPy со всеми показаниями датчиков температуры. Но все «неправильные» значения заменены теми, которые лучше отражают реальность.

Резюмируя: данный однострочник иллюстрирует понятие формы массива и ее изменения, а также использование свойства `axis` в агрегирующих функциях наподобие `np.average()`. И несмотря на некоторую специфичность этого конкретного их приложения, они могут пригодиться во множестве ситуаций. Далее мы расскажем вам о чрезвычайно широком понятии: сортировке в NumPy.

Когда использовать в NumPy функцию sort(), а когда — argsort()

Сортировка удобна и даже необходима во множестве ситуаций. Скажем, вы хотите найти на полке книгу «Лаконичный Python. Однострочники и регулярные выражения». Правда же, искать ее будет намного проще, если книги на полке отсортированы по алфавиту?

Общее описание

Сортировка — центральный элемент более сложных приложений, таких как коммерческие расчеты, планирование процессов в операционных системах (очереди по приоритету) и поисковые алгоритмы. К счастью, библиотека NumPy предоставляет самые разнообразные алгоритмы сортировки. По умолчанию используется популярный алгоритм *быстрой сортировки*. В главе 6 вы узнаете, как реализовать алгоритм быстрой сортировки самостоятельно. Однако в следующем однострочнике мы будем придерживаться более высокоуровневого подхода и рассматривать функцию сортировки как «черный ящик», на входе которого — массив NumPy, а на выходе — все он же, но уже отсортированный.

На рис. 3.1 приведен алгоритм преобразования неотсортированного массива в отсортированный — цель функции `sort()` NumPy.

Но зачастую бывает необходимо получить массив индексов, преобразующий неупорядоченный массив в отсортированный. Например, пусть индекс элемента 1 неупорядоченного массива — 7. Поскольку элемент 1 — первый элемент отсортированного массива, его индекс 7 будет первым элементом индексов отсортированного массива. Именно для этого и служит функция `argsort()`: она создает новый массив индексов исходного массива после сортировки (см. пример на рис. 3.1). Попросту говоря, этими индексами можно

воспользоваться для сортировки элементов исходного массива. С помощью данного массива можно восстановить как отсортированный, так и исходный массив.

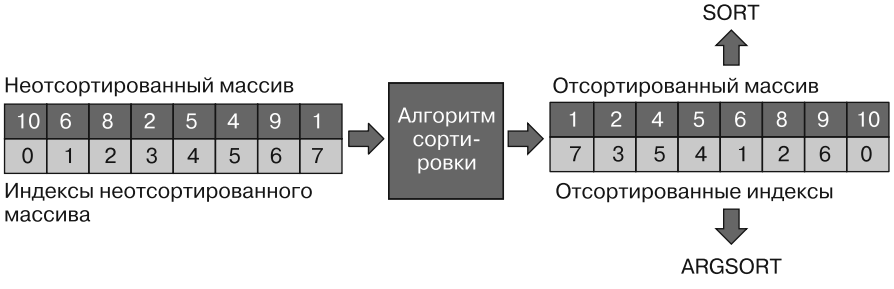


Рис. 3.1. Различие между функциями sort() и argsort()

Листинг 3.21 демонстрирует использование в NumPy функций `sort()` и `argsort()`.

Листинг 3.21. Функции sort() и argsort() в в NumPy

```
import numpy as np

a = np.array([10, 6, 8, 2, 5, 4, 9, 1])
print(np.sort(a))
# [ 1  2  4  5  6  8  9 10]

print(np.argsort(a))
# [7 3 5 4 1 2 6 0]
```

Мы создали неупорядоченный массив `a`, отсортировали его с помощью вызова `np.sort(a)`, а также получили исходные индексы в новом отсортированном порядке с помощью вызова `np.argsort(a)`. Функция `sort()` библиотеки NumPy отличается от функции `sorted()` языка Python тем, что может сортировать и многомерные массивы!

На рис. 3.2 приведены два способа сортировки двумерного массива.

У данного массива — две оси координат: ось 0 (строки) и ось 1 (столбцы). Сортировать можно либо по оси 0 (так называемая *вертикальная сортировка*), либо по оси 1 (так называемая *горизонтальная сортировка*). Вообще говоря, ключевое слово `axis` задает направление, по которому происходит операция NumPy. Листинг 3.22 демонстрирует технические подробности.

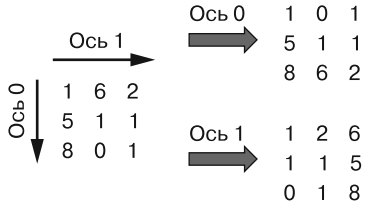


Рис. 3.2. Сортировка по разным осям координат

Листинг 3.22. Сортировка по различным осям

```
import numpy as np

a = np.array([[1, 6, 2],
              [5, 1, 1],
              [8, 0, 1]])

print(np.sort(a, axis=0))
"""
[[1 0 1]
 [5 1 1]
 [8 6 2]]
"""

print(np.sort(a, axis=1))
"""
[[1 2 6]
 [1 1 5]
 [0 1 8]]
"""
```

Необязательный аргумент `axis` служит для сортировки массива NumPy по конкретной оси. Сначала мы отсортировали по столбцам, начиная с минимального значения. А затем — по строкам. В этом главное преимущество функции `sort()` библиотеки NumPy по сравнению со встроенной функцией `sorted()` языка Python.

Код

Следующий однострочник выясняет имена трех абитуриентов с самыми высокими оценками SAT. Обратите внимание, что нам нужны имена абитуриентов, а не отсортированные оценки SAT. Взгляните на данные и попытайтесь сами найти однострочное решение. И лишь затем разберите листинг 3.23.

Листинг 3.23. Однострочное решение, включающее функцию argsort() и срез с отрицательным значением шага

```
## Зависимости
import numpy as np

## Данные: оценки за экзамен SAT для различных абитуриентов
sat_scores = np.array([1100, 1256, 1543, 1043, 989, 1412, 1343])
students = np.array(["John", "Bob", "Alice", "Joe", "Jane", "Frank", "Carl"])

## Однострочник
top_3 = students[np.argsort(sat_scores)][:-4:-1]

## Результат
print(top_3)
```

Как обычно, попробуйте догадаться, какими будут результаты.

Принцип работы

Наши исходные данные состоят из оценок SAT абитуриентов в виде одномерного массива данных и еще одного массива с соответствующими именами абитуриентов. Например, Джон набрал на этом экзамене вполне приличную оценку в 1100, а Фрэнк показал великолепный результат — 1412.

Наша задача — выяснить имена трех лучших абитуриентов. Для этого мы не просто отсортировали оценки SAT, а воспользовались функцией `argsort()`, чтобы получить массив с исходными индексами в новых, отсортированных позициях.

Вот результаты работы функции `argsort()` для оценок SAT:

```
print(np.argsort(sat_scores))
# [4 3 0 1 6 5 2]
```

Индексы необходимо сохранить, чтобы узнать имена абитуриентов из массива `students`, соответствующие исходным позициям в массиве. На первой позиции результата находится индекс 4, поскольку у Джейн самая низкая оценка SAT, равная 989 баллам. Обратите внимание, что и `sort()`, и `argsort()` сортируют в порядке возрастания, от самых низких значений к самым высоким.

Получив отсортированные индексы, можно узнать имена соответствующих абитуриентов с помощью доступа по индексу к массиву `students`:

```
print(students[np.argsort(sat_scores)])
# ['Jane' 'Joe' 'John' 'Bob' 'Carl' 'Frank' 'Alice']
```

Возможность библиотеки NumPy переупорядочивать последовательность с помощью расширенного доступа по индексу очень удобна. Если указать последовательность индексов, то NumPy запускает расширенный доступ по индексу и возвращает новый массив NumPy с элементами, переупорядоченными так, как указано в этой последовательности. Например, результат вычисления команды `students[np.argsort(sat_scores)]` равен `students[[4 3 0 1 6 5 2]]`, вследствие чего NumPy создает следующий новый массив:

```
[students[4]  students[3]  students[0]  students[1]  students[6]  students[5]
students[2]]
```

Из этого ясно, что у Джейн самые низкие оценки SAT, а у Алисы — самые высокие. Осталось только инвертировать список и извлечь из него трех лучших абитуриентов с помощью простого среза:

```
## Однострочник
top_3 = students[np.argsort(sat_scores)][:-4:-1]

## Результат
print(top_3)
# ['Alice' 'Frank' 'Carl']
```

У Алисы, Фрэнка и Карла самые высокие оценки SAT: 1543, 1412 и 1343 соответственно.

Итак, вы изучили приложение двух важных функций NumPy: `sort()` и `argsort()`. Далее вам предстоит еще больше улучшить свое знание доступа по индексу и срезов в NumPy, воспользовавшись булевым доступом по индексу и лямбда-функциями в практическом примере исследования данных.

Фильтрация массивов с помощью лямбда-функций и булева доступа по индексу

Встречающиеся на практике данные обычно зашумлены. Исследователям, собственно, и платят деньги за то, чтобы очистить данные от этого шума, сделать их удобными для восприятия и выяснить содержащийся в них смысл. А значит, фильтрация данных жизненно важна для практических

приложений науки о данных. В этом разделе показано, как создать простейшую функцию для фильтрации из одной строки кода.

Общее описание

Для создания однострочной функции нам понадобятся *лямбда-функции*. Как вы помните из главы 2, это анонимные функции, которые можно описать в одной строке кода:

```
lambda аргументы : выражение
```

Разделенный запятыми список аргументов играет роль входных данных. Лямбда-функция вычисляет выражение и возвращает результат.

Посмотрим, как решить поставленную задачу путем создания фильтрующей функции, описанной в виде лямбда-функции.

Код

Рассмотрим следующую задачу, показанную в листинге 3.24: создать функцию для фильтрации, принимающую на входе список книг `x` и их рейтинг `y` и возвращающую список потенциальных бестселлеров, рейтинг которых превышает заданное пороговое значение: `y'>y`.

Листинг 3.24. Однострочное решение, использующее лямбда-функции, преобразование типов и булевы операторы

```
## Зависимости
import numpy as np

## Данные (строка = [название, рейтинг])
books = np.array([['Coffee Break NumPy', 4.6],
                  ['Lord of the Rings', 5.0],
                  ['Harry Potter', 4.3],
                  ['Winnie-the-Pooh', 3.9],
                  ['The Clown of God', 2.2],
                  ['Coffee Break Python', 4.7]])

## Однострочник
predict_bestseller = lambda x, y : x[x[:,1].astype(float) > y]

## Результат
print(predict_bestseller(books, 3.9))
```

Прежде чем читать дальше, попробуйте догадаться, какие результаты даст этот код.

Принцип работы

Наши данные состоят из двумерного массива NumPy, каждая строка которого содержит название книги и средний рейтинг ее у читателей (число с плавающей точкой от 0.0 до 5.0). В этом наборе данных — шесть книг.

Наша цель состоит в создании функции фильтрации, принимающей на входе набор данных о рейтингах книг `x` и пороговое значение рейтинга `y` и возвращающей список книг, рейтинг которых превышает это пороговое значение `y`. В данном случае мы установили пороговое значение равным `3.9`.

Для этого мы описали анонимную лямбда-функцию, возвращающую результат следующего выражения:

```
x[❶x[:,1] ❷.astype(float)❸> y]
```

В массиве `x` должно быть два столбца, как и в нашем массиве рейтингов книг `books`. Для доступа к потенциальным бестселлерам мы воспользовались схемой расширенного доступа по индексу, которая похожа на схему из листинга 3.17.

Во-первых, мы извлекли второй столбец ❶ с рейтингами книг и преобразовали его в массив значений с плавающей точкой с помощью метода `astype(float)` ❷ массива NumPy `x`. Это необходимо, поскольку исходный массив `x` состоит из данных различных типов (значения с плавающей точкой и строковые значения).

Во-вторых, создали булев массив, содержащий значение `True`, если рейтинг книги в строке с соответствующим индексом превышает `y` ❸. Обратите внимание, что значение с плавающей точкой `y` неявно транслируется на новый массив NumPy, поэтому формы обоих операндов булева оператора `>` одинаковы. Далее мы создали булев массив, элементы которого указывают для каждой книги, можно ли ее считать бестселлером: `x[:,1].astype(float)> y = [ True True True False False True]`. Таким образом, первые три книги и последняя — бестселлеры.

В-третьих, мы воспользовались упомянутым булевым массивом в качестве массива индексов для исходного массива рейтингов книг, чтобы извлечь все

книги с рейтингом, превышающим пороговое значение. Точнее говоря, воспользовались булевым доступом по индексу `x[[ True True True False False True]]`, чтобы получить подмассив, содержащий только четыре книги, которым соответствует значение `True`. И получаем следующий окончательный результат работы нашего однострочника:

```
## Результат
print(predict_bestseller(books, 3.9))
"""
[['Coffee Break NumPy' '4.6']
 ['Lord of the Rings' '5.0']
 ['Harry Potter' '4.3']
 ['Coffee Break Python' '4.7']]
"""
```

Резюмируя: вы научились фильтровать данные с помощью одного только булева доступа по индексу и лямбда-функций. Далее мы займемся логическими операторами и изучим полезный трюк для лаконичного написания операции логического И.

Создание расширенных фильтров массивов с помощью статистических показателей, а также математических и логических операций

В этом разделе вы познакомитесь с простейшим алгоритмом обнаружения аномальных значений: наблюдаемое значение считается *аномальным* (outlier), если отличается от среднего более чем на стандартное отклонение. Мы пройдемся по примеру анализа данных сайта в целях определения количества его активных пользователей, показателя отказов и средней длительности сеанса в секундах. (*Показатель отказов* (bounce rate) — это процент посетителей, которые уходят с сайта сразу же после посещения одной страницы. Высокий показатель отказов — плохой сигнал, означающий, что сайт, возможно, неинтересен или бесполезен.) Мы рассмотрим данные и выявим аномальные значения.

Общее описание

Чтобы решить задачу обнаружения аномальных значений, вам сначала нужно разобраться, что такое среднее значение и стандартное отклонение,

как вычислить абсолютное значение и выполнить операцию логического И.

Среднее значение и стандартное отклонение

Во-первых, мы понемногу сформулируем определение аномального значения на основе простейших статистических понятий. Предположим, что все наблюдаемые данные нормально распределены вокруг среднего значения. Например, рассмотрим следующую последовательность значений данных:

```
[ 8.78087409 10.95890859  8.90183201  8.42516116  9.26643393 12.52747974
  9.70413087 10.09101284  9.90002825 10.15149208  9.42468412 11.36732294
  9.5603904   9.80945055 10.15792838 10.13521324 11.0435137  10.06329581
--сокращено--
 10.74304416 10.47904781]
```

Если построить гистограмму этой последовательности, то получится вот что (рис. 3.3).

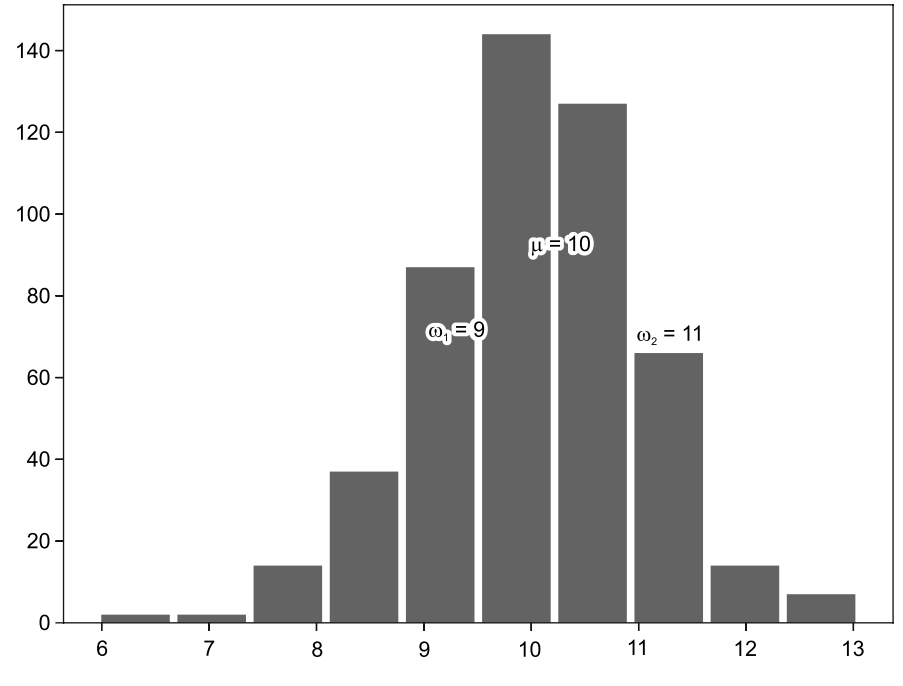


Рис. 3.3. Гистограмма последовательности значений данных

Эта последовательность напоминает *нормальное распределение* с *математическим ожиданием* 10 и *стандартным отклонением* 1. Математическое ожидание, обозначаемое символом μ, представляет собой среднее значение по всем значениям последовательности. Стандартное отклонение, обозначаемое символом σ, представляет собой меру отклонения набора данных от среднего значения. По определению, в случае истинно нормального распределения данных 68,2 % всех точек данных попадает в интервал стандартного отклонения $[\omega_1 = \mu - \sigma, \omega_2 = \mu + \sigma]$. Из этого следует определение аномальных значений: любое значение, не попадающее в упомянутый интервал, — аномальное.

В этом примере мы сгенерировали данные из нормального распределения с $\mu = 10$ и $\sigma = 1$, в результате чего получается интервал $\omega_1 = \mu - 1 = 9$ и $\omega_2 = \mu + 1 = 11$. В дальнейшем мы просто предполагаем, что *любое наблюдаемое значение, выходящее за пределы интервала, определяемого стандартным отклонением от математического ожидания, — аномальное*. Применительно к нашим данным это означает, что любое значение, выходящее за рамки интервала [9, 11] — аномальное.

Простой код, с помощью которого я сгенерировал данный график, приведен в листинге 3.25. Попробуйте найти в нем строки с описанием математического ожидания и стандартного отклонения.

Листинг 3.25. Построение гистограммы с помощью библиотеки Matplotlib

```
import numpy as np
import matplotlib.pyplot as plt

sequence = np.random.normal(10.0, 1.0, 500)
print(sequence)

plt.xkcd()
plt.hist(sequence)
plt.annotate(r"$\omega_1=9$", (9, 70))
plt.annotate(r"$\omega_2=11$", (11, 70))
plt.annotate(r"$\mu=10$", (10, 90))
plt.savefig("plot.jpg")
plt.show()
```

Код демонстрирует, как построить гистограмму с помощью библиотеки Matplotlib для Python. Однако для нас здесь это не главное; я лишь хотел подчеркнуть, как можно сгенерировать вышеупомянутую последовательность значений данных.

Достаточно просто импортировать библиотеку NumPy и воспользоваться модулем `np.random`, предоставляющим функцию `normal(`*`математическое_ожидание, отклонение, форма`*`)`, создающую новый массив NumPy, значения которого выбраны из нормального распределения с заданными математическим ожиданием и стандартным отклонением. Именно при ее вызове мы задаем *`математическое_ожидание`*`=10.0` и *`отклонение`*`=1.0` для создания данных в последовательности. В данном случае параметр *`форма`*`=500` указывает, что мы хотим получить одномерный массив данных, включающий 500 точек данных. Оставшийся код импортирует специальный стиль отрисовки графиков `plt.xkcd()`, строит с помощью функции `plt.hist(`*`последовательность`*`)` гистограмму на основе сгенерированной последовательности с нужными метками и выводит полученный график.

ПРИМЕЧАНИЕ

Название графика xkcd взято с популярной страницы веб-комиксов xkcd (https://xkcd.com/).

Прежде чем заняться исследованием нашего однострочника, вкратце рассмотрим два оставшихся навыка, которые нам понадобятся, чтобы довести до конца решение нашей задачи.

Поиск абсолютного значения

Во-вторых, нам придется преобразовывать отрицательные значения в положительные, чтобы проверить, отклоняется ли потенциальное аномальное значение от среднего более чем на стандартное отклонение. Нам важен только модуль отклонения, а не его знак. Это и называется *абсолютным значением*. Функция библиотеки NumPy в листинге 3.26 создает новый массив NumPy, содержащий модули значений исходного массива.

Листинг 3.26. Вычисление абсолютного значения в NumPy

```
import numpy as np

a = np.array([1, -1, 2, -2])

print(a)
# [ 1 -1  2 -2]

print(np.abs(a))
# [1 1 2 2]
```

Функция `np.abs()` преобразует отрицательные значения массива NumPy в соответствующие им положительные.

Операция логического И

В-третьих, следующая функция NumPy производит поэлементную операцию *логического И*, объединяя два булевых массива `a` и `b` и возвращая массив, значения в котором представляют собой комбинацию *отдельных* булевых значений из тех массивов с помощью операции логического И (листинг 3.27).

Листинг 3.27. Применение к массивам NumPy операции логического И

```
import numpy as np

a = np.array([True, True, True, False])
b = np.array([False, True, True, False])

print(np.logical_and(a, b))
# [False  True  True False]
```

Мы сочетаем элемент массива `a` с индексом *i* и элемент массива `b` с таким же индексом, обратившись к операции `np.logical_and(a, b)`. Результат представляет собой массив булевых значений, равных `True`, если оба операнда `a[i]` и `b[i]` равны `True`, и `False` в противном случае. Это позволяет схлопывать несколько булевых массивов в один с помощью стандартных логических операций. Один из удобных сценариев применения этого — объединение *массивов булевых фильтров*, подобно тому, как это было сделано в предыдущем однострочнике.

Обратите внимание: можно *умножить* и два булева массива `a` и `b`, что также эквивалентно логической операции `np.logical_and(a, b)`. Дело в том, что Python представляет значение `True` как целочисленное значение `1` (или даже любое отличное от `0` целочисленное значение), а значение `False` — как целочисленное значение `0`. Если умножить что-либо на `0`, то получится `0`, то есть `False`. Это значит, что `True` (целочисленное значение $\neq 0$) получится только тогда, когда все операнды равны `True`.

С этими знаниями вы теперь полностью готовы понять следующий однострочный фрагмент кода.

Код

Следующий однострочник находит все аномальные дни, для которых статистические показатели отличаются более чем на стандартное отклонение от среднего значения.

Листинг 3.28. Однострочное решение, использующее функции для математического ожидания, стандартного отклонения и булевы операторы с транслированием

```
## Зависимости
import numpy as np

## Данные анализа сайта
## (строка = день), (столбцы = пользователи, отказы, длительность)
a = np.array([[815, 70, 115],
              [767, 80, 50],
              [912, 74, 77],
              [554, 88, 70],
              [1008, 65, 128]])
mean, stdev = np.mean(a, axis=0), np.std(a, axis=0)
# [811.2  76.4  88. ], [152.97764543   6.85857128  29.04479299]

## Однострочник
outliers = ((np.abs(a[:,0] - mean[0]) > stdev[0])
            * (np.abs(a[:,1] - mean[1]) > stdev[1])
            * (np.abs(a[:,2] - mean[2]) > stdev[2]))

## Результат
print(a[outliers])
```

Сможете догадаться, какие результаты выведет этот фрагмент кода?

Принцип работы

Набор данных состоит из строк, соответствующих различным дням, и трех столбцов, отражающих количество активных пользователей за день, показатель отказов и среднюю длительность сеанса в секундах соответственно.

Для каждого столбца мы вычисляем среднее значение и стандартное отклонение. Например, среднее значение столбца «Активных пользователей за день» равно 811,2, а его стандартное отклонение равно 152,97. Обратите внимание, что аргумент `axis` используется аналогично изложенному в разделе «Очистка каждого i-го элемента массива с помощью транслирования, присваивания срезу и изменения формы» на с. 94.

Наша задача — выявить аномальные по всем трем столбцам сайты. Что касается столбца «Активных пользователей за день», то любое наблюдаемое значение меньше 811,2 – 152,97 = 658,23 или больше 811,2 + 152,23 = 963,43 считается аномальным.

Однако целый *день* мы считаем аномальным только в том случае, если значения во всех трех наблюдаемых столбцах являются аномальными. Для этого мы объединяем три булевых массива с помощью оператора логического И. Результат представляет собой единственную строку, в которой все три столбца — аномальные значения:

```
[[1008   65  128]]
```

Резюмируя: вы научились использовать оператор логического И библиотеки NumPy в целях простейшего обнаружения аномальных значений с помощью простых статических средств из библиотеки NumPy. Далее вы узнаете о секретном ингредиенте успеха Amazon: адекватные рекомендации продуктов для покупки.

Простейший анализ ассоциаций: купившие товар X покупают и товар Y

Случалось ли вам покупать товары по рекомендациям Amazon? В основе алгоритмов рекомендации часто лежит методика так называемого *анализа ассоциаций* (association analysis). В этом разделе вы узнаете основные идеи такого анализа и окунетесь в бездну рекомендательных систем.

Общее описание

Анализ ассоциаций основывается на данных об истории покупок, например, можно получить информацию о том, что «купившие товар X покупают и товар Y» на Amazon. Подобная связь различных товаров — один из важнейших принципов маркетинга, поскольку она не только связывает дополняющие друг друга товары, но и дает покупателю определенное *социальное доказательство* — знание о том, что другие люди тоже купили данный товар, делающее психологически более комфортной его покупку. А значит, это прекрасный инструмент для маркетологов.

Рассмотрим практический пример на рис. 3.4.

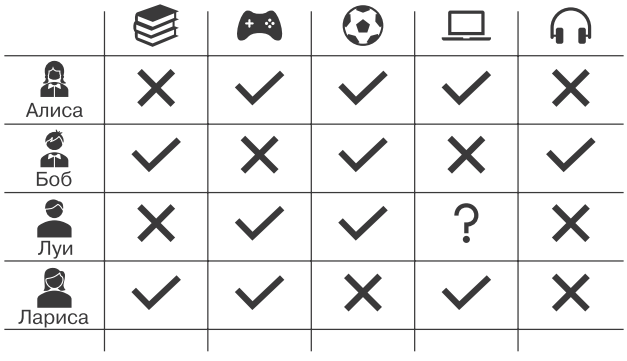


Рис. 3.4. Матрица «Товар — Покупатель»: какой покупатель купил какие товары?

Четыре покупателя — Алиса, Боб, Луи и Лариса — купили различные сочетания товаров: книгу, игру, футбольный мяч, ноутбук и наушники. Пусть вам известны все продукты, купленные каждым из них, за исключением того, что Луи купил ноутбук. Как вы думаете, насколько вероятно, что Луи его купит?

Анализ ассоциаций (*коллаборативная фильтрация*) дает ответ на этот вопрос. В основе его лежит допущение, что два человека, производивших схожие действия в прошлом (например, покупавших схожие товары), скорее всего, будут поступать схожим образом и в будущем. Поведение Луи как покупателя схоже с поведением Алисы, а она купила ноутбук. Следовательно, рекомендательная система должна предсказать, что Луи тоже, вероятно, купит ноутбук.

Следующий фрагмент кода упрощает решение этой задачи.

Код

Попробуем ответить на вопрос: какая доля покупателей купит две книги сразу? На основе этих данных рекомендательная система может предложить покупателям купить «набор» книг, если изначально похоже, что они собираются купить только одну (листинг 3.29).

Листинг 3.29. Однострочное решение, использующее срезы, аргумент axis, свойство shape и простейшие арифметические операции над массивами с транслированием

```
## Зависимости
import numpy as np

## Данные: каждая строка соответствует корзине для покупок конкретного покупателя
```

```
## строка = [курс 1, курс 2, эл. книга 1, эл. книга 2]
## значение 1 означает, что товар был куплен
basket = np.array([[0, 1, 1, 0],
                   [0, 0, 0, 1],
                   [1, 1, 0, 0],
                   [0, 1, 1, 1],
                   [1, 1, 1, 0],
                   [0, 1, 1, 0],
                   [1, 1, 0, 1],
                   [1, 1, 1, 1]])

## Однострочник
copurchases = np.sum(np.all(basket[:,2:], axis = 1)) / basket.shape[0]

## Результат
print(copurchases)
```

Какими же будут результаты выполнения этого фрагмента кода?

Принцип работы

Массив данных `basket` содержит по одной строке для каждого покупателя и по столбцу для каждого товара. Первые два товара с индексами столбцов 0 и 1 — онлайн-курсы, а последние два с индексами столбцов 2 и 3 — электронные книги. Значение `1` в ячейке `(i,j)` означает, что покупатель `i` купил товар `j`.

Наша задача — вычислить долю покупателей, купивших обе электронные книги, так что нас интересуют только столбцы 2 и 3. Следовательно, прежде всего мы выделяем соответствующие столбцы из исходного массива и получаем такой подмассив:

```
print(basket[:,2:])
"""
[[1 0]
 [0 1]
 [0 0]
 [1 1]
 [1 0]
 [1 0]
 [0 1]
 [1 1]]
"""
```

В итоге мы получили массив, состоящий только из третьего и четвертого столбцов.

Функция `all()` библиотеки NumPy проверяет, все ли значения в заданном массиве NumPy равны `True`. В этом случае она возвращает `True`. В противном случае она возвращает `False`. При указании аргумента `axis` функция `all()` делает то же самое, но по заданной оси.

ПРИМЕЧАНИЕ

Вы наверняка обратите внимание, что аргумент `axis` снова и снова встречается во многих функциях NumPy, поэтому имеет смысл потратить немного времени и разобраться с ним как следует. Указанная ось координат схлопывается в одно значение в зависимости от соответствующей агрегирующей функции (в данном случае `all()`).

Следовательно, в результате применения функции `all()` к подмассиву получится следующее:

```
print(np.all(basket[:,2:], axis = 1))
# [False False False  True False False False  True]
```

Говоря простым языком, только четвертый и последний покупатели приобрели обе электронные книги.

А поскольку нас интересует доля покупателей, мы суммируем этот булев массив, получая в результате `2`, и делим полученное на количество покупателей, 8. В результате мы получаем `0.25` — долю покупателей, купивших обе электронные книги.

Резюмируя: вы укрепили свое знание основ библиотеки NumPy, таких как атрибут `shape` и аргумент `axis`, а также научились использовать их вместе для анализа совместных покупок различных товаров. Далее мы продолжим работу над этим примером и изучим более продвинутые методики агрегирования массивов с помощью сочетания нескольких особых возможностей NumPy и Python, а именно *транслирования* и *спискового включения*.

Поиск лучше всего продающихся наборов с помощью промежуточного анализа ассоциаций

Исследуем анализ ассоциаций более подробно.

Общее описание

Возьмем пример из предыдущего раздела: покупатели приобретают отдельные товары из каталога четырех товаров. Ваша компания хотела бы повысить продажи сопутствующих товаров (предложив покупателям дополнительные, зачастую связанные товары). Для каждого сочетания товаров необходимо вычислить, насколько часто их приобретает один и тот же покупатель, и найти два товара, чаще всего приобретаемых вместе.

Вы уже знаете все, что нужно для решения данной задачи, поэтому приступим!

Код

Следующий однострочник предназначен для поиска двух чаще всего приобретаемых вместе товаров (листинг 3.30).

Листинг 3.30. Однострочное решение, использующее лямбда-функцию в качестве параметра key функции max, списковое включение и булевы операторы с транслированием

```
## Зависимости
import numpy as np

## Данные: каждая строка соответствует корзине для покупок конкретного покупателя
## строка = [курс 1, курс 2, эл. книга 1, эл. книга 2]
## значение 1 означает, что товар был куплен
basket = np.array([[0, 1, 1, 0],
                   [0, 0, 0, 1],
                   [1, 1, 0, 0],
                   [0, 1, 1, 1],
                   [1, 1, 1, 0],
                   [0, 1, 1, 0],
                   [1, 1, 0, 1],
                   [1, 1, 1, 1]])

## Однострочник (разбит на две строки)
copurchases = [(i,j,np.sum(basket[:,i] + basket[:,j] == 2))
               for i in range(4) for j in range(i+1,4)]

## Результат
print(max(copurchases, key=lambda x:x[2]))
```

Какими же будут результаты выполнения этого однострочного решения?

Принцип работы

Массив данных состоит из данных о покупках, по одной строке на покупателя и по столбцу на товар. Наша задача — получить список кортежей, каждый из которых описывает конкретное сочетание товаров и частоту покупки этих товаров вместе. Первые два значения кортежа в каждом элементе списка должны быть индексами столбцов (сочетания двух товаров), а третье должно отражать число раз, когда они покупались вместе. Например, кортеж `(0,1,4)` означает, что покупатели, приобретавшие *товар 0*, также 4 раза покупали *товар 1*.

Как же добиться этого? Рассмотрим наш однострочник по частям, немного его переформатировав, поскольку он слишком длинный и не помещается на одной строке книги:

```
## Однострочник (разбит на две строки)
copurchases = [(i,j,np.sum(basket[:,i] + basket[:,j] == 2))
               for i in range(4) for j in range(i+1,4)]
```

Как видно из внешней формы `[(..., ..., ...) for ... in ... for ... in ...]`, мы создали список кортежей с помощью спискового включения (см. главу 2). Нас интересуют все уникальные сочетания индекса столбца массива с четырьмя столбцами. Вот результат одной только внешней части нашего однострочника:

```
print([(i,j) for i in range(4) for j in range(i+1,4)])
# [(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
```

Итак, в списке содержится шесть кортежей, все — уникальные сочетания индексов столбцов.

Теперь можно заняться и третьим элементом кортежа: количеством раз, когда товары `i` и `j` покупали вместе:

```
np.sum(basket[:,i] + basket[:,j] == 2)
```

Мы выделяем оба столбца `i` и `j` из исходного массива NumPy с помощью срезов. А затем складываем их поэлементно. И проверяем поэлементно в полученном массиве, равна ли сумма 2, что означает наличие 1 в обоих столбцах, а значит, и то, что были куплены оба товара. Результат представляет собой булев массив, в котором значения `True` соответствуют покупке обоих товаров вместе одним покупателем.

Все полученные кортежи мы сохраняем в списке `copurchases`. Вот его элементы:

```
print(copurchases)
# [(0, 1, 4), (0, 2, 2), (0, 3, 2), (1, 2, 5), (1, 3, 3), (2, 3, 2)]
```

Теперь осталось только найти два товара, которые покупали вместе чаще всего:

```
## Результат
print(max(copurchases, key=lambda x:x[2]))
```

Для поиска наибольшего элемента списка мы воспользовались функцией `max()`. Мы описали ключевую функцию, принимающую на входе кортеж и возвращающую третье значение кортежа (количество одновременных покупок), а затем нашли максимальное из этих значений. Результат выполнения однострочника выглядит следующим образом:

```
## Результат
print(max(copurchases, key=lambda x:x[2]))
# (1, 2, 5)
```

Второй и третий товары покупали вместе пять раз. Ни одно из прочих сочетаний товаров не достигло таких показателей. Следовательно, можете смело сказать начальнику, что нужно стараться дополнительно продать *товар 2* при продаже *товара 1*, и наоборот.

Итак, вы узнали о разнообразных базовых возможностях как Python, так и NumPy, в частности о транслировании, списковом включении, лямбда-функциях и ключевых функциях. Зачастую своей выразительностью код Python обязан как раз сочетанию множества различных элементов языка, функций и трюков.

Итоги главы

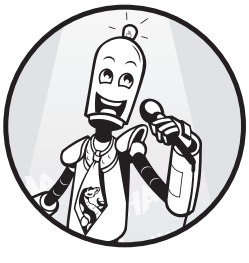
В этой главе вы познакомились с основами NumPy: массивами, формами, осями координат, типами, транслированием, расширенным доступом по индексу, срезами, сортировкой, поиском, агрегированием и статистическими показателями. Вы также улучшили основные навыки работы с Python, поработав на практике с такими важными вещами, как списковое включение, логические операции и лямбда-функции. Наконец, что не менее важно, вы

научились лучше читать, понимать и писать лаконичный код, по ходу дела решая основополагающие задачи науки о данных.

Продолжим столь же быстрое изучение различных интересных вопросов в сфере Python. Далее мы углубимся в захватывающий мир машинного обучения. Вы познакомитесь с простейшими алгоритмами машинного обучения и узнаете, как воспользоваться всеми их возможностями в одной строке кода с помощью популярной библиотеки scikit-learn. Она хорошо знакома любому специалисту по машинному обучению. Но не бойтесь — только что полученные вами навыки работы с NumPy помогут вам разобраться в описанных далее фрагментах кода.

4

Машинное обучение



Машинное обучение (МО) встречается практически во всех сферах computer science. В последние несколько лет я был на конференциях, посвященных распределенным системам, базам данных и потоковой обработке, и везде сталкивался с машинным обучением. На некоторых конференциях в основе более чем половины представленных исследований лежали методы машинного обучения.

Специалисту в области computer science необходимо знать основные понятия и алгоритмы машинного обучения в дополнение к общему набору профессиональных навыков. В этой главе вы познакомитесь с важнейшими алгоритмами и методами МО и разберете десять однострочников, ориентированных на практическое применение этих алгоритмов в ваших проектах.

Основы машинного обучения с учителем

Основная цель машинного обучения — безошибочные прогнозы на основе имеющихся данных. Пусть нам нужно написать алгоритм для предсказания курса конкретных акций на следующие два дня. Для этого необходимо обучить модель МО. Но что такое *модель*?

С точки зрения пользователя машинного обучения, модель МО представляет собой «черный ящик» (рис. 4.1), на вход которого подаются данные, а на выходе получаются предсказания.

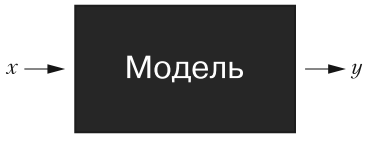


Рис. 4.1. Модель машинного обучения в виде «черного ящика»

В этой модели входные данные — числовое значение или многомерный вектор числовых значений — обозначаются переменной x и называются *признаками* (features). Далее «черный ящик» делает фокус-покус и обрабатывает введенные данные. Через некоторое время он возвращает предсказание y — выходной сигнал модели на основе заданных входных признаков. В задачах регрессии предсказание состоит из одного или нескольких числовых значений — как и входные признаки.

Машинное обучение с учителем делится на два отдельных этапа: обучение и выполнение вывода.

Этап обучения

На *этапе обучения* мы демонстрируем модели, каким должен быть выходной сигнал y' при заданном входном сигнале x. Выданное моделью предсказание y сравнивается с y', и в случае их расхождения модель обновляется так, чтобы генерировать более близкий к y' выходной сигнал, как показано на рис. 4.2. Посмотрим на пример из сферы распознавания образов. Представьте, что мы обучаем модель предсказывать названия фруктов (выходные сигналы) по их изображениям (входные сигналы). Например, на одном из обучающих изображений показан банан, но модель ошибочно говорит, что это *яблоко*. Поскольку желаемый выходной сигнал отличается от прогноза модели, мы меняем ее, чтобы в следующий раз она правильно выдавала предсказание *банан*.

Демонстрация модели желаемых выходных сигналов для различных входных сигналов и ее подгонка и означают обучение модели на *обучающих данных*. Со временем модель усваивает, какие выходные сигналы вы хотели бы видеть для конкретных входных сигналов. Именно поэтому в XXI столетии

так важны данные: модель хороша настолько, насколько хороши ее обучающие данные. Без хороших обучающих данных модель заведомо не покажет хороших результатов. Проще говоря, обучающие данные играют роль «учителя», направляющего процесс машинного обучения. Отсюда и название *машинное обучение с учителем*.

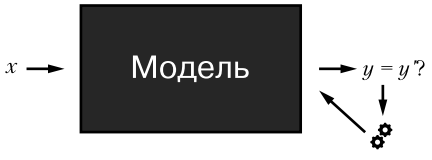


Рис. 4.2. Этап обучения модели

Этап выполнения вывода

На *этапе выполнения вывода* обученная модель используется с целью предсказания выходных значений для новых входных признаков x. Обратите внимание, что модель способна предсказывать выходные сигналы для никогда не встречавшихся в обучающих данных входных сигналов. Например, модель предсказания фруктов с *этапа обучения* теперь может определять их названия (усвоенные из обучающих данных) на изображениях, никогда ранее ей не встречавшихся. Другими словами, адекватные модели машинного обучения обладают способностью к *обобщению* (generalization), то есть умеют использовать накопленный на обучающих данных опыт, чтобы предсказывать результаты для новых входных сигналов. Проще говоря, хорошо обобщающие модели генерируют безошибочные предсказания для новых входных данных. Полученные в результате обобщения предсказания для входных данных, не встречавшихся модели, — одна из сильных сторон машинного обучения, а также основная причина его популярности среди широкого спектра приложений.

Линейная регрессия

Линейная регрессия — наиболее часто встречающийся в начальных руководствах по машинному обучению алгоритм. Он часто применяется для решения *задач регрессии*, в которых модель предсказывает отсутствующие значения данных на основе имеющихся. Немаловажное преимущество линейной регрессии как для преподавателей, так и и для тех, кто ею пользуется, — ее

простота. Что, впрочем, отнюдь не означает, что с ее помощью нельзя решать реальные задачи! У линейной регрессии множество сценариев применения на практике в самых разнообразных сферах, например исследованиях рынка, астрономии и биологии. Из этого раздела вы узнаете все, что нужно для начала работы с линейной регрессией.

Общее описание

Как с помощью линейной регрессии предсказать курс акций в заданный день? Прежде чем ответить на этот вопрос, приведем несколько определений.

Любая модель машинного обучения состоит из параметров модели. *Параметры модели* — внутренние переменные ее конфигурации, оцениваемые на основе имеющихся данных. Эти параметры модели определяют вычисленнные моделью предсказания по заданным входным признакам. В случае линейной регрессии параметры модели называются *коэффициентами*. Наверное, вы помните из школьного курса формулу двумерной прямой: $f(x) = ax + c$. Переменные a и c — коэффициенты линейного уравнения $ax + c$. Оно описывает преобразование каждого из входных сигналов x в выходной сигнал $f(x)$, так что все выходные сигналы вместе описывают прямую в двумерном пространстве. Меняя коэффициенты, можно описать любую прямую в двумерном пространстве.

Модель линейной регрессии сочетает заданные входные признаки $x_1, x_2, ..., x_k$ с коэффициентами $a_1, a_2, ..., a_k$ для вычисления предсказываемого выходного сигнала y по формуле:

$$y = f(x) = a_0 + a_1 \times x_1 + a_2 \times x_2 + ... + a_k \times x_k.$$

В нашем примере с курсами акций один входной признак: x — день. Мы подаем на вход день x в надежде получить курс акций, то есть выходной сигнал y. Что упрощает модель линейной регрессии до формулы двумерной прямой:

$$y = f(x) = a_0 + a_1 x.$$

Посмотрим на три прямые, отличающиеся только двумя параметрами модели a_0 и a_1, на рис. 4.3. Первая ось координат отражает входной сигнал x. Вторая — выходной сигнал y. Прямые отражают (линейную) связь между входным и выходным сигналами.

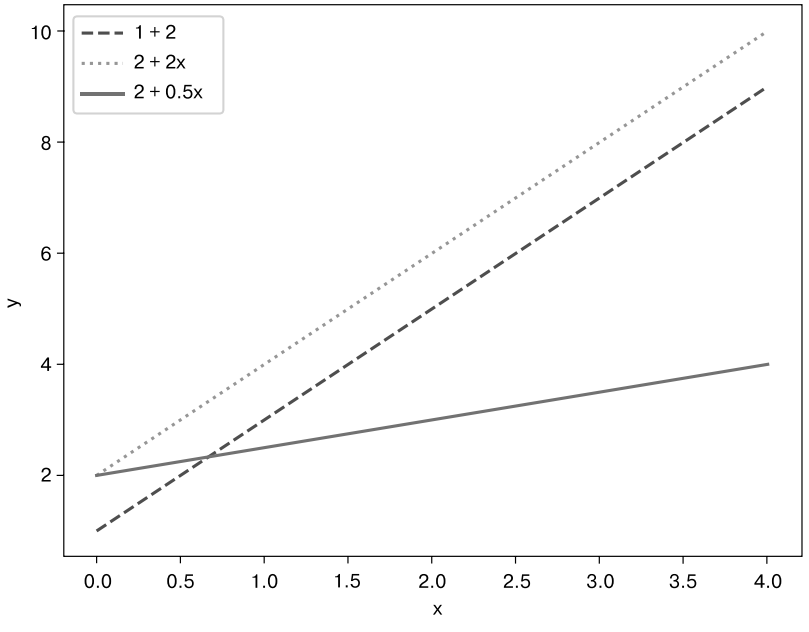


Рис. 4.3. Три модели (прямые) линейной регрессии, соответствующие различным параметрам (коэффициентам) модели. Каждая из этих прямых отражает взаимосвязь между входной и выходной величинами

Пусть в нашем примере с курсами акций обучающие данные представляют собой индексы для трех дней, `[0, 1, 2]`, соответствующие курсам акций `[155, 156, 157]`. Другими словами:

- входной сигнал `x = 0` должен приводить к выходному сигналу `y = 155`;
- входной сигнал `x = 1` должен приводить к выходному сигналу `y = 156`;
- входной сигнал `x = 2` должен приводить к выходному сигналу `y = 157`.

Какая же прямая хорошо соответствует нашим обучающим данным? Я отобразил обучающие данные на рис. 4.4.

Чтобы найти прямую, лучше всего описывающую наши данные, а значит, лучшую и для создания модели линейной регрессии, необходимо определить коэффициенты. Именно для этого и служит машинное обучение. Существует два основных способа определения параметров модели для линейной регрессии. Во-первых, можно аналитически вычислить лучше всего подходящую прямую между этими точками (стандартный подход для линейной регрессии).

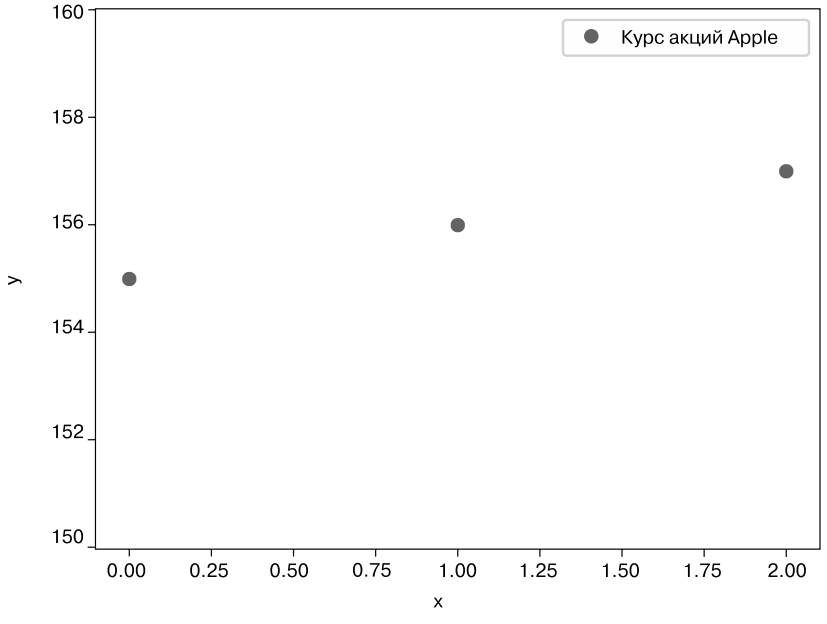


Рис. 4.4. Наши обучающие данные с индексом в массиве в качестве координаты x и ценой в качестве координаты y

Во-вторых, можно пробовать различные модели, проверяя каждую на маркированной выборке данных, и в конце концов найти лучшую. В любом случае «лучшая» модель выбирается путем *минимизации погрешности* (error minimization), при которой модель минимизирует квадрат разности (выбирает минимизирующие ее коэффициенты) предсказанных моделью значений и идеального выходного сигнала, выбирая модель с самой низкой погрешностью.

Для наших данных в итоге получаются коэффициенты $a_0 = 155.0$ и $a_1 = 1.0$. Подставляем их в нашу формулу линейной регрессии:

$$y = f(x) = a_0 + a_1 x = 155.0 + 1.0 \times x$$

и строим на одном графике прямую и добавляем обучающие данные, как показано на рис. 4.5.

Идеально! Квадрат расстояния между прямой (прогнозом модели) и обучающими данными равен нулю — мы нашли модель, которая минимизирует погрешность. С помощью этой модели теперь можно предсказывать курсы

акций для любых значений x. Например, пусть мы хотим предсказать курс акций в день $x = 4$. Для этого мы просто можем воспользоваться моделью и вычислить $f(x) = 155.0 + 1.0 \times 4 = 159.0$. Итак, предсказанный курс акций на четвертый день равен 159 долларам. Конечно, совсем другое дело, насколько хорошо это предсказание отражает реальность.

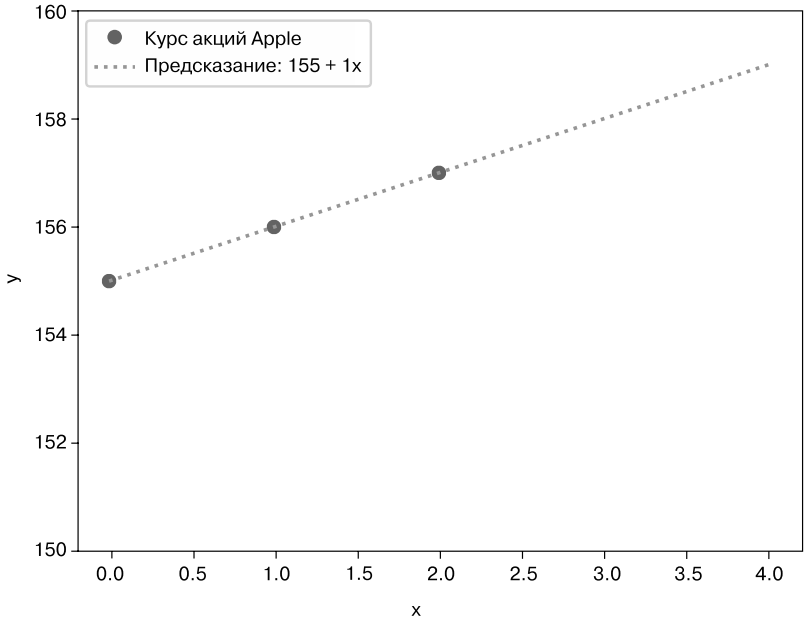


Рис. 4.5. Прямая предсказания на основе нашей модели линейной регрессии

Такова общая картина происходящего. Теперь посмотрим на то, как выглядит соответствующий код.

Код

В листинге 4.1 показано, как создать простую модель линейной регрессии с помощью одной строки кода (возможно, вам нужно будет сначала установить библиотеку scikit-learn путем выполнения в командной оболочке команды `pip install sklearn`).

Листинг 4.1. Простая модель линейной регрессии

```
from sklearn.linear_model import LinearRegression
import numpy as np
```

```
## Данные (курс акций Apple)
apple = np.array([155, 156, 157])
n = len(apple)

## Однострочник
model = LinearRegression().fit(np.arange(n).reshape((n,1)), apple)

## Результат
print(model.predict([[3],[4]]))
```

Можете ли вы уже угадать, какие результаты вернет этот фрагмент кода?

Принцип работы

В этом однострочнике используется две библиотеки Python: NumPy и scikit-learn. Первая из них — фактически стандартная библиотека для численных вычислений (например, операций с матрицами). Вторая — самая обширная библиотека для машинного обучения, включающая реализации сотен алгоритмов и методик машинного обучения.

Возможно, вы спросите: «Почему вы используете библиотеки в однострочнике Python? Не жульничество ли это?» Хороший вопрос, ответ на который — да. Любая программа на языке Python — с библиотеками или без — использует высокоуровневую функциональность, в основе которой лежат низкоуровневые операции. Нет смысла изобретать колесо, когда можно повторно задействовать уже существующую базу кода (то есть встать на плечи гигантов). Начинающие разработчики часто стремятся реализовать все самостоятельно, что снижает их производительность. В этой книге мы хотели бы не игнорировать широкий спектр функциональности, реализованной лучшими разработчиками и первопроходцами Python, а использовать во всей полноте. Разработка, оптимизация и шлифовка каждой из этих библиотек заняла у высококвалифицированных разработчиков многие годы.

Рассмотрим листинг 4.1 шаг за шагом. Во-первых, мы создали простой набор данных из трех значений и сохранили его длину в отдельной переменной `n` ради сокращения кода. Наши данные состоят из трех курсов акций Apple за три последовательных дня. Этот набор данных хранится в переменной `apple` в виде одномерного массива NumPy.

Во-вторых, мы создали модель с помощью вызова `LinearRegression()`. Но какие значения параметров будут у этой модели? Чтобы их найти, мы обучаем

модель с помощью вызова функции `fit()`. Она принимает два аргумента: входные признаки обучающих данных и желаемые выходные сигналы для этих входных сигналов. Роль желаемых выходных сигналов играют настоящие курсы акций Apple. Входные же признаки необходимо передать `fit()` в виде массива в следующем формате:

```
[<обучающие_данные_1>,
<обучающие_данные_2>,
--сокращено--
<обучающие_данные_n>]
```

где каждое значение обучающих данных представляет собой последовательность значений признаков:

```
<обучающие_данные> = [признак_1, признак_2, ..., признак_k]
```

В нашем случае входной сигнал состоит лишь из одного признака x (текущий день). Более того, предсказание также состоит лишь из одного значения y (текущий курс акций). Чтобы изменить форму входного массива на нужную, необходимо привести его к виду следующей матрицы, выглядящей странновато:

```
[[0],
 [1],
 [2]]
```

Матрица из одного столбца называется *вектором-столбцом*. Для создания последовательности возрастающих значений x мы применим метод `np.arange()`, а затем воспользуемся `reshape((n, 1))` для преобразования одномерного массива NumPy в двумерный, содержащий один столбец и `n` строк (см. главу 3). Обратите внимание, что scikit-learn допускает одномерный массив в качестве выходного сигнала (иначе нам бы пришлось изменить и форму массива данных `apple`).

Получив обучающие данные и желаемые выходные сигналы, функция `fit()` производит минимизацию погрешности: находит такие параметры модели (то есть *прямую*), что разность между предсказанными моделью значениями и желаемыми выходными сигналами минимальна.

Когда функция `fit()` сочтет, что модель доведена до ума, она возвращает модель, пригодную для предсказания двух новых значений курсов акций

с помощью функции `predict()`. Требования к входным данным у функции `predict()` те же, что и у `fit()`, поэтому для их удовлетворения мы передаем матрицу из одного столбца с нашими двумя новыми значениями, для которых требуются предсказания:

```
print(model.predict([[3],[4]]))
```

А поскольку минимизированная погрешность равна нулю, то должны получиться идеально линейные результаты: 158 и 159, которые прекрасно соответствуют прямой, построенной выше на рис. 4.5. Но зачастую найти столь прекрасно подходящую линейную модель не получается. Например, если запустить ту же функцию для курсов акций `[157, 156, 159]` и построить соответствующий график, то получится прямая, изображенная на рис. 4.6.

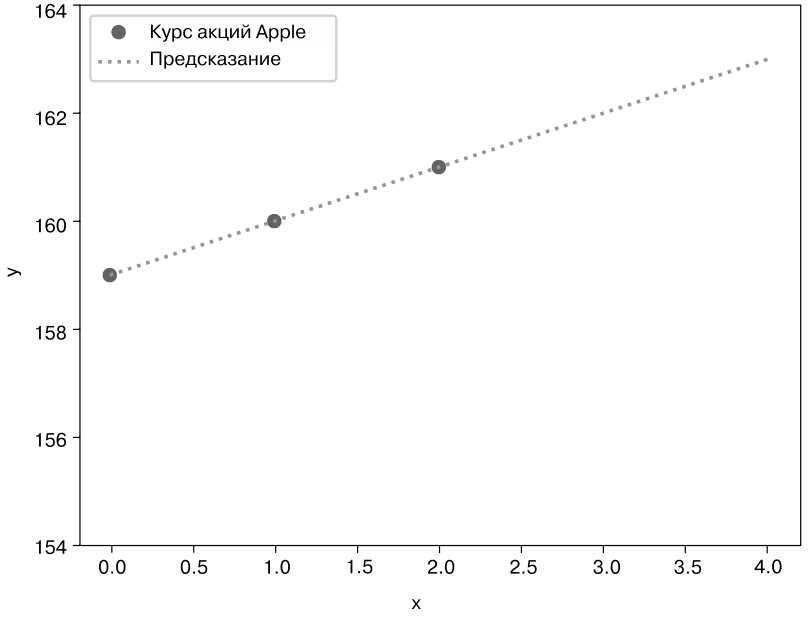


Рис. 4.6. Неидеально подогнанная модель линейной регрессии

В этом случае функция `fit()` находит прямую, минимизирующую квадрат погрешности между обучающими данными и предсказаниями, как и упоминалось ранее.

Резюмируем: линейная регрессия — методика машинного обучения, при которой модель усваивает коэффициенты как параметры модели. Полученная в итоге линейная модель (например, прямая в двумерном пространстве) может непосредственно выполнять предсказания на основе новых входных данных. Задача предсказания числовых значений по заданным числовым входным значениям относится к классу задач регрессии. Из следующего раздела вы узнаете еще об одной важной сфере машинного обучения — классификации.

Логистическая регрессия в одной строке

Логистическая регрессия часто применяется для *задач классификации*, в которых предсказывается, относится ли конкретный пример данных к конкретной категории (классу). Эта постановка отличается от задач регрессии, в которых по заданному примеру данных необходимо предсказать числовое значение, относящееся к непрерывному диапазону. Пример задачи классификации: разбиение пользователей Twitter на мужчин и женщин по заданным различным входным признакам, например *частоте отправки ими твитов* или *количеству ответов на твиты*. Модель логистической регрессии относится к наиболее основополагающим моделям машинного обучения. Многие из понятий, с которыми вы познакомитесь в этом разделе, станут основой для более продвинутых методик машинного обучения.

Общее описание

Чтобы познакомиться с логистической регрессией, кратко рассмотрим, как работает линейная регрессия: по входным данным вычисляется прямая, лучше всего подходящая для этих обучающих данных, и предсказывается выходной сигнал для входного сигнала x. В целом линейная регрессия прекрасно подходит для предсказания *непрерывного* выходного сигнала, величина которого может принимать бесконечное количество значений. Предсказанные ранее курсы акций, например, могут теоретически принимать любые положительные значения.

Но что, если выходной сигнал не непрерывный, а *категориальный*, то есть принадлежит к ограниченному количеству групп или категорий? Например, пусть мы хотим предсказать правдоподобие рака легких, исходя из количества выкуренных пациентом сигарет. У каждого пациента либо есть рак

легких, либо нет. В отличие от курсов акций, возможных исходов только два. Предсказание правдоподобия категориальных исходов — основная причина использования логистической регрессии.

Сигма-функция

Если линейная регрессия подгоняет к обучающим данным прямую, то логистическая регрессия подгоняет к ним S-образную кривую — так называемую *сигма-функцию* (the sigmoid function). S-образная кривая упрощает выбор из двух альтернатив (например, да/нет). Для большинства входных сигналов сигма-функция возвращает значение, очень близкое либо к 0 (одна категория), либо к 1 (другая категория). Неоднозначный результат относительно маловероятен. Отметим, что для конкретных входных значений могут быть сгенерированы и равные 0.5 вероятности, но форма кривой специально выбрана таким образом, чтобы минимизировать возможность этого на практике (для большинства значений горизонтальной оси координат величина вероятности очень близка либо к нулю, либо к единице). На рис. 4.7 приведена кривая логистической регрессии для прогноза рака легких.

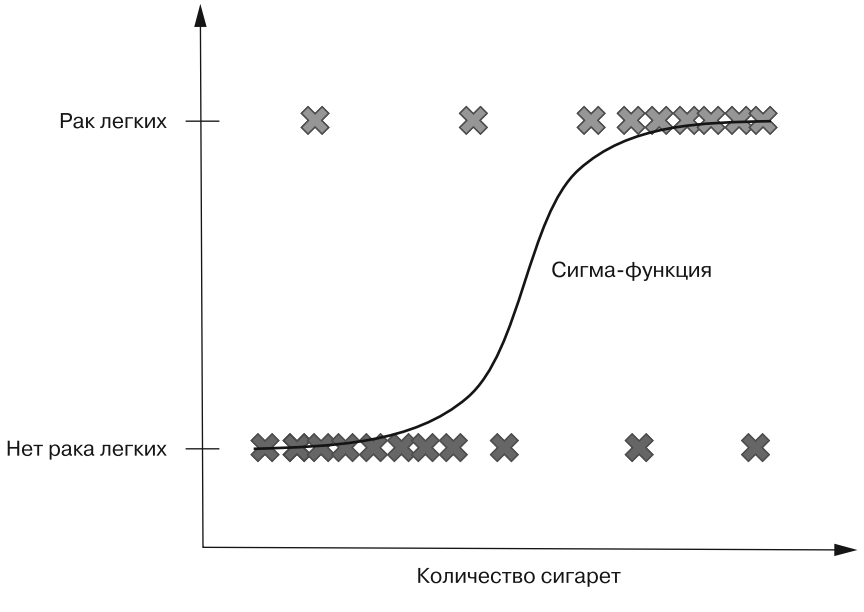


Рис. 4.7. Кривая логистической регрессии для предсказания рака по количеству выкуриваемых сигарет

ПРИМЕЧАНИЕ

Логистическую регрессию можно применять и для полиномиальной классификации (multinomial classification), при которой данные классифицируются более чем по двум классам. Для этого используется обобщение сигма-функции — так называемая многомерная логистическая функция (softmax function), возвращающая кортеж вероятностей, по одной для каждого класса. Сигма-функция же преобразует входной (-ые) признак (-и) в одно значение вероятности. Впрочем, ради простоты и удобочитаемости здесь я сосредоточу свое внимание на биномиальной классификации (binomial classification).

Сигма-функция на рис. 4.7 аппроксимирует вероятность наличия у пациента рака легких в зависимости от количества выкуриваемых им сигарет. Исходя из этой вероятности, можно четко определить при наличии одной только информации о количестве выкуриваемых пациентом сигарет, есть ли у пациента рак легких.

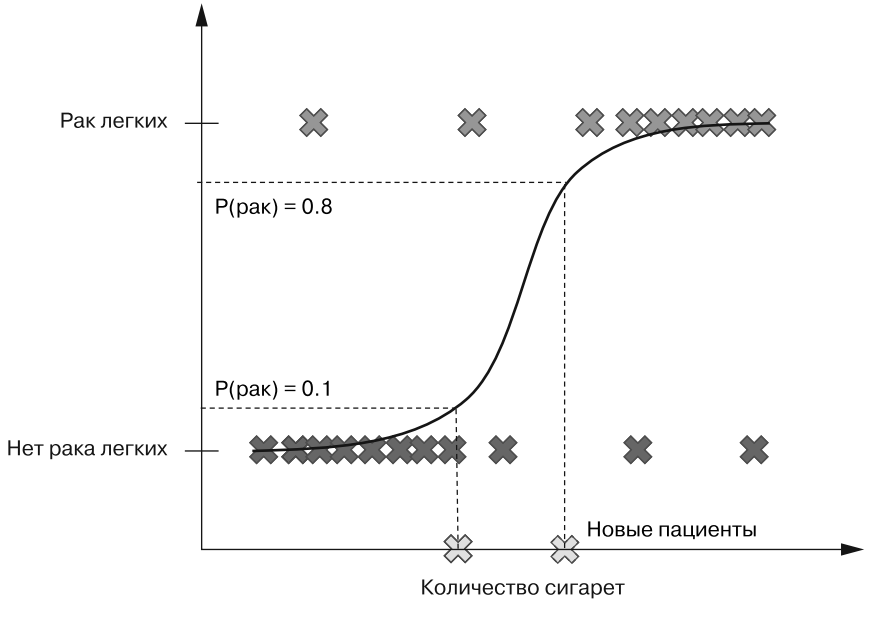


Рис. 4.8. Оценка вероятностей исходов с помощью логистической регрессии

Взгляните на предсказание на рис. 4.8, на котором представлены два новых пациента (изображены светло-серым цветом внизу графика). Нам ничего о них не известно, кроме количества выкуриваемых ими сигарет. Мы обучаем модель логистической регрессии (сигма-функцию), возвращающую вероятность для любого нового входного значения x. Если возвращающая сигма-функцией вероятность выше 50 %, то модель выдает предсказание *онкопозитивный*, в противном случае — *онконегативный*.

Поиск модели максимального правдоподобия

Основной вопрос метода логистической регрессии — как выбрать правильную сигма-функцию, лучше всего соответствующую обучающим данным. Для ответа на него используется такое понятие, как *правдоподобие* (likelihood) модели: возвращаемая моделью вероятность для наблюдаемых обучающих данных. Желательно выбрать модель с максимальным правдоподобием. Идея в том, чтобы эта модель лучше всего аппроксимировала реальный процесс, в результате которого были сгенерированы обучающие данные.

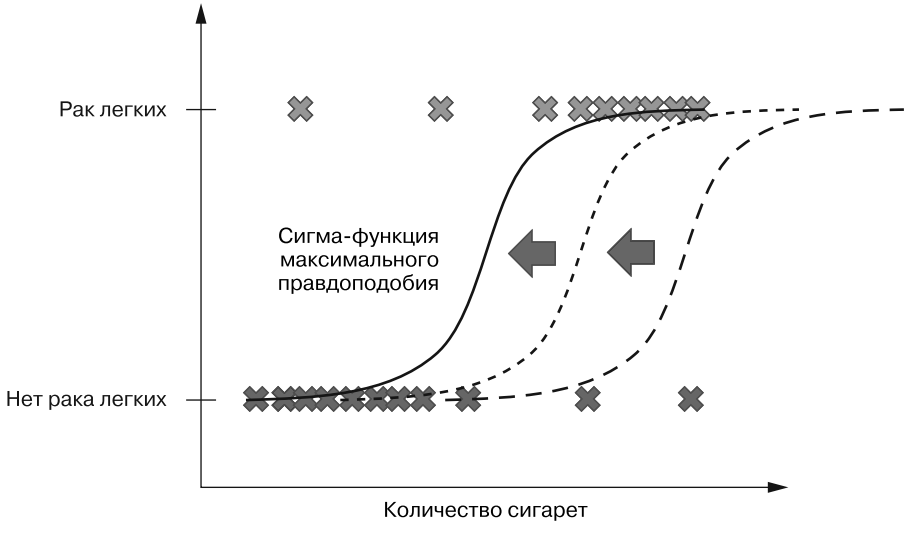


Рис. 4.9. Поиск максимального правдоподобия путем проверки различных сигма-функций

В целях вычисления правдоподобия заданной модели для заданного набора обучающих данных вычисляется правдоподобие для каждой из обучающих точек данных, и в результате их перемножения получается общее правдоподобие для всего набора обучающих данных. Как же вычислить правдоподобие отдельной обучающей точки данных? Достаточно просто применить к ней сигма-функцию модели, чтобы получить вероятность для указанной точки данных при использовании этой модели. Чтобы выбрать модель максимального правдоподобия для всех точек данных, необходимо повторить это вычисление правдоподобия для различных сигма-функций (выбираемых с небольшим сдвигом), как показано на рис. 4.9.

В предыдущем абзаце я описал, как определить сигма-функцию (модель) максимального правдоподобия, которая лучше всего описывает данные, поэтому можно ее использовать для предсказания новых точек данных.

С теоретической частью покончено. Теперь посмотрим, как можно реализовать логистическую регрессию в виде однострочника Python.

Код

Мы продемонстрировали пример использования логистической регрессии в сфере здравоохранения (корреляция потребления сигарет с вероятностью рака). Подобное приложение — «виртуальный доктор» — прекрасная идея для мобильного приложения, правда? Создадим ваше первое приложение — «виртуальный доктор» на основе логистической регрессии, как показано в листинге 4.2, с помощью одной строки кода Python!

Листинг 4.2. Модель логистической регрессии

```
from sklearn.linear_model import LogisticRegression
import numpy as np

## Данные (#сигареты, рак)
n = 4
X = np.array([[0, "No"],
              [10, "No"],
              [60, "Yes"],
              [90, "Yes"]])

## Однострочник
model = LogisticRegression().fit(X[:,0].reshape(n,1), X[:,1])

## Результат
print(model.predict([[2],[12],[13],[40],[90]]))
```

Попробуйте догадаться, каковы будут результаты выполнения этого фрагмента кода.

Принцип работы

Обучающие данные `X` состоят из четырех медицинских карт (строк), включающих два столбца. В первом столбце содержится количество выкуриваемых пациентами сигарет *(входной признак)*, а во втором — *метки классов* (class labels), отражающие, был ли у них в итоге диагностирован рак легких.

Мы создали модель с помощью вызова конструктора `LogisticRegression()`, а затем вызвали для этой модели функцию `fit()`; она принимает два аргумента — входной сигнал (количество выкуриваемых сигарет) и выходные метки классов (рак). Функция `fit()` ожидает на входе двумерный массив данных, по одной строке на каждый обучающий пример данных и по одному столбцу для каждого признака этого обучающего примера данных. В данном случае у нас только одно значение признака, так что мы преобразуем одномерный входной сигнал в двумерный массив NumPy с помощью операции `reshape()`. Первый аргумент функции `reshape()` задает количество строк, а второй — количество столбцов. Нам важно только количество столбцов, равное `1`. В качестве желаемого количества строк мы передаем `-1`, указывая тем самым NumPy определить количество строк автоматически.

Входные обучающие данные после изменения формы будут выглядеть следующим образом (фактически мы просто удалили метки классов, сохранив форму двумерного массива неизменной):

```
[[0],
 [10],
 [60],
 [90]]
```

Далее мы предсказываем, есть ли у пациента рак легких по выкуриваемому количеству сигарет: входные данные включают значения 2, 12, 13, 40 и 90 сигарет. На выходе получаем следующее:

```
# ['No' 'No' 'No' 'Yes' 'Yes']
```

Модель предсказывает, что первые три пациента — онконегативны, а последние два — онкопозитивны.

Посмотрим подробнее на возвращаемые сигма-функцией вероятности, которые привели к такому предсказанию! Просто выполните следующий фрагмент кода после листинга 4.2:

```
for i in range(20):
    print("x=" + str(i) + " --> " + str(model.predict_proba([[i]])))
```

Функция `predict_proba()` принимает на входе количество сигарет и возвращает массив с вероятностями онконегативности (индекс 0) и вероятностями онкопозитивности (индекс 1). При выполнении этого кода вы, вероятно, получите следующее:

```
x=0 --> [[0.67240789 0.32759211]]
x=1 --> [[0.65961501 0.34038499]]
x=2 --> [[0.64658514 0.35341486]]
x=3 --> [[0.63333374 0.36666626]]
x=4 --> [[0.61987758 0.38012242]]
x=5 --> [[0.60623463 0.39376537]]
x=6 --> [[0.59242397 0.40757603]]
x=7 --> [[0.57846573 0.42153427]]
x=8 --> [[0.56438097 0.43561903]]
x=9 --> [[0.55019154 0.44980846]]
x=10 --> [[0.53591997 0.46408003]]
x=11 --> [[0.52158933 0.47841067]]
x=12 --> [[0.50722306 0.49277694]]
x=13 --> [[0.49284485 0.50715515]]
x=14 --> [[0.47847846 0.52152154]]
x=15 --> [[0.46414759 0.53585241]]
x=16 --> [[0.44987569 0.55012431]]
x=17 --> [[0.43568582 0.56431418]]
x=18 --> [[0.42160051 0.57839949]]
x=19 --> [[0.40764163 0.59235837]]
```

Если вероятность онконегативности выше вероятности онкопозитивности, то будет возвращено предсказание *онконегативный*. Последний раз такое встречается при `x=12`. Если пациент выкуривает более 12 сигарет, то алгоритм классифицирует его как *онкопозитивного*.

Резюмируя: вы научились легко решать задачи классификации с помощью логистической регрессии с использованием библиотеки scikit-learn. Основная идея логистической регрессии заключается в подгонке к данным S-образной кривой (сигма-функции). Эта функция ставит каждой точке данных и каждому возможному классу в соответствие числовое значение от 0 до 1, которое моделирует вероятность принадлежности этой точки данных заданному классу. Однако на практике часто попадаются обучающие данные

без меток классов. Например, у вас могут быть данные о покупателях (допустим, их возраст и уровень дохода), но никаких меток классов для точек данных. Чтобы извлечь полезную информацию из подобных данных, понадобится еще одна разновидность машинного обучения: машинное обучение без учителя. А точнее, мы научимся искать кластеры схожих точек данных — это важное подмножество машинного обучения без учителя.

Кластеризация методом k-средних в одной строке кода

Если и есть алгоритм кластеризации, который пригодится и обычному специалисту в области компьютерных наук, и исследователю данных, и специалисту по машинному обучению, то это *алгоритм кластеризации методом k-средних* (k-means algorithm). В текущем разделе мы обсудим общую идею, а также рассмотрим, когда и как использовать его с помощью всего одной строки кода на Python.

Общее описание

В предыдущем примере мы рассматривали обучение с учителем, при котором обучающие данные *маркированы*, то есть известны выходные значения для всех входных признаков в обучающих данных. Но на практике так бывает далеко не всегда. Зачастую исследователи сталкиваются с *немаркированными* данными, особенно в приложениях аналитической обработки данных, когда непонятно, какой выходной сигнал будет «оптимальным». В подобном случае предсказать что-либо невозможно (поскольку отсутствует эталонный выходной сигнал), но все равно можно извлечь из этих немаркированных наборов данных немало полезной информации (например, найти кластеры схожих немаркированных данных). Модели, работающие с немаркированными данными, относятся к категории моделей *машинного обучения без учителя* (unsupervised learning).

В качестве примера представьте, что работаете над стартапом, обслуживающим различную целевую аудиторию, разного возраста и с разным доходом. Ваш начальник просит найти определенное количество персон, лучше всего соответствующих вашей целевой аудитории. Для выявления *усредненных персон заказчиков* в вашей компании можно воспользоваться методами кластеризации. На рис. 4.10 приведен пример.

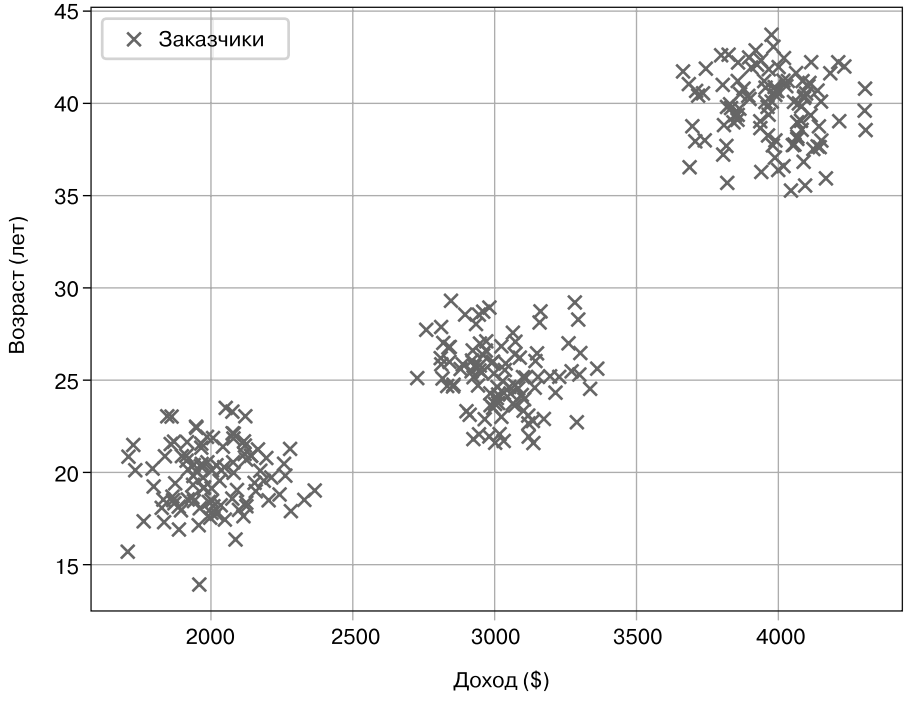


Рис. 4.10. Наблюдаемые данные о заказчиках в двумерном пространстве

На данном рисунке можно с легкостью выделить три типа персон различного уровня доходов и возраста. Но как сделать это алгоритмически? Тут-то и вступают в дело алгоритмы кластеризации наподобие очень популярного алгоритма кластеризации методом *k*-средних. При заданном наборе данных и целом числе *k* алгоритм кластеризации методом *k*-средних находит *k* кластеров данных, таких что расстояние между центром кластера (так называемым *центроидом*) и данными в этом кластере минимально. Другими словами, путем выполнения алгоритма кластеризации методом *k*-средних можно найти различные персоны в ваших наборах данных, как показано на рис. 4.11.

Центры кластеров (черные точки) отмечают кластеризованные данные о заказчиках. Каждый центр кластера можно считать одной персоной заказчика. Таким образом, у нас есть три персоны: 20-летний заказчик с доходом в 2000 долларов, 25-летний с доходом в 3000 долларов и 40-летний с доходом в 4000 долларов. Замечательно то, что алгоритм кластеризации методом *k*-средних находит эти центры кластеров даже в многомерном

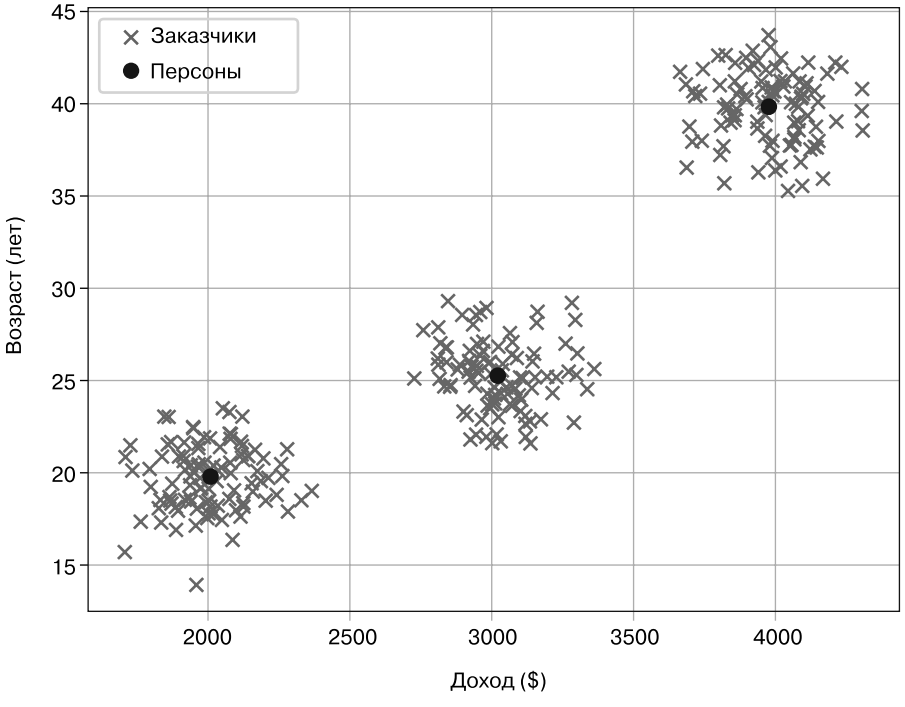


Рис. 4.11. Данные о заказчиках с персонами заказчиков (центроидами кластеров) в двумерном пространстве

пространстве (в котором найти их визуально для человека было бы непросто).

Алгоритм кластеризации методом *k*-средних требует на входе «количество центров кластеров *k*». В данном случае мы смотрим на данные и каким-то чудесным образом выбираем $k = 3$. Более продвинутые алгоритмы могут находить количество центров кластеров автоматически (например, взгляните на статью *Learning the k in K-Means* 2004 года, написанную Грегом Хамерли и Чарльзом Илканом).

Как же работает алгоритм кластеризации методом *k*-средних? По существу, он сводится к следующей процедуре:

```
Задать случайные начальные значения для центров кластеров (центроидов)
Повторять до достижения сходимости
    Распределить все точки данных по ближайшим к ним центрам кластеров
Повторить вычисление всех центров кластеров, приписанных к ним всех точек данных
    как центроидов
```

Все это приводит к многочисленным итерациям цикла: сначала данные приписываются к k центрам кластеров, а затем каждый центр кластера пересчитывается как центроид приписанных к нему данных.

Реализуем его!

Рассмотрим следующую задачу: найти в данном наборе двумерных данных о зарплатах (*отработанные часы*, *заработанные деньги*) два кластера сотрудников, работающих одинаковое количество часов и зарабатывающих примерно одинаковые деньги.

Код

Как реализовать все это в одной строке кода? К счастью, библиотека scikit-learn Python включает готовую эффективную реализацию алгоритма кластеризации методом k-средних. Листинг 4.3 демонстрирует фрагмент кода с однострочником, выполняющим кластеризацию методом k-средних.

Листинг 4.3. Кластеризация методом k-средних в одной строке

```
## Зависимости
from sklearn.cluster import KMeans
import numpy as np

## Данные (Отработано (ч) / Зарплата ($))
X = np.array([[35, 7000], [45, 6900], [70, 7100],
              [20, 2000], [25, 2200], [15, 1800]])

## Однострочник
kmeans = KMeans(n_clusters=2).fit(X)

## Результат
cc = kmeans.cluster_centers_
print(cc)
```

Каковы же будут результаты выполнения этого фрагмента кода? Попробуйте догадаться, даже если не понимаете некоторых нюансов синтаксиса. Это поможет вам осознать пробелы в своих знаниях и намного лучше подготовиться к восприятию алгоритма.

Принцип работы

В первых строках мы импортируем модуль `KMeans` из пакета `sklearn.cluster`. Этот модуль отвечает за саму кластеризацию. Необходимо также

импортировать библиотеку NumPy, поскольку модуль `KMeans` использует в своей работе ее массивы.

Наши данные — двумерные, они соотносят количество отработанных часов с зарплатой некоторых работников. На рис. 4.12 показаны шесть точек данных из этого набора данных.

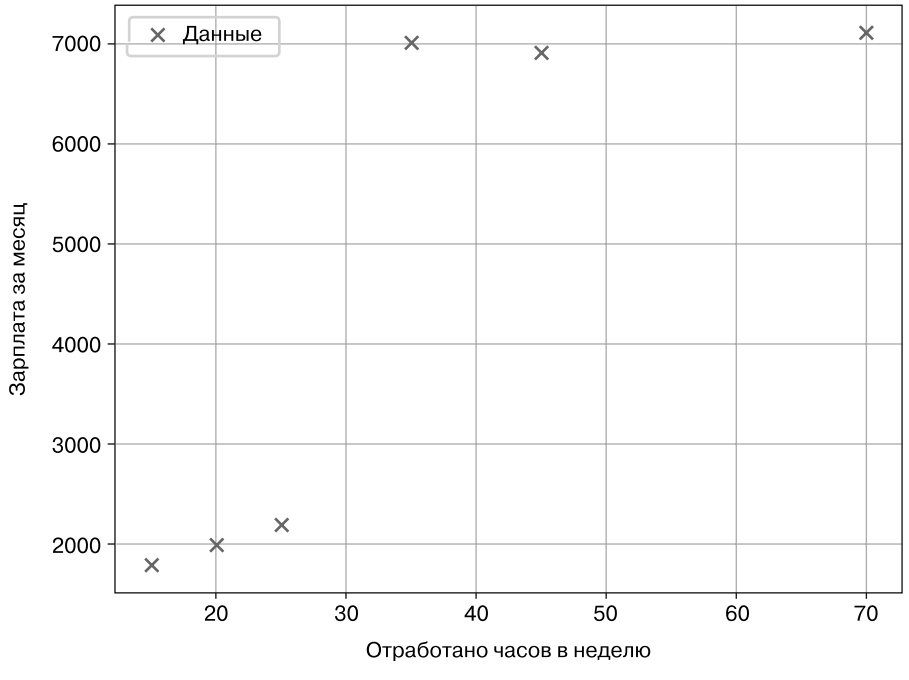


Рис. 4.12. Данные по зарплатам сотрудников

Задача — найти два центра кластеров, лучше всего удовлетворяющих этим данным:

```
## Однострочник
kmeans = KMeans(n_clusters=2).fit(X)
```

В этом однострочнике создается новый объект `KMeans`, который отвечает за выполнение алгоритма. При создании объекта `KMeans` описывается количество центров кластеров с помощью аргумента функции `n_clusters`. А затем мы просто вызываем метод экземпляра `fit(X)` для выполнения алгоритма кластеризации методом k-средних на входных данных `X`. Теперь

все результаты содержатся в объекте `KMeans`. Осталось только извлечь эти результаты из его атрибутов:

```
cc = kmeans.cluster_centers_
print(cc)
```

Обратите внимание, что по соглашению в пакете `sklearn` в некоторых названиях атрибутов в конце указывается подчеркивание (например, `cluster_centers_`), означающее, что эти атрибуты были созданы динамически на этапе обучения (в ходе вызова функции `fit()`). До этого их не существовало. Для Python такое соглашение не является стандартным (к подчеркиванию в конце названий прибегают обычно лишь во избежание конфликтов названий с ключевыми словами Python — например, переменная `list_` вместо `list`). Однако когда вы привыкнете, то почувствуете удобство согласованного использования атрибутов в пакете `sklearn`. Какие же будут центры кластеров и вообще результат работы данного фрагмента кода? Взгляните на рис. 4.13.

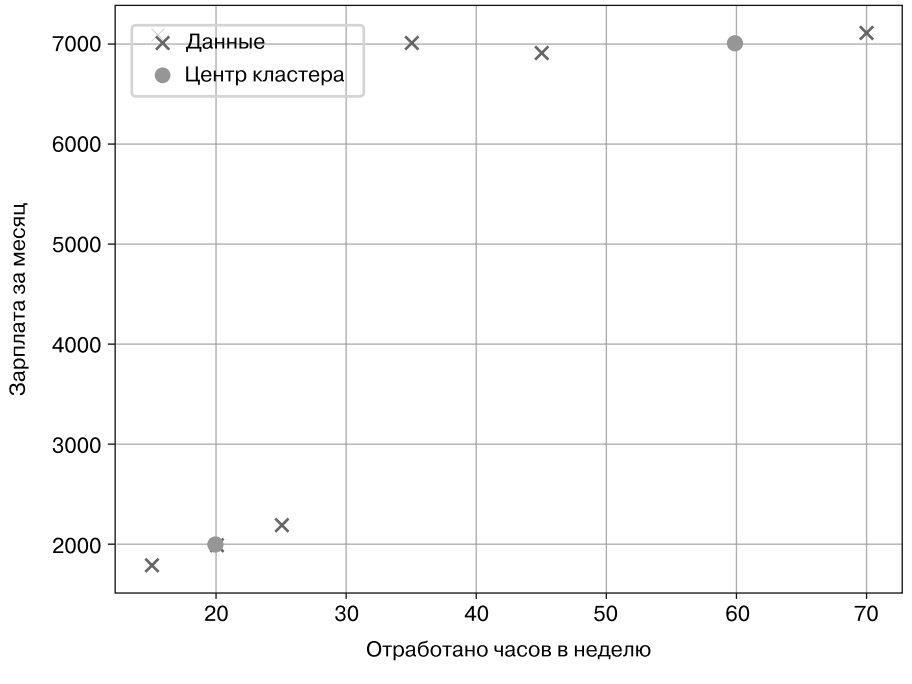


Рис. 4.13. Данные по зарплатам сотрудников с центрами кластеров в двумерном пространстве

На рисунке видны два центра кластеров: (20, 2000) и (50, 7000). Это также результат нашего однострочника Python. Эти кластеры соответствуют двум персонам сотрудников: первый работает 20 часов в неделю и зарабатывает 2000 долларов в месяц, а второй работает 50 часов в неделю и зарабатывает 7000 долларов в месяц. Эти два типа персон неплохо удовлетворяют нашим данным. Следовательно, результат выполнения нашего однострочного фрагмента кода выглядит так:

```
## Результат
cc = kmeans.cluster_centers_
print(cc)
'''
[[  50. 7000.]
 [  20. 2000.]]
'''
```

Резюмируя: в этом разделе вы познакомились с важным подвидом машинного обучения без учителя: кластеризацией. Алгоритм кластеризации методом *k*-средних — простой, эффективный и популярный способ выделения *k* кластеров из многомерных данных. «Под капотом» алгоритм в цикле пересчитывает центры кластеров и перераспределяет все точки данных по ближайшим к ним центрам кластеров, пока не будут найдены оптимальные кластеры. Однако кластеры не всегда идеально подходят для поиска схожих элементов данных. Многие наборы данных не проявляют кластерной организации, но информацию о расстоянии все равно хотелось бы использовать для машинного обучения и предсказания. Не будем покидать многомерное пространство и рассмотрим еще один способ задействовать (евклидово) расстояние между значениями данных: алгоритм *k*-ближайших соседей.

Метод k-ближайших соседей в одной строке кода

Популярный алгоритм *k-ближайших соседей* (K-Nearest Neighbors, KNN) используется для регрессии и классификации во многих приложениях, в частности в рекомендательных системах, а также при классификации изображений и финансовом прогнозе. На нем основаны многие продвинутые алгоритмы машинного обучения (например, предназначенные для информационного поиска). Вне всякого сомнения, понимание KNN — важный элемент качественного обучения в сфере computer science.

Общее описание

Алгоритм KNN — надежный, простой и популярный метод машинного обучения. Несмотря на простоту реализации, это конкурентоспособная и быстрая методика машинного обучения. Во всех прочих моделях машинного обучения, обсуждавшихся выше, обучающие данные использовались для вычисления *представления* исходных данных, на основе которого затем можно будет предсказывать, классифицировать или кластеризовать новые данные. Например, в алгоритмах линейной и логистической регрессии описываются параметры обучения, в то время как в алгоритме кластеризации вычисляются центры кластеров, исходя из обучающих данных. Алгоритм KNN отличается в этом отношении. В отличие от других подходов, в нем не вычисляется новая модель (или представление), а используется в качестве модели *весь набор данных целиком*.

Да, все правильно. Модель машинного обучения — всего лишь набор наблюдений. Каждый элемент обучающих данных — часть модели. У такого подхода есть свои преимущества и недостатки. Неудобен он тем, что размеры модели могут резко увеличиваться с ростом объема обучающих данных, а значит, может понадобиться предварительный этап обработки — фильтрации или выборки. Большое преимущество его, впрочем, в относительной простоте этапа обучения (достаточно просто добавить в модель новые значения данных). Кроме того, алгоритм KNN можно использовать как для предсказания, так и для классификации. При заданном входном векторе x алгоритм выглядит следующим образом.

1. Найти k ближайших соседей x (в соответствии с заранее выбранной метрикой расстояния).
2. Агрегировать k ближайших соседей в одно значение предсказания или классификации. При этом может использоваться любая агрегирующая функция, например взятие среднего, максимального или минимального значения.

Посмотрим для примера на компанию, торгующую недвижимостью. У нее есть большая база покупателей и цен на дома (рис. 4.14). В один прекрасный момент клиент спрашивает, сколько может стоить дом площадью 52 м^2. Вы запрашиваете свою модель KNN и немедленно получаете ответ: 33 167 долларов. И действительно, в ту же неделю клиент находит дом за 33 489 долларов. Как же системе KNN удалось произвести настолько безошибочное предсказание?

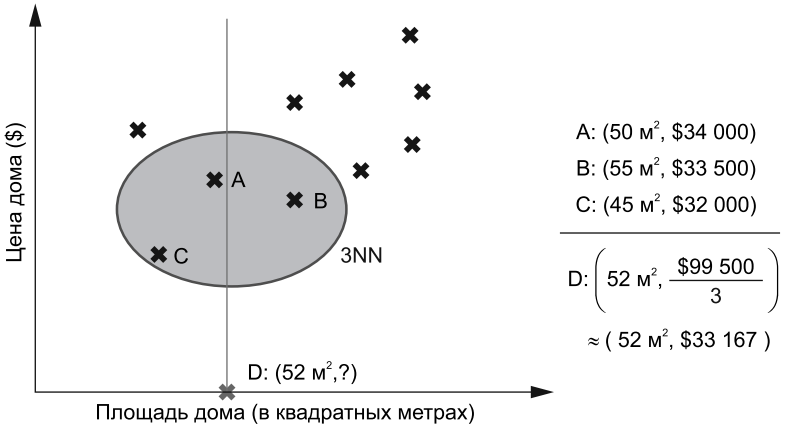


Рис. 4.14. Вычисление цены дома D на основе цен его трех ближайших соседей A, B и C

Прежде всего, система KNN просто вычисляет $k = 3$ ближайших соседей для запроса $D = 52$ м² (при использовании евклидового расстояния). Три ближайших соседа: A, B и C с ценами 34 000, 33 500 и 32 000 долларов соответственно. Далее эти три ближайших соседа агрегируются, а именно вычисляется их среднее арифметическое значение. А поскольку в этом примере $k = 3$, мы назовем нашу модель *3NN*. Конечно, можно использовать различные функции подобия, параметр k и метод агрегирования, получая в результате все более изощренные прогностические модели.

Еще одно преимущество KNN — легкость его адаптации к поступающим новым наблюдениям, что справедливо не для всех моделей машинного обучения. Из этого следует очевидный его недостаток — рост вычислительной сложности поиска k-ближайших соседей по мере добавления новых точек данных. Чтобы решить эту проблему, можно непрерывно исключать из модели устаревшие значения. Как я уже упоминал, с помощью KNN можно также решать задачи классификации. Вместо усреднения по k ближайшим соседям можно использовать механизм голосования: все ближайшие соседи «голосуют» за свои классы, и побеждает класс, за который «отдано больше всего голосов».

Код

Посмотрим, как можно использовать KNN на языке Python — в одной строке кода (листинг 4.4).

Листинг 4.4. Выполнение алгоритма KNN с помощью одной строки кода на Python

```
## Зависимости
from sklearn.neighbors import KNeighborsRegressor
import numpy as np

## Данные (площадь дома (в квадратных метрах) / цена дома ($))
X = np.array([[35, 30000], [45, 45000], [40, 50000],
              [35, 35000], [25, 32500], [40, 40000]])

## Однострочник
KNN = KNeighborsRegressor(n_neighbors=3).fit(X[:,0].reshape(-1,1), X[:,1])

## Результат
res = KNN.predict([[30]])
print(res)
```

Каковы же будут результаты выполнения этого фрагмента кода?

Принцип работы

Чтобы наглядно представить результат, построим график данных по ценам на жилье (рис. 4.15).

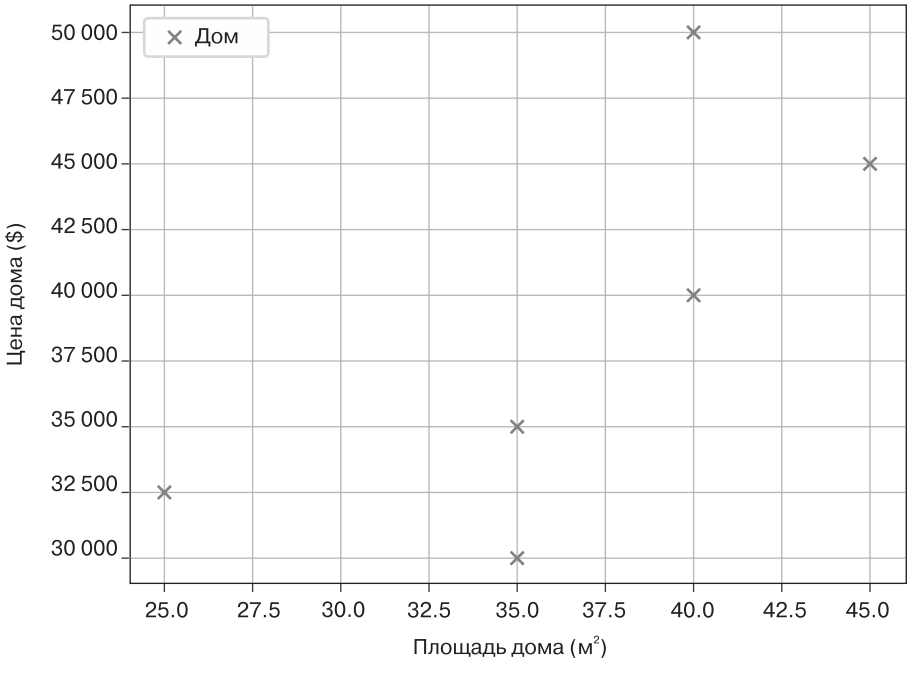


Рис. 4.15. Данные по ценам на жилье

Видите общую тенденцию? Можно предположить, что с ростом площади дома его рыночная стоимость также будет линейно расти. Вдвое больше квадратных метров — вдвое больше цена.

В коде (см. листинг 4.4) клиент запрашивает предсказание цены для дома площадью 30 квадратных метров. Что же предскажет KNN при $k = 3$ (то есть 3NN)? Взгляните на рис. 4.16.

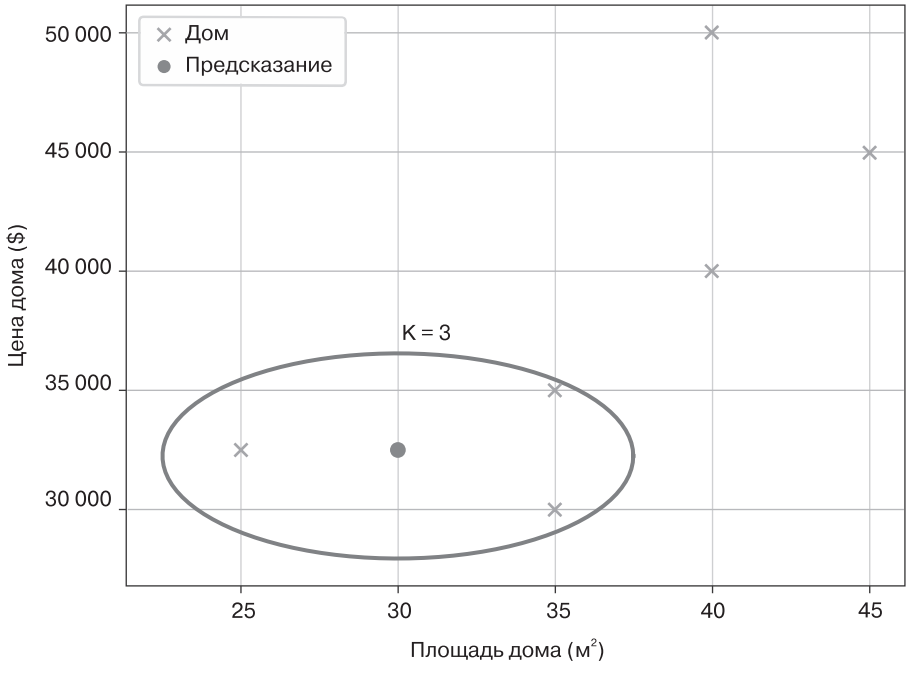


Рис. 4.16. Данные о стоимости домов в двумерном пространстве с предсказанием цены дома для новой точки данных (площадь дома равна 30 м^2) с помощью алгоритма KNN

Красота, правда? Алгоритм KNN находит три ближайших по площади дома и возвращает предсказанную стоимость дома как среднее $k = 3$ ближайших соседей. Таким образом, в результате получается 32 500 долларов.

Если вам не вполне понятны преобразования данных в этом однострочнике, то я вкратце поясню их:

```
KNN = KNeighborsRegressor(n_neighbors=3).fit(X[:,0].reshape(-1,1), X[:,1])
```

Прежде всего мы создаем новую модель машинного обучения — `KNeighborsRegressor`. Если нужно использовать KNN для классификации, то следует применить `KNeighborsClassifier`.

Далее мы обучаем модель с помощью функции `fit()` с двумя параметрами, первый из которых определяет входной сигнал (размер дома), а второй — выходной сигнал (стоимость дома). Оба параметра должны представлять собой массивоподобные структуры данных. Например, если нужно передать значение `30` в качестве входных данных, то следует передать его в виде `[30]`. Дело в том, что входные данные могут быть многомерными, а не только одномерными. Поэтому мы меняем форму входного сигнала:

```
print(X[:,0])
"[35 45 40 35 25 40]"

print(X[:,0].reshape(-1,1))
"""
[[35]
 [45]
 [40]
 [35]
 [25]
 [40]]
"""
```

Обратите внимание, что при использовании этого одномерного массива NumPy в качестве входных данных функции `fit()` она не будет работать, поскольку ожидает массив (массивоподобных) наблюдений, а не массив целых чисел.

Резюмируя: из этого однострочника вы узнали, как создать первый KNN-регрессор одной строкой кода. Если у вас много меняющихся данных и обновлений модели, то KNN — как раз для вас! А теперь перейдем к безумно популярной сейчас модели машинного обучения: нейронным сетям.

Нейросетевой анализ в одной строке кода

В последние годы популярность нейронных сетей сильно возросла. В частности, благодаря усовершенствованию алгоритмов и методик обучения в этой сфере, но также и вследствие усовершенствования аппаратного обеспечения и возникновения технологии универсальных GPU (GPGPU). В этом разделе мы расскажем вам о *многослойном перцептроне* (multilayer

perceptron, MLP) — одном из самых популярных нейросетевых представлений. Прочитав раздел, вы сможете создать собственную нейронную сеть в одной строке кода Python!

Общее описание

Для этого однострочника я вместе с коллегами по посвященной Python рассылке подготовил специальный набор данных. Я ставил перед собой задачу создания правдоподобного набора данных, так что попросил своих подписчиков поучаствовать в эксперименте по генерации данных для текущей главы.

Данные

Если вы читаете эту книгу, значит, вас интересует изучение Python. Для создания интересного набора данных я задал моим подписчикам шесть анонимных вопросов об их опыте работы с Python и доходах. Ответы на эти вопросы и будут играть роль обучающих данных для простого примера нейронной сети (реализованной в однострочнике Python).

В основе обучающих данных лежат ответы на следующие шесть вопросов.

1. Сколько часов за последние семь дней вы работали с кодом на Python?
2. Сколько лет назад вы начали изучать компьютерные науки?
3. Сколько книг по написанию кода стоит на ваших полках?
4. Какой процент вашего времени, посвященного Python, вы тратите на работу над реальными проектами?
5. Сколько вы зарабатываете в месяц (округленно до тысяч долларов) благодаря своим профессиональным навыкам (в самом широком смысле)?
6. Какой у вас приблизительно рейтинг на Finxter, округленно до сотен баллов?

Первые пять вопросов[1] — входной сигнал нейронной сети, а шестой — выходной. В этом однострочнике мы производим регрессию на основе нейронной сети. Другими словами, мы предсказываем числовое значение (уровень владения Python) по числовым входным признакам. Мы не станем изучать в нашей книге классификацию с помощью нейронных сетей — еще одну их сильную сторону.

[1] Точнее, ответы на них.

Ответ на шестой вопрос приближенно описывает уровень владения Python программистом. Finxter (https://finxter.com/) — наше приложение, обучающее с помощью маленьких задач на Python и присваивающее программистам на Python рейтинг в зависимости от успешности решения ими этих задач. Таким образом, оно помогает количественно выразить уровень владения Python.

Начнем с визуализации степени влияния на выходной сигнал каждого из вопросов (рейтинг владения Python разработчиком), как показано на рис. 4.17.

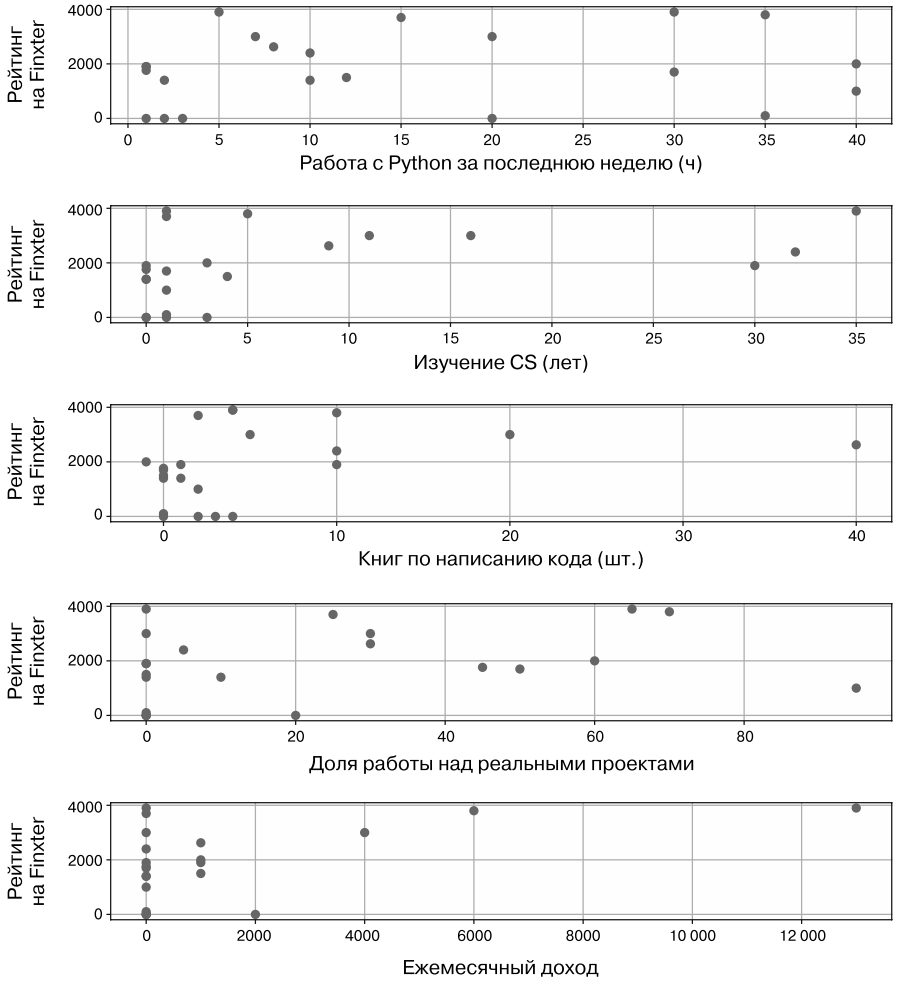


Рис. 4.17. Взаимосвязь между ответами на опросник и рейтингом владения Python на Finxter

Обратите внимание, что эти графики демонстрируют лишь влияние отдельных признаков (ответов на вопросы) на итоговый рейтинг Finxter, но ничего не говорят о влиянии сочетаний двух или более признаков. Отметим также, что некоторые разработчики Python не ответили на всех шесть вопросов; в подобных случаях использовалось фиктивное значение `-1`.

Искусственные нейронные сети

Идея создания теоретической модели человеческого мозга (естественной нейронной сети) всесторонне изучается в последние десятилетия. Но основы искусственных нейронных сетей были заложены еще в 1940-х и 1950-х! С тех пор идеи искусственных нейронных сетей непрерывно пересматривались и усовершенствовались.

Основная идея состоит в разбиении обширной задачи обучения и вывода на множество микрозадач, не независимых друг от друга, а напротив, тесно взаимосвязанных. Мозг содержит миллиарды нейронов, связанных с помощью триллионов синапсов. В упрощенной модели обучение представляет собой просто подстройку *мощности* синапсов (называемых также в искусственных нейронных сетях *весами*, или *параметрами*). Как же создать в такой модели новый синапс? Очень просто — всего лишь увеличить его вес с нуля до ненулевого значения.

На рис. 4.18 приведена простейшая нейронная сеть из трех слоев (входной, скрытый и выходной), каждый из которых состоит из нескольких нейронов, связанных между собой, начиная от входного слоя через скрытый и до выходного.

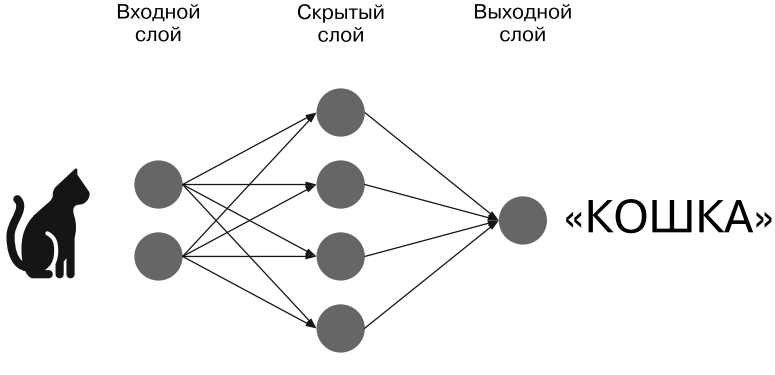


Рис. 4.18. Простая нейронная сеть для классификации животных

В этом примере нейронная сеть обучается обнаруживать животных на изображениях. На практике в качестве входного слоя используется по одному входному нейрону на каждый пиксел изображения. В результате получаются миллионы входных нейронов, связанных с миллионами скрытых нейронов. Обычно каждый выходной нейрон отвечает за один бит общего выходного сигнала. Например, для обнаружения двух различных животных (скажем, кошки и собаки) можно воспользоваться одним нейроном выходного слоя, моделирующим два различных состояния (`0=кошка`, `1=собака`).

Идея заключается в активации (возбуждении) нейрона при попадании в него определенного входного импульса. При этом каждый нейрон возбуждается (или не возбуждается) отдельно от других, в зависимости от мощности его входного сигнала. Таким образом моделируется человеческий мозг, в котором нейроны активируют друг друга через импульсы. Активация входных нейронов распространяется по сети, пока не достигнет выходных нейронов. Часть их будет активирована, а часть — нет. Конкретная картина возбуждения выходных нейронов и формирует итоговый выходной сигнал (предсказание) искусственной нейронной сети. Возбужденный выходной нейрон в модели, допустим, может кодировать 1, а невозбужденный — 0. Таким образом можно обучить нейронную сеть предсказывать все, что только можно кодировать с помощью последовательности 0 и 1 (то есть все, что может представить компьютер).

Посмотрим на математическую картину работы нейронов (рис. 4.19).

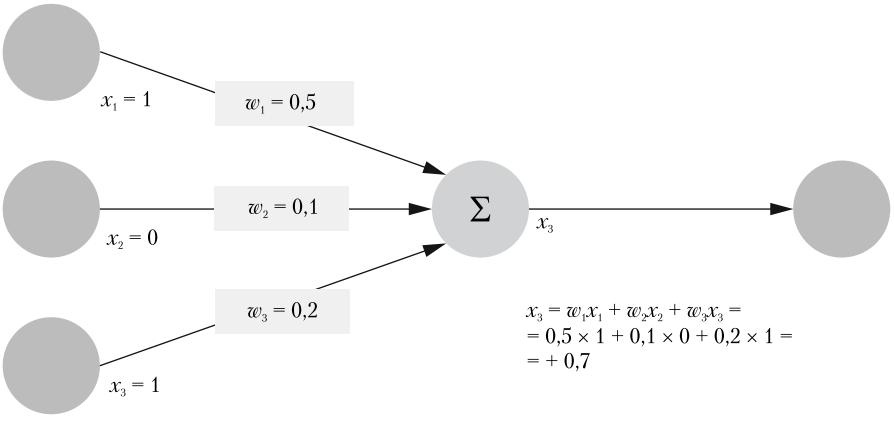


Рис. 4.19. Математическая модель отдельного нейрона: выходной сигнал представляет собой функцию трех входных сигналов

Каждый нейрон соединен с другими, но не все соединения равноценны, у каждого — свой вес. Строго говоря, возбужденный нейрон распространяет импульс 1 на следующие нейроны, а невозбужденный распространяет импульс 0. Вес, образно говоря, задает долю импульса возбуждающегося входного нейрона, передаваемую в следующий нейрон через соединение. Математически входной сигнал следующего нейрона вычисляется путем умножения импульса на вес соединения.

В нашем примере нейрон просто вычисляет свой выходной сигнал, суммируя все входные сигналы. Конкретная схема генерации нейроном выходного сигнала на основе входных называется его *функцией активации* (activation function). В нашем примере вероятность возбуждения нейрона выше, если соответствующие входные нейроны также возбуждаются. Именно так импульсы и распространяются по нейронной сети.

Как работает алгоритм обучения? Обучающие данные используются для выбора весов w нейронной сети. При фиксированном входном значении x различные веса w приводят к различным выходным сигналам. Поэтому алгоритм обучения постепенно меняет веса w — за много итераций, — пока выходной слой не начинает выдавать результаты, аналогичные обучающим данным. Другими словами, алгоритм обучения постепенно сокращает погрешность предсказания обучающих данных.

Существует множество схем сетей, алгоритмов обучения и функций активации. В этой главе показан практичный подход, позволяющий воспользоваться нейронной сетью прямо сейчас, с помощью одной строки кода. Если захотите, то можете изучить потом дополнительные подробности (например, можете начать с чтения статьи «Нейронная сеть» в «Википедии», https://ru.wikipedia.org/wiki/Нейронная_сеть).

Код

Наша цель — создать нейронную сеть для предсказания уровня владения Python (рейтинга Finxter), исходя из пяти входных признаков (ответов на вопросы):

- `WEEK` — сколько часов за последние семь дней вы работали с кодом на Python?
- `YEARS` — сколько лет назад вы начали изучать компьютерные науки?

- `BOOKS` — сколько книг по написанию кода стоит на ваших полках?
- `PROJECTS` — какой процент вашего времени, посвященного Python, вы тратите на разработку реальных проектов?
- `EARN` — сколько вы зарабатываете в месяц (округленно до тысяч долларов) благодаря своим профессиональным навыкам (в самом широком смысле)?

Снова встанем на плечи гигантов и воспользуемся библиотекой scikit-learn (`sklearn`) для нейросетевой регрессии, как показано в листинге 4.5.

Листинг 4.5. Нейросетевой анализ в одной строке кода

```
## Зависимости
from sklearn.neural_network import MLPRegressor
import numpy as np

## Данные опросника (WEEK, YEARS, BOOKS, PROJECTS, EARN, RATING)
X = np.array(
    [[20,  11,  20,  30,  4000,  3000],
     [12,   4,   0,   0, 1000,  1500],
     [2,   0,   1,  10,   0,  1400],
     [35,   5,  10,  70,  6000,  3800],
     [30,   1,   4,  65,   0,  3900],
     [35,   1,   0,   0,   0, 100],
     [15,   1,   2,  25,   0,  3700],
     [40,   3,  -1,  60,  1000,  2000],
     [40,   1,   2,  95,   0,  1000],
     [10,   0,   0,   0,   0,  1400],
     [30,   1,   0,  50,   0,  1700],
     [1,   0,   0,  45,   0,  1762],
     [10,  32,  10,   5,   0,  2400],
     [5,  35,   4,   0, 13000,  3900],
     [8,   9,  40,  30,  1000,  2625],
     [1,   0,   1,   0,   0,  1900],
     [1,  30,  10,   0,  1000,  1900],
     [7,  16,   5,   0,   0,  3000]])

## Однострочник
neural_net = MLPRegressor(max_iter=10000).fit(X[:,:-1], X[:,-1])

## Результат
res = neural_net.predict([[0, 0, 0, 0, 0]])
print(res)
```

Вычислить результаты этого кода человеку поистине невозможно, но не хотите ли попытаться?

Принцип работы

В первых нескольких строках кода мы создаем набор данных. Входной формат алгоритмов машинного обучения в библиотеке scikit-learn одинаков. Строки соответствуют отдельным наблюдениям, состоящим из нескольких признаков. Чем больше строк, тем больше обучающих данных; чем больше столбцов, тем больше признаков в каждом наблюдении. В данном случае у нас пять входных и один выходной признак для каждого элемента обучающих данных.

Наш однострочник создает нейронную сеть с помощью конструктора класса `MLPRegressor`, в который я передаю `max_iter=10000` в качестве аргумента, поскольку обучение не сходится при использовании количества итераций по умолчанию (`max_iter=200`).

После этого мы вызываем функцию `fit()` для определения параметров нейронной сети. После вызова `fit()` начальные значения нейронной сети успешно заданы. Функция `fit()` принимает в качестве параметров многомерный входной массив (по одному наблюдению в строке) и одномерный выходной массив (размер которого равен количеству наблюдений).

Осталось только вызвать функцию `predict` с какими-нибудь входными значениями:

```
## Результат
res = neural_net.predict([[0, 0, 0, 0, 0]])
print(res)
# [94.94925927]
```

Учтите, что фактические результаты ее выполнения могут немного отличаться вследствие недетерминистичной природы самой функции и различного характера сходимости.

Говоря простым языком: если...

- ... за последние семь дней вы работали с кодом на Python 0 часов;
- ... начали изучать компьютерные науки 0 лет назад;
- ... на ваших полках стоит 0 книг по написанию кода;
- ... вы тратите на работу над реальными проектами 0 % вашего времени, посвященного Python;
- ... вы зарабатываете благодаря своим профессиональным навыкам в месяц 0 долларов;

то нейронная сеть оценит ваш уровень владения Python как *очень* низкий (рейтинг Finxter в 94 балла означает, что вы с трудом понимаете даже программу `print("hello, world")` на Python)[1].

Поменяем параметры: что будет, если вы потратили 20 часов на изучение Python и вернулись к программе через неделю:

```
## Результат
res = neural_net.predict([[20, 0, 0, 0, 0]])
print(res)
# [440.40167562]
```

Совсем неплохо — ваш уровень владения Python существенно вырос! Но такой рейтинг вас все еще не вполне удовлетворяет, правда? (У более или менее приличного программиста на Python рейтинг на Finxter, по крайней мере, составляет 1500–1700.)

Никаких проблем. Купите десять книг по Python (с учетом этой только девять). Посмотрим, что произойдет с вашим рейтингом:

```
## Результат
res = neural_net.predict([[20, 0, 10, 0, 0]])
print(res)
# [953.6317602]
```

И снова заметен существенный прогресс, ваш рейтинг удвоился! Но одна только покупка книг по Python не слишком помогает, нужно еще и изучить их! Потратим на это год:

```
## Результат
res = neural_net.predict([[20, 1, 10, 0, 0]])
```

---

[1] При запуске программа дает не те результаты, что описывает здесь автор. В частности, выходной признак не увеличивается, а уменьшается при WEEK=20 по сравнению с WEEK=0:

neural_net = MLPRegressor(max_iter=10000).fit(X[:,:–1], X[:,–1])
res = neural_net.predict([[0, 0, 0, 0, 0]])
print(res)
[1320.6305953]
res = neural_net.predict([[20, 0, 0, 0, 0]])
print(res)
[816.44206182]

Это видно даже в репозитории автора на GitHub: https://github.com/finxter/PythonOneLiners/blob/master/book/machine_learning/sklearn_one_liner_05.py.

```
print(res)
# [999.94308353]
```

Рейтинг практически не изменился. Вот теперь я не слишком верю в результаты нашей нейронной сети. Мне кажется, что рейтинг должен был вырасти хотя бы до 1500. Но это означает лишь то, что нейронная сеть демонстрирует настолько хорошие результаты, насколько хороши ее обучающие данные. А данных у нас было очень немного, и нейронная сеть явно не может обойти такое ограничение: в представленной горстке точек данных просто слишком мало информации.

Но мы не сдадимся, правда? Далее вы попробуете потратить 50 % своего времени, посвященного Python, на работу в качестве Python-фрилансера[1]:

```
## Результат
res = neural_net.predict([[20, 1, 10, 50, 1000]])
print(res)
# [1960.7595547]
```

Ух ты! Внезапно нейронная сеть стала считать вас экспертом по Python. Несомненно, мудрое предсказание нейронной сети! Изучайте Python хотя бы год и решайте реальные задачи — и станете прекрасным программистом!

Резюмируя: вы изучили основы нейронных сетей и научились использовать их с помощью всего одной строки кода на Python. Любопытно, что наш опросник демонстрирует, что работа с реальными проектами — возможно, даже выполнение с самого начала каких-либо проектов в качестве фрилансера — вносит немалый вклад в успешность вашей учебы. Наша нейронная сеть явно об этом знает. Если хотите узнать о моей личной стратегии того, как стать фрилансером, то можете присоединиться к бесплатному вебинару о последних достижениях в сфере фриланса на Python: https://blog.finxter.com/webinar-freelancer/.

В следующем разделе мы займемся еще одним многообещающим представлением: деревьями принятия решений. В отличие от нейронных сетей, обучение которых порой требует немалых вычислительных ресурсов (зачастую множества компьютеров и многих часов, а иногда и недель), деревья принятия решений весьма «дешевы» в этом отношении. И притом представляют собой быстрый и эффективный способ выделения закономерностей в обучающих данных.

---

[1] С зарплатой 1000 долларов.

Машинное обучение с помощью деревьев принятия решений в одной строке кода

Деревья принятия решений (decision trees) — интуитивный инструмент для вашего набора инструментов машинного обучения. Он обладает большими возможностями. Важное преимущество деревьев принятия решений в том, что, в отличие от многих других методик машинного обучения, деревья удобны для непосредственного восприятия человеком. Вы можете легко обучить дерево принятия решений и показать его своему начальству, которому не нужно знать ничего о машинном обучении, чтобы понять, что делает модель. Особенно это удобно для исследователей данных, которым часто приходится демонстрировать свои результаты перед руководством и защищать их. В этом разделе я покажу вам, как использовать деревья принятия решений с помощью одной строки кода Python.

Общее описание

В отличие от многих алгоритмов машинного обучения, идеи деревьев принятия решений, вероятно, знакомы вам из жизненного опыта. Это просто структурированный способ принятия решений. Каждое решение приводит к дальнейшему ветвлению. Ответив на серию вопросов, вы в конечном итоге приходите к желаемой рекомендации. На рис. 4.20 приведен пример.

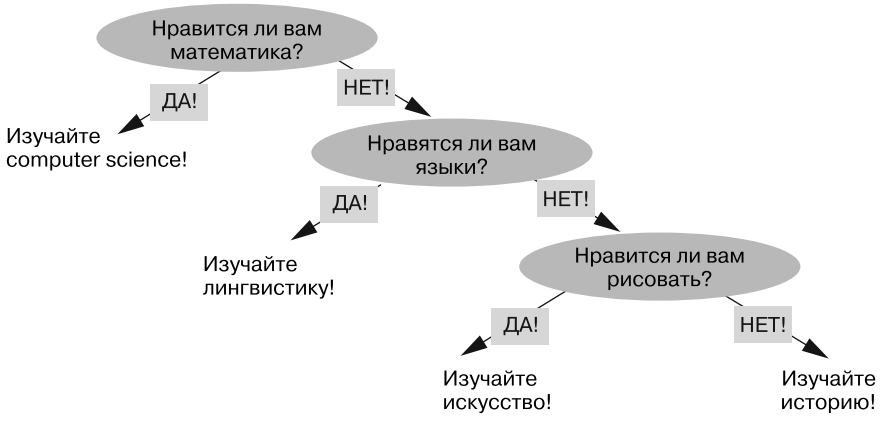


Рис. 4.20. Упрощенное дерево принятия решений для рекомендаций изучаемых предметов

Деревья принятия решений используются для задач классификации наподобие «Какой предмет мне изучать в зависимости от моих интересов?». Начинается принятие решения сверху, после чего последовательно даются ответы на вопросы и выбираются варианты, лучше всего описывающие имеющиеся признаки. Наконец, достигается *лист* дерева — вершина, у которой отсутствуют *потомки*. Она и будет классом, рекомендуемым в соответствии с выбранными признаками.

У обучения на основе деревьев принятия решений есть множество нюансов. В предыдущем примере вес первого вопроса существенно превышает вес последнего. Если вам нравится математика, то дерево принятия решений никогда не порекомендует вам изучать искусства или лингвистику. Это удобно, ведь некоторые признаки могут быть важнее для классификации, чем другие. Например, система классификации, предсказывающая текущее состояние здоровья, может использовать пол (признак) для исключения многих болезней (классов).

Следовательно, порядок вершин дерева можно использовать для оптимизации работы системы: признаки, сильнее всего влияющие на итоговую классификацию, нужно помещать сверху. При обучении на основе деревьев принятия решений это позволяет затем агрегировать вопросы, слабо влияющие на итоговую классификацию, как показано на рис. 4.21.

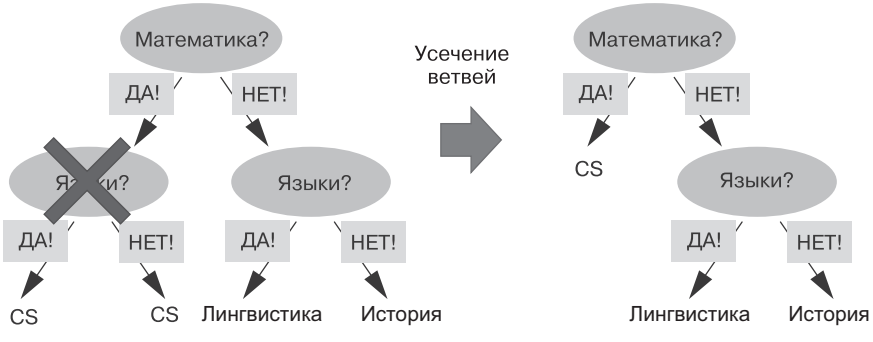


Рис. 4.21. Усечение ветвей повышает эффективность обучения на основе деревьев принятия решений

Пусть полное дерево принятия решений выглядит так, как показано слева на рис. 4.21. Для любого сочетания признаков есть отдельный итог

классификации (лист дерева). Однако некоторые признаки могут не давать никакой дополнительной информации относительно задачи классификации (например, первая вершина «Язык» в нашем примере). Обучение на основе деревьев принятия решений позволяет избавиться от этих вершин из соображений эффективности — процесс так называемого *усечения ветвей* (pruning).

Код

Создать собственное дерево принятия решений можно с помощью одной строки кода на языке Python. Листинг 4.6 демонстрирует, как именно.

Листинг 4.6. Дерево принятия решений с помощью одной строки кода Python

```
## Зависимости
from sklearn import tree
import numpy as np

## Данные: оценки студентов по (математика, языки, творческие
## способности) --> предмет для изучения
X = np.array([[9, 5, 6, "computer science"],
              [1, 8, 1, "linguistics"],
              [5, 7, 9, "art"]])

## Однострочник
Tree = tree.DecisionTreeClassifier().fit(X[:,:-1], X[:,-1])

## Результат
student_0 = Tree.predict([[8, 6, 5]])
print(student_0)

student_1 = Tree.predict([[3, 7, 9]])
print(student_1)
```

Догадайтесь, какими будут результаты выполнения этого фрагмента кода!

Принцип работы

Данные в коде описывают трех студентов с оценками их успехов (от 1 до 10) в трех областях: математика, языки и творческие способности. Кроме того, нам известны предметы изучения этих студентов. Например, первый студент демонстрирует способности к математике и изучает computer science. Второй студент демонстрирует бóльшие способности к языкам, чем к двум другим областям, и изучает лингвистику. Третий студент демонстрирует хорошие творческие способности и изучает искусство.

Наш однострочник создает новый объект дерева принятия решений и обучает модель на маркированных обучающих данных (последний столбец — метки) с помощью функции `fit()`. «Под капотом» при этом создаются три вершины, по одной для каждого признака: математика, языки, творческие способности. При предсказании класса для `student_0` (математика=8, языки=6, творческие способности=5) дерево принятия решений возвращает `computer science`. Оно усвоило, что такая закономерность признаков (высокий, средний, средний) указывает на первый класс. С другой стороны, при наборе признаков (3, 7, 9) дерево принятия решений предсказывает `art`, поскольку усвоило, что оценки (низкая, средняя, высокая) указывают на третий класс.

Обратите внимание, что алгоритм — недетерминистичный. Другими словами, при выполнении одного и того же кода дважды могут быть возвращены различные результаты. Подобное поведение часто встречается в алгоритмах машинного обучения, имеющих дело с генераторами случайных чисел. В данном случае порядок признаков случаен, так что порядок признаков в итоговом дереве принятия решений может различаться.

Резюмируя: деревья принятия решений — интуитивный способ создания моделей машинного обучения, удобных для восприятия человеком. Каждая ветвь отражает выбор, основанный на отдельном признаке нового примера данных. Листья дерева отражают итоговое предсказание (классификацию или регрессию). Далее мы ненадолго оставим конкретные алгоритмы машинного обучения и рассмотрим важнейшее для машинного обучения понятие: дисперсию.

Получение строки с минимальной дисперсией в одной строке кода

Возможно, вы читали о пяти V больших данных: объем (**v**olume), скорость (**v**elocity), разнообразие (**v**ariety), достоверность (**v**eracity) и ценность (**v**alue)[1]. *Дисперсия* (**v**ariance) — еще одно важное V: это мера ожидаемого (квадратичного) отклонения данных от среднего значения. На практике дисперсия — важный показатель, нашедший свои приложения в финансах, прогнозе погоды и обработке изображений.

[1] Иногда выделяют не пять, а три V, а порой даже семь: помимо перечисленных, еще и изменчивость (variability) и визуализацию (visualization).

Общее описание

Дисперсия — это мера того, насколько данные разбросаны вокруг их среднего значения в одномерном или многомерном пространстве. Чуть ниже вы увидите наглядный пример. Фактически дисперсия — один из важнейших показателей в машинном обучении. Она захватывает обобщенные закономерности в данных, а машинное обучение прежде всего ориентировано на распознавание закономерностей.

В основе многих алгоритмов машинного обучения лежит дисперсия в той или иной форме. Например, *подбор правильного соотношения систематической ошибки и дисперсии* — хорошо известная задача в машинном обучении: хитроумные модели машинного обучения подвержены риску переобучения (высокая дисперсия), но очень точно отражают обучающие данные (маленькая систематическая ошибка). С другой стороны, простые модели часто хорошо обобщаются (низкая дисперсия), но плохо отражают данные (большая систематическая ошибка).

Что же такое дисперсия? Это простой статистический показатель, отражающий степень разбросанности данных относительно их среднего значения. На рис. 4.22 приведен пример в виде графиков двух наборов данных: один с низкой дисперсией, а второй — с высокой.

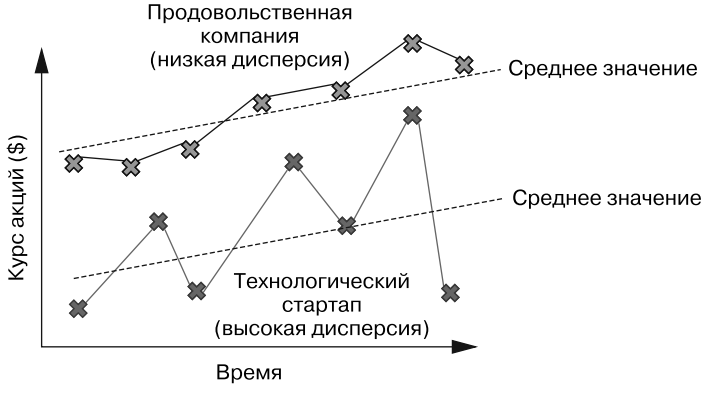


Рис. 4.22. Сравнение дисперсии курсов акций двух компаний

В этом примере показаны курсы акций двух компаний. Курс акций технологического стартапа сильно отклоняется от среднего значения. Курс акций продовольственной компании довольно стабилен и лишь незначительно

отклоняется от среднего значения. Другими словами, у технологического стартапа высокая дисперсия, а у продовольственной компании — низкая.

На математическом языке вычислить дисперсию $var(X)$ множества числовых значений X можно с помощью следующей формулы:

$$var(X) = \sum_{x \in X} (x - \bar{x})^2.$$

Величина представляет собой среднее значение данных во множестве X.

Код

По мере старения многие инвесторы стремятся сократить общий риск своего инвестиционного портфеля. Согласно доминирующей философии инвестиций, следует стремиться к акциям с более низкой дисперсией как менее рискованным инвестиционным активам. Проще говоря, риск потерять деньги будет ниже при инвестициях в стабильную, предсказуемую, крупную компанию, чем при вложениях в маленький технологический стартап.

Задача однострочника из листинга 4.7 — найти в портфеле акции с минимальной дисперсией. При инвестициях в эти акции можно ожидать более низкую общую дисперсию портфеля.

Листинг 4.7. Вычисление минимальной дисперсии в одной строке кода

```
## Зависимости
import numpy as np

## Данные (строки: акции / столбцы: курсы акций)
X = np.array([[25,27,29,30],
              [1,5,3,2],
              [12,11,8,3],
              [1,1,2,2],
              [2,6,2,2]])

## Однострочник
## Поиск акций с наименьшей дисперсией
min_row = min([(i,np.var(X[i,:])) for i in range(len(X))], key=lambda x: x[1])

## Результат
print("Row with minimum variance: " + str(min_row[0]))
print("Variance: " + str(min_row[1]))
```

Какими же будут результаты выполнения этого фрагмента кода?

Принцип работы

Как обычно, мы сначала описываем данные, на которых будет работать наш однострочник (см. верхнюю часть листинга 4.7). Массив NumPy `X` содержит пять строк (по одной для каждого вида акций в портфеле), каждая из которых включает четыре значения (курсы акций).

Задача — найти идентификатор и дисперсию акций с минимальной дисперсией. Поэтому внешняя функция нашего однострочника — `min()`. Мы выполняем ее для последовательности кортежей `(a,b)`, в которых первое значение кортежа `a` представляет собой индекс строки (индекс вида акций), а второе значение кортежа `b` — дисперсию строки.

Возможно, вы спросите: а что такое минимальное значение последовательности кортежей? Конечно, необходимо четко определить эту операцию перед использованием. Для этого мы обратимся к аргументу `key` функции `min()`. В него должна передаваться функция, получающая последовательность и возвращающая допускающее сравнение объектное значение. Опять же, значения в нашей последовательности — кортежи, и нам нужно найти кортеж с минимальной дисперсией (вторым значением кортежа). А поскольку дисперсия — второе значение в кортеже, мы возвращаем для сравнения `x[1]`. Другими словами, «победителем» становится кортеж с минимальным вторым значением.

Посмотрим, как создать эту последовательность значений-кортежей. Чтобы создать кортеж для индекса строки (вида акций), мы прибегнем к списковому включению. Первый элемент кортежа — просто индекс строки *i*. Второй элемент кортежа — дисперсия этой строки. Для вычисления дисперсии строки мы используем функцию `var()` библиотеки NumPy в сочетании со срезом.

Результат выполнения нашего однострочника выглядит так:

```
"""
Row with minimum variance: 3
Variance: 0.25
"""
```

Хотелось бы добавить, что существует и альтернативный способ решения этой задачи. Если бы наша книга не была посвящена однострочникам Python, возможно, я предпочел бы следующее решение вместо упомянутого однострочника:

```
var = np.var(X, axis=1)
min_row = (np.where(var==min(var)), min(var))
```

В первой строке вычисляется дисперсия массива NumPy `X` по столбцам (`axis=1`). Во второй создается кортеж. Первое значение кортежа представляет собой индекс минимума в массиве дисперсий. А второе — сам этот минимум в массиве дисперсий. Обратите внимание, что одна и та же (минимальная) дисперсия может быть у нескольких строк.

Такое решение удобнее для восприятия. Поэтому явно необходимо выбирать между лаконичностью и удобочитаемостью кода. Возможность втиснуть что-либо в одну строку кода не означает, что так следует поступать всегда. При прочих равных условиях лучше писать лаконичный *и* удобочитаемый код, а не раздувать его ненужными определениями, комментариями или промежуточными шагами.

Теперь, изучив в этом разделе основы понятия дисперсии, вы готовы узнать, как вычислять основные статистические показатели.

Основные статистические показатели с помощью одной строки кода

Исследователю данных, как и специалисту по машинному обучению, необходимо знать основные статистические показатели. Некоторые алгоритмы машинного обучения базируются исключительно на статистических показателях (например, байесовские сети).

Например, вычисление основных статистических показателей матриц (среднего значения, дисперсии или стандартного отклонения) — неотъемлемая составляющая анализа разнообразных наборов данных, в том числе финансовых, медицинских и данных соцсетей. По мере роста популярности машинного обучения и науки о данных умение применять библиотеку NumPy, используемую в Python для исследования данных, статистики и линейной алгебры, становится все более востребованным на рынке.

Из этого однострочника вы увидите, как вычислять основные статистические показатели с помощью NumPy.

Общее описание

В этом разделе рассказывается, как вычислить среднее значение, стандартное отклонение и дисперсию по одной из осей координат. Вычисления этих трех

величин очень похожи; если вы разберетесь с одной, то и остальные будут для вас понятны.

Вот что мы хотим сделать: по заданному массиву NumPy данных об акциях, в котором строки соответствуют различным компаниям, а столбцы — курсам их акций по дням, найти среднее значение и стандартное отклонение курса акций каждой из компаний (рис. 4.23).

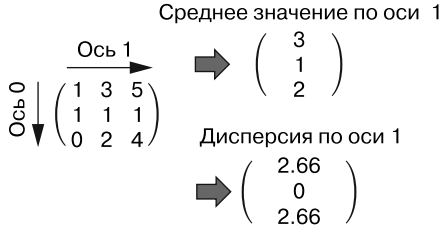


Рис. 4.23. Среднее значение и дисперсия по оси 1

В примере показан двумерный массив NumPy, но на практике массивы бывают намного большей размерности.

Среднее арифметическое, дисперсия, стандартное отклонение

Прежде чем выяснить, как вычислить все это в NumPy, понемногу разберемся со всем, что нужно для этого знать. Пусть нам нужно вычислить просто среднее значение, дисперсию или стандартное отклонение по всем значениям массива NumPy. Вы уже видели в главе примеры вычисления среднего значения и дисперсии. Стандартное отклонение равно просто квадратному корню дисперсии. Вычислить их все легко можно с помощью следующих функций:

```
import numpy as np

X = np.array([[1, 3, 5],
              [1, 1, 1],
              [0, 2, 4]])

print(np.average(X))
# 2.0

print(np.var(X))
# 2.4444444444444446

print(np.std(X))
# 1.5634719199411433
```

Наверное, вы обратили внимание, что функции применяются тут к двумерному массиву NumPy `X`. Но NumPy просто «сплющивает» этот массив и вычисляет функции на основе полученного результата. Например, среднее арифметическое «сплющенного» массива NumPy `X` вычисляется следующим образом:

$$(1 + 3 + 5 + 1 + 1 + 1 + 0 + 2 + 4) / 9 = 18 / 9 = 2.0.$$

Вычисление среднего значения, дисперсии, стандартного отклонения по одной из осей

Иногда, впрочем, бывает нужно вычислить эти функции по одной из осей. Для этого можно указать ключевое слово `axis` в качестве аргумента функций вычисления среднего значения, дисперсии или стандартного отклонения (см. подробное описание аргумента `axis` в главе 3).

Код

В листинге 4.8 показано, как именно вычислить среднее значение, дисперсию и стандартное отклонение по одной из осей. Наша задача: вычислить среднее значение, дисперсию и стандартное отклонение всех типов акций в двумерной матрице, строки которой соответствуют типам акций, а столбцы — курсам по дням.

Листинг 4.8. Вычисление простейших статистических показателей по одной из осей

```
## Зависимости
import numpy as np

## Данные о курсах акций: 5 компаний
## (строка=[курс_день_1, курс_день_2, ...])
x = np.array([[8, 9, 11, 12],
              [1, 2, 2, 1],
              [2, 8, 9, 9],
              [9, 6, 6, 3],
              [3, 3, 3, 3]])

## Однострочник
avg, var, std = np.average(x, axis=1), np.var(x, axis=1), np.std(x, axis=1)

## Результат
print("Averages: " + str(avg))
print("Variances: " + str(var))
print("Standard Deviations: " + str(std))
```

Угадайте, какой результат вернет этот код!

Принцип работы

В нашем однострочнике с помощью ключевого слова `axis` задается ось координат, по которой вычисляется среднее значение, дисперсия и стандартное отклонение. Например, если выполнять эти три функции по оси `axis=1`, то каждая из строк будет агрегирована в одно значение. А значит, размерность полученного в итоге массива NumPy уменьшится до единицы.

Приведенный выше код возвращает следующее:

```
"""
Averages: [10.   1.5  7.   6.   3. ]
Variances: [2.5  0.25 8.5  4.5  0.  ]
Standard Deviations: [1.58113883 0.5   2.91547595 2.12132034 0.   ]
"""
```

Прежде чем перейти к следующему однострочнику, я хотел бы продемонстрировать применение той же идеи для массивов NumPy еще большей размерности.

При агрегировании по одной из осей многомерного массива NumPy всегда агрегируется ось, указанная в аргументе `axis`. Вот пример:

```
import numpy as np

x = np.array([[[1,2], [1,1]],
              [[1,1], [2,1]],
              [[1,0], [0,0]]])

print(np.average(x, axis=2))
print(np.var(x, axis=2))
print(np.std(x, axis=2))

"""
[[1.5 1. ]
 [1.  1.5]
 [0.5 0. ]]
[[0.25 0.  ]
 [0.   0.25]
 [0.25 0.  ]]
[[0.5 0. ]
 [0.  0.5]
 [0.5 0. ]]
"""
```

Здесь приведены три примера вычисления среднего значения, дисперсии и стандартного отклонения по оси 2 (самая внутренняя ось координат;

см. главу 3). Другими словами, все значения оси 2 будут схлопнуты в одно, в результате чего ось 2 пропадет из итогового массива. Взгляните на эти три примера внимательнее и разберитесь, как именно ось 2 схлопывается в одно среднее значение, дисперсию или стандартное отклонение.

Резюмируя: умение извлекать хотя бы простейшую полезную информацию необходимо для широкого спектра наборов данных (в том числе финансовых, медицинских и данных соцсетей). Из этого раздела вы узнали больше о том, как быстро и эффективно вычислять основные статистические показатели для многомерных массивов, необходимый шаг предварительной обработки для многих алгоритмов машинного обучения.

Классификация путем метода опорных векторов с помощью одной строки кода

Популярность *метода опорных векторов* (support-vector machines, SVM) сильно выросла за последние годы благодаря его высокой ошибкоустойчивости при классификации, в том числе в пространствах высокой размерности. Как ни удивительно, SVM работает, даже если количество измерений (признаков) превышает количество элементов данных, что необычно для алгоритмов классификации из-за так называемого *проклятия размерности*: при росте размерности данные становятся очень разреженными, а это усложняет поиск закономерностей в наборе данных. Понимание основных идей SVM — необходимый шаг становления опытного специалиста по машинному обучению.

Общее описание

Как функционируют алгоритмы классификации? На основе обучающих данных они ищут границу решений, отделяющую данные из одного класса от данных из другого (в разделе «Логистическая регрессия в одной строке» на с. 130 границей решений может служить пороговое значение 0.5 вероятности сигма-функции).

Общая картина классификации

На рис. 4.24 приведен пример классификатора.

Пусть нам нужно создать рекомендательную систему для студентов начальных курсов университета. На рис. 4.24 визуализированы обучающие данные,

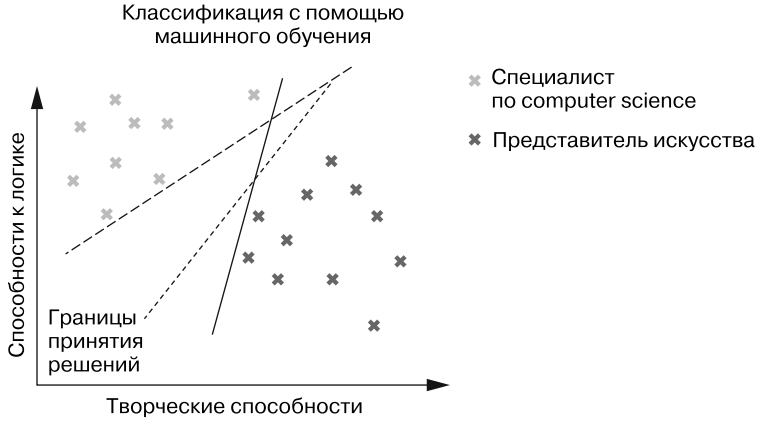


Рис. 4.24. Различия наборов способностей специалистов по computer science и представителей искусства

состоящие из пользователей, классифицированных по их способностям в двух сферах: логика и творчество. Одни студенты отличаются сильными логическими способностями и относительно низким уровнем творческих; другие — выраженными творческими способностями и относительно низким уровнем логических. Первую группу мы обозначили как *специалисты по computer science*, а вторую — *представители искусства*.

Для классификации новых пользователей модель машинного обучения должна отыскать границу решений, разделяющую специалистов по computer science и представителей искусства. В общих чертах будем классифицировать пользователей в зависимости от того, по какую сторону границы решений они попадают. В нашем примере мы классифицируем пользователей слева от границы решений как специалистов по computer science, а справа — как представителей искусства.

В двумерном пространстве роль границы решений может играть либо прямая, либо кривая (более высокого порядка). В первом случае классификатор называется *линейным* (linear classifier), а во втором — *нелинейным* (nonlinear classifier). В этом разделе мы будем рассматривать только линейные классификаторы.

На рис. 4.24 (см. выше) показаны три границы решений, вполне приемлемо разделяющие данные. В нашем примере невозможно определить, какая из этих границ решений лучше; они все обеспечивают абсолютную безошибочность при классификации обучающих данных.

Но какая же граница решений лучшая?

Метод опорных векторов дает уникальный и очень красивый ответ на этот вопрос. Вполне логично, что лучшая граница решений — та, которая обеспечивает максимальный «запас прочности». Другими словами, метод опорных векторов максимизирует расстояние между границей решений и ближайшими точками данных. Цель состоит в минимизации погрешности для новых точек, близких к границе решений.

Пример этого можно увидеть на рис. 4.25.

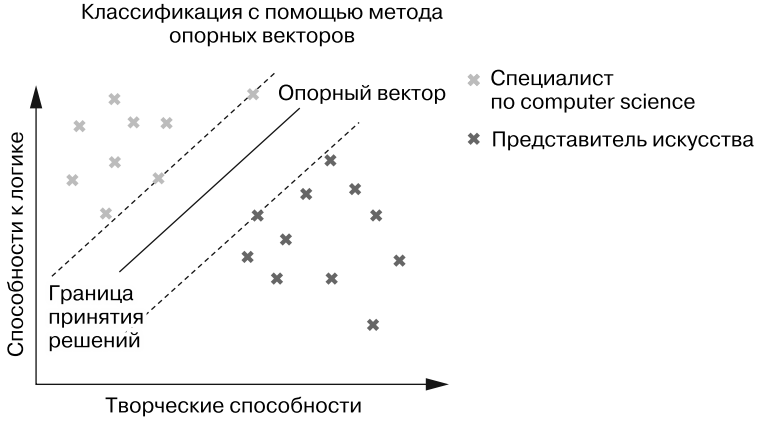


Рис. 4.25. Метод опорных векторов максимизирует допустимую погрешность

Классификатор SVM находит такие опорные векторы, чтобы область между ними была максимально широка. В данном случае роль опорных векторов играют точки данных, лежащие на двух пунктирных линиях, параллельных границе решений. Эти прямые называются *отступами* (margins). Граница решений — прямая посередине, расстояние от которой до отступов максимально. Вследствие максимизации области между отступами и границей решений можно ожидать, что *допустимая погрешность* также будет максимальной при классификации новых точек данных.

Код

Можно ли создать собственный алгоритм SVM с помощью одной строки кода на Python? Взгляните на листинг 4.9.

Листинг 4.9. Классификация с помощью SVM в одной строке кода

```
## Зависимости
from sklearn import svm
import numpy as np

## Данные: оценки студентов по (математика, языки, творческие
## способности) --> предмет для изучения
X = np.array([[9, 5, 6, "computer science"],
              [10, 1, 2, "computer science"],
              [1, 8, 1, "literature"],
              [4, 9, 3, "literature"],
              [0, 1, 10, "art"],
              [5, 7, 9, "art"]])

## Однострочник
svm = svm.SVC().fit(X[:,:-1], X[:,-1])

## Результат
student_0 = svm.predict([[3, 3, 6]])
print(student_0)

student_1 = svm.predict([[8, 1, 1]])
print(student_1)
```

Угадайте, что вернет этот код.

Принцип работы

Из кода понятно, как применять (в простейшем варианте) метод опорных векторов на Python. В массиве NumPy содержатся маркированные обучающие данные, по одной строке на пользователя и одному столбцу на признак (способности студентов к математике, языкам и творческие способности). Последний столбец — метка (класс).

Поскольку данные у нас — трехмерные, метод опорных векторов разделяет их с помощью двумерных плоскостей (линейный разделитель), а не одномерных прямых. Как вы, наверное, видите, можно также разделять три класса, а не два, как в предыдущих примерах.

Сам однострочник очень прост: сначала мы создаем модель с помощью конструктора класса `svm.SVC` (SVC расшифровывается как support-vector classification — *классификация с помощью опорных векторов*). Далее мы вызываем функцию `fit()`, производящую обучение на основе наших маркированных обучающих данных.

В части «Результат» фрагмента кода мы вызываем функцию `predict()`, передавая ей новые наблюдения. Поскольку для `student_0` указано

математика = 3, языки = 3 и творческие способности = 6, то метод опорных векторов предсказывает, что способностям студента соответствует метка `art`. Аналогично, для `student_1` с математика = 8, языки = 1 и творческие способности = 1 метод опорных векторов предсказывает, что способностям студента соответствует метка `computer science`.

Вот как выглядят итоговые результаты нашего однострочника:

```
## Результат
student_0 = svm.predict([[3, 3, 6]])
print(student_0)
# ['art']

student_1 = svm.predict([[8, 1, 1]])
print(student_1)
## ['computer science']
```

Резюмируя: SVM демонстрирует хорошие результаты даже в многомерных пространствах при количестве признаков, превышающем количество обучающих векторов данных. Идея максимизации *«запаса прочности»* вполне интуитивна и демонстрирует хорошие результаты даже при классификации *граничных случаев* (boundary cases) — векторов, попадающих в рамки этого «запаса прочности». В последней части данной главы мы рассмотрим более общую картину — метаалгоритм классификации: обучение ансамблей с помощью случайных лесов.

Классификация с помощью случайных лесов в одной строке кода

Посмотрим теперь на замечательную методику машинного обучения: *обучение ансамблей* (ensemble learning). Если степень безошибочности предсказаний вашей модели недостаточно высока, а срок сдачи проекта уже на носу — мой совет: попробуйте этот подход метаобучения, сочетающий предсказания (классификации) нескольких алгоритмов машинного обучения. Во многих случаях с его помощью вы сможете добиться в последнюю минуту лучших результатов.

Общее описание

В предыдущих разделах мы изучили несколько алгоритмов машинного обучения, с помощью которых можно быстро получить неплохие результаты.

Однако у разных алгоритмов — разные сильные стороны. Например, основанные на нейронных сетях классификаторы способны давать великолепные результаты для сложных задач, однако подвержены риску переобучения именно вследствие своих потрясающих способностей к усвоению тонких закономерностей данных. Обучение ансамблей для задач классификации частично решает проблему, связанную с тем, что заранее неизвестно, какой алгоритм машинного обучения сработает лучше всего.

Как работает этот подход? Создается метаклассификатор, состоящий из нескольких типов или экземпляров простых алгоритмов машинного обучения. Другими словами, обучается несколько моделей. В целях классификации конкретного наблюдения входные данные передаются по отдельности всем моделям. А роль *метапредсказания* играет класс, который эти модели возвращали чаще всего при этих входных данных. Он и становится итоговым результатом алгоритма обучения ансамблей.

Случайные леса (random forests) — особая разновидность алгоритмов обучения ансамблей, использующая обучение на основе деревьев принятия решений. Лес состоит из множества деревьев. Аналогично, случайный лес состоит из множества деревьев принятия решений. Отдельные деревья принятия решений получаются путем внесения стохастичности в процесс генерации деревьев на этапе обучения (например, выбор в качестве первой различных вершин дерева). В результате получаются различные деревья принятия решений — как раз то, что нужно.

На рис. 4.26 показан процесс предсказания для обученного случайного леса при следующем сценарии. У Алисы выраженные способности к математике и языкам. *Ансамбль* состоит из трех деревьев принятия решений (составляющих случайный лес). Чтобы классифицировать Алису, мы просим все эти деревья ее классифицировать. Два из трех деревьев классифицируют Алису как специалиста по computer science. Этот класс как получивший максимум «голосов» и возвращается в качестве окончательного результата классификации.

Код

Продолжим работать с этим примером классификации изучаемых предметов на основе демонстрируемых студентами способностей в трех областях: математика, языки и творчество. Возможно, вам кажется, что реализовать метод обучения ансамблей на Python непросто. Но благодаря многогранности библиотеки scikit-learn это не так (листинг 4.10).

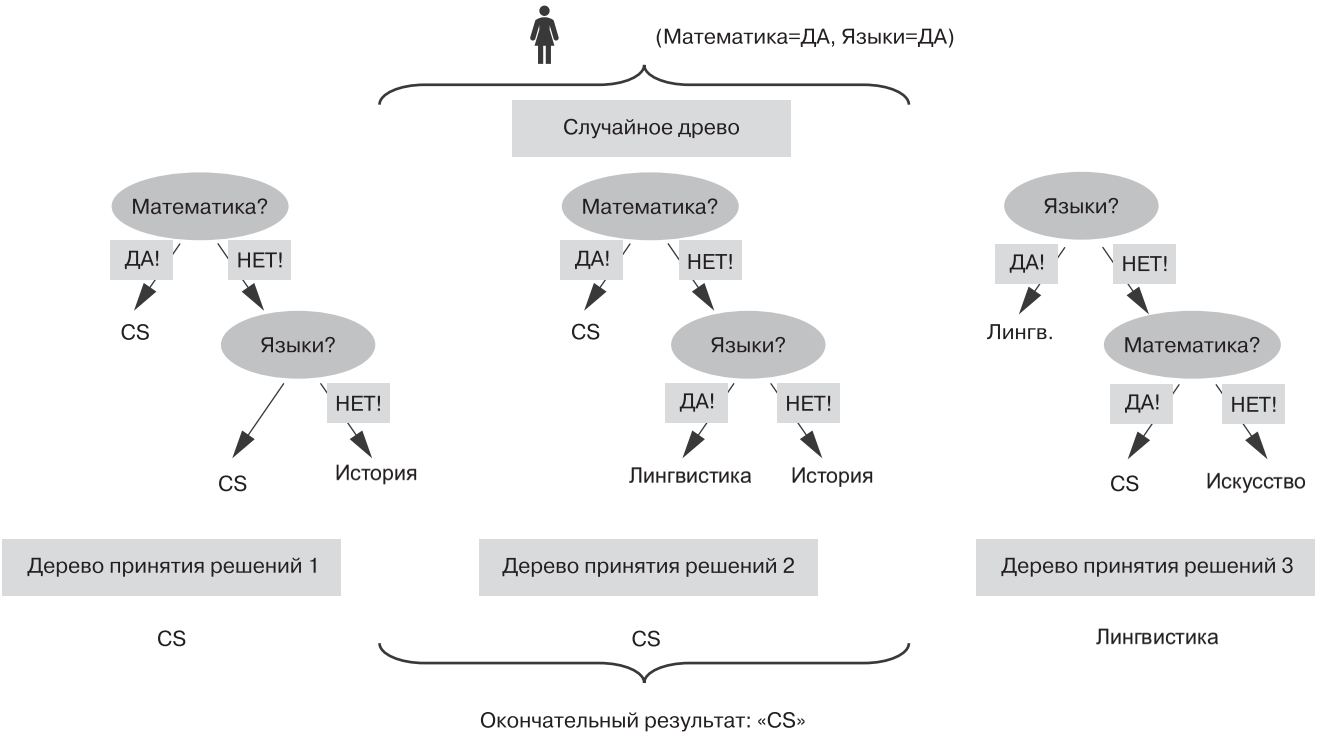


Рис. 4.26. Классификатор на основе случайного леса агрегирует результаты трех деревьев принятия решений

Листинг 4.10. Обучение ансамблей с помощью классификаторов на основе случайных лесов

```
## Зависимости
import numpy as np
from sklearn.ensemble import RandomForestClassifier

## Данные: оценки студентов по (математика, языки, творческие
## способности) --> предмет для изучения
X = np.array([[9, 5, 6, "computer science"],
              [5, 1, 5, "computer science"],
              [8, 8, 8, "computer science"],
              [1, 10, 7, "literature"],
              [1, 8, 1, "literature"],
              [5, 7, 9, "art"],
              [1, 1, 6, "art"]])

## Однострочник
Forest = RandomForestClassifier(n_estimators=10).fit(X[:,:-1], X[:,-1])

## Результат
students = Forest.predict([[8, 6, 5],
                           [3, 7, 9],
                           [2, 2, 1]])
print(students)
```

Попробуйте догадаться, каковы будут результаты выполнения этого фрагмента кода.

Принцип работы

Инициализировав массив маркированных обучающих данных в листинге 4.10, код создает случайный лес с помощью конструктора класса `RandomForestClassifier` с одним параметром — `n_estimators`, — задающим количество деревьев в лесу. Далее мы вызываем функцию `fit()`, заполняя данными полученную при инициализации модель (пустой лес). Используемые для этого входные обучающие данные состоят из всех столбцов массива `X`, кроме последнего, а метки обучающих данных задаются в этом последнем столбце. Как и в предыдущих примерах, соответствующие столбцы из массива данных `X` мы выделяем с помощью срезов.

Относящаяся к классификации часть этого фрагмента кода несколько отличается. Я хотел показать вам, как классифицировать много наблюдений, а не только одно. Это можно сделать тут путем создания многомерного массива, в котором каждому наблюдению соответствует одна строка.

Вот результаты работы нашего фрагмента кода:

```
## Результат
students = Forest.predict([[8, 6, 5],
                           [3, 7, 9],
                           [2, 2, 1]])
print(students)
# ['computer science' 'art' 'art']
```

Обратите внимание, что результаты по-прежнему недетерминистичны (могут отличаться при различных запусках этого кода), поскольку в алгоритме случайных лесов используется генератор случайных чисел, возвращающий различные числа в различные моменты времени. Детерминизировать этот вызов можно с помощью целочисленного аргумента `random_state`. Например, можно задать параметр `random_state=1` при вызове конструктора случайного леса: `RandomForestClassifier(n_estimators=10, random_state=1)`. В этом случае при каждом создании классификатора на основе случайных лесов будут возвращаться одни и те же результаты, поскольку будут генерироваться одни и те же случайные числа: в основе их всех лежит начальное значение генератора 1.

Резюмируя: в этом разделе представлен метаподход к классификации: снижение дисперсии погрешности классификации за счет использования результатов работы нескольких различных деревьев решений — одна из версий обучения ансамблей, при котором несколько базовых моделей объединяется в одну метамодель, способную задействовать все сильные стороны каждой из них.

ПРИМЕЧАНИЕ

Использование двух различных деревьев принятия решений может привести к высокой дисперсии погрешности, если одно возвращает хорошие результаты, а второе — нет. Последствия этого эффекта можно уменьшить с помощью случайных лесов.

Различные вариации этой идеи очень распространены в машинном обучении. Чтобы быстро повысить степень безошибочности модели, просто запустите несколько моделей машинного обучения и найдите среди их результатов лучшие (маленький секрет специалистов по машинному обучению). Методики обучения ансамблей в некотором смысле автоматически выполняют задачи, часто возлагаемые на экспертов по конвейерам машинного обучения:

выбор, сравнение и объединение результатов различных моделей машинного обучения. Основное преимущество обучения ансамблей — возможность выполнения его по отдельности для каждого значения данных во время выполнения.

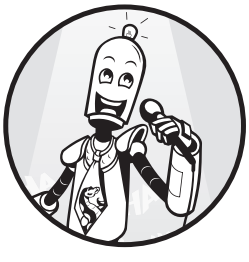
Итоги главы

В этой главе мы рассмотрели десять простых алгоритмов машинного обучения, необходимых для успешной работы в данной сфере. Вы посмотрели на предсказание значений с помощью алгоритмов регрессии, в частности, линейной, KNN и нейронных сетей. Вы также узнали об алгоритмах классификации: логистической регрессии, обучении с помощью деревьев принятия решений, SVM и случайных лесах. Более того, научились вычислять основные статистические показатели многомерных данных и использовать алгоритм k-средних для обучения без учителя. Эти алгоритмы и методы входят в число важнейших в сфере машинного обучения, и вам придется изучить еще очень много всего, чтобы стать специалистом по машинному обучению. Подобные усилия окупятся с лихвой — такие специалисты зарабатывают в США шестизначные суммы (в этом можно легко убедиться с помощью простого поиска в Google). Студентам, желающим узнать больше о машинном обучении, я рекомендую замечательный бесплатный курс Coursera от Эндрю Энга. Вы легко можете найти материал этого курса в интернете.

В следующей главе вы научитесь работать с одним из важнейших (и самых недооцененных) инструментов эффективных программистов: регулярными выражениями. И если в этой главе материал излагался на довольно схематичном уровне (вы изучили основные идеи, а все сложные вычисления взяла на себя библиотека scikit-learn), то далее вас ждет немало технических подробностей. Так что засучите рукава и приступайте!

5

Регулярные выражения



Вы офисный работник, студент, разработчик программного обеспечения, менеджер, блогер, исследователь, автор, составитель рекламных текстов, учитель или самозанятый фрилансер? Скорее всего, вы каждый день проводите немало времени за своим компьютером. Возможность увеличить свою ежедневную выработку — даже всего на доли процента — позволит заработать (суммарно за все годы) лишние тысячи, а то и десятки тысяч долларов и освободить сотни часов времени.

В этой главе вы познакомитесь с часто недостаточно ценимой методикой, повышающей эффективность программистов при работе с текстовыми данными: регулярными выражениями. Глава продемонстрирует вам десять способов решения повседневных задач с помощью регулярных выражений, позволяющих экономить усилия, время и энергию. Очень внимательно изучите главу — ее материал оправдает ваши ожидания!

Поиск простых комбинаций символов в строковых значениях

В этом разделе вам предстоит познакомиться с регулярными выражениями, используя модуль `re` и одну из важных его функций `re.findall()`. Начнем с рассказа о нескольких простейших регулярных выражениях.

Общее описание

Регулярное выражение (regular expression, или, сокращенно, regex) формально описывает поисковый *шаблон*[1] (search pattern), на основе которого можно находить соответствующие части текста. Простой пример на рис. 5.1 демонстрирует поиск слова `Juliet` в тексте пьесы Шекспира «Ромео и Джульетта».

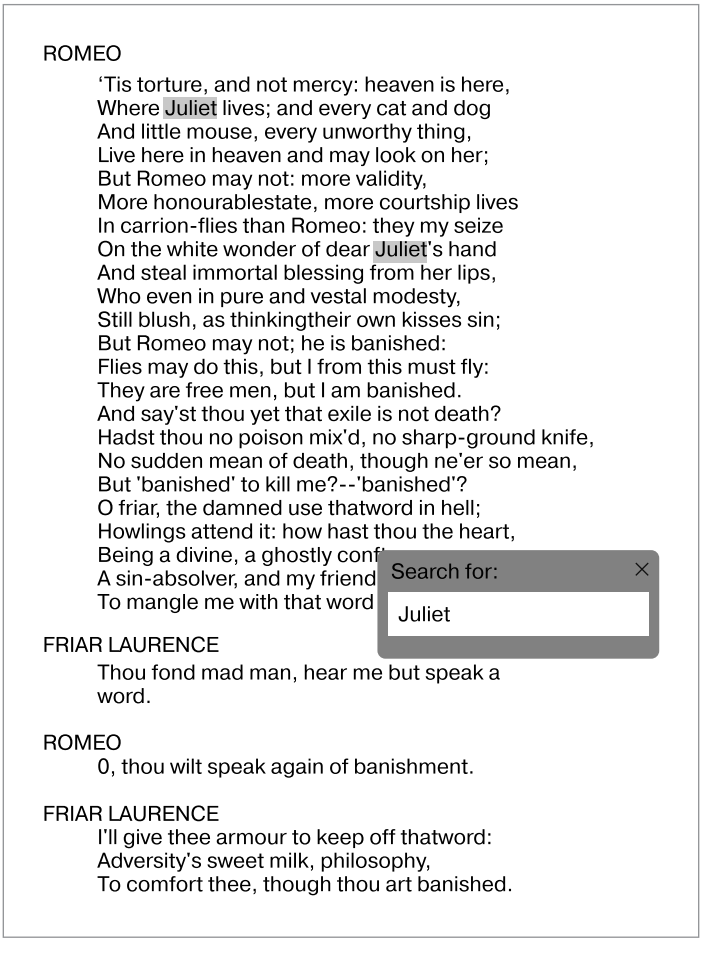


Рис. 5.1. Поиск слова Juliet в тексте пьесы Шекспира «Ромео и Джульетта»

[1] В русскоязычной литературе часто встречается также термин «паттерн».

Как показывает рис. 5.1, простейшее регулярное выражение — обычная символьная строка. Символьная строка `'Juliet'` — вполне допустимое регулярное выражение.

Возможности регулярных выражений очень широки, они подходят отнюдь не только для простого текстового поиска; в их основе лежит всего несколько основных команд. Изучите эти основные команды, и вы сможете разобраться в самых сложных регулярных выражениях и писать их самостоятельно. Мы сосредоточим свое внимание на трех важнейших командах регулярных выражений, значительно расширяющих возможности простого поиска шаблонов символов в заданном тексте.

Регулярное выражение «точка»

Во-первых, нужно разобраться, как найти произвольный символ с помощью *регулярного выражения «точка»*, то есть символа `.`. Регулярное выражение «точка» соответствует любому символу (включая пробельные). С его помощью можно указать, что неважно, какой именно символ найден, лишь бы был найден *ровно один*:

```
import re

text = '''A blockchain, originally block chain,
is a growing list of records, called blocks,
which are linked using cryptography.
'''
print(re.findall('b...k', text))
# ['block', 'block', 'block']
```

В этом примере используется метод `findall()` модуля `re`. Первый его аргумент — собственно, само регулярное выражение: мы ищем произвольную комбинацию символов, начинающуюся с символа `'b'`, за которым следуют три произвольных символа, `...`, за которыми следует символ `'k'`. Регулярному выражению `b...k` соответствует не только строка символов `'block'`, но и `'boook'`, `'b erk'` и `'bloek'`. Второй параметр метода `findall()` — текст, в котором производится поиск. Строковая переменная `text` содержит три подходящих шаблона символов, как видно из выведенных оператором `print` результатов.

Регулярное выражение «звездочка»

Во-вторых, пусть требуется найти текст, который будет начинаться и заканчиваться символом `'y'`, с произвольным количеством символов посередине.

Как это сделать? С помощью *регулярного выражения «звездочка»*, то есть символа *. В отличие от регулярного выражения «точка», регулярное выражение «звездочка» не является самостоятельным, а лишь модифицирует смысл других регулярных выражений. Рассмотрим следующий пример:

```
print(re.findall('y.*y', text))
# ['yptography']
```

Оператор «звездочка» применяется к расположенному непосредственно перед ним регулярному выражению. В этом примере задаваемый регулярным выражением шаблон начинается с `'y'`, далее следует произвольное количество символов, `.*`, за которыми снова следует символ `'y'`. Как видите, слово `'cryptography'` содержит одно вхождение этого шаблона: `'yptography'`.

Возможно, вы недоумеваете, почему этот код не находит длинную подстроку между `'originally'` и `'cryptography'`, которая тоже вроде бы соответствует шаблону регулярного выражения `y.*y`. Дело в том, что оператор «точка» соответствует любому символу, кроме символа новой строки. В переменной `text` хранится многострочное строковое значение, включающее три символа новой строки. Оператор «звездочка» можно использовать и в сочетании с любым другим регулярным выражением. Например, регулярному выражению `abc*` соответствуют строки символов `'ab'`, `'abc'`, `'abcc'` и `'abccdc'`.

Регулярное выражение «один или ни одного»

В-третьих, нужно уметь находить соответствие типа «один или ни одного» с помощью регулярного выражения, символа `?`. Подобно оператору *, знак вопроса модифицирует какое-либо другое регулярное выражение, как можно видеть из следующего примера:

```
print(re.findall('blocks?', text))
# ['block', 'block', 'blocks']
```

Регулярное выражение «один или ни одного», `?`, применяется к регулярному выражению, располагающемуся непосредственно перед ним, в данном случае к символу `s`. Регулярное выражение «один или ни одного» означает, что модифицируемый им шаблон необязателен.

В пакете `re` Python знак вопроса может использоваться и по-другому, но к регулярному выражению «один или ни одного» это отношения не имеет: знак вопроса в сочетании с оператором «звездочка», `*?`, служит для *«нежадного»* (nongreedy) поиска соответствия шаблону. Например, при указании регулярного выражения `.*?` Python ищет минимальное количество произвольных символов. И наоборот, при указании оператора «звездочка» * без знака вопроса он *«жадно»* ищет соответствие как можно большего количества символов.

Рассмотрим пример. При поиске в строке HTML-кода `'<div>hello world</div>'` по регулярному выражению `<.*>` возвращается вся строка символов `'<div>hello world</div>'`, а не только префикс `'<div>'`. Если же нужен только префикс, необходимо воспользоваться «нежадным» регулярным выражением `<.*?>`:

```
txt = '<div>hello world</div>'

print(re.findall('<.*>', txt))
# ['<div>hello world</div>']

print(re.findall('<.*?>', txt))
# ['<div>', '</div>']
```

Вооружившись знаниями этих трех инструментов — регулярных выражений «точка» `.`, «звездочка» * и «один или ни одного» `?`, — вы уже сможете разобраться в следующем однострочном решении.

Код

Роль входных данных играет строковое значение, а задача состоит в поиске с помощью «нежадного» подхода всех комбинаций символов, начинающихся с символа `'p'`, заканчивающихся символом `'r'` и включающих посередине между ними хотя бы одно вхождение символа `'e'` (и, возможно, произвольное количество других символов)!

Подобные текстовые запросы встречаются очень часто, особенно в компаниях, занимающихся обработкой текста, распознаванием речи или машинным переводом (например, компаниях, разрабатывающих поисковые системы, социальные сети и видеоплатформы). Взгляните на листинг 5.1.

Листинг 5.1. Однострочное решение для поиска («нежадного») конкретных шаблонов символов

```
## Зависимости
import re

## Данные
text = 'peter piper picked a peck of pickled peppers'

## Однострочник
result = re.findall('p.*?e.*?r', text)

## Результат
print(result)
```

Этот код выводит список всех подходящих фраз в тексте. Каких по-вашему?

Принцип работы

Поисковый запрос регулярного выражения — `p.*?e.*?r`. Рассмотрим его по частям. Мы ищем фразу, начинающуюся с символа `'p'` и заканчивающуюся символом `'r'`. Кроме того, между ними должно встречаться хотя бы одно вхождение символа `'e'`. Кроме того, допускается произвольное количество символов (как пробельных, так и прочих). Поиск производится «нежадным» образом, с помощью `.*?`, поэтому Python будет искать минимальное количество произвольных символов. Вот результат:

```
## Результат
print(result)
# ['peter', 'piper', 'picked a peck of pickled pepper']
```

Сравните этот результат с получаемым при использовании «жадного» регулярного выражения `p.*e.*r`:

```
result = re.findall('p.*e.*r', text)
print(result)
# ['peter piper picked a peck of pickled pepper']
```

Первый «жадный» оператор «звездочка» * захватывает практически всю строку до конца.

Создание вашего первого веб-скрапера с помощью регулярных выражений

Из предыдущего раздела вы узнали о самом эффективном способе поиска произвольных шаблонов текста в строковых значениях: регулярных

выражениях. Этот раздел еще больше вдохновит вас на использование регулярных выражений и укрепит ваши знания с помощью практического примера.

Общее описание

Допустим, вы разработчик-фрилансер. Вашему заказчику — финансово-технологическому стартапу — постоянно нужны последние новости в сфере криптовалют. Они наняли вас для написания веб-скрапера, который бы регулярно извлекал исходный HTML-код из новостных сайтов и искал в нем слова, начинающиеся с `'crypto'` (например, `'cryptocurrency'`, `'crypto-bot'`, `'crypto-crash'` и т. д.).

Первая наша попытка — следующий фрагмент кода:

```
import urllib.request

search_phrase = 'crypto'

with urllib.request.urlopen('https://www.wired.com/') as response:
   html = response.read().decode("utf8") # convert to string
   first_pos = html.find(search_phrase)
   print(html[first_pos-10:first_pos+10])
```

Метод `urlopen()` (из модуля `urllib.request`) извлекает исходный HTML-код по указанному URL. Поскольку результат представляет собой байтовый массив, необходимо сначала преобразовать его в строковое значение с помощью метода `decode()`. А затем воспользоваться строковым методом `find()` для поиска позиции первого вхождения искомой строки. С помощью среза (см. главу 2) мы извлекаем подстроку, содержащую непосредственное окружение искомого места. В результате получаем следующее строковое значение:

```
# ,r=window.crypto||wi
```

Ой, выглядит не очень. Как оказалось, поисковая фраза двусмысленна — большинство слов, содержащих `'crypto'`, с *криптовалютами* никак не связаны. Наш веб-скрапер генерирует *ложноположительные результаты* (находит строковые значения, которые мы вовсе не хотели находить). Как же исправить эту ситуацию?

К счастью, вы как раз читаете эту книгу по Python, так что ответ очевиден: регулярные выражения! Возникает идея: исключить ложноположительные

результаты за счет поиска только тех вхождений, в которых за словом `'crypto'` следует до 30 произвольных символов, за которыми следует слово `'coin'`. Грубо говоря, поисковый запрос выглядит так: `crypto + <до 30 произвольных символов> + coin`. Рассмотрим следующие два примера:

- `'crypto-bot that is trading Bitcoin'` — да;
- `'cryptographic encryption methods that can be cracked easily with quantum computers'` — нет.

Итак, проблема состоит в том, что регулярное выражение должно допускать до 30 произвольных символов между двумя строками символов. Как решить эту выходящую за пределы простого поиска строк задачу? Перебрать все комбинации символов не получится — их количество практически бесконечно. Например, нашему поисковому шаблону должны соответствовать все следующие строковые значения: `'cryptoxxxcoin'`, `'crypto coin'`, `'crypto bitcoin'`, `'crypto is a currency. Bitcoin'`, а также остальные сочетания до 30 символов между двумя строками. Даже если в алфавите всего 26 символов, количество теоретически удовлетворяющих нашему требованию строк символов превышает[1] 26^{30} = 2 813 198 901 284 745 919 258 621 029 615 971 520 741 376. Далее мы расскажем вам, как искать в тексте задаваемый регулярным выражением шаблон, которому соответствует большое количество возможных комбинаций символов.

Код

В этом коде мы ищем в заданном строковом значении вхождения, в которых за строкой символов `'crypto'` следует до 30 произвольных символов, за которыми следует слово `'coin'`. Посмотрим сначала на листинг 5.2, а затем обсудим, как этот код решает поставленную задачу.

Листинг 5.2. Однострочное решение для поиска фрагментов текста вида crypto(какой-то текст)coin

```
## Зависимости
import re

## Данные
text_1 = "crypto-bot that is trading Bitcoin and other currencies"
```

[1] Поскольку помимо символов алфавита в них могут входить различные пробельные символы и знаки препинания. Впрочем, на деле далеко не все сочетания символов осмысленны, так что реальное количество сочетаний намного меньше.

```
text_2 = "cryptographic encryption methods that can be cracked easily with
quantum computers"

## Однострочник
pattern = re.compile("crypto(.{1,30})coin")

## Результат
print(pattern.match(text_1))
print(pattern.match(text_2))
```

Данный код производит поиск в двух строковых переменных, `text_1` и `text_2`. Соответствуют ли они поисковому запросу (шаблону)?

Принцип работы

Во-первых, мы импортируем стандартный модуль для работы с регулярными выражениями в Python, `re`. Самое интересное происходит в однострочнике, где компилируется поисковый запрос `crypto(.{1,30})coin`. С помощью этого запроса мы и будем производить поиск в различных строковых значениях. В нем используются специальные символы регулярных выражений. Прочитайте их список внимательно, от начала до конца, и вы поймете смысл шаблона из листинга 5.2:

- шаблон `()` предназначен для поиска соответствия указанному внутри него регулярному выражению;
- шаблону `.` соответствует любой произвольный символ;
- шаблону `{1,30}` соответствует от 1 до 30 вхождений *предыдущего* регулярного выражения;
- шаблону `({1,30})` соответствует строковое значение, включающее от 1 до 30 произвольных символов;
- шаблону `crypto(.{1,30})coin` соответствует строковое значение, состоящее из трех частей: слова `'crypto'`, последовательности, включающей от 1 до 30 символов, и следующего за ними слова `'coin'`.

Мы упомянули, что шаблон *скомпилирован*, поскольку Python создает объект шаблона, который можно повторно применять в разных местах — подобно тому, как скомпилированную программу можно использовать многократно. Теперь можно вызвать функцию `match()` нашего скомпилированного шаблона, и будет произведен поиск по тексту. В результате получим следующее:

```
## Результат
print(pattern.match(text_1))
# <re.Match object; span=(0, 34), match='crypto-bot that is trading Bitcoin'>

print(pattern.match(text_2))
# None
```

Строковая переменная `text_1` соответствует шаблону (что видно из полученного объекта `Match`), а `text_2` — нет (что видно из результата `None`). И хотя текстовое представление первого объекта выглядит не слишком изящно, но ясно указывает, что заданная строка `'crypto-bot that is trading Bitcoin'` соответствует регулярному выражению.

Анализ гиперссылок HTML-документов

В предыдущем разделе вы узнали, как искать в строке большое количество комбинаций символов с помощью шаблона регулярного выражения `.{x,y}`. В этом разделе мы продвинемся еще на шаг и познакомимся со множеством других регулярных выражений.

Общее описание

Чем больше регулярных выражений вы знаете, тем быстрее и лаконичнее сможете решать реальные задачи. Какие же регулярные выражения наиболее важны? Изучите следующий список внимательно, поскольку все эти выражения будут применяться в данной главе. Можете считать те из них, которые уже видели выше, маленьким упражнением на повторение.

- Регулярному выражению `.` соответствует любой символ.
- Регулярному выражению «звездочка» *<шаблон>*`*` соответствует произвольное количество вхождений, в том числе нулевое, соответствующих *<шаблон>*.
- Регулярному выражению «по крайней мере один» *<шаблон>*`+` может соответствовать произвольное количество *<шаблон>*, но не менее одного.
- Регулярному выражению «один или ни одного» *<шаблон>*`?` соответствует один экземпляр *<шаблон>* или ни одного.
- «Нежадному» регулярному выражению «звездочка» *<шаблон>*`*?` соответствует как можно меньшее количество символов, необходимых для того, чтобы удовлетворить регулярному выражению в целом.

- Регулярному выражению <*шаблон*>{m} соответствует ровно m копий <*шаблон*>.
- Регулярному выражению <*шаблон*>{m,n} соответствует от m до n копий <*шаблон*>.
- Регулярному выражению <*шаблон_1*>|<*шаблон_2*> соответствует как <*шаблон_1*>, так и <*шаблон_2*>.
- Регулярному выражению <*шаблон_1*><*шаблон_2*> соответствует <*шаблон_1*>, за которым следует <*шаблон_2*>.
- Регулярному выражению (<*шаблон*>) соответствует <*шаблон*>. Скобки служат для группировки регулярных выражений с целью контроля порядка выполнения (например, регулярное выражение (<*шаблон_1*><*шаблон_2*>)|<*шаблон_3*> отличается от <*шаблон_1*>(<*шаблон_2*>|<*шаблон_3*>)). Скобочные группы регулярного выражения также служат для создания групп соответствий, как вы увидите далее в этом разделе.

Рассмотрим короткий пример. Пускай мы создали регулярное выражение b?(.a)*. Каким комбинациям символов оно соответствует? Всем, начинающимся с символа b (он может и отсутствовать) и содержащим произвольное количество последовательностей из пар символов, заканчивающихся 'a'. Таким образом, ему соответствуют все следующие строковые значения: 'bcacaca', 'cadaea', '' (пустая строка) и 'aaaaaa'.

Прежде чем углубиться в следующий однострочник, вкратце поговорим о том, когда использовать ту или иную *функцию регулярных выражений*. Три важнейшие функции регулярных выражений — re.match(), re.search() и re.findall(). Две из них вы уже видели, но посмотрим на них внимательнее в следующем примере:

```
import re

text = '''
"One can never have enough socks", said Dumbledore.
"Another Christmas has come and gone and I didn't
get a single pair. People will insist on giving me books."
Christmas Quote
'''

regex = 'Christ.*'

print(re.match(regex, text))
```

```
# None

print(re.search(regex, text))
# <re.Match object; span=(62, 102), match="Christmas has come and gone
# and I didn't">

print(re.findall(regex, text))
# ["Christmas has come and gone and I didn't", 'Christmas Quote']
```

Все три эти функции принимают на входе регулярное выражение и строковое значение, в котором производится поиск. Функции `match()` и `search()` возвращают объект `Match` (или `None`, если соответствия регулярному выражению не нашлось). В объекте `Match` хранится позиция найденного соответствия и дополнительная метаинформация. Функция `match()` не нашла соответствия регулярному выражению в нашей строке (вернула `None`). Почему? А потому, что эта функция ищет шаблон только *с начала* строки. Функция же `search()` ищет первое вхождение шаблона *в любом* месте строки. А потому находит соответствие `"Christmas has come and gone and I didn't"`.

Результаты работы функции `findall()` наиболее интуитивно понятны, но и наименее удобны для дальнейшей обработки. Результаты работы функции `findall()` представляет собой последовательность строковых значений, а не объект `Match`, поэтому точной информации о месте вхождения они не дают. Тем не менее функция `findall()` также бывает полезна: в отличие от методов `match()` и `search()`, функция `findall()` извлекает все подходящие шаблоны символов. Это удобно, если требуется найти количество вхождений слова в тексте (например, строки символов `'Juliet'` в тексте «Ромео и Джульетты» или строки символов `'crypto'` в статье о криптовалютах).

Код

Представьте, что начальник попросил вас создать маленький веб-бот для сканирования веб-страниц и проверки того, содержится ли на них ссылка на домен finxter.com. Кроме того, оно попросило вас убедиться, содержат ли описания гиперссылок строку символов `'test'` или `'puzzle'`. В HTML гиперссылки заключены в теги `<a></a>`. Сама гиперссылка задается в виде значения атрибута `href`. Поэтому точная формулировка задачи (отраженной в листинге 5.3) звучит так: по заданному строковому значению найти все гиперссылки, указывающие на домен finxter.com и содержащие строки символов `'test'` или `'puzzle'` в описании ссылки.

Листинг 5.3. Однострочное решение для анализа ссылок на веб-страницах

```
## Зависимости
import re

## Данные
page = '''
<!DOCTYPE html>
<html>
<body>

<h1>My Programming Links</h1>
<a href="https://app.finxter.com/">test your Python skills</a>
<a href="https://blog.finxter.com/recursion/">Learn recursion</a>
<a href="https://nostarch.com/">Great books from NoStarchPress</a>
<a href="http://finxter.com/">Solve more Python puzzles</a>

</body>
</html>
'''

## Однострочник
practice_tests = re.findall("(<a.*?finxter.*?(test|puzzle).*?>)", page)

## Результат
print(practice_tests)
```

Этот код находит два вхождения, соответствующих регулярному выражению. Какие?

Принцип работы

Наши данные состоят из простой веб-страницы (в виде многострочного строкового значения) на HTML, содержащей набор гиперссылок (теги `<a href="">`*текст ссылки*`</a>`). В нашем однострочном решении для поиска соответствий регулярному выражению `(<a.*?finxter.*?(test|puzzle).*?>)` используется функция `re.findall()`. Благодаря этому регулярное выражение возвращает все вхождения в тегах `<a. . .>` со следующими ограничениями.

После открывающего тега мы ищем произвольное количество символов («нежадным» образом, чтобы регулярное выражение не захватывало несколько тегов HTML), за которым следует строка символов `'finxter'`. Далее ищем произвольное количество символов («нежадным» образом), за которым следует одно вхождение строки символов `'test'` либо строки символов `'puzzle'`. И снова ищем произвольное количество символов («нежадным»

образом), за которым следует закрывающий тег. Таким образом мы находим все теги гиперссылок, содержащие интересующие нас строки символов. Обратите внимание, что данному регулярному выражению соответствуют и теги, в которых строка символов `'test'` или `'puzzle'` встречается внутри самой ссылки. Учтите также, что мы используем только «нежадные» операторы «звездочка» `'.*?'`, чтобы всегда искать лишь минимальные соответствующие строки символов, вместо, например, очень длинной строки символов, заключенной во множество вложенных тегов.

Результат нашего однострочника:

```
## Результат
print(practice_tests)
# [('<a href="https://app.finxter.com/">test your Python skills</a>', 'test'),
#  ('<a href="http://finxter.com/">Solve more Python puzzles</a>', 'puzzle')]
```

Нашему регулярному выражению соответствуют две гиперссылки — результат однострочника представляет собой список из двух элементов. Однако каждый из этих элементов не просто строковое значение, а кортеж строковых значений, что отличается от результатов `findall()`, обсуждавшихся в предыдущих фрагментах кода. В чем тут дело? Возвращаемый тип — список кортежей, по одному значению кортежа для каждой заключенной в скобки *группы*. Например, скобочная нотация служит для создания группы в регулярном выражении `(test|puzzle)`. При использовании скобочных групп в регулярном выражении функция `re.findall()` добавляет по одному значению кортежа для каждой найденной группы. Значение кортежа представляет собой подстроку, соответствующую этой конкретной группе. Например, в нашем случае группе `(test|puzzle)` удовлетворяет подстрока `'puzzle'`. Посмотрим внимательнее на скобочные группы, чтобы лучше разобраться с этим понятием.

Выделение сумм в долларах из строковых значений

Этот однострочник демонстрирует еще одно применение регулярных выражений на практике. В данном случае представьте себя финансовым аналитиком. Ваша компания планирует приобрести другую, и вам поручили прочитать ее финансовые отчеты. Особенно вас интересуют все суммы в долларах. Конечно, вы можете просмотреть весь документ вручную, но это утомительная работа и вы не хотели бы тратить на нее самое продуктивное

время дня. Поэтому вы решили написать небольшой сценарий на Python. Но как лучше всего это сделать?

Общее описание

К счастью, вы уже прочли мое руководство по регулярным выражениям, поэтому вместо того чтобы тратить огромное количество времени на написание собственного, очень большого и чреватого ошибками средства синтаксического разбора на Python, вы решили воспользоваться более аккуратной реализацией на основе регулярных выражений — мудрое решение. Но прежде чем углубиться в решение поставленной задачи, обсудим еще три понятия, связанные с регулярными выражениями.

Во-первых, рано или поздно вы захотите найти какой-либо специальный символ, используемый в этом качестве и языком регулярных выражений. В таком случае необходимо задействовать префикс `\` для *экранирования* этого специального символа. Например, для поиска символа правой скобки `'('`, используемого для группировки регулярных выражений, необходимо экранировать его следующим образом: `\(`. При этом символ `'('` теряет особый смысл в регулярных выражениях.

Во-вторых, с помощью квадратных скобок `[ ]` можно описывать диапазоны конкретных символов. Например, регулярному выражению `[0-9]` соответствует любой из следующих символов: `'0'`, `'1'`, `'2'`, `. . .`, `'9'`. Еще один пример — регулярное выражение `[a-e]`, которому соответствует любой из следующих символов: `'a'`, `'b'`, `'c'`, `'d'`, `'e'`.

В-третьих, как мы обсуждали в посвященном предыдущему однострочнику разделе, регулярное выражение «скобки» `(<pattern>)` задает *группу*. Одна или несколько групп могут быть в любом регулярном выражении. При использовании функции `re.findall()` для включающего группы регулярного выражения в виде кортежа строковых значений будут возвращены только совпадающие группы, а не вся совпадающая строка. Например, регулярному выражению `hello(world)`, вызванному для строки `'helloworld'`, соответствует вся строка, но возвращена будет только соответствующая ему группа `world`. С другой стороны, при использовании двух вложенных групп в регулярном выражении `(hello(world))` результат функции `re.findall()` будет представлять собой кортеж всех подходящих групп `('helloworld', 'world')`. Внимательно изучите следующий код, чтобы лучше разобраться с понятием вложенных групп:

```
string = 'helloworld'

regex_1 = 'hello(world)'
regex_2 = '(hello(world))'

res_1 = re.findall(regex_1, string)
res_2 = re.findall(regex_2, string)

print(res_1)
# ['world']
print(res_2)
# [('helloworld', 'world')]
```

Теперь вы уже знаете все необходимое для понимания следующего фрагмента кода.

Код

Напомню, что мы хотим посмотреть на все денежные суммы из заданного отчета компании. А именно, нам нужно решить следующую задачу: получить по заданной строке список всех вхождений сумм в долларах, возможно, с десятичными значениями. Например, следующих строк символов: $10, $10. и $10.00021. Как эффективно решить эту задачу с помощью одной строки кода? Взгляните на листинг 5.4.

Листинг 5.4. Однострочное решение для поиска в тексте всех сумм в долларах

```
## Зависимости
import re

## Данные
report = '''
If you invested $1 in the year 1801, you would have $18087791.41 today.
This is a 7.967% return on investment.
But if you invested only $0.25 in 1801, you would end up with $4521947.8525.
'''

## Однострочник
dollars = [x[0] for x in re.findall('(\$[0-9]+(\.[0-9]*)?)', report)]

## Результат
print(dollars)
```

Попробуйте догадаться, каковы будут результаты выполнения этого фрагмента кода.

Принцип работы

Отчет содержит четыре суммы в долларах в различных форматах. Необходимо разработать регулярное выражение, которому удовлетворяли бы они все. Мы разработали регулярное выражение `(\$[0-9]+(\.[0-9]*)?)`, которому удовлетворяют следующие комбинации символов. Во-первых, знак доллара `$` (мы его экранировали, поскольку он представляет собой специальный символ в регулярных выражениях). Во-вторых, число из произвольного количества цифр от 0 до 9 (но не менее одной). В-третьих, произвольное количество десятичных значений после (экранированного) символа точки `'.'` (необязательного, как указывает регулярное выражение типа «один или ни одного» `?`).

Более того, мы воспользовались списковым включением для извлечения только первого значения кортежа из всех трех найденных соответствий. Опять же, по умолчанию функция `re.findall()` возвращает список кортежей, по одному кортежу для каждого найденного соответствия и по одному значению кортежа для каждой группы в этом найденном соответствии:

```
[('$1', ''), ('$18087791.41', '.41'), ('$0.25', '.25'), ('$4521947.8525', '.8525')]
```

Нас интересует только общая группа — первое значение в кортеже. Остальные значения мы отфильтровываем с помощью спискового включения и получаем следующий результат:

```
## Результат
print(dollars)
# ['$1 ', '$18087791.41', '$0.25', '$4521947.8525']
```

Стоит опять отметить, насколько сложна и чревата ошибками была бы реализация даже простейшего синтаксического анализатора, не будь замечательных возможностей регулярных выражений!

Поиск небезопасных HTTP URL

Данный однострочник демонстрирует, как решить одну из тех маленьких, но требующих много времени задач, с которыми так часто сталкиваются веб-разработчики. Допустим, вы ведете блог по программированию и только что перевели свой сайт с небезопасного протокола `http` на (более) безопасный

протокол `https`. Однако ссылки в ваших старых статьях по-прежнему указывают на старые URL. Как же найти все эти ссылки на старые URL?

Общее описание

В предыдущем разделе вы научились с помощью нотации с квадратными скобками задавать произвольные диапазоны символов. Например, регулярному выражению `[0-9]` соответствует любое число из одной цифры от 0 до 9. Однако потенциальные возможности нотации с квадратными скобками намного шире. Можно точно задавать символы, на соответствие (или несоответствие) которым необходимо проверить, путем указания в квадратных скобках произвольных сочетаний символов. Например, регулярному выражению `[0-3a-c]+` соответствуют строки символов `'01110'` и `'01c22a'`, но не строка `'443'` или `'00cd'`. Можно также задавать фиксированный набор символов, которые *не* должны содержаться в найденном шаблоне, с помощью символа `^`: регулярному выражению `[^0-3a-c]+` соответствуют строковые значения`'4444d'` и `'Python'` и не соответствуют строки символов `'001'` и `'01c22a'`.

Код

Входные данные здесь представляют собой многострочное строковое значение, в котором нам нужно найти все вхождения допустимых URL, начинающихся с префикса `http://`. Однако рассматривать недопустимые URL без домена верхнего уровня (в найденном URL должна содержаться хотя бы одна `.`) не следует. Взгляните на листинг 5.5.

Листинг 5.5. Однострочное решение для поиска правильных URL вида http://

```
## Зависимости
import re

## Данные
article = '''
The algorithm has important practical applications
http://blog.finxter.com/applications/
in many basic data structures such as sets, trees,
dictionaries, bags, bag trees, bag dictionaries,
hash sets, https://blog.finxter.com/sets-in-python/
hash tables, maps, and arrays. http://blog.finxter.com/
http://not-a-valid-url
```

```
http:/bla.ba.com
http://bo.bo.bo.bo.bo.bo/
http://bo.bo.bo.bo.bo.bo/333483--33343-/
'''

## Однострочник
stale_links = re.findall('http://[a-z0-9_\-.]+\.[a-z0-9_\-/]+', article)

## Результат
print(stale_links)
```

Опять же, попытайтесь догадаться, какой результат вернет этот код, прежде чем смотреть на приведенный ниже результат.

Принцип работы

В данном регулярном выражении мы анализируем многострочное строковое значение (возможно, старое сообщение из блога) в поисках всех URL, начинающихся со строкового префикса `http://`. На входе это регулярное выражение ожидает положительное количество символов в нижнем регистре, чисел, знаков подчеркивания, дефисов или точек (`[a-z0-9_\-\.]+`). Обратите внимание, что мы экранируем дефис (`\-`), поскольку в регулярных выражениях с его помощью задаются диапазоны внутри квадратных скобок. Аналогично необходимо экранировать точку (`\.`), поскольку мы хотим найти именно точку, а не произвольный символ. В результате получаем следующее:

```
## Результаты
print(stale_links)
# ['http://blog.finxter.com/applications/',
#  'http://blog.finxter.com/',
#  'http://bo.bo.bo.bo.bo.bo/',
#  'http://bo.bo.bo.bo.bo.bo/333483--33343-/']
```

Похоже, необходимо перенести на более защищенный протокол HTTPS четыре правильных URL.

Вы уже овладели основными возможностями регулярных выражений. Но действительно глубокого понимания можно достичь, лишь применяя их на практике и изучая большое количество примеров, и регулярные выражения не исключение. Рассмотрим еще несколько примеров того, как регулярные выражения могут облегчить жизнь программиста.

Проверка формата времени во вводимых пользователем данных, часть 1

Научимся проверять правильность формата вводимых пользователем данных. Пусть вы пишете веб-приложение, вычисляющее медицинские показатели на основе длительности сна пользователей. Они вводят время отхода ко сну и время пробуждения. Пример времени в правильном формате — `12:45`, но из-за спама от веб-ботов большое количество испорченных данных приводит к излишней вычислительной нагрузке на сервер. Для решения этой проблемы вы хотите написать средство проверки формата времени, которое бы определяло, имеет ли смысл дальнейшая обработка ваших входных данных приложением, расположенным в прикладной части. Написание такого кода с помощью регулярных выражений занимает всего несколько минут.

Общее описание

В предыдущих нескольких разделах вы узнали про функции `re.match()`, `re.search()` и `re.findall()`. Они не единственные функции для работы с регулярными выражениями. В этом разделе мы воспользуемся функцией `re.fullmatch(`*`регулярное_выражение`*`, `*`строковое_значение`*`)`, проверяющей, соответствует ли регулярному выражению полное *`строковое_значение`*, как можно предположить из ее названия.

Кроме того, мы воспользуемся синтаксисом `<`*`шаблон`*`>{m,n}` регулярных выражений, применяемым для поиска строки символов, включающей от `m` до `n` копий *`шаблон`*, но не более и не менее данного количества. Обратите внимание, что при этом производится попытка найти строку символов с максимальным количеством вхождений `<`*`шаблон`*`>`. Ниже представлен пример:

```
import re

print(re.findall('x{3,5}y', 'xy'))
# []
print(re.findall('x{3,5}y', 'xxxy'))
# ['xxxy']
print(re.findall('x{3,5}y', 'xxxxxy'))
# ['xxxxxy']
print(re.findall('x{3,5}y', 'xxxxxxy'))
# ['xxxxxy']
```

При использовании нотации с фигурными скобками код не ищет соответствий подстрок, включающих менее трех и более пяти символов `'x'`.

Код

Наша задача — написать функцию `input_ok`, принимающую строковый аргумент и проверяющий его на соответствие формату (времени) `XX:XX`, где `X` — число от 0 до 9 (листинг 5.6). Учтите, что пока мы не отбрасываем семантически неправильные форматы времени наподобие 12:86. Этой более сложной задачей мы займемся в следующем разделе.

Листинг 5.6. Однострочное решение для проверки соответствия введенных пользователем данных общему формату времени XX:XX

```
## Зависимости
import re

## Данные
inputs = ['18:29', '23:55', '123', 'ab:de', '18:299', '99:99']

## Однострочник
input_ok = lambda x: re.fullmatch('[0-9]{2}:[0-9]{2}', x) != None

## Результат
for x in inputs:
    print(input_ok(x))
```

Прежде чем продолжить чтение, попробуйте определить результаты шести вызовов функций в этом коде.

Принцип работы

Наши данные состоят из шести входных строковых значений, получаемых клиентской частью веб-приложения. Правильного ли они формата? Чтобы проверить, мы создаем функцию `input_ok` с помощью лямбда-выражения с одним входным аргументом `x` и булевым выходным значением. Далее с помощью функции `fullmatch(`*`регулярное_выражение`*`, x)` мы пытаемся установить соответствие входному аргументу `x` нашему регулярному выражению для формата времени. Если это не удается, то результат принимает значение `None`, а булев выходной результат функции — `False`. В противном случае булев результат функции будет `True`.

Само регулярное выражение очень простое: `[0-9]{2}:[0-9]{2}`. Ему соответствуют два числа от 0 до 9, за которыми следует двоеточие `:`, за которым следуют еще два числа от 0 до 9. Таким образом, результат листинга 5.6 выглядит следующим образом:

```
## Результат
for x in inputs:
    print(input_ok(x))

'''
True
True
False
False
False
True
'''
```

Функция `input_ok` корректно находит правильно отформатированные значения времени в `inputs`. В этом однострочнике показано, как часто встречающиеся на практике задачи, которые в противном случае потребовали бы бóльших усилий и многих строк кода, можно выполнить за несколько секунд с помощью правильного набора инструментов.

Проверка формата времени во вводимых пользователем данных, часть 2

В этом разделе мы еще больше углубимся в вопросы проверки формата времени во вводимых пользователями данных, чтобы решить проблему, упомянутую в предыдущем разделе: некорректные данные, введенные пользователем, например `99:99` не должны считаться допустимыми при поиске соответствий.

Общее описание

Одна из удобных стратегий решения задач — иерархическая. Для начала сведите задачу к простейшему виду и решите этот более простой ее вариант. Далее уточните решение, чтобы оно соответствовало конкретной (и более сложной) задаче, стоящей перед вами. В данном разделе мы существенно совершенствуем решение из предыдущего раздела: исключаем некорректные варианты вводимого времени наподобие `99:99` или `28:66`. Поэтому наша

задача становится более конкретной (и сложной), но мы можем повторно использовать отдельные части нашего старого решения.

Код

Наша задача — написать функцию `input_ok`, принимающую строковый аргумент и проверяющий его на соответствие формату (времени) `XX:XX`, где `X` — число от 0 до 9 (листинг 5.7). Кроме того, должен соблюдаться правильный формат времени во входных данных — в 24-часовом диапазоне от 00:00 до 23:59.

Листинг 5.7. Однострочное решение для проверки соблюдения общего формата времени XX:XX и 24-часового диапазона во введенных пользователем данных

```
## Зависимости
import re

## Данные
inputs = ['18:29', '23:55', '123', 'ab:de', '18:299', '99:99']

## Однострочник
input_ok = lambda x: re.fullmatch('([01][0-9]|2[0-3]):[0-5][0-9]', x) != None

## Результат
for x in inputs:
    print(input_ok(x))
```

Этот код выводит шесть строк. Каких?

Принцип работы

Как уже упоминалось во введении к этому разделу, мы можем повторно использовать части предыдущего однострочника для упрощения решения нашей задачи. Код остается тем же самым, меняется только регулярное выражение. `([01][0-9]|2[0-3]):[0-5][0-9]`. Первая часть `([01][0-9]|2[0-3])` представляет собой группу, которой соответствуют все возможные часы в сутках. Оператор логического ИЛИ `|` здесь позволяет учесть часы от 00 до 19, с одной стороны, и от 20 до 23 — с другой. Второй части, `[0-5][0-9]`, соответствуют минуты от 00 до 59. В результате выводится следующее:

```
## Результат
for x in inputs:
    print(input_ok(x))
```

```
'''
True
True
False
False
False
False
'''
```

Обратите внимание, что шестая из строк результата демонстрирует: время `99:99` более не считается допустимыми входными данными. Этот однострочник показывает, как с помощью регулярных выражений проверять соответствие вводимых пользователем данных семантическим требованиям приложения.

Поиск удвоенных символов в строковых значениях

Этот однострочник откроет для вас замечательную возможность регулярных выражений: повторное использование частей, для которых уже производился поиск соответствий в том же регулярном выражении. Такое расширение возможностей дает возможность решать новый спектр задач, включая поиск строковых значений с удвоенными символами.

Общее описание

На этот раз представьте, что вы как исследователь в сфере вычислительной лингвистики анализируете изменения частоты использования определенных слов с течением времени. Вы классифицируете слова и изучаете частоту их использования в различных напечатанных книгах. Ваш научный руководитель дает задание выяснить наличие тенденции к повышению частоты использования удвоенных символов в словах. Например, слово `'hello'` содержит удвоенную букву `'l'`, а слово `'spoon'` содержит удвоенную букву `'o'`. Слово же `'mama'` не относится к словам с удвоенной буквой `'a'`.

Наивное решение этой задачи — перечислить все возможные варианты удвоения букв `'aa'`, `'bb'`, `'cc'`, `'dd'`, `...` , `'zz'` и объединить их в одном регулярном выражении. Это решение утомительно для реализации и плохо

поддается обобщению. Что, если ваш научный руководитель передумает и попросит теперь искать удвоенные буквы, которые разделяет один символ (например, такие как в `'mama'`)?

Никаких проблем: существует простое, аккуратное и эффективное решение для тех, кто умеет обращаться с поименованными группами в регулярных выражениях. Вы уже встречали скобочные группы вида `(...)`. Как ясно из названия, *поименованная группа* (named group) — это просто группа, у которой есть название. Например, описать поименованную группу для шаблона `...` с названием `name` можно с помощью синтаксиса `(?P<name>...)`. Описанную поименованную группу можно использовать где угодно в регулярном выражении, прибегнув к синтаксису `(?P=name)`. Рассмотрим следующий пример:

```
import re

pattern = '(?P<quote>[\'"]).*(?P=quote)'
text = 'She said "hi"'
print(re.search(pattern, text))
# <re.Match object; span=(9, 13), match='"hi"'>
```

В этом коде мы ищем подстроки, заключенные в одинарные или двойные кавычки. Для этого сначала ищем соответствие для открывающей кавычки с помощью регулярного выражения `[\'"]` (одинарную кавычку мы экранируем, чтобы Python ошибочно не счел ее символом окончания строкового значения). Далее с помощью той же группы ищем аналогичную закрывающую кавычку (одинарную или двойную).

Прежде чем заняться кодом, отметим, что можно искать соответствие произвольных пробельных символов с помощью регулярного выражения `\s`. Можно также искать символы, не входящие во множество `Y`, посредством синтаксиса `[^Y]`. Вот и все, что нам нужно для решения поставленной задачи.

Код

Рассмотрим задачу, изложенную в листинге 5.8: найти в заданном тексте все слова с удвоенными символами. Под *словом* в данном случае понимается произвольная последовательность непробельных символов, отделенных произвольным количеством пробельных символов.

Листинг 5.8. Однострочное решение для поиска всех слов с удвоенными символами

```
## Зависимости
import re

## Данные
text = '''
It was a bright cold day in April, and the clocks were
striking thirteen. Winston Smith, his chin nuzzled into
his breast in an effort to escape the vile wind, slipped
quickly through the glass doors of Victory Mansions,
though not quickly enough to prevent a swirl of gritty
dust from entering along with him.
-- George Orwell, 1984
'''

## Однострочник
duplicates = re.findall('([^\s]*(?P<x>[^\s])(?P=x)[^\s]*)', text)

## Результат
print(duplicates)
```

Какие же слова с удвоенными символами найдет этот код?

Принцип работы

Регулярное выражение `(?P<x>[^\s])` задает новую группу с названием `x`, которая состоит из одного произвольного непробельного символа. Сразу за поименованной группой `x` следует регулярное выражение `(?P=x)`, которому соответствует тот же символ, что и группе `x`. Мы нашли наши удвоенные символы! Однако задача состояла в поиске не просто удвоенных символов, а слов с удвоенными символами. Поэтому мы захватываем еще и произвольное число непробельных символов `[^\s]*` до и после удвоенного символа.

Стоит отметить, что это решение не вполне корректно обрабатывает слова, в которых встречается несколько удвоений символов. Оставим усовершенствование кода читателю в качестве упражнения.

Результаты работы листинга 5.8 выглядят следующим образом:

```
## Результаты
print(duplicates)
'''
[('thirteen.', 'e'), ('nuzzled', 'z'), ('effort', 'f'),
('slipped', 'p'), ('glass', 's'), ('doors', 'o'),
('gritty', 't'), ('--', '-'), ('Orwell,', 'l')]
'''
```

Это регулярное выражение находит в тексте все слова с удвоенными символами. Отметим, что регулярное выражение из листинга 5.8 включает две группы, так что каждый возвращаемый функцией `re.findall()` элемент состоит из кортежа, части которого соответствуют этим группам. Вы уже видели подобное поведение в предыдущих разделах.

В этом разделе вы добавили в свой набор инструментов регулярных выражений еще один замечательный инструмент: поименованные группы. В сочетании с двумя более незначительными возможностями поиска соответствий произвольным пробельным символам с помощью `\s` и описания набора символов, которые не должны встречаться на данном месте в шаблоне, с помощью оператора `[^...]` вы существенно приблизились к овладению регулярными выражениями Python в совершенстве.

Поиск повторов слов

В предыдущем разделе мы рассказали вам о поименованных группах. Задача этого раздела — продемонстрировать продвинутые способы использования данной возможности.

Общее описание

Работая в последние несколько лет исследователем, бóльшую часть времени я тратил на написание, чтение и редактуру научных статей. Редактируя мои статьи, один из моих коллег часто жаловался, что я постоянно (и очень близко в тексте) использую одни и те же слова. Утилита для проверки написанного программным образом мне совсем бы не помешала.

Код

Пусть дано строковое значение, состоящее из слов в нижнем регистре, разделенных пробелами, без каких-либо специальных символов. Необходимо найти подстроку, в которой первое и последнее слова одинаковы (повторяются), причем их разделяет не более десяти слов (листинг 5.9).

Листинг 5.9. Однострочное решение для поиска повторов слов

```
## Зависимости
import re

## Данные
```

```
text = 'if you use words too often words become used'

## Однострочник
style_problems = re.search('\s(?P<x>[a-z]+)\s+([a-z]+\s+){0,10}(?P=x)\s',
' ' + text + ' ')

## Результат
print(style_problems)
```

Действительно ли этот код находит повторы слов?

Принцип работы

Опять же, мы предполагаем, что `text` состоит только из разделенных пробелами слов в нижнем регистре. И ищем требуемое в `text` с помощью регулярного выражения. Оно может показаться запутанным на первый взгляд, но мы рассмотрим его по частям:

```
'❶\s(?P<x>[a-z]+)\s+❷([a-z]+\s+){0,10}❸(?P=x)\s'
```

Мы начинаем с отдельного пробельного символа. Это позволяет гарантировать, что первый найденный фрагмент — целое слово (а не суффикс). Далее мы ищем соответствие поименованной группе `x`, состоящее из ненулевого числа символов в нижнем регистре от `'a'` до `'z'`, за которым следует ненулевое число пробельных символов ❶.

Затем мы ищем от 0 до 10 слов, каждое из которых состоит из ненулевого числа символов в нижнем регистре от `'a'` до `'z'`, за которым следует ненулевое число пробельных символов ❷.

И заканчиваем поименованной группой `x`, за которой следует пробельный символ, гарантируя, что последний найденный фрагмент — целое слово (а не префикс) ❸.

Результаты работы этого фрагмента кода выглядят так:

```
## Результаты
print(style_problems)
# <re.Match object; span=(12, 35), match=' words too often words '>
```

Мы нашли соответствующую регулярному выражению подстроку, стиль которой можно счесть (или не счесть) плохим.

В данном однострочнике мы свели задачу поиска повторов слов к простейшей и решили этот более простой вариант. Учтите, что на практике придется учитывать и более сложные случаи, например наличие специальных символов, сочетание символов в нижнем и верхнем регистре, чисел и т. д. Либо можно выполнить предварительную обработку и привести текст в желаемый вид: нижний регистр, никаких специальных символов, слова разделяются пробелами.

УПРАЖНЕНИЕ 5.1

Напишите сценарий Python, учитывающий возможность наличия специальных символов, например символов, используемых для структурной организации предложений (точек, двоеточий, запятых).

Модификация задаваемых регулярными выражениями шаблонов в многострочном строковом значении

В последнем однострочнике вы научитесь модифицировать текст, а не только находить его части, соответствующие регулярному выражению.

Общее описание

Для замены всех вхождений конкретного шаблона *регулярное_выражение* новой строкой символов `замена` в заданном тексте служит функция `re.sub(`*регулярное_выражение*`,` *замена*`,` *текст*`)`. С ее помощью можно быстро редактировать большие массивы текста, обходясь без выполнения ручной работы.

В предыдущих разделах вы научились находить соответствие в тексте шаблонам регулярных выражений. Но что, если определенный шаблон нужно считать найденным только в том случае, если не встречается какой-то другой шаблон? Задаваемый регулярным выражением `A(?!X)` шаблон *негативного опережающего просмотра* (negative lookahead) считается соответствующим регулярному выражению `A`, только если далее не найдено регулярное выражение `X`. Например, регулярному выражению `(?!good)` удовлетворяет строка `'this is not great'`, но не строка `'this is not good'`.

Код

Наши данные представляют собой строковое значение, и задача — заменить вхождения `Alice Wonderland` на `Alice Doe`, не заменяя при этом вхождений `'Alice Wonderland'` (заключенных в одинарные кавычки) (листинг 5.10).

Листинг 5.10. Однострочное решение для замены комбинаций символов в тексте

```
## Зависимости
import re

## Данные
text = '''
Alice Wonderland married John Doe.
The new name of former 'Alice Wonderland' is Alice Doe.
Alice Wonderland replaces her old name 'Wonderland' with her new name 'Doe'.
Alice's sister Jane Wonderland still keeps her old name.
'''

## Однострочник
updated_text = re.sub("Alice Wonderland(?!')", 'Alice Doe', text)

## Результат
print(updated_text)
```

Этот код выводит модифицированный текст. Какой?

Принцип работы

Мы заменяем все вхождения `Alice Wonderland` на `Alice Doe`, кроме тех, которые заканчиваются одинарной кавычкой `'`. Для этого мы воспользовались негативным опережающим просмотром. Обратите внимание, что мы проверяем только наличие закрывающей кавычки. Например, нашему регулярному выражению удовлетворяет строковое значение *с* открывающей кавычкой, но *без* закрывающей, так что мы просто заменим его. В общем случае это может быть нежелательно, но в нашем конкретном примере строкового значения приводит к желаемому поведению:

```
## Результат
print(updated_text)
'''
Alice Doe married John Doe.
The new name of former 'Alice Wonderland' is Alice Doe.
Alice Doe replaces her old name 'Wonderland' with her new name 'Doe'.
Alice's sister Jane Wonderland still keeps her old name.
'''
```

Как видите, исходное имя `'Alice Wonderland'` не меняется, когда заключено в одинарные кавычки, — что нам и нужно было.

Итоги главы

В этой главе мы охватили очень многое. Вы узнали о том, как искать соответствие шаблону в заданной строке с помощью регулярных выражений. В частности, узнали о функциях `re.compile()`, `re.match()`, `re.search()`, `re.findall()` и `re.sub()`. Вместе взятые, они охватывают значительную долю сценариев использования регулярных выражений. По мере применения регулярных выражений на практике вы познакомитесь и с другими функциями.

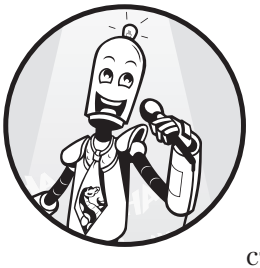
Кроме того, вы изучили различные простые регулярные выражения, комбинации которых позволяют создавать более сложные. Вдобавок вы узнали о пробельных символах, экранировании символов, «жадных»/«нежадных» операторах, наборах символов (и негативных наборах символов), группировке и поименованных группах, а также о негативном опережающем просмотре. И наконец, вы узнали, что зачастую лучше решить упрощенный вариант исходной задачи, чем пытаться с самого начала получить обобщенное решение.

Вам осталось только начать применять новоприобретенные знания о регулярных выражениях на практике. Чтобы привыкнуть к регулярным выражениям, лучше всего пробовать использовать их в своей любимой программе для редактирования текста. Большинство продвинутых программ для редактирования текста и кода (включая Notepad++) включает широкую функциональность по работе с регулярными выражениями. Кроме того, задумайтесь о применении регулярных выражений при работе с текстовыми данными (например, при написании электронных писем, сообщений в блогах, книг и кода). Регулярные выражения очень упрощают жизнь и экономят многие часы утомительной работы.

В следующей главе мы углубимся в важнейшую составляющую программирования: алгоритмы.

6

Алгоритмы



Алгоритмы известны уже очень давно. *Алгоритм* — это просто набор инструкций наподобие кулинарного рецепта. Однако важность *роли* алгоритмов в обществе стремительно растет: алгоритмы и основанный на них процесс принятия решений проникают во все сферы по мере того, как компьютеры становятся все более важной частью нашей жизни.

В исследовании 2018 года подчеркивается, что «данные в виде наблюдений над нашим миром насквозь пропитывают современное общество... Эта информация может, в свою очередь, использоваться для принятия взвешенных — а в некоторых случаях даже полностью автоматизированных — решений... Вполне вероятно, что подобные алгоритмы в будущем будут активно участвовать в процессах принятия решений людьми, что необходимо для общественного признания, а значит, и широкомасштабного применения».

ПРИМЕЧАНИЕ

Больше информации об этом исследовании вы можете найти в статье The Growing Ubiquity of Algorithms in Society: Implications, Impacts, and Innovations авторов S. C. Olhede и P. J. Wolfe: https://royalsocietypublishing.org/doi/full/10.1098/rsta.2017.0364#d2696064e1.

По мере развития в обществе тенденций к автоматизации, искусственному интеллекту и повсеместному применению компьютеров социальная пропасть между теми, кто понимает алгоритмы, и теми, кто их не понимает, стремительно ширится. Например, в сфере логистики наблюдается масштабная тенденция к автоматизации, в том числе появляются беспилотные автомобили и грузовики, и профессиональные водители вынуждены принять тот факт, что алгоритмы постепенно отбирают у них работу.

Постоянная смена спектра востребованных навыков и специальностей в XXI столетии требует от молодежи понимания, умения настраивать и менять простые алгоритмы. И когда неизменно только изменение, основные идеи алгоритмов и теории алгоритмов формируют фундамент, на котором основываются многие грядущие изменения. Проще говоря, понимание алгоритмов гарантирует ваше процветание в ближайшие десятилетия.

Эта глава призвана расширить ваше понимание алгоритмов, делая упор скорее на интуитивном и всестороннем понимании основных идей и реализаций на практике, чем на теории. И хотя теория алгоритмов ничуть не менее важна, чем реализация их на практике и понимание на понятийном уровне, многие замечательные книги посвящены в основном теории. Прочитав данную главу, вы будете интуитивно понимать некоторые наиболее популярные в computer science алгоритмы и еще больше разовьете навыки практической реализации на языке Python. Это обеспечит вам надежный фундамент для будущих научно-технических прорывов.

ПРИМЕЧАНИЕ

Книга Томаса Кормена (Thomas Cormen) и др. Introduction to Algorithms (MIT Press, 2009)[1] — замечательный источник дополнительной информации по теории алгоритмов.

Начнем с маленького алгоритма, предназначенного для решения простой задачи, актуальной для программистов, желающих найти хорошую работу.

[1] *Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К.* Алгоритмы: построение и анализ. 3-е изд. — М.: Вильямс, 2013.

Поиск анаграмм с помощью лямбда-функций и сортировки

Анаграммы — часто встречающийся при собеседованиях программистов вопрос на проверку кругозора в области компьютерных наук и умения разрабатывать собственные простые алгоритмы. В этом разделе мы расскажем о простом алгоритме для поиска анаграмм на языке Python.

Общее описание

Два слова являются *анаграммами*, если состоят из одинаковых символов и каждый символ первого из них встречается во втором ровно один раз. Ниже в списке и на рис. 6.1 даны несколько примеров:

- *listen → silent*;
- *funeral → real fun*;
- *elvis → lives*.

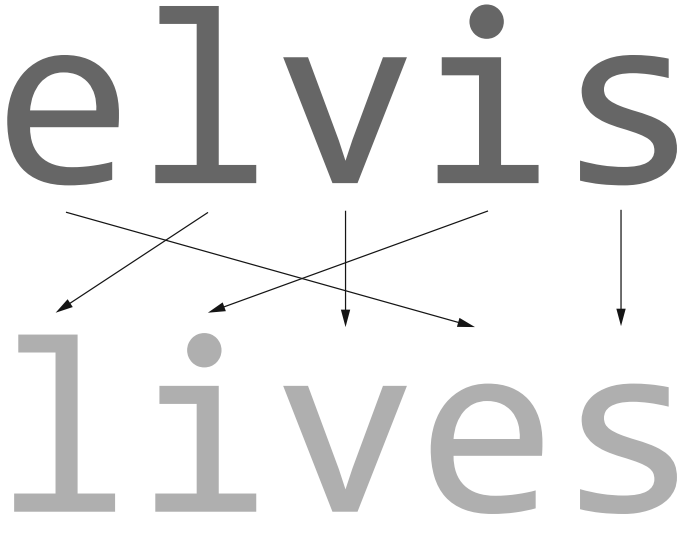


Рис. 6.1. Слово elvis — анаграмма слова lives

Займемся этой задачей и найдем лаконичное решение в стиле Python для определения того, являются ли два слова анаграммами. Что ж, приступим к написанию кода.

Код

Наша задача — написать функцию `is_anagram()`, которая принимает на входе два строковых значения `x1` и `x2` и возвращает `True`, если они — анаграммы! Прежде чем продолжить чтение, на минуту задумайтесь об этой задаче. Как бы вы стали решать ее на Python? Одно из возможных решений приведено в листинге 6.1.

Листинг 6.1. Однострочное решение для проверки того, являются ли два строковых значения анаграммами

```
   ## Однострочник
❶  is_anagram = lambda x1, x2: sorted(x1) == sorted(x2)

   ## Результат
   print(is_anagram("elvis", "lives"))
   print(is_anagram("elvise", "livees"))
   print(is_anagram("elvis", "dead"))
```

Этот код выводит три строки. Какие?

Принцип работы

Два строковых значения — анаграммы, если у них совпадают отсортированные последовательности символов, так что мы сортируем их и сравниваем поэлементно. Это несложно и не требует никаких внешних зависимостей. Просто создаем функцию `is_anagram()` ❶ путем описания лямбда-функции (см. главу 1) с двумя аргументами `x1` и `x2`, которая возвращает результат выражения `sorted(x1) == sorted(x2)` — `True`, если отсортированные последовательности символов совпадают. Ниже представлен результат сортировки двух последовательностей символов:

```
print(sorted("elvis"))
# ['e', 'i', 'l', 's', 'v']

print(sorted("lives"))
# ['e', 'i', 'l', 's', 'v']
```

Обе строки `'elvis'` и `'lives'` состоят из одних символов, так что их представления в виде отсортированного списка одинаковы. Результаты вышеприведенных трех операторов `print`:

```
## Результаты
print(is_anagram("elvis", "lives")) # True
print(is_anagram("elvise", "livees")) # True
print(is_anagram("elvis", "dead")) # False
```

В качестве небольшого примечания для опытных программистов скажем вот что: сложность сортировки последовательности n элементов на языке Python растет асимтотически, как функция от $n \log(n)$. Это означает, что наш однострочный алгоритм эффективнее «наивного» решения, состоящего в проверке наличия каждого символа в обоих строковых значениях и его удаления в этом случае. Сложность «наивного» алгоритма растет асимтотически, как квадратичная функция $n**2$.

Однако существует и другой эффективный способ решения этой задачи — создание *гистограмм* для обоих строковых значений на основе подсчета количества вхождений всех символов строки с последующим сравнением обеих гистограмм. Если размер алфавита не меняется, то сложность вычисления при таком подходе линейна и растет асимптотически как функция n. Оставляем реализацию этого алгоритма в качестве маленького упражнения для читателей!

Поиск палиндромов с помощью лямбда-функций и негативных срезов

В этом разделе вы познакомитесь еще с одним термином computer science, часто встречающимся в вопросах на собеседованиях: палиндромы. Мы проверим с помощью однострочника, являются ли два слова палиндромами друг друга.

Общее описание

Для начала: что такое палиндром? *Палиндром* — это последовательность элементов (например, строка или список), которая читается одинаково от начала к концу и наоборот. Рассмотрим несколько забавных примеров палиндромов (без учета пробелов).

- *Mr Owl ate my metal worm.*
- *Was it a car or a cat I saw?*
- *Go hang a salami, I'm a lasagna hog.*
- *Rats live on no evil star.*
- *Hannah.*

- *Anna.*
- *Bob.*

Наше однострочное решение потребует некоторых знаний о срезах. Как вы знаете из главы 2, срезы в Python означают «вырезание» диапазона значений из различных типов последовательностей, например строк или списков. Для среза, начинающегося с индекса *начало* (включая его) и заканчивающего на индексе *конец* (исключая его), используется очень лаконичная нотация [*начало*:*конец*:*шаг*]. Третий параметр *шаг* позволяет задавать *размер шага* — количество элементов исходной последовательности, пропускаемых перед следующим элементом среза (например, *шаг*=2 означает, что срез будет включать только каждый второй элемент). При отрицательном размере шага последовательность обходится в обратном порядке.

Вот и все, что нужно знать для создания простого и лаконичного однострочного решения на Python.

Код

Наш код должен определять, совпадают ли символы заданной строки символов в обратном порядке с исходной строкой, то есть определять, является ли эта строка палиндромом.

Листинг 6.2. Однострочное решение, проверяющее, является ли строковое значение палиндромом

```
## Однострочник
is_palindrome = lambda phrase: phrase == phrase[::-1]

## Результат
print(is_palindrome("anna"))
print(is_palindrome("kdljfasjf"))
print(is_palindrome("rats live on no evil star"))
```

Принцип работы

Наше простое однострочное решение не требует для работы никаких внешних библиотек. Мы описываем лямбда-функцию, которая принимает один аргумент `phrase` — проверяемую строку символов — и возвращает булево значение, указывающее, остается ли последовательность символов такой же в обратном порядке. Для получения строки символов в обратном порядке мы используем срез (см. главу 2).

Результаты этого фрагмента кода выглядят следующим образом:

```
## Результат
print(is_palindrome("anna")) # True
print(is_palindrome("kdljfasjf")) # False
print(is_palindrome("rats live on no evil star")) # True
```

Первая и третья строки символов — палиндромы, а вторая — нет. Далее мы займемся еще одним популярным в computer science понятием: перестановками.

Подсчет количества перестановок с помощью рекурсивных функций вычисления факториалов

В этом разделе мы продемонстрируем простой и эффективный способ вычисления факториала в одной строке кода для определения максимального количества возможных перестановок в наборе данных.

Общее описание

Рассмотрим следующую задачу: английская Премьер-лига состоит из 20 футбольных команд, каждая из которых может занимать по итогам сезона одно из 20 мест в рейтинге. При фиксированном количестве в 20 команд можно вычислить, сколько существует возможных вариантов рейтинга. Обратите внимание, что вопрос не в том, сколько мест в рейтинге может занять одна команда (разумеется, ответ на этот вопрос — 20), а сколько вообще существует рейтингов всех команд. На рис. 6.2 показаны всего лишь три возможных рейтинга.

Говоря языком теории computer science, каждый конкретный рейтинг называется *перестановкой*, то есть определенным порядком элементов множества. Наша задача — найти количество возможных перестановок заданного множества. Количество перестановок играет важную роль в приложениях, связанных с игрой на тотализаторе, предсказанием результатов матчей и анализом игр. Например, если начальные вероятности каждого из 100 различных рейтингов одинаковы, то вероятность каждого отдельного рейтинга равна 1/100 = 1 %. Это значение может использоваться в качестве базовой *(априорной)* вероятности для алгоритмов предсказания результатов игр. При таких допущениях вероятность выбранного случайным образом рейтинга оказаться правильным исходом в конце сезона равна 1 %.

Рис. 6.2. Три возможных рейтинга футбольных команд английской Премьер-лиги

Вычислить количество перестановок заданного множества из n элементов можно с помощью функции факториала $n!$. Из следующих нескольких абзацев вы узнаете почему. Определение факториала выглядит следующим образом:

$$n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1.$$

Например:

$$1! = 1$$

$$3! = 3 \times 2 \times 1 = 6$$

$$10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 3\,628\,800$$

$$20! = 20 \times 19 \times 18 \times \ldots \times 3 \times 2 \times 1 = 2\,432\,902\,008\,176\,640\,000.$$

Посмотрим, как работает эта функция. Пусть дано множество из 10 элементов $S = \{s0, s1, s2, \ldots, s9\}$ и 10 корзин $B = \{b0, b1, b2, \ldots, b9\}$. Мы хотим поместить в каждую корзину ровно один элемент из множества *S*. В нашем футбольном примере 20 команд играют роль элементов, а позиции рейтинга — роль корзин. Каждая конкретная перестановка множества *S* получается просто путем помещения всех элементов во все корзины. Количество различных способов распределения элементов по корзинам и будет общим количеством перестановок элементов множества *S*.

Количество перестановок множества из десяти элементов (которые необходимо разместить по десяти корзинам) можно определить с помощью следующего алгоритма.

1. Берем первый элемент из множества *S*. *Пустых корзин — десять*, так что у нас *десять вариантов* того, куда поместить данный элемент. Помещаем этот один элемент в какую-то корзину.
2. Одна корзина уже занята. Берем из множества *S* еще один элемент. Осталось *девять пустых корзин*, а значит, *девять вариантов*.
3. Наконец, берем из множества *S* десятый (последний) элемент. Девять корзин уже занято. Осталась только одна *пустая корзина*, а значит, *только один вариант*.

В целом получаем $10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 10!$ вариантов. Каждое из потенциальных размещений элементов по корзинам соответствует одной перестановке элементов множества. Следовательно, количество перестановок элементов множества из *n* элементов равно $n!$.

Рекурсивно функцию факториала можно определить следующим образом:

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

Граничные случаи рекурсии задаются вот так:

$$1! = 0! = 1.$$

Интуитивно эти значения понятны, поскольку у множества из одного элемента существует только одна возможная перестановка, как и у множества из нуля элементов (существует только один способ распределения нуля элементов по нулю корзин).

Код

Однострочник из листинга 6.3 вычисляет количество $n!$ перестановок множества из n элементов.

Листинг 6.3. Однострочное решение для рекурсивного вычисления функции факториала

```
## Данные
n = 5

## Однострочник
factorial = lambda n: n * factorial(n-1) if n > 1 else 1

## Результат
print(factorial(n))
```

Попробуйте догадаться, какой результат вернет этот код.

Принцип работы

В этом коде используется рекурсивное определение факториала. Вкратце формализуем наше интуитивное представление о рекурсии. Стивен Хокинг придумал лаконичный способ пояснить, что такое рекурсия: «Чтобы понять, что такое рекурсия, нужно сначала понять, что такое рекурсия».

Словарь Merriam-Webster дает определение рекурсии как «методики программирования компьютеров, при котором функция вызывает саму себя один или несколько раз до тех пор, пока не выполнится определенное условие, причем остальные вызовы при каждом из этих повторов обрабатываются, начиная с последнего вызова и заканчивая первым». Краеугольный камень этого определения — *рекурсивная функция*, то есть просто функция, вызывающая сама себя. Но если функция вызывает сама себя, то ее выполнение никогда не закончится. Поэтому задается определенное граничное условие. По его достижении последний вызов функции завершается и возвращает результат в предпоследний вызов. Предпоследний вызов, в свою очередь, возвращает результат в предпредпоследний вызов. Возникает цепная реакция распространения результатов на верхние уровни рекурсии до тех пор, пока первый вызов не вернет вызывающей стороне окончательный результат. Возможно, эту идею непросто изложить в нескольких строках, но потерпите немного: мы обсудим ее далее на примере нашего однострочника.

В общем случае создание рекурсивной функции *f* включает четыре этапа.

1. Разбиение исходной задачи на меньшие подзадачи.
2. Использование этих меньших подзадач в качестве входных данных для функции *f* (которая затем разобьет их на еще меньшие шаги и т. д.).
3. Описание *граничного случая* (base case) — минимального варианта входных данных, вычисление которого возможно без дальнейших вызовов функции *f*.
4. Указание способа объединения полученных меньших результатов в общий результат.

Мы создали лямбда-функцию с одним аргументом `n` и присвоили ее переменной `factorial`. Далее мы вызвали поименованную функцию `factorial(n-1)` для вычисления результата вызова функции `factorial(n)`. Значение `n` может представлять собой количество футбольных команд в премьер-лиге (`n=20`) или любое другое значение, например, как в листинге 6.3 (см. выше).

Попросту говоря, мы формируем более сложное решение задачи `factorial(n)`, умножая более простое решение `factorial(n-1)` на входной аргумент `n`. По достижении граничного случая `n <= 1` мы просто возвращаем «жестко зашитое» в код решение `factorial(1) = factorial(0) = 1`.

Данный алгоритм демонстрирует: если сначала тщательно обдумать задачу, то часто можно найти простой, лаконичный и эффективный способ ее решения. Выбор простейшего решения — один из важнейших элементов создания собственных алгоритмов. Начинающие часто замечают, что пишут громоздкий и переусложненный код.

В данном случае рекурсивное (однострочное) определение факториала короче итеративного (однострочного) определения без рекурсии. В качестве упражнения попробуйте переписать этот однострочник без рекурсии и без внешних библиотек — это отнюдь не тривиально и явно потребует намного более длинного кода!

Вычисление расстояния Левенштейна

В этом разделе вы узнаете о важном алгоритме, используемом для вычисления расстояния Левенштейна. Разобраться в данном алгоритме сложнее,

чем в предыдущих, так что заодно вы потренируете и свои навыки четкого анализа задачи.

Общее описание

Расстояние Левенштейна (Levenshtein distance) — метрика вычисления расстояния между двумя строками символов; другими словами, оно служит для количественной оценки подобия двух строк символов. Другое его название, *расстояние редактирования* (edit distance), в точности описывает, что именно измеряется с его помощью: сколько правок символов (вставок, удалений и замен) необходимо для преобразования одной строки в другую. Чем меньше расстояние Левенштейна, тем более схожи строки.

Расстояние Левенштейна нашло применение, в частности, в программе автоисправления в вашем смартфоне. Если вы введете в своем мессенджере Whatsapp *helo*, то смартфон обнаружит слово, не входящее в словарь, и выберет несколько наиболее вероятных слов для замены, после чего отсортирует их по расстоянию Левенштейна. Например, в данном случае слово с минимальным расстоянием Левенштейна, а значит максимальным подобием, — строка символов *hello*, так что телефон автоматически заменит *helo* на *hello*.

Рассмотрим пример с двумя менее схожими строками `'cat'` и `'chello'`. В табл. 6.1 приведена минимальная последовательность правок, необходимая для получения второй строки из первой, определяющая расстояние Левенштейна.

Таблица 6.1. Минимальная последовательность правок для преобразования 'cat' в 'chello'

Текущее слово	Выполненная правка
cat	—
cht	Заменяем a на h
che	Заменяем t на e
chel	Вставляем l на позиции 3
chell	Вставляем l на позиции 4
chello	Вставляем o на позиции 5

Таблица 6.1 демонстрирует преобразование строки символов `'cat'` в `'chello'` за пять шагов, так что расстояние Левенштейна равно 5.

Код

Теперь напишем однострочник Python для вычисления расстояния Левенштейна между строками символов `a` и `b`, `a` и `c`, `b` и `c` (листинг 6.4).

Листинг 6.4. Вычисление расстояния Левенштейна для двух строк символов в одной строке кода

```
## Данные
a = "cat"
b = "chello"
c = "chess"

## Однострочник
ls = ❶lambda a, b: len(b) if not a else len(a) if not b else min(
  ❷ls(a[1:], b[1:])+(a[0] != b[0]),
  ❸ls(a[1:], b)+1,
  ❹ls(a, b[1:])+1)

## Результат
print(ls(a,b))
print(ls(a,c))
print(ls(b,c))
```

Попробуйте вычислить результаты работы этого фрагмента кода до запуска программы.

Принцип работы

Прежде чем углубляться в код, посмотрим на важный трюк Python, активно используемый в этом однострочнике. В Python у *каждого* объекта есть булево значение, равное `True` или `False`. На самом деле большинство объектов равны `True`, и интуитивно понятно, что можно догадаться о том, что лишь немногие объекты равны `False`:

- числовое значение `0` равно `False`;
- пустая строка `''` равна `False`;
- пустой список `[]` равен `False`;
- пустое множество `set[]` равно `False`;
- пустой ассоциативный массив `{}` равен `False`.

В качестве эмпирического правила: объекты Python считаются `False`, если они пусты или равны нулю. Вооружившись этой информацией, мы можем взглянуть на первую часть функции для вычисления расстояния Левенштейна: создадим лямбда-функцию, принимающую на входе два строковых значения `a` и `b` и возвращающую количество правок, необходимое для преобразования `a` в `b` ❶. Существуют два тривиальных случая: если строковое значение `a` пусто, то минимальное расстояние редактирования равно `len(b)`, поскольку достаточно просто вставить все символы строкового значения `b` по одному. Аналогично, если пусто строковое значение `b`, то минимальное расстояние редактирования равно `len(a)`. Это значит, если одно из этих строковых значений пусто, то можно сразу вернуть правильное расстояние редактирования.

Пусть оба строковых значения не пусты. Мы можем упростить задачу, вычислив расстояние Левенштейна меньших суффиксов исходных строковых значений `a` и `b`, как показано на рис. 6.3.

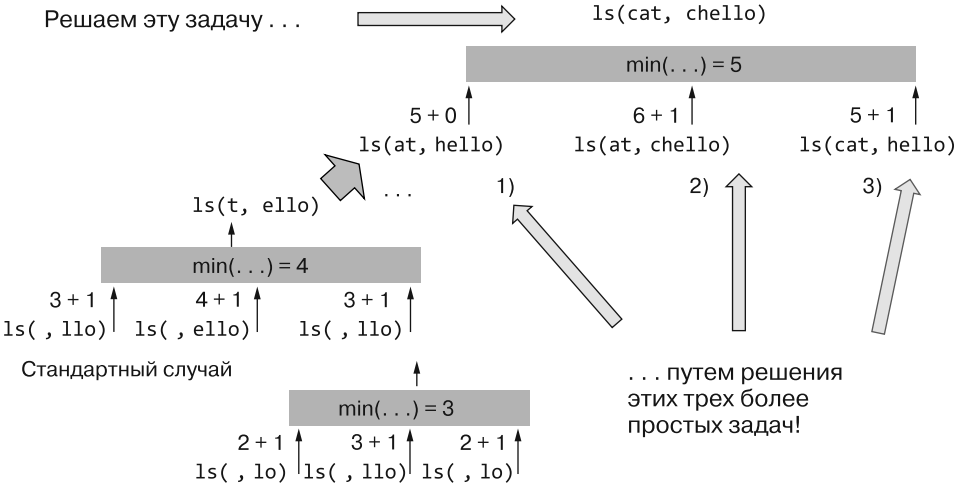


Рис. 6.3. Рекурсивное вычисление расстояния Левенштейна для слов 'cat' и 'chello' путем вычисления сначала меньших шагов задачи

Чтобы рекурсивно вычислить расстояние Левенштейна для слов `'cat'` и `'chello'`, сначала мы решаем более простые задачи вычисления (рекурсивно).

1. Вычисляем расстояние между суффиксами `at` и `hello`, поскольку если известно, как преобразовать `at` в `hello`, то можно легко преобразовать

`cat` в `chello`, модифицируя сначала первый символ (или сохраняя его неизменным, если оба строковых значения начинаются с одного символа). Если это расстояние равно 5, то можно сделать вывод, что расстояние между `cat` и `chello` также не превышает 5, ведь можно получить одно из другого с помощью той же самой последовательности правок (оба слова начинаются с символа `c`, поэтому его редактировать не нужно).

2. Вычисляем расстояние между суффиксами `at` и `chello`. Если оно равно 6, то можно сделать вывод, что расстояние между `cat` и `chello` не превышает 6 + 1 = 7, поскольку можно просто убрать символ `c` в начале первого слова (одна дополнительная операция). А затем повторно использовать то же самое решение для получения `chello` из `at`.
3. Вычисляем расстояние между `cat` и `hello`. Если оно равно 5, то можно сделать вывод, что расстояние между `cat` и `chello` не превышает 5 + 1 = 6, поскольку можно просто вставить символ `c` в начале второго слова (одна дополнительная операция).

А поскольку это все возможные операции над первым символом (замена, удаление, вставка), то расстояние Левенштейна между `cat` и `chello` равно минимальному из трех случаев 1, 2 и 3. Теперь подробнее рассмотрим три случая в листинге 6.4.

Во-первых, мы рекурсивно вычисляем расстояние редактирования между `a[1:]` и `b[1:]` ❷. Если ведущие символы `a[0]` и `b[0]` различаются, то необходимо исправить ситуацию, *заменив* `a[0]` на `b[0]`, так что мы увеличиваем расстояние редактирования на единицу. При совпадении ведущих символов решение более простой задачи `ls(a[1:], b[1:])` совпадает с решением более сложной задачи `ls(a, b)`, как вы видели на рис. 6.3.

Во-вторых, мы рекурсивно вычисляем расстояние между `a[1:]` и `b` ❸. Допустим, нам известен результат вычисления (преобразования `a[1:]` в `b`) — как теперь вычислить расстояние на один шаг дальше, от `a` до `b`? Ответ: просто *удалить* первый символ `a[0]` из начала `a`, то есть произвести ровно на одну операцию больше. Благодаря этому мы свели более сложную задачу к более простой.

В-третьих, мы рекурсивно вычисляем расстояние между `a` и `b[1:]` ❹. Допустим, нам известен результат вычисления (преобразования `a` в `b[1:]`). Как теперь вычислить расстояние от `a` до `b`? В данном случае мы просто проходим на шаг больше (от `a` до `b[1:]` до `b`), *вставляя* символ `b[0]` в начале слова `b[1:]`, что увеличивает расстояние на единицу.

Наконец, мы просто берем минимальное расстояние редактирования из трех (замены первого символа, удаления первого символа и вставки первого символа).

Данное однострочное решение опять демонстрирует важность навыков работы с рекурсией. Возможно, понимание рекурсии не дастся вам очень легко, но несомненно придет к вам по мере изучения множества задач на рекурсию, подобных этой.

Вычисление булеана с помощью функционального программирования

В этом разделе мы расскажем вам о важном математическом понятии булеана: множества всех подмножеств. Булеаны используются в статистике, теории множеств, функциональном программировании, теории вероятностей и алгоритмическом анализе.

Общее описание

Булеан (powerset) — это множество всех подмножеств заданного множества `s`, включающее пустое множество `{}`, исходное множество `s` и все прочие возможные подмножества исходного множества. Ниже даны несколько примеров.

Пример 1:

- множество: `s = {1}`;
- его булеан: `P = {{},{1}}`.

Пример 2:

- множество: `s = {1, 2}`;
- его булеан: `P = {{},{1},{2},{1,2}}`.

Пример 3:

- множество: `s = {1, 2, 3}`;
- его булеан: `P = {{},{1},{2},{3},{1,2},{1,3},{2,3},{1,2,3}}`.

Для вычисления булеана P_n множества *s*, состоящего из *n* элементов, можно использовать булеан P_{n-1} подмножества *s*, состоящего из $(n - 1)$ элемента. Пусть нам требуется вычислить булеан множества `s = {1, 2, 3}`.

1. Зададим начальное значение булеана P_0 с нулем элементов как $P_0 = \{\{\}\}$. Другими словами, это булеан пустого множества. Он содержит только само пустое множество.
2. Для создания на основе булеана P_{n-1} множества, состоящего из $(n - 1)$ элемента булеана P_n, состоящего из *n* элементов, возьмем один (произвольный) элемент *x* из множества *s* и включим все получающиеся подмножества в больший булеан P_n с помощью следующей процедуры.
3. Проходим по всем множествам *p* в P_{n-1} и создаем новые подмножества, состоящие из объединения *x* и *p*. В результате получаем новое временное множество множеств *T*. Например, если $P_2 = \{\{\}, \{1\}, \{2\}, \{1,2\}\}$, то мы создадим временное множество множеств $T = \{\{3\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$ путем добавления элемента *x* ко всем множествам из P_2.
4. Объединяем это новое множество множеств *T* с булеаном P_{n-1} и получаем булеан P_n. Например, булеан P_3 получается путем объединения временного множества *T* с булеаном P_2 следующим образом: $P_3 = T \cup P_2$.
5. Переходим на шаг 2 до тех пор, пока исходное множество *s* не окажется пустым.

Ниже вас ждут более подробные пояснения этой стратегии.

Функция reduce()

Но сначала необходимо как следует разобраться с важной функцией Python, которую мы применим в нашем однострочнике: `reduce()`. Она встроена в Python 2, но создатели Python решили, что использовали ее настолько мало, что в Python 3 она включена не была, поэтому придется сначала импортировать ее из библиотеки functools.

Функция `reduce()` принимает три аргумента: `reduce(`*`функция, итерируемый_объект, начальное_значение`*`)`. Аргумент *`функция`* определяет способ свертки двух значений `x` и `y` в одно (например, `lambda x, y: x + y`). Таким образом можно в цикле свертывать два значения *`итерируемый_объект`* (второй аргумент) в одно до тех пор, пока в *`итерируемый_объект`* не останется только одно значение. Аргумент *`начальное_значение`* — необязательный, если его

не указать, Python будет использовать по умолчанию первое значение *итерируемый_объект*.

Например, при вызове `reduce(lambda x, y: x + y, [0, 1, 2, 3])` производятся следующие вычисления: `(((0 + 1)+ 2)+ 3) = 6`. Другими словами, сначала два значения `x=0` и `y=1` свертываются в их сумму `x + y = 0 + 1 = 1`. Этот результат первого вызова лямбда-функции служит входными данными для второго ее вызова: `x=1` и `y=2`. В результате получается сумма `x + y = 1 + 2 = 3`. Наконец, результат этого второго вызова лямбда-функции служит входными данными для третьего ее вызова: `x=3` и `y=3`. В результате получается сумма `x + y = 3 + 3 = 6`.

В этом последнем примере, как вы могли заметить, значение `x` всегда было равно результату предыдущего вызова лямбда-функции. Аргумент `x` играет роль значения-накопителя, а аргумент `y` — *обновляемого* значения из *итерируемый_объект*. Такое поведение нацелено на «свертку» в цикле всех значений из *итерируемый_объект* в одно значение. Необязательный третий аргумент задает начальное значение для `x`. Все это позволяет описать *агрегатор для последовательностей*, как показано ниже в листинге 6.5.

Арифметические операции над списками

Прежде чем заняться непосредственно однострочником, вам нужно понимать, как работают еще две операции над списками. Первая из них — оператор конкатенации списков `+`, склеивающий два списка. Например, результат операции `[1, 2] + [3, 4]` — новый список `[1, 2, 3, 4]`. Вторая — оператор объединения `|`, производящий простую операцию объединения двух множеств. Например, результат выражения `{1, 2} | {3, 4}` — новое множество `{1, 2, 3, 4}`.

Код

В листинге 6.5 приведено однострочное решение для вычисления булеана заданного множества *s*.

Листинг 6.5. Однострочное решение для вычисления булеана заданного множества

```
## Зависимости
from functools import reduce

## Данные
s = {1, 2, 3}

## Однострочник
```

```
ps = lambda s: reduce(lambda P, x: ❶P + [subset | {x} for subset in P],
     s, ❷[set()])

## Результат
print(ps(s))
```

Угадайте, что вернет этот фрагмент кода!

Принцип работы

Идея данного однострочника заключается в том, чтобы начать формирование булеана с пустого множества ❷ и последовательно добавлять в него подмножества ❶, пока их больше не останется.

Изначально булеан содержит только пустое множество. На каждом шаге мы берем один элемент `x` из набора данных `s` и создаем новые подмножества, получающиеся естественным образом путем добавления `x` в каждое из подмножеств булеана ❷. Как вы уже видели чуть выше, размер булеана удваивается каждый раз, когда берется дополнительный элемент `x` из набора данных `s`. Таким образом можно наращивать булеан из *n* подмножеств по одному элементу набора данных за раз (но по *n* подмножеств за раз). Обратите внимание, что размер булеана растет экспоненциально: каждый новый элемент набора данных `x` приводит к удвоению размера булеана. Это неотъемлемое свойство булеанов: они стремительно заполняют любое хранилище — даже в случае относительно небольших наборов данных всего из нескольких десятков элементов.

С помощью функции `reduce()` мы производим хранение текущего булеана в переменной `P` (изначально содержащей только пустое множество). С помощью спискового включения функция `reduce()` создает новые подмножества — по одному для каждого существующего подмножества — и добавляет их в булеан `P`. В частности, она добавляет значение `x` из набора данных в каждое из подмножеств и тем самым удваивает размер булеана (который теперь содержит подмножества *с* элементом `x` набора данных и *без* него). Благодаря этому функция `reduce()` последовательно «сливает воедино» два элемента: булеан `P` и элемент `x` из набора данных.

Поэтому результат нашего однострочника выглядит следующим образом:

```
# Результат
print(ps(s))
# [set(), {1}, {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}]
```

Данный однострочник наглядно демонстрирует важность понимания лямбда-функций, списковых включений и операций над множествами.

Реализация шифра Цезаря с помощью расширенного доступа по индексу и спискового включения

В этом разделе мы расскажем вам о старинной методике шифрования — *шифре Цезаря*, — с помощью которого сам Гай Юлий Цезарь делал личные сообщения непонятными для врагов. К сожалению, шифр Цезаря слишком легко взламывается и никакой настоящей защиты не дает, но часто служит для развлечения и скрытия содержимого форумов (тех, которые можно найти в интернете, а не в Древнем Риме), которое необходимо защитить от глаз наивных пользователей.

Общее описание

В основе шифра Цезаря лежит идея сдвига шифруемых символов на фиксированное количество позиций в алфавите. Далее мы рассмотрим один из частных случаев шифра Цезаря — алгоритм ROT13.

ROT13 — простой алгоритм шифрования, используемый на многих форумах (например, Reddit) для защиты от спойлеров или скрытия смысла дискуссии от новичков. Дешифровка алгоритма ROT13 проста — злоумышленник может взломать ваш код с помощью вероятностного анализа распределения букв в зашифрованном тексте, даже не зная, на сколько позиций сдвинуты символы. Никогда не следует использовать этот алгоритм для настоящего шифрования сообщений! Тем не менее существует немало приложений алгоритма ROT13:

- скрытие ответов на головоломки на интернет-форумах;
- скрытие возможных спойлеров на фильмы или книги;
- высмеивание других слабых алгоритмов шифрования: «56-битный алгоритм DES по крайней мере лучше, чем ROT13»;
- скрытие адресов электронной почты на сайтах от 99,999 % спам-ботов.

Таким образом, ROT13 — скорее популярная дежурная шутка в интернет-культуре и образовательный инструмент, а не серьезный шифр.

Этот алгоритм можно объяснить одной фразой: ROT13 = *сдвиг шифруемой строки символов на 13 позиций (по модулю 26) в алфавите из 26 символов* (рис. 6.4).

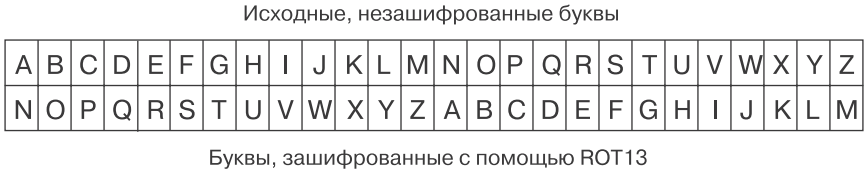


Рис. 6.4. Таблица демонстрирует, как зашифровывается и расшифровывается каждый из символов алфавита при использовании алгоритма ROT13

Другими словами, каждый символ сдвигается на 13 позиций по алфавиту. При выходе за пределы последнего символа, *z*, происходит переход к первому символу алфавита, *a*.

Код

В листинге 6.6 мы создаем однострочник для шифрования строки `s` с помощью алгоритма ROT13!

Листинг 6.6. Однострочное решение для шифрования строки с помощью алгоритма ROT13

```
## Данные
abc = "abcdefghijklmnopqrstuvwxyz"
s = "xthexrussiansxarexcoming"

## Однострочник
rt13 = lambda x: "".join([abc[(abc.find(c) + 13) % 26] for c in x])

## Результат
print(rt13(s))
print(rt13(rt13(s)))
```

Воспользуйтесь рис. 6.4, чтобы взломать этот шифр: каковы результаты выполнения данного фрагмента кода?

Принцип работы

Наше однострочное решение шифрует каждую из букв по отдельности, сдвигая ее на 13 позиций вправо по алфавиту, хранящемуся в переменной `abc`, после чего создает список этих зашифрованных букв и объединяет элементы этого списка в целую зашифрованную фразу `x`.

Рассмотрим подробнее способ шифрования каждой из букв. Для создания списка зашифрованных букв используется списковое включение (см. главу 2), каждая буква `c` заменяется буквой, расположенной на 13 позиций дальше в алфавите. При этом важно предотвратить выход за пределы алфавита для букв, где *индекс >= 13*. Например, при сдвиге буквы *z* с индексом 25 на 13 позиций получается недопустимый для алфавита индекс 25 + 13 = 38. Для решения этой проблемы мы используем оператор сложения по модулю, чтобы даже при выходе за максимальный индекс 25 для буквы *z* шифрование продолжалось, если *индекс = 0* (буква `a`). И продолжаем сдвигать вправо на оставшиеся из 13 позиций, которые еще не были учтены до перехода в начало алфавита (рис. 6.5). Например, буква *z* сдвигается на 13 позиций до индекса 38 по модулю 26 (в виде кода Python: `38%26`), то есть до индекса 12 (буквы `m`).

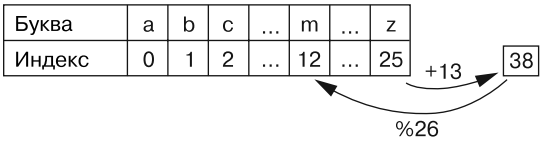


Рис. 6.5. Предотвращаем выход за пределы алфавита, начиная отсчет индекса заново с 0, в результате чего получается такая последовательность сдвига: 25 > 0 > 1 > > 12

Вот главная часть кода, описывающая сдвиг каждого символа `c` на 13 позиций:

```
abc[(abc.find(c) + 13) % 26]
```

Во-первых, мы находим индекс символа `c` в алфавите `abc`. Во-вторых, сдвигаем этот индекс, прибавляя целое число 13 к индексу символа `c` в алфавите `abc`, с учетом нашего приема с модулем 26 (как пояснялось в предыдущих абзацах).

Результат выполнения кода однострочника выглядит следующим образом:

```
## Результат
print(rt13(s))
# kgurkehffvnafknerkpbzvat

print(rt13(rt13(s)))
# xthexrussiansxarexcoming
```

Резюмируя: вы изучили частный случай шифра Цезаря, алгоритм ROT13, при котором каждая буква в строке сдвигается на 13 позиций в алфавите. При повторном сдвиге еще на 13 позиций индекса (13+13=26) получается исходная буква, так что для шифрования и дешифрования применяется один алгоритм.

Поиск простых чисел с помощью решета Эратосфена

Поиск простых чисел играет важнейшую роль на практике, в частности в криптографии. Многие методы с открытым ключом безопасны (с точки зрения криптографии) лишь потому, что вычисление простых множителей больших чисел — трудоемкий и медленный процесс. Мы создадим однострочник, отыскивающий все простые числа в заданном диапазоне с помощью одного древнего алгоритма.

Общее описание

Простое число n — целое число, которое не делится без остатка ни на какое целое число, кроме самого себя и 1. Другими словами, для простого числа n не существует двух целых чисел $a > 1$ и $b > 1$, чье произведение равнялось бы ему: $ab = n$.

Допустим, мы хотим проверить, является ли заданное число n простым. Начнем с «наивного» алгоритма поиска простых чисел (листинг 6.7).

Листинг 6.7. «Наивная» реализация проверки заданного числа n на простоту

```
def prime(n):
❶  for i in range(2,n):
    ❷ if n % i == 0:
            return False
```

```
    return True

print(prime(10))
# False

print(prime(11))
# True

print(prime(7919))
# True
```

Данный алгоритм проверяет, делится ли `n` на каждое из чисел от 2 до `n-1` ❶ без остатка ❷. Например, при проверке на простоту числа `n = 10` алгоритм быстро обнаруживает, что выражение `n % i == 0` равно `True` при `i = 2`. Алгоритм нашел число `i`, являющееся делителем `n`, поэтому `n` не может быть простым числом. В данном случае алгоритм прерывает все дальнейшие вычисления и возвращает `False`.

Временная сложность проверки отдельного числа совпадает (точнее, линейно зависит) с `n`: при наихудшем сценарии алгоритму понадобится `n` итераций цикла, чтобы проверить, простое ли `n`.

Пусть нам нужно вычислить все простые числа от 2 до определенного максимального числа `m`. Для этого можно просто повторить проверку из листинга 6.7 `m-1` раз (листинг 6.8). Однако вычислительные затраты при этом колоссальны.

Листинг 6.8. Поиск всех простых чисел, не превышающих максимального числа `m`

```
# Поиск всех простых чисел <= m
m = 20
primes = [n for n in range(2,m+1) if prime(n)]

print(primes)
# [2, 3, 5, 7, 11, 13, 17, 19]
```

Для создания списка всех простых чисел, не превышающих `m`, мы воспользовались списковым включением (см. главу 2). Включение в алгоритм цикла `for` означает необходимость `m` вызовов функции `is_prime(n)`, так что временная сложность ограничивается `m**2`. Количество операций растет квадратично относительно `m`. Поиск всех простых чисел меньше `m = 100` может потребовать до `m**2 = 10000` операций!

А теперь мы создадим однострочник, резко сокращающий подобные затраты времени.

Код

С помощью этого однострочника мы напишем алгоритм для поиска всех простых чисел, не превышающих максимального целого числа `m`, более эффективный, чем наша наивная реализация. Однострочник в листинге 6.9 основывается на древнем алгоритме под названием «решето Эратосфена», о котором я расскажу в этом подразделе.

Листинг 6.9. Однострочное решение, реализующее решето Эратосфена

```
## Зависимости
from functools import reduce

## Данные
n=100

## Однострочник
primes = reduce(lambda r, x: r - set(range(x**2, n, x)) if x in r else r,
                range(2, int(n**0.5) + 1), set(range(2, n)))

## Результат
print(primes)
# {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
#  47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}
```

Вероятно, вам понадобятся еще некоторые предварительные знания, чтобы понять происходящее.

Принцип работы

Откровенно говоря, я сомневался, включать ли этот однострочник в книгу из-за его запутанности, сложности и неудобочитаемости. Тем не менее именно такой код вы встретите на практике, и с помощью книги я хочу гарантировать, что вы сможете понять каждую его строку, даже если это займет немало времени. На одну из версий данного однострочника я натолкнулся на StackOverflow. Он является вольной интерпретацией древнего алгоритма под названием *решето Эратосфена*, предназначенного для вычисления простых чисел.

ПРИМЕЧАНИЕ

Я внес некоторые изменения в первоначальный однострочник StackOverflow ради ясности. На момент написания данной книги первоначальный однострочник можно было найти здесь: https://stackoverflow.com/questions/10639861/python-prime-generator-in-one-line/.

Алгоритм решета Эратосфена

По существу, данный алгоритм создает огромный массив чисел, от 2 до `m`, заданного максимального целого числа. Все числа в массиве — *кандидаты на роль простых чисел*, то есть алгоритм считает их *потенциально* (но не *обязательно*) простыми. В ходе алгоритма кандидаты, которые не могут быть простыми, отсеиваются. И лишь оставшиеся после этого процесса фильтрации числа окончательно считаются простыми.

Для этого алгоритм вычисляет и помечает в массиве те числа, которые не являются простыми. В конце его выполнения все непомеченные числа — заведомо простые.

Данный алгоритм повторяет следующие шаги.

1. Начинает с первого числа `2` и увеличивает его на единицу на каждом шаге процесса до тех пор, пока не будет найдено простое число x. Мы знаем, что x простое, если оно осталось непомеченным, ведь в этом случае ни одно меньшее x число не является его делителем, а это и есть определение простого числа.
2. Помечаем все числа, кратные x, поскольку они также не являются простыми: делителем для них является число x.
3. Простая оптимизация: начинаем помечать кратные числа, начиная с числа $x \times x$, а не $2x$, поскольку все числа между $2x$ и $x \times x$ уже помечены. Существует простое математическое доказательство этого факта, которое я опишу далее. А пока просто поверьте, что можно начинать помечать с $x \times x$.

Этот алгоритм показан пошагово на рис. с 6.6 по 6.11.

Изначально все числа от 2 до $m = 100$ не помечены (незакрашенные ячейки). Первое непомеченное число — 2 — простое.

Переходим к следующему непомеченному числу, 3. Поскольку оно еще не помечено, значит, простое: мы пометили все числа, кратные числам меньше текущего числа 3, так что никакое меньшее число не является его делителем. И, по определению, число 3 — простое. Помечаем все числа, кратные 3, поскольку они — не простые, начиная с числа 3×3, так как все кратные 3 числа между 3 и $3 \times 3 = 9$ уже помечены.

Начало

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	100

Рис. 6.6. Начальное состояние решета Эратосфена

Простое Помечаем все числа, кратные 2

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	100

Рис. 6.7. Помечаем все числа, кратные 2, как не простые. Игнорируем в оставшейся части алгоритма все помеченные числа

Переходим к следующему непомеченному числу, 5 (которое тоже простое). Помечаем все числа, кратные 5. Начинаем с числа 5×5, поскольку все кратные 5 числа между 5 и $5 \times 5 = 25$ уже помечены.

Простое ↓ Помечаем все числа, кратные 3 (начиная с 3^2)

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	100

Рис. 6.8. Помечаем все числа, кратные 3, как не простые

Простое ↓ Помечаем все числа, кратные 5 (начиная с 5^2)

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	100

Рис. 6.9. Помечаем все числа, кратные 5, как не простые

Переходим к следующему непомеченному числу, 7 (которое тоже простое). Помечаем все числа, кратные 7. Начинаем с числа 7×7, поскольку все кратные 7 числа между 7 и $7 \times 7 = 49$ уже помечены.

Помечаем все числа, кратные 7 (начиная с 7^2)

Простое ↓

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	100

Рис. 6.10. Помечаем все числа, кратные 7, как не простые

Переходим к следующему непомеченному числу, 11 (которое тоже простое). Помечаем все числа, кратные 11. Мы должны начать с числа $11 \times 11 = 121$, но понимаем, что оно превышает наш максимум $m = 100$. Так что алгоритм завершается. Все оставшиеся непомеченными числа не делятся ни на какое число, а значит, являются простыми.

Помечаем все числа, кратные 11 (начиная с 11^2) → Закончено

	1	2	3	4	5	6	7	8	9	10
Простое →	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	100

Рис. 6.11. Помечаем все числа, кратные 11, как не простые

Решето Эратосфена намного эффективнее, чем «наивный» алгоритм, поскольку последний проверяет все числа *независимо друг от друга*, без учета всех предыдущих вычислений. Решето же Эратосфена, напротив, *повторно использует* результаты предыдущих шагов вычислений — частый прием во многих сферах оптимизации алгоритмов. Каждый раз, вычеркивая число, кратное простому, мы, по существу, избавляем себя от утомительной работы по проверке того, является ли это число простым: мы заранее знаем, что оно им не является.

Наверное, вы недоумеваете, почему мы начинаем помечать числа с квадрата простого, а не самого простого. Например, в алгоритме на рис. 6.10 мы только что обнаружили простое число 7 и начали помечать с числа $7 \times 7 = 49$. Дело в том, что все остальные кратные числа ($7 \times 2, 7 \times 3, 7 \times 4, 7 \times 5, 7 \times 6$) мы уже пометили на предыдущих итерациях, когда помечали числа, кратные всем числам, меньшим нашего текущего простого числа 7: 2, 3, 4, 5, 6.

Пояснения к однострочнику

Досконально понимая алгоритм, мы можем наконец приступить к изучению нашего однострочного решения:

```
## Однострочник
primes = reduce(lambda r, x: r - set(range(x**2, n, x)) if x in r else r,
                range(2, int(n**0.5) + 1), set(range(2, n)))
```

В этом однострочнике для удаления по одному всех помеченных чисел из начального множества чисел от 2 до `n` (в однострочнике: `set(range(2, n))`) используется функция `reduce()`.

Это множество служит начальным значением множества непомеченных значений `r`, поскольку изначально все значения не помечены. Далее однострочник проходит по всем числам `x` от 2 до квадратного корня из `n` (в однострочнике: `range(2, int(n**0.5) + 1)`) и удаляет из множества `r` все числа, кратные `x` (начиная с `x**2`), но только если число `x` — простое, то есть не удалено на текущий момент из множества `r`.

Потратьте 5–15 минут, чтобы прочитать это объяснение снова, и внимательно изучите все части данного однострочника. Я обещаю, что вы потратите это время не напрасно и в результате существенно улучшите свои навыки понимания кода на языке Python.

Вычисление последовательности Фибоначчи с помощью функции reduce()

Знаменитый итальянский математик Фибоначчи (Леонардо Пизанский) открыл числа Фибоначчи в 1202 году, неожиданно обнаружив, что они играют важную роль в различных сферах: математике, искусстве и биологии. В этом разделе я покажу вам, как вычислить их с помощью одной строки кода.

Общее описание

Ряд Фибоначчи начинается с чисел 0 и 1, а каждое последующее число равно сумме двух предыдущих элементов ряда. Ряд Фибоначчи представляет собой готовый алгоритм!

Код

В листинге 6.10 вычисляется список первых *n* чисел Фибоначчи, начиная с 0 и 1.

Листинг 6.10. Вычисление последовательности Фибоначчи в одной строке кода на Python

```
## Зависимости
from functools import reduce

## Данные
n = 10

## Однострочник
fibs = reduce(lambda x, _: x + [x[-2] + x[-1]], [0] * (n-2), [0, 1])

## Результат
print(fibs)
```

Взгляните на код и попробуйте угадать результаты его выполнения.

Принцип работы

Мы опять воспользовались замечательной функцией `reduce()`. Вообще говоря, она удобна, когда необходимо агрегировать вычисляемую на ходу информацию о состоянии, например на основе только что полученных чисел Фибоначчи вычислить следующее число Фибоначчи. С помощью

спискового включения реализовать подобное непросто (см. главу 2), ведь оно не позволяет обычно обращаться к только что созданным им значениям. Мы передаем в функцию `reduce()` три аргумента, соответствующие `reduce(`*`функция, итерируемый_объект, начальное_значение`*`)`, чтобы последовательно добавлять новые числа Фибоначчи в объект-агрегатор, включающий по одному значению за раз из *итерируемого_объекта*, по задаваемому *функцией* сценарию.

АЛЬТЕРНАТИВНОЕ РЕШЕНИЕ

Суммирование раз за разом двух чисел Фибоначчи — простая идея, лежащая в основе однострочника из листинга 6.10. В листинге 6.11 приведено другое красивое решение.

Листинг 6.11. Однострочное решение для поиска чисел Фибоначчи другим путем

```
n = 10
x = [0,1]
fibs = x[0:2] + [x.append(x[-1] + x[-2]) or x[-1] for i in range(n-2)]
print(fibs)
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Этот фрагмент кода прислал один из подписчиков моей рассылки (не стесняйтесь присоединяться к ней по адресу https://blog.finxter.com/subscribe/), в нем используется списковое включение с побочным эффектом: переменная `x` обновляется новым элементом ряда Фибоначчи `(n-2)` раз. Обратите внимание, что функция `append()` не возвращает значения, а только `None`, эквивалентное `False`. Таким образом, оператор спискового включения генерирует список целых чисел на основе следующей идеи:

```
print(0 or 10)
# 10
```

На первый взгляд кажется, что нельзя производить операцию `or` над двумя целыми числами, но, как вы помните, в основе типа Boolean лежит тип integer. Любое отличное от нуля целочисленное значение интерпретируется как `True`. Таким образом, операция `or` позволяет просто возвращать второе целочисленное значение, вместо того чтобы преобразовывать его в булево значение `True`. Очень изящный фрагмент кода Python!

Объектом-агрегатором тут служит простой список с двумя начальными значениями: `[0, 1]`. Напомним, что объект-агрегатор передается *функции* в качестве первого аргумента (в нашем примере `x`).

Второй аргумент — следующий элемент из *итерируемого_объекта*. Мы задали `(n-2)` фиктивных значения в качестве начального значения *итерируемый_объект*, чтобы заставить функцию `reduce()` выполнить *функцию* `(n-2)` раз (чтобы найти первые `n` чисел Фибоначчи — первые два, 0 и 1, мы уже знаем). Чтобы показать, что фиктивные значения *итерируемого_объекта* нас не интересуют, мы воспользовались «мусорным» параметром `_` и просто добавляем новые числа Фибоначчи, вычисленные как сумма двух предыдущих чисел Фибоначчи, в конец списка агрегатора `x`.

Резюмируя: вы научились работать еще с одним паттерном однострочников Python, а именно созданием с помощью функции `reduce()` списка, с динамическим использованием только что обновленных или добавленных элементов для вычисления его новых элементов. Этот удобный паттерн очень часто встречается на практике.

Рекурсивный алгоритм бинарного поиска

В этом разделе вы узнаете о простом алгоритме, который обязан знать любой специалист по computer science: алгоритме бинарного поиска. У бинарного поиска есть множество важных приложений во многих реализациях простых структур данных: множеств, деревьев, ассоциативных массивов, хешированных множеств, хешированных таблиц, хеш-карт и массивов. Эти структуры данных используются в любой нетривиальной программе.

Общее описание

Если коротко, *алгоритм бинарного поиска* (binary search algorithm) производит в отсортированной последовательности `l` поиск конкретного значения `x` путем многократного сокращения размера этой последовательности вдвое, до тех пор, пока не останется только одно значение. Либо это будет искомое значение, либо его вообще не было в последовательности. Далее мы рассмотрим эту общую идею подробнее.

Например, представьте, что нужно найти в отсортированной последовательности значение 56. При «наивном» алгоритме мы бы начали с первого

элемента списка, проверили, не равен ли он 56, и перешли к следующему, и так до тех пор, пока не проверили бы все элементы или нашли искомое значение. В наихудшем случае алгоритм проверяет все элементы списка. Отсортированный список из 10 000 значений требует примерно 10 000 операций для проверки всех элементов списка на равенство искомому значению. На языке теории алгоритмов можно сказать, что сложность вычисления *линейна* относительно количества элементов списка. Алгоритм не использует всю доступную информацию, чтобы добиться максимальной эффективности.

Первая часть полезной информации — то, что список отсортирован! С помощью этого факта можно создать алгоритм, которому, чтобы абсолютно точно выяснить, присутствует ли в списке искомый элемент, достаточно проверить лишь несколько элементов списка. Алгоритм бинарного поиска обходит лишь $\log_2(n)$ элементов (логарифм по основанию 2). Для поиска в том же списке из 10 000 элементов достаточно всего лишь $\log_2(10\,000) < 14$ операций.

При бинарном поиске предполагается, что список отсортирован в порядке возрастания. Алгоритм проверяет сначала элемент в середине списка. Если это срединное значение больше искомого, значит, все значения от середины и до последнего элемента списка больше требуемого. Искомое значение не входит в эту половину списка, так что одной операции достаточно, чтобы сразу отбросить половину элементов.

Аналогично, если искомое значение больше срединного, то можно отбросить первую половину элементов списка. А затем эта процедура сокращения вдвое фактического размера списка проверяемых элементов повторяется на каждом шаге алгоритма. На рис. 6.12 приведен наглядный пример.

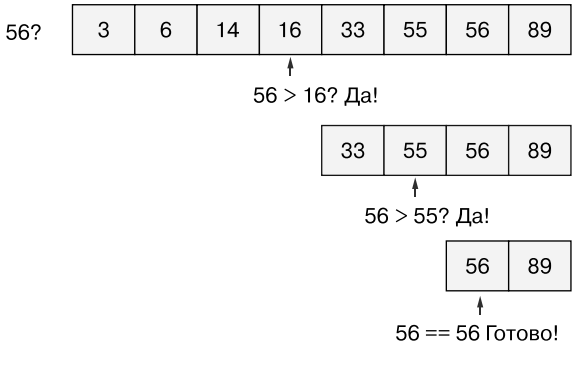


Рис. 6.12. Пример работы алгоритма бинарного поиска

Если подсписок содержит четное количество элементов, то явного срединного элемента не существует. В этом случае мы округляем индекс срединного элемента.

Нам нужно найти значение 56 в отсортированном списке из восьми целочисленных значений, просмотрев при этом как можно меньше элементов. Алгоритм бинарного поиска проверяет расположенный посередине (с учетом округления) элемент `x` и отбрасывает половину списка, в которой 56 заведомо не может быть. У этой проверки могут быть три исхода:

- элемент *x* больше 56. Алгоритм игнорирует правую часть списка;
- элемент *x* меньше значения 56. Алгоритм игнорирует левую часть списка;
- элемент *x* равен 56, как на последней строке рис. 6.12. Поздравляем — мы нашли искомое значение!

В листинге 6.12 показана реализация алгоритма бинарного поиска.

Листинг 6.12. Алгоритм бинарного поиска

```
def binary_search(lst, value):
    lo, hi = 0, len(lst)-1
    while lo <= hi:
        mid = (lo + hi) // 2
        if lst[mid] < value:
            lo = mid + 1
        elif value < lst[mid]:
            hi = mid - 1
        else:
            return mid
    return -1

l = [3, 6, 14, 16, 33, 55, 56, 89]
x = 56
print(binary_search(l,x))
# 6 (индекс найденного элемента)
```

Этот алгоритм получает в качестве аргумента список и значение для поиска, после чего последовательно делит пополам область поиска с помощью двух переменных `lo` и `hi`, задающих интервал элементов списка, в котором может находиться искомое значение: `lo` задает начальный индекс данного интервала, а `hi` — конечный. Мы проверяем, какой из вышеупомянутых случаев имеет место для срединного элемента, и подгоняем интервал поиска соответствующим образом, модифицируя значения `lo` и `hi`.

И хотя это вполне нормальная, удобочитаемая и эффективная реализация алгоритма бинарного поиска, она занимает отнюдь не одну строку!

Код

А теперь мы реализуем алгоритм бинарного поиска в одной строке кода (листинг 6.13)!

Листинг 6.13. Однострочная реализация алгоритма бинарного поиска

```
   ## Данные
   l = [3, 6, 14, 16, 33, 55, 56, 89]
   x = 33

   ## Однострочник
❶ bs = lambda l, x, lo, hi: -1 if lo>hi else \
       ❷ (lo+hi)//2 if l[(lo+hi)//2] == x else \
       ❸ bs(l, x, lo, (lo+hi)//2-1) if l[(lo+hi)//2] > x else \
       ❹ bs(l, x, (lo+hi)//2+1, hi)

   ## Результат
   print(bs(l, x, 0, len(l)-1))
```

Угадайте, какие результаты вернет этот фрагмент кода!

Принцип работы

Благодаря тому, что бинарный поиск естественным образом подходит для реализации рекурсивного подхода, изучение данного однострочника укрепит ваше понимание этого важного понятия теории computer science. Отмечу, что я разбил данное однострочное решение на четыре строки ради удобства чтения, хотя, конечно, вы можете записать его в одной строке кода. В данном однострочнике используется рекурсивный способ реализации алгоритма бинарного поиска.

Мы создаем новую функцию `bs` с помощью оператора `lambda` с четырьмя аргументами: `l`, `x`, `lo` и `hi` ❶. Первые два аргумента `l` и `x` представляют собой переменные, содержащие отсортированный список и значение для поиска. Аргументы `lo` и `hi` задают минимальный и максимальный индекс текущего подсписка, в котором производится поиск значения `x`. На каждом уровне рекурсии код проверяет подсписок, заданный индексами `lo` и `hi`, все уменьшающийся по мере увеличения индекса `lo` и уменьшения индекса `hi`. После конечного количества шагов условие `lo>hi` становится

`True`. Просматриваемый подсписок пуст — и мы не нашли значение `x`. Это граничный случай нашей рекурсии. Поскольку мы не нашли элемент `x`, то возвращаем `-1`, указывая, что искомого элемента не существует.

Для поиска срединного элемента подсписка мы прибегнем к формуле `(lo+hi)//2`. Если он оказывается искомым, то возвращаем его индекс ❷. Обратите внимание, что здесь используется целочисленное деление для округления вниз к ближайшему целочисленному значению, которое можно применять в качестве индекса списка.

Если срединный элемент больше желаемого значения, значит, все элементы справа тоже больше него, поэтому можно произвести рекурсивный вызов функции, но изменить индекс `hi` так, чтобы далее анализировать только элементы списка слева от срединного элемента ❸.

Аналогично, если срединный элемент меньше желаемого значения, то можно не просматривать элементы слева от него, поэтому можно произвести рекурсивный вызов функции, но изменить индекс `lo` так, чтобы далее анализировать только элементы списка справа от срединного элемента ❹.

Поиск значения `33` в списке `[3, 6, 14, 16, 33, 55, 56, 89]` возвращает индекс `4`.

Материал этого раздела должен укрепить ваше общее понимание кода в том, что касается условного выполнения, основных ключевых слов и арифметических операций, а также важного вопроса доступа по индексу к последовательностям программным образом. Но что еще важнее, вы узнали, как упрощать решение сложных задач с помощью рекурсии.

Рекурсивный алгоритм быстрой сортировки

А сейчас мы создадим однострочник для популярного алгоритма *быстрой сортировки* (Quicksort) — алгоритма сортировки, который, как ясно из его названия, быстро сортирует данные.

Общее описание

Быстрая сортировка — частый вопрос на многих собеседованиях (его задают в Google, Facebook и Amazon), а также быстрый, лаконичный и удобочитаемый

алгоритм сортировки. Благодаря изяществу быстрой сортировки о ней рассказывается в большинстве курсов для начинающих.

Быстрая сортировка сортирует список путем рекурсивного разбиения большой задачи на меньшие и объединения решений этих меньших задач так, чтобы получить решение большей.

Для решения каждой из меньших задач используется рекурсивно та же стратегия: меньшие задачи разбиваются на еще меньшие подзадачи, решаемые по отдельности и объединяемые, так что быстрая сортировка относится к алгоритмам, работающим по принципу *разделяй и властвуй*.

При быстрой сортировке выбирается *опорный элемент* (pivot element), а затем все элементы, которые больше него, помещаются справа от него, а все элементы, которые меньше или равны ему, — слева. Таким образом, большая задача сортировки списка разбивается на две меньшие подзадачи сортировки двух меньших списков. А затем эта процедура повторяется рекурсивно до тех пор, пока не получится список из нуля элементов, попытка сортировки которого приводит к завершению рекурсии.

На рис. 6.13 показан алгоритм быстрой сортировки в действии.

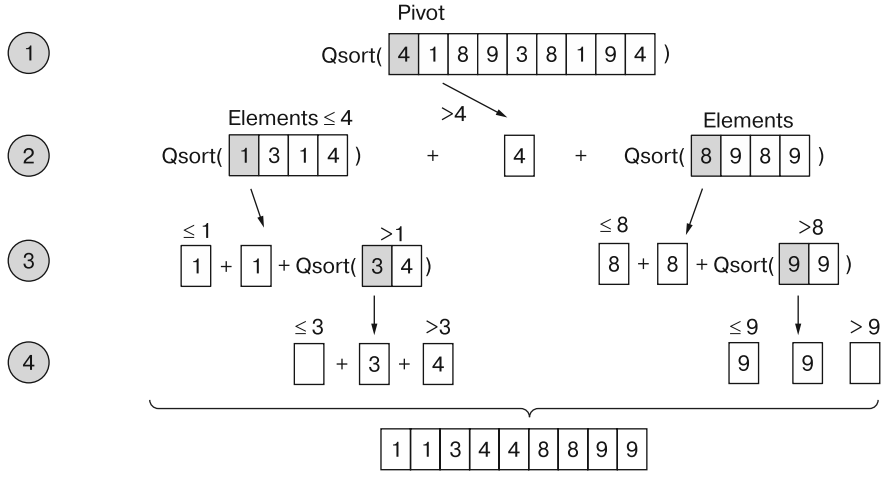


Рис. 6.13. Пример работы алгоритма быстрой сортировки

На рис. 6.13 показано применение алгоритма быстрой сортировки к неотсортированному списку целых чисел [4, 1, 8, 9, 3, 8, 1, 9, 4]. Сначала

он выбирает 4 в качестве опорного элемента, разбивает список на неотсортированный подсписок [1, 3, 1, 4], все элементы которого меньше или равны опорному, и неотсортированный подсписок [8, 9, 8, 9], все элементы которого больше опорного.

Далее алгоритм быстрой сортировки вызывается рекурсивно для сортировки двух неотсортированных подсписков. Как только размер подсписков доходит до не более чем одного элемента, считается, что они отсортированы по определению, и рекурсия завершается.

На каждом шаге рекурсии производится конкатенация трех подсписков (левого, опорного и правого) перед передачей итогового списка на более высокий уровень рекурсии.

Код

Мы создадим функцию `q`, которая реализует алгоритм быстрой сортировки в одной строке кода Python и сортирует любой аргумент, переданный в нее в виде списка целых чисел (листинг 6.14).

Листинг 6.14. Однострочная реализация алгоритма быстрой сортировки с помощью рекурсии

```
## Данные
unsorted = [33, 2, 3, 45, 6, 54, 33]

## Однострочник
q = lambda l: q([x for x in l[1:] if x <= l[0]]) + [l[0]] + q([x for x in l
              if x > l[0]]) if l else []

## Результат
print(q(unsorted))
```

Можете угадать — последний раз — результаты выполнения фрагмента кода?

Принцип работы

Этот однострочник полностью воспроизводит алгоритм, который мы только что обсуждали. Во-первых, мы создаем новую лямбда-функцию `q`, принимающую в качестве аргумента список `l`, который нужно отсортировать. В укрупненном виде структура этой лямбда-функции выглядит следующим образом:

```
lambda l: q(левый) + опорный_элемент + q(правый) if l else []
```

При граничном случае рекурсии — когда список пуст и, следовательно, сортируется тривиальным образом — лямбда-функция возвращает пустой список `[]`.

Во всех прочих случаях функция берет в качестве первого элемента списка `l` *`опорный_элемент`* и делит все элементы на два подсписка (*левый* и *правый*), в зависимости от того, меньше или больше они, чем *`опорный_элемент`*. Для этого мы воспользуемся обычным списковым включением (см. главу 2). А поскольку эти два подсписка, вероятно, не отсортированы, мы рекурсивно вызываем алгоритм быстрой сортировки и для них. Наконец, мы объединяем все три списка и возвращаем итоговый отсортированный список. Результат выглядит следующим образом:

```
## Результат
print(q(unsorted))
# [2, 3, 6, 33, 33, 45, 54]
```

Итоги главы

В этой главе вы изучили алгоритмы, имеющие большое значение в computer science. Они предназначены для решения широкого спектра задач, связанных с анаграммами, палиндромами, булеанами, перестановками, факториалами, простыми числами, числами Фибоначчи, шифрованием, поиском и сортировкой. Многие из них лежат в основе более сложных алгоритмов и входят во все полноценные курсы теории алгоритмов. Совершенствование практических и теоретических знаний алгоритмов — один из самых эффективных способов стать профессиональным программистом. Я бы даже рискнул утверждать, что недостаточное понимание алгоритмов — основная причина того, что большинство посредственных программистов не могут продвинуться дальше в обучении.

Чтобы помочь решить эту проблему, я регулярно рассказываю о различных алгоритмах в своей почтовой рассылке Coffee Break Python («Python за чашкой кофе») (подписаться можно по адресу https://blog.finxter.com/subscribe/). Я высоко ценю то время и усилия, которые вы потратили на изучение всех фрагментов кода однострочников и пояснений к ним, и надеюсь, что вы уже заметили значительный прогресс в своих навыках. По моему опыту обучения тысяч изучающих Python, более половины программистов среднего уровня с трудом понимают простейшие однострочники на языке Python. Немного прилежания и упорства — и вы обойдете их и станете настоящим знатоком Python (или, по крайней мере, войдете в 10 % лучших программистов).

Послесловие

Поздравляю! Вы одолели всю книгу и овладели *однострочниками Python* на уровне, доступном лишь немногим. Вы заложили прочный фундамент, который поможет вам выйти за рамки своих нынешних навыков программирования на Python. Благодаря внимательному изучению всех приведенных однострочников Python вы сможете справиться с любой строкой кода на Python, какую только встретите.

Подобные знания, как и любые другие, следует использовать с умом. Злоупотребление однострочниками может только навредить вашим проектам. В этой книге все алгоритмы сжимались в одну строку кода для того, чтобы вы лучше стали понимать свой код. Но будьте осторожны и *не злоупотребляйте* обретенными навыками в своих реальных проектах. Не старайтесь впихнуть все в одну строку кода просто для того, чтобы похвастаться своими способностями по написанию однострочников.

Вместо этого воспользуйтесь ими, чтобы повысить удобочитаемость существующих баз кода путем распутывания самых сложных однострочников. Подобно тому как Супермен использует свои сверхспособности, чтобы помочь обычным людям жить нормальной жизнью, так и вы можете помочь программистам вести нормальную программистскую жизнь.

Главная задача этой книги — помочь вам овладеть в совершенстве однострочниками Python. Если вы считаете, что она справилась с данной задачей, напишите отзыв о ней, чтобы и другие могли открыть ее для себя. Кроме того, я призываю вас сообщать мне по адресу chris@finxter.com обо всех найденных в этой книге ошибках, а также присылать любые позитивные или негативные замечания и комментарии. Мы планируем непрерывно улучшать эту книгу и учитывать в будущих изданиях обратную связь читателей, так что я обещаю бесплатную копию моей электронной книги Coffee Break Python Slicing за любые конструктивные замечания.

Наконец, если вы стремитесь постоянно развивать свои навыки Python, то подпишитесь на мою рассылку о Python по адресу https://blog.finxter.com/subscribe/, в которой я почти каждый день публикую свежие обучающие материалы по computer science наподобие шпаргалок по Python, открывая тем самым вам и тысячам других программистов путь к непрерывному совершенствованию и, в конечном итоге, овладению всеми секретами Python.

Теперь, научившись разбираться в отдельных строках кода, вы можете переключить внимание на более крупные проекты. Почитайте материалы об объектно-ориентированном программировании и управлении проектами, а главное, найдите для себя реальные проекты, над которыми могли бы непрерывно работать. Это повысит ваши способности к обучению, мотивирует и вдохновит на новые достижения, повысит вашу стоимость на рынке труда, да и вообще это наиболее эффективная форма обучения. Ничто не сравнится с реальной работой в смысле эффективности обучения.

Я рекомендую моим студентам тратить по крайней мере 70 % выделенного на учебу времени на практическую работу над проектами. Если у вас есть 100 минут в день на учебу, то тратьте 70 из них на практическую работу над кодом и лишь 30 — на чтение книг и изучение различных курсов и руководств. Как это ни очевидно, большинство людей делают наоборот и в итоге так никогда и не чувствуют готовности начать работать над реальными проектами.

Мне было очень приятно провести с вами столько времени, и я высоко ценю усилия, приложенные вами к изучению этого руководства. Надеюсь, это окупится сторицей! Всего самого лучшего в вашей карьере программиста, и надеюсь, что мы еще встретимся!

Удачного написания кода!

Крис

Кристиан Майер

Однострочники Python: лаконичный и содержательный код

Перевел с английского *И. Пальти*

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 11.2021. 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 13.10.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 1000. Заказ 0000.

Михал Яворски, Тарек Зиаде

PYTHON. ЛУЧШИЕ ПРАКТИКИ И ИНСТРУМЕНТЫ

Python — это динамический язык программирования, используемый в самых разных предметных областях. Хотя писать код на Python просто, гораздо сложнее сделать этот код удобочитаемым, пригодным для многократного использования и легким в поддержке. Третье издание «Python. Лучшие практики и инструменты» даст вам инструменты для эффективного решения любой задачи разработки и сопровождения софта.

Авторы начинают с рассказа о новых возможностях Python 3.7 и продвинутых аспектах синтаксиса Python. Продолжают советами по реализации популярных парадигм, в том числе объектно-ориентированного, функционального и событийно-ориентированного программирования. Также авторы рассказывают о наилучших практиках именования, о том, какими способами можно автоматизировать развертывание программ на удаленных серверах. Вы узнаете, как создавать полезные расширения для Python на C, C++, Cython и CFFI.

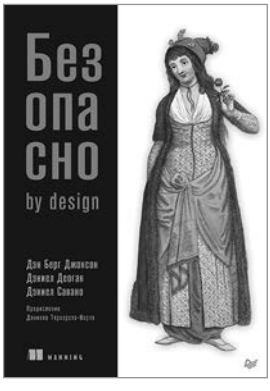
Джастин Зейтц, Тим Арнольд

BLACK HAT PYTHON: ПРОГРАММИРОВАНИЕ ДЛЯ ХАКЕРОВ И ПЕНТЕСТЕРОВ, 2-е изд.

Когда речь идет о создании мощных и эффективных хакерских инструментов, большинство аналитиков по безопасности выбирают Python. Во втором издании бестселлера Black Hat Python вы исследуете темную сторону возможностей Python — все от написания сетевых снифферов, похищения учетных данных электронной почты и брутфорса каталогов до разработки мутационных фаззеров, анализа виртуальных машин и создания скрытых троянов.

Дэн Берг Джонсон, Дэниел Деоган, Дэниел Савано

БЕЗОПАСНО BY DESIGN



«Безопасно by Design» не похожа на другие книги по безопасности. В ней нет дискуссий на такие классические темы, как переполнение буфера или слабые места в криптографических хэш-функциях. Вместо собственно безопасности авторы концентрируются на подходах к разработке ПО. Поначалу это может показаться немного странным, но вы поймете, что недостатки безопасности часто вызваны плохим дизайном. Значительного количества уязвимостей можно избежать, используя передовые методы проектирования. Изучение того, как дизайн программного обеспечения соотносится с безопасностью, является целью этой книги. Вы узнаете, почему дизайн важен для безопасности и как его использовать для создания безопасного программного обеспечения.

Том Лащевски, Камаль Арора, Эрик Фарр, Пийум Зонуз

ОБЛАЧНЫЕ АРХИТЕКТУРЫ: РАЗРАБОТКА УСТОЙЧИВЫХ И ЭКОНОМИЧНЫХ ОБЛАЧНЫХ ПРИЛОЖЕНИЙ

Облачные вычисления — это, пожалуй, наиболее революционная разработка в IT со времен виртуализации. Облачно-ориентированные архитектуры обеспечивают большую гибкость по сравнению с системами предыдущего поколения. В этой книге продемонстрированы три важнейших аспекта развертывания современных cloud native архитектур: организационное преобразование, модернизация развертывания, паттерны облачного проектирования.

Книга начинается с краткого знакомства с облачно-ориентированными архитектурами — на примерах объясняется, какие черты им присущи, а какие нет. Вы узнаете, как организуется внедрение и разработка облачных архитектур с применением микросервисов и бессерверных вычислений как основ проектирования. Далее вы изучите такие столпы облачно-ориентированного проектирования, как масштабируемость, оптимизация издержек, безопасность и способы достижения безупречной эксплуатационной надежности. В заключительных главах будет рассказано о различных общедоступных архитектурах cloud native, — от AWS и Azure до Google Cloud Platform.