PYTHON ДЛЯ DATA SCIENCE

ЮЛИЙ ВАСИЛЬЕВ

PYTHON ДЛЯ DATA SCIENCE

ЮЛИЙ ВАСИЛЬЕВ

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.973.233.02
УДК 004.62
В19

Васильев Юлий

В19 Python для data science. — СПб.: Питер, 2023. — 272 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2392-6

Python — идеальный выбор для манипулирования и извлечения информации из данных всех видов. «Python для data science» познакомит программистов с питоническим миром анализа данных. Вы научитесь писать код на Python, применяя самые современные методы, для получения, преобразования и анализа данных в управлении бизнесом, маркетинге и поддержке принятия решений.

Познакомьтесь с богатым набором встроенных структур данных Python для выполнения основных операций, а также о надежной экосистеме библиотек с открытым исходным кодом для data science, включая NumPy, pandas, scikit-learn, matplotlib и другие. Научитесь загружать данные в различных форматах, упорядочивать, группировать и агрегировать датасеты, а также создавать графики, карты и другие визуализации. На подробных примерах стройте реальные приложения, в том числе службу такси, использующую геолокацию, анализ корзины для определения товаров, которые обычно покупаются вместе, а также модель машинного обучения для прогнозирования цен на акции.

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

ББК 32.973.233.02
УДК 004.62

978-1718502208 англ.

ISBN 978-5-4461-2392-6

Краткое содержание

Оглавление

Об авторе

Юлий Васильев — программист, писатель и консультант по разработке открытого исходного кода, построению структур и моделей данных, а также реализации бэкенда баз данных. Он является автором книги «Natural Language Processing with Python and spaCy»[1] (No Starch Press, 2020).

[1] Васильев Ю. «Обработка естественного языка. Python и spaCy на практике». Санкт-Петербург, издательство «Питер».

О научном редакторе

Даниэль Зингаро (Dr. Daniel Zingaro) — доцент кафедры информатики и заслуженный преподаватель Университета Торонто. Его исследования направлены на то, чтобы улучшить качество изучения студентами компьютерных наук. Он является автором двух вышедших в издательстве No Starch Press книг: первая — «Algorithmic Thinking»[1], 2020, практическое руководство по алгоритмам и структурам данных без математики, и вторая — «Learn to Code by Solving Problems, a Python-based Introduction»[2], 2021, пособие по Python и вычислительному мышлению для начинающих.

[1] Зингаро Д. «Алгоритмы на практике». Санкт-Петербург, издательство «Питер».

[2] Зингаро Д. «Python без проблем: решаем реальные задачи и пишем полезный код». Санкт-Петербург, издательство «Питер».

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.

Введение

Мы живем в мире информационных технологий, где компьютерные системы собирают огромные объемы данных, обрабатывают их и извлекают полезную информацию. Эта реальность, ориентированная на данные, влияет не только на деятельность современного бизнеса, но и на нашу повседневную жизнь. Без многочисленных устройств и систем, которые используют технологии, основанные на данных, многим из нас было бы трудно существовать в социуме. Мобильные карты и навигация, онлайн-шопинг и умные домашние устройства — вот несколько известных примеров применения в повседневной жизни технологий, ориентированных на данные.

В деловой сфере компании часто используют IT-системы для принятия решений, извлекая полезную информацию из больших объемов данных. Эти данные могут поступать из различных источников, в любом формате, и иногда их требуется преобразовать, прежде чем анализировать. Так, например, многие компании, которые ведут бизнес онлайн, используют аналитику данных для привлечения и удержания клиентов, собирая и измеряя все, что только можно. Это позволяет им моделировать и понимать поведение клиентов. Компании часто объединяют и анализируют как количественные, так и качественные данные о пользователях из разных источников: личных профилей, социальных сетей и сайтов организаций. И во многих случаях эти задачи выполняются с помощью языка программирования Python.

Эта книга познакомит вас с питоническим подходом к работе с данными без сложных научных терминов. Вы научитесь использовать Python для приложений, ориентированных на работу с данными, тренируясь писать код для сервиса каршеринга, рекомендаций товаров, прогнозирования тенденций фондового рынка и многого другого. На реальных примерах, перечисленных выше, вы получите практический опыт работы с ключевыми библиотеками Python для data science.

Использование Python для data science

Простой и понятный язык программирования Python идеально подходит для получения и понимания данных любого типа, а также для выполнения с ними различных действий. Он сочетает в себе богатый набор встроенных структур данных для базовых операций и надежную экосистему библиотек с открытым исходным кодом для анализа и работы с данными любого уровня сложности. В этой книге мы рассмотрим множество таких библиотек: NumPy, pandas, scikit-learn, Matplotlib и др.

На языке Python вы сможете писать лаконичный и интуитивно понятный код с минимальными затратами времени и усилий, реализуя большинство идей всего в нескольких строках. На самом деле гибкий синтаксис позволяет реализовать несколько операций с данными даже в одной строке. Например, можно написать строку кода, которая одновременно фильтрует, преобразует и агрегирует данные.

Будучи языком общего назначения, Python подходит для решения широкого круга задач. Работая с этим языком, можно легко интегрировать анализ данных с другими задачами для создания полнофункциональных, хорошо продуманных приложений. Например, можно создать бота, который выдает прогнозы фондового рынка в ответ на запрос пользователя на естественном языке. Чтобы создать такое приложение, понадобится API для бота, прогнозная модель машинного обучения и инструмент обработки естественного языка (NLP) для взаимодействия с пользователями. Для всего этого существуют мощные библиотеки Python.

Для кого эта книга?

Книга предназначена для разработчиков, желающим лучше понять возможности Python по обработке и анализу данных. Возможно, вы работаете в компании, которая хочет использовать данные для улучшения бизнес-процессов, принятия более обоснованных решений и привлечения большего количества покупателей.

Или, может быть, вы хотите создать собственное приложение на основе данных или просто расширить знания о применении Python в области data science.

Книга предполагает, что у вас уже есть базовый опыт работы с Python и для вас не составит труда следовать таким инструкциям, как установка базы данных или получение ключа API. Тем не менее концепции data science объясняются с нуля на практических, тщательно разобранных примерах. Поэтому опыт работы с данными не требуется.

О чем эта книга?

Мы начнем с понятийного введения в обработку и анализ данных и разбора типичного пайплайна обработки данных. Затем рассмотрим встроенные в Python структуры данных и несколько сторонних библиотек, которые широко используются для приложений на основе данных. Далее перейдем к более сложным методам получения, объединения, агрегирования, группировки, анализа и визуализации датасетов разных размеров, содержащих разные типы данных. По ходу изучения книги мы будем применять методы языка Python для работы с данными к реальным ситуациям из мира управления бизнесом, маркетинга и финансов. В каждой главе есть раздел «Упражнения», чтобы вы могли попрактиковаться и закрепить полученные знания.

Вот краткое содержание каждой из глав:

Глава 1. Базовые знания о данных готовит читателя к пониманию основ работы с данными. Вы познакомитесь с различными категориями данных: структурированными, неструктурированными и слабоструктурированными. Затем пройдетесь по стадиям типичного процесса анализа данных.

Глава 2. Структуры данных Python представляет четыре структуры данных, встроенные в Python: списки (lists), словари (dictionaries), кортежи (tuples) и множества (sets). Вы увидите, как использовать каждую из них и объединять в более сложные структуры, которые могут описывать объекты реального мира.

Глава 3. Библиотеки Python для data science рассматривает надежную экосистему сторонних библиотек Python для анализа и операций с данными.

Вы познакомитесь с библиотекой pandas и ее основными структурами данных — Series и DataFrame, которые уже стали стандартом для Python-приложений, ориентированных на работу с данными. Вы также узнаете о двух других библиотеках для data science — NumPy и scikit-learn.

Глава 4. Доступ к данным из файлов и API подробно рассказывает, как получить данные и загрузить их в скрипт. Вы научитесь загружать данные из различных источников, таких как файлы и API, и формировать структуры данных в Python-скриптах для дальнейшей обработки.

Глава 5. Работа с базами данных продолжает обсуждение импортирования данных в Python, рассказывая о том, как работать с информацией из базы данных. Вы изучите примеры получения и обработки данных, хранящихся в базах разных типов: реляционных, например MySQL, и нереляционных (NoSQL), например MongoDB.

Глава 6. Агрегирование данных предлагает в целях обобщения данных выполнять их сортировку по группам и проводить агрегированные вычисления. Вы научитесь использовать pandas для группировки данных и получения промежуточных и итоговых значений, а также прочих возможных совокупностей.

Глава 7. Объединение датасетов рассказывает о том, как объединить данные из разных источников в единый датасет. Вы изучите методы, которые разработчики SQL используют для объединения таблиц баз данных, и примените их к встроенным в Python структурам данных, массивам NumPy и объектам DataFrame библиотеки pandas.

Глава 8. Визуализация посвящена наглядному отображению как наиболее естественному способу выявления скрытых закономерностей в данных. Вы узнаете о различных типах визуализации, таких как линейные графики, столбчатые диаграммы и гистограммы, и научитесь создавать их с помощью Matplotlib, главной библиотеки Python для построения графиков. А для создания карт будем использовать библиотеку Cartopy.

Глава 9. Анализ данных о местоположении объясняет, как работать с данными о местоположении с помощью библиотек geopy и Shapely. Вы узнаете о способах получения и использования GPS-координат как стационарных, так и движущихся объектов. Также вы изучите реальный пример того, как сервис каршеринга определяет ближайший к заданной точке автомобиль.

Глава 10. Анализ данных временных рядов представляет несколько методов анализа, которые можно применить к временным рядам для извлечения значимых статистических данных. В частности, примеры в этой главе иллюстрируют, как подобный анализ применим к данным фондового рынка.

Глава 11. Получение инсайтов из данных изучает стратегии получения полезной информации из данных для принятия обоснованных решений.

Например, вы узнаете, как обнаружить связи между товарами, продаваемыми в супермаркете, и определить, какие группы товаров часто покупаются в одном чеке (полезно для рекомендаций и рекламных акций).

Глава 12. Машинное обучение для анализа данных рассматривает использование библиотеки scikit-learn для продвинутых задач анализа данных. Вы научите модели машинного обучения классифицировать отзывы о товарах по их рейтингу и предсказывать тенденции в цене акций.

1

Базовые знания о данных

Для разных людей *данные* имеют разное значение: для биржевого маклера данные — это котировки акций в реальном времени, а для инженера NASA — сигналы, поступающие от марсохода. Однако когда дело касается обработки и анализа данных, к различным датасетам, независимо от их происхождения, могут применяться одинаковые или похожие подходы и методы, поскольку для этих целей важна лишь структура данных.

В этой главе объясняются основные понятия сферы обработки и анализа данных. Прежде всего, мы разберем основные категории данных, с которыми вам, скорее всего, придется иметь дело, а затем коснемся распространенных источников данных. Далее мы изучим этапы типичного пайплайна обработки данных (то есть фактического процесса получения, подготовки и анализа данных). И наконец, рассмотрим уникальные преимущества Python как инструмента работы с данными.

Категории данных

Программисты разделяют данные на три основные категории: неструктурированные, структурированные и слабоструктурированные. В пайплайне обработки исходные данные обычно являются неструктурированными; из них формируются структурированные или слабоструктурированные датасеты для дальнейшей обработки. Однако в некоторых пайплайнах сразу используются структурированные данные. К примеру, приложения, работающие с геолокацией,

могут получать структурированные данные непосредственно с датчиков GPS. В следующих разделах мы подробно рассмотрим эти три основные категории данных, а также данные временных рядов — особый тип данных, которые могут быть структурированными или слабоструктурированными.

Неструктурированные данные

Неструктурированные данные — это данные, не имеющие предопределенной организационной системы или схемы. Это наиболее распространенный вид данных, к нему относятся изображения, видео, аудио и текст на естественном языке. Для примера рассмотрим следующий финансовый отчет фармацевтической компании:

```
GoodComp shares soared as much as 8.2% on 2021-01-07 after the company announced
positive early-stage trial results for its vaccine.[1]
```

Этот текст относится к неструктурированному типу данных, поскольку содержащаяся в нем информация не организована по заранее определенной схеме. Вместо этого информация разбросана хаотично. Предложение можно переписать как угодно, при этом содержащаяся в нем информация останется прежней. Например:

```
Following the January 7, 2021, release of positive results from its vaccine
trial, which is still in its early stages, shares in GoodComp rose by 8.2%.
```

Несмотря на отсутствие структуры, неструктурированные данные могут содержать важную информацию, которую можно извлечь и преобразовать в структурированные или слабоструктурированные данные с помощью соответствующих методов обработки и анализа. Например, инструменты распознавания изображений сначала преобразуют набор пикселей изображения в датасет заранее определенного формата, а затем анализируют эти данные для идентификации содержимого. В следующем разделе показано несколько аналогичных способов структурирования данных, извлеченных из финансового отчета.

Структурированные данные

Структурированные данные организуются в заранее определенном формате. Такие данные обычно расположены в хранилище, например в реляционной базе данных или просто в файле *.csv* (comma-separated values — файл данных

[1] Акции GoodComp взлетели на 8.2% 07.01.2021 г. после того, как компания объявила о положительных результатах первого этапа испытаний своей вакцины.

с разделителями-запятыми). Данные, содержащиеся в таком хранилище, называются *записью*, а информация в записи организована в *полях*, которые должны поступать в последовательности, соответствующей ожидаемой структуре. В базе данных записи одинаковой структуры логически группируются в контейнер, называемый *таблицей*. База данных может содержать различные таблицы, и каждая из них имеет определенную структуру полей.

Существует два основных типа структурированных данных: числовые и категориальные. *Категориальные данные* — это данные, которые можно разделить по категориям на основе сходных характеристик; например, автомобили можно категоризировать по маркам и моделям. А *числовые данные* представляют информацию в числовой форме и позволяют выполнять над собой математические операции.

Следует отметить, что категориальные данные иногда могут принимать числовые значения. Возьмем, к примеру, почтовые индексы или телефонные номера. Хотя эти данные представлены в числовом выражении, выполнять над ними математические операции, например находить медианный почтовый индекс или среднее значение номеров телефонов, не имеет смысла.

Как преобразовать текст, представленный в примере из предыдущего раздела, в структурированные данные? Нас интересует конкретная информация из текста, например названия компаний, даты и цены на акции. Необходимо соответствующим образом форматировать эту информацию и разместить ее в полях, готовых для вставки в базу данных:

```
Компания:    ABC
Дата:        yyyy-mm-dd
Акция:       nnnnn
```

Используя методы *обработки естественного языка* (NLP, natural language processing), дисциплины, которая обучает машины понимать человекочитаемый текст, можно извлечь информацию для этих полей. Так, название компании можно найти, распознавая категориальную переменную данных, принимающую одно из заданных значений, например Google, Apple или GoodComp. Аналогично можно распознать дату, сопоставив последовательность ее символов с одним из возможных форматов даты, например `yyyy-mm-dd`. В нашем примере мы распознаем, извлекаем и представляем данные в заранее определенном формате следующим образом:

```
Компания:    GoodComp
Дата:        2021-01-07
Акция:       +8.2%
```

Чтобы сохранить эту запись в базе данных, лучше представить ее в виде строки, состоящей из последовательности полей. Таким образом ее можно реорганизовать в виде прямоугольного объекта данных, или двумерной матрицы:

```
Компания | Дата       | Акция
--------------------------
GoodComp |2021-01-07 | +8.2%
```

Из одного и того же источника неструктурированных данных при необходимости можно извлекать разную информацию. Наш пример не только содержит информацию об изменении стоимости акций GoodComp на определенную дату, но и указывает причину этого изменения, выраженную фразой «the company announced positive early-stage trial results for its vaccine[1]». Рассматривая предложение с этой точки зрения, можно создать запись с такими полями:

```
Компания:   GoodComp
Дата:       2021-01-07
Продукт:    vaccine
Стадия:     early-stage trial
```

Сравните с первой извлеченной записью:

```
Компания:   GoodComp
Дата:       2021-01-07
Акция:      +8.2%
```

Обратите внимание, что эти две записи содержат разные поля и потому имеют разные структуры. По этой причине они должны храниться в двух разных таблицах базы данных.

Слабоструктурированные данные

В случаях, когда структура информации не соответствует строгим требованиям форматирования, может понадобиться формат слабоструктурированных данных, позволяющий хранить записи различных структур в одном контейнере (таблице базы данных или документе). Как и неструктурированные данные, слабоструктурированные данные не привязаны к заданной схеме организации. Однако экземпляры слабоструктурированных данных, в отличие от неструктурированных,

[1] Компания объявила о положительных результатах первого этапа испытаний своей вакцины.

демонстрируют определенную структуру, которая обычно выражена в виде самоописываемых тегов или других маркеров.

Самый известный пример слабоструктурированного формата данных — XML и JSON. Вот как может выглядеть финансовый отчет в формате JSON:

```
{
    "Компания": "GoodComp",
    "Дата":     "2021-01-07",
    "Акция":    8.2,
    "Детали":   "the company announced positive early-stage trial results for
its
vaccine."
}
```

Здесь мы видим ту же ключевую информацию, которую извлекли ранее. Каждый фрагмент данных сопровождается описательным тегом, например `"Компания"` или `"Дата"`. Благодаря этим тегам информация организуется аналогично тому, как она была представлена в предыдущем разделе, однако теперь у нас есть четвертый тег, `"Детали"`, которым помечена неструктурированная часть исходного высказывания. Из примера видно, что в рамках одной записи в формате слабоструктурированных данных могут содержаться и структурированные, и неструктурированные фрагменты.

Более того, в один контейнер можно поместить несколько записей различной структуры. Например, две разные записи, полученные из финансового отчета, хранятся в одном документе JSON:

```
[
    {
        "Компания": "GoodComp",
        "Дата":     "2021-01-07",
        "Акция":    8.2
    },
    {
        "Компания": "GoodComp",
        "Дата":     "2021-01-07",
        "Продукт":  "vaccine",
        "Этап":     "early-stage trial"
    }
]
```

Помните, что в предыдущем разделе мы обсуждали реляционную базу данных, которая, будучи жестко структурированным хранилищем, не допускает размещения записей разной структуры в одной таблице.

Данные временных рядов

Временны́е ряды — это множество точек данных, индексированных или перечисленных в порядке от самой ранней до самой поздней. Многие датасеты с финансовыми данными хранятся в виде временных рядов, поскольку обычно включают наблюдения в конкретное время.

Данные временных рядов могут быть как структурированными, так и слабоструктурированными. Представьте, что через равные промежутки времени вы получаете данные о местоположении такси в виде записей от GPS-трекера. Такие данные могут иметь следующий формат:

```
[
    {
        "машина":     "cab_238",
        "координаты": (43.602508,39.715685),
        "время":      "14:47",
        "состояние":  "доступен"
    },
    {
        "машина":     "cab_238",
        "координаты": (43.613744,39.705718),
        "время":      "14:48",
        "состояние":  "доступен"
    }
    ...
]
```

Каждую минуту от машины `cab_238` поступает новая запись, содержащая последние координаты местоположения (широта/долгота). Каждая запись имеет одинаковую последовательность полей, и каждое поле имеет последовательную структуру от одной записи к следующей, что позволяет хранить временные ряды в таблице реляционной базы данных как обычные структурированные данные.

А теперь предположим, что данные поступают через разные промежутки времени (что часто бывает на практике) и за минуту вы получаете несколько наборов координат. Тогда структура может выглядеть так:

```
[
    {
        "машина": "cab_238",
        "координаты": [(43.602508,39.715685),(43.602402,39.709672)],
        "время": "14:47",
        "состояние": "доступен"
```

```
    },
    {
        "машина": "cab_238",
        "координаты": (43.613744,39.705718),
        "время": "14:48",
        "состояние": "доступен"
    }
]
```

Обратите внимание, что первое поле `"координаты"` состоит из двух пар значений и, таким образом, отличается от второго поля `"координаты"`. Такие данные являются слабоструктурированными.

Источники данных

Теперь, когда вы узнали об основных категориях данных, сможете ли вы назвать источники, из которых их можно получить? Вообще говоря, данные могут поступать из различных источников: текстов, видео, изображений, датчиков устройств и т. д. Однако с точки зрения Python-скриптов, которые вам предстоит писать, наиболее привычные — это:

- интерфейс прикладного программирования (API);
- веб-страница;
- база данных;
- файл.

Этот список не является ни исчерпывающим, ни строгим; существует множество других источников данных. В главе 9, к примеру, вы увидите, как использовать смартфон в качестве источника GPS-данных для пайплайна обработки, в частности, применяя бота как промежуточное звено между смартфоном и Python-скриптом.

Технически все перечисленные варианты требуют использования соответствующих библиотек Python. Так, например, прежде чем получить данные из API, нужно установить обертку Python для API или использовать библиотеку Python Requests для выполнения HTTP-запросов к API напрямую. Аналогично, чтобы получить доступ к данным из базы, необходимо установить коннектор из Python-кода, который позволит подключиться к базам данных конкретного типа.

Многие из этих библиотек необходимо загружать и устанавливать, однако некоторые из них поставляются с Python по умолчанию. Например, чтобы

загрузить данные из файла JSON, можно воспользоваться встроенным пакетом Python `json`.

В главах 4 и 5 мы более подробно рассмотрим процесс получения данных. В частности, вы узнаете, как преобразовывать конкретную информацию из различных источников в структуры данных Python-скрипта для дальнейшей обработки. А пока кратко рассмотрим каждый источник из списка выше.

API

API (программный посредник, позволяющий двум приложениям взаимодействовать друг с другом) на сегодняшний день, вероятно, самый известный способ получения данных. Как уже упоминалось, чтобы воспользоваться преимуществами API в Python, понадобится установить обертку в виде библиотеки Python. Чаще всего это делается с помощью команды `pip`.

Не все API имеют собственную обертку для Python, но это не значит, что не получится обратиться к ним из кода. Если API обслуживает HTTP-запросы, с ним можно взаимодействовать с помощью библиотеки Requests. Это открывает доступ к тысячам API, которые можно использовать в коде, запрашивая датасеты и затем обрабатывая их.

При выборе API для конкретной задачи следует обращать внимание на такие факторы:

Функциональность. Многие API предоставляют схожие функции, поэтому важно как можно точнее определиться с требованиями. Например, многие API позволяют осуществлять поиск в интернете из скрипта, но лишь в некоторых предусмотрена функция выбора результатов поиска по дате публикации.

Стоимость. Многие API позволяют использовать так называемый *ключ разработчика*. Обычно он предоставляется бесплатно, однако может иметь определенные ограничения, например, по количеству вызовов в день.

Стабильность. Благодаря репозиторию Python Package Index (PyPI)[1] любой желающий может упаковать API в пакет `pip` и сделать его общедоступным. По этой причине API (или даже несколько API) существует почти для любой задачи, которую только можно представить, но не все из этих API надежны. К счастью, репозиторий PyPI отслеживает производительность и использование пакетов.

[1] https://pypi.org

Документация. Популярные API обычно имеют соответствующий сайт с документацией, на котором можно изучить все команды и примеры использования. Хороший образец — страница с документацией для API Nasdaq Data Link (он же Quandl)[1], где доступны примеры выполнения различных запросов временных рядов.

Большинство API возвращают результаты в одном из следующих трех форматов: JSON, XML или CSV. Данные в любом из этих форматов легко преобразуются в структуры данных, которые либо встроены в Python, либо часто в нем используются. Yahoo Finance API, например, извлекает и анализирует данные фондового рынка, а затем возвращает информацию, уже переведенную в тип pandas DataFrame, повсеместно используемую структуру, которую мы обсудим в главе 3.

Веб-страницы

Веб-страницы могут быть статичными или создаваться «на лету» в ответ на действия пользователя, и в таком случае они, вероятно, будут содержать информацию из множества источников. В обоих случаях программа может читать веб-страницу и извлекать ее части. Это называется *веб-скрейпингом* (web scraping). Такая операция вполне законна, если страница находится в открытом доступе.

Типичный алгоритм скрейпинга в Python требует наличия двух библиотек: Requests и BeautifulSoup. Requests получает исходный код страницы, а BeautifulSoup создает *дерево разбора* (parse tree) для страницы. Такое дерево является иерархическим представлением содержимого страницы. По нему можно выполнять поиск и извлекать информацию, используя стандартный синтаксис Python. К примеру, следующий фрагмент дерева разбора:

```
[<td title="03/01/2020 00:00:00"><a href="Download.aspx?ID=630751"
id="lnkDownload630751"
 target="_blank">03/01/2020</a></td>,
<td title="03/01/2020 00:00:00"><a href="Download.aspx?ID=630753"
id="lnkDownload630753"
 target="_blank">03/01/2020</a></td>,
<td title="03/01/2020 00:00:00"><a href="Download.aspx?ID=630755"
id="lnkDownload630755"
 target="_blank">03/01/2020</a></td>]
```

[1] https://docs.data.nasdaq.com/docs/python-time-series

можно легко преобразовать в список элементов, используя в Python-скрипте цикл `for`:

```
[
    {'Document_Reference': '630751', 'Document_Date': '03/01/2020',
     'link': 'http://www.dummy.com/Download.aspx?ID=630751'}
    {'Document_Reference': '630753', 'Document_Date': '03/01/2020',
     'link': 'http://www.dummy.com/Download.aspx?ID=630753'}
    {'Document_Reference': '630755', 'Document_Date': '03/01/2020',
     'link': 'http://www.dummy.com/Download.aspx?ID=630755'}
]
```

Это пример преобразования слабоструктурированных данных в структурированные.

Базы данных

Еще один популярный источник данных — реляционная база данных (БД), обеспечивающая эффективное хранение и обработку структурированных данных, а также простой и удобный доступ к ним. Получение данных или отправка их в таблицы осуществляется с помощью запросов на языке SQL (Structured Query Language — язык структурированных запросов). Например, следующий запрос к таблице `employees` базы данных извлекает список только тех программистов, которые работают в IT-отделе. Таким образом, не нужно получать таблицу целиком:

```
SELECT first_name, last_name FROM employees WHERE department = 'IT' and title =
'programmer'
```

В Python есть встроенный движок базы данных, SQLite. В качестве альтернативы можно использовать любую другую БД. Для доступа к базе данных необходимо установить клиент БД в своей среде.

Помимо обычных жестко структурированных баз данных, в последние годы все чаще возникает потребность в хранении неоднородных и неструктурированных данных в контейнерах, подобных базам данных, что привело к появлению так называемых *NoSQL* (*non-SQL* или *not only SQL*) баз данных. БД NoSQL используют гибкие модели данных, позволяя хранить большие объемы неструктурированной информации с помощью метода *«ключ — значение»*, когда доступ к каждому фрагменту данных осуществляется с помощью связанного ключа. Вот как мог бы выглядеть наш финансовый отчет, если бы он хранился в базе данных NoSQL:

```
Key   value
---   -----

...

26   GoodComp shares soared as much as 8.2% on 2021-01-07 after the company
announced ...
```

Все высказывание сопряжено с ключом 26. Может показаться странным хранить отчет целиком в БД. Вспомним, однако, что из одного высказывания можно извлечь несколько записей. Хранение целого отчета в дальнейшем обеспечит гибкость извлечения частей информации.

Файлы

Файлы могут содержать структурированные, слабоструктурированные и неструктурированные данные. Встроенная в Python функция `open()` позволяет открыть файл и использовать его данные в скрипте. Однако в зависимости от формата данных (например, CSV, JSON или XML) может потребоваться импортировать соответствующую библиотеку, чтобы иметь возможность выполнять операции чтения, записи и/или добавления данных.

Файлам с обычным текстом не нужна библиотека для дальнейшей обработки, они просто рассматриваются в Python как последовательности строк. Возьмем в качестве примера следующее сообщение, которое маршрутизатор Cisco мог бы отправить в файл журнала:

```
dat= 'Jul 19 10:30:37'
host='sm1-prt-highw157'
syslogtag='%SYS-1-CPURISINGTHRESHOLD:'
msg=' Threshold: Total CPU Utilization(Total/Intr): 17%/1%,
          Top 3 processes(Pid/Util):   85/9%, 146/4%, 80/1%'
```

Можно читать сообщение построчно, вычленяя необходимую информацию. Так, если ваша задача — найти сообщения, содержащие информацию о загрузке процессора, и извлечь из них определенные показатели, ваш скрипт должен распознать последнюю строку фрагмента как сообщение, которое нужно выбрать.

В главе 2 мы рассмотрим пример извлечения конкретной информации из текстовых данных с помощью методов обработки текста.

Пайплайн обработки данных

В этом разделе мы рассмотрим этапы обработки данных, также известные как пайплайн обработки данных (data processing pipeline). Вот привычный алгоритм действий с данными:

1. Получение.
2. Очистка.
3. Преобразование.
4. Анализ.
5. Хранение.

Однако вы увидите, что он соблюдается не всегда. В некоторых приложениях можно объединять несколько этапов в один или вообще пропускать некоторые из них.

Получение

Прежде чем что-то делать с данными, их необходимо получить. Именно поэтому первый шаг в любом пайплайне обработки — сбор данных. В предыдущем разделе вы узнали о распространенных типах источников данных. Некоторые из них позволяют загружать конкретную часть данных в соответствии с запросом.

Например, запрос к API Yahoo Finance требует указать тикер компании и период времени, за который необходимо получить данные о ценах на акции. Аналогично News API, который позволяет извлекать новостные статьи, может обрабатывать ряд параметров для сужения списка запрашиваемых статей, включая источник и дату публикации. Однако несмотря на эти параметры, полученный список может потребовать дополнительной фильтрации. Это означает, что необходима очистка.

Очистка

Очистка данных — это процесс обнаружения и исправления поврежденных/неточных данных или удаления ненужных. В некоторых случаях этот шаг необязателен и получаемые данные сразу готовы к анализу. Например, библиотека `yfinance` (обертка Python для Yahoo Finance API) возвращает данные об акциях в виде удобного объекта pandas DataFrame, что позволяет пропустить этапы очистки и преобразования и сразу перейти к анализу данных. Однако если ваш инструмент сбора данных — веб-скрейпер (web scraper), данные, безусловно, будут нуждаться в очистке, поскольку среди полезных данных, скорее всего, будут присутствовать фрагменты HTML-разметки:

```
6.\t3астройщик должен удовлетворить следующие требования отдела водоотведения
DCC.\x80\x99 следующим образом\r\n\r\n\r\n
```

После очистки этот фрагмент текста будет выглядеть так:

```
6. Застройщик должен удовлетворить следующие требования отдела водоотведения DCC.
```

Помимо HTML-разметки, собранный скрейпером текст может содержать и другую ненужную информацию, как в следующем примере, где фраза `Просмотреть полный текст` является просто текстом гиперссылки. Чтобы получить доступ к тексту, может понадобиться открыть эту ссылку:

```
Разрешение на предлагаемые поправки к разрешению на строительство, полученное
30го Просмотреть полный текст
```

Этап очистки данных также можно использовать для выделения определенных сущностей. К примеру, после запроса выборки статей из News API может потребоваться отобрать из них только статьи за указанный период, в заголовках которых присутствует упоминание о денежной сумме или процентах. Такую фильтрацию можно считать операцией очистки, поскольку ее целью является удаление ненужных данных и подготовка к операциям преобразования и анализа данных.

Преобразование

Преобразование данных — это процесс изменения формата или структуры данных при подготовке к анализу. Например, чтобы извлечь информацию из неструктурированных текстовых данных о GoodComp, как мы делали в разделе «Структурированные данные», можно разбить текст на отдельные слова, или *лексемы*, чтобы инструмент распознавания именованных сущностей (NER, named entity recognition) мог искать нужную информацию. При извлечении информации *именованная сущность* обычно представляет собой объект реального мира, такой как человек, организация или продукт, то есть имя существительное. Существуют также именованные сущности, отражающие даты, проценты, финансовые термины и многое другое.

Большинство инструментов NLP может выполнять это преобразование автоматически. После такой «нарезки» данные GoodComp будут выглядеть так:

```
['GoodComp', 'shares', 'soared', 'as', 'much', 'as', '8.2%', 'on',
 '2021-01-07', 'after', 'the', 'company', 'announced', 'positive',
 'early-stage', 'trial', 'results', 'for', 'its', 'vaccine']
```

Еще одно, более глубокое преобразование — превращение текстовых данных в числовые. Например, коллекцию новостных статей можно преобразовать, выполнив *анализ тональности*, или *сентимент-анализ*, — метод обработки текста, который генерирует число, представляющее эмоции, выраженные в тексте.

Анализ тональности можно проводить с помощью такого инструмента, как `SentimentAnalyzer` из пакета `nltk.sentiment`. Типичный результат анализа может выглядеть следующим образом:

```
Тональность   URL
-----------   ------------------------------------------------------------
0.9313        https://mashable.com/uk/shopping/amazon-face-mask-store-july-28/
0.9387        https://skillet.lifehacker.com/save-those-crustacean-shells-to
                    -make-a-sauce-base-1844520024
```

Каждая запись в датасете теперь содержит число, например `0.9313`, представляющее эмоциональную окраску соответствующей статьи. Когда настроение каждой статьи выражено численно, можно рассчитать среднее значение тональности по всему датасету, что позволяет определить общую окраску интересующего объекта, например компании или продукта.

Анализ

Анализ — основной этап пайплайна обработки данных — это интерпретация исходных данных, позволяющая сделать выводы, которые очевидны не сразу.

В сценарии с сентимент-анализом, возможно, возникнет необходимость изучить, как менялось отношение к компании за определенный период времени, в зависимости от цены ее акций. Также можно сравнить показатели биржевого индекса, например S&P 500, с тональностью, выраженной в широкой подборке новостных статей за тот же период. Следующий фрагмент иллюстрирует, как может выглядеть датасет, где данные S&P 500 представлены вместе с общей тональностью новостей того же дня:

```
Дата          Тональность    S&P_500
-------------------------------------
2021-04-16    0.281074       4185.47
2021-04-19    0.284052       4163.26
2021-04-20    0.262421       4134.94
```

Поскольку и показатели тональности, и показатели акций выражены в числовом формате, можно построить два соответствующих графика на одном рисунке для визуального анализа, как показано на рис. 1.1.

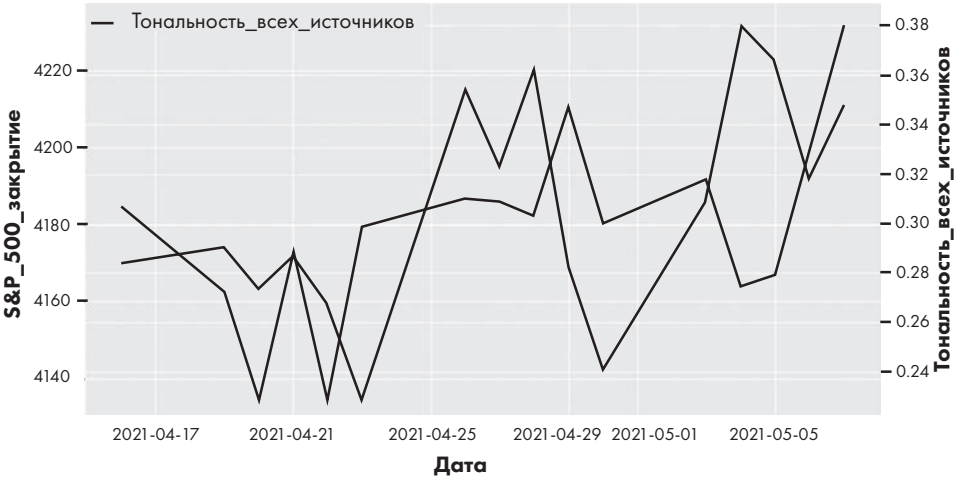

Рис. 1.1. Пример визуального анализа данных

Визуальный анализ — один из наиболее часто используемых и эффективных методов интерпретации данных. Мы подробнее разберем его в главе 8.

Хранение

В большинстве случаев результаты, полученные в процессе анализа данных, необходимо хранить для дальнейшего использования. Обычно есть два варианта хранения: файлы и базы данных. Последний предпочтителен, если предполагается, что данные будут использоваться часто.

Питонический стиль

Предполагается, что при работе с данными на Python вы будете писать код в *питоническом стиле*, то есть лаконично и продуктивно. Например, в питоническом коде часто используются *списковые включения* (list comprehensions) — метод реализации полезных функций обработки данных в одной строке кода.

Более подробно мы рассмотрим списковые включения в главе 2, а пока — краткий пример того, как концепция Python работает на практике. Допустим, необходимо обработать фрагмент текста, состоящий из нескольких предложений:

```
txt = ''' Eight dollars a week or a million a year - what is the difference?
A mathematician or a wit would give you the wrong answer. The magi brought
valuable gifts, but that was not among them. - The Gift of the Magi, O'Henry'''
```

В частности, необходимо разделить текст на предложения, создать список слов каждого из них и исключить знаки препинания. Благодаря функционалу списковых включений Python, все это можно реализовать одной строкой кода, так называемым однострочником (one-liner):

```
word_lists = [[w.replace(',','') ❶ for w in line.split() if w not in ['-']]
            ❷ for line in txt.replace('?','.').split('.')]
```

Цикл `for line in txt` ❷ разбивает текст на предложения и сохраняет их в список. Затем цикл `for w in line` ❶ разбивает каждое предложение на отдельные слова и сохраняет их в список внутри большого списка. В результате получается следующий список списков:

```
[['Eight', 'dollars', 'a', 'week', 'or', 'a', 'million', 'a', 'year', 'what',
  'is', 'the',      'difference'], ['A', 'mathematician', 'or', 'a', 'wit',
  'would', 'give', 'you', 'the', 'wrong', 'answer'], ['The', 'magi',
  'brought', 'valuable', 'gifts', 'but', 'that', 'was', 'not', 'among',
  'them'], ['The', 'Gift', 'of', 'the', 'Magi', "O'Henry"]]
```

Здесь в одной строке кода выполняется два этапа пайплайна обработки данных: очистка и преобразование. Мы очистили текст, удалив из него знаки препинания, и преобразовали, отделив слова друг от друга и сформировав список слов каждого предложения.

Если вы перешли на Python с другого языка программирования, попробуйте реализовать эту же задачу на другом языке. Сколько строк займет такой код?

Выводы

После прочтения этой главы вы должны понимать, какие основные категории данных существуют, откуда они берутся и как организован типичный пайплайн обработки данных.

Как вы увидели, существует три основные категории данных: неструктурированные, структурированные и слабоструктурированные. Исходным материалом в пайплайне обработки данных обычно являются неструктурированные данные, которые становятся готовыми к анализу, проходя этапы очистки и преобразования в структурированные или слабоструктурированные данные. Вы также узнали о пайплайнах обработки исходно структурированных или слабоструктурированных данных, полученных из API или реляционных БД.

2

Структуры данных Python

Структуры данных организуют и хранят информацию, упрощая к ней доступ. В Python встроены четыре структуры данных: списки, кортежи, словари и множества.

С этими структурами легко работать, при этом их можно использовать для выполнения сложных операций с данными, что делает Python одним из самых популярных языков для анализа данных.

В этой главе мы рассмотрим четыре встроенных типа данных Python и уделим особое внимание возможностям, которые позволяют с минимумом кода создавать функциональные приложения, ориентированные на данные. Вы также узнаете, как объединять базовые структуры в более сложные, например в список словарей, чтобы точнее представить объекты реального мира. Кроме того, вы попробуете применить полученные знания для обработки естественного языка и фотографий.

Списки

Список Python — это упорядоченная коллекция объектов. Элементы в списке разделяются запятыми, а сам список заключается в квадратные скобки:

```
[2,4,7]
['Bob', 'John', 'Will']
```

Список изменяем, то есть в него можно добавлять элементы, удалять их и изменять. В отличие от множеств, которые мы обсудим позже, списки могут содержать повторяющиеся элементы.

Списки содержат последовательность элементов, которые можно логически сгруппировать по общему признаку. Эта структура данных, как правило, содержит элементы одной категории (то есть однородные данные, такие как имена людей, названия статей или номера участников). Очень важно понимать эту особенность, выбирая подходящий инструмент для выполнения поставленной задачи. Если вам нужна структура, содержащая объекты с разными свойствами, используйте кортеж или словарь.

ПРИМЕЧАНИЕ

Хотя списки обычно считаются однородными, Python допускает наличие в них элементов разных типов. Этот список, например, включает как строки, так и целые числа:

```
['Ford', 'Mustang', 1964]
```

Создание списка

Чтобы создать простой список, поместите последовательность элементов внутрь квадратных скобок и присвойте этой последовательности имя:

```
regions = ['Asia', 'America', 'Europe']
```

На практике, однако, списки обычно заполняются с нуля динамически, часто с помощью цикла, который вычисляет один элемент за итерацию. В таких случаях первым шагом будет создание пустого списка:

```
regions = []
```

Теперь можно добавлять, удалять и сортировать элементы в созданном списке. Для этих (и некоторых других) целей в Python используются методы списков.

Использование общих методов списков

Методы списков — это функции, которые реализуют определенное поведение внутри списка. В этом разделе мы рассмотрим некоторые общие методы, включая

`append()`, `index()`, `insert()` и `count()`. Начнем практику с создания пустого списка. Попробуем создать список домашних обязанностей: шаг за шагом мы будем наполнять список делами и упорядочивать его:

```
my_list = []
```

`append()`, пожалуй, самый популярный метод списков. Он добавляет элемент в конец последовательности. Можно использовать `append()` для добавления обязанностей по дому в список:

```
my_list.append('Pay bills')
my_list.append('Tidy up')
my_list.append('Walk the dog')
my_list.append('Cook dinner')
```

Теперь список содержит четыре элемента в том порядке, в каком они были добавлены:

```
['Pay bills', 'Tidy up', 'Walk the dog', 'Cook dinner']
```

Каждый элемент списка имеет числовой ключ, известный как *индекс*. Он позволяет сохранять элементы списка в определенном порядке. В Python используется нулевая индексация, то есть начальному элементу последовательности присваивается индекс 0.

Чтобы получить доступ к конкретному элементу списка, укажите название списка, а затем индекс этого элемента в квадратных скобках. Например, с помощью кода ниже можно вывести на экран первый пункт списка дел:

```
print(my_list[0])
```

Функция `print()` выведет:

```
Pay bills
```

Индексы можно использовать не только для доступа к нужному элементу, но и для вставки нового элемента в заданное место. Скажем, необходимо добавить в список новую обязанность между выгулом собаки (walk the dog) и приготовлением ужина (cook dinner). Для этого сначала используем метод `index()`, чтобы

определить индекс элемента, перед которым нужно вставить новый. Сохраним его в переменной `i`:

```
i = my_list.index('Cook dinner')
```

Это значение будет индексом нового элемента. Добавим этот элемент с помощью метода `insert()`:

```
my_list.insert(i, 'Go to the pharmacy')
```

Новая обязанность добавляется в список по указанному индексу, при этом индексы всех последующих элементов увеличиваются на один. Обновленный список будет выглядеть так:

```
['Pay bills', 'Tidy up', 'Walk the dog', 'Go to the pharmacy', 'Cook dinner']
```

Поскольку списки допускают повтор элементов, может понадобиться проверка того, сколько раз встречается тот или иной элемент. Это можно сделать с помощью метода `count()`, как в следующем примере:

```
print(my_list.count('Tidy up'))
```

Функция `print()` выявляет в списке только один экземпляр `'Tidy up'`(делать уборку). Хотя, возможно, стоило бы почаще включать этот пункт в список дел!

ПРИМЕЧАНИЕ

Список всех методов списка можно найти в документации Python[1].

Использование срезов

Получить доступ к диапазону элементов последовательности, например списка, можно с помощью нотации *среза*. Чтобы получить фрагмент списка, укажите индекс первого элемента фрагмента и индекс последнего элемента фрагмента плюс 1. Разделите индексы двоеточием и заключите в квадратные скобки. Например, вывести на экран первые три пункта из списка дел можно следующим образом:

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

[1] https://docs.python.org/3/tutorial/datastructures.html

Результатом будет список элементов с индексами от 0 до 2:

```
['Pay bills', 'Tidy up', 'Walk the dog']
```

Начальный и конечный индексы в срезе указывать необязательно. Если опустить начальный индекс, срез начнется с нулевого элемента списка. Это означает, что срез из предыдущего примера можно смело изменить так:

```
print(my_list[:3])
```

Если опустить конечный индекс, то срез будет продолжаться до конца списка. Вот так можно вывести элементы с индексом 3 и выше:

```
print(my_list[3:])
```

В результате получим два последних пункта списка дел:

```
['Go to the pharmacy', 'Cook dinner']
```

Наконец, вы можете опустить оба индекса, и тогда получите копию всего списка:

```
print(my_list[:])
```

Результат:

```
['Pay bills', 'Tidy up', 'Walk the dog', 'Go to the pharmacy', 'Cook dinner']
```

Нотация срезов не ограничивается извлечением подпоследовательности элементов списка. Ее также можно использовать вместо методов `append()` и `insert()` для заполнения списка. Так, например, можно добавить два пункта в конец списка:

```
my_list[len(my_list):] = ['Mow the lawn', 'Water plants']
```

Функция `len()` возвращает количество элементов в списке. Это же значение является первым индексом, выходящим за пределы списка. Можно смело добавлять новые элементы, начиная с этого индекса. Теперь список выглядит так:

```
['Pay bills', 'Tidy up', 'Walk the dog', 'Go to the pharmacy', 'Cook dinner',
'Mow the lawn', 'Water plants']
```

Аналогично с помощью команды `del` и среза можно удалить элементы:

```
del my_list[5:]
```

Эта команда удалит элементы с индексами 5 и выше, тем самым возвращая список к прежнему виду:

```
['Pay bills', 'Tidy up', 'Walk the dog', 'Go to the pharmacy', 'Cook dinner']
```

Использование списка в качестве очереди

Очередь — это абстрактный тип данных, который может быть реализован с помощью структуры данных списка. Один конец очереди всегда используется для добавления элементов (*enqueue*), а другой — для их удаления (*dequeue*) согласно методологии *первым вошел — первым вышел* (FIFO, first-in, first-out). На практике методология FIFO часто используется при хранении товаров на складе: продукция, поступающая первой, первой склад и покидает. Подобная организация помогает предотвратить истечение сроков годности, поскольку в первую очередь продаются товары, произведенные раньше.

Список Python легко превратить в очередь, используя объект deque (сокращение от *double-ended queue*, двусторонняя очередь). Рассмотрим, как это работает, на примере нашего списка дел. Чтобы список функционировал как очередь, завершенные задачи должны вычеркиваться из начала, а новые — появляться в конце, как показано на рис. 2.1.

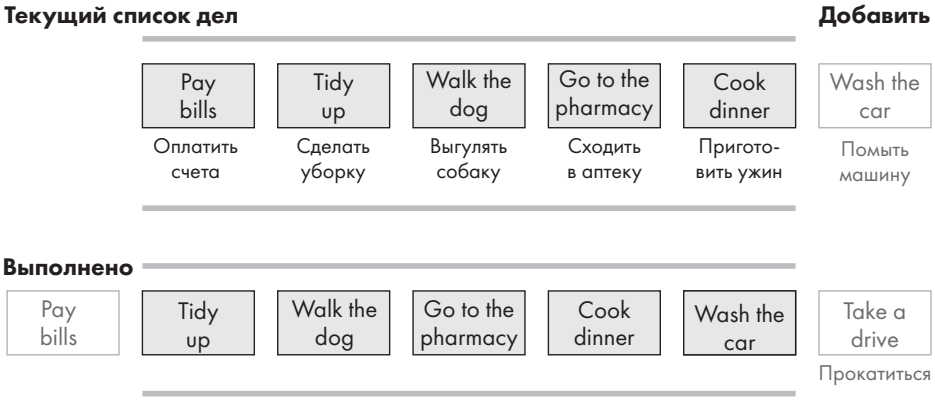

Рис. 2.1. Пример использования списка как очереди

Вот как реализовать процесс, показанный на рисунке:

```
from collections import deque
queue = deque(my_list)
queue.append('Wash the car')
print(queue.popleft(), ' - Done!')
my_list_upd = list(queue)
```

В этом скрипте мы сначала превращаем исходный объект `my_list` в объект `deque`, который является частью модуля Python `collections`. Конструктор объекта `deque()` добавляет набор методов к передаваемому в него объекту списка, и тогда этот список можно легко использовать в качестве очереди. В данном примере мы добавляем новый элемент в правую часть очереди с помощью метода `append()`, а затем удаляем элемент из левой части с помощью метода `popleft()`. Этот метод не только удаляет крайний левый элемент, но и возвращает его, вставляя в выводимое на экран сообщение. Вот каким оно должно быть:

```
Pay bills - Done!
```

После обратного преобразования объекта `deque` в список (последняя строка скрипта) новый перечень дел будет выглядеть так:

```
['Tidy up', 'Walk the dog', 'Go to the pharmacy', 'Cook dinner', 'Wash the car']
```

Как видите, первый элемент был вытеснен из списка, а новый добавлен.

Использование списка в качестве стека

Как и очередь, *стек* является абстрактной структурой данных, которую можно организовать поверх списка. Стек реализует методологию *последним вошел, — первым вышел* (LIFO, last-in, first-out), когда последний добавленный элемент извлекается первым. Чтобы наш список дел функционировал как стек, мы должны выполнять задачи в обратном порядке, начиная с самой правой. Вот как эта концепция реализуется в Python:

```
my_list = ['Pay bills', 'Tidy up', 'Walk the dog', 'Go to the pharmacy', 'Cook
dinner']
stack = []
for task in my_list:
    stack.append(task)
while stack:
```

```
    print(stack.pop(), ' - Done!'))
print('\nThe stack is empty')
```

В цикле `for` мы переносим задачи, начиная с первой, из списка дел в стек, объявленный как другой список. Это пример использования `append()` в цикле для динамического заполнения пустого списка. Затем, в цикле `while`, мы удаляем задачи из стека, начиная с последней. Мы делаем это с помощью метода `pop()`, который удаляет последний элемент из списка и возвращает его. Вывод стека будет таким:

```
Cook dinner - Done!
Go to the pharmacy - Done!
Walk the dog - Done!
Tidy up - Done!
Pay bills - Done!

The stack is empty
```

Использование списков и стеков для обработки естественного языка

Списки и стеки имеют широкое применение, в том числе в области обработки естественного языка (NLP, Natural Language Processing). Например, они используются для извлечения из текста именных групп (noun chunk). Такая группа состоит из существительного и зависимых слов (то есть всех слов слева[1] от существительного, которые синтаксически от него зависят, например прилагательных или артиклей в английском языке). Таким образом, чтобы извлечь из текста все именные группы, нужно найти в нем все существительные с зависимыми словами, располагающимися слева от существительного. Это можно сделать с помощью алгоритма на основе стека, как показано на рис. 2.2.

На рисунке в качестве примера представлена одна именная группа — *A ubiquitous data structure*[2]. Стрелки на синтаксическом дереве справа показывают, каким образом слова *A, ubiquitous* и *data* синтаксически связаны с существительным *structure*, известным как *вершина* (head) дочерних элементов. Алгоритм пословно анализирует текст слева направо и помещает слово в стек, если оно является существительным или расположенным слева от него зависимым словом. Если алгоритм встречает слово, которое не подходит под это описание, или если в тексте не осталось слов, значит, именная группа найдена целиком, и она извлекается из стека.

[1] Порядок расположения зависимых слов по отношению к главному зависит от языка. — *Примеч. ред.*

[2] «A ubiquitous data structure» — «распространенная структура данных». — *Примеч. пер.*

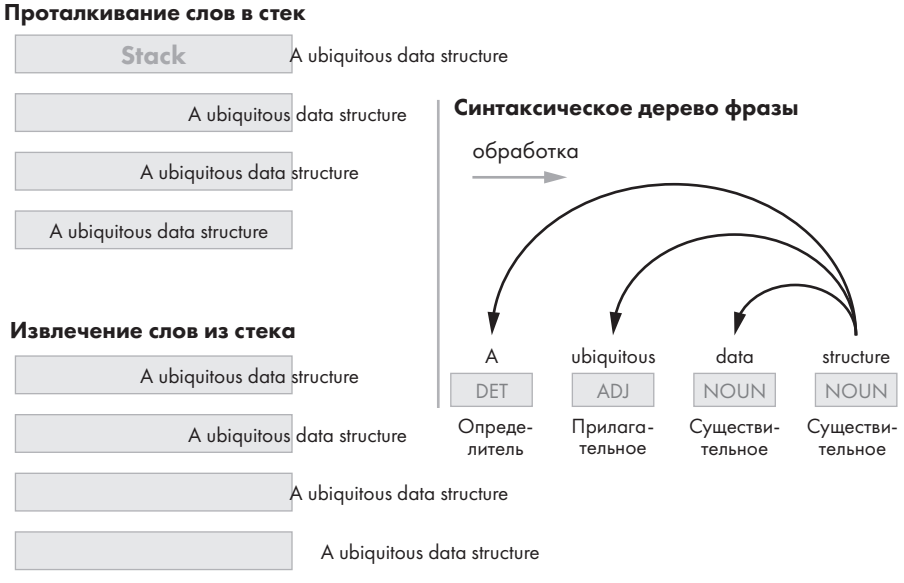

Рис. 2.2. Пример использования списка в качестве стека

Чтобы реализовать такой алгоритм, потребуется установить spaCy, основную библиотеку Python с открытым исходным кодом для обработки естественного языка, а также одну из ее моделей для английского языка. Используйте следующие команды:

```
$ pip install -U spacy
$ python -m spacy download en_core_web_sm
```

В следующем скрипте spaCy используется для реализации ведущей библиотеки Python для обработки естественного языка:

```
import spacy
txt = 'List is a ubiquitous data structure in the Python programming language.'

nlp = spacy.load('en_core_web_sm')
doc = nlp(txt)
stk = []
for w in doc:
   if w.pos_ == 'NOUN' or w.pos_ == 'PROPN': ❶
      stk.append(w.text)
   elif (w.head.pos_ == 'NOUN' or w.head.pos_ == 'PROPN') and
                        (w in w.head.lefts):
❷
```

```
        stk.append(w.text)
    elif stk: ❸
        chunk = ''
        while stk:
            chunk = stk.pop() + ' ' + chunk ❹
        print(chunk.strip())
```

Первые несколько строк скрипта — стандартный процесс настройки анализа текстовой фразы с помощью spaCy. Импортируем библиотеку spaCy, выбираем предложение, которое нужно разобрать, и загружаем модель spaCy для английского языка. После этого применяем к предложению пайплайн `nlp` и даем spaCy инструкцию сгенерировать синтаксическую структуру предложения. Это необходимо для таких задач, как извлечение именных групп.

ПРИМЕЧАНИЕ

Чтобы узнать более подробную информацию о spaCy, обратитесь к документации[1].

Далее реализуем алгоритм, описанный ранее, по каждому слову текста. Если мы нашли существительное ❶ или один из дочерних элементов, расположенных слева от этого существительного, ❷, отправляем слово в стек с помощью метода `append()`. Определяем эти слова, используя встроенные свойства spaCy, например `w.head.lefts`, которое позволяет перемещаться по синтаксической структуре предложения и находить нужные слова. Так, с помощью `w in w.head.lefts` мы ищем вершину (`w.head`), а затем дочерние слова, расположенные слева от нее (`.lefts`), и определяем, является ли данное слово (`w`) одним из них. Например, при анализе слова *ubiquitous* свойство `w.head` вернет слово *structure* (его синтаксическую вершину), а `.lefts`, примененное к *structure*, вернет слова *a*, *ubiquitous* и *data*, демонстрируя таким образом, что *ubiquitous* действительно является дочерним элементом, расположенным слева от *structure*.

В конце определяем, что следующее слово в тексте не является частью именной группы (ни существительным, ни дочерним словом слева от него) ❸, а значит, полученная группа полная, и мы извлекаем ее из стека ❹. Итак, скрипт находит и выводит три именные группы:

```
List
a ubiquitous data structure
the Python programming language.
```

[1] https://spacy.io

Расширение функциональности с помощью списковых включений

В главе 1 вы видели пример создания списка с помощью списковых включений. В этом разделе мы используем их для оптимизации алгоритма извлечения именных групп. Расширение функциональности решения часто требует значительного улучшения существующего кода. Однако в данном случае улучшения будут довольно компактными, поскольку речь идет о списковых включениях.

Взглянув на дерево синтаксических зависимостей, показанное на рис. 2.2, видим, что от каждого элемента фразы проведена синтаксическая дуга непосредственно к существительному *structure*. Однако именная группа может также строиться по другой схеме, когда слова внутри нее не связаны прямой синтаксической связью. Рисунок 2.3 иллюстрирует, как может выглядеть дерево зависимостей такой фразы. Обратите внимание, что наречие *most* является дочерним для прилагательного *useful*, а не существительного *type*, однако оно все еще является частью именной группы, в котором слово *type* является вершиной.

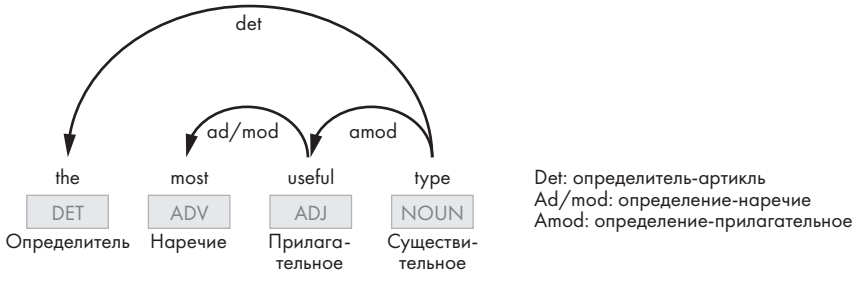

Рис. 2.3. Дерево синтаксических зависимостей для более сложной именной группы

Улучшим скрипт из предыдущего раздела, чтобы он также мог извлекать именные группы из таких предложений, как то, что показано на рис. 2.3, где некоторые слова не связаны с существительным группы напрямую. Чтобы доработать алгоритм, сначала сравним деревья синтаксических зависимостей, изображенные на рис. 2.2 и 2.3, и найдем то, что их объединяет. Важное сходство заключается в том, что в обоих деревьях вершина для каждого зависимого слова в именной группе находится справа. Однако так бывает не всегда. Например, в предложении «List is a ubiquitous data structure in the Python programming language»[1] слово *structure* является вершиной именной группы, но его собственная вершина — глагол *is* —

[1] «Список — это распространенная структура данных в языке программирования Python». — *Примеч. пер.*

находится слева. Чтобы убедиться в этом, можно запустить следующий скрипт, который выведет вершину для каждого слова в предложении:

```
txt = 'List is a ubiquitous data structure in the Python programming language.'
import spacy
nlp = spacy.load('en')
doc = nlp(txt)
for t in doc:
   print(t.text, t.head.text)
```

Наш новый алгоритм должен проходить по тексту в поисках слов, вершины для которых находятся справа, — вероятный признак того, что это именная группа. Идея в том, чтобы создать своего рода матрицу для предложения, показывающую, находится ли вершина для каждого конкретного слова справа. Для удобства чтения можно добавлять в матрицу слова, вершины для которых находятся справа, в том же виде, в каком они представлены в предложении, а все остальные слова заменять нулями. Тогда для предложения

```
List is arguably the most useful type in the Python programming language.
```

получим следующую матрицу:

```
['List', 0, 0, 'the', 'most', 'useful', 0, 0, 'the', 'Python', 'programming', 0, 0]
```

Такую матрицу можно создать с помощью списковых включений:

```
   txt = 'List is arguably the most useful type in the Python programming
         language.'
   import spacy
   nlp = spacy.load('en')
   doc = nlp(txt)
❶  head_lefts = [t.text if t in t.head.lefts else 0 for t in doc]
   print(head_lefts)
```

В этом фрагменте кода мы перебираем слова предложения в цикле внутри спискового включения, заменяя нулями те слова, вершины для которых не находятся справа ❶.

Полученный список будет выглядеть так:

```
['List', 0, 0, 'the', 'most', 'useful', 0, 0, 'the', 'Python', 'programming', 0, 0]
```

Видим, что список содержит на один элемент больше, чем количество слов в предложении. Это происходит потому, что spaCy фактически разбивает текст на лексемы (tokens), которые могут быть как словами, так и знаками препинания. Последний `0` в списке — это точка в конце предложения.

Теперь нам нужен способ перемещения по этому списку, чтобы найти и извлечь именные группы. Создадим набор фрагментов текста, начинающихся с определенного места и продолжающихся до конца текста. В следующем фрагменте кода будем двигаться пословно от начала до конца текста, на каждой итерации создавая матрицу с позицией вершины:

```
for w in doc:
   head_lefts = [t.text if t in t.head.lefts else 0 for t in ❶ doc[w.i:]]
   print(head_lefts)
```

Используем срез в объекте `doc`, чтобы получить нужный фрагмент ❶. Этот механизм позволяет сдвигать крайнюю левую позицию итогового фрагмента на одно слово вправо на каждой итерации цикла `for`. Код формирует следующий набор матриц:

```
['List', 0, 0, 'the', 'most', 'useful', 0, 0, 'the', 'Python', 'programming', 0, 0]
[0, 0, 'the', 'most', 'useful', 0, 0, 'the', 'Python', 'programming', 0, 0]
[0, 'the', 'most', 'useful', 0, 0, 'the', 'Python', 'programming', 0, 0]
['the', 'most', 'useful', 0, 0, 'the', 'Python', 'programming', 0, 0]
['most', 'useful', 0, 0, 'the', 'Python', 'programming', 0, 0]
['useful', 0, 0, 'the', 'Python', 'programming', 0, 0]
[0, 0, 'the', 'Python', 'programming', 0, 0]
[0, 'the', 'Python', 'programming', 0, 0]
['the', 'Python', 'programming', 0, 0]
['Python', 'programming', 0, 0]
['programming', 0, 0]
[0, 0]
[0]
```

Далее проанализируем каждый фрагмент в поисках первого встретившегося нуля. Слова до нуля (включительно) потенциально могут составлять именную группу. Код для этой операции:

```
for w in doc:
   head_lefts = [t.text if t in t.head.lefts else 0 for t in doc[w.i:]]
❶  i0 = head_lefts.index(0)
   if i0 > 0:
   ❷ noun = [1 if t.pos_== 'NOUN' or t.pos_== 'PROPN' else 0 for t in
```

```
                    reversed(doc[w.i:w.i+i0 +1])]
    try:
    ❸ i1 = noun.index(1)+1
    except ValueError:
       pass
    print(head_lefts[:i0 +1])
❹   print(doc[w.i+i0 +1-i1])
```

Приравниваем `i0` к `head_lefts.index(0)`, чтобы найти индекс первого нуля во фрагменте ❶. Если нулевых элементов несколько, `head_lefts.index(0)` возвращает индекс первого элемента. Затем проверяем, что `i0 > 0`, чтобы отсеять фрагменты, которые не начинаются с элемента слева от вершины.

Далее используем еще одно списковое включение для обработки именных групп, которые должны быть отправлены в стек. В этом втором списковом включении внутри каждой потенциальной именной группы мы ищем просто существительное (noun) или имя собственное (proper noun). Проходимся по фрагменту в цикле в обратном порядке (от конца к началу), чтобы в первую очередь найти существительное или имя собственное, которое образует фрагмент и которое, следовательно, должно располагаться в конце фрагмента ❷. При нахождении существительного или имени собственного в список отправляется `1`, а при нахождении любого другого элемента — `0`. Таким образом, первая `1` в списке указывает на положение главного существительного во фрагменте относительно конца этого фрагмента ❸. Это понадобится при нахождении фрагмента текста, представляющего именную группу ❹.

Пока что мы просто выводим сформированные фрагменты вместе с найденными в них существительными. Вывод будет таким:

```
['List', 0]
List
['the', 'most', 'useful', 0]
type
['most', 'useful', 0]
type
['useful', 0]
type
['the', 'Python', 'programming', 0]
language
['Python', 'programming', 0]
language
['programming', 0]
language
```

Теперь можно включить новый код в решение из предыдущего раздела. Собрав все воедино, получим такой скрипт:

```
   txt = 'List is arguably the most useful type in the Python
                                        programming language.'
   import spacy
   nlp = spacy.load('en')
   doc = nlp(txt)
   stk = []
❶  for w in doc:
   ❷ head_lefts = [1 if t in t.head.lefts else 0 for t in doc[w.i:]]
      i0 = 0
      try: i0 = head_lefts.index(0)
      except ValueError: pass
      i1 = 0
      if i0 > 0:
         noun = [1 if t.pos_== 'NOUN' or t.pos_== 'PROPN' else 0 for t in
                        reversed(doc[w.i:w.i+i0 +1])]
         try: i1 = noun.index(1)+1
         except ValueError: pass
      if w.pos_ == 'NOUN' or w.pos_ == 'PROPN':
      ❸ stk.append(w.text)
      elif (i1 > 0):
      ❹ stk.append(w.text)
      elif stk:
         chunk = ''
         while stk:
         ❺ chunk = stk.pop() + ' ' + chunk
         print(chunk.strip())
```

Мы проходим по лексемам в предложении ❶, создавая список `head_lefts` на каждой итерации ❷. Напомню, что этот список представляет собой матрицу, содержащую нули для тех слов в предложении, синтаксическая вершина которых находится слева от них. Эти матрицы используются для определения именных групп. Из каждой найденной группы мы отправляем в стек существительное или имя собственное ❸, а также любое другое слово, которое относится к этой группе, но не является существительным ❹. Когда мы доходим до конца группы, мы извлекаем лексемы из стека, формируя фразу ❺.

Скрипт выведет следующий результат:

```
List
the most useful type
the Python programming language
```

ПРИМЕЧАНИЕ

Если вы хотите узнать больше об обработке естественного языка, рекомендую мою книгу «Natural Language Processing with Python and spaCy», также изданную No Starch Press[1].

Кортежи

Так же как и список, *кортеж* представляет собой упорядоченную коллекцию объектов. Однако, в отличие от списков, кортеж неизменяем (immutable), то есть после создания его нельзя изменить. Элементы в кортеже разделяются запятыми и могут быть дополнительно заключены в круглые скобки:

```
('Ford', 'Mustang', 1964)
```

Обычно кортежи используются для хранения коллекций разнородных данных, то есть данных разных типов, например марки, модели и года выпуска автомобиля. Как показывает пример выше, кортежи особенно полезны при хранении свойств реального объекта.

Список кортежей

Достаточно часто структуры данных Python вложены друг в друга. Например, у вас может быть список, каждый элемент которого является кортежем, что позволяет присвоить элементу более одного признака. Допустим, нам необходимо задать время начала для каждой задачи в списке дел, который мы создали ранее в этой главе. Каждый элемент списка станет самостоятельной структурой данных, состоящей из двух элементов: описания задачи и запланированного времени начала ее выполнения.

Для реализации такой структуры кортежи — идеальный выбор, поскольку они предназначены для сбора неоднородных данных в единую структуру. Наш список кортежей может выглядеть так:

```
[('8:00','Pay bills'), ('8:30','Tidy up'), ('9:30','Walk the dog'),
 ('10:00','Go to the pharmacy'), ('10:30','Cook dinner')]
```

Его можно создать из двух простых списков:

[1] Васильев Ю. «Обработка естественного языка. Python и spaCy на практике». Санкт-Петербург, издательство «Питер».

```
task_list = ['Pay bills', 'Tidy up', 'Walk the dog', 'Go to the pharmacy',
             'Cook dinner']
tm_list = ['8:00', '8:30', '9:30', '10:00', '10:30']
```

Как видите, первый список — это исходный `my_list`, а второй — список, содержащий соответствующее время. Самый простой способ объединить их в список кортежей — использовать списковое включение, как показано ниже:

```
sched_list = [(tm, task) for tm, task in zip(tm_list, task_list)]
```

Внутри спискового включения используется функция `zip()`, которая проходит по двум простым спискам одновременно, объединяя соответствующее время и задачу в кортеж.

Как и в случае со списками, для доступа к элементу в кортеже необходимо указать индекс элемента, заключенный в квадратные скобки после имени кортежа. Однако обратите внимание, что кортежам, вложенным в список, не присваиваются имена. Чтобы получить доступ к элементу во вложенном кортеже, сначала нужно указать имя списка, затем индекс кортежа в списке и, наконец, индекс элемента в кортеже. Например, чтобы увидеть время выполнения второй задачи в списке дел, можно использовать следующий синтаксис:

```
print(sched_list[1][0])
```

Получим следующий вывод:

```
8:30
```

Неизменяемость

Важно помнить, что кортежи неизменяемы. Например, при попытке изменить время начала выполнения одной из обязанностей по дому:

```
sched_list[1][0] = '9:00'
```

получим ошибку:

```
TypeError: 'tuple' object does not support item assignment
```

Поскольку кортежи неизменяемы, они не подходят для хранения значений данных, которые необходимо периодически обновлять.

Словари

Словарь — это еще одна широко используемая в Python структура данных. Словарь — изменяемая, неупорядоченная коллекция пар *«ключ — значение»* (key-value), где каждый *ключ* является уникальным именем, которое указывает на элемент данных — *значение*. Словарь заключается в фигурные скобки. Каждый ключ отделяется от своего значения двоеточием, а пары «ключ — значение» разделяются запятыми, как показано ниже:

```
{'Make': 'Ford', 'Model': 'Mustang', 'Year': 1964}
```

Словари, как и кортежи, полезны для хранения разнородных данных об объектах реального мира. Как показано в этом примере, словари имеют дополнительное преимущество — возможность присвоить метку каждому элементу данных.

Список словарей

Как и другие типы данных, словари могут быть вложены в другие структуры. Наш список дел, реализованный в виде списка словарей, будет выглядеть следующим образом:

```
dict_list = [
   {'time': '8:00', 'name': 'Pay bills'},
   {'time': '8:30', 'name': 'Tidy up'},
   {'time': '9:30', 'name': 'Walk the dog'},
   {'time': '10:00', 'name': 'Go to the pharmacy'},
   {'time': '10:30', 'name': 'Cook dinner'}
]
```

В отличие от кортежей, словари изменяемы. Это означает, что в паре «ключ — значение» значение можно изменить:

```
dict_list[1]['time'] = '9:00'
```

Этот пример также показывает, как обращаться к значениям словаря: в отличие от списков и кортежей, для этого используются имена ключей, а не числовые индексы.

Добавление элементов в словарь с помощью setdefault()

Метод `setdefault()` — удобный способ добавления новых данных в словарь. В качестве параметра он принимает пару «ключ — значение». Если указанный ключ уже существует, метод просто возвращает текущее значение этого ключа, а если не существует, `setdefault()` вставляет его в словарь с указанным значением. Для примера сначала создадим словарь под названием `car` с моделью `Jetta`:

```
car = {
   "brand": "Volkswagen",
   "style": "Sedan",
   "model": "Jetta"
}
```

Теперь попробуем добавить новое значение `Passat` по ключу `model` с помощью `setdefault()`:

```
print(car.setdefault("model", "Passat"))
```

В результате получим вывод, показывающий, что значение ключа `model` не изменилось:

```
Jetta
```

Однако если указать новый ключ, `setdefault()` вставит пару «ключ — значение» и вернет значение:

```
print(car.setdefault("year", 2022))
```

Вывод:

```
2022
```

Теперь выведем на экран словарь целиком:

```
print(car)
```

Вот что мы получим:

```
{
   "brand": "Volkswagen",
   "style": "Sedan",
   "model": "Jetta",
   "year": 2022
}
```

Как видите, метод `setdefault()` избавляет от необходимости вручную проверять, есть ли уже в словаре ключ из вставляемой пары «ключ — значение». Ее можно добавлять в словарь без риска перезаписать значение уже существующего ключа.

Теперь, когда вы знаете, как работает `setdefault()`, рассмотрим практический пример. Подсчет количества вхождений каждого слова во фразу — частая задача NLP. В следующем примере показано, как решить ее с помощью словаря, используя метод `setdefault()`. Текст, который необходимо обработать:

```
txt = '''Python is one of the most promising programming languages today. Due to
the simplicity of Python syntax, many researchers and scientists prefer Python
over many other languages.'''
```

Первый шаг — удаление из текста знаков препинания. Без этого шага `"languages"` и `"languages."` будут считаться двумя отдельными словами. Удаляем точки и запятые:

```
txt = txt.replace('.', '').replace(',', '')
```

Далее разбиваем текст на слова и помещаем их в список:

```
lst = txt.split()
print(lst)
```

Получаем следующий список слов:

```
['Python', 'is', 'one', 'of', 'the', 'most', 'promising', 'programming',
 'languages', 'today', 'Due', 'to', 'the', 'simplicity', 'of', 'Python',
 'syntax', 'many', 'researchers', 'and', 'scientists', 'prefer', 'Python',
 'over', 'many', 'other', 'languages']
```

Теперь можем подсчитать количество вхождений каждого слова в список. Это можно реализовать с помощью словаря, используя метод `setdefault()`:

```
dct = {}
for w in lst:
   c = dct.setdefault(w,0)
   dct[w] += 1
```

Сначала создаем пустой словарь. Затем добавляем в него пары «ключ — значение», используя слова из списка в качестве ключей. Метод `setdefault()` устанавливает начальное значение для каждого ключа, равное `0`. Затем, при первом появлении определенного слова, значение увеличивается на 1, то есть значение счетчика теперь равно единице. При последующих появлениях этого слова к уже имеющемуся значению добавляется 1 с помощью оператора `+=`, точно подсчитывая вхождения.

Перед тем как выводить словарь, возможно, понадобится отсортировать слова по количеству вхождений:

```
dct_sorted = dict(sorted(dct.items(), key=lambda x: x[1], reverse=True))
print(dct_sorted)
```

Используя метод словаря `items()`, можно преобразовать словарь в список кортежей, где каждый кортеж будет состоять из ключа и его значения. Таким образом, когда внутри функции `sorted()` в `lambda` мы указываем `x[1]` для параметра `key`, мы делаем сортировку по первому элементу кортежа (с индексом `1`), то есть по значениям (количеству слов) исходного словаря. Полученный словарь выглядит следующим образом:

```
{'Python': 3, 'of': 2, 'the': 2, 'languages': 2, 'many': 2, 'is': 1, 'one': 1,
 'most': 1, 'promising': 1, 'programming': 1, 'today': 1, 'Due': 1, 'to': 1,
 'simplicity': 1, 'syntax': 1, 'researchers': 1, 'and': 1, 'scientists': 1,
 'prefer': 1, 'over': 1, 'other': 1}
```

Преобразование JSON в словарь

С помощью словарей структуры данных Python можно преобразовывать в строки JSON и наоборот. Вот как загрузить строку, представляющую собой документ JSON, в словарь, используя лишь оператор присваивания:

```
d = { "PONumber"             : 2608,
      "ShippingInstructions" : {"name"   : "John Silver",
                                "Address": {  "street"  : "426 Light Street",
                                              "city"    : "South San Francisco",
                                              "state"   : "CA",
                                              "zipCode" : 99237,
                                              "country" : "United States of America" },
                                "Phone" : [ { "type" : "Office", "number" : "809-123-9309" },
                                            { "type" : "Mobile", "number" : "417-123-4567" }
                                          ]
                               }
}
```

Как видим, у этого словаря сложная структура. Значение ключа `ShippingInstructions` само является словарем, в нем значение ключа `Address` тоже словарь, а значение ключа `Phone` является списком словарей.

Можно сохранить словарь непосредственно в файл JSON с помощью модуля `json`, используя метод `json.dump()`:

```
import json
with open("po.json", "w") as outfile:
    json.dump(d, outfile)
```

Аналогично, можно использовать метод `json.load()` для преобразования содержимого файла JSON в словарь Python:

```
with open("po.json",) as fp:
    d = json.load(fp)
```

В результате получим такой же словарь, как тот, что представлен выше. Более подробно о работе с файлами мы поговорим в главе 4.

Множества

Множество Python — это неупорядоченная коллекция неповторяющихся элементов. Дублирование элементов в множестве не допускается. Множество объявляется с помощью фигурных скобок, а элементы в нем разделяются запятыми:

```
{'London', 'New York', 'Paris'}
```

Удаление дубликатов из последовательности

Поскольку элементы множества должны быть уникальными, эта структура данных полезна, когда нужно удалить дублирующиеся элементы из списка или кортежа. Предположим, что требуется просмотреть список корпоративных клиентов. Такой список можно получить, извлекая имена клиентов из размещенных заказов. Поскольку клиент может сделать несколько заказов, имена в списке могут повторяться. Такие дубликаты можно удалить с помощью множества:

```
lst = ['John Silver', 'Tim Jemison', 'John Silver', 'Maya Smith']
lst = list(set(lst))
print(lst)
```

Мы просто приводим список к типу множества, а затем преобразуем обратно в список. Конструктор множества автоматически удаляет дубликаты. Обновленный список будет выглядеть так:

```
['Maya Smith', 'Tim Jemison', 'John Silver']
```

Недостаток этого подхода в том, что начальный порядок элементов не сохраняется, поскольку множество — это неупорядоченная коллекция элементов. Действительно, если запустить предыдущий код несколько раз, порядок вывода каждый раз будет разным.

Чтобы выполнить ту же операцию, сохранив первоначальный порядок, используйте функцию `sorted()`:

```
lst = ['John Silver', 'Tim Jemison', 'John Silver', 'Maya Smith']
lst = list(sorted(set(lst), key=lst.index))
```

Она сортирует множество по индексам исходного списка, сохраняя таким образом порядок. Новый список будет выглядеть так:

```
['John Silver', 'Tim Jemison', 'Maya Smith']
```

Общие операции с множеством

У объектов множеств есть методы для выполнения обычных математических операций над последовательностями, например объединения и пересечения. Эти

методы позволяют с легкостью объединять множества или извлекать элементы, общие для нескольких множеств.

Представьте, что вам нужно распределить огромное количество фотографий по группам исходя из того, что на них изображено. Чтобы автоматизировать эту задачу, можно начать с инструмента визуального распознавания (visual recognition), например Clarifai API, который будет генерировать множество описательных тегов для каждой фотографии. Затем множества тегов можно сравнить друг с другом с помощью метода `intersection()`. Этот метод сравнивает два множества и создает новое множество, содержащее элементы, которые есть в обоих множествах. В данном конкретном случае чем больше тегов в каждом множестве, тем более схожа тематика этих двух изображений.

Для простоты в следующем примере возьмем только две фотографии. Используя соответствующие множества описательных тегов, можно определить степень совпадения тематики этих двух фото:

```
photo1_tags = {'coffee', 'breakfast', 'drink', 'table', 'tableware', 'cup', 'food'}
photo2_tags = {'food', 'dish', 'meat', 'meal', 'tableware', 'dinner', 'vegetable'}
intersection = photo1_tags.intersection(photo2_tags)
if len(intersection) >= 2:
  print("The photos contain similar objects.")
```

В этом фрагменте кода мы выполняем операцию поиска пересечений, чтобы найти элементы, общие для обоих множеств. Если количество общих предметов в множествах равно или больше двух, можно сделать вывод, что фотографии имеют схожую тематику и, следовательно, их можно сгруппировать.

УПРАЖНЕНИЕ № 1: ПРОДВИНУТЫЙ АНАЛИЗ ТЕГОВ ФОТОГРАФИЙ

А сейчас попробуйте применить на практике то, чему вы научились в этой главе. Продолжите работать с примером с множествами из предыдущего раздела. В упражнении вам также понадобятся словари и списки.

В примере мы сравнили описательные теги только двух фотографий, по пересечению определив общие. Расширим функциональность кода, чтобы он обрабатывал произвольное количество фотографий, быстро группируя их в категории на основе пересекающихся тегов.

Предположим, что входными данными служит список словарей, где каждый словарь представляет собой фотографию (конечно, вы можете создать

свой собственный список, содержащий гораздо больше элементов). Список словарей ниже доступен для загрузки из репозитория GitHub для книги[1]:

```
l = [
 {
  "name": "photo1.jpg",
  "tags": {'coffee', 'breakfast', 'drink', 'table', 'tableware', 'cup', 'food'}
 },
 {
  "name": "photo2.jpg",
  "tags": {'food', 'dish', 'meat', 'meal', 'tableware', 'dinner', 'vegetable'}
 },
 {
  "name": "photo3.jpg",
  "tags": {'city', 'skyline', 'cityscape', 'skyscraper', 'architecture', 'building',
           'travel'}
 },
 {
  "name": "photo4.jpg",
  "tags": {'drink', 'juice', 'glass', 'meal', 'fruit', 'food', 'grapes'}
 }
]
```

Ваша задача — распределить фотографии по группам с помощью пересечений тегов и сохранить результат в словарь:

```
photo_groups = {}
```

Для этого необходимо перебрать все возможные пары фотографий из списка. Это можно реализовать с помощью вложенной пары циклов `for`, организованных следующим образом:

```
for i in range(1, len(l)):
   for j in range(i+1,len(l)+1):
      print(f"Intersecting photo {i} with photo {j}")
      # Реализуйте поиск пересечений здесь, сохраняя результаты
      # в photo_groups
```

Вам потребуется самостоятельно написать код тела цикла так, чтобы он выполнял поиск пересечений между `l[i]['tags']` и `l[j]['tags']` и создавал новую пару «ключ — значение» в словаре `photo_groups`, если

[1] https://github.com/pythondatabook/sources/blob/main/ch2/list_of_dicts.txt

результат пересечения окажется не пуст. Ключ можно составить из имен пересекающихся тегов, а значение должно представлять собой список с именами соответствующих файлов. Если для определенного множества пересекающихся тегов ключ уже существует, просто добавьте имена соответствующих файлов в список значений. Для реализации этого функционала можете воспользоваться методом `setdefault()`.

Для списка фотографий выше вы получите следующие группы:

```
{
  'tableware_food': ['photo1.jpg', 'photo2.jpg'],
  'drink_food': ['photo1.jpg', 'photo4.jpg'],
  'meal_food': ['photo2.jpg', 'photo4.jpg']
}
```

Если вы используете свой собственный набор фотографий с большим количеством элементов, то ключей и файлов, связанных с каждым ключом в итоговом словаре, может быть больше.

Решение для этого и всех других упражнений из книги можно найти в сопутствующем репозитории GitHub.

Выводы

В этой главе мы рассмотрели четыре встроенные структуры данных Python: списки, кортежи, словари и множества. Мы разобрали примеры, показывающие, как эти структуры могут представлять объекты реального мира и как объединять их во вложенные структуры, например список кортежей, список словарей или словарь, значениями которого являются списки.

Вы также узнали о возможностях Python, позволяющих легко создавать функциональные приложения для анализа данных. Например, мы разобрались, как использовать списковые включения для создания новых списков из существующих и метод `setdefault()` для эффективного доступа к данным в словаре и для их обработки. На практических примерах мы увидели, как эти функции применяются для решения распространенных задач, например обработки текста и анализа фотографий.

3

Библиотеки Python для data science

Python предоставляет доступ к надежной экосистеме сторонних библиотек, которые будут полезны при анализе данных и выполнении операций с данными. Это глава познакомит вас с тремя самыми популярными библиотеками для работы с данными: NumPy, pandas и scikit-learn. Вы убедитесь, что многие приложения для анализа данных прямо или косвенно используют эти библиотеки.

NumPy

Библиотека NumPy (сокращение от Numeric Python) полезна для работы с *массивами* — структурами данных, хранящими значения одного типа. Многие другие библиотеки Python, выполняющие операции с числами, опираются на NumPy.

Массив NumPy представляет собой сетку элементов одного типа. Эта структура данных является ключевым компонентом библиотеки NumPy. Элементы массива NumPy индексируются кортежем целых неотрицательных чисел. Массивы NumPy похожи на списки Python, за исключением того, что они требуют меньше памяти и обычно быстрее, поскольку используют оптимизированный, прекомпилированный код на языке C.

Массивы NumPy поддерживают *поэлементные операции*, что позволяет выполнять основные арифметические вычисления сразу над всем массивом в компактном и удобочитаемом коде. Поэлементная операция — это операция над двумя массивами одинаковой размерности, в результате которой получается другой массив той же размерности, где каждый элемент *i*, *j* является результатом вычисления, выполненного над элементами *i*, *j* двух исходных массивов. На рис. 3.1 показана поэлементная операция, выполняемая над двумя массивами NumPy.

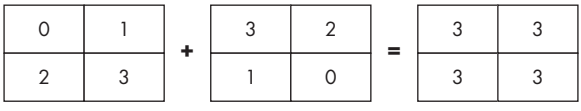

Рис. 3.1. Сложение двух массивов NumPy

Как видите, массив, полученный в результате, имеет те же размеры, что и исходные массивы, а каждый новый элемент является суммой их соответствующих элементов.

Установка NumPy

NumPy — сторонняя библиотека, то есть она не является частью стандартной библиотеки Python. Проще всего установить ее с помощью следующей команды:

```
$ pip install NumPy
```

Python воспринимает NumPy как модуль, поэтому перед использованием библиотеку необходимо импортировать в свой скрипт.

Создание массива NumPy

Массив NumPy можно создать из данных одного или нескольких списков Python. Предположим, для каждого сотрудника компании существует список выплат базового оклада за последние три месяца. Чтобы собрать всю информацию о зарплате в одну структуру данных, можно использовать следующий код:

```
❶ import numpy as np
❷ jeff_salary = [2700,3000,3000]
  nick_salary = [2600,2800,2800]
  tom_salary = [2300,2500,2500]
❸ base_salary = np.array([jeff_salary, nick_salary, tom_salary])
  print(base_salary)
```

Начинаем с импортирования библиотеки NumPy ❶. Затем определяем несколько списков, каждый из которых содержит данные о базовом окладе сотрудника за последние три месяца ❷. Наконец, объединяем списки в массив NumPy ❸. Вот как он выглядит:

```
[[2700 3000 3000]
 [2600 2800 2800]
 [2300 2500 2500]]
```

Это двумерный массив. У него две оси, которые имеют целочисленные индексы, начиная с 0. Ось 0 проходит вертикально вниз по строкам массива, а ось 1 — горизонтально по столбцам.

Точно так же можно создать массив, содержащий ежемесячные премии сотрудников:

```
jeff_bonus = [500,400,400]
nick_bonus = [600,300,400]
tom_bonus = [200,500,400]
bonus = np.array([jeff_bonus, nick_bonus, tom_bonus])
```

Выполнение поэлементных операций

Выполнять поэлементные операции сразу над несколькими массивами NumPy одинаковой размерности очень просто. Например, можно сложить массивы `base_salary` и `bonus`, чтобы определить общую сумму, выплачиваемую ежемесячно каждому сотруднику:

```
❶ salary_bonus = base_salary + bonus
  print(type(salary_bonus))
  print(salary_bonus)
```

Как видим, операция сложения реализуется в одной строке ❶. Полученный набор данных представляет собой массив NumPy, в котором каждый элемент является суммой соответствующих элементов массивов `base_salary` и `bonus`:

```
<class 'NumPy.ndarray'>
[[3200 3400 3400]
 [3200 3100 3200]
 [2500 3000 2900]]
```

Использование статистических функций NumPy

Статистические функции NumPy позволяют анализировать содержимое массива. Например, можно найти наибольшее значение по всему массиву или на заданной оси.

Допустим, требуется найти наибольшее значение в массиве `salary_bonus`, созданном в предыдущем разделе. Это можно сделать с помощью функции `max()` массива NumPy:

```
print(salary_bonus.max())
```

Функция возвращает максимальную сумму, выплаченную за последние три месяца сотрудникам, добавленным в датасет:

```
3400
```

NumPy также может найти максимальное значение массива на заданной оси. Если требуется определить максимальную сумму, выплаченную за последние три месяца каждому из сотрудников, можно использовать функцию NumPy `amax()`, как показано ниже:

```
print(np.amax(salary_bonus, axis = 1))
```

Указав `axis = 1`, мы сообщаем функции `amax()`, что искать максимум в массиве `salary_bonus` нужно горизонтально (по столбцам); таким образом, функция применяется к каждой строке. В результате рассчитывается максимальная ежемесячная сумма, выплаченная за последние три месяца, отдельно по каждому сотруднику:

```
[3400 3200 3000]
```

Аналогичным образом можно рассчитать максимальную сумму, выплаченную за каждый из последних трех месяцев, среди всех сотрудников, изменив параметр `axis` на `0`:

```
print(np.amax(salary_bonus, axis = 0))
```

Результат:

```
[3200 3400 3400]
```

ПРИМЕЧАНИЕ

Полный список статистических функций, поддерживаемых NumPy, приведен в документации NumPy[1].

УПРАЖНЕНИЕ № 2:
ИСПОЛЬЗОВАНИЕ СТАТИСТИЧЕСКИХ ФУНКЦИЙ NUMPY

Каждый набор результатов, возвращаемый `np.amax()` в примерах выше, сам является массивом NumPy. Это означает, что можно передать полученный набор в функцию `amax()` снова, если требуется найти максимальное значение во всем наборе. Аналогично можно передать этот набор любой другой статистической функции NumPy, например `np.median()` или `np.average()`. Попробуйте это сделать. Например, попробуйте найти среднее значение максимальных ежемесячных выплат среди всех сотрудников.

pandas

Библиотека pandas является фактическим стандартом для приложений Python, ориентированных на работу с данными (название библиотеки сложилось из фразы «Python Data Analysis Library»). Библиотека включает две структуры данных: *Series* (одномерную) и *DataFrame* (двумерную). Хотя DataFrame является основной структурой данных pandas, фактически она представляет собой коллекцию объектов Series. Поэтому важно понимать обе структуры.

Установка pandas

В стандартный дистрибутив Python не входит модуль pandas.

Его можно установить с помощью команды:

```
$ pip install pandas
```

Команда `pip` также разрешает зависимости библиотеки, неявно устанавливая пакеты NumPy, pytz и python-dateutil.

Как и в случае с NumPy, чтобы использовать модуль pandas, его необходимо импортировать в скрипт.

[1] https://NumPy.org/doc/stable/reference/routines.statistics.html

pandas Series

pandas Series — это одномерный массив. По умолчанию элементы серии маркируются целыми числами в соответствии с их позицией, как в списке Python. Однако можно задать и свои метки. Они необязательно должны быть уникальными, но должны быть хешируемого типа, например целыми числами (integer), числами с плавающей точкой (float), строками (string) или кортежами (tuple).

Элементы внутри серии могут быть любого типа (целыми числами, строками, числами с плавающей точкой, объектами Python и т. д.), но лучше всего Series работает, если все ее элементы имеют одинаковый тип. В конечном счете Series может стать столбцом в более крупном DataFrame, и вряд ли вы захотите хранить разные типы данных в одном столбце.

Создание Series

Есть несколько способов создать объект Series. В большинстве случаев мы передаем ему одномерный набор данных. Вот так можно создать Series из списка Python:

```
❶ import pandas as pd
❷ data = ['Jeff Russell','Jane Boorman','Tom Heints']
❸ emps_names = pd.Series(data)
  print(emps_names)
```

Прежде всего импортируем библиотеку pandas и даем ей короткое имя `pd` ❶. Затем формируем список элементов, которые будут использоваться в качестве данных для Series ❷. Наконец, создаем объект-серию, передавая список в метод конструктора `Series` ❸.

Мы получаем один список с проиндексированными по умолчанию элементами, начиная с 0:

```
0    Jeff Russell
1    Jane Boorman
2      Tom Heints
dtype: object
```

Атрибут `dtype` указывает тип данных, содержащихся в данной серии. По умолчанию для хранения строк pandas использует тип `object`.

Вы можете создать структуру Series, задав собственные индексы, следующим образом:

```
data = ['Jeff Russell','Jane Boorman','Tom Heints']
emps_names = pd.Series(data,index=[9001,9002,9003])
print(emps_names)
```

И теперь данные в серии `emps_names` выглядят так:

```
9001    Jeff Russell
9002    Jane Boorman
9003      Tom Heints
dtype: object
```

Доступ к данным в Series

Чтобы получить доступ к элементу в серии, укажите ее имя, а затем индекс элемента в квадратных скобках, как показано ниже:

```
print(emps_names[9001])
```

Выводится элемент, соответствующий индексу 9001:

```
Jeff Russell
```

В качестве альтернативы можно использовать свойство `loc` объекта Series:

```
print(emps_names.loc[9001])
```

Несмотря на то что в данном случае мы используем собственные индексы, получить доступ к элементам по позиции тоже можно (то есть использовать целочисленный индекс расположения элемента). Это делается с помощью свойства `iloc`. Так, например, можно вывести на экран первый элемент:

```
print(emps_names.iloc[0])
```

Можно получить доступ к нескольким элементам по их индексам с помощью среза, как описывалось в главе 2:

```
print(emps_names.loc[9001:9002])
```

В результате получается следующий вывод:

```
9001    Jeff Russell
9002    Jane Boorman
```

Обратите внимание, что срез со свойством `loc` включает и правую, конечную, точку (в данном случае индекс 9002), тогда как обычный срез в Python этого не делает.

Срезы также можно использовать, чтобы определить диапазон элементов по их позиции, а не по индексу. Например, тот же результат можно получить с помощью следующего кода:

```
print(emps_names.iloc[0:2])
```

или так:

```
print(emps_names[0:2])
```

Как видите, в отличие от `loc`, квадратные скобки (`[]`) или свойство `iloc` работают так же, как и обычный срез в Python: начальная позиция включается, а конечная — нет. Таким образом, `[0:2]` не берет элемент с индексом 2 и возвращает только первые два элемента.

Объединение Series в DataFrame

Несколько серий могут быть объединены в датафрейм (DataFrame). Попробуем сделать это, создав еще одну серию и объединив ее с серией `emps_names`:

```
   data = ['jeff.russell','jane.boorman','tom.heints']
❶  emps_emails = pd.Series(data,index=[9001,9002,9003], name = 'emails')
❷  emps_names.name = 'names'
❸  df = pd.concat([emps_names,emps_emails], axis=1)
   print(df)
```

Чтобы создать новую серию, мы вызываем конструктор `Series()` ❶, передавая следующие аргументы: список, который должен быть преобразован в серию, индексы и имя серии.

Перед объединением серий в структуру DataFrame необходимо присвоить сериям имена, поскольку эти названия станут именами соответствующих столбцов

датафрейма. Ранее, при создании серии `emps_names`, мы не указали имя явно, поэтому присвоим значение `names` свойству `name` ❷. Затем объединим `emps_names` с серией `emps_emails` ❸. Укажем `axis=1`, чтобы выполнить конкатенацию по столбцам.

В результате должен получиться следующий датафрейм:

```
              names          emails
9001   Jeff Russell   jeff.russell
9002   Jane Boorman   jane.boorman
9003     Tom Heints     tom.heints
```

УПРАЖНЕНИЕ № 3: ОБЪЕДИНЕНИЕ ТРЕХ СЕРИЙ

В предыдущем разделе мы создали датафрейм путем объединения двух серий. Используя этот же подход, попробуйте создать датафрейм из трех серий. Для этого вам потребуется создать еще один объект Series (скажем, `emps_phones`).

pandas DataFrame

Объект pandas DataFrame — это двумерная маркированная структура данных, столбцы которой могут содержать значения разных типов. DataFrame можно рассматривать как контейнер для объектов Series. Он похож на словарь, где ключ — метка столбца, а значение — серия.

Если вы знакомы с реляционными базами данных, то заметите, что pandas DataFrame похож на обычную таблицу SQL. На рис. 3.2 показан пример pandas DataFrame.

Обратите внимание, что в датафрейме есть столбец с индексами. Для структуры данных DataFrame по умолчанию использует целочисленную индексацию с нуля так же, как для Series. Однако индекс, установленный по умолчанию, можно заменить одним или несколькими существующими столбцами. На рис. 3.3 изображен тот же DataFrame, но в качестве индекса выбран столбец Date.

В данном конкретном примере индекс представляет собой столбец типа `date`. На самом деле в pandas можно устанавливать для датафрейма индексы любого типа. Чаще всего используют целые числа и строки. Однако вы не ограничены использованием простых типов данных. Можно выбрать в качестве индекса

последовательность, например список или кортеж, или даже использовать объектный тип, который не встроен в Python, то есть любой сторонний тип, даже собственный.

Столбец с индексами

	Date	Open	High	Low	Close	Volume
0	2020-08-26	412.00	433.20	410.73	430.63	71197000
1	2020-08-27	436.09	459.12	428.50	447.75	118465000
2	2020-08-28	459.02	463.70	437.30	442.68	20081200
3	2020-08-31	444.61	500.14	440.11	498.32	117841900
4	2020-09-01	502.14	502.49	470.51	493.43	43843641

Рис. 3.2. Пример pandas DataFrame

Столбец с индексами

Date	Open	High	Low	Close	Volume
2020-08-26	412.00	433.20	410.73	430.63	71197000
2020-08-27	436.09	459.12	428.50	447.75	118465000
2020-08-28	459.02	463.70	437.30	442.68	20081200
2020-08-31	444.61	500.14	440.11	498.32	117841900
2020-09-01	502.14	502.49	470.51	494.28	45409943

Рис. 3.3. pandas DataFrame со столбцом в качестве индекса

Создание pandas DataFrame

Мы разобрали, как создать pandas DataFrame путем объединения нескольких объектов Series. Также создать DataFrame можно, загрузив данные из базы данных, файла CSV, запроса API или из другого внешнего источника с помощью одного из *методов для чтения* из библиотеки pandas. Эти методы позволяют читать различные типы данных, такие как JSON и Excel, и преобразовывать их в датафрейм.

Рассмотрим датафрейм на рис. 3.2. Вероятно, он создан в результате запроса к Yahoo Finance API через библиотеку yfinance. Чтобы создать этот датафрейм самостоятельно, установите yfinance с помощью команды `pip` следующим образом:

```
$ pip install yfinance
```

Затем запросите данные биржевого рынка, как показано ниже:

```
   import yfinance as yf
❶  tkr = yf.Ticker('TSLA')
❷  hist = tkr.history(period= "5d")
❸  hist = hist.drop("Dividends", axis = 1)
   hist = hist.drop("Stock Splits", axis = 1)
❹  hist = hist.reset_index()
```

В данном скрипте мы отправляем API запрос на данные о цене акций для заданного тикера ❶ и используем метод yfinance `history()`, чтобы указать, что нам нужны данные за пятидневный период ❷. Полученные данные, хранящиеся в переменной `hist`, уже имеют вид pandas DataFrame. То есть yfinance создает датафрейм самостоятельно. После получения датафрейма нужно удалить некоторые столбцы ❸ и переиндексировать датафрейм ❹. В итоге должна получиться структура как на рис. 3.2.

Чтобы в качестве индекса установить столбец Date, как было показано на рис. 3.3, нужно выполнить следующую строку кода:

```
hist = hist.set_index('Date')
```

ПРИМЕЧАНИЕ

yfinance автоматически проиндексирует датафрейм по столбцу Date. В примерах выше мы перешли к числовой индексации, а затем обратно к индексации по дате, чтобы проиллюстрировать методы `reset_index()` и `set_index()`.

Теперь попробуем преобразовать документ JSON в объект pandas. Используемый в примере датасет содержит данные о месячной заработной плате трех сотрудников. Каждому сотруднику присвоен уникальный идентификатор (ID) в столбце `Empno`:

```
import json
import pandas as pd
data = [
```

```
    {"Empno":9001, "Salary":3000},
    {"Empno":9002, "Salary":2800},
    {"Empno":9003, "Salary":2500}
  ]
❶ json_data = json.dumps(data)
❷ salary = pd.read_json(json_data)
❸ salary = salary.set_index('Empno')
  print(salary)
```

Используем метод pandas `read_json()` для чтения и превращения JSON-строки в DataFrame ❷. Для простоты в данном примере используется строка JSON, преобразованная из списка с помощью `json.dumps()` ❶. Как вариант, можно передать в метод чтения объект `path` (путь до нужного JSON-файла) или URL-адрес HTTP API, который публикует данные в формате JSON. Наконец, выбираем столбец `Empno` в качестве индекса датафрейма ❸ и тем самым заменяем числовой индекс, установленный по умолчанию.

В результате получим следующий датафрейм:

```
        Salary
Empno
9001      3000
9002      2800
9003      2500
```

Еще одна распространенная практика — создание pandas DataFrame из стандартных структур данных Python, представленных в предыдущей главе. Например, можно создать датафрейм из списка списков:

```
  import pandas as pd
❶ data = [['9001','Jeff Russell', 'sales'],
          ['9002','Jane Boorman', 'sales'],
          ['9003','Tom Heints', 'sales']]
❷ emps = pd.DataFrame(data, columns = ['Empno', 'Name', 'Job'])
❸ column_types = {'Empno': int, 'Name': str, 'Job': str}
  emps = emps.astype(column_types)
❹ emps = emps.set_index('Empno') print(emps)
```

Сначала мы инициализируем список списков из данных, которые хотим отправить в датафрейм ❶. В нем каждый вложенный список станет строкой. Затем мы явно создаем датафрейм, определяя названия столбцов ❷. Далее используем словарь `column_types`, чтобы изменить типы данных, установленные для столбцов по умолчанию ❸. Этот шаг необязателен, но он может иметь решающее значение при объединении одного датафрейма с другим, поскольку сделать это

можно только со столбцами, содержащими данные одного типа. Наконец, мы устанавливаем столбец `Empno` в качестве индекса датафрейма ❹. Полученный датафрейм будет выглядеть так:

```
                Name      Job
Empno
9001    Jeff Russell    sales
9002    Jane Boorman    sales
9003      Tom Heints    sales
```

Обратите внимание, что в обоих датафреймах — `emps` и `salary` — столбец `Empno` используется в качестве индекса, то есть для уникальной идентификации каждой строки. Это необходимо, чтобы упростить процесс объединения двух этих датафреймов в один. Разберем данную операцию в следующем разделе.

Объединение датафреймов

pandas позволяет объединять датафреймы подобно тому, как мы объединяем различные таблицы в реляционной базе данных. Это позволяет собирать данные для анализа вместе. Методы объединения, поддерживаемые структурой данных DataFrame, напоминают операции при работе с базами данных: `merge()` и `join()`. Эти методы содержат несколько разных параметров, но так или иначе могут выступать как взаимозаменяемые.

Для начала объединим датафреймы `emps` и `salary`, объявленные в предыдущем разделе. Такой тип объединения называется *один-к-одному* (one-to-one join), где каждая строка одного датафрейма связана с соответствующей строкой другого датафрейма. На рис. 3.4 показано, как это работает.

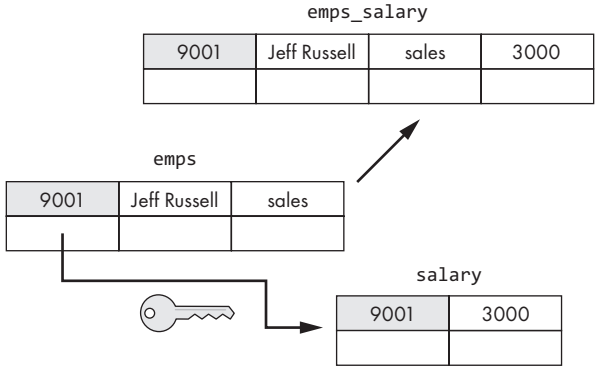

Рис. 3.4. Объединение двух датафреймов методом one-to-one

Здесь мы видим запись из датафрейма `emps` и запись из датафрейма `salary`. Обе записи имеют значение индекса `9001`, поэтому их можно объединить в одну запись в новом датафрейме — `emps_salary`. В терминологии реляционных баз данных столбцы, с помощью которых таблицы связаны между собой, называются *ключевыми*. Хотя в pandas существует термин *index* для таких столбцов, на рис. 3.4 используется значок ключа для визуального отображения объединения.

С помощью метода `join()` проводить объединение довольно просто:

```
emps_salary = emps.join(salary)
print(emps_salary)
```

Он предназначен для простого объединения датафреймов на основании их индексов. В этом конкретном примере нам даже не нужно было предоставлять дополнительные параметры; объединение по индексу — поведение по умолчанию. В результате получаем следующий датасет:

```
                Name      Job   Salary
Empno
9001    Jeff Russell    sales     3000
9002    Jane Boorman    sales     2800
9003      Tom Heints    sales     2500
```

На практике бывает необходимо объединить два датафрейма, в одном из которых есть строки, не имеющие совпадений в другом датафрейме. Предположим, у нас есть еще одна строка в датафрейме `emps`, а в `salary` соответствующей строки нет:

```
new_emp = pd.Series({'Name': 'John Hardy', 'Job': 'sales'}, name = 9004)
emps = emps.append(new_emp)
print(emps)
```

В этом случае создадим объект pandas Series, а затем добавим его к датафрейму `emps` с помощью метода `append()`. Это популярный способ добавления новых строк.

Обновленный датафрейм `emps` будет выглядеть следующим образом:

```
                Name      Job
Empno
9001    Jeff Russell    sales
9002    Jane Boorman    sales
9003      Tom Heints    sales
9004      John Hardy    sales
```

Если мы снова применим операцию join:

```
emps_salary = emps.join(salary)
print(emps_salary)
```

то получим такой датафрейм:

```
                Name      Job   Salary
Empno
9001    Jeff Russell    sales   3000.0
9002    Jane Boorman    sales   2800.0
9003      Tom Heints    sales   2500.0
9004      John Hardy    sales      NaN
```

Обратите внимание: строка, добавленная в `emps`, появляется в итоговом датасете, даже если в `salary` нет связанной с ней строки. Запись `NaN` в поле Salary последней строки означает, что значение оклада отсутствует. В некоторых случаях пустые значения в строках допустимы, как в данном примере, однако в других ситуациях может понадобиться исключить строки, которые не имеют соответствий в другом датафрейме.

По умолчанию метод `join()` использует индексы вызывающего датафрейма в результирующем, объединенном датафрейме, выполняя таким образом *left join*. В примере выше вызывающим датафреймом был `emps`. Он считается левым датафреймом в операции объединения, и поэтому все строки из него включаются в результирующий датасет. Это поведение можно изменить, передав параметр `how` в метод `join()`. Данный параметр принимает следующие значения:

`left` Использует столбец с индексами вызывающего датафрейма (или другой столбец, если указан параметр `on`), возвращая все строки из вызывающего (левого) датафрейма и только совпадающие строки из другого (правого) датафрейма.

`right` Использует индекс другого (правого) датафрейма, возвращая из него все строки и только совпадающие строки из вызывающего (левого) датафрейма.

`outer` Создает комбинацию индексов вызывающего датафрейма (или иного столбца, если указан параметр `on`) и индексов другого датафрейма, возвращая все строки из обоих датафреймов.

`inner` Формирует пересечение индексов вызывающего датафрейма (или иного столбца, если указан параметр `on`) с индексами другого датафрейма,

возвращая только те строки, индексы которых встречаются в обоих датафреймах.

На рис. 3.5 показаны примеры всех способов объединения методом join.

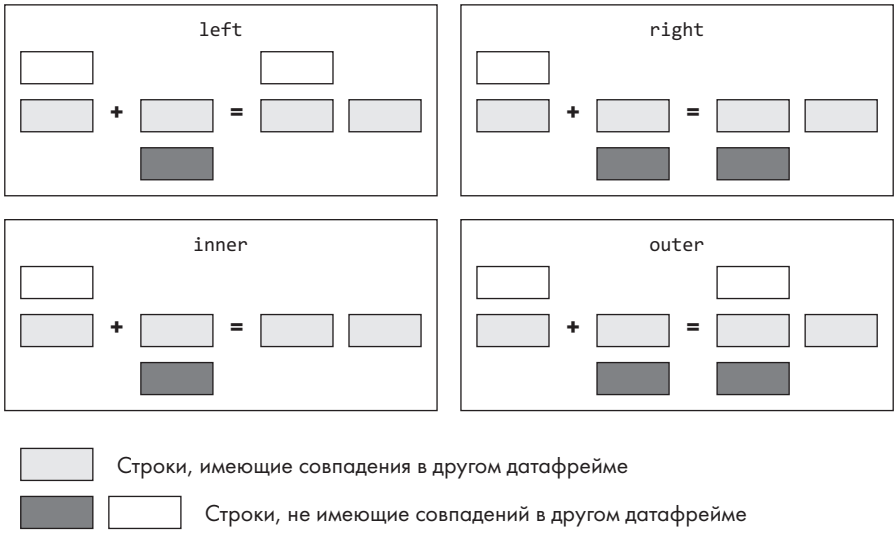

Рис. 3.5. Результаты различных способов объединения

Если необходимо, чтобы итоговый датафрейм включал только те строки из `emps`, которые имеют соответствия в `salary`, выберите значение `inner` для параметра `how` в методе `join()`:

```
emps_salary = emps.join(salary, how = 'inner')
print(emps_salary)
```

В результате должен получиться такой датафрейм:

```
                Name      Job   Salary
Empno
9001    Jeff Russell    sales     3000
9002    Jane Boorman    sales     2800
9003      Tom Heints    sales     2500
```

Еще один способ получить такой же результат — передать `right` в параметр `how`. В этом случае `join()` вернет все строки из датафрейма `salary`, добавив к ним

поля совпадающих строк из датафрейма `emps`. Однако важно понимать, что во множестве других случаев объединение методом right join — это не то же самое, что объединение inner join. Например, если в датафрейм `salary` добавить строку, для которой в `emps` нет совпадений, то объединение right join включит ее наряду со строками, имеющими совпадения в этом датафрейме.

ПРИМЕЧАНИЕ

Подробнее об объединении датафреймов см. в документации pandas[1].

УПРАЖНЕНИЕ № 4: ИСПОЛЬЗОВАНИЕ РАЗНЫХ ТИПОВ JOIN

Важно понимать, как различные значения параметра `how`, переданного в метод `join()`, влияют на итоговый датафрейм.

Добавьте новую строку к датафрейму `salary` так, чтобы значение поля `Empno` для этой строки в датафрейме `emps` отсутствовало (в предыдущем разделе мы рассмотрели, как добавить строку в датафрейм с помощью объекта Series). После этого объедините датафреймы `emps` и `salary` так, чтобы новый датафрейм включал только те строки из `emps`, которые имеют совпадения в `salary`. Затем снова объедините `emps` и `salary`, но теперь так, чтобы новый датафрейм включал все строки из обоих датафреймов.

Объединение «один-ко-многим»

При объединении методом «*один-ко-многим*» (one-to-many) строка из одного датафрейма может соответствовать сразу нескольким строкам из другого датафрейма. Рассмотрим ситуацию, когда каждый продавец из датафрейма `emps` обработал несколько заказов. Это можно отобразить в датафрейме `orders`:

```
import pandas as pd
data = [[2608, 9001,35], [2617, 9001,35], [2620, 9001,139],
        [2621, 9002,95], [2626, 9002,218]]
orders = pd.DataFrame(data, columns = ['Pono', 'Empno', 'Total'])
print(orders)
```

[1] https://pandas.pydata.org/pandas-docs/stable/reference/frame.html#combining-comparing-joining-merging

Так будет выглядеть `orders`:

```
    Pono   Empno   Total
0   2608    9001      35
1   2617    9001      35
2   2620    9001     139
3   2621    9002      95
4   2626    9002     218
```

Теперь, когда у вас есть датафрейм `orders` с заказами, можно объединить его с датафреймом сотрудников, созданным ранее. Это объединение типа «один-ко-многим», поскольку один сотрудник из датафрейма `emps` может быть связан с несколькими строками из датафрейма `orders`:

```
emps_orders = emps.merge(orders, how='inner', left_on='Empno',
                         right_on='Empno').set_index('Pono')
print(emps_orders)
```

В этом фрагменте кода мы применяем метод `merge()` для объединения данных из датафреймов `emps` и `orders`, имеющих связь «один-ко-многим». Метод `merge()` позволяет указать столбцы, по которым необходимо проводить объединение, сразу из обоих датафреймов, используя `left_on` для указания столбца из вызывающего датафрейма и `right_on` — для столбца из другого датафрейма. При использовании `join()` можно указать столбец, по которому нужно проводить объединение, только из вызывающего датафрейма. А из другого датафрейма `join()` берет столбец индексов.

В этом примере мы применяем внутренний тип объединения (inner), чтобы включить только связанные строки из обоих датафреймов. В результате получаем следующий датасет:

```
      Empno            Name      Job    Total
Pono
2608   9001    Jeff Russell    sales       35
2617   9001    Jeff Russell    sales       35
2620   9001    Jeff Russell    sales      139
2621   9002    Jane Boorman    sales       95
2626   9002    Jane Boorman    sales      218
```

Рисунок 3.6 иллюстрирует, как работает объединение типа «один-ко-многим».

Как видно на рисунке, объединение «один-ко-многим» включает по одной строке для каждой строки в датасете на стороне *ко-многим*. Поскольку мы используем тип

объединения `inner`, никакие другие строки не будут включены. А при объединении типа `left` или `outer` в результат будут также включены те строки из датасета на стороне *один-*, которые не имеют совпадений в датасете на стороне *ко-многим*.

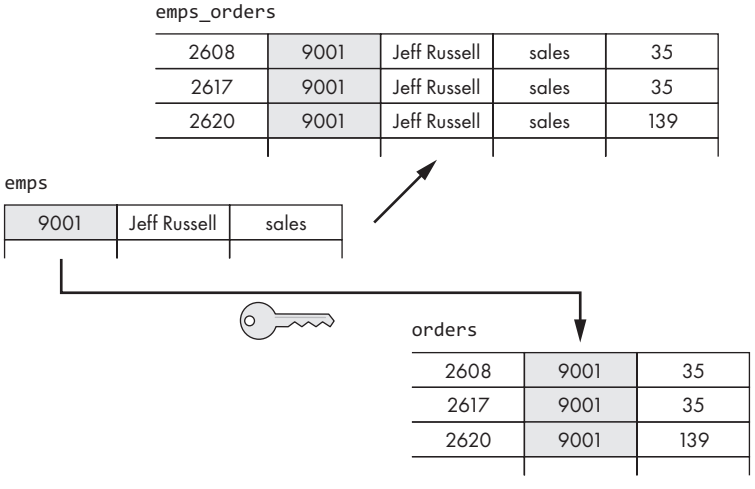

Рис. 3.6. Объединение двух датафреймов, связанных отношением «один-ко-многим»

Помимо отношений «один-ко-многим» и «один-к-одному», существует связь *«многие-ко-многим»* (many-to-many). В качестве примера такой взаимосвязи рассмотрим два датасета: в одном будут перечислены книги, а в другом — авторы. Каждая запись в датасете с авторами может быть связана с одной или несколькими записями в датасете с книгами, и наоборот, каждая запись в датасете с книгами может быть связана с одной или несколькими записями из набора данных с авторами. Мы обсудим этот тип отношений в главе 7, где более подробно рассматривается конкатенация, слияние и объединение датасетов.

Агрегирование данных методом groupby()

Метод pandas `groupby()` позволяет агрегировать данные по нескольким строкам датафрейма. Например, он может найти сумму значений столбца или получить среднее подмножество значений в столбце.

Предположим, что необходимо рассчитать среднюю сумму заказов по каждому сотруднику в датафрейме `orders`, созданном ранее. Для этого можно воспользоваться методом `groupby()`:

```
print(orders.groupby(['Empno'])['Total'].mean())
```

`groupby()` возвращает объект GroupBy, который поддерживает несколько статистических функций. В этом конкретном примере мы используем `mean()` для расчета средней общей суммы для группы заказов конкретного сотрудника. Для этого группируем строки из датафрейма `orders` по столбцу `Empno`, а затем применяем операцию `mean()` к столбцу `Total`. Созданный набор данных представляет собой объект Series:

```
Empno
9001     69.666667
9002    156.500000
Name: Total, dtype: float64
```

Теперь предположим, что требуется узнать сумму всех заказов для каждой группы (сотрудника). Здесь пригодится функция `sum()` объекта GroupBy:

```
print(orders.groupby(['Empno'])['Total'].sum())
```

Мы получим объект Series со следующими данными:

```
Empno
9001    209
9002    313
Name: Total, dtype: int64
```

ПРИМЕЧАНИЕ

Чтобы узнать больше о функциях, поддерживаемых объектом GroupBy, см. документацию по API pandas[1].

scikit-learn

scikit-learn — это библиотека Python, разработанная для приложений машинного обучения. Наряду с NumPy и pandas, это базовый компонент экосистемы Python для data science. scikit-learn предоставляет эффективные, простые в использовании инструменты, которые можно применять для решения общих задач машинного обучения, включая исследовательский и предиктивный анализ данных. В конце книги мы более подробно рассмотрим машинное обучение. А в этом разделе просто ознакомимся с тем, как можно применять Python в машинном обучении, особенно для предиктивного анализа данных.

[1] https://pandas.pydata.org/pandas-docs/stable/reference/groupby.html

Предиктивный анализ данных — это область машинного обучения, которая опирается на алгоритмы классификации и регрессии. И классификация, и регрессия используют уже имеющиеся данные для составления прогнозов относительно новых данных, но в процессе классификации данные сортируются по дискретным категориям, в то время как результатом регрессии служит непрерывный диапазон числовых значений. В этом разделе мы рассмотрим пример классификации, реализованной с помощью scikit-learn. Мы создадим прогностическую модель, анализирующую отзывы покупателей о продукции и сортирующую их по двум классам: положительные отзывы и отрицательные. Модель будет учиться на уже классифицированных образцах предсказывать класс других образцов. После того как мы обучим модель, мы покажем ей несколько новых отзывов, чтобы она классифицировала их как положительные или отрицательные.

Установка scikit-learn

Как NumPy и pandas, scikit-learn — сторонняя библиотека Python. Установить ее можно командой:

```
$ pip install sklearn
```

Библиотека scikit-learn имеет множество подмодулей с собственной функциональностью. Поэтому обычно импортируется не весь модуль, а только подмодуль, необходимый для конкретной задачи (скажем, `sklearn.model_selection`).

Получение набора образцов

Чтобы получить точную прогностическую модель, ее необходимо обучить на большом количестве маркированных образцов, поэтому первый шаг на пути к созданию модели, способной классифицировать отзывы о товарах, — получение набора отзывов, которые уже маркированы как положительные или отрицательные. Это избавит вас от необходимости самостоятельно собирать отзывы и вручную маркировать их.

Существует несколько онлайн-ресурсов с размеченными наборами данных. Один из лучших — Machine Learning Repository Калифорнийского университета UC Irvine[1]. Выполните поиск в репозитории по запросу «customer product reviews», и вы найдете ссылку на датасет Sentiment Labelled Sentences (или про-

[1] https://archive.ics.uci.edu/ml/index.php

сто вставьте в адресной строке браузера ссылку[1]). Скачайте и распакуйте файл *sentiment labelled sentences.zip* с веб-страницы датасета.

ПРИМЕЧАНИЕ

Набор данных Sentiment Labelled Sentences был создан для статьи Димитриоса Котзиаса и др. (Dimitrios Kotzias et al.) «От групповых меток к индивидуальным с помощью глубоких признаков» («From Group to Individual Labels Using Deep Features»).

Архив *.zip* содержит отзывы с IMDb, Amazon и Yelp в трех разных файлах с расширением *.txt*. Отзывы маркированы по тональности: положительные или отрицательные (`1` или `0` соответственно); для каждого источника представлено 500 положительных и 500 отрицательных отзывов, всего в датасете 3000 маркированных отзывов. Для простоты будем использовать только экземпляры с Amazon. Они находятся в файле *amazon_cells_labelled.txt*.

Преобразование загруженного датасета в pandas DataFrame

Чтобы упростить дальнейшие вычисления, необходимо преобразовать загруженный текстовый файл с отзывами в более управляемую структуру данных. Можно прочитать данные из *amazon_cells_labelled.txt* и преобразовать pandas DataFrame следующим образом:

```
import pandas as pd
df = pd.read_csv('/usr/Downloads/sentiment labelled
sentences/amazon_cells_labelled.txt',
                 names=[❶ 'review', ❷ 'sentiment'], ❸ sep='\t')
```

Здесь мы используем метод pandas `read_csv()` для преобразования данных в датафрейм. Указываем два столбца: первый для хранения отзывов ❶ и второй для соответствующих оценок тональности ❷. Поскольку в исходном файле для отделения отзывов и соответствующих оценок используется табуляция (tab), мы указываем `\t` в качестве разделителя ❸.

Разделение набора данных на обучающий и тестовый

Теперь, когда мы импортировали датасет, нужно разделить его на две части: одну — для обучения прогностической модели, а другую — для проверки ее точности. В scikit-learn это можно сделать всего несколькими строками кода:

[1] https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences

```
from sklearn.model_selection import train_test_split
reviews = df['review'].values
sentiments = df['sentiment'].values
reviews_train, reviews_test, sentiment_train, sentiment_test =
train_test_split(reviews,
                                        sentiments, ❶ test_size=0.2, ❷
random_state=500)
```

Разбиваем датасет с помощью функции `train_test_split()` из модуля `sklearn.model_selection`. Отзывы и соответствующие им оценки тональности передаются в функцию в виде массивов NumPy, полученных через свойство `values` соответствующих серий, извлеченных из датафрейма. Параметр `test_size` ❶ мы передаем, чтобы управлять тем, как именно будет разделен датасет. Значение `0.2` означает, что 20% отзывов будут случайным образом отнесены к тестовому набору. Соответственно, мы следуем схеме 80/20 — оставшиеся 80% отзывов составят обучающий набор. Параметр `random_state` ❷ инициализирует внутренний генератор случайных чисел. Он необходим для разбиения данных случайным образом.

Преобразование текста в числовые векторы признаков

Для обучения и тестирования модели требуется численно выразить текстовые данные. И здесь в дело вступает модель под названием *мешок слов* (bag of words, BoW). Эта модель представляет текст в виде набора (мешка) входящих в него слов в числовом формате. Наиболее типичной числовой характеристикой, генерируемой моделью BoW, является частота слов — сколько раз каждое слово встречается в тексте. Следующий простой пример показывает, как модель BoW преобразует текст в числовой вектор признаков на основе частоты встречаемости слов:

```
Text: I know it. You know it too.
BoW: { "I":1, "know":2, "it":2, "You":1, "too":1}
Vector: [1,2,2,1,1]
```

Для создания BoW-матрицы текстовых данных можно использовать функцию scikit-learn `CountVectorizer()`. Она преобразует текстовые данные в числовые векторы признаков (*n*-мерные векторы числовых признаков, представляющих некоторый объект) и выполняет токенизацию (разделение текста на отдельные слова и знаки препинания), используя либо стандартный, либо пользовательский токенизатор. Пользовательский токенизатор может быть реализован с помощью инструмента обработки естественного языка, такого как spaCy, из предыдущей главы. В этом примере для простоты используем вариант по умолчанию.

Вот как можно преобразовать отзывы в векторы признаков:

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
vectorizer.fit(reviews)
X_train = vectorizer.transform(reviews_train)
X_test = vectorizer.transform(reviews_test)
```

Прежде всего, создаем объект `vectorizer`. Затем применяем метод векторизатора `fit()` для построения словаря из лексем (tokens), найденных в датасете `reviews`, содержащем все отзывы из обучающего и тестового наборов. После этого используем метод `transform()` объекта `vectorizer` для преобразования текстовых данных в обучающем и тестовом наборах в числовые векторы признаков.

Обучение и оценка модели

Теперь, когда у нас есть обучающий и тестовый наборы в виде числовых векторов, можно приступать к обучению и тестированию модели. Сначала обучим классификатор scikit-learn `LogisticRegression()` для прогнозирования тональности отзыва. *Логистическая регрессия* — это простой, но популярный алгоритм для решения задач классификации.

Во фрагменте кода ниже мы создаем классификатор `LogisticRegression()`, затем используем его метод `fit()` для обучения модели на обучающих данных:

```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression()
classifier.fit(X_train, sentiment_train)
```

Теперь нужно оценить, насколько точно модель делает прогнозы, на новых данных. Именно поэтому обычно набор маркированных данных разделяют на обучающий и тестовый, как мы сделали это выше. Так можно оценить модель с помощью тестового набора:

```
accuracy = classifier.score(X_test, sentiment_test)
print("Accuracy:", accuracy)
```

Оценка точности обычно выглядит следующим образом:

```
Accuracy: 0.81
```

Это означает, что точность модели — 81%. Если поэкспериментировать с параметром `random_state` в функции `train_test_split()`, можно получить немного другое значение, поскольку обучающая и тестовая выборки формируются из исходного набора случайным образом.

Создание прогнозов на новых данных

Теперь, когда мы обучили и протестировали модель, она готова к анализу новых, немаркированных данных. Это даст нам более полное представление о том, насколько хорошо работает модель. Добавим в нее несколько новых отзывов:

```
new_reviews = ['Old version of python useless', 'Very good effort, but not
               five stars', 'Clear and concise']
X_new = vectorizer.transform(new_reviews)
print(classifier.predict(X_new))
```

На первом шаге создадим список новых отзывов, затем преобразуем текст каждого из них в числовые векторы признаков. И наконец, спрогнозируем классы тональности для новых образцов, которые возвращаются в виде списка:

```
[0, 1, 1]
```

Помните: `0` означает отрицательный отзыв, а `1` — положительный. Как видим, модель сработала для этих отзывов. Она показала, что первый — отрицательный, а два других — положительные.

Выводы

В этой главе вы познакомились с некоторыми из наиболее популярных сторонних библиотек Python для анализа данных и машинного обучения. Мы начали с изучения библиотеки NumPy и ее объектов многомерных массивов, а затем рассмотрели библиотеку pandas и ее структуры данных Series и DataFrame. Вы научились создавать массивы NumPy, объекты pandas Series и объекты pandas DataFrame из встроенных структур Python (например, списков) и из источников данных, хранящихся в стандартных форматах, таких как JSON. Мы также исследовали, как получать доступ к данным в этих объектах и управлять ими. Наконец, мы использовали scikit-learn, популярную библиотеку машинного обучения на Python, для построения прогностической модели классификации.

4

Доступ к данным из файлов и API

Получение доступа к данным и добавление их в скрипт — первый шаг анализа данных. В этой главе рассматриваются несколько способов импортирования данных из файлов и других источников в приложение на Python, а также способы экспортирования данных в файлы. Мы увидим, как получить доступ к содержимому файлов разных типов, включая те, что хранятся локально на компьютере, и те, что мы получаем удаленно через HTTP-запросы. Вы также узнаете, как получать данные, отправляя запросы к API, доступным по URL-адресу. Наконец, вы научитесь преобразовывать различные типы данных в структуру pandas DataFrames.

Импортирование данных с помощью функции open()

Встроенная в Python функция `open()` служит для открытия разных типов файлов, с которыми предстоит работать в скрипте. Функция возвращает объект `file`, который оснащен методами, позволяющими получить доступ к содержимому файла и совершать с ним различные операции. Однако если файл содержит данные в определенных форматах, таких как CSV, JSON или HTML, понадобится также импортировать соответствующую библиотеку. Для обработки файлов

с открытым текстом специальная библиотека не нужна, методов объекта `file`, возвращаемого `open()`, будет достаточно.

Текстовые файлы

Текстовые файлы (*.txt*) — это, пожалуй, самый распространенный тип файлов. Для Python текстовый файл — это последовательность строковых объектов. Каждый строковый объект представляет собой одну строку текстового файла, то есть последовательность символов, заканчивающуюся скрытым символом новой строки (`\n`) или жестким переносом (намеренным разрывом строки).

ПРИМЕЧАНИЕ

Одна строка текстового файла может отображаться на экране в виде нескольких строк, что зависит от ширины окна, но Python будет воспринимать ее как одну, если она не разбита символами новой строки.

В Python есть встроенные функции для чтения, записи и добавления текстовых файлов. В этом разделе мы рассмотрим, как читать данные из текстового файла. Начнем с ввода следующего отрывка в текстовом редакторе, а затем сохраним его как *excerpt.txt*. Обязательно дважды нажмите ENTER в конце первого абзаца, чтобы создать пустую строку между абзацами (но не нажимайте ENTER для разрыва длинных строк):

```
Today, robots can talk to humans using natural language, and they're getting
smarter. Even so, very few people understand how these robots work or how they
might use these technologies in their own projects.
Natural language processing (NLP) – a branch of artificial intelligence that
helps machines understand and respond to human language – is the key technology
that lies at the heart of any digital assistant product.[1]
```

Для человека отрывок состоит из двух абзацев, включающих всего три предложения. Однако для Python отрывок состоит из двух непустых строк и одной пустой строки, расположенной между ними. Прочитать содержимое всего файла в Python-скрипт и вывести его на экран можно так:

[1] «Сейчас роботы могут разговаривать с людьми на естественном языке, и они становятся все умнее. Но при этом очень немногие люди понимают, как работают эти роботы либо как использовать подобные технологии в собственных проектах.

Обработка естественного языка (NLP) — раздел искусственного интеллекта, необходимый для того, чтобы машины понимали и отвечали на человеческий язык, — это ключевая технология, которая лежит в основе любого продукта — цифрового помощника». — *Примеч. пер.*

```
❶ path = "/path/to/excerpt.txt"
  with open(❷ path, ❸ "r") as ❹ f:
  ❺ content = f.read()
  print(content)
```

Сначала указываем путь к файлу ❶. Вам нужно будет заменить */path/to/excerpt.txt* на собственный путь в зависимости от места сохранения файла. Путь передается в функцию `open()` в качестве первого параметра ❷. Второй параметр отвечает за то, как файл будет использоваться. По умолчанию параметр устанавливается в режим чтения текста, то есть содержимое файла будет открыто только для чтения (не для редактирования) и будет восприниматься как набор строк. Для открытия файла в режиме *чтения* ❸ можно явно указать `"r"`, но в этом нет строгой необходимости. (Если передать `"rt"`, то будет явно задан режим чтения именно текста.) Функция `open()` возвращает объект `file` в указанном режиме ❹. Затем используем метод `read()` объекта `file` для чтения всего содержимого файла ❺.

Использование ключевого слова `with` с `open()` гарантирует, что объект `file` будет правильно закрыт по окончании работы с ним, даже если было вызвано исключение. В противном случае для закрытия объекта файла и освобождения потребляемых им системных ресурсов необходимо вызвать `f.close()`.

Следующий фрагмент кода построчно читает содержимое того же файла с расположением */path/to/excerpt.txt*, выводя только непустые строки:

```
path = "/path/to/excerpt.txt"
with open(path,"r") as f:
❶ for i, line in enumerate(f):
  ❷ if line.strip():
        print(f"Line {i}: ", line.strip())
```

В этом примере мы добавляем порядковый номер к каждой строке с помощью функции `enumerate()` ❶. Затем отфильтровываем пустые строки с помощью метода `strip()` ❷, который удаляет пробельные символы из начала и конца каждой строки. Вторая пустая строка текстового файла содержит только один символ — новую строку, которую `strip()` удаляет. Таким образом, вторая строка становится пустой. Оператор `if` определит ее как false и пропустит. Мы получим следующий вывод. Как видите, `Line 1` отсутствует:

```
Line 0: Today, robots can talk to humans using natural language, and they're
getting smarter. Even so, very few people understand how these robots work or
how they might use these technologies in their own projects.
Line 2: Natural language processing (NLP) - a branch of artificial intelligence
that helps machines understand and respond to human language - is the key
technology that lies at the heart of any digital assistant product.
```

Вместо того чтобы печатать строки, их можно отправить в список, используя списковое включение:

```
path = "/path/to/excerpt.txt"
with open(path,"r") as f:
  lst = [line.strip() for line in f if line.strip()]
```

Каждая непустая строка станет отдельным элементом списка.

Файлы с табличными данными

Файл с *табличными* данными — это файл, в котором данные структурированы по строкам. Каждая строка обычно содержит информацию о ком-то или о чем-то, как показано ниже:

```
Jeff Russell, jeff.russell, sales
Jane Boorman, jane.boorman, sales
```

Это пример *плоского файла*, наиболее распространенного типа файла табличных данных. Такое название обусловлено структурой: плоские файлы содержат записи с простой (плоской) структурой, в них нет вложенных структур или подзаписей. Как правило, плоский файл — это текстовый файл в формате CSV или TSV (tab-separated values), содержащий по одной записи в строке. В файлах CSV значения в записи разделяются запятыми, а в файлах TSV в качестве разделителя используется табуляция. Оба формата имеют широкую поддержку; они часто используются для перемещения табличных данных между приложениями.

Ниже приведен пример данных в формате CSV, где первая строка содержит заголовки, которые описывают содержание строк, расположенных под ними. Описания заголовков используются в качестве *ключей* для получения данных, которые следуют за этими заголовками. Скопируйте эти данные в текстовый редактор и сохраните их как *cars.csv*:

```
Year,Make,Model,Price
1997,Ford,E350,3200.00
1999,Chevy,Venture,4800.00
1996,Jeep,Grand Cherokee,4900.00
```

Функция Python `open()` открывает файлы CSV в текстовом режиме. Затем можно преобразовать данные в объект Python с помощью функции чтения из модуля `csv`, как показано здесь:

```
import csv
path = "/path/to/cars.csv"
with open(path, "r") as ❶ csv_file:
   csv_reader = ❷ csv.DictReader(csv_file)
   cars = []
   for row in csv_reader:
   ❸ cars.append(dict(row))
print(cars)
```

Функция `open()` возвращает объект `file` ❶, который мы передаем считывателю из модуля `csv`. В данном случае используется `DictReader()` ❷, преобразующий данные в каждой строке в словарь, с использованием в качестве ключей соответствующих заголовков из первой строки. Далее мы добавляем эти словари в список ❸. В результате получается следующий список словарей:

```
[
  {'Year': '1997', 'Make': 'Ford', 'Model': 'E350', 'Price': '3200.00'},
  {'Year': '1999', 'Make': 'Chevy', 'Model': 'Venture', 'Price': '4800.00'},
  {'Year': '1996', 'Make': 'Jeep', 'Model': 'Grand Cherokee', 'Price':
'4900.00'}
]
```

В качестве альтернативы можно использовать метод `reader()` из модуля `csv`. Тогда файл CSV преобразуется в список списков, где каждый внутренний список будет представлять строку и первой будет строка заголовка:

```
import csv
path = "cars.csv"
with open(path, "r") as csv_file:
   csv_reader = csv.reader(csv_file)
   cars = []
   for row in csv_reader:
   cars.append(row)
print(cars)
```

Вывод:

```
[
  ['Year', 'Make', 'Model', 'Price']
  ['1997', 'Ford', 'E350', '3200.00']
  ['1999', 'Chevy', 'Venture', '4800.00']
  ['1996', 'Jeep', 'Grand Cherokee', '4900.00']
]
```

Методы `csv.DictReader()` и `csv.reader()` содержат необязательный параметр `delimiter`, позволяющий указать символ, разделяющий поля в файле табличных данных. По умолчанию в этом параметре выбрана запятая, что идеально подходит для файлов CSV. Однако, установив параметр `delimiter = "\t"`, можно считывать данные из файлов TSV, где поля разделены табуляцией.

УПРАЖНЕНИЕ № 5: ОТКРЫТИЕ JSON-ФАЙЛОВ

Открыть JSON-файл можно в текстовом режиме с помощью функции `open()`, а затем использовать модуль `json` для дальнейшей обработки. Как и `csv`, `json` является встроенным пакетом Python, поэтому его не нужно устанавливать отдельно. Мы показывали пример использования модуля `json` в главе 3, когда преобразовывали документ JSON в объект pandas. В этом упражнении требуется использовать модуль `json` для сохранения следующего текста в файл JSON:

```
{"cars":
  [{"Year": "1997", "Make": "Ford", "Model": "E350", "Price": "3200.00"},
   {"Year": "1999", "Make": "Chevy", "Model": "Venture", "Price": "4800.00"},
   {"Year": "1996", "Make": "Jeep", "Model": "Grand Cherokee", "Price":
"4900.00"}
]}
```

Откройте файл для чтения с помощью функции `open()` и отправьте полученный объект файла в метод `json.load()`, который десериализует JSON в объект Python. Из этого объекта извлеките часть, содержащую строки с описанием автомобилей. Пройдите по этим строкам в цикле, выводя значения следующим образом:

```
Year: 1997
Make: Ford
Model: E350
Price: 3200.00

Year: 1999
Make: Chevy
Model: Venture
Price: 4800.00

Year: 1996
Make: Jeep
Model: Grand Cherokee
Price: 4900.00
```

Двоичные файлы

Текстовые файлы — не единственный тип файлов, с которыми вам придется работать. Существуют также исполняемые (*.exe*) и графические (*.jpeg*, *.bmp* и др.) файлы, которые содержат данные в двоичном формате в виде последовательности байтов. Поскольку эти байты обычно интерпретируются не как текстовые символы, вы не можете получить доступ к содержимому двоичного файла, открыв его в текстовом режиме. Вместо этого необходимо использовать функцию `open()` в двоичном режиме.

В следующем примере показано, как открыть файл изображения в двоичном режиме. Попытка сделать это в текстовом режиме приведет к ошибке. Вы можете запустить этот код с любым файлом *.jpg* на компьютере:

```
   image = "/path/to/file.jpg"
   with open(image, ❶ "rb") as image_file:
      content = ❷ image_file.read()
❸ print(len(content))
```

Мы сообщаем функции `open()`, что нужно открыть файл для чтения в двоичном режиме, передавая `"rb"` в качестве второго параметра ❶. Извлеченный объект, как и объект, полученный при открытии файла в текстовом режиме, содержит метод `read()`, который позволяет получить содержимое файла ❷. В данном примере содержимое извлекается в виде объекта `bytes`. И мы просто определяем количество байтов, прочитанных из файла ❸.

Экспортирование данных в файл

После обработки данных вам может понадобиться сохранить их в файл, чтобы использовать при следующем исполнении скрипта, импортировать в другие скрипты или приложения либо просто просматривать. Например, вам может понадобиться регистрировать информацию об ошибках и исключениях приложения для последующего анализа.

Вы можете записать данные, создав новый файл из Python-скрипта или перезаписав существующий. Рассмотрим второй вариант. Возвращаясь к примеру из раздела «Файлы с табличными данными», предположим, что требуется изменить строку в файле *cars.csv*, скорректировав цену определенного автомобиля. Напомним, что данные были считаны из файла *cars.csv* в список словарей с именем `cars`. Чтобы просмотреть значения каждого словаря в этом списке, можно запустить следующий цикл:

```
for row in cars:
   print(list(row.values()))
```

В теле цикла мы вызываем метод `values()` для каждого словаря списка, тем самым преобразуя значения словаря в объект `dict_values`, который легко преобразовать в список. Каждый список представляет собой строку из исходного файла CSV, как показано здесь:

```
['1997', 'Ford', 'E350', '3200.00']
['1999', 'Chevy', 'Venture', '4800.00']
['1996', 'Jeep', 'Grand Cherokee', '4900.00']
```

Предположим, нам нужно обновить поле `Price` во второй строке (для автомобиля Chevy Venture) и сохранить это изменение в исходном файле *cars.csv*. Это можно сделать так:

```
❶ to_update = ['1999', 'Chevy', 'Venture']
❷ new_price = '4500.00'
❸ with open('path/to/cars.csv', 'w') as csvfile:
   ❹ fieldnames = cars[0].keys()
   ❺ writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
     writer.writeheader()
   ❻ for row in cars:
       if set(to_update).issubset(set(row.values())):
          row['Price'] = new_price
       writer.writerow(row)
```

Прежде всего, нам нужен способ идентифицировать строку, которую требуется обновить. Мы создаем список с названием `to_update`, элементов в котором будет достаточно для того, чтобы однозначно идентифицировать строку ❶. Затем определяем новое значение для изменяемого поля как `new_price` ❷. Далее открываем файл для записи, передавая параметр `w` в функцию `open()` ❸. Используемый здесь режим `w` будет перезаписывать существующее содержимое файла. Соответственно, далее нужно определить имена полей, которые будут отправлены в файл ❹. Это имена ключей словаря, который представляет собой строку с информацией об автомобиле.

Используя функцию ❺ `csv.DictWriter()`, мы создаем объект `writer`, который будет преобразовывать словари из списка `cars` в выходные строки для отправки в файл *cars.csv*. Проходя в цикле по словарям списка `cars` ❻, проверяем, соответствует ли каждая строка указанному идентификатору. Если да, то для данной строки обновляем поле `Price`. Наконец, все еще внутри цикла, записываем каждую строку в файл, используя метод `writer.writerow()`.

Вот что вы увидите в файле *cars.csv* после выполнения скрипта:

```
Year,Make,Model,Price
1997,Ford,E350,3200.00
1999,Chevy,Venture,4500.00
1996,Jeep,Grand Cherokee,4900.00
```

Как видим, файл выглядит как оригинал, но значение поля `Price` во второй строке изменилось.

Доступ к удаленным файлам и API

Некоторые сторонние библиотеки Python, включая urllib3 и Requests, позволяют получить данные из удаленного файла, доступного по URL. Библиотеки также можно использовать для отправки запросов к HTTP API (API, использующим HTTP в качестве протокола передачи данных). Многие из них возвращают запрашиваемые данные в формате JSON. И urllib3, и Requests формируют пользовательские HTTP-запросы на основе введенной информации.

HTTP (HyperText Transfer Protocol — *протокол передачи гипертекста*), клиент-серверный протокол, составляющий основу обмена данными в интернете, структурирован как серия запросов и ответов. HTTP-сообщение, посылаемое клиентом, — это *запрос* (request), а сообщение, возвращаемое сервером, — *ответ* (response). Например, всякий раз, когда вы щелкаете по ссылке в браузере, выступающем в роли клиента, он отправляет HTTP-запрос для получения нужной веб-страницы с соответствующего веб-сервера. То же самое можно сделать из Python-скрипта. Скрипт, выступающий в роли клиента, получает запрашиваемые данные в виде документа JSON или XML.

Как работают HTTP-запросы

Существует несколько типов HTTP-запросов. Наиболее популярные из них — `GET`, `POST`, `PUT` и `DELETE`. Они также известны как *методы HTTP-запросов* или просто *HTTP-команды* (HTTP command / HTTP verb). Команда HTTP в любом HTTP-запросе определяет действие, которое необходимо выполнить с указанным ресурсом. Например, `GET`-запрос получает данные из ресурса, а `POST`-запрос отправляет данные в место назначения.

HTTP-запрос включает *цель запроса* (target) — обычно это URL-адрес, а также *заголовки* (headers), которые представляют собой поля, передающие вместе с запросом дополнительную информацию. В некоторых запросах также есть *тело*

(body), которое содержит фактические данные запроса, например отправку формы. `POST`-запросы обычно имеют тело, а `GET`-запросы — нет.

В качестве примера рассмотрим следующий HTTP-запрос:

```
❶ GET ❷ /api/books?bibkeys=ISBN%3A1718500521&format=json HTTP/1.1
❸ Host: openlibrary.org
❹ User-Agent: python-requests/2.24.0
❺ Accept-Encoding: gzip, deflate
  Accept: */*
❻ Connection: keep-alive
```

В этом запросе используется HTTP-команда `GET` ❶ для получения данных с заданного сервера (указан как `Host` ❸) по заданному URI ❷. Остальные строки — это заголовки с дополнительной информацией. Заголовок запроса `User-Agent` определяет приложение, отправляющее запрос, и его версию ❹. Заголовок `Accept` сообщает, какие типы содержимого клиент способен обработать ❺. Заголовок `Connection` со значением `keep-alive` ❻ дает указание серверу установить постоянное соединение с клиентом, что позволяет выполнять последующие запросы.

Чтобы отправлять HTTP-запросы с помощью Python и получать ответы, необязательно полностью понимать их внутреннюю структуру. Как вы узнаете из следующих разделов, такие библиотеки, как Requests и urllib3, позволяют легко и эффективно управлять HTTP-запросами, просто вызывая соответствующий метод и передавая в него необходимые параметры.

С помощью библиотеки Requests предыдущий HTTP-запрос можно сгенерировать простым Python-скриптом:

```
import requests
PARAMS = {'bibkeys':'ISBN:1718500521', 'format':'json'}
requests.get('http://openlibrary.org/api/books', params = PARAMS)
```

Скоро мы подробно обсудим библиотеку Requests. А пока обратите внимание, что она избавляет от необходимости задавать заголовки запроса вручную. Requests скрыто устанавливает значения по умолчанию, автоматически генерируя полностью отформатированный HTTP-запрос от вашего имени всего несколькими строками кода.

Библиотека urllib3

urllib3 — это библиотека для работы с URL. Она позволяет получать доступ к URL-ресурсам, таким как HTTP API, веб-сайты и файлы, и управлять ими.

Эта библиотека предназначена для эффективной обработки HTTP-запросов с помощью потокобезопасного пула соединений, что минимизирует количество необходимых серверных ресурсов.

По сравнению с библиотекой Requests, которую мы рассмотрим далее, urllib3 требует больше ручных операций, однако она предоставляет больше прямого контроля над создаваемыми запросами, что может быть полезно при настройке поведения пула или расшифровке HTTP-ответов.

Установка urllib3

Поскольку urllib3 входит в перечень зависимостей многих популярных библиотек для Python (например, Requests и `pip`), скорее всего, она уже установлена в вашей среде Python. Чтобы проверить это, импортируйте ее в текущей сессии. Если вы получаете ошибку `ModuleNotFoundError`, установите библиотеку с помощью следующей команды:

```
$ pip install urllib3
```

Доступ к файлам через urllib3

Чтобы потренироваться с помощью urllib3 загружать данные из файла, хранящегося по какому-либо URL-адресу, можно обратиться к файлу *excerpt.txt*, который мы создали ранее. Чтобы сделать этот файл доступным по URL, поместите его в папку document HTTP-сервера, запущенного на локальном хосте. Или используйте URL-адрес файла из сопутствующего репозитория GitHub[1].

Запустите следующий код, при необходимости изменив URL-адрес:

```
  import urllib3
❶ http = urllib3.PoolManager()
❷ r = http.request('GET', 'http://localhost/excerpt.txt')
  for i, line in enumerate(❸ r.data.decode('utf-8').split('\n')):
    if line.strip():
    ❹ print("Line %i: " %(i), line.strip())
```

Сначала мы создаем экземпляр `PoolManager` ❶, с помощью которого urllib3 выполняет запросы. После этого выполняется HTTP-запрос на указанный URL-адрес с помощью метода `request()` `PoolManager` ❷. Метод `request()` возвращает объект `HTTPResponse`. Мы получаем доступ к запрошенным данным через атрибут

[1] https://github.com/pythondatabook/sources/blob/main/ch4/excerpt.txt

`data` этого объекта ❸. Затем выводим только непустые строки вместе с их порядковым номером в начале каждой строки ❹.

Запросы API через urllib3

urllib3 также можно использовать для выполнения запросов к HTTP API. В примере ниже мы делаем запрос к News API[1], который выполняет поиск наиболее релевантных запросу статей из множества новостных источников. Как и многие другие современные API, News API требует, чтобы API-ключ передавался при каждом запросе. API-ключ разработчика можно получить бесплатно на сайте[2] после заполнения простой регистрационной формы. Затем используйте код ниже, чтобы найти статьи о языке программирования Python:

```
   import json
   import urllib3
   http = urllib3.PoolManager()
   r = http.request('GET', 'https://newsapi.org/v2/everything? ❶ q=Python
           programming language& ❷ apiKey=your_api_key_here& ❸ pageSize=5')
❹  articles = json.loads(r.data.decode('utf-8'))
   for article in articles['articles']:
     print(article['title'])
     print(article['publishedAt'])
     print(article['url'])
     print()
```

Вы передаете поисковую фразу в качестве параметра `q` в URL-адресе запроса ❶. Единственный параметр, который необходимо указать в URL запроса, — `apiKey` ❷, передающий ключ API. Существует также множество необязательных параметров. Например, можно указать источники новостей или блоги, из которых вы хотите получать статьи. В этом конкретном примере мы используем `pageSize`, чтобы установить количество получаемых статей равным пяти ❸. Полный список поддерживаемых параметров можно найти в документации News API[3].

Атрибут `data` объекта `HTTPResponse`, возвращаемого функцией `request()`, — документ JSON в виде объекта `bytes`. Мы преобразовываем его в строку, которую затем передаем методу `json.loads()` для преобразования в словарь ❹. Чтобы увидеть, как структурированы данные в этом словаре, вы можете вывести его, но

[1] https://newsapi.org

[2] https://newsapi.org/register

[3] https://newsapi.org/docs

в данном листинге этот шаг опущен. Если бы вы посмотрели на вывод, то увидели бы, что информация о статьях находится в списке `articles` возвращаемого документа и каждая запись в этом списке содержит поля `title`, `publishedAt` и `url`.

Используя эту информацию, можно распечатать полученный список статей в более удобном для чтения виде, например:

```
A Programming Language To Express Programming Frustration
2021-12-15T03:00:05Z
https://hackaday.com/2021/12/14/a-programming-language-to-express-programming-
frustration/

Raise Your Business's Potential by Learning Python
2021-12-24T16:30:00Z
https://www.entrepreneur.com/article/403981

TIOBE Announces that the Programming Language of the Year Was Python
2022-01-08T19:34:00Z
https://developers.slashdot.org/story/22/01/08/017203/tiobe-announces-that-the-
programming-language-of-the-year-was-python

Python is the TIOBE programming language of 2021 — what does this title even
mean?
2022-01-04T12:28:01Z
https://thenextweb.com/news/python-c-tiobe-programming-language-of-the-year-
title-analysis

Which programming language or compiler is faster
2021-12-18T02:15:28Z
```

В этом примере показано, как интегрировать News API в Python-приложение, используя прямые HTTP-запросы через библиотеку urllib3. Еще один вариант — использование неофициальной клиентской библиотеки Python[1].

Библиотека Requests

Requests — еще одна популярная библиотека для работы с URL-адресами. Она позволяет с легкостью отправлять HTTP-запросы. Requests в своей основе использует urllib3, что еще больше упрощает выполнение запросов и получение данных. Установить библиотеку Requests можно командой `pip`:

```
$ pip install requests
```

[1] https://newsapi.org/docs/client-libraries/python

HTTP-команды реализуются как методы библиотеки (например, `requests.get()` используется для выполнения HTTP-запроса `GET`). Ниже показано, как получить удаленный доступ к *excerpt.txt* с помощью Requests. Замените URL-адрес GitHub-ссылкой на файл, если необходимо:

```
   import requests
❶  r = requests.get('http://localhost/excerpt.txt')
   for i, line in enumerate(❷ r.text.split('\n')):
     if line.strip():
     ❸ print("Line %i: " %(i), line.strip())
```

Мы выполняем `GET`-запрос с помощью метода `requests.get()`, передавая URL-адрес файла в качестве параметра ❶. Метод возвращает объект `Response`, в атрибуте `text` которого содержатся полученные данные ❷. Requests автоматически декодирует полученный контент, делая обоснованные предположения о кодировке, так что вам не придется заниматься этим самостоятельно. Как и в примере с urllib3, мы выводим только непустые строки, добавляя номер каждой строки в начале ❸.

УПРАЖНЕНИЕ № 6: ДОСТУП К API С ПОМОЩЬЮ REQUESTS

Как и urllib3, библиотека Requests может взаимодействовать с HTTP API. Попробуйте переписать код, который отправляет `GET`-запрос к News API, так, чтобы в нем использовалась библиотека Requests вместо urllib3. Обратите внимание, что при использовании Requests не нужно вручную добавлять параметры запроса к URL, передаваемому в запросе. Вместо этого передавайте параметры в виде словаря строк.

Перемещение данных в DataFrame и из него

В pandas входит ряд методов чтения, каждый из которых предназначен для загрузки данных в определенном формате и/или из определенного типа источника. Эти методы позволяют преобразовать табличные данные в DataFrame одним вызовом, благодаря чему импортированный датасет сразу готов к анализу. В pandas также есть методы для преобразования данных DataFrame в другие форматы, например JSON. В этом разделе рассматриваются методы, позволяющие перемещать данные в DataFrame или из него. Мы также рассмотрим библиотеку pandas-datareader, которая полезна для загрузки данных из различных онлайн-источников и преобразования в pandas DataFrame.

Импортирование вложенных структур JSON

Поскольку JSON стал фактическим стандартом для обмена данными между приложениями, важно иметь возможность быстро импортировать JSON-документ и преобразовывать его в структуру данных Python. В предыдущей главе мы рассмотрели пример преобразования простой, невложенной структуры JSON в объект DataFrame с помощью функции pandas `read_json()`. В этом разделе вы узнаете, как выполнить подобные преобразования сложного, вложенного JSON-документа, такого как этот:

```
data = [{"Emp":"Jeff Russell",
   "POs":[{"Pono":2608,"Total":35},
          {"Pono":2617,"Total":35},
          {"Pono":2620,"Total":139}
   ]},
   {"Emp":"Jane Boorman",
   "POs":[{"Pono":2621,"Total":95},
         {"Pono":2626,"Total":218}
   ]
}]
```

Как видите, каждая запись в документе JSON начинается с простой структурной пары «ключ — значение» с ключом `Emp`, за которой следует вложенная структура с ключом `POs`. Иерархическую структуру JSON можно преобразовать в pandas DataFrame с помощью метода чтения библиотеки pandas `json_normalize()`, который принимает вложенную структуру и делает ее плоской, или *нормализует*, превращая в простую таблицу. Вот как это делается:

```
import json
import pandas as pd
df = pd.json_normalize(❶ data, ❷ "POs", ❸ "Emp").set_index([❹ "Emp","Pono"])
print(df)
```

Помимо экземпляра JSON ❶, который будет обработан `json_normalize()`, мы указываем `POs` как вложенный массив ❷, который необходимо сделать плоским, и `Emp` в качестве поля, которое будет использоваться как часть сложного индекса в итоговой таблице ❸. В той же строке кода мы задаем два столбца в качестве индекса: `Emp` и `Pono` ❹. Результатом будет следующий датафрейм pandas:

```
                    Total
Emp             Pono
Jeff Russell    2608     35
                2617     35
```

```
                2620    139
Jane Boorman    2621     95
                2626    218
```

ПРИМЕЧАНИЕ

Использование двухколоночного индекса упрощает агрегирование данных внутри групп. Более подробно мы рассмотрим датафреймы с многоколоночными индексами в главе 6.

Конвертирование DataFrame в JSON

На практике часто приходится выполнять обратную операцию — преобразование pandas DataFrame в JSON. Следующий код преобразует датафрейм из примера обратно в экземпляр JSON, из которого он был первоначально сгенерирован:

```
❶ df = df.reset_index()
  json_doc = (❷ df.groupby(['Emp'], as_index=True)
              ❸ .apply(lambda x: x[['Pono','Total']].to_dict('records'))
              ❹ .reset_index()
              ❺ .rename(columns={0:'POs'})
              ❻ .to_json(orient='records'))
```

Начинаем с удаления двухколоночного индекса датафрейма, чтобы `Emp` и `Pono` стали обычными столбцами ❶. Затем используем составной однострочник для преобразования DataFrame в документ JSON. Сначала применим к датафрейму операцию `groupby`, группируя строки по столбцу `Emp` ❷. Используем `groupby()` в сочетании с `apply()` для применения лямбда-функции к каждой записи в каждой группе ❸. Внутри лямбда-функции укажем список полей для отображения в строке вложенного массива для каждой записи `Emp`. Применим метод `DataFrame.to_dict()` с параметром `records`, чтобы отформатировать поля в массиве следующим образом: `[{`*колонка*`:`*значение*`}, ... , {`*колонка*`:`*значение*`}]`, где каждый словарь — заказ, связанный с конкретным сотрудником.

На данном этапе у нас есть объект Series с индексом `Emp` и столбцом, содержащим массив заказов, связанных с сотрудником. Чтобы дать колонке название (в данном случае `POs`), необходимо преобразовать объект Series в DataFrame. Простой способ сделать это — метод `reset_index()` ❹. Помимо преобразования Series в DataFrame, `reset_index()` изменяет `Emp`, так что это больше не индекс, а обычный столбец, что будет важно при преобразовании датафрейма в формат JSON. Наконец, мы явно задаем имя столбца, содержащего вложенный массив (`POs`), используя метод `rename()` ❺, и конвертируем измененный DataFrame в JSON ❻.

УПРАЖНЕНИЕ № 7: ОБРАБОТКА СЛОЖНЫХ СТРУКТУР JSON

JSON, использованный в предыдущем разделе, имел одно поле с простой структурой (`Emp`) на верхнем уровне каждой записи. В реальном JSON-документе таких полей может быть больше. В записях в примере ниже есть второе простое поле на верхнем уровне — `Emp_email`:

```
data = [{"Emp":"Jeff Russell",
   "Emp_email":"jeff.russell",
   "POs":[{"Pono":2608,"Total":35},
          {"Pono":2617,"Total":35},
          {"Pono":2620,"Total":139}
   ]},
   {"Emp":"Jane Boorman",
   "Emp_email":"jane.boorman",
   "POs":[{"Pono":2621,"Total":95},
         {"Pono":2626,"Total":218}
    ]
}]
```

Чтобы преобразовать эти данные в DataFrame, необходимо передать список всех полей верхнего уровня с простой структурой в третий параметр `json_normalize()`, как показано ниже:

```
df = pd.json_normalize(data, "POs",
["Emp","Emp_email"]).set_index(["Emp","Emp_email","Pono"])
```

Содержимое датафрейма будет таким:

```
                                    Total
Emp            Emp_email      Pono
Jeff Russell   jeff.russell   2608     35
                              2617     35
                              2620    139
Jane Boorman   jane.boorman   2621     95
                              2626    218
```

Попробуйте преобразовать этот датафрейм обратно в исходный документ JSON, изменив операцию `groupby`, рассмотренную в предыдущем разделе.

Содержимое `json_doc` выглядит следующим образом:

```
[{"Emp": "Jeff Russell",
      "POs": [{"Pono": 2608, "Total": 35},
         {"Pono": 2617, "Total": 35},
         {"Pono": 2620, "Total": 139}
      ]},
   {"Emp": "Jane Boorman",
      "POs": [{"Pono": 2621, "Total": 95},
         {"Pono": 2626, "Total": 218}
      ]
}]
```

Для удобочитаемости можно вывести документ на экран с помощью следующей команды:

```
print(json.dumps(json.loads(json_doc), indent=2))
```

Преобразование онлайн-данных в DataFrame с помощью pandas-datareader

Некоторые сторонние библиотеки имеют совместимые с pandas методы чтения для получения доступа к данным из различных онлайн-источников, таких как Quandl[1] и Stooq[2]. Самая популярная подобная библиотека — pandas-datareader. На момент написания книги эта библиотека включала 70 методов для загрузки данных из определенного источника и их преобразования в pandas DataFrame. Многие методы библиотеки являются обертками для финансовых API, позволяя легко получать финансовые данные в формате pandas.

Установка pandas-datareader

Запустите следующую команду для установки pandas-datareader:

```
$ pip install pandas-datareader
```

Описание методов чтения библиотеки можно найти в документации pandas-datareader[3]. Кроме того, можно вывести на экран список доступных методов с помощью функции `dir()`:

[1] https://data.nasdaq.com

[2] https://stooq.com

[3] https://pandas-datareader.readthedocs.io/en/latest/remote_data.html

```
import pandas_datareader.data as pdr
print(dir(pdr))
```

Получение данных из Stooq

В следующем примере мы используем метод `get_data_stooq()` для получения данных об индексах S&P500 за указанный период:

```
import pandas_datareader.data as pdr
spx_index = pdr.get_data_stooq('^SPX', '2022-01-03', '2022-01-10')
print(spx_index)
```

Метод `get_data_stooq()` получает данные со Stooq, бесплатного сайта, который предоставляет информацию по ряду рыночных индексов. В качестве первого параметра передайте тикер нужного рыночного индекса. Доступные опции можно найти на сайте[1].

Полученные данные обычно имеют следующий формат:

```
                Open      High       Low     Close       Volume
Date
2022-01-10   4655.34   4673.02   4582.24   4670.29   2668776356
2022-01-07   4697.66   4707.95   4662.74   4677.03   2414328227
2022-01-06   4693.39   4725.01   4671.26   4696.05   2389339330
2022-01-05   4787.99   4797.70   4699.44   4700.58   2810603586
2022-01-04   4804.51   4818.62   4774.27   4793.54   2841121018
2022-01-03   4778.14   4796.64   4758.17   4796.56   2241373299
```

Столбец `Date` по умолчанию устанавливается в качестве индекса датафрейма.

Выводы

В этой главе вы узнали, как получать данные из различных источников и добавлять их в Python-скрипты для дальнейшей обработки. В частности, мы рассмотрели, как импортировать данные из файлов с помощью встроенных функций Python, как отправлять HTTP-запросы к онлайн-API из скриптов и как использовать преимущества объекта pandas reader для получения данных в разном формате из различных источников. Вы также научились экспортировать данные в файлы и преобразовывать DataFrame в JSON.

[1] https://stooq.com/t

5

Работа с базами данных

База данных (БД) — это организованная коллекция данных, к которой легко получить доступ, управлять ею и обновлять. Даже если изначально в архитектуре проекта нет БД, данные, проходящие через приложение, скорее всего, в какой-то момент будут обращаться к одной или нескольким базам.

Продолжая обсуждение импортирования данных в Python-приложение, в этой главе мы рассмотрим, как работать с данными из БД. Приведенные примеры продемонстрируют, как получить доступ к данным, хранящимся в базах различных типов, и обрабатывать их. Мы также исследуем, как управлять БД, в которых язык SQL является основным инструментом работы с данными, и базами, для которых он таковым не является. Вы узнаете, как использовать Python для взаимодействия с несколькими популярными базами данных: MySQL, Regis и MongoDB.

Базы данных предоставляют множество преимуществ. Например, с их помощью можно сохранять данные между вызовами скрипта и эффективно обмениваться информацией между различными приложениями. Более того, языки баз данных помогают систематизировать данные и отвечать на вопросы о них. Кроме того, многие системы БД позволяют внедрять программный код в саму базу, что может повысить производительность, модульность и возможность повторного использования приложения. Например, в базе данных можно хранить *trigger* — часть кода, которая автоматически вызывается каждый раз, когда происходит конкретное событие, например добавление новой строки в таблицу.

Базы данных могут быть реляционными и нереляционными (NoSQL). Реляционные базы данных имеют строго определенную структуру, реализованную в виде схемы для хранимых данных. Такой подход помогает обеспечить целостность, последовательность и общую точность данных. Однако у него есть и недостаток: реляционные БД плохо масштабируются при увеличении объема данных. А базы данных NoSQL, напротив, не накладывают ограничений на структуру хранимых данных, что обеспечивает большую гибкость, адаптивность и масштабируемость. В этой главе мы рассмотрим хранение и поиск данных как в реляционных, так и в нереляционных базах данных.

Реляционные базы данных

Реляционные базы данных, также известные как *базы данных из строк и столбцов*, — самый распространенный сегодня тип БД. Они обеспечивают структурированное хранение данных. Подобно тому как список книг на Amazon имеет определенную структуру для хранения информации (с полями для названий книг, авторов, описаний, рейтингов и т. д.), данные, хранящиеся в реляционной базе данных, должны соответствовать заранее определенной формальной схеме. Работа с реляционной БД начинается с разработки формальной схемы: вы определяете набор таблиц, каждая из которых состоит из нескольких полей (столбцов), и указываете, какой тип данных будет храниться в каждом поле. Также необходимо установить отношения между таблицами. После этого можно сохранять данные в базе, получать их или обновлять по мере необходимости.

Реляционные базы данных предназначены для оперативной вставки, обновления и/или удаления малых и больших объемов структурированных данных. Существует множество приложений, для которых такой тип БД будет лучшим выбором. В частности, реляционные базы данных хорошо подходят для систем *обработки транзакций в реальном времени* (OLTP, online transaction processing), которые обрабатывают большой объем операций множества пользователей.

Среди распространенных систем реляционных баз данных — MySQL, MariaDB, PostgreSQL. В этом разделе мы рассмотрим MySQL, — пожалуй, самую популярную в мире базу данных с открытым исходным кодом. На ее примере мы проиллюстрируем, как взаимодействовать с БД.

Вы узнаете, как настроить MySQL, создать новую базу данных, определить ее структуру и написать Python-скрипты для сохранения данных в базе и их получения.

Понимание инструкций SQL

SQL, или *язык структурированных запросов* (Structured Query Language), — основной инструмент взаимодействия с реляционной базой данных. Хотя наша цель — изучить принципы работы с БД при помощи Python, сам код Python должен содержать инструкции SQL. В этой книге мы не будем подробно рассматривать SQL, но краткое введение в этот язык запросов все же необходимо.

Инструкции SQL — это текстовые команды, распознаваемые и выполняемые ядром базы данных, например MySQL. Так, следующая инструкция SQL сообщает БД, что нужно извлечь все строки из таблицы `orders`, в поле `status` которых установлено значение `Shipped`:

```
SELECT * FROM orders WHERE status = 'Shipped';
```

Инструкции SQL обычно состоят из трех основных компонентов: *операции*, которая должна быть выполнена, *цели* этой операции и *условия*, которое сужает область действия операции. В предыдущем примере `SELECT` — это операция SQL, то есть мы получаем доступ к строкам из базы данных. Таблица `orders` является целью операции, которая определяется с помощью оператора `FROM`, а условие следует за оператором `WHERE`. Все SQL-инструкции должны иметь операцию и цель, но условие не обязательно. Следующая инструкция, например, не содержит условия, поэтому она извлекает все строки из таблицы `orders`:

```
SELECT * FROM orders;
```

Можно уточнить инструкцию так, чтобы она касалась только определенных столбцов таблицы. Например, получить только столбцы `pono` и `date` для всех строк таблицы `orders` можно так:

```
SELECT pono, date FROM orders;
```

По соглашению зарезервированные в SQL слова, такие как `SELECT` и `FROM`, пишутся прописными буквами. Однако язык SQL нечувствителен к регистру, поэтому заглавные буквы — не строгая необходимость. В конце каждой инструкции должна стоять точка с запятой.

Операции `SELECT`, подобные тем, что описаны выше, являются примерами инструкций *языка манипулирования данными* (DML, Data Manipulation Language). Этот тип инструкций используется для доступа к данным базы и манипулирования ими. К инструкциям DML также относятся: `INSERT`, `UPDATE` и `DELETE`, которые,

соответственно, добавляют, изменяют и удаляют записи из базы данных. Также существуют инструкции *языка определения данных* (DDL, Data Definition Language). Это еще одна распространенная категория инструкций SQL. Они предназначены для определения структуры базы данных. Стандартные операции DDL, `CREATE`, `ALTER` и `DROP`, используются для создания, изменения и удаления контейнеров данных соответственно, будь то столбцы, таблицы или целые базы данных.

Начало работы с MySQL

MySQL доступен на большинстве современных операционных систем, включая Linux, Unix, Windows и macOS. Существуют как бесплатные, так и коммерческие версии. Для целей данной главы достаточно MySQL Community Edition[1], бесплатной версии MySQL, которая доступна по лицензии GPL. Подробную инструкцию по установке MySQL для вашей операционной системы см. в справочном руководстве для последней версии MySQL[2].

Чтобы запустить сервер MySQL после установки, необходимо использовать команду, указанную в руководстве по установке для вашей операционной системы. Затем можно подключиться к серверу MySQL через терминал/консоль, используя клиент *mysql*:

```
$ mysql -uroot
```

ПРИМЕЧАНИЕ

На macOS может понадобиться использовать весь путь до MySQL, например: `/usr/local/mysql/bin/mysql -uroot -p`.

Вам будет предложено ввести пароль, который вы задали в процессе установки сервера MySQL. После этого вы увидите командную строку MySQL:

```
mysql>
```

Можно задать новый пароль для пользователя root с помощью следующей команды SQL:

```
mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'your_new_pswd';
```

[1] https://www.mysql.com/products/community

[2] https://dev.mysql.com/doc

Теперь можно создавать базу данных для приложения.

Введите следующую команду в командной строке `mysql>`:

```
mysql> CREATE DATABASE sampledb;
Query OK, 1 row affected (0.01 sec)
```

Эта команда создает базу данных под названием `sampledb`. Далее необходимо выбрать БД:

```
mysql> USE sampledb;
Database changed
```

Все последующие команды будут применяться к базе данных `sampledb`.

Определение структуры базы данных

Структура реляционной базы данных формируется на основе входящих в нее таблиц и из связей между ними. Поля, которые связывают таблицы, называются *ключами*. Существует два типа ключей: *первичный ключ* (primary key) и *внешний ключ* (foreign key). Первичный ключ служит для уникальной идентификации записи в таблице. Внешний ключ — это поле в другой таблице, которое соответствует первичному ключу в первой таблице. Как правило, первичный ключ и соответствующий ему внешний ключ имеют одинаковое имя в обеих таблицах.

> **ПРИМЕЧАНИЕ**
>
> Термины «поле» и «столбец» часто используются как взаимозаменяемые. Строго говоря, столбец становится полем, когда вы ссылаетесь на него в контексте одной строки.

Теперь, когда мы создали базу данных `sampledb`, можно создать несколько таблиц и определить их структуру. Для наглядности создадим таблицы с той же структурой, что и датафреймы pandas, с которыми мы работали в главе 3. Вот три табличные структуры данных для реализации в нашей БД:

```
emps

empno   empname         job
-----------------------------
9001    Jeff Russell    sales
9002    Jane Boorman    sales
```

```
9003    Tom Heints      sales

salary

empno   salary
---------------
9001    3000
9002    2800
9003    2500

orders

pono    empno   total
--------------------
2608    9001    35
2617    9001    35
2620    9001    139
2621    9002    95
2626    9002    218
```

Чтобы проанализировать, какие отношения можно установить между этими структурами, вернитесь к рис. 3.4 и 3.6. Как видно на рис. 3.4, строки в таблицах `emps` и `salary` связаны отношением «один-к-одному». Связь устанавливается через поле `empno`. Таблицы `emps` и `orders` тоже связаны через поле `empno`. Это связь типа «один-ко-многим», как видно на рис. 3.6.

Можно добавить эти структуры данных в реляционную БД, введя следующие SQL-команды в командной строке `mysql>`. Начнем с создания таблицы сотрудников `emps`:

```
mysql> CREATE TABLE emps (
        empno INT NOT NULL,
        empname VARCHAR(50),
        job VARCHAR(30),
        PRIMARY KEY (empno)
       );
```

С помощью команды `CREATE TABLE` создаем таблицу, определяя каждый столбец, его тип и, по желанию, размер данных, которые будем в нем хранить. Например, в столбце `empno` должны храниться целые числа (тип `INT`), и примененное к этой колонке ограничение `NOT NULL` гарантирует, что не получится вставить строку с пустым значением в поле `empno`. В свою очередь, столбец `empname` может хранить строки (тип `VARCHAR`) длиной до 50 символов, а `job` — строки длиной до 30 символов. Также указываем, что `empno` — столбец первичного ключа таблицы. Это означает, что он не должен содержать повторяющихся значений.

При успешном выполнении этой команды вы увидите следующее сообщение:

```
Query OK, 0 rows affected (0.03 sec)
```

Аналогично создадим таблицу с размерами оклада (`salary`):

```
mysql> CREATE TABLE salary (
        empno INT NOT NULL,
        salary INT,
        PRIMARY KEY (empno)
       );

Query OK, 0 rows affected (0.05 sec)
```

Далее добавляем ограничение внешнего ключа по столбцу `empno` таблицы `salary`, ссылающегося на столбец `empno` таблицы `emps`:

```
mysql> ALTER TABLE salary ADD FOREIGN KEY (empno) REFERENCES emps (empno);
```

Эта команда создает связь между таблицами `salary` и `empno`, то есть номер сотрудника в таблице `salary` должен соответствовать номеру сотрудника в таблице `emps`. Это ограничение гарантирует, что не получится вставить строку в таблицу `salary`, если она не имеет соответствующей строки в таблице `emps`.

Поскольку в таблице `salary` пока нет строк, операция `ALTER TABLE` не затрагивает ни одной строки, что видно из полученного сообщения:

```
Query OK, 0 rows affected (0.14 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Наконец, создаем таблицу с заказами (`orders`):

```
mysql> CREATE TABLE orders (
        pono INT NOT NULL,
        empno INT NOT NULL,
        total INT,
        PRIMARY KEY (pono),
        FOREIGN KEY (empno) REFERENCES emps (empno)
       );

Query OK, 0 rows affected (0.13 sec)
```

На этот раз мы добавляем ограничение внешнего ключа внутри команды `CREATE TABLE`, тем самым определяя внешний ключ сразу после создания таблицы.

Вставка данных в БД

Теперь все готово, чтобы вставлять строки во вновь созданные таблицы. Хотя это можно сделать в командной строке `mysql>`, данная операция обычно выполняется из приложения. Мы будем взаимодействовать с базой данных из Python-кода через драйвер MySQL Connector/Python. Его можно установить с помощью команды `pip`, как показано ниже:

```
$ pip install mysql-connector-python
```

Запустите следующий скрипт, чтобы заполнить таблицы БД данными:

```
import mysql.connector

try:
❶ cnx = mysql.connector.connect(user='root', password='your_pswd',
                                host='127.0.0.1',
                                database='sampledb')
❷ cursor = cnx.cursor()
  # объявление строк с сотрудниками
❸ emps = [
    (9001, "Jeff Russell", "sales"),
    (9002, "Jane Boorman", "sales"),
    (9003, "Tom Heints", "sales")
  ]
  # объявление запроса
❹ query_add_emp = ("""INSERT INTO emps (empno, empname, job)
                      VALUES (%s, %s, %s)""")
  # вставка строк с сотрудниками
  for emp in emps:
  ❺ cursor.execute(query_add_emp, emp)
  # определение и вставка размеров оклада
  salary = [
    (9001, 3000),
    (9002, 2800),
    (9003, 2500)
  ]
  query_add_salary = ("""INSERT INTO salary (empno, salary)
                         VALUES (%s, %s)""")
  for sal in salary:
    cursor.execute(query_add_salary, sal)
  # объявление и вставка заказов
```

```
    orders = [
      (2608, 9001, 35),
      (2617, 9001, 35),
      (2620, 9001, 139),
      (2621, 9002, 95),
      (2626, 9002, 218)
    ]
    query_add_order = ("""INSERT INTO orders(pono, empno, total)
                          VALUES (%s, %s, %s)""")
    for order in orders:
      cursor.execute(query_add_order, order)
    # делаем вставку в БД постоянной
    ❻ cnx.commit()
❼ except mysql.connector.Error as err:
      print("Error-Code:", err.errno)
      print("Error-Message: {}".format(err.msg))
❽ finally:
      cursor.close()
      cnx.close()
```

В скрипте мы импортируем драйвер MySQL Connector/Python как `mysql.connector`. Затем открываем блок `try/except`, который предоставляет шаблон для любых операций, связанных с базой данных, выполняемых в скрипте. Код для операции пишем в блоке `try`, и если при ее выполнении возникает ошибка, срабатывает переход в блок `except`.

Прежде всего, в блоке `try` устанавливаем соединение с базой данных, указывая имя пользователя, пароль, IP-адрес хоста (в данном случае локального хоста) и имя БД ❶. Затем получаем объект `cursor`, связанный с этим соединением ❷. Объект `cursor` предоставляет средства для выполнения инструкции, а также интерфейс для получения результатов.

Определяем строки для таблицы `emps` как список кортежей ❸. Затем объявляем инструкцию SQL для вставки этих строк в таблицу ❹. В инструкции `INSERT` указываем поля, которые должны быть заполнены данными, а также заполнители `%s`, которые сопоставляют эти поля с элементами каждого кортежа. Выполняем инструкцию в цикле, вставляя строки по одной с помощью метода `cursor.execute()` ❺. Аналогичным образом вставляем строки в таблицы `salary` и `orders`. В конце блока `try` делаем все вставки в базу данных с помощью метода `commit()` ❻.

Если какая-либо операция, связанная с базой данных, завершается сбоем, остальная часть блока `try` пропускается и выполняется пункт `except` ❼, выводя код ошибки, сгенерированный сервером MySQL, и соответствующее сообщение об ошибке.

Блок `finally` выполняется в любом случае ❽. В этом блоке мы явно закрываем `cursor`, а затем и соединение.

Запрос к базе данных

Теперь, когда мы заполнили таблицы информацией, можно запросить эти данные, чтобы дальше использовать их в Python-коде. Допустим, требуется получить все строки из таблицы `emps`, где значение поля `empno` больше `9001`. Для этого в качестве образца будем использовать скрипт из предыдущего раздела, изменив только блок `try` следующим образом:

```
  --фрагмент--
  try:
    cnx = mysql.connector.connect(user='root', password='your_pswd',
                                  host='127.0.0.1',
                                  database='sampledb')
      cursor = cnx.cursor()
      query = ("SELECT ❶ * FROM emps WHERE ❷ empno > %s")
❸  empno = 9001
❹  cursor.execute(query, (empno,))
❺  for (empno, empname, job) in cursor:
       print("{}, {}, {}".format(
         empno, empname, job))
  --фрагмент--
```

В отличие от операции вставки, для выбора строк не нужно выполнять операцию `cursor.execute()` в цикле для каждой строки. Вместо этого мы пишем запрос, указывающий критерии для выбора строк, а затем получаем их все разом с помощью одной операции `cursor.execute()`.

В инструкции `SELECT`, которая формирует наш запрос, мы указываем символ звездочки (*), это означает, что необходимо отобразить все поля извлеченных строк ❶. В блоке `WHERE` прописывается условие, которому должна удовлетворять выбранная строка. Здесь мы указываем, что у данной строки значение поля `empno` должно быть больше, чем значение переменной, связанной с заполнителем `%s` ❷. Во время выполнения переменная `empno` связана с заполнителем ❸. Когда мы делаем запрос с помощью `cursor.execute()`, мы передаем связанную переменную из кортежа в качестве второго параметра ❹. Метод `execute()` требует, чтобы связанные переменные передавались в кортеже или в словаре, даже если необходима только одна переменная.

Доступ к полученным строкам осуществляется через итерацию по объекту `cursor` с помощью цикла. Каждая строка хранится в виде кортежа, элементы которого

представляют собой значения полей этой строки ❺. Значения полей выводятся строка за строкой:

```
9002, Jane Boorman, sales
9003, Tom Heints, sales
```

Можно написать инструкции SELECT, которые объединяют строки из разных таблиц. Объединение таблиц реляционной базы данных повторяет процесс объединения датафреймов pandas, который был описан в главе 3. Обычно таблицы объединяются с помощью отношений внешнего ключа, определяемого при настройке базы данных.

Предположим, что необходимо объединить таблицы emps и salary, сохраняя условие о том, что значение empno должно быть больше 9001. Мы делаем это через их общие столбцы (empno), поскольку определили это поле в таблице salary как внешний ключ, ссылающийся на empno в таблице emps. Это соединение можно реализовать с помощью еще одной модификации блока try внутри скрипта:

```
--фрагмент--
try:
  cnx = mysql.connector.connect(user='root', password='your_pswd',
                                host='127.0.0.1',
                                database='sampledb')
  cursor = cnx.cursor()
  query = ("""SELECT ❶ e.empno, e.empname, e.job, s.salary
              FROM ❷ emps e JOIN salary s ON ❸ e.empno = s.empno
              WHERE ❹ e.empno > %s""")

  empno = 9001
  cursor.execute(query, (empno,))
  for (empno, empname, job, salary) in cursor:
    print("{}, {}, {}, {}".format(
      empno, empname, job, salary))
--фрагмент--
```

На этот раз запрос содержит оператор SELECT, объединяющий таблицы emps и salary. В блоке SELECT перечисляются столбцы из обеих таблиц, которые необходимо включить в объединение ❶. В блоке FROM расположены ключевое слово JOIN и названия таблиц для объединения, вместе с короткими именами e и s, которые необходимы, чтобы различать столбцы с одинаковым именем в обеих таблицах ❷. В блоке ON мы определяем условие объединения, указывая, что значения в столбцах empno обеих таблиц должны совпадать ❸. В блоке WHERE, как и в предыдущем примере, используем заполнитель %s, чтобы установить минимальное значение empno ❹.

В результате выполнения скрипта выводятся строки с зарплатой каждого сотрудника вместе записью из таблицы `emps`:

```
9002, Jane Boorman, sales, 2800
9003, Tom Heints, sales, 2500
```

УПРАЖНЕНИЕ № 8: ОБЪЕДИНЕНИЕ «ОДИН-КО-МНОГИМ»

Измените код из предыдущего раздела, так чтобы запрос объединял таблицу `emps` с таблицей `orders`. Можете сохранить условие о том, что значение `empno` должно превышать `9001`. Измените вызов функции `print()` так, чтобы выводить строки, полученные в результате нового объединения.

Использование инструментов аналитики баз данных

При сохранении данных в MySQL можно воспользоваться встроенными в БД средствами аналитики, такими как аналитический SQL, чтобы значительно сократить объем данных, передаваемых между приложением и БД. *Аналитический SQL* — это дополнительный набор команд SQL, предназначенный не просто для хранения, извлечения и обновления данных, а для их фактического анализа. В качестве примера представим, что нам нужно импортировать биржевые данные только тех компаний, чьи акции упали в цене не более чем на 1% по сравнению с предыдущим днем заданного периода. Этот предварительный анализ выполняется с помощью аналитического SQL, что избавляет от необходимости загружать из БД в скрипт весь датасет с ценами на акции.

Чтобы посмотреть, как это работает, получим биржевые данные с помощью библиотеки yfinance, представленной в главе 3, и сохраним их в таблице БД. Затем запросим таблицу из Python-скрипта, загрузив только ту часть данных, которая удовлетворяет заданному условию. Начнем с создания таблицы в нашей БД `sampledb`, которая хранит биржевые данные. В таблице должны быть три колонки: `ticker` (тикер), `date` (дата) и `price` (цена). Введите следующую команду в командной строке `mysql>`:

```
mysql> CREATE TABLE stocks(
        ticker VARCHAR(10),
        date VARCHAR(10),
        price DECIMAL(15,2)
       );
```

Используем этот скрипт для получения биржевых данных с yfinance:

```
   import yfinance as yf
❶ data = []
❷ tickers = ['TSLA', 'FB', 'ORCL', 'AMZN']
for ticker in tickers:
❸ tkr = yf.Ticker(ticker)
   hist = tkr.history(period='5d')
   ❹ .reset_index()
❺ records = hist[['Date','Close']].to_records(index=False)❻ records =
list(records)
   records = [(ticker, ❼ str(elem[0])[:10], round(elem[1],2)) for elem in
records]
❽ data = data + records
```

Сначала определяем пустой список `data`, который будет заполнен биржевыми данными ❶. Как мы уже видели выше в этой главе, метод `cursor.execute()` в блоке `INSERT` ожидает получить данные в виде списка (list). Затем определяем список тикеров, по которым хотим извлечь данные ❷. После этого в цикле передаем каждый тикер из списка `tickers` в функцию yfinance `Ticker()` ❸. Функция возвращает объект `Ticker`. Его метод `history()` предоставляет данные, связанные с соответствующим тикером. В этом примере мы получаем данные об акциях для каждого тикера за пять последних рабочих дней (`period='5d'`).

Метод `history()` возвращает данные о котировках в виде pandas DataFrame со столбцом `Date` в качестве индекса. Наконец, преобразуем этот датафрейм в список кортежей для вставки в базу данных. Поскольку в датасет нам нужно включить столбец `Date`, удаляем его из индекса с помощью метода DataFrame `reset_index()`, тем самым превращая `Date` в обычный столбец ❹. Затем из полученного датафрейма выбираем только столбцы `Date` и `Close`, где значение поля `Close` содержит цены акций на конец дня, и преобразуем эти столбцы в массив записей NumPy. Это промежуточный шаг в процессе преобразования входных данных ❺. После этого превращаем данные в список кортежей ❻. Вслед за этим необходимо переформатировать каждый кортеж, чтобы его можно было вставить в таблицу базы данных `stocks` в виде строки. Например, значения колонки `Date` содержат много лишней информации (часы, минуты, секунды и т. д.). Взяв только первые 10 символов поля 0 в каждом кортеже, мы извлечем год, месяц и день, чего вполне достаточно для анализа ❼. К примеру, `2022-01-06T00:00:00.000000000` превратится просто в `2022-01-06`. Наконец, все еще внутри цикла, добавляем кортежи, связанные с тикером, в список `data` ❽.

В результате содержимое списка кортежей `data` будет выглядеть следующим образом:

```
[
 ('TSLA', '2022-01-06', 1064.7),
 ('TSLA', '2022-01-07', 1026.96),
 ('TSLA', '2022-01-10', 1058.12),
 ('TSLA', '2022-01-11', 1064.4),
 ('TSLA', '2022-01-12', 1106.22),
 ('FB', '2022-01-06', 332.46),
 ('FB', '2022-01-07', 331.79),
 ('FB', '2022-01-10', 328.07),
 ('FB', '2022-01-11', 334.37),
 ('FB', '2022-01-12', 333.26),
 ('ORCL', '2022-01-06', 86.34),
 ('ORCL', '2022-01-07', 87.51),
 ('ORCL', '2022-01-10', 89.28),
 ('ORCL', '2022-01-11', 88.48),
 ('ORCL', '2022-01-12', 88.31),
 ('AMZN', '2022-01-06', 3265.08),
 ('AMZN', '2022-01-07', 3251.08),
 ('AMZN', '2022-01-10', 3229.72),
 ('AMZN', '2022-01-11', 3307.24),
 ('AMZN', '2022-01-12', 3304.14)
]
```

Чтобы вставить этот датасет в таблицу `stocks` в виде строк, добавьте следующий код к предыдущему скрипту и выполните его заново:

```
import mysql.connector
from mysql.connector import errorcode
try:
   cnx = mysql.connector.connect(user='root', password='your_pswd',
                                 host='127.0.0.1',
                                 database='sampledb')
   cursor = cnx.cursor()
   # объявление запроса
   query_add_stocks = ("""INSERT INTO stocks (ticker, date, price)
                          VALUES (%s, %s, %s)""")
   # добавление строк с ценами на акции
❶  cursor.executemany(query_add_stocks, data)
   cnx.commit()
except mysql.connector.Error as err:
   print("Error-Code:", err.errno)
   print("Error-Message: {}".format(err.msg))
finally:
   cursor.close()
   cnx.close()
```

Код строится по той же схеме, что и запрос для вставки данных в БД, приведенный выше. Однако на этот раз мы используем метод `cursor.executemany()`, который позволяет эффективно выполнить инструкцию `INSERT` для каждого кортежа из списка `data` ❶.

Теперь, когда в БД есть данные, можно составлять к ней различные запросы с помощью аналитического SQL, чтобы отвечать на поставленные вопросы. Например, чтобы отфильтровать акции, цена которых упала более чем на 1% по сравнению с ценой в предыдущие дни, как было предложено в начале этого раздела, нам понадобится запрос, который может анализировать цены на один и тот же тикер в течение нескольких дней. В качестве первого шага создадим датасет, который в одной строке будет хранить как текущую цену акции, так и ее цену за предыдущий день. Введите следующий код в командной строке `mysql>`:

```
SELECT
   date,
   ticker,
   price,
   LAG(price) OVER(PARTITION BY ticker ORDER BY date) AS prev_price
FROM stocks;
```

Функция `LAG()` внутри `SELECT` является аналитической функцией SQL. Она позволяет получить доступ к данным предыдущей строки из текущей строки. Блок `PARTITION BY` в `OVER` делит датасет на группы по тикеру. Функция `LAG()` применяется отдельно в каждой группе; это гарантирует, что данные не будут перетекать из одного тикера в другой. Результат запроса будет выглядеть примерно так:

```
+------------+-----------+---------+------------+
| date       |   ticker  | price   | prev_price |
+------------+-----------+---------+------------+
| 2022-01-06 |   AMZN    | 3265.08 |      NULL  |
| 2022-01-07 |   AMZN    | 3251.08 |   3265.08  |
| 2022-01-10 |   AMZN    | 3229.72 |   3251.08  |
| 2022-01-11 |   AMZN    | 3307.24 |   3229.72  |
| 2022-01-12 |   AMZN    | 3304.14 |   3307.24  |
| 2022-01-06 |   FB      |  332.46 |      NULL  |

| 2022-01-07 |   FB      |  331.79 |    332.46  |
| 2022-01-10 |   FB      |  328.07 |    331.79  |
| 2022-01-11 |   FB      |  334.37 |    328.07  |
| 2022-01-12 |   FB      |  333.26 |    334.37  |
| 2022-01-06 |   ORCL    |   86.34 |      NULL  |
```

```
| 2022-01-07 |   ORCL    |   87.51 |     86.34  |
| 2022-01-10 |   ORCL    |   89.28 |     87.51  |
| 2022-01-11 |   ORCL    |   88.48 |     89.28  |
| 2022-01-12 |   ORCL    |   88.31 |     88.48  |
| 2022-01-06 |   TSLA    | 1064.70 |      NULL  |
| 2022-01-07 |   TSLA    | 1026.96 |   1064.70  |
| 2022-01-10 |   TSLA    | 1058.12 |   1026.96  |
| 2022-01-11 |   TSLA    | 1064.40 |   1058.12  |
| 2022-01-12 |   TSLA    | 1106.22 |   1064.40  |
+------------+-----------+---------+------------+
20 rows in set (0.00 sec)
```

Запрос создал новый столбец `prev_price` с ценой акций за предыдущий день. Как видите, `LAG()`совмещает в одной строке данные сразу двух строк, то есть можно управлять данными из обеих строк одним математическим выражением в виде части запроса. Например, можно разделить одну цену на другую, чтобы рассчитать изменение цены ото дня ко дню в процентах. Исходя из этого требования, напишем запрос, отбирающий строки только тех тикеров, чьи цены упали не более чем на 1% по сравнению с предыдущими днями заданного периода:

```
❶ SELECT s.* FROM stocks AS s
LEFT JOIN
❷ (SELECT DISTINCT(ticker) FROM
    ❸ (SELECT
        ❹ price/LAG(price) OVER(PARTITION BY ticker ORDER BY date) AS dif,
          ticker
   ❺ WHERE dif <0.99) AS a
❻ ON a.ticker = s.ticker
❼ WHERE a.ticker IS NULL;
```

Инструкция SQL объединяет два различных запроса к одной и той же таблице — `stocks`. Первый join извлекает все строки из таблицы `stocks` ❶, а второй — только названия тикеров, цены которых упали на 1% или более по сравнению с предыдущим днем хотя бы один раз за период анализа ❷. Этот join имеет сложную структуру: он выбирает данные из подзапроса, а не непосредственно из таблицы `stocks`. Подзапрос, который начинается с ❸, извлекает из таблицы те строки, значения поля `price` которых по крайней мере на 1% меньше, чем в предыдущей строке. Мы определяем это, разделив `price` на `LAG(price)` ❹ и проверив, меньше ли результат, чем `0.99` ❺. Затем, внутри списка `SELECT` основного запроса к полю `ticker`, применяется функция `DISTINCT()` для исключения дублируемых названий тикеров из итогового множества ❷.

Мы объединяем запросы по столбцу `ticker` ❻. В блоке `WHERE` сообщаем join, что нужно получить только те строки, в которых не найдено соответствия между

полем `a.ticker` (тикеры, цена которых упала более чем на 1%) и полем `s.ticker` (все тикеры) ❼. Поскольку используется left join, будут получены только совпадающие строки из первого запроса. В результате join возвращает все строки таблицы `stocks` с тикером, не найденным среди тикеров, полученных из второго запроса.

Учитывая биржевые данные, показанные выше, датасет, полученный в результате запроса, выглядит следующим образом:

```
+--------+------------+---------+
| ticker | date       | price   |
+--------+------------+---------+
| ORCL   | 2022-01-06 |   86.34 |
| ORCL   | 2022-01-07 |   87.51 |
| ORCL   | 2022-01-10 |   89.28 |
| ORCL   | 2022-01-11 |   88.48 |
| ORCL   | 2022-01-12 |   88.31 |
| AMZN   | 2022-01-06 | 3265.08 |
| AMZN   | 2022-01-07 | 3251.08 |
| AMZN   | 2022-01-10 | 3229.72 |
| AMZN   | 2022-01-11 | 3307.24 |
| AMZN   | 2022-01-12 | 3304.14 |
+-------- + ------------ + -------- +
10 rows in set (0.00 sec)
```

Как видите, из таблицы `stocks` извлечены не все строки. В частности, вы не найдете строк, связанных с тикерами FB и TSLA. Последний, к примеру, не был взят, поскольку в выводе предыдущего запроса была получена следующая строка:

```
+------------+--------+---------+------------+
| date       | ticker | price   | prev_price |
+------------+--------+---------+------------+
  ...
  2022-01-07 | TSLA   | 1026.96 | 1064.70    |
  ...
```

Здесь видно снижение на 3.54%, что превышает порог в 1%.

В следующем фрагменте кода мы выполняем тот же запрос из Python и получаем результаты в формате pandas DataFrame:

```
import pandas as pd
import mysql.connector
from mysql.connector import errorcode
```

```
try:
   cnx = mysql.connector.connect(user='root', password='your_pswd',
                                 host='127.0.0.1',
                                 database='sampledb')
   query = ("""
      SELECT s.* FROM stocks AS s
      LEFT JOIN
        (SELECT DISTINCT(ticker) FROM
          (SELECT
            price/LAG(price) OVER(PARTITION BY ticker ORDER BY date) AS dif,
            ticker
          FROM stocks) AS b
        WHERE dif <0.99) AS a
      ON a.ticker = s.ticker
      WHERE a.ticker IS NULL""")
❶  df_stocks = pd.read_sql(query, con=cnx)
❷  df_stocks = df_stocks.set_index(['ticker','date']) except
               mysql.connector.Error as err:
   print("Error-Code:", err.errno)
   print("Error-Message: {}".format(err.msg))
finally:
   cnx.close()
```

В основном скрипт похож на приведенный выше в этой главе. Ключевое отличие в том, что данные из БД загружаются непосредственно в pandas DataFrame. Для этого используется метод pandas `read_sql()`, который принимает SQL-запрос в виде строки в качестве первого параметра и объект подключения к базе данных в качестве второго ❶. Затем мы устанавливаем столбцы `ticker` и `date` в качестве индекса датафрейма ❷.

Исходя из данных, представленных выше, итоговый датафрейм `df_stocks` будет выглядеть так:

```
                       price
ticker   date
ORCL     2022-01-06     86.34
         2022-01-07     87.51
         2022-01-10     89.28
         2022-01-11     88.48
         2022-01-12     88.31
AMZN     2022-01-06   3265.08
         2022-01-07   3251.08
         2022-01-10   3229.72
         2022-01-11   3307.24
         2022-01-12   3304.14
```

Теперь, когда данные хранятся в формате DataFrame, можно приступить к дальнейшему анализу с помощью Python. Например, рассчитать среднюю цену каждого тикера за определенный период. В следующей главе вы увидите, как решать такие задачи, применяя соответствующую агрегатную функцию на уровне групп датафрейма.

Базы данных NoSQL

Базы данных NoSQL, или *нереляционные базы данных*, не требуют заранее определенной схемы организации хранимых данных; кроме того, они не поддерживают стандартные операции реляционных БД, например объединение. Зато они предоставляют способы хранения информации с большей структурной гибкостью, что облегчает работу с большими объемами данных. Хранилища «ключ — значение» (один из типов баз данных NoSQL) позволяют хранить и извлекать данные в виде пар «ключ — значение», например время — событие. Документно-ориентированные базы данных — еще один тип баз данных NoSQL — предназначены для работы с контейнерами данных с гибкой структурой, такими как документы JSON. Они позволяют хранить всю информацию, относящуюся к определенному объекту, в виде одной записи в БД, а не разбивать данные по нескольким таблицам, как в реляционных базах.

Хотя базы данных NoSQL существуют не так давно, как их реляционные аналоги, они быстро набрали популярность, поскольку позволяют разработчикам хранить данные в простых понятных форматах и не требуют высокой квалификации для доступа к данным и их обработки. Благодаря своей гибкости, они особенно хорошо подходят для приложений реального времени и для приложений, обрабатывающих большие объемы данных, таких как Google Gmail или LinkedIn.

ПРИМЕЧАНИЕ

О происхождении термина NoSQL нет единого мнения. Одни считают, что оно обозначает non-SQL, а другие — что название происходит от not only SQL. Уместны оба варианта: базы данных NoSQL хранят данные в формате, отличном от реляционных (SQL) таблиц, и в то же время многие из этих баз данных поддерживают запросы типа SQL.

Хранилища «ключ — значение»

Хранилище «ключ — значение» — это база данных, которая хранит пары «ключ — значение», подобно словарю Python. Хороший пример такого типа хранилищ —

Redis (сокращение от Remote Dictionary Service). Redis поддерживает такие команды, как `GET`, `SET` и `DEL`, для доступа к парам «ключ — значение» и выполнения операций с ними, как показано в простом примере ниже:

```
$ redis-cli
127.0.0.1:6379> SET emp1 "Maya Silver"
OK
127.0.0.1:6379> GET emp1
"Maya Silver"
```

Здесь мы используем команду `SET` для создания ключа `emp1` со значением `Maya Silver`, а затем применяем `GET` для получения значения через его ключ.

Установка Redis

Чтобы самостоятельно изучить Redis, необходимо установить его. Более подробную информацию можно найти на странице быстрого запуска Redis[1]. После установки сервера Redis в систему также потребуется установить redis-py, библиотеку Python, которая позволяет взаимодействовать с Redis из Python-скрипта. Это можно сделать с помощью команды `pip`:

```
$ pip install redis
```

Затем импортируйте redis-py в свой скрипт командой `import redis`.

Доступ к Redis с помощью Python

Ниже приведен простой пример доступа к серверу Redis с помощью Python из библиотеки redis-py:

```
  > import redis
❶ > r = redis.Redis()
❷ > r.mset({"emp1": "Maya Silver", "emp2": "John Jamison"})
  True
❸ > r.get("emp1")
  b'Maya Silver'
```

[1] https://redis.io/topics/quickstart

Используем метод `redis.Redis()` для установки соединения с сервером Redis ❶. Поскольку параметры метода не указаны, они будут установлены по умолчанию, то есть предполагается, что сервер запущен на локальной машине:

```
host='localhost', port=6379 и db=0.
```

ПРИМЕЧАНИЕ

Redis нумерует базы данных с помощью индексации с нуля. Новые соединения по умолчанию используют базу данных с индексом 0.

После установки соединения используем метод `mset()`, чтобы задать несколько пар «ключ — значение» ❷ (*m* — сокращение от *multiple*). Сервер возвращает `True`, если данные успешно сохранены. Затем с помощью метода `get()` можно получить значение любого из хранимых ключей ❸.

Как и любая другая БД, Redis позволяет сохранять добавляемые данные. Это означает, что по ключу можно будет получить соответствующее значение в другой сессии Python или в другом скрипте. Redis также позволяет установить *флаг истечения срока действия* (expire flag) для ключа при задании пары. Он будет указывать, как долго ее следует хранить. Это может быть особенно полезно в приложениях реального времени, где входные данные теряют актуальность через какое-то время. Например, если вы разрабатываете приложение для заказа такси, вам, возможно, потребуется хранить данные о доступности каждой отдельной машины. Поскольку эти данные будут часто меняться, необходимо ограничить срок их действия. Вот как можно это сделать:

```
--snip--
> from datetime import timedelta
> r.setex("cab26", timedelta(minutes=1), value="in the area now")
True
```

Используем метод `setex()` для задания пары «ключ — значение», которая будет автоматически удалена из базы данных через определенное время. В нем мы указываем время истечения срока действия в виде объекта `timedelta`. Другой способ — задать число в секундах.

До сих пор мы рассматривали только простые пары «ключ — значение», но в Redis можно хранить и несколько фрагментов информации об одном и том же объекте, как показано ниже:

```
> cabDict = {"ID": "cab48", "Driver": "Dan Varsky", "Brand": "Volvo"}
> r.hmset("cab48", cabDict)
> r.hgetall("cab48")
{'Cab': 'cab48', 'Driver': 'Dan Varsky', 'Brand': 'Volvo'}
```

Сначала мы определяем словарь Python, который может содержать произвольное количество пар «ключ — значение». Затем отправляем весь этот словарь в базу данных, сохраняя его под ключом `cab48` с помощью функции `hmset()` (*h* — сокращение от *hash*). После этого используем функцию `hgetall()` для получения всех пар «ключ — значение», хранящихся под ключом `cab48`.

Документоориентированные базы данных

Документоориентированная база данных хранит каждую запись как отдельный документ. Вместо соответствия заранее определенной схеме, такой как поля таблицы реляционной базы данных, каждый документ в документоориентированной БД может иметь свою собственную структуру. Подобная гибкость сделала эти базы данных наиболее популярным типом БД NoSQL. А самая популярная документоориентированная база данных — MongoDB. Она предназначена для управления коллекциями документов вида JSON. В этом разделе мы посмотрим, как с ней работать.

Установка MongoDB

Существует несколько способов работы с MongoDB. Первый — установить ее в свою систему. Чтобы узнать, как это сделать, обратитесь к документации MongoDB[1]. Другой вариант не требует установки. Он заключается в создании бесплатной платформы с MongoDB с помощью MongoDB Atlas. Для этого необходимо зарегистрироваться на сайте MongoDB Atlas[2].

Прежде чем начать работу с базой данных MongoDB из Python, необходимо установить PyMongo, официальный драйвер Python для MongoDB. Это можно сделать с помощью команды `pip`:

```
$ pip install pymongo
```

[1] https://docs.mongodb.com/manual/installation

[2] https://www.mongodb.com/cloud/atlas/register

Доступ к MongoDB с помощью Python

Первый шаг к работе с MongoDB на Python — установка соединения с сервером БД через объект PyMongo `MongoClient`, как показано ниже:

```
> from pymongo import MongoClient
> client = MongoClient('connection_string')
```

Строкой подключения (`connection_string`) может быть URI-адрес подключения к MongoDB, например *mongodb://localhost:27017*. Для ее использования необходимо установить MongoDB в локальной системе. А при использовании MongoDB Atlas потребуется вставить строку подключения, предоставленную Atlas. Подробнее см. на странице Connect via Driver документации Atlas[1], а также на странице Connection String URI Format[2].

Вместо строки подключения можно указать хост и порт в виде отдельных параметров конструктора `MongoClient()`:

```
> client = MongoClient('localhost', 27017)
```

Один экземпляр MongoDB может поддерживать несколько баз данных, поэтому после установки соединения с сервером необходимо указать ту базу, с которой вы хотите работать. MongoDB не предоставляет отдельную команду для создания БД, поэтому для создания новой базы данных и доступа к существующей используется один и тот же синтаксис. Например, чтобы создать БД с именем `sampledb` (или получить к ней доступ, если она уже существует), можно использовать синтаксис обращения к словарю:

```
> db = client['sampledb']
```

или синтаксис обращения к атрибуту:

```
> db = client.sampledb
```

В отличие от реляционных баз данных, MongoDB не хранит данные в таблицах. Документы группируются в *коллекции*. Создание коллекции и доступ к ней аналогичны созданию БД и доступу к ней:

[1] https://docs.atlas.mongodb.com/driver-connection

[2] https://docs.mongodb.com/manual/reference/connection-string

```
> emps_collection = db['emps']
```

Эта команда создаст коллекцию `emps` в базе данных `sampledb`, если она еще не была создана. Затем можно использовать метод `insert_one()` для вставки документа в коллекцию. В примере ниже мы вставляем документ `emp`, отформатированный как словарь:

```
> emp = {"empno": 9001,
...         "empname": "Jeff Russell",
...         "orders": [2608, 2617, 2620]}
> result = emps_collection.insert_one(emp)
```

При вставке документа в него автоматически добавляется поле `"_id"`. Значение этого поля формируется таким образом, чтобы оно было уникальным в рамках всей коллекции.

Доступ к ID можно получить через поле `inserted_id` объекта, возвращаемого командой `insert_one()`:

```
> result.inserted_id
ObjectId('69y67385ei0b650d867ef236')
```

Теперь БД содержит данные, но как их запрашивать? Самый распространенный способ — запрос методом `find_one()`, который возвращает единственный документ, соответствующий критериям поиска:

```
> emp = emps_collection.find_one({"empno": 9001})
> print(emp)
```

Как видите, `find_one()` не требует ID документа, он был автоматически добавлен при вставке. Вместо этого можно запросить конкретные элементы, которым, как предполагается, соответствует документ.

Результат будет выглядеть примерно так:

```
{
  u'empno': 9001,
  u'_id': ObjectId('69y67385ei0b650d867ef236'),
  u'empname': u'Jeff Russell',
  u'orders': [2608, 2617, 2620]
}
```

УПРАЖНЕНИЕ № 9: ВСТАВКА И ЗАПРОС НЕСКОЛЬКИХ ДОКУМЕНТОВ

В предыдущем разделе вы научились вставлять один документ в MongoDB и извлекать его. Продолжите работу с коллекцией `emps`, созданной в базе данных `sampled`, и попробуйте выполнить вставку сразу нескольких документов с помощью метода `insert_many()`, а затем запросите более одного документа с помощью метода `find()`. Более подробная информация об этих методах содержится в документации PyMongo[1].

Выводы

В этой главе мы рассмотрели примеры помещения данных в БД и их извлечения из БД разных типов — реляционных и NoSQL. Мы поработали с MySQL, одной из самых популярных реляционных баз данных. Затем рассмотрели Redis, хранилище типа NoSQL, которое позволяет эффективно сохранять и извлекать пары «ключ — значение». Мы также познакомились с MongoDB, пожалуй, самой популярной на сегодняшний день базой данных NoSQL для хранения документов, и научились работать с документами формата JSON с использованием синтаксиса языка Python.

[1] https://pymongo.readthedocs.io/en/stable

6

Агрегирование данных

Зачастую данные необходимо агрегировать для того, чтобы принимать взвешенные решения на их основе. *Агрегирование* — это процесс сбора данных для их представления в обобщенном виде путем группировки по промежуточным, итоговым, средним или другим статистическим показателям. В этой главе мы рассмотрим методы агрегирования, встроенные в pandas, и обсудим, как их можно использовать для анализа данных.

Агрегирование — эффективный способ получить общую картину большого датасета и лучше понять его значения. Например, крупному розничному бизнесу может быть интересно узнать показатели брендов или посмотреть на общие объемы продаж в разных регионах. Владельцу сайта может понадобиться определить наиболее привлекательные материалы, основываясь на количестве посетителей. Климатологу может потребоваться определить самые солнечные места в регионе на основе среднего количества солнечных дней в году.

Агрегирование помогает ответить на подобные вопросы, обеспечивая объединение отдельных значений и их представление в обобщенном виде. Поскольку агрегирование представляет информацию на основе связанных кластеров данных, предполагается, что первым шагом должна стать группировка этих данных по одному или нескольким признакам. Например, в случае крупного розничного продавца сгруппировать данные можно по брендам или по регионам и датам.

В приведенных ниже примерах вы увидите, как реализовать группировку с помощью *агрегатных функций* при работе со строками структуры pandas DataFrame. Для каждой группы строк агрегатная функция возвращает единую строку с совокупным результатом.

Данные для агрегирования

Чтобы посмотреть, как работает агрегирование, создадим набор датафреймов, содержащих информацию о продажах модной верхней одежды в онлайн-магазине. Эти данные будут включать номера и даты заказов; подробную информацию о товарах из каждого заказа (например, цену и количество); сотрудников, выполняющих каждый заказ, и местонахождение складов компании. В реальном приложении эти данные, скорее всего, будут храниться в БД, к которой можно будет подключаться из скрипта, как описано в главе 5. Для простоты мы будем просто преобразовывать данные из списков кортежей в объект DataFrame. Список кортежей можно загрузить из репозитория GitHub для этой книги.

Рассмотрим примеры заказов. Список с именем `orders` содержит несколько кортежей, каждый из которых представляет один заказ. Каждый кортеж содержит три поля: номер заказа, дату и уникальный номер (ID) сотрудника, который выполнил заказ:

```
orders = [
 (9423517, '2022-02-04', 9001),
 (4626232, '2022-02-04', 9003),
 (9423534, '2022-02-04', 9001),
 (9423679, '2022-02-05', 9002),
 (4626377, '2022-02-05', 9003),
 (4626412, '2022-02-05', 9004),
 (9423783, '2022-02-06', 9002),
 (4626490, '2022-02-06', 9004)
]
```

Начнем с импортирования библиотеки pandas и преобразования списка в DataFrame:

```
import pandas as pd
df_orders = pd.DataFrame(orders, columns =['OrderNo', 'Date', 'Empno'])
```

Детали заказа (также известные как *строка заказа*) обычно хранятся в другом контейнере данных. В данном случае у нас есть список кортежей с именем

`details`, который мы преобразуем в еще один датафрейм. Каждый кортеж представляет собой строку заказа с полями, соответствующими номеру заказа, названию товара, бренду, цене и количеству:

```
details = [
 (9423517, 'Jeans', 'Rip Curl', 87.0, 1),
 (9423517, 'Jacket', 'The North Face', 112.0, 1),
 (4626232, 'Socks', 'Vans', 15.0, 1),
 (4626232, 'Jeans', 'Quiksilver', 82.0, 1),
 (9423534, 'Socks', 'DC', 10.0, 2),
 (9423534, 'Socks', 'Quiksilver', 12.0, 2),
 (9423679, 'T-shirt', 'Patagonia', 35.0, 1),
 (4626377, 'Hoody', 'Animal', 44.0, 1),
 (4626377, 'Cargo Shorts', 'Animal', 38.0, 1),
 (4626412, 'Shirt', 'Volcom', 78.0, 1),
 (9423783, 'Boxer Shorts', 'Superdry', 30.0, 2),
 (9423783, 'Shorts', 'Globe', 26.0, 1),
 (4626490, 'Cargo Shorts', 'Billabong', 54.0, 1),
 (4626490, 'Sweater', 'Dickies', 56.0, 1)
]
# преобразуем список в DataFrame
df_details = pd.DataFrame(details, columns =['OrderNo', 'Item', 'Brand',
'Price', 'Quantity'])
```

Информацию о сотрудниках компании будем хранить в третьем датафрейме. Создадим его из списка кортежей `emps`, который содержит номера, имена и местоположение сотрудников:

```
emps = [
 (9001, 'Jeff Russell', 'LA'),
 (9002, 'Jane Boorman', 'San Francisco'),
 (9003, 'Tom Heints', 'NYC'),
 (9004, 'Maya Silver', 'Philadelphia')
]

df_emps = pd.DataFrame(emps, columns =['Empno', 'Empname', 'Location'])
```

Наконец, у нас есть информация о городе и регионе каждого склада. Она хранится в списке кортежей с именем `locations`. Из этих данных формируем четвертый датафрейм:

```
locations = [
 ('LA', 'West'),
 ('San Francisco', 'West'),
 ('NYC', 'East'),
```

```
 ('Philadelphia', 'East')
]

df_locations = pd.DataFrame(locations, columns =['Location', 'Region'])
```

После преобразования данных в формат DataFrame их можно по-разному агрегировать, что позволит ответить на всевозможные вопросы о состоянии бизнеса. Например, можно просмотреть показатели продаж по различным регионам, генерируя промежуточные итоги на конкретную дату. Для этого сначала нужно объединить соответствующие данные в один датафрейм, затем распределить их по группам и применить к получившимся группам агрегатные функции.

Объединение датафреймов

Необходимые для агрегирования данные часто приходится собирать из множества различных контейнеров. Наш пример не исключение. Даже данные заказов распределены между двумя датафреймами: `df_orders` и `df_details`. Наша цель — получить суммы продаж по регионам и по датам. Какие датафреймы нужно объединить и какие именно столбцы из них взять?

Поскольку нам нужен общий объем продаж, из `df_details` нужно взять колонки `Price` (цена) и `Quantity` (количество), из `df_orders` колонку `Date` (дата), а из `df_locations` — `Region` (регион). Следовательно, необходимо объединить следующие датафреймы: `df_orders`, `df_details` и `df_locations`.

Датафреймы `df_orders` и `df_details` можно объединить непосредственно вызовом метода pandas `merge()`, как показано ниже:

```
df_sales = df_orders.merge(df_details)
```

ПРИМЕЧАНИЕ

Чтобы разобраться, как работает `merge()`, обратитесь к разделу «Объединения «один-ко-многим»» в главе 3.

Мы объединяем датафреймы по столбцу `OrderNo` (номер заказа). Указывать это в явном виде необязательно, поскольку `OrderNo` присутствует в обоих датафреймах и будет выбран по умолчанию. Объединенный датафрейм теперь содержит одну запись для каждой строки заказа, как в `df_details`, но теперь вместе с информацией, добавленной из соответствующих записей из `df_orders`.

Чтобы увидеть записи в объединенном датафрейме, можно просто вывести его на экран:

```
print(df_sales)
```

Содержимое `df_sales` будет выглядеть следующим образом:

	OrderNo	Date	Empno	Item	Brand	Price	Quantity
0	9423517	2022-02-04	9001	Jeans	Rip Curl	87.0	1
1	9423517	2022-02-04	9001	Jacket	The North Face	112.0	1
2	4626232	2022-02-04	9003	Socks	Vans	15.0	1
3	4626232	2022-02-04	9003	Jeans	Quiksilver	82.0	1
4	9423534	2022-02-04	9001	Socks	DC	10.0	2
5	9423534	2022-02-04	9001	Socks	Quiksilver	12.0	2
6	9423679	2022-02-05	9002	T-shirt	Patagonia	35.0	1
7	4626377	2022-02-05	9003	Hoody	Animal	44.0	1
8	4626377	2022-02-05	9003	Cargo Shorts	Animal	38.0	1
9	4626412	2022-02-05	9004	Shirt	Volcom	78.0	1
10	9423783	2022-02-06	9002	Boxer Shorts	Superdry	30.0	2
11	9423783	2022-02-06	9002	Shorts	Globe	26.0	1
12	4626490	2022-02-06	9004	Cargo Shorts	Billabong	54.0	1
13	4626490	2022-02-06	9004	Sweater	Dickies	56.0	1

Как видно из столбца `Quantity`, в одной строке заказа может содержаться более одного экземпляра изделия, поэтому необходимо перемножить значения полей `Price` и `Quantity`, чтобы вычислить общую сумму заказа. Результат перемножения можно сохранить в новом поле датафрейма:

```
df_sales['Total'] = df_sales['Price'] * df_sales['Quantity']
```

Этот код добавит новый столбец `Total` (общая сумма) к датафрейму, в дополнение к семи существующим. Теперь у нас есть возможность удалить столбцы, которые не понадобятся для генерации суммы продаж по регионам и датам. На данном этапе необходимо сохранить только столбцы `Date`, `Total` и `Empno` (номер сотрудника). Первые два очевидно нужны для расчетов. А необходимость `Empno` мы обсудим в ближайшее время.

Чтобы отфильтровать датафрейм, сохранив только нужные столбцы, передадим список названий столбцов в оператор `[]` датафрейма, как показано ниже:

```
df_sales = df_sales[['Date','Empno','Total']]
```

Теперь нужно объединить датафрейм `df_sales`, который мы только что создали, и датафрейм `df_regions`. Однако не получится объединить их напрямую, потому что у них нет общих столбцов. Вместо этого объединим их через датафрейм `df_emps`, который имеет один общий столбец с `df_sales` и один — с `df_regions`. В частности, `df_sales` и `df_emps` можно объединить по столбцу `Empno`, именно поэтому мы сохранили данный столбец в `df_sales`, а `df_emps` и `df_locations` объединяются по столбцу `Location` (местоположение). Реализуем объединение с помощью метода `merge()`:

```
df_sales_emps = df_sales.merge(df_emps)
df_result = df_sales_emps.merge(df_locations)
```

Выводим датафрейм `df_result`:

```
     Date        Empno   Total        Empname        Location Region
0    2022-02-04   9001    87.0   Jeff Russell              LA   West
1    2022-02-04   9001   112.0   Jeff Russell              LA   West
2    2022-02-04   9001    20.0   Jeff Russell              LA   West
3    2022-02-04   9001    24.0   Jeff Russell              LA   West
4    2022-02-04   9003    15.0     Tom Heints             NYC   East
5    2022-02-04   9003    82.0     Tom Heints             NYC   East
6    2022-02-05   9003    44.0     Tom Heints             NYC   East
7    2022-02-05   9003    38.0     Tom Heints             NYC   East
8    2022-02-05   9002    35.0   Jane Boorman   San Francisco   West
9    2022-02-06   9002    60.0   Jane Boorman   San Francisco   West
10   2022-02-06   9002    26.0   Jane Boorman   San Francisco   West
11   2022-02-05   9004    78.0    Maya Silver    Philadelphia   East
12   2022-02-06   9004    54.0    Maya Silver    Philadelphia   East
13   2022-02-06   9004    56.0    Maya Silver    Philadelphia   East
```

И снова вы, возможно, захотите удалить ненужные столбцы и оставить только те, которые действительно полезны. На этот раз можно избавиться от `Empno`, `Empname` и `Location`, оставив только `Date`, `Region` и `Total`:

```
df_result = df_result[['Date','Region','Total']]
```

Теперь содержимое `df_result` выглядит следующим образом:

```
          Date Region   Total
0    2022-02-04   West    87.0
1    2022-02-04   West   112.0
2    2022-02-04   West    20.0
3    2022-02-04   West    24.0
```

```
4    2022-02-04   East    15.0
5    2022-02-04   East    82.0
6    2022-02-05   East    44.0
7    2022-02-05   East    38.0
8    2022-02-05   West    35.0
9    2022-02-06   West    60.0
10   2022-02-06   West    26.0
11   2022-02-05   East    78.0
12   2022-02-06   East    54.0
13   2022-02-06   East    56.0
```

Теперь, без лишних столбцов, датафрейм `df_result` идеально отформатирован для агрегирования данных о продажах по регионам и датам.

Группировка и агрегирование данных

Чтобы выполнить агрегированные расчеты на основе данных, нужно сначала распределить данные по соответствующим группам. Встроенная в pandas функция `groupby()` разбивает данные датафрейма на подмножества, которые имеют совпадающие значения в одном или нескольких столбцах. В нашем случае для группировки датафрейма `df_result` по дате и региону можно использовать `groupby()`. Затем мы применяем агрегатную функцию pandas `sum()` к каждой группе. Обе операции можно выполнить в одной строке кода:

```
df_date_region = df_result.groupby(['Date','Region']).sum()
```

Первая группа формируется на основе столбца `Date`. А затем для каждой даты создаем группы по столбцу `Region`. Функция `groupby()` возвращает объект, к которому затем применяется агрегатная функция `sum()`. Эта функция суммирует значения числовых столбцов. В данном примере `sum()` применяется только к столбцу `Total`, поскольку это единственный числовой столбец в датафрейме. (Если бы в датафрейме были другие числовые столбцы, функция агрегирования была бы применена и к ним.) В итоге датафрейм `df_result` будет выглядеть так:

```
                  Total
Date         Region
2022-02-04   East    97.0
             West   243.0
2022-02-05   East   160.0
             West    35.0
2022-02-06   East   110.0
             West    86.0
```

Оба столбца, `Date` и `Region`, являются индексными столбцами нового датафрейма. Вместе они образуют *иерархический индекс*, также известный как *многоуровневый индекс*, или просто *MultiIndex*.

MultiIndex позволяет работать с данными, имеющими произвольное количество измерений, в рамках двумерной структуры датафрейма, используя несколько столбцов для уникальной идентификации каждой строки. В нашем случае датафрейм `df_date_region` можно рассматривать как трехмерный датасет с тремя осями: дата, регион и агрегированное значение (каждая ось представляет соответствующее измерение), как показано в табл. 6.1.

Таблица 6.1. Три измерения датафрейма df_date_region

Ось	Координаты
Date	2022-02-04, 2022-02-05, 2022-02-06
Region	West, East
Aggregation	Total

ПРИМЕЧАНИЕ

В данном контексте координаты — возможные значения для данной оси.

MultiIndex нашего датафрейма позволяет писать запросы для перемещения по измерениям датафрейма и получения доступа к итоговым данным (`total`) по дате, региону или сразу обоим. Мы сможем однозначно идентифицировать каждую строку датафрейма и получить доступ к выбранным агрегированным значениям в различных группах данных.

Просмотр конкретных агрегированных показателей по MultiIndex

Просмотр определенных категорий информации внутри датафрейма — распространенный запрос. Например, из только что созданного датафрейма `df_date_region` может понадобиться получить агрегированные данные о продажах только на определенную дату или в конкретном регионе и на определенную дату одновременно. Для поиска требуемого агрегированного показателя можно использовать индекс датафрейма (или MultiIndex).

Чтобы понять, как работать с MultiIndex, полезно посмотреть, в каком виде каждое его значение представлено в Python. Для этого можно воспользоваться свойством `index` датафрейма `df_date_region`:

```
print(df_date_region.index)
```

Свойство `index` возвращает все значения индексов или метки строк датафрейма независимо от того, имеет этот датафрейм простой индекс или MultiIndex. Вот значения MultiIndex датафрейма `df_date_region`:

```
MultiIndex([('2022-02-04', 'East'),
            ('2022-02-04', 'West'),
            ('2022-02-05', 'East'),
            ('2022-02-05', 'West'),
            ('2022-02-06', 'East'),
            ('2022-02-06', 'West')],
           names=['Date', 'Region'])
```

Как видите, каждое значение MultiIndex представляет собой кортеж, который можно использовать для доступа к соответствующему значению в поле `Total`. Исходя из этого, получить доступ к общим показателям на определенную дату и регион можно следующим образом:

```
df_date_region❶ [df_date_region.index.isin(❷ [('2022-02-05', 'West')])]
```

Мы помещаем кортеж, представляющий собой нужный MultiIndex, в оператор `[]` ❷ и передаем его методу pandas `index.isin()`. Метод требует, чтобы передаваемый параметр был итерируемым (список, кортеж, серия, датафрейм или словарь), поэтому заключаем MultiIndex в квадратные скобки. Метод возвращает булев массив, указывающий, соответствуют ли данные в каждом из значений индекса датафрейма заданному значению(ям): при соответствии возвращается `True`, в противном случае — `False`. В данном примере метод `isin()` генерирует массив `[False, False, False, True, False, False]`, следовательно, совпадает четвертое значение индекса.

Затем мы передаем массив с булевыми значениями в датафрейм `df_date_region` внутри оператора `[]` ❶, и в результате выбирается соответствующий показатель продаж, как показано ниже:

```
                   Total
Date       Region
2022-02-05 West     35.0
```

Ограничений по количеству извлекаемых из датафрейма строк нет. В `index.isin()` можно передавать несколько индексов и получать соответствующие значения:

```
df_date_region[df_date_region.index.isin([('2022-02-05', 'East'), ('2022-02-05',
'West')])]
```

Код выше вернет следующие строки из `df_date_region`:

```
                  Total
Date       Region
2022-02-05 East    160.0
           West     35.0
```

Хотя в данном примере используются два соседних индекса, в `index.isin()` можно передавать любые индексы в произвольном порядке, например так:

```
df_date_region[df_date_region.index.isin([('2022-02-06', 'East'), ('2022-02-04',
'East'), ('2022-02-05', 'West')])]
```

Набор извлеченных строк будет выглядеть следующим образом:

```
                  Total
Date       Region
2022-02-04 East     97.0
2022-02-05 West     35.0
2022-02-06 East    110.0
```

Обратите внимание, что порядок полученных записей соответствует их порядку в датафрейме, а не порядку, в котором указаны индексы.

Срез диапазона агрегированных значений

Подобно тому как срез используется для получения диапазона значений из списка, его можно применять для извлечения диапазона агрегированных значений из датафрейма. Создать срез из датафрейма `df_date_region` можно, предоставив два кортежа, указывающих ключи MultiIndex начальной и конечной позиций диапазона среза. В следующем примере получим диапазон агрегированных значений по всем регионам с `2022-02-04` по `2022-02-05`. Просто поместим начальное и конечное значения MultiIndex в квадратные скобки и разделим их двоеточием:

```
df_date_region[('2022-02-04', 'East'):('2022-02-05', 'West')]
```

В результате получим следующие строки:

```
                  Total
Date       Region
2022-02-04 East     97.0
```

```
           West    243.0
2022-02-05 East    160.0
           West     35.0
```

Поскольку в данном конкретном примере извлекаются данные о продажах за указанный период времени по всем регионам, можно не указывать названия регионов и передавать только даты:

```
df_date_region['2022-02-04':'2022-02-05']
```

Этот код должен вывести точно такой же результат, как и в предыдущем примере.

Срезы на разных уровнях агрегирования

Возможно, вам понадобятся срезы агрегированных значений на различных уровнях согласно иерархическому индексу. В нашем примере самым высоким уровнем агрегации является `Date` (дата), внутри него находится уровень `Region` (регион). Допустим, нам нужно получить данные о продажах для определенного среза дат, выбрав все содержимое уровня `Region`. Это можно сделать с помощью функции Python `slice()` в сочетании со свойством датафрейма `loc`, как показано ниже:

```
df_date_region.loc[(slice('2022-02-05', '2022-02-06'), slice(None)), :]
```

Здесь мы дважды использовали функцию `slice()`. В первом случае `slice()` определяет диапазон среза по дате (`Date`), самому высокому уровню агрегации. В результате создается объект `slice`, указывающий начальную и конечную даты. Во второй раз `slice()` вызывается на уровне региона (`Region`). Это следующий, более низкий уровень. Указав `None`, мы выбираем все содержимое уровня `Region`. В оператор `[]` свойства `loc` также включается запятая, за которой следует двоеточие (`:`). Этот синтаксис указывает, что используются метки строк, а не столбцов.

В результате получим следующий набор строк:

```
                  Total
Date       Region
2022-02-05 East    160.0
           West     35.0
2022-02-06 East    110.0
           West     86.0
```

В следующем примере заменим `slice(None)` на `slice('East')`, тем самым сократив количество получаемых данных о продажах за указанный диапазон дат до строк, содержащих `East`:

```
df_date_region.loc[(slice('2022-02-05', '2022-02-06'), slice('East')), :]
```

Получим следующие строки:

```
                  Total
Date       Region
2022-02-05 East    160.0
2022-02-06 East    110.0
```

На уровне `Region` тоже можно указать не одно значение, а диапазон. Однако в данном конкретном примере этот диапазон может начинаться только с `'East'` и заканчиваться `'West'`, что реализуется как: `slice('East','West')`. Поскольку это единственный возможный диапазон, вызов `slice('East','West')` будет эквивалентен вызову `slice(None)`.

Добавление общего итога

Когда речь идет об агрегировании данных о продажах, в конечном счете может понадобиться рассчитать общий итог, или сумму всех итоговых значений продаж, и добавить его в датафрейм. Поскольку в нашем примере все итоговые значения находятся в одном датафрейме (`df_date_region`), можно использовать метод pandas `sum()` для нахождения общего объема продаж по всем регионам и всем датам. Данный метод вычисляет сумму значений по указанной оси, как показано ниже:

```
ps = df_date_region.sum(axis = 0)
print(ps)
```

Здесь `sum()` возвращает объект pandas Series с суммарными значениями по столбцу `Total` датафрейма `df_date_region`. Помните, что указывать столбец `Total` при вызове `sum()` нет необходимости, поскольку эта функция автоматически применяется к любым числовым данным. Содержимое серии `ps` будет следующим:

```
Total    731.0
dtype: float64
```

Чтобы добавить вновь созданную серию к датафрейму `df_date_region`, необходимо сначала дать ей имя. В датафрейме оно будет использоваться в качестве индекса строки с общим итогом. Поскольку ключи индексов в `df_date_region` являются кортежами, в качестве имени серии тоже нужно использовать кортеж:

```
ps.name=('All','All')
```

Первый `'All'` в кортеже относится к составляющей индекса `Date`, а второй — к компоненту `Region`. Теперь серию можно добавить в датафрейм:

```
df_date_region_total = df_date_region.append(ps)
```

Вывод на экран только что созданного датафрейма будет выглядеть так:

```
                  Total
Date       Region
2022-02-04 East     97.0
           West    243.0
2022-02-05 East    160.0
           West     35.0
2022-02-06 East    110.0
           West     86.0
All        All     731.0
```

Получить доступ к строке общего итога, как и к любой другой строке датафрейма, можно по ее индексу. Передадим кортеж, представляющий индекс строки, методу `index.isin()`, как делали ранее:

```
df_date_region_total[df_date_region_total.index.isin([('All', 'All')])]
```

В результате получим строку с общим итогом:

```
                  Total
Date       Region
All        All     731.0
```

Добавление промежуточных итогов

Помимо общего итога, можно добавить промежуточные итоги для каждой даты. В этом случае результирующий датафрейм будет выглядеть так:

```
                  Total
Date       Region
2022-02-04 East     97.0
           West    243.0
All        All     340.0
2022-02-05 East    160.0
           West     35.0
All        All     195.0
2022-02-06 East    110.0
           West     86.0
All        All     196.0
All        All     731.0
```

Создание такого датафрейма потребует нескольких шагов. Сначала мы группируем датафрейм по уровню `Date`. Затем проходим по полученному объекту GroupBy, обращаясь к каждой дате с соответствующим набором строк (известным как *субфрейм* (subframe)), содержащих информацию о регионе и сумме для этой даты. Выбираем и добавляем каждый субфрейм в пустой датафрейм вместе с соответствующей строкой промежуточного итога. Таким образом:

```
❶ df_totals = pd.DataFrame()
  for date, date_df in ❷ df_date_region.groupby(level=0):
  ❸ df_totals = df_totals.append(date_df)
  ❹ ps = date_df.sum(axis = 0)
  ❺ ps.name=(date,'All')
  ❻ df_totals = df_totals.append(ps)
```

Начинаем с создания пустого датафрейма `df_totals`, в который будем помещать итоговые данные ❶. Затем создаем объект GroupBy ❷, группирующий `df_date_region` по верхнему иерархическому уровню индекса (`level=0`), то есть по столбцу `Date`, и проходим в цикле `for` по объекту GroupBy. На каждой итерации получаем дату и соответствующий ей субфрейм. Добавляем субфрейм к датафрейму `df_totals`❸, а затем создаем строку с промежуточным итогом в виде серии, содержащей сумму строк субфрейма ❹. Затем присваиваем серии имя с соответствующей датой и `'All'`, чтобы обозначить все регионы ❺, после чего серия добавляется к датафрейму `df_totals` ❻.

Наконец, добавляем строку с итоговой суммой в датафрейм, как показано ниже:

```
df_totals = df_totals.append(df_date_region_total.loc[('All','All')])
```

В результате получаем датафрейм, который содержит как сумму продаж за каждую дату, так и общий итог за все даты.

УПРАЖНЕНИЕ № 10:
ИСКЛЮЧЕНИЕ ИЗ ДАТАФРЕЙМА СТРОК С ИТОГОВОЙ СУММОЙ

Наличие в DataFrame строк с суммарными значениями позволяет использовать его в качестве отчета без необходимости совершать с ним дополнительные действия. Однако если вы собираетесь использовать датафрейм в дальнейших операциях агрегирования, такие строки лучше исключить.

Попробуйте отфильтровать датафрейм `df_totals`, созданный в предыдущем разделе, исключив строки с общим и промежуточным итогом. Используйте методы срезов, рассмотренные в этой главе.

Выбор всех строк в группе

Помимо помощи в агрегировании, функция `groupby()` позволяет выбрать все строки, принадлежащие определенной группе. Для этого к объекту, возвращаемому `groupby()`, применяется метод `get_group()`. Вот как это работает:

```
group = df_result.groupby(['Date','Region'])
group.get_group(('2022-02-04','West'))
```

Мы группируем датафрейм `df_result` по `Date` и `Region`, передавая имена столбцов в `groupby()` в виде списка, как уже делали это ранее. Затем применяем метод `get_group()` к полученному объекту GroupBy, передавая кортеж с нужным индексом. В результате получаем следующий датафрейм:

```
         Date  Region   Total
0  2022-02-04    West    87.0
1  2022-02-04    West   112.0
2  2022-02-04    West    20.0
3  2022-02-04    West    24.0
```

Как видите, результирующие значения не являются агрегированными. Мы получили все строки с заказами, относящиеся к указанной дате и региону.

Выводы

В этой главе вы узнали, что агрегирование — это процесс сбора данных и их представления в обобщенном виде. Как правило, он включает разделение данных по группам и вычисление итоговых показателей по каждой группе. На примерах этой главы мы показали, как агрегировать данные, содержащиеся в pandas DataFrame, используя методы и свойства данной структуры, такие как `merge()`, `groupby()`, `sum()`, `index` и `loc`. Вы научились использовать преимущества иерархического индекса (или MultiIndex) датафрейма для моделирования многоуровневых отношений в агрегируемых данных. Вы также научились выборочно просматривать агрегированные данные и получать их срезы с помощью MultiIndex.

7

Объединение датасетов

Обычно данные распределяются по нескольким контейнерам, поэтому часто возникает необходимость объединить различные датасеты в один. В предыдущих главах мы уже выполняли некоторые действия по объединению, а в этой рассмотрим методы объединения датасетов более подробно.

В некоторых случаях такое объединение представляет собой просто добавление одного датасета в конец другого. Например, финансовый аналитик может каждую неделю получать новые биржевые данные, которые необходимо добавить к уже существующей коллекции подобных данных. В других случаях требуется более избирательное объединение нескольких датасетов, имеющих общий столбец, в один сводный набор данных. Например, ритейлеру нужно объединить общие данные об онлайн-заказах с конкретной информацией о товарах, как мы делали в главе 6. В любом случае после объединения данные готовы к дальнейшему анализу — с объединенным датасетом можно выполнять операции фильтрации, группировки или агрегирования.

Как вы узнали из предыдущих глав, датасеты в Python могут храниться в виде встроенных структур данных, таких как списки, кортежи и словари, или быть организованы в виде сторонних структур данных — массивов NumPy или pandas DataFrame. В последнем случае у вас будет более богатый набор инструментов для объединения данных и, следовательно, больше возможностей для удовлетворения определенных условий объединения. Однако это не означает, что успешно объединить встроенные структуры данных Python невозможно. В данной главе

будет показано, как это сделать. Мы также узнаем, как объединять сторонние структуры данных.

Объединение встроенных структур данных

Синтаксис для объединения встроенных структур данных Python довольно прост. В этом разделе вы узнаете, как объединять списки или кортежи с помощью оператора +, а также научитесь объединять словари с помощью оператора **. Кроме того, мы увидим, как выполнять объединение, агрегирование и другие операции со списками кортежей, по сути, рассматривая их как таблицы базы данных, где каждый кортеж представляет собой строку.

Объединение списков и кортежей с помощью оператора +

Самый простой способ объединения двух или более списков или кортежей — это оператор +. Операция сложения списков или кортежей записывается точно так же, как для числовых значений. Этот метод хорошо работает, когда нужно поместить элементы из объединяемых структур в одну новую структуру, не изменяя самих элементов. Такой процесс часто называют *конкатенацией*.

Для демонстрации вернемся к примеру с интернет-магазином модной одежды из предыдущей главы. Предположим, что информация о заказах, сделанных в течение дня, собирается в список, так что у нас есть отдельный список для каждого дня. Допустим, мы имеем три списка:

```
orders_2022_02_04 = [
 (9423517, '2022-02-04', 9001),
 (4626232, '2022-02-04', 9003),
 (9423534, '2022-02-04', 9001)
]
orders_2022_02_05 = [
 (9423679, '2022-02-05', 9002),
 (4626377, '2022-02-05', 9003),
 (4626412, '2022-02-05', 9004)
]
orders_2022_02_06 = [
 (9423783, '2022-02-06', 9002),
 (4626490, '2022-02-06', 9004)
]
```

Для дальнейшего анализа может понадобиться объединить эти списки в один. Оператор + упрощает эту задачу; мы просто складываем списки:

```
orders = orders_2022_02_04 + orders_2022_02_05 + orders_2022_02_06
```

Вот итоговый список `orders`:

```
[
 (9423517, '2022-02-04', 9001),
 (4626232, '2022-02-04', 9003),
 (9423534, '2022-02-04', 9001),
 (9423679, '2022-02-05', 9002),
 (4626377, '2022-02-05', 9003),
 (4626412, '2022-02-05', 9004),
 (9423783, '2022-02-06', 9002),
 (4626490, '2022-02-06', 9004)
]
```

Как видите, элементы всех трех исходных списков теперь находятся в одном списке, а их порядок определяется порядком их расположения в инструкции конкатенации. В этом конкретном примере все элементы объединяемых списков были кортежами. Однако оператор + может конкатенировать списки с элементами любого типа. Это означает, что с таким же успехом можно объединять списки с целыми числами, строками, словарями и т. д.

Тот же синтаксис (+) подходит и для объединения нескольких кортежей. Однако если попытаться использовать + для объединения словарей, получим ошибку `unsupported operand type(s)`. В следующем разделе мы разберем правильный синтаксис для объединения словарей.

*Объединение словарей с помощью оператора ***

Оператор ** разбивает, или распаковывает, словарь на отдельные пары «ключ — значение». Чтобы объединить два словаря в один, их нужно распаковать с помощью ** и сохранить результаты в новом словаре. Это работает, даже если один из словарей (или оба) имеет иерархическую структуру. В контексте нашего примера с онлайн-магазином рассмотрим словарь, который содержит дополнительные поля с информацией о заказе:

```
extra_fields_9423517 = {
   'ShippingInstructions' : { 'name'   : 'John Silver',
                              'Phone' :   [{ 'type' : 'Office', 'number' : '809-
123-9309' },
                                           { 'type' : 'Mobile', 'number' : '417-
123-4567' }
                                          ]}
}
```

Вложенная структура словаря понятна, поскольку в ней используются осмысленные имена ключей. Действительно, благодаря возможности доступа к данным по ключам, а не по позициям словари предпочтительнее списков при работе с иерархическими структурами данных.

Теперь предположим, что в другом словаре есть еще поля для того же заказа:

```
order_9423517 = {'OrderNo':9423517, 'Date':'2022-02-04', 'Empno':9001}
```

Наша задача — объединить эти словари так, чтобы новый словарь включал все пары «ключ — значение» двух исходных словарей. Воспользуемся оператором **:

```
order_9423517 = {**order_9423517, **extra_fields_9423517}
```

Мы помещаем словари, которые хотим объединить, внутрь фигурных скобок, добавляя перед каждым из них **, который распаковывает оба словаря, получая пары «ключ — значение», а затем фигурные скобки упаковывают их обратно в единый словарь. Теперь `order_9423517` выглядит так:

```
{
   'OrderNo': 9423517,
   'Date': '2022-02-04',
   'Empno': 9001,
   'ShippingInstructions': {'name': 'John Silver',
                            'Phone': [{'type': 'Office', 'number': '809-123-9309'},
                                      {'type': 'Mobile', 'number': '417-123-4567'}
                                     ]}
}
```

Как видите, все элементы из исходных словарей присутствуют и их иерархическая структура сохранена.

Объединение строк из двух структур

Вы уже знаете, как объединить несколько списков в один без изменения хранящихся в них элементов. На практике вам также часто придется объединять две или более структур данных, имеющих общий столбец, так, чтобы соответствующие строки из этих структуры данных сливались в одну. Если вы имеете дело со структурой данных pandas DataFrame, можете использовать такие методы, как `join()` и `merge()`, как мы делали в предыдущих главах. Однако если вы работаете со списками, содержащими «строки» кортежей, эти методы применить

не получится. Вместо этого необходимо проходить по спискам и присоединять каждую строку по отдельности.

В качестве иллюстрации объединим список `orders`, который мы создали ранее в этой главе, в разделе «Объединение списков и кортежей с помощью оператора +», со списком `details` из раздела «Данные для агрегирования» главы 6. Напомню, как выглядит тот список:

```
details = [
 (9423517, 'Jeans', 'Rip Curl', 87.0, 1),
 (9423517, 'Jacket', 'The North Face', 112.0, 1),
 (4626232, 'Socks', 'Vans', 15.0, 1),
 (4626232, 'Jeans', 'Quiksilver', 82.0, 1),
 (9423534, 'Socks', 'DC', 10.0, 2),
 (9423534, 'Socks', 'Quiksilver', 12.0, 2),
 (9423679, 'T-shirt', 'Patagonia', 35.0, 1),
 (4626377, 'Hoody', 'Animal', 44.0, 1),
 (4626377, 'Cargo Shorts', 'Animal', 38.0, 1),
 (4626412, 'Shirt', 'Volcom', 78.0, 1),
 (9423783, 'Boxer Shorts', 'Superdry', 30.0, 2),
 (9423783, 'Shorts', 'Globe', 26.0, 1),
 (4626490, 'Cargo Shorts', 'Billabong', 54.0, 1),
 (4626490, 'Sweater', 'Dickies', 56.0, 1)
]
```

Оба списка содержат кортежи, первым элементом которых является номер заказа. Цель — найти кортежи с совпадающими номерами заказов, объединить их в один кортеж и сохранить в списке. Вот как это сделать:

```
❶ orders_details = []
❷ for o in orders:
    for d in details:
    ❸ if d[0] == o[0]:
          orders_details.append(o + ❹ d[1:])
```

Сначала создается пустой список для хранения объединенных кортежей ❶. Затем мы используем вложенную пару циклов для итерации по спискам ❷ и оператор `if` ❸ для объединения кортежей с совпадающими порядковыми номерами. Чтобы избежать повторов номера заказа в кортеже при объединении и добавлении в список `orders_details`, из каждого кортежа внутри `details` берется срез ❹, с помощью которого выбираются все поля, кроме первого, номер заказа в котором будет избыточным.

Глядя на этот код, вы, вероятно, зададитесь вопросом, можно ли реализовать его более элегантно, одной строкой. Действительно, с помощью списковых включений можно добиться того же результата:

```
orders_details = [[o for o in orders if d[0] == o[0]][0] + d[1:] for d in
details]
```

Во внешнем списковом включении мы выполняем итерацию по кортежам списка `details`. Во внутреннем — находим в списке `orders` кортеж, номер заказа которого совпадает с текущим кортежем из `details`. Поскольку строке заказа из `details` должен соответствовать только один кортеж из `orders`, внутреннее списковое включение должно генерировать список с одним элементом (кортежем, представляющим заказ). Итак, берем первый элемент внутреннего спискового включения с помощью оператора `[0]`, а затем конкатенируем кортеж этого заказа с соответствующим кортежем из `details` с помощью оператора `+`, исключая лишний номер заказа (`[1:]`).

Независимо от того, создали мы `orders_details` посредством спискового включения или с помощью двух циклов `for`, как было показано выше, полученный список будет выглядеть следующим образом:

```
[
 (9423517, '2022-02-04', 9001, 'Jeans', 'Rip Curl', 87.0, 1),
 (9423517, '2022-02-04', 9001, 'Jacket', 'The North Face', 112.0, 1),
 (4626232, '2022-02-04', 9003, 'Socks', 'Vans', 15.0, 1),
 (4626232, '2022-02-04', 9003, 'Jeans', 'Quiksilver', 82.0, 1),
 (9423534, '2022-02-04', 9001, 'Socks', 'DC', 10.0, 2),
 (9423534, '2022-02-04', 9001, 'Socks', 'Quiksilver', 12.0, 2),
 (9423679, '2022-02-05', 9002, 'T-shirt', 'Patagonia', 35.0, 1),
 (4626377, '2022-02-05', 9003, 'Hoody', 'Animal', 44.0, 1),
 (4626377, '2022-02-05', 9003, 'Cargo Shorts', 'Animal', 38.0, 1),
 (4626412, '2022-02-05', 9004, 'Shirt', 'Volcom', 78.0, 1),
 (9423783, '2022-02-06', 9002, 'Boxer Shorts', 'Superdry', 30.0, 2),
 (9423783, '2022-02-06', 9002, 'Shorts', 'Globe', 26.0, 1),
 (4626490, '2022-02-06', 9004, 'Cargo Shorts', 'Billabong', 54.0, 1),
 (4626490, '2022-02-06', 9004, 'Sweater', 'Dickies', 56.0, 1)
]
```

Список содержит все кортежи из `details`, и каждый кортеж также содержит дополнительную информацию из соответствующего кортежа списка `orders`.

Реализация join-объединений списков

Операция, которую мы выполнили в предыдущем разделе, является стандартным объединением типа «один-ко-многим»: каждая строка заказа в `details` имеет соответствующий заказ в `orders`, а каждый заказ в `orders` — одну или несколько строк в `details`. Однако на практике любой из объединяемых датасетов

(или оба) может содержать строки, не имеющие совпадений в другом. Чтобы учесть такие ситуации, необходимо научиться выполнять операции объединения, эквивалентные тем, что используются в базах данных (мы обсуждали их в главе 3): left join, right join, inner join и outer join.

Например, список `details` может содержать строки заказов, которых нет в списке `orders`. Это может произойти, если `orders` фильтруется по определенному диапазону дат; поскольку `details` не содержит поля с датой, невозможно отфильтровать этот список соответствующим образом. Чтобы смоделировать эту ситуацию, добавьте в `details` строку с заказом, которого нет в `orders`:

```
details.append((4626592, 'Shorts', 'Protest', 48.0, 1))
```

Теперь попытаемся сгенерировать список `orders_details`, как мы делали раньше:

```
orders_details = [[o for o in orders if d[0] == o][0] + d[1:] for d in details]
```

Получаем ошибку:

```
IndexError: list index out of range
```

Проблема возникает, когда мы добираемся до номера заказа в списке `details`, для которого нет совпадений, и пытаемся получить первый элемент соответствующего внутреннего спискового включения. Такого элемента не существует, поскольку номер заказа отсутствует в списке `orders`. Один из способов решения этой проблемы — добавление блока `if` в цикл `for d in details` внутри внешнего спискового включения, проверяющего, есть ли номер заказа из `details` в какой-то из строк `orders`:

```
orders_details = [[o for o in orders if d[0] in o][0] + d[1:] for d in details
                ❶ if d[0] in [o[0] for o in orders ]]
```

Вы устраните проблему, исключив все строки `details`, не имеющие совпадений в `orders`, с помощью проверки в блоке `if` ❶, который следует за циклом `for d in details`. Таким образом, это списковое включение соответствует объединению типа inner join.

Но что, если требуется включить все строки `details` в итоговый список `orders_details`, например, для нахождения итоговой суммы всех заказов, а не только заказов из текущего списка `orders` (который гипотетически был отфильтрован

по дате)? Затем можно найти итоговую сумму заказов *из текущего списка* `orders` и сравнить полученные значения.

В таком случае необходимо реализовать right join, предполагая, что список `orders` находится в левой части «отношений», а список `details` — в правой. Напомним, что объединение типа right join возвращает все строки из правого датасета и только совпадающие строки из левого. Обновим предыдущее списковое включение:

```
orders_details_right = [[o for o in orders if d[0] in o][0] + d[1:] if d[0] in
[o[0] for o
                        in orders] ❶ else (d[0], None, None) + d[1:] for d in
details]
```

Здесь мы добавляем к условию `if` блок `else` ❶, который находится в теле цикла `for d in details`. Этот блок `else` будет срабатывать для каждой строки из `details`, не имеющей совпадений в `orders`. `else` создает новый кортеж, содержащий номер заказа и две записи `None` на месте отсутствующих в `orders` полей, и объединяет этот кортеж со строкой из `details`, получая строку со структурой, подобной структуре остальных строк. Таким образом, в дополнение ко всем строкам, которые есть в обоих списках, созданный набор данных будет содержать строку из `details`, для которой нет соответствия в `orders`:

```
[
 --фрагмент--
 (4626490, '2022-02-06', 9004, 'Sweater', 'Dickies', 56.0, 1),
 (4626592, None, None, 'Shorts', 'Protest', 48.0, 1)
]
```

Теперь, когда у нас есть список `orders_details_right` (объединение списков `orders` и `details` методом right join), можно суммировать все заказы и сравнить результаты с суммой только тех заказов, которые есть в списке `orders`. Вычисляем общую сумму всех заказов с помощью встроенной в Python функции `sum()`:

```
sum(pr*qt for _, _, _, _, _, pr, qt in orders_details_right)
```

Цикл `for`, передаваемый в качестве параметра в `sum()`, немного похож на цикл, используемый в списковом включении, поскольку позволяет брать только нужные элементы на каждой итерации цикла. В данном конкретном примере все, что нам нужно найти на каждой итерации, — это `pr*qt`, произведение значений цены (Price) и количества (Quantity) из имеющегося кортежа. Поскольку другие

значения кортежей нас, по сути, не интересуют, для их обозначения после ключевого слова `for` мы используем заполнители `_`.

Выполнив все шаги, представленные выше, получим:

```
779.0
```

Подсчитать итоговую сумму только для тех заказов, которые есть в списке `orders`, можно с помощью модифицированной версии `sum()`:

```
sum(pr*qt for _, dt, _, _, _, pr, qt in orders_details_right ❶ if dt != None)
```

Здесь мы добавляем в цикл блок `if`, чтобы отфильтровать заказы, которых не было в списке `orders` ❶. Игнорируем строки, у которых поле с датой (`dt`) содержит `None`, указывающее, что информация о заказе не была получена из `orders`. Результат составит:

```
731.0
```

Конкатенация массивов NumPy

Для конкатенации массивов NumPy не получится использовать оператор `+` как со списками. Это связано с тем, что, как говорилось в главе 3, в NumPy оператор `+` зарезервирован для выполнения операций поэлементного сложения над несколькими массивами. Вместо этого для конкатенации двух массивов NumPy используется функция `numpy.concatenate()`.

Для демонстрации будем использовать массив `base_salary` из раздела «Создание массива NumPy» главы 3. Данный массив был создан следующим образом (здесь назовем его `base_salary1`):

```
import numpy as np
jeff_salary = [2700,3000,3000]
nick_salary = [2600,2800,2800]
tom_salary = [2300,2500,2500]
base_salary1 = np.array([jeff_salary, nick_salary, tom_salary])
```

Напомню, что каждая строка массива содержит данные о базовом окладе конкретного сотрудника за три месяца. Теперь предположим, что у нас есть информация об окладе еще двух сотрудников в другом массиве — `base_salary2`:

```
maya_salary = [2200,2400,2400]
john_salary = [2500,2700,2700]
base_salary2 = np.array([maya_salary, john_salary])
```

Требуется хранить информацию об окладе всех пяти сотрудников в одном массиве. Для этого конкатенируем `base_salary1` и `base_salary2`, используя `numpy.concatenate()`:

```
base_salary = np.concatenate((base_salary1, base_salary2), axis=0)
```

Первый параметр функции — кортеж с массивами, подлежащими конкатенации. Второй параметр — `axis` — является критическим: он определяет, должны ли массивы быть конкатенированы горизонтально или вертикально, другими словами, будет ли второй массив добавлен в виде новых строк или новых столбцов. Первая ось (0) проходит по вертикали. Таким образом, `axis=0` сообщает функции `concatenate()`, что нужно расположить строки из `base_salary2` под строками `base_salary1`. В результате получаем следующий массив:

```
[[2700 3000 3000]
[2600 2800 2800]
[2300 2500 2500],
[2200 2400 2400],
[2500 2700 2700]]
```

Теперь представьте, что поступила информация об окладе за следующий месяц. Можем поместить эти значения в новый массив NumPy, как показано ниже:

```
new_month_salary = np.array([[3000],[2900],[2500],[2500],[2700]])
```

На экране этот массив будет выглядеть так:

```
[[3000]
[2900]
[2500]
[2500]
[2700]]
```

Теперь нам нужно добавить массив `new_month_salary` к массиву `base_salary` в виде дополнительного столбца. Предполагая, что порядок перечисления сотрудников одинаков в обоих массивах, будем использовать `concatenate()`:

```
base_salary = np.concatenate((base_salary, new_month_salary), axis=1)
```

Поскольку ось 1 проходит горизонтально по столбцам, `axis=1` сообщает функции `concatenate()`, что нужно расположить массив `new_month_salary` в виде столбца справа от других столбцов массива `base_salary`. Теперь `base_salary` выглядит так:

```
[[2700 3000 3000 3000]
[2600 2800 2800 2900]
[2300 2500 2500 2500]
[2200 2400 2400 2500]
[2500 2700 2700 2700]]
```

УПРАЖНЕНИЕ № 11:
ДОБАВЛЕНИЕ НОВЫХ СТРОК/СТОЛБЦОВ В МАССИВ NUMPY

Продолжая предыдущий пример, создайте новый массив NumPy с двумя столбцами, который будет содержать информацию об окладе каждого сотрудника еще за два месяца. Затем объедините существующий массив `base_salary` с вновь созданным. Так же как мы делали выше, добавляйте оклад сотрудников к массиву `base_salary` в виде новой строки. Обратите внимание, что при добавлении одной строки или столбца в массив NumPy можно использовать функцию `numpy.append()`, а не `numpy.concatenate()`.

Объединение структур данных pandas

В главе 3 мы рассмотрели некоторые основные приемы объединения структур данных pandas. Мы разобрали примеры того, как объединять объекты Series в DataFrame и как объединить две структуры DataFrame по индексам. Вы также узнали о различных типах объединений двух датафреймов в один при использовании параметра `how` внутри метода pandas `join()` или `merge()`. В этом разделе вы увидите больше примеров применения этого параметра для создания нестандартных объединений датафреймов, например right join. Однако перед этим вам нужно научиться конкатенировать два датафрейма по определенной оси.

Конкатенация датафреймов

Как и в случае с массивами NumPy, вам может понадобиться объединить два объекта DataFrame по определенной оси, добавив строки или столбцы одного датафрейма к другому. На практических примерах в этом разделе покажем, как это сделать с помощью функции pandas `concat()`. Прежде чем перейти к рассмотрению примеров, создадим два датафрейма, которые будем конкатенировать.

Можно создать датафрейм, поместив в словарь списки `jeff_salary`, `nick_salary` и `tom_salary`, приведенные ранее в этой главе, следующим образом:

```
import pandas as pd
salary_df1 = pd.DataFrame(
    {'jeff': jeff_salary,
     'nick': nick_salary,
     'tom': tom_salary
    })
```

Каждый список становится значением в словаре, который, в свою очередь, становится столбцом в создаваемом датафрейме. Ключи словаря, которые являются именами соответствующих сотрудников, становятся метками столбцов. Каждая строка датафрейма содержит все данные об окладе за один месяц. По умолчанию используется числовое индексирование строк, но оптимальной в данном случае была бы индексация с помощью месяцев. Обновить индексы можно таким образом:

```
salary_df1.index = ['June', 'July', 'August']
```

Тогда датафрейм `salary_df1` будет иметь следующий вид:

```
         jeff   nick    tom
June     2700   2600   2300
July     3000   2800   2500
August   3000   2800   2500
```

Возможно, удобнее просматривать данные о зарплате сотрудника в виде строки, а не столбца. Внести такое изменение можно с помощью свойства датафрейма `T`, которое является сокращением от названия метода `DataFrame.transpose()`:

```
salary_df1 = salary_df1.T
```

Эта операция *транспонирует* датафрейм, превращая его столбцы в строки или наоборот. Теперь датафрейм индексируется по имени сотрудника и выглядит так:

```
       June   July   August
jeff   2700   3000     3000
nick   2600   2800     2800
tom    2300   2500     2500
```

Теперь необходимо создать еще один датафрейм с теми же столбцами для конкатенации с `salary_df1`. Как и при конкатенации массивов NumPy, создаем датафрейм, который содержит данные об окладе еще двух сотрудников:

```
salary_df2 = pd.DataFrame(
    {'maya': maya_salary,
     'john': john_salary
    },
    index = ['June', 'July', 'August']
).T
```

Создаем датафрейм, устанавливаем индекс и транспонируем строки и столбцы в одной операции. Вот как будет выглядеть вновь созданный датафрейм:

```
       June   July   August
maya   2200   2400     2400
john   2500   2700     2700
```

Теперь, когда мы создали оба датафрейма, их можно конкатенировать.

Конкатенация по оси 0

Функция pandas `concat()` конкатенирует объекты pandas по определенной оси. По умолчанию эта функция использует ось 0, то есть строки датафрейма, который стоит вторым в списке аргументов, будут добавлены после строк датафрейма, который является первым. Таким образом, для конкатенации `salary_df1` и `salary_df2` можно вызвать `concat()`, не передавая аргумент `axis` явно. Все, что нужно сделать, — указать имена датафреймов в квадратных скобках:

```
salary_df = pd.concat([salary_df1, salary_df2])
```

Получим следующий датафрейм:

```
       June   July   August
jeff   2700   3000     3000
nick   2600   2800     2800
tom    2300   2500     2500
maya   2200   2400     2400
john   2500   2700     2700
```

Как видите, строки `maya` и `john` из второго датафрейма добавлены после строк первого.

Конкатенация по оси 1

При конкатенации по оси 1 функция `concat()` добавит столбцы второго датафрейма справа от столбцов первого. Чтобы посмотреть, как это работает, в качестве первого датафрейма используем `salary_df` из предыдущего раздела. А в качестве второго создадим структуру, которая будет содержать данные об окладе еще за два месяца:

```
salary_df3 = pd.DataFrame(
    {'September': [3000,2800,2500,2400,2700],
     'October': [3200,3000,2700,2500,2900]
    },
    index = ['jeff', 'nick', 'tom', 'maya', 'john']
)
```

Теперь вызовем `concat()`, передавая два датафрейма и указывая `axis=1` для горизонтальной конкатенации:

```
salary_df = pd.concat([salary_df, salary_df3], axis=1)
```

Полученный датафрейм:

```
       June   July   August   September   October
jeff   2700   3000     3000        3000      3200
nick   2600   2800     2800        2800      3000
tom    2300   2500     2500        2500      2700
maya   2200   2400     2400        2400      2500
john   2500   2700     2700        2700      2900
```

Данные об окладе из второго датафрейма представлены в новых столбцах справа от данных об окладе из первого датафрейма.

Удаление столбцов/строк из датафрейма

После объединения датафреймов может понадобиться удалить ненужные строки или столбцы. Скажем, нужно удалить столбцы `September` и `October` из датафрейма `salary_df`. Сделать это можно с помощью метода `DataFrame.drop()`:

```
salary_df = salary_df.drop(['September', 'October'], axis=1)
```

Первый аргумент принимает названия столбцов или строк, которые необходимо удалить из датафрейма. Затем следует аргумент `axis`, который уточняет,

являются эти имена строками или столбцами. В этом примере мы удаляем столбцы, поскольку `axis` имеет значение `1`.

Метод `drop()` не ограничивается удалением только последних столбцов/строк датафрейма. Можно передать произвольный список столбцов или строк, которые нужно удалить, как показано ниже:

```
salary_df = salary_df.drop(['nick', 'maya'], axis=0)
```

После выполнения двух предыдущих операций `salary_df` будет выглядеть следующим образом:

```
       June   July   August
jeff   2700   3000     3000
tom    2300   2500     2500
john   2500   2700     2700
```

Мы удалили столбцы `September` и `October`, а также строки с индексами `nick` и `maya`.

Конкатенация датафреймов с иерархическим индексом

До сих пор мы разбирали только примеры конкатенации датафреймов с простыми индексами. Теперь рассмотрим, как конкатенировать датафреймы с MultiIndex. В следующем примере используется датафрейм `df_date_region`, представленный в разделе «Группировка и агрегирование данных» главы 6. Мы создали датафрейм с помощью нескольких последовательных операций. Вот как он выглядел:

```
                   Total
Date        Region
2022-02-04  East    97.0
            West   243.0
2022-02-05  East   160.0
            West    35.0
2022-02-06  East   110.0
            West    86.0
```

Чтобы повторно создать этот датафрейм, необязательно повторять шаги из главы 6. Вместо этого выполните следующую операцию:

```
df_date_region1 = pd.DataFrame(
 [
  ('2022-02-04', 'East', 97.0),
```

```
  ('2022-02-04', 'West', 243.0),
  ('2022-02-05', 'East', 160.0),
  ('2022-02-05', 'West', 35.0),
  ('2022-02-06', 'East', 110.0),
  ('2022-02-06', 'West', 86.0)
 ],
columns =['Date', 'Region', 'Total']).set_index(['Date','Region'])
```

Теперь нам понадобится другой датафрейм, который будет также индексироваться по `Date` и `Region`. Создайте его следующим образом:

```
df_date_region2 = pd.DataFrame(
 [
  ('2022-02-04', 'South', 114.0),
  ('2022-02-05', 'South', 325.0),
  ('2022-02-06', 'South', 212.0)
 ],
columns =['Date', 'Region', 'Total']).set_index(['Date','Region'])
```

Второй датафрейм содержит те же три даты, что и первый, но данные в нем относятся к новому региону — `South`. Проблема объединения этих двух датафреймов в том, что в результате нужно получить набор данных, отсортированный по дате, а не просто добавить второй датафрейм после первого. Вот как это сделать:

```
df_date_region = pd.concat([df_date_region1,
            df_date_region2]).sort_index(level=['Date','Region'])
```

Сначала вызываем `concat()`. Вызов этой функции выглядит так же, как при конкатенации датафреймов с одним индексным столбцом. Определяем датафреймы для конкатенации, и поскольку параметр `axis` опущен, по умолчанию они будут объединены по вертикали. Чтобы отсортировать строки в полученном датафрейме по дате и региону, необходимо вызвать метод `sort_index()`. В результате получим следующий датафрейм:

```
                   Total
Date        Region
2022-02-04  East    97.0
            South  114.0
            West   243.0
2022-02-05  East   160.0
            South  325.0
            West    35.0
2022-02-06  East   110.0
            South  212.0
            West    86.0
```

Как видите, строки из второго датафрейма распределены между строками из первого, при этом верхнеуровневая группировка по дате сохранилась.

Join-объединение двух датафреймов

Объединение двух датафреймов — это не просто добавление строк или столбцов одного датафрейма после или рядом со столбцами другого, а совмещение каждой строки из одного датасета с соответствующей строкой (строками) второго датасета. Чтобы рассмотреть основы join-объединения датафреймов, вернитесь к подразделу «Объединение датафреймов» в главе 3, где рассматриваются различные типы объединений. Благодаря этому разделу вы расширите знания, полученные в главе 3, реализовав механизм right join и объединение на основе отношений «многие-ко-многим».

Реализация Right Join

Метод right join берет все строки из второго датафрейма и объединяет их с подходящими строками из первого датафрейма. Вы увидите, что при данном типе объединения в некоторых строках итогового датафрейма могут появиться неопределенные поля, что чревато неожиданными последствиями.

Чтобы посмотреть, как это работает, выполните right join датафреймов `df_orders` и `df_details`, созданных из списков `orders` и `details` главы 6. Мы использовали их в примерах вступительного раздела этой главы. Сформировать датафреймы из этих списков можно так:

```
import pandas as pd
df_orders = pd.DataFrame(orders, columns =['OrderNo', 'Date', 'Empno'])
df_details = pd.DataFrame(details, columns =['OrderNo', 'Item', 'Brand',
                                             'Price', 'Quantity'])
```

Напомню, что каждой строке исходного списка `details` соответствует определенная строка в списке `orders`. То же справедливо и для датафреймов `df_details` и `df_orders`. Чтобы понятнее проиллюстрировать right join, необходимо добавить одну или несколько новых строк в `df_details`, для которых не будет соответствующих строк из `df_orders`. Добавить строку можно с помощью метода `DataFrame.append()`, который принимает добавляемую строку в виде словаря или серии.

Если вы следовали примеру из раздела «Реализация join-объединений списков» этой главы, то у вас уже добавлена новая строка в список `details`, и поэтому она должна появиться и в датафрейме `df_details`. В этом случае можно пропустить

следующую операцию. Иначе добавьте приведенную ниже строку в `df_details` в виде словаря. Обратите внимание, что значения поля `OrderNo` новой строки нет среди значений столбца `OrderNo` датафрейма `df_orders`:

```
df_details = df_details.append(
   {'OrderNo': 4626592,
    'Item': 'Shorts',
    'Brand': 'Protest',
    'Price': 48.0,
    'Quantity': 1
   },
❶  ignore_index = True
)
```

Необходимо добавить параметр `ignore_index` со значением `True` ❶, в противном случае добавить словарь к датафрейму не получится. Значение параметра `True` также сбрасывает индексы датафрейма и устанавливает последовательную нумерацию строк (`0, 1, ...`).

Затем мы объединяем датафреймы `df_orders` и `df_details` с помощью метода `merge()`. Как обсуждалось в главе 3, `merge()` обеспечивает удобный способ объединения двух датафреймов с общим столбцом:

```
df_orders_details_right = df_orders.merge(df_details, ❶ how='right',
                  ❷ left_on='OrderNo', right_on='OrderNo')
```

Используем параметр `how` для указания типа объединения — в данном примере это right join ❶. С помощью параметров `left_on` и `right_on` указываем объединяемые столбцы из `df_orders` и `df_details` соответственно ❷.

Полученный датафрейм выглядит так:

```
    OrderNo        Date   Empno          Item           Brand   Price   Quantity
0   9423517  2022-02-04  9001.0         Jeans        Rip Curl    87.0          1
1   9423517  2022-02-04  9001.0        Jacket  The North Face   112.0          1
2   4626232  2022-02-04  9003.0         Socks            Vans    15.0          1
3   4626232  2022-02-04  9003.0         Jeans      Quiksilver    82.0          1
4   9423534  2022-02-04  9001.0         Socks              DC    10.0          2
5   9423534  2022-02-04  9001.0         Socks      Quiksilver    12.0          2
6   9423679  2022-02-05  9002.0       T-shirt       Patagonia    35.0          1
7   4626377  2022-02-05  9003.0         Hoody          Animal    44.0          1
8   4626377  2022-02-05  9003.0  Cargo Shorts          Animal    38.0          1
9   4626412  2022-02-05  9004.0         Shirt          Volcom    78.0          1
10  9423783  2022-02-06  9002.0  Boxer Shorts        Superdry    30.0          2
11  9423783  2022-02-06  9002.0        Shorts           Globe    26.0          1
```

```
12  4626490  2022-02-06  9004.0  Cargo Shorts       Billabong    54.0          1
13  4626490  2022-02-06  9004.0       Sweater         Dickies    56.0          1
14  4626592         NaN     NaN        Shorts         Protest    48.0          1
```

Поскольку для вновь добавленной в `df_details` строки нет соответствия в `df_orders`, поля `Date` и `Empno` соответствующей строки итогового датафрейма будут содержать `NaN` (маркер по умолчанию для отсутствующего значения). Однако это вызовет проблему: `NaN` не может храниться в колонках с целыми числами (integer), и поэтому при добавлении этого маркера pandas автоматически преобразует целочисленный столбец в колонку чисел с плавающей точкой (float). Именно по этой причине значения столбца `Empno` конвертировались во float. В этом можно убедиться, обратившись к свойству `dtypes` датафрейма `df_orders_details_right`, которое показывает тип хранимых в каждой колонке значений:

```
print(df_orders_details_right.dtypes)
```

Вы получите следующий вывод:

```
OrderNo         int64
Date           object
Empno         float64
Item           object
Brand          object
Price         float64
Quantity        int64
dtype: object
```

Как видите, столбец `Empno` имеет тип `float64`. Если аналогичным образом проверить свойство `dtypes df_orders`, вы увидите, что столбец `Empno` изначально имел тип `int64`.

Очевидно, что такое преобразование целых чисел в числа с плавающей точкой нежелательно: в идентификаторах сотрудников не должно быть десятичных знаков! Есть ли способ вновь преобразовать столбец `Empno` в целочисленный? Один из обходных путей — заменить `NaN` в этих столбцах на некоторое целое значение, скажем `0`. Если `0` не является идентификатором какого-либо сотрудника, такая замена вполне приемлема. Вот ее реализация:

```
df_orders_details_right =
df_orders_details_right.fillna({'Empno':0}).astype({'Empno':'int64'})
```

Мы используем метод `fillna()`, который заменяет `NaN` в выбранном столбце(-ах) на указанное значение. Столбец и значение для замены объявляются в виде словаря. В этом конкретном примере мы заменяем `NaN` в столбце `Empno` на `0`. Затем с помощью метода `astype()` преобразуем тип колонки в `int64`. И снова столбец и новый тип указываются в виде пары «ключ — значение» словаря.

В итоге получаем следующий датафрейм:

```
    OrderNo         Date    Empno     Item            Brand   Price   Quantity
0   9423517   2022-02-04     9001    Jeans         Rip Curl    87.0          1
1   9423517   2022-02-04     9001   Jacket   The North Face   112.0          1
--snip--
14  4626592          NaN        0   Shorts          Protest    48.0          1
```

Значения `NaN` в столбце `Empno` превратились в `0`, а идентификаторы сотрудников снова отображаются как целые числа. Больше никаких десятичных знаков!

Реализация объединения типа «многие-ко-многим»

Между датасетами может быть установлено отношение типа «многие-ко-многим», когда строка в каждом наборе данных связана с несколькими строками в другом. Предположим, что у нас есть два датасета, содержащих книги и авторов соответственно. Каждая запись в датасете с авторами может быть связана с одной или несколькими записями в датасете с книгами, а каждая запись в наборе данных книг может быть связана с одной или несколькими записями в наборе данных авторов.

Обычно для объединения датасетов, имеющих отношения «многие-ко-многим», используется *хеш-таблица*, также известная как *таблица соответствия*. Эта таблица сопоставляет два (или более) датасета, ссылаясь на первичные ключи каждого из них. Хеш-таблица имеет отношения типа «один-ко-многим» с каждым из датасетов и выступает как посредник между ними, позволяя их объединять.

Чтобы изучить, как реализовать объединение «многие-ко-многим» через хеш-таблицу, создайте датафреймы `books` и `authors` следующим образом:

```
import pandas as pd
books = pd.DataFrame({'book_id': ['b1', 'b2', 'b3'],
                      'title': ['Beautiful Coding', 'Python for Web
                                Development', 'Pythonic Thinking'],
                      'topic': ['programming', 'Python, Web', 'Python']})
authors = pd.DataFrame({'author_id': ['jsn', 'tri', 'wsn'],
                        'author': ['Johnson', 'Treloni', 'Willson']})
```

Датафрейм `books` содержит три книги, у каждой из которых есть уникальный идентификатор — `book_id`, а датафрейм `authors` включает трех авторов (тоже с уникальными идентификаторами — `author_id`). Теперь сформируем третий датафрейм, `matching`, который будет служить в качестве хеш-таблицы, связывающей каждую книгу с соответствующим автором(ами) и наоборот:

```
matching = pd.DataFrame({'author_id': ['jsn', 'jsn','tri', 'wsn'],
                         'book_id': ['b1', 'b2', 'b2', 'b3']})
```

В датафрейме `matching` два столбца: один соответствует идентификаторам авторов, а другой — идентификаторам книг. В отличие от двух других датафреймов, каждая строка в `matching` представляет не только одного автора или одну книгу. Она содержит информацию об *отношениях между* одним конкретным автором и одной конкретной книгой. Вот как выглядит этот датафрейм:

```
   author_id   book_id
0        jsn        b1
1        jsn        b2
2        tri        b2
3        wsn        b3
```

В поле `author_id` и первой и второй строки указано значение `jsn`, то есть Джонсон (Johnson) является автором двух разных книг. Аналогично во второй и третьей строках одинаковый `book_id` — `b2`, это означает что у книги Python for Web Development два автора.

Теперь можно создать объединение типа «многие-ко-многим» датасетов `authors` и `books` с помощью датасета `matching`, как показано ниже:

```
authorship = books.merge(matching).merge(authors)[['title','topic','author']]
```

На самом деле операция состоит из двух отдельных объединений с использованием метода `merge()`. Сначала мы объединяем `books` и `matching` по столбцам `book_id`. Затем объединяем результат первого объединения с датафреймом `authors` по столбцам `author_id`. Обе эти операции являются простыми объединениями «один-ко-многим», однако вместе они создают новый датафрейм, иллюстрирующий отношения типа «многие-ко-многим» между датасетами `books` и `authors`. Фильтруем датафрейм так, чтобы он включал только столбцы `title`, `topic` и `author`, и получим следующий результат:

```
                         title         topic    author
0            Beautiful Coding   programming   Johnson
1  Python for Web Development   Python, Web   Johnson
2  Python for Web Development   Python, Web   Treloni
3           Pythonic Thinking        Python   Willson
```

Как видите, Johnson дважды указан как автор: сначала для Beautiful Coding, а затем для Python for Web Development; книга Python for Web Development тоже повторяется (по одному разу для каждого автора).

Выводы

В этой главе мы рассмотрели различные способы объединения датасетов, представленных в виде встроенных (например, списков) или сторонних структур данных (например, массивов NumPy и pandas DataFrame). Вы научились конкатенировать датасеты, добавляя столбцы или строки одного набора данных к другому, а также объединять датасеты по совпадающим строкам.

8

Визуализация

Графические данные более наглядны, чем сухие цифры. Например, можно создать линейный график, отображающий изменение цены акции с течением времени, или отследить интерес к статьям на сайте с помощью гистограммы ежедневных просмотров каждой из них. Визуализация такого рода поможет сразу же распознать тенденции в данных.

В этой главе приводится обзор самых распространенных типов визуализации данных и рассказывается, как строить графики с помощью Matplotlib, популярной библиотеки Python. Мы также рассмотрим, как интегрировать Matplotlib с pandas и как создавать карты с помощью библиотек Matplotlib и Cartopy.

Распространенные способы визуализации

Существует несколько типов диаграмм для визуализации данных: линейные, столбчатые, круговые и гистограммы. В данном разделе мы обсудим все эти типы, а также исследуем типичные примеры использования каждого из них.

Линейные диаграммы

Линейные диаграммы, также известные как *линейные графики*, полезны, когда нужно проиллюстрировать тенденции в данных за определенный период

времени. На линейной диаграмме столбец меток времени датасета располагается вдоль оси x, а один или несколько числовых столбцов — по оси y.

В качестве примера рассмотрим веб-сайт, на котором пользователи могут просматривать различные статьи. Можно создать график статьи, где по оси x будет отображаться определенное количество дней, а по оси y — просмотры в каждый из этих дней. Такой график показан на рис. 8.1.

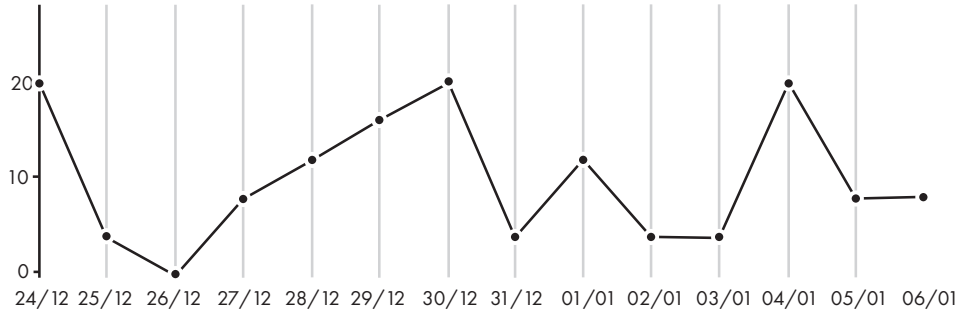

Рис. 8.1. Линейный график изменения количества просмотров статьи с течением времени

Можно наложить несколько параметров на одну линейную диаграмму, отображая каждый из них линией своего цвета, чтобы выявить корреляцию между ними. Например, на рис. 8.2 в дополнение к количеству просмотров статьи показано количество уникальных пользователей сайта в каждый из дней.

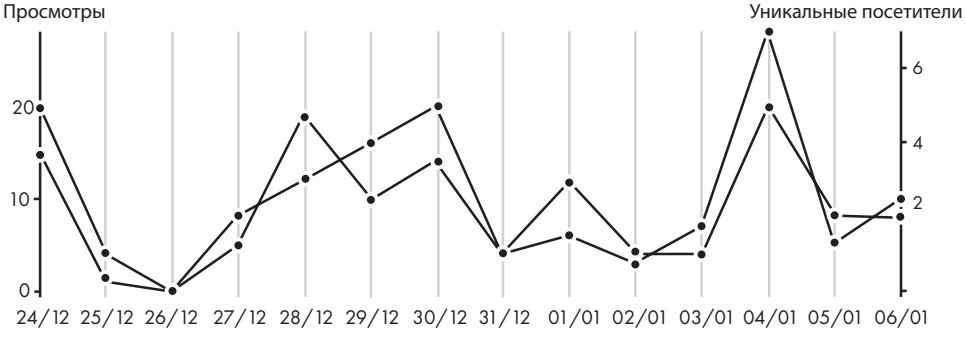

Рис. 8.2. Линейный график, показывающий взаимосвязь между параметрами

Левая ось y на этом графике показывает количество просмотров статьи, а правая ось y — количество уникальных посетителей. Наложение данных по обоим

параметрам наглядно демонстрирует общую корреляцию между количеством просмотров и уникальными посетителями.

ПРИМЕЧАНИЕ

Просмотры статей можно также отображать на гистограммах. Гистограммы мы обсудим позже в этом же разделе.

Столбчатые диаграммы

Столбчатые диаграммы, также называемые *столбчатыми графиками*, отображают категориальные данные с помощью прямоугольных столбиков с высотой, пропорциональной значениям представляемых данных, что позволяет сравнивать категории. Для примера рассмотрим значения, представляющие совокупный годовой объем продаж компании в разных регионах:

```
Новая Англия     $882 703
Северо-восток    $532 648
Средний Запад    $714 406
```

На рис. 8.3 показана столбчатая диаграмма с данными о продажах.

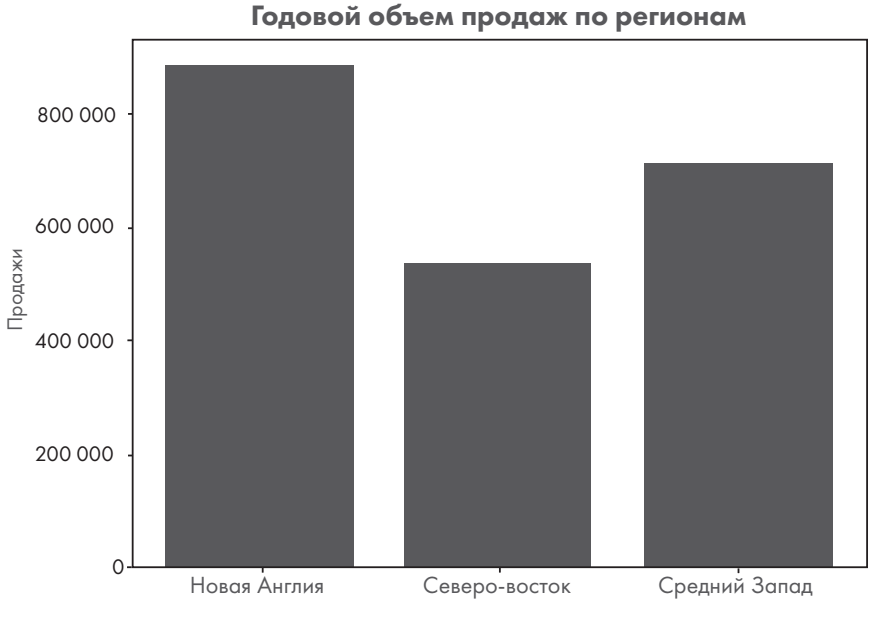

Рис. 8.3. Столбчатая диаграмма сравнения категориальных данных

На этом графике по оси *y* отображаются сравнительные показатели продаж для регионов, расположенных на оси *x*.

Круговые диаграммы

Круговые диаграммы иллюстрируют процентное соотношение категорий в датасете. На рис. 8.4 показаны объемы продаж из предыдущего примера в виде круговой диаграммы.

Рис. 8.4. Круговая диаграмма с процентным соотношением категорий в виде круговых секторов

Размер сектора здесь дает наглядное представление о доле вклада каждой категории в общую картину. Можно с легкостью сравнить продажи в каждом регионе. Такая диаграмма эффективна, когда каждый из секторов занимает значительную часть круга, но как можно догадаться, круговая диаграмма не лучший выбор, если нужно отобразить очень маленькие сектора. Например, сектор, составляющий 0.01% от целого, может быть даже не виден на диаграмме.

Гистограммы

Гистограммы отображают частотные распределения, то есть сколько раз определенное значение либо диапазон значений встречается в датасете. Каждое значение (или итоговый показатель) представлено вертикальным столбцом, высота которого соответствует частоте этого значения. Например,

гистограмма на рис. 8.5 показывает частотность диапазонов окладов сотрудников отдела продаж.

На гистограмме оклады сгруппированы в диапазоны по $50, а каждый столбик представляет собой количество сотрудников с определенным размером оклада. Визуализация наглядно демонстрирует, насколько отличается количество сотрудников, зарабатывающих от $1200 до $1250, от количества сотрудников с окладом в других диапазонах, например от $1250 до $1300.

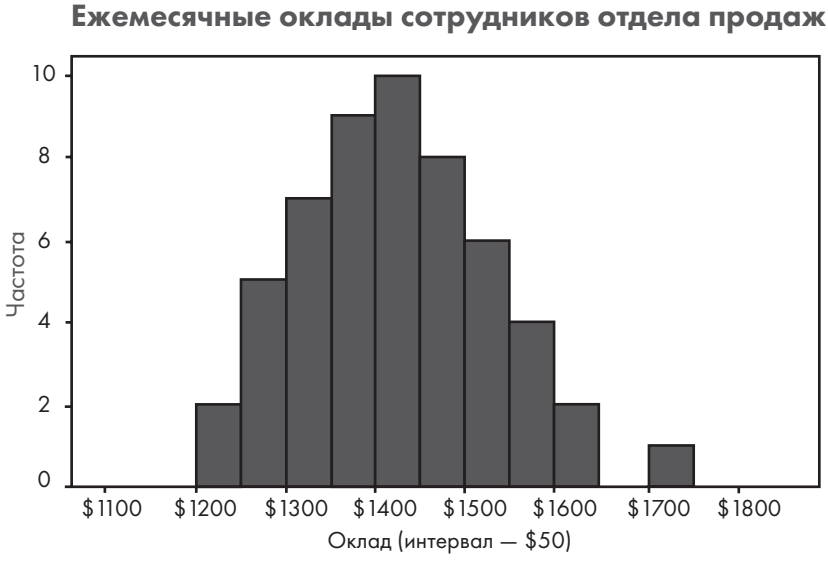

Рис. 8.5. Гистограмма распределения окладов

Построение графиков с помощью Matplotlib

Теперь, когда вы познакомились с наиболее распространенными типами диаграмм, рассмотрим, как их создавать с помощью Matplotlib, одной из самых популярных библиотек Python для визуализации данных. Вы научитесь строить линейные, круговые, столбчатые диаграммы и гистограммы.

Каждая визуализация Matplotlib, или *figure*, строится из иерархии вложенных объектов. Можно работать с этими объектами напрямую, чтобы создавать визуализации с широкими возможностями настроек, или управлять объектами опосредованно — через функции модуля `matplotlib.pyplot`. Последний подход проще, и часто его вполне достаточно для создания базовых графиков и диаграмм.

Установка Matplotlib

Проверьте, установлен ли уже Matplotlib, импортировав его в текущую сессию интерпретатора Python:

```
> import matplotlib
```

Если вы получите `ModuleNotFoundError`, установите Matplotlib с помощью `pip` следующим образом:

```
$ python -m pip install -U matplotlib
```

Использование matplotlib.pyplot

Модуль `matplotlib.pyplot`, называемый обычно в коде `plt`, представляет собой набор функций для построения привлекательных графиков. Модуль позволяет легко задавать различные характеристики фигуры, такие как ее название, метки осей и т. д. Например, код ниже строит линейный график, отображающий цену акций Tesla на момент закрытия торгов в течение пяти дней подряд:

```
from matplotlib import pyplot as plt

days = ['2021-01-04', '2021-01-05', '2021-01-06', '2021-01-07', '2021-01-08']
prices = [729.77, 735.11, 755.98, 816.04, 880.02]

plt.plot(days,prices)
plt.title('NASDAQ: TSLA')
plt.xlabel('Дата')
plt.ylabel('USD')
plt.show()
```

Сначала мы определяем набор данных в виде двух списков: `days`, содержащий даты, которые будут отображаться по оси *x*, и `prices` с ценами, которые будут располагаться по оси *y*. Затем мы создаем объект *plot*, часть графика, которая собственно содержит данные, с помощью функции `plt.plot()`, передавая ей данные для осей *x* и *y*. В следующих трех строках кода мы настраиваем график: добавляем заголовок с помощью `plt.title()` и метки для осей *x* и *y* с помощью `plt.xlabel()` и `plt.ylabel()`. Наконец, выводим рисунок на экран с помощью `plt.show()`. На рис. 8.6 показан результат.

По умолчанию `plt.plot()` строит визуализацию в виде набора линий, соединяющих точки данных, которые наносятся на оси *x* и *y*. Для оси *y* Matplotlib

автоматически выбрал диапазон от 720 до 880 с интервалами по $20, чтобы легко видеть цену акций в каждый из дней.

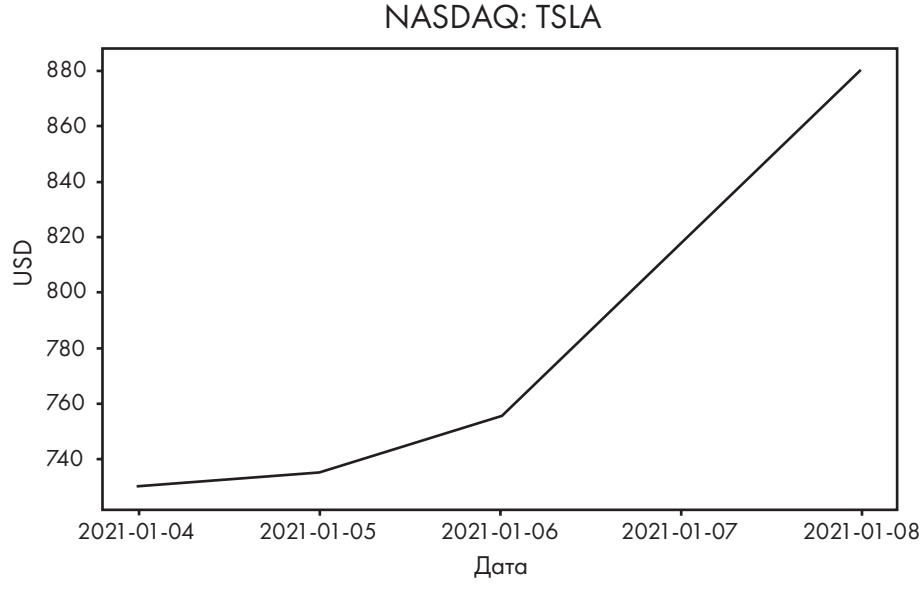

Рис. 8.6. Простой линейный график, созданный с помощью модуля `matplotlib.pyplot`

Построить базовую круговую диаграмму так же просто, как и линейный график. Например, следующий код генерирует круговую диаграмму с рис. 8.4:

```
import matplotlib.pyplot as plt

regions = ['Новая Англия', 'Северо-восток', 'Средний Запад']
sales = [882703, 532648, 714406]

plt.pie(sales, labels=regions, autopct='%1.1f%%')
plt.title('Продажи по регионам')
plt.show()
```

Скрипт следует той же базовой схеме, которую мы использовали для создания линейного графика: определяем данные для построения графика, создаем его, настраиваем характеристики и выводим на экран. На этот раз данные включают список регионов, которые будут служить метками для каждого среза круговой диаграммы, а также итоговые продажи для каждого региона. Значения списка

определяют размер каждого среза. Чтобы составить круговой график, а не линейный, вызываем функцию `plt.pie()`, передав `sales` в качестве данных для построения графика и `regions` в качестве меток для данных. Мы также используем параметр `autopct`, чтобы значения в частях круга выражались в процентах. Используем форматирование строк Python для отображения значений с точностью до десятых долей процента.

Теперь визуализируем те же входные данные в виде столбчатой диаграммы, как показано на рис. 8.3:

```
import matplotlib.pyplot as plt
regions = ['Новая Англия', 'Северо-восток', 'Средний Запад']
sales = [882703, 532648, 714406]

plt.bar(regions, sales)
plt.xlabel("Регионы")
plt.ylabel("Продажи")
plt.title("Годовой объем продаж по регионам")

plt.show()
```

Мы передаем список `regions` в функцию `plt.bar()` в качестве меток оси *x* для столбцов. Второй аргумент, который мы передаем в `plt.bar()`, — это список со значениями продаж, соответствующих товарам в списке `regions`. И здесь, и в примере с круговой диаграммой можно использовать отдельные списки для меток и объема продаж, поскольку порядок элементов в списке Python является постоянным.

Работа с объектами Figure и Axes

По своей сути визуализация Matplotlib строится из двух основных типов объектов: объекта `Figure` и одного или нескольких объектов `Axes`. В предыдущих примерах `matplotlib.pyplot` служил интерфейсом для непрямого взаимодействия с этими объектами, позволяя настраивать некоторые элементы визуализации. Однако можно получить больше контроля над процессом визуализации, работая непосредственно с самими объектами `Figure` и `Axes`.

Объект `Figure` является внешним верхнеуровневым контейнером визуализации Matplotlib. Он может включать один или несколько графиков. Объект `Figure` полезен, когда необходимо совершить операции с отображаемым рисунком в целом, например изменить размер или сохранить в файл. Между тем каждый объект `Axes` представляет один график на рисунке. Объект `Axes` используется для настройки графика и определения его расположения на изображении. Например, можно задать его систему координат и отметить позиции на оси.

Доступ к объектам `Figure` и `Axes` осуществляется через функцию `matplotlib.pyplot.subplots()`. При вызове без аргументов эта функция возвращает экземпляр `Figure` и один экземпляр `Axes`, связанный с `Figure`. Добавляя аргументы к функции `subplots()`, можно создать экземпляр `Figure` и несколько `Axes`, то есть, другими словами, создать рисунок с несколькими графиками. Например, вызов `subplots(2,2)` создает рисунок с четырьмя графиками, расположенными в два ряда по два. Каждый график представлен одним объектом `Axes`.

ПРИМЕЧАНИЕ

Более подробную информацию об использовании `subplots()` см. в документации Matplotlib[1].

Создание гистограммы с помощью subplots()

В следующем скрипте мы используем `subplots()` для создания объекта `Figure` и одного объекта `Axes`. Затем обработаем объекты для создания гистограммы с рис. 8.5, отображающей распределение окладов сотрудников. Кроме объектов `Figure` и `Axes`, мы будем работать с модулем Matplotlib `matplotlib.ticker` для форматирования отметок на оси *x*, а также с NumPy для определения размера шага гистограммы ($50).

```
# импортирование модулей
import numpy as np
from matplotlib import pyplot as plt import matplotlib.ticker as ticker

# данные для графика
❶ salaries = [1215, 1221, 1263, 1267, 1271, 1274, 1275, 1318, 1320, 1324, 1324,
              1326, 1337, 1346, 1354, 1355, 1364, 1367, 1372, 1375, 1376, 1378,
              1378, 1410, 1415, 1415, 1418, 1420, 1422, 1426, 1430, 1434, 1437,
              1451, 1454, 1467, 1470, 1473, 1477, 1479, 1480, 1514, 1516, 1522,
              1529, 1544, 1547, 1554, 1562, 1584, 1595, 1616, 1626, 1717]
  # подготовка гистограммы
❷ fig, ax = plt.subplots()
❸ fig.set_size_inches(5.6, 4.2)
❹ ax.hist(salaries, bins=np.arange(1100, 1900, 50), edgecolor='black',
          linewidth=1.2)
❺ formatter = ticker.FormatStrFormatter('$%1.0f')
❻ ax.xaxis.set_major_formatter(formatter)
❼ plt.title('Ежемесячные оклады сотрудников отдела продаж')
  plt.xlabel('Оклад (интервал — $50)')
```

[1] https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.subplots.html

```
plt.ylabel('Частота')
# вывод гистограммы на экран
plt.show()
```

Начинаем с объявления списка `salaries` с данными об окладе, которые хотим визуализировать ❶. Затем вызываем функцию `subplots()` без параметров ❷, тем самым задаем создание рисунка с одним графиком. Функция возвращает кортеж, содержащий два объекта: `fig` и `ax`, представляющие собой рисунок и график соответственно.

Теперь, когда у нас есть экземпляры `Figure` и `Axes`, можем приступить к их настройке. В первую очередь вызываем метод `set_size_inches()` объекта `Figure` для изменения размера всего рисунка ❸. Затем — метод `hist()` объекта `Axes` для построения гистограммы ❹. Передаем методу список `salaries` в качестве входных данных для гистограммы, а также массив NumPy с точками оси *x* для определения интервалов. Функция NumPy `arange()` создает массив равномерно расположенных значений в заданном интервале (в данном случае с шагом `50` от `1100` до `1900`). Параметр `edgecolor` метода `hist()` нужен для отображения интервальных границ (bins) в виде линий черного цвета, а параметр `linewidth` позволяет определить ширину этих границ.

Далее используем функцию `FormatStrFormatter()` из модуля `matplotlib.ticker` для создания средства форматирования, которое будет добавлять знак доллара к каждой метке оси *x* ❺. Применяем средство форматирования к меткам оси *x* с помощью метода `set_major_formatter()` объекта `ax.xaxis` ❻. Наконец, задаем общие настройки графика, такие как его заголовок и метки главной оси, через интерфейс `matplotlib.pyplot` ❼ и выводим график на экран.

Частотное распределение на круговой диаграмме

Хотя гистограммы хорошо подходят для визуализации частотного распределения, для отображения такого рода данных в процентах можно использовать круговую диаграмму. В качестве примера в этом разделе приводится преобразование гистограммы распределения окладов, которую мы только что создали, в круговую диаграмму, отображающую данные в виде частей целого.

Прежде чем создать такую круговую диаграмму, из гистограммы необходимо извлечь основную информацию и упорядочить ее. В частности, нужно узнать количество значений окладов в каждом диапазоне $50. Для этого можно использовать функцию NumPy `histogram()`, которая рассчитывает гистограмму без ее отображения:

```
import numpy as np
count, labels = np.histogram(salaries, bins=np.arange(1100, 1900, 50))
```

Здесь мы вызываем функцию `histogram()`, передавая ей тот же список `salaries`, который создали ранее, и снова используем функцию NumPy `arange()` для создания равномерно распределенных областей. Вызов `histogram()` возвращает два массива NumPy: `count` и `labels`.

Массив `count` представляет собой количество сотрудников с окладом в каждом интервале и выглядит следующим образом:

```
[0,  0,  2,  5,  7,  9,  10,  8,  6,  4,  2,  0,  1,  0,  0]
```

А массив `labels` содержит границы интервалов областей:

```
[1100, 1150, 1200, 1250, 1300, 1350, 1400, 1450, 1500, 1550, 1600, 1650,
1700, 1750, 1800, 1850]
```

Далее необходимо объединить соседние элементы массива `labels`, превратив их в метки для частей круговой диаграммы. Так, соседние элементы `1100` и `1150` должны стать единой меткой, приведенной к виду `'$1100-1150'`. Используем следующее списковое включение:

```
labels = ['$'+str(labels[i])+'-'+str(labels[i+1]) for i, _ in
enumerate(labels[1:])]
```

Полученный в результате список `labels` будет выглядеть так:

```
['$1100-1150', '$1150-1200', '$1200-1250', '$1250-1300', '$1300-1350',
 '$1350-1400', '$1400-1450', '$1450-1500', '$1500-1550', '$1550-1600',
 '$1600-1650', '$1650-1700', '$1700-1750', '$1750-1800', '$1800-1850']
```

Каждый элемент в `labels` соответствует элементу в массиве `count` с тем же индексом. Однако, взглянув на массив `count`, можно заметить проблему: в некоторых интервалах количество окладов равно `0`. Вряд ли стоит включать эти пустые интервалы в круговую диаграмму. Чтобы исключить их, необходимо сформировать список индексов, соответствующих непустым интервалам в массиве `count`:

```
non_zero_pos = [i for i, x in enumerate(count) if x != 0]
```

Теперь можно использовать `non_zero_pos` для фильтрации `count` и `labels`, исключая те элементы, которые представляют собой пустые интервалы:

```
labels = [e for i, e in enumerate(labels) if i in non_zero_pos]
count = [e for i, e in enumerate(count) if i in non_zero_pos]
```

Остается только создать и отобразить круговую диаграмму с помощью интерфейса `matplotlib.pyplot` и `plt.pie()`:

```
from matplotlib import pyplot as plt
plt.pie(count, labels=labels, autopct='%1.1f%%')
plt.title('Ежемесячные оклады сотрудников отдела продаж')
plt.show()
```

На рис. 8.7 показан результат выполнения кода.

Рис. 8.7. Круговая диаграмма с распределением частотности

Круговая диаграмма визуализирует те же данные, что и гистограмма на рис. 8.5, однако каждый интервал на ней выражается процентом от целого, а не числом сотрудников с окладом, попадающим в эту область.

УПРАЖНЕНИЕ № 12:
ОБЪЕДИНЕНИЕ ИНТЕРВАЛОВ В СЕГМЕНТ OTHER (ДРУГОЕ)

Глядя на график на рис. 8.7, можно заметить, что некоторые интервалы представлены очень узким сектором круга. Это интервалы с одним или двумя сотрудниками. Измените график так, чтобы эти интервалы были объединены в один срез (сегмент), обозначенный `Other`. Для этого необходимо изменить массив `count` и список `labels`, а затем заново построить график.

Совместимость Matplotlib с другими библиотеками

Matplotlib легко взаимодействует с остальными библиотеками Python для визуализации данных из других источников или для построения графиков иного типа. Например, можно использовать Matplotlib в сочетании с pandas, чтобы построить график для данных из датафрейма, или создать карту, объединив Matplotlib с Cartopy, библиотекой, которая специализируется на работе с данными геолокации.

Построение графиков для данных pandas

Библиотека pandas тесно интегрирована с Matplotlib. Более того, у каждого объекта pandas Series или DataFrame есть метод `plot()`, который фактически является оберткой для метода `matplotlib.pyplot.plot()`. Он позволяет напрямую преобразовывать структуру данных pandas в график Matplotlib. Чтобы посмотреть, как это работает, создадим гистограмму из объекта DataFrame с данными о населении городов США. Будем использовать исходные данные из файла *us-cities-top-1k.csv*[1]. Гистограмма показывает количество мегаполисов (городов с населением 1 миллион человек и более) в каждом штате США. Вот как это сделать:

```
   import pandas as pd
   import matplotlib.pyplot as plt
   # подготовка датафрейма
❶ us_cities = pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/
                                  master/us-cities-top-1k.csv")
```

[1] https://github.com/plotly/datasets

```
❷ top_us_cities = us_cities[us_cities.Population.ge(1000000)]
❸ top_cities_count = top_us_cities.groupby(['State'], as_index = False)
                     .count().rename(columns={'City': 'cities_count'})
                        [['State','cities_count']]
   # построение графика
❹ top_cities_count.plot.bar('State', 'cities_count', rot=0)
❺ plt.xlabel("Штаты")
   plt.ylabel("Количество мегаполисов")
   plt.title("Количество мегаполисов в штатах США")
❻ plt.yticks(range(min(top_cities_count['cities_count']),
                   max(top_cities_count['cities_count'])+1 ))
   plt.show()
```

Сначала загружаем набор данных и преобразуем его в DataFrame с помощью метода pandas `read_csv()` ❶. Датасет содержит данные о населении, широте и долготе 1000 крупнейших городов США. Чтобы отфильтровать датасет так, что останутся только мегаполисы, используется метод DataFrame `ge()`, сокращение от *greater than or equal to* (больше или равно), запрашивающий только строки, значение поля `Population` которых больше или равно `1000000` ❷. Затем группируем данные по столбцу `State` и применяем агрегирующую функцию `count()` для нахождения общего количества мегаполисов в каждом штате ❸. Внутри `groupby` для параметра `as_index` устанавливаем значение `False`, чтобы избежать преобразования столбца `State` в индекс итогового датафрейма. Это необходимо потому, что позже в скрипте придется обратиться к этой колонке. Переименовываем столбец `City` в `cities_count`, чтобы показать, что он теперь содержит агрегированную информацию, и в итоговый датафрейм `top_cities_count` включаем только столбцы `State` и `cities_count`.

Затем с помощью метода DataFrame `plot.bar()` строим гистограмму ❹. Помните, что `plot()` фактически является оберткой для метода Matplotlib `pyplot.plot()`. В вызове этой функции мы указываем имена столбцов датафрейма, которые будут использоваться на графике в качестве осей x и y, и поворачиваем метки оси x до 0 градусов. После создания рисунок можно настроить с помощью интерфейса `matplotlib.pyplot`, как мы уже делали в предыдущих примерах. Задаем метки осей и заголовок рисунка ❺. Для установки числовых меток для оси y используем метод `plt.yticks()`, который отразит количество наиболее населенных городов ❻. Наконец, выводим на экран рисунок с помощью `plt.show()`. На рис. 8.8 показан результат.

Как видим, в целом рисунок имеет тот же вид, что и графики, которые мы создали ранее в этой главе, в частности гистограмма с рис. 8.3. Это неудивительно, поскольку они были сгенерированы той же библиотекой Matplotlib, которую использует pandas.

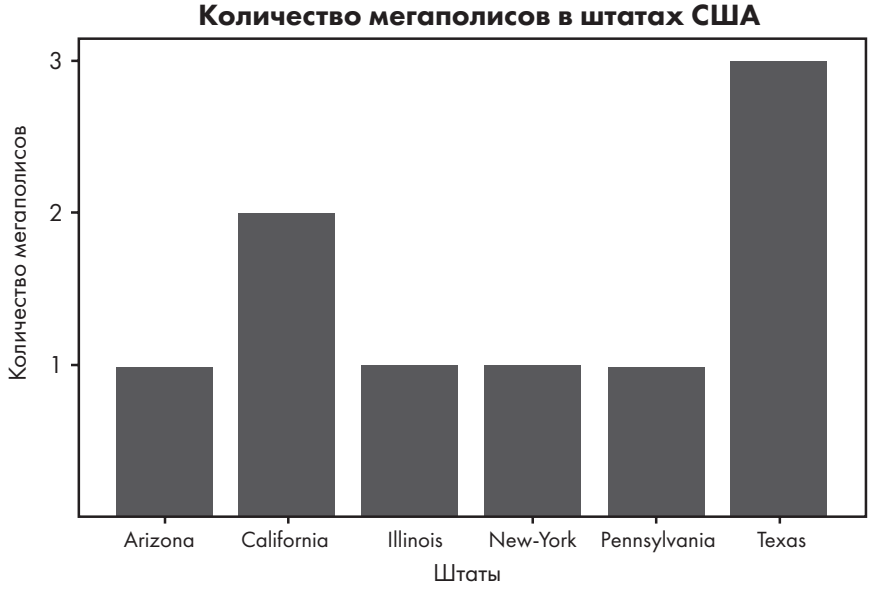

Рис. 8.8. Гистограмма, сгенерированная из объекта pandas DataFrame (названия городов взяты из файла us-cities-top-1k.csv)

Отображение данных геолокации с помощью Cartopy

Cartopy — это библиотека Python для геопространственной визуализации, или создания карт. Она включает в себя программный интерфейс `matplotlib.pyplot` для удобного рисования карт.

По сути, создание карты с помощью Cartopy сводится к созданию рисунка Matplotlib с координатами долготы, нанесенными по оси *x*, и координатами широты по оси *y*. Cartopy выполняет перевод сферической формы Земли в двумерную плоскость графика. Для демонстрации будем использовать набор данных *us-cities-top-1k.csv* из предыдущего раздела, чтобы построить контурные карты Южной Калифорнии с указанием расположения различных городов. Но прежде нужно установить Cartopy.

Использование Cartopy в Google Colab

Установить Cartopy может быть непросто, в зависимости от системы. Поэтому в данном разделе показано, как использовать Cartopy через веб-интерфейс Google Colab, который позволяет писать и выполнять код Python через браузер.

Для загрузки перейдите на официальный сайт Google Colab[1]. Затем нажмите New Notebook (Создать блокнот), чтобы начать работу в новом блокноте Colab, где можно создавать, заполнять и запускать произвольное количество ячеек кода. В каждой ячейке может быть одна или несколько строк кода Python. Выполнить код можно нажатием кнопки запуска в левом верхнем углу ячейки. Colab запоминает состояние выполнения всех ранее запущенных ячеек, подобно сессии интерпретатора Python. Новую ячейку кода можно создать с помощью кнопки +Code (+Код) в левом верхнем углу окна Colab.

В первой ячейке введите и выполните следующую команду для установки Cartopy в ваш блокнот Colab:

```
!pip install cartopy
```

После установки Cartopy можно перейти к примерам следующего раздела, выполняя каждый отдельный фрагмент кода в отдельной ячейке.

ПРИМЕЧАНИЕ

Если вы хотите установить Cartopy непосредственно в систему, обратитесь к документации[2].

Создание карт

В этом разделе мы используем Cartopy для создания двух карт Южной Калифорнии. Сначала построим карту, на которой будут показаны все города Южной Калифорнии из датасета *us-cities-top-1k.csv*. Начнем с импортирования необходимых модулей:

```
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
from cartopy.mpl.ticker import LongitudeFormatter, LatitudeFormatter
```

Нам понадобится pandas, интерфейс `matplotlib.pyplot` и несколько модулей Cartopy: `cartopy.crs` для создания карт и `LongitudeFormatter` для правильного форматирования меток. Команда `%matplotlib inline` нужна для отображения рисунков Matplotlib в блокноте Google Colab рядом с кодом.

[1] https://colab.research.google.com

[2] https://scitools.org.uk/cartopy/docs/latest/installing.html

Далее загружаем необходимые данные и рисуем карту:

```
❶ us_cities = pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/
                                master/us-cities-top-1k.csv")
❷ calif_cities = us_cities[us_cities.State.eq('California')]
❸ fig, ax = plt.subplots(figsize=(15,8))
❹ ax = plt.axes(projection=ccrs.Mercator())
❺ ax.coastlines('10m')
❻ ax.set_yticks([32,33,34,35,36], crs=ccrs.PlateCarree())
   ax.set_xticks([-121, -120, -119, -118, -117, -116, -115],
                 crs=ccrs.PlateCarree())
❼ lon_formatter = LongitudeFormatter()
   lat_formatter = LatitudeFormatter()
   ax.xaxis.set_major_formatter(lon_formatter)
   ax.yaxis.set_major_formatter(lat_formatter)
❽ ax.set_extent([-121, -115, 32, 36])
   X = calif_cities['lon']
   Y = calif_cities['lat']
❾ ax.scatter(X, Y, color='red', marker='o', transform=ccrs.PlateCarree())
   plt.show()
```

Загружаем датасет *us-cities-top-1k.csv* и преобразуем его в объект DataFrame ❶, как в предыдущем разделе. Помните, что он содержит геолокацию в виде координат широты и долготы, а также данные о населении. Затем фильтруем данные, чтобы включить только города штата Калифорния, используя метод `eq()` ❷, сокращение от *equal to* (равно).

Поскольку отображение карты требует более тщательной настройки, чем можно осуществить в интерфейсе `matplotlib.pyplot`, необходимо работать непосредственно с основными объектами визуализации Matplotlib. Поэтому вызываем функцию `plt.subplots()` для получения `Figure` и единственного объекта `Axes`, параллельно задавая размер фигуры ❸. Затем вызываем `plt.axes()`, чтобы переопределить объект `Axes` как карту Cartopy ❹. Для этого нужно сообщить Matplotlib, что при построении координат на плоской поверхности рисунка нужно использовать проекцию Mercator Cartopy. Проекция Mercator — это стандартный метод создания карт, который превращает поверхность Земли из сферы в цилиндр, а затем разворачивает этот цилиндр в прямоугольник.

Далее мы вызываем `ax.coastlines()` для отображения контуров суши на карте ❺. Контуры побережья добавляются к текущему объекту `Axes` из коллекции векторных географических файлов Natural Earth *coastline*. Обозначение `10m` указывает, что береговые линии нарисованы в масштабе 1 : 10 миллионов, то есть 1 сантиметр на карте соответствует 100 километрам на местности.

ПРИМЕЧАНИЕ

Посетите сайт Natural Earth Data[1], чтобы узнать больше о датасетах Natural Earth.

Для определения типа меток на осях *y* и *x* используются методы `set_yticks()` и `set_xticks()`, передающие список значений широты и долготы соответственно ❻. В частности, в качестве меток передаем значения `32...36` для оси *y* и `-121 ... -115` для оси *x* (то есть от 32°N до 36°N и от 121°W до 115°W), поскольку территория Южной Калифорнии ограничена именно этими координатами. В обоих случаях добавляем `crs=ccrs.PlateCarree()`, чтобы указать, как проецировать информацию о широте и долготе на плоскую поверхность.

Как и Mercator, проекция Plate Carrée отображает Землю как развернутый в виде прямоугольника цилиндр.

Далее с помощью объектов Cartopy `LongitudeFormatter()` и `LatitudeFormatter()` создаем средства форматирования и применяем к осям *x* и *y* ❼. Это гарантирует, что значения долготы и широты будут сопровождаться знаками градусов и *W* и *N* для *запада* и *севера* соответственно. Также мы задаем границы участка, указывая соответствующие значения долготы и широты, чтобы ограничить карту территорией Южной Калифорнии ❽. Затем извлекаем два объекта pandas Series из датафрейма: `X` для значений долготы и `Y` — для широты. Наконец, рисуем карту с помощью метода Matplotlib `scatter()` ❾, передавая данные для осей *x* и *y* вместе с инструкцией отображать города в виде красных точек. На рис. 8.9 показан результат.

Карта дает наглядное представление о районах с высокой плотностью населения. Но что, если нам нужны только самые крупные города и их названия? В этом случае делаем следующее:

```
❶ top_calif_cities = calif_cities[calif_cities.Population.ge(400000)]
  fig, ax = plt.subplots(figsize=(15,8))
  ax = plt.axes(projection=ccrs.Mercator())
  ax.coastlines('10m')
  ax.set_yticks([32,33,34,35,36], crs=ccrs.PlateCarree())
  ax.set_xticks([-121, -120, -119, -118, -117, -116, -115],
                crs=ccrs.PlateCarree())
  lon_formatter = LongitudeFormatter()
  lat_formatter = LatitudeFormatter()
  ax.xaxis.set_major_formatter(lon_formatter)
  ax.yaxis.set_major_formatter(lat_formatter)
  ax.set_extent([-121, -115, 32, 36])
```

[1] https://www.naturalearthdata.com

```
   X = top_calif_cities['lon']
   Y = top_calif_cities['lat']
❷  cities = top_calif_cities['City']
   ax.scatter(X, Y, color='red', marker='o', transform=ccrs.PlateCarree())
❸  for i in X.index: label = cities[i]
     plt.text(X[i], Y[i]+0.05, label, clip_on = True, fontsize = 20,
              horizontalalignment='center', transform=ccrs.Geodetic())
   plt.show()
```

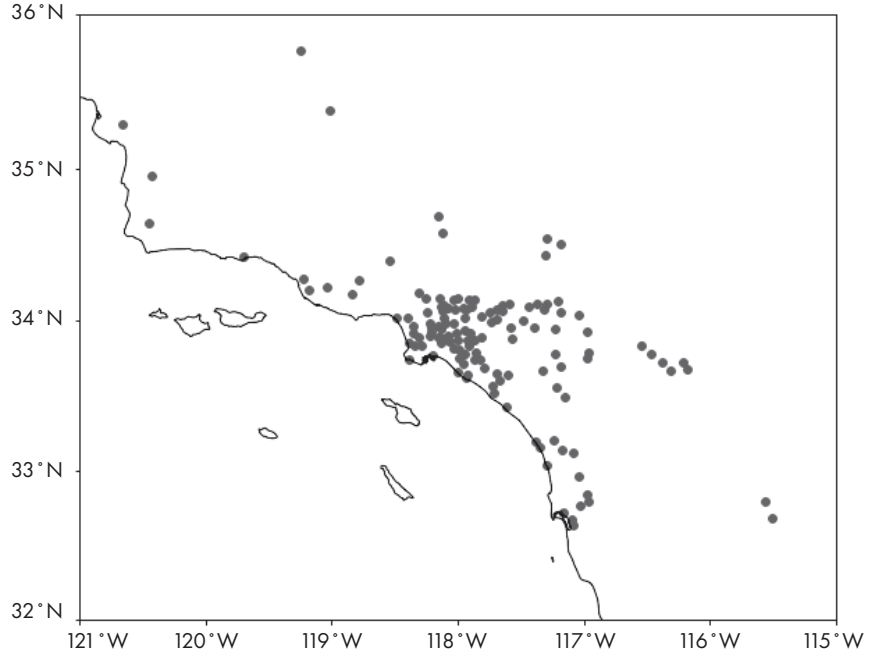

Рис. 8.9. Контурная карта Южной Калифорнии с обозначением городов

Мы фильтруем датафрейм `calif_cities`, созданный в предыдущем фрагменте кода, чтобы включить только города с населением 400 000 человек и более ❶. Затем создаем график, следуя тому же алгоритму, что и раньше, но с несколькими дополнительными шагами для добавления меток городов. Названия городов сохраняем в серию pandas `cities` ❷, а затем проходим по ней, назначая названия городов центрированными метками точек на карте с помощью метода Matplotlib `plt.text()` ❸. Указываем `transform=ccrs.Geodetic()`, чтобы Matplotlib использовал геодезическую систему координат Cartopy при добавлении меток. Эта

система воспринимает Землю как сферу и определяет координаты как значения широты и долготы. На рис. 8.10 показан результат.

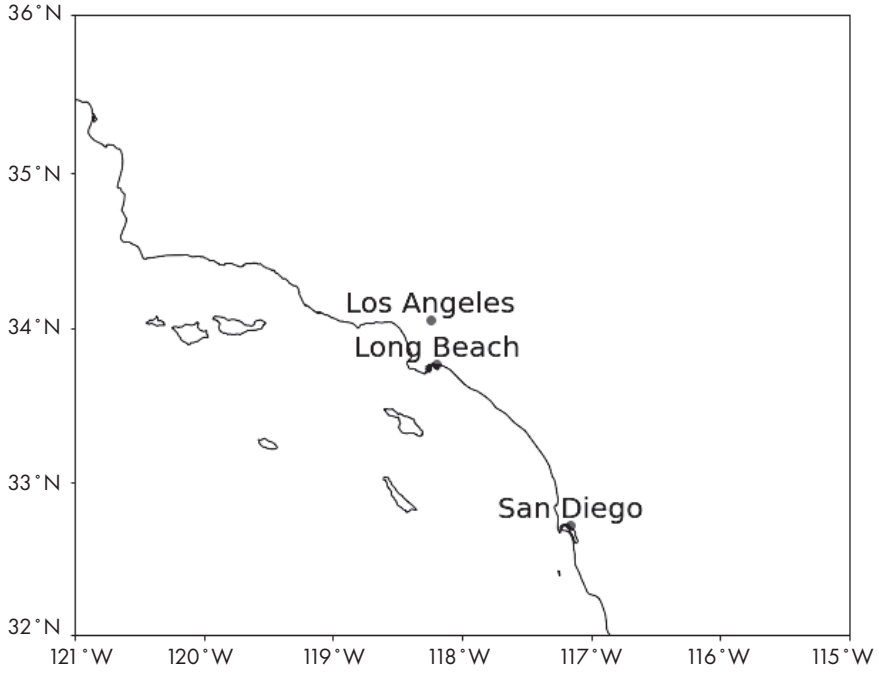

Рис. 8.10. Крупнейшие города Южной Калифорнии (названия городов взяты из файла us-cities-top-1k.csv)

Теперь на карте показано расположение и названия трех городов Южной Калифорнии с населением более 400 000 человек.

УПРАЖНЕНИЕ № 13: СОСТАВЛЕНИЕ КАРТЫ С ПОМОЩЬЮ CARTOPY И MATPLOTLIB

Теперь, когда вы знаете, как использовать Cartopy и Matplotlib, создайте карту другого региона США с указанием городов, например Северной Калифорнии. Вам необходимо будет задать другие значения широты и долготы для осей y и x.

Выводы

Как вы видели, визуализация — мощный инструмент для определения трендов и аналитики данных. Например, линейная диаграмма наглядно демонстрирует тренд изменения цены акций, а на карте можно точно отобразить районы с высокой плотностью населения. Из этой главы вы узнали, как с помощью библиотеки Matplotlib создавать такие распространенные типы визуализации, как линейные и столбчатые диаграммы, круговые диаграммы и гистограммы. Мы разобрали, как рисовать простые, но мощные графики с помощью интерфейса `matplotlib.pyplot` и как получать больший контроль над результатом, напрямую управляя основными объектами визуализации — `Figure` и `Axes`. Мы также рассмотрели, как использовать Matplotlib вместе с pandas для визуализации данных объекта DataFrame, и попрактиковались в создании карт с помощью Matplotlib и библиотеки обработки данных геолокации Cartopy.

9

Анализ данных о местоположении

Любое событие происходит в каком-либо месте, поэтому при анализе данных местоположение объекта может быть столь же важным, как и его непространственные характеристики. На самом деле пространственные и непространственные данные почти всегда взаимосвязаны.

В качестве примера рассмотрим приложение для заказа такси. Заказав поездку, вы, вероятно, захотите отслеживать местоположение машины на карте в режиме реального времени, пока она едет к вам. И вам также понадобится базовая информация, не связанная с местоположением, например данные об автомобиле и водителе, назначенном на ваш заказ: марка и модель, рейтинг водителя и т. д.

Из предыдущей главы вы узнали, как на основе данных о местоположении создавать карты. В этой главе вы узнаете больше о том, как использовать Python для сбора и анализа геоданных, и увидите, как интегрировать в анализ и пространственные, и непространственные характеристики. Мы рассмотрим пример работы приложения службы такси и попытаемся ответить на главный вопрос: какой автомобиль назначать на каждый конкретный заказ.

Получение данных о местоположении

Первый шаг пространственного анализа — получение данных о местоположении интересующих объектов. В частности, эти данные о местоположении должны

быть представлены *географическими координатами* (или просто *геокоординатами*) — значениями широты и долготы. Такая система координат позволяет задать любую точку на планете в виде набора чисел. Это означает, что местоположение можно анализировать программно. В этом разделе мы рассмотрим способы получения геокоординат как неподвижных, так и движущихся объектов. Это покажет, несколько хорошо наша гипотетическая служба такси сможет определять место посадки клиента и локацию автомобилей в реальном времени.

Преобразование стандартного вида адреса в геокоординаты

Большинство людей ориентируется по названиям улиц и номерам домов, а не по геокоординатам. Именно поэтому в службах такси, доставки еды и т. п. места подачи машины, получения заказа или высадки обычно указываются в виде фактического адреса. Однако многие из этих служб скрыто преобразуют понятные для человека адреса в соответствующие геокоординаты. Таким образом, приложение выполняет вычисления с данными о местоположении, например определяет ближайшее к указанному месту посадки доступное такси.

Как преобразовать названия улиц и номера домов в геокоординаты? Один из способов — использовать Geocoding (Google API, предназначенный именно для этой цели). Чтобы взаимодействовать с Geocoding API из Python-скрипта, понадобится библиотека googlemaps. Установите ее с помощью команды `pip`:

```
$ pip install -U googlemaps
```

Также необходимо получить API-ключ для Geocoding API, используя учетную запись Google Cloud. Информацию о приобретении ключа API см. на Google Maps Platform[1]. Подробнее о стоимости API можно узнать в разделе Pricing[2]. На момент написания этой статьи Google предоставлял пользователям API ежемесячный кредит в размере $200, которого вполне достаточно, чтобы экспериментировать с кодом, приведенным в этой книге.

В следующем фрагменте кода показан пример вызова Geocoding API с использованием googlemaps. В нем мы получаем координаты широты и долготы, соответствующие адресу 1600 Amphitheatre Parkway, Mountain View, CA:

```
import googlemaps

gmaps = googlemaps.Client(key='YOUR_API_KEY_HERE')
address = '1600 Amphitheatre Parkway, Mountain View, CA'
```

[1] https://developers.google.com/maps/documentation/geocoding/get-api-key

[2] https://cloud.google.com/maps-platform/pricing

```
geocode_result = gmaps.geocode(address)

print(geocode_result[0]['geometry']['location'].values())
```

В приведенном выше скрипте мы устанавливаем соединение с API и отправляем адрес, который требуется преобразовать. API возвращает документ JSON с вложенной структурой. Геокоординаты хранятся в поле `location`, которое является подполем `geometry`. В последней строке мы обращаемся к координатам и выводим их на экран, получая следующий результат:

```
dict_values([37.422388, -122.0841883])
```

Получение геокоординат движущегося объекта

Теперь вы знаете, как получить геокоординаты статичного объекта по его адресу, но как получить местоположение движущегося объекта, например автомобиля в режиме реального времени? Некоторые службы такси используют для этой цели специализированные GPS-устройства, но мы остановимся на недорогом и простом в реализации решении. Все, что нам потребуется, — это смартфон.

Смартфоны определяют свое местоположение с помощью встроенных датчиков GPS и могут быть настроены на обмен этой информацией. Здесь мы рассмотрим, как собирать GPS-координаты смартфона через популярный мессенджер Telegram. Используя Telegram Bot API, создадим *telegram-бота*. Боты обычно используются для обработки естественного языка, однако наш бот будет собирать и регистрировать геолокацию пользователей Telegram, которые решили поделиться данными с ботом.

Установка telegram-бота

Чтобы создать telegram-бота, установите приложение Telegram и создайте учетную запись. Затем выполните следующие действия, используя смартфон либо ПК:

1. В приложении Telegram найдите @BotFather. BotFather — это telegram-бот, который управляет всеми остальными ботами вашего аккаунта.
2. В диалоге с BotFather нажмите Start, чтобы увидеть список команд для настройки telegram-ботов.
3. Введите `/newbot` в поле сообщения. Вам будет предложено ввести название и имя пользователя для нового бота. Затем вы получите для него токен авторизации. Запишите этот токен; он понадобится при программировании бота.

После выполнения этих шагов вы можете создать бота на языке Python с помощью библиотеки python-telegram-bot. Установите библиотеку следующим образом:

```
$ pip install python-telegram-bot –upgrade
```

Инструменты, необходимые, чтобы написать бота, находятся в модуле библиотеки `telegram.ext`. Он построен на базе Telegram Bot API.

Программирование бота

Мы используем модуль `telegram.ext` библиотеки python-telegram-bot, чтобы запрограммировать бота на прослушивание и логирование GPS-координат:

```
from telegram.ext import Updater, MessageHandler, Filters
from datetime import datetime
import csv

❶ def get_location(update, context):
    msg = None
    if update.edited_message:
      msg = update.edited_message
    else:
      msg = update.message
  ❷ gps = msg.location
    sender = msg.from_user.username
    tm = datetime.now().strftime("%H:%M:%S")
    with open(r'/HOME/PI/LOCATION_BOT/LOG.CSV', 'a') as f:
      writer = csv.writer(f)
    ❸ writer.writerow([sender, gps.latitude, gps.longitude, tm])
    ❹ context.bot.send_message(chat_id=msg.chat_id, text=str(gps))
def main():
  ❺ updater = Updater('TOKEN', use_context=True)
  ❻ updater.dispatcher.add_handler(MessageHandler(Filters.location,
                                   get_location))
  ❼ updater.start_polling()
  ❽ updater.idle()
if __name__ == '__main__':
    main()
```

Функция `main()` содержит общие вызовы, находящиеся в скрипте, где реализован telegram-бот. Начинаем с создания объекта `Updater` ❺, передав ему токен авторизации бота (сгенерированный BotFather). Этот объект организует процесс выполнения бота на протяжении всего скрипта. Затем используем объект

`Dispatcher`, связанный с `Updater`, чтобы добавить функцию-обработчик `get_location()` для входящих сообщений ❻. Указав `Filters.location`, мы добавляем фильтр к обработчику так, чтобы он вызывался только тогда, когда бот получает сообщения, содержащие данные о местоположении отправителя. Запускаем бота, вызывая метод `start_polling()` объекта `Updater` ❼. Поскольку `start_polling()` является неблокирующим методом, необходимо также вызвать метод `idle()` объекта `Updater` ❽ для блокировки скрипта до получения сообщения.

В начале скрипта мы объявляем обработчик `get_location()` ❶. В обработчике сохраняем входящее сообщение как переменную `msg`, затем извлекаем данные о местоположении отправителя, используя свойство сообщения `location` ❷. Также мы логируем имя отправителя и генерируем строку, содержащую текущее время. Затем, используя модуль Python `csv`, сохраняем всю эту информацию в виде строки в CSV-файле ❸ по выбранному пути. Мы также отправляем данные о местоположении назад отправителю, чтобы он знал, что его местоположение получено ❹.

Получение данных от бота

Запустите скрипт на машине, подключенной к интернету. Как только вы сделаете это, пользователи смогут начать делиться с ботом данными о своем местоположении в режиме реального времени, выполняя несколько простых шагов.

1. Создайте учетную запись Telegram.
2. В Telegram нажмите на имени бота.
3. Коснитесь значка скрепки и выберите в меню Location (Геопозиция).
4. Нажмите Share My Location For (Транслировать мою геопозицию) и выберите время, в течение которого Telegram будет делиться с ботом данными о местоположении в реальном времени. Возможные варианты: 15 минут, 1 час или 8 часов.

На рис. 9.1 представлен скриншот, демонстрирующий эту процедуру.

Бот будет отправлять полученные от пользователей данные об их местоположении в CSV-файл в виде строк, имеющих вид:

```
cab_26,43.602508,39.715685,14:47:44
cab_112,43.582243,39.752077,14:47:55
cab_26,43.607480,39.721521,14:49:11
cab_112,43.579258,39.758944,14:49:51
cab_112,43.574906,39.766325,14:51:53
cab_26,43.612203,39.720491,14:52:48
```

Рис. 9.1. Как поделиться информацией о местоположении в Telegram

Первое поле в каждой строке содержит имя пользователя, второе и третье поля хранят широту и долготу местоположения пользователя, а четвертое — метку времени. Для некоторых задач, например для поиска ближайшего автомобиля к определенному месту подачи, понадобится только последняя строка с данными об автомобиле. Однако для других задач, таких как расчет общего расстояния поездки, пригодятся несколько строк с информацией о машине, отсортированных по времени.

Анализ пространственных данных с помощью geopy и Shapely

Анализ пространственных данных сводится к ответам на вопросы о взаимосвязях: какой объект находится ближе всего к определенному месту? Находятся ли два объекта в одной области? В этом разделе мы ответим на эти вопросы пространственного анализа в контексте нашего примера службы такси. Для этого будем использовать две библиотеки Python — geopy и Shapely.

Поскольку geopy предназначена для выполнения расчетов на основе геокоординат, она особенно хорошо справляется с ответами на вопросы о расстоянии. В свою очередь, Shapely специализируется на определении и анализе геометрических плоскостей, поэтому она идеально подходит, чтобы выяснить, попадает ли тот или иной объект в указанную область. Вы увидите, что обе библиотеки играют роль в определении самого подходящего автомобиля для конкретного заказа.

Прежде чем двигаться дальше, установим их:

```
$ pip install geopy
$ pip install shapely
```

Поиск ближайшего объекта

Продолжая пример со службой такси, рассмотрим, как использовать данные о местоположении для определения машины, ближайшей к месту подачи. Для начала нам понадобится образец данных о местоположении. Если вы развернули telegram-бота, о котором говорилось в предыдущем разделе, то возможно, у вас уже есть некоторые данные в виде CSV-файла. Во фрагменте кода ниже мы загружаем данные и преобразуем их в объект pandas DataFrame, чтобы их было легко сортировать и фильтровать:

```
import pandas as pd
df = pd.read_csv("HOME/PI/LOCATION_BOT/LOG.CSV", names=['cab', 'lat',
                                                        'long', 'tm'])
```

Если вы не развернули telegram-бота, можно создать список кортежей с примерами данных о местоположении и привести его к типу DataFrame следующим образом:

```
import pandas as pd
locations = [
  ('cab_26',43.602508,39.715685,'14:47:44'),
  ('cab_112',43.582243,39.752077,'14:47:55'),
  ('cab_26',43.607480,39.721521,'14:49:11'),
  ('cab_112',43.579258,39.758944,'14:49:51'),
  ('cab_112',43.574906,39.766325,'14:51:53'),
  ('cab_26',43.612203,39.720491,'14:52:48')
]

df = pd.DataFrame(locations, columns =['cab', 'lat', 'long', 'tm'])
```

В обоих случаях у вас будет датафрейм `df` со следующими столбцами: идентификатор такси (`cab`), широта (`lat`), долгота (`long`) и метка времени (`tm`).

ПРИМЕЧАНИЕ

Если вы хотите создать свои собственные образцы данных с геолокацией, то самый простой способ сделать это — найти координаты широты и долготы с помощью Google Maps. При щелчке правой кнопкой мыши места на карте координаты широты и долготы этой геолокации будут отображаться в первой строке меню.

Для каждого автомобиля в датафрейме содержится несколько строк, но для определения ближайшего к месту подачи автомобиля понадобятся только последние координаты машины. Отфильтровать ненужные строки можно так:

```
latestrows = df.sort_values(['cab','tm'],ascending=False).drop_duplicates('cab')
```

Здесь мы сортируем строки по полям `cab` и `tm` в порядке убывания. Эта операция группирует датасет по столбцу `cab` и помещает последнюю строку для каждой группы (автомобиля) на первое место. Затем мы применяем метод `drop_duplicates()` для удаления всех строк с данными об автомобиле, кроме первой. Итоговый датафрейм `latestrows` выглядит так:

```
        cab         lat        long         tm
5    cab_26   43.612203   39.720491   14:52:48
3   cab_112   43.574906   39.766325   14:51:53
```

Теперь у нас есть датафрейм, содержащий только данные о последнем местоположении каждого такси. Для удобства дальнейших вычислений преобразуем датафрейм в более простую структуру Python — список списков. Таким образом легче добавлять новые поля в каждую строку, например поле для расстояния между такси и местом подачи:

```
latestrows = latestrows.values.tolist()
```

Свойство `values` датафрейма `latestrows` возвращает NumPy-представление датафрейма, которое мы затем преобразуем в список списков с помощью `tolist()`.

Теперь можно рассчитать расстояние от места посадки до каждого автомобиля. Используем библиотеку geopy, с помощью которой эта задача решается всего

несколькими строками кода. Во фрагменте кода ниже применим функцию `distance()` из модуля `geopy` для проведения необходимых расчетов:

```
from geopy.distance import distance
pick_up = 43.578854, 39.754995

for i,row in enumerate(latestrows):
❶  dist = distance(pick_up, (row[1],row[2])).m
   print(row[0] + ':', round(dist))
   latestrows[i].append(round(dist))
```

Для простоты указываем место подачи, вручную задавая координаты широты и долготы. Однако на практике можно использовать Google Geocoding API для автоматической генерации координат из улицы и номера дома, как обсуждалось выше. Затем проходим по каждой строке датасета и вычисляем расстояние между каждым автомобилем и местом посадки с помощью вызова `distance()` ❶. Эта функция принимает в качестве аргументов два кортежа с координатами широты и долготы. Добавив `.m`, мы получим расстояние в метрах. Чтобы посмотреть результат, выводим на экран каждое вычисленное расстояние, а затем добавляем его в конец строки как новое поле. Ниже представлен результат выполнения скрипта:

```
cab_112: 1015
cab_26: 4636
```

Очевидно, что `cab_112` ближе, но как это определить программным путем? Для этого подойдет встроенная в Python функция `min()`:

```
closest = min(latestrows, key=lambda x: x[4])
print('Ближайшая машина: ', closest[0], ' – расстояние в метрах: ',
closest[4])
```

Передаем данные в `min()` и применяем лямбда-функцию, чтобы задать порядок сортировки строк по элементу с индексом `4`. Таким образом мы добавим расчет расстояния. Выводим результат в формате, пригодном для чтения, и получаем:

```
Ближайшая машина:   cab_112   - расстояние в метрах:   1015
```

В данном примере мы рассчитали расстояние по прямой между каждой машиной и местом подачи. Хотя эта информация, безусловно, полезна, в реальности автомобили почти никогда не ездят по идеально прямой линии. Фактическое

расстояние до места посадки пассажира будет больше, чем расстояние по прямой. Исходя из этого, рассмотрим более надежный способ подбора автомобиля для конкретного места подачи.

Поиск объектов в определенной области

Часто, чтобы определить наиболее подходящее такси, нужно задать вопрос не «Какая машина ближе всего?», а «Какая машина находится в районе места подачи?». Это справедливо не только потому, что фактическое расстояние от одной точки до другой оказывается почти всегда больше, чем расстояние между точками по прямой. На практике такие преграды, как река или железнодорожные пути, часто разделяют географические области на отдельные зоны, которые соединяются только в ограниченном количестве точек в виде мостов, туннелей и т. п. Поэтому непосредственное расстояние может быть весьма обманчивым. Рассмотрим пример на рис. 9.2.

Рис. 9.2. Препятствие в виде реки может ввести в заблуждение при измерении расстояния (pick-up — место подачи)

Видим, что на рисунке `cab_26` находится ближе всего к месту посадки, но из-за реки `cab_112`, скорее всего, доберется туда быстрее. До этого несложно додуматься, глядя на карту, но как прийти к этому же выводу в Python-скрипте? Один из способов — разделить территорию на несколько небольших *многоугольников*, или областей, окруженных набором связанных прямых линий, а затем проверить, какие автомобили находятся в том же многоугольнике, что и место посадки.

В данном конкретном примере необходимо определить многоугольник, который охватывает место посадки и ограничен рекой. Границы многоугольника можно определить вручную через Google Maps: щелкните правой кнопкой мыши на нескольких точках, которые, соединяясь, образуют замкнутый многоугольник, и запишите геокоординаты каждой точки. Получив координаты, объявите многоугольник в скрипте с помощью библиотеки Shapely.

Вот как создать многоугольник с помощью Shapely и проверить, находится ли заданная точка внутри этого многоугольника:

```
❶ from shapely.geometry import Point, Polygon
  coords = [(46.082991, 38.987384), (46.075489, 38.987599), (46.079395,
            38.997684), (46.073822, 39.007297), (46.081741, 39.008842)]
❷ poly = Polygon(coords)
❸ cab_26 = Point(46.073852, 38.991890)
  cab_112 = Point(46.078228, 39.003949)
  pick_up = Point(46.080074, 38.991289)

❹ print('cab_26 within the polygon:', cab_26.within(poly))
  print('cab_112 within the polygon:', cab_112.within(poly))
  print('pick_up within the polygon:', pick_up.within(poly))
```

Сначала импортируем два класса Shapely: `Point` и `Polygon` ❶. Затем создаем объект `Polygon` с помощью списка кортежей с широтой и долготой ❷. Этот объект представляет собой область к северу от реки, включающую место посадки. Далее создаем несколько объектов `Point`, представляющих местоположение `cab_26`, `cab_112` и место подачи соответственно ❸. Наконец, выполняем несколько запросов, содержащих геоданные, чтобы определить, находится ли определенная точка внутри многоугольника, используя метод Shapely's `within()` ❹. В результате выполнения скрипта получаем:

```
cab_26 within the polygon: False
cab_112 within the polygon: True
pick_up within the polygon: True
```

УПРАЖНЕНИЕ № 14: ОПРЕДЕЛЕНИЕ ДВУХ И БОЛЕЕ МНОГОУГОЛЬНИКОВ

В предыдущем разделе мы использовали многоугольник, охватывающий одну область на карте. Теперь попробуйте задать два или более соседних многоугольника, охватывающих определенные районы города, разделенные препятствием, например рекой. Получи́те координаты этих многоугольников, используя Google-карты своего города или любого другого населенного пункта планеты. Вам также понадобятся координаты нескольких точек, входящих в многоугольники, чтобы смоделировать местоположение автомобилей и места подачи.

В скрипте задайте многоугольники с помощью Shapely и сгруппируйте их в словарь, а точки, представляющие машины, соберите в другой словарь. Затем разделите автомобили на группы в зависимости от того, в каком многоугольнике они находятся. Это можно сделать с помощью двух циклов: внешнего (для перебора многоугольников) и внутреннего (для перебора точек, представляющих автомобили). На каждой итерации внутреннего цикла проверяйте, находится ли точка подачи внутри многоугольника. Следующий фрагмент кода иллюстрирует, как это реализовать:

```
--фрагмент--
cabs_dict ={}
polygons = {'poly1': poly1, 'poly2': poly2}
cabs = {'cab_26': cab_26, 'cab_112': cab_112}
for poly_name, poly in polygons.items():
  cabs_dict[poly_name] = []
  for cab_name, cab in cabs.items():
    if cab.within(poly):
      cabs_dict[poly_name].append(cab_name)
--фрагмент--
```

Далее необходимо определить, в каком многоугольнике находится место посадки. Узнав эту информацию, выберите соответствующий список машин из словаря `cabs_dict`, используя название многоугольника в качестве ключа. Наконец, с помощью geopy определите, какое такси в выбранном многоугольнике находится ближе всего к месту подачи.

Объединение двух подходов

До сих пор мы выбирали наиболее подходящий автомобиль либо в пределах определенной области, либо рассчитывая линейное расстояние до точки. На самом деле, самый точный результат может дать комбинирование этих двух подходов, поскольку не всегда нужно слепо исключать все такси, которые не находятся в том же многоугольнике, что и пассажир. Такси в соседней области может оказаться ближе всего по фактическому расстоянию поездки, даже если учесть, что придется делать крюк до моста через реку или объезжать другое препятствие. Ключевой момент — учесть пункты проезда из одного многоугольника в другой. На рис. 9.3 показано, как это сделать.

Пунктирная линия, проходящая через середину рисунка, представляет собой границу, разделяющую территорию на два многоугольника: к северу и к югу от

реки. Знак равенства, установленный на мосту, обозначает пункт проезда, через который автомобили могут попасть из одного многоугольника в другой. Расстояние от такси, находящихся в многоугольнике, который граничит с областью, где находится точка посадки, состоит из двух частей: расстояния между текущим местоположением такси и пунктом проезда и расстояния между пунктом проезда и точкой посадки.

Рис. 9.3. Использование пунктов проезда между смежными областями

Чтобы найти ближайшую машину, необходимо определить, в каком многоугольнике находится каждый автомобиль, и на основе этих данных рассчитать расстояние от такси до места подачи: либо по прямой, если такси находится в том же многоугольнике, либо через пункт проезда, если оно находится в соседнем многоугольнике. Рассчитаем расстояние только для `cab_26`:

```
from shapely.geometry import Point, Polygon
from geopy.distance import distance

coords = [(46.082991, 38.987384), (46.075489, 38.987599), (46.079395,
          38.997684), (46.073822, 39.007297), (46.081741, 39.008842)]
❶ poly = Polygon(coords)
❷ cab_26 = Point(46.073852, 38.991890)
pick_up = Point(46.080074, 38.991289)
entry_point = Point(46.075357, 39.000298)

if cab_26.within(poly):
❸ dist = distance((pick_up.x, pick_up.y), (cab_26.x,cab_26.y)).m
else:
❹ dist = distance((cab_26.x,cab_26.y), (entry_point.x,entry_point.y)).m +
         distance((entry_point.x,entry_point.y), (pick_up.x, pick_up.y)).m

print(round(dist))
```

В скрипте используется как Shapely, так и geopy. Сначала задаем объект Shapely `Polygon`, включающий местоположение пассажира, как мы делали ранее ❶. Аналогично задаем объекты `Point` для автомобиля, места подачи и пункта проезда ❷. Затем вычисляем расстояние в метрах с помощью функции geopy `distance()`. Если такси находится внутри многоугольника, находим расстояние непосредственно между такси и местом подачи ❸. А если нет, то сначала вычисляем расстояние между машиной и пунктом проезда, а затем расстояние между этим пунктом и местом подачи. Складывая их, получаем общее расстояние ❹:

```
1544
```

УПРАЖНЕНИЕ № 15:
СОВЕРШЕНСТВОВАНИЕ АЛГОРИТМА ПОДБОРА МАШИНЫ

В скрипте выше мы обработали данные о местоположении лишь одного автомобиля, определив расстояние между ним и местом подачи. Измените скрипт таким образом, чтобы он вычислял расстояния между местом посадки и каждой машиной. Для этого объедините точки, представляющие такси, в список, а затем пройдите по этому списку в цикле, используя в теле блоки `if/else` из предыдущего скрипта. Затем определите ближайший к месту подачи автомобиль.

Объединение пространственных и непространственных данных

До сих пор в этой главе мы работали исключительно с пространственными данными, но важно понимать, что пространственный анализ часто требует учета и непространственных данных. Например, какой смысл от информации о том, что магазин находится в 16 километрах от текущего местоположения, если мы не уверены, что в нем есть нужный товар? Или, возвращаясь к примеру с такси, что нам даст возможность определить ближайшую машину к месту посадки, если мы не знаем, доступна она или же водитель выполняет другой заказ? В этом разделе мы рассмотрим, как учитывать непространственные данные в рамках пространственного анализа.

Получение непространственных характеристик

Информацию о доступности такси можно получить из датасета, содержащего заказы поездок. Как только на автомобиль будет назначен заказ, можно поместить

эту информацию в структуру данных `orders`, где будет указываться заказ и его статус: открытый (open) — в процессе либо закрытый (closed) — завершенный. Согласно этой схеме, отфильтровав открытые заказы, мы поймем, какие автомобили недоступны для выполнения нового заказа. Вот как реализовать эту логику в Python:

```
import pandas as pd
orders = [
  ('order_039', 'open', 'cab_14'),
  ('order_034', 'open', 'cab_79'),
  ('order_032', 'open', 'cab_104'),
  ('order_026', 'closed', 'cab_79'),
  ('order_021', 'open', 'cab_45'),
  ('order_018', 'closed', 'cab_26'),
  ('order_008', 'closed', 'cab_112')
]

df_orders = pd.DataFrame(orders, columns =['order','status','cab'])
df_orders_open = df_orders[df_orders['status']=='open']
unavailable_list = df_orders_open['cab'].values.tolist()
print(unavailable_list)
```

Список кортежей `orders`, используемый в этом примере, можно получить из более полного датасета, например коллекции всех заказов, открытых за последние два часа, которая включает дополнительную информацию о каждом заказе (место подачи, место высадки, время начала, время окончания и т. д.). Для простоты здесь датасет уже сокращен до полей, необходимых для текущей задачи. Мы преобразуем список в объект DataFrame, затем фильтруем его, чтобы включить только заказы со статусом `open`. Наконец, преобразуем DataFrame в список, содержащий только значения из столбца `cab`. Список недоступных такси выглядит следующим образом:

```
['cab_14', 'cab_79', 'cab_104', 'cab_45']
```

Вооружившись этим списком, проверим другие автомобили и определим, какой из них находится ближе всего к месту подачи. Добавим этот код к предыдущему скрипту:

```
from geopy.distance import distance
pick_up = 46.083822, 38.967845
cab_26 = 46.073852, 38.991890
cab_112 = 46.078228, 39.003949
cab_104 = 46.071226, 39.004947
```

```
cab_14 = 46.004859, 38.095825
cab_79 = 46.088621, 39.033929
cab_45 = 46.141225, 39.124934
cabs = {'cab_26': cab_26, 'cab_112': cab_112, 'cab_14': cab_14,
        'cab_104': cab_104, 'cab_79': cab_79, 'cab_45': cab_45}
dist_list = []

for cab_name, cab_loc in cabs.items():
  if cab_name not in unavailable_list:
    dist = distance(pick_up, cab_loc).m
    dist_list.append((cab_name, round(dist)))

print(dist_list)
print(min(dist_list, key=lambda x: x[1]))
```

Для демонстрации вручную определяем геокоординаты места подачи и всех машин в виде кортежей и передаем их в словарь, где ключами являются названия такси. Затем проходим по словарю и для каждого автомобиля, не входящего в `unavailable_list`, используем geopy для вычисления расстояния до места подачи. Наконец, выводим на экран весь список доступных такси с указанием расстояния до места подачи, а также ближайший автомобиль:

```
[('cab_26', 2165), ('cab_112', 2861)]
('cab_26', 2165)
```

В данном примере `cab_26` — ближайший доступный автомобиль.

УПРАЖНЕНИЕ № 16: ФИЛЬТРАЦИЯ ДАННЫХ С ПОМОЩЬЮ СПИСКОВОГО ВКЛЮЧЕНИЯ

В предыдущем разделе мы отфильтровали список `orders` так, чтобы остались лишь недоступные машины, предварительно преобразовав `orders` в DataFrame. Теперь попробуйте сгенерировать список `unavailable_list`, не используя pandas; вместо этого примените списковые включения. При таком подходе вы сможете получать список такси, назначенных на открытые заказы, одной строкой кода:

```
unavailable_list = [x[2] for x in orders if x[1] == 'open']
```

Вам не придется ничего менять в остальной части скрипта.

Объединение датасетов с пространственными и непространственными данными

В предыдущем примере мы хранили пространственные данные (местоположение каждого такси) и непространственные данные (доступные такси) в отдельных структурах данных. Однако иногда выгодно объединить пространственные и непространственные данные в одну структуру.

Учитывайте, что для назначения автомобиля на заказ кроме доступности может потребоваться проверка некоторых других условий. Например, клиенту может понадобиться такси с детским креслом. Чтобы найти нужную машину, необходимо использовать датасет, который включает непространственную информацию о машине, а также расстояние от каждого автомобиля до места подачи. Для вышеупомянутого случая можно использовать датасет, содержащий всего два столбца: название такси и наличие детского кресла. Вот как его создать:

```
cabs_list = [
  ('cab_14',1),
  ('cab_79',0),
  ('cab_104',0),
  ('cab_45',1),
  ('cab_26',0),
  ('cab_112',1)
]
```

В машинах, для которых указано значение `1` во второй колонке, есть детское кресло. Затем мы преобразуем список в датафрейм. Мы также создаем второй датафрейм из dist_list, списка доступных такси и их расстояния до места подачи, который мы сформировали в предыдущем разделе:

```
df_cabs = pd.DataFrame(cabs_list, columns =['cab', 'seat'])
df_dist = pd.DataFrame(dist_list, columns =['cab', 'dist'])
```

Теперь объединяем эти датафреймы по столбцу `cab`:

```
df = pd.merge(df_cabs, df_dist, on='cab', how='inner')
```

Используем inner join, то есть объединяем только такси, которые есть и в `df_cabs`, и в `df_dist`. На практике, поскольку `df_dist` содержит только доступные на текущий момент автомобили, недоступные машины будут исключены из итогового датасета. Теперь объединенный датафрейм включает как пространственные

(расстояние до места подачи для каждого автомобиля), так и непространственные данные (наличие или отсутствие детского кресла в каждом такси):

```
        cab   seat   dist
0    cab_26      0   2165
1   cab_112      1   2861
```

Преобразуем датафрейм в список кортежей, который затем фильтруем, оставляя только те строки, в поле `seat` которых установлено значение `1`:

```
result_list = list(df.itertuples(index=False,name=None))
result_list = [x for x in result_list if x[1] == 1]
```

Используем метод датафрейма `itertuples()` для преобразования каждой строки в кортеж, затем оборачиваем кортежи в список с помощью функции `list()`.

Последний шаг — определение строки с наименьшим значением в поле с расстоянием (индекс `2`):

```
print(min(result_list, key=lambda x: x[2]))
```

Вывод:

```
('cab_112', 1, 2861)
```

Сравните его с результатом предыдущего раздела. Как видите, из-за потребности в детском кресле на этот заказ пришлось назначить другую машину.

Выводы

На реальном примере службы такси в этой главе показано, как анализировать пространственные данные. Мы начали с рассмотрения примера преобразования стандартного вида адреса в геокоординаты с помощью Google Geocoding API и библиотеки Python googlemaps. Затем научились использовать telegram-бота для сбора данных о местоположении со смартфонов. Далее мы воспользовались библиотеками geopy и Shapely для выполнения фундаментальных геопространственных операций, таких как измерение расстояния между точками и определение того, находятся ли точки в определенной области. С помощью этих библиотек, встроенных структур данных Python и pandas DataFrame мы разработали приложение, определяющее подходящий автомобиль для указанного места подачи на основе различных пространственных и непространственных критериев.

10

Анализ данных временных рядов

Данные временно́го ряда, или *данные с метками времени*, — это набор точек, индексированных в хронологическом порядке. Классические примеры информации такого типа — экономические индексы, данные о погоде и показатели здоровья пациентов, которые фиксируются во времени. В этой главе рассматриваются методы анализа данных временных рядов и извлечения из них значимой статистики с помощью библиотеки pandas. Мы разберем их на примере анализа данных фондового рынка, но те же методы можно применять ко всем видам временных рядов.

Регулярные и нерегулярные временные ряды

Временной ряд может быть создан на основе любой переменной, которая изменяется во времени, и эти изменения могут регистрироваться либо через регулярные, либо через нерегулярные промежутки времени. Регулярные интервалы встречаются чаще. В финансовой сфере, например, временные ряды часто используются для отслеживания изменения цен на акции по дням, как показано ниже:

```
Дата          Цена закрытия
-----------   -------------
16-FEB-2022   10.26
17-FEB-2022   10.34
18-FEB-2022   10.99
```

Как видите, колонка `Date` содержит метки времени, расположенные в хронологическом порядке, для сохранения последовательности дней. Соответствующие точки данных, часто называемые *наблюдениями*, представлены в колонке `Цена закрытия`. Временные ряды такого типа называются *регулярными* или *непрерывными*, поскольку наблюдения фиксируются непрерывно через одинаковые промежутки времени.

Еще один пример регулярных временных рядов — ежеминутно записываемые координаты широты и долготы местонахождения автомобиля:

```
Время      Координаты
-------    ----------------
20:43:00   37.801618, -122.374308
20:44:00   37.796599, -122.379432
20:45:00   37.788443, -122.388526
```

Здесь метки времени не содержат дат, тем не менее они расположены в хронологическом порядке, минута за минутой.

В отличие от регулярных временных рядов, *нерегулярные* используются для регистрации последовательности событий по мере их фактического или планируемого возникновения, а не через регулярные промежутки времени. В качестве простого примера рассмотрим программу конференции:

```
Время     Событие
-------   ----------------------
8:00      Регистрация участников
9:00      Утренние сессии
12:10     Обед
12:30     Дневные сессии
```

Метки времени для этой серии точек распределены неравномерно — они зависят от того, сколько времени должно занять каждое событие.

Нерегулярные временные ряды обычно используются в приложениях, где данные поступают непредсказуемо. Разработчикам знакомы нерегулярные временные ряды по журналу ошибок, возникающих при работе сервера или выполнении приложения. Предсказать, когда возникнут ошибки, трудно, и почти наверняка они не будут происходить через регулярные промежутки времени. Приведем еще один пример: приложение, отслеживающее потребление электроэнергии, может использовать нерегулярный временной ряд для регистрации аномалий, таких как всплески и сбои, которые возникают произвольным образом.

Регулярные и нерегулярные временные ряды объединяет то, что точки данных в них расположены в хронологическом порядке. По сути, анализ временных рядов основывается на этой ключевой особенности. Строгий хронологический порядок позволяет последовательно сравнивать события или значения во временном ряду, выявляя ключевые статистические показатели и тенденции.

Например, в случае с фондовыми данными хронологический порядок позволяет отслеживать динамику акций с течением времени. В случае ежеминутно обновляемых геокоординат автомобиля можно использовать соседние значения для расчета расстояния, преодолеваемого за каждую минуту, а затем использовать это расстояние для сравнения изменения средней скорости автомобиля. В свою очередь, хронологический порядок программы конференции позволяет сразу увидеть ожидаемую продолжительность каждого мероприятия.

В некоторых случаях для анализа сами метки времени могут и не понадобиться; важно лишь, чтобы записи в ряду располагались хронологически. Рассмотрим следующий пример нерегулярного временного ряда, представленного в виде двух последовательных сообщений об ошибках, возвращаемых скриптом при попытке подключиться к базе данных MySQL с неправильным паролем:

```
_mysql_connector.MySQLInterfaceError: Access denied for user
'root'@'localhost' (using password: YES)
NameError: name 'cursor' is not defined
```

Второе сообщение об ошибке говорит о том, что переменная с именем `cursor` не определена. Однако первопричину проблемы можно понять, лишь взглянув на первое сообщение об ошибке: поскольку введен неверный пароль, соединение с базой данных не установлено и объект `cursor` не создан.

Анализ последовательности с сообщениями об ошибках — привычная задача для программистов, только обычно она выполняется не с помощью кода, а вручную. В оставшейся части главы мы сосредоточимся на временных рядах с числовыми точками данных, поскольку их легко анализировать с помощью Python. В частности, мы рассмотрим, как извлечь значимую информацию из регулярных временных рядов, содержащих данные фондового рынка.

Общие методы анализа временных рядов

Предположим, необходимо проанализировать временной ряд с ценами закрытия на каждый день в течение определенного периода времени. В этом разделе мы разберем некоторые общие методы, которые можно использовать в анализе, но сначала вам понадобятся данные о котировках.

Как вы видели в главах 3 и 5, биржевые данные можно получить через Python-скрипт с помощью библиотеки yfinance. Вот, например, данные для тикера TSLA (Tesla, Inc.) за пять последних биржевых дней:

```
import yfinance as yf
ticker = 'TSLA'
tkr = yf.Ticker(ticker)
df = tkr.history(period='5d')
```

Результат представлен в формате pandas DataFrame и выглядит примерно следующим образом (у вас фактические даты и возвращаемые данные будут отличаться):

	Open	High	Low	Close	Volume	Dividends	Stock Splits
Date							
2022-01-10	1000.00	1059.09	980.00	1058.11	30605000	0	0
2022-01-11	1053.67	1075.84	1038.81	1064.40	22021100	0	0
2022-01-12	1078.84	1114.83	1072.58	1106.21	27913000	0	0
2022-01-13	1109.06	1115.59	1026.54	1031.56	32403300	0	0
2022-01-14	1019.88	1052.00	1013.38	1049.60	24246600	0	0

Как видите, датафрейм индексируется по дате. Это означает, что данные представляют собой полноценный, хронологически упорядоченный временной ряд. Есть колонки с ценой открытия и закрытия, а также с самой высокой и низкой ценой за день. В свою очередь, колонка `Volume` показывает общее количество акций, участвующих в торгах в этот день, а две крайние правые колонки содержат подробную информацию о дивидендах и сплитах, которые компания выплатила своим акционерам.

Скорее всего, для анализа вам не понадобятся все эти столбцы. На самом деле сейчас вам нужен лишь столбец `Close`. Выведем его на экран в виде pandas Series:

```
print(df['Close'])
```

Серия будет выглядеть следующим образом:

```
Date
2022-01-10   1058.11
2022-01-11   1064.40
2022-01-12   1106.21
2022-01-13   1031.56
2022-01-14   1049.60
```

Теперь мы готовы приступить к анализу временных рядов. Разберем два самых распространенных метода: вычисление процентного изменения цены со временем и выполнение агрегированных расчетов в пределах скользящего временного окна. Вы увидите, как эти методы можно комбинировать для выявления тенденций в данных.

Вычисление процентных изменений

Пожалуй, наиболее типичный метод анализа временных рядов — отслеживание того, насколько сильно изменяются наблюдаемые данные с течением времени. Если речь идет о данных фондового рынка, этот метод анализа, вероятно, подразумевает расчет процентного изменения стоимости акций за определенный промежуток времени. Таким образом можно количественно оценить динамику стоимости акций и разработать краткосрочную инвестиционную стратегию.

С технической точки зрения процентное изменение — это разница (выраженная в процентах) между значениями, полученными в два разных момента времени, поэтому для расчета такого изменения необходимо иметь возможность сдвигать точки данных во времени. Это означает — сдвинуть имеющуюся точку данных вперед во времени так, чтобы она совпала с новой точкой данных, затем сравнить эти точки данных и рассчитать процентное изменение.

Когда временной ряд реализован в виде pandas Series или DataFrame, можно использовать метод `shift()` для сдвига точек данных во времени на нужное количество периодов. Вспомните пример с тикером TSLA. Вам, возможно, будет интересно, насколько изменилась цена закрытия за два дня. В этом случае необходимо использовать `shift(2)`, чтобы сдвинуть цену закрытия, которая была зафиксирована двумя днями ранее, вперед до цены закрытия на данный день. Чтобы понять, как работает сдвиг, сдвиньте столбец `Close` на два дня вперед, сохраните результат как серию `2DaysShift` и объедините ее с исходным столбцом `Close`:

```
print(pd.concat([df['Close'], df['Close'].shift(2)], axis=1, keys= ['Close',
'2DaysShift']))
```

Вы получите следующий вывод:

```
              Close   2DaysShift
Date
2022-01-10   1058.11          NaN
2022-01-11   1064.40          NaN
2022-01-12   1106.21      1058.11
2022-01-13   1031.56      1064.40
2022-01-14   1049.60      1106.21
```

Как видите, значения в колонке `Close` находят отражение в колонке `2DaysShift`, но со смещением в два дня. Первые два значения в колонке `2DaysShift` пустые (`NaN`), потому что цен за дни, предшествующие первым двум значениям временного ряда, у нас нет.

Чтобы найти процентное изменение цены за двухдневный период, можно взять разницу между стоимостью акции в данный день и ценой, которая была зафиксирована два дня назад, и разделить эту разницу на второе значение:

```
(df['Close'] - df['Close'].shift(2))/ df['Close'].shift(2)
```

Однако в финансовом анализе принято делить новое значение на старое и брать натуральный логарифм получившегося значения. Этот расчет обеспечивает почти точное приближение процентного изменения, когда оно находится в диапазоне от +/–5 до +/–20%. Во фрагменте кода ниже мы рассчитываем двухдневную разницу в процентах, используя натуральный логарифм, и сохраняем результат в виде нового столбца `2daysRise` в датафрейме `df`:

```
import numpy as np
df['2daysRise'] = np.log(df['Close'] / df['Close'].shift(2))
```

Мы получаем цену закрытия в текущий день и делим ее на цену закрытия двумя биржевыми днями ранее, используя `shift(2)`. Затем используем функцию NumPy `log()`, чтобы вычислить натуральный логарифм полученного значения. Теперь выводим на экран колонки `Close` и `2daysRise`:

```
print(df[['Close','2daysRise']])
```

Полученный временной ряд:

```
              Close   2daysRise
Date
2022-01-10   1058.11         NaN
2022-01-11   1064.40         NaN
2022-01-12   1106.21    0.044455
2022-01-13   1031.56   -0.031339
2022-01-14   1049.60   -0.052530
```

Колонка `2daysRise` показывает процентное изменение цены акции по сравнению с ее стоимостью двумя днями ранее. И снова первые два значения

в столбце — NaN, потому что цен за два предыдущих дня для первых двух значений временного ряда у нас нет.

Вычисление скользящего окна

Другой популярный метод анализа временных рядов — сравнение каждого значения со средним значением за *n* периодов. Это называется *вычислением скользящего окна (rolling window calculation)*: вы создаете временное окно фиксированного размера и выполняете агрегированный расчет значений в пределах этого окна по мере его перемещения, или *скольжения*, по временному ряду. В случае с акциями можно использовать скользящее окно, чтобы найти среднюю цену закрытия двух предыдущих дней, а затем сравнить цену закрытия текущего дня с этим значением. Это даст представление о стабильности цены акций с течением времени.

Каждый объект pandas имеет метод `rolling()` для просмотра скользящего окна значений. Используем его в сочетании с `shift()` и `mean()` для нахождения средней цены акций Tesla за предыдущие два дня:

```
df['2daysAvg'] = df['Close'].shift(1).rolling(2).mean()
print(df[['Close', '2daysAvg']])
```

В первой строке применяем `shift(1)` для сдвига точек данных в серии на один день назад, поскольку не хотим включать цену текущего дня в расчет среднего значения. Далее формируем скользящее окно с помощью функции `rolling(2)`, которая для выполнения вычислений берет две последовательные строки. Наконец, вызываем метод `mean()` для расчета среднего значения для каждой пары последовательных строк в скользящем окне. Сохраняем результаты в новом столбце `2daysAvg`, который выводим на экран вместе со столбцом `Close`. Итоговый датафрейм будет выглядеть так:

```
              Close   2daysAvg
Date
2022-01-10   1058.11       NaN
2022-01-11   1064.40       NaN
2022-01-12   1106.21   1061.26
2022-01-13   1031.56   1085.30
2022-01-14   1049.60   1068.89
```

Цены в колонке `2daysAvg` являются средними значениями двух предыдущих биржевых дней. Например, значение в строке с датой `2022-01-12` является средним значением цен за `2022-01-10` и `2022-01-11`.

Вычисление процентного изменения скользящего среднего

Следующий логический шаг — расчет разницы в процентах между ценой каждого текущего дня и соответствующей средней ценой двух предшествующих дней, вычисленной с помощью скользящего окна. Выполняем этот расчет, снова используя натуральный логарифм для приближения процентного изменения:

```
df['2daysAvgRise'] = np.log(df['Close'] / df['2daysAvg'])
print(df[['Close','2daysRise','2daysAvgRise']])
```

Сохраняем результаты в новом столбце под названием `2daysAvgRise`. Затем выводим на экран столбцы `Close`, `2daysRise` и `2daysAvgRise` вместе. Вывод будет приблизительно таким:

```
              Close   2daysRise   2daysAvgRise
Date
2022-01-10   1058.11         NaN            NaN
2022-01-11   1064.40         NaN            NaN
2022-01-12   1106.21    0.044455       0.041492
2022-01-13   1031.56   -0.031339      -0.050793
2022-01-14   1049.60   -0.052530      -0.018202
```

Для этого конкретного временного ряда обе вновь созданные метрики, `2daysRise` и `2daysAvgRise`, содержат как отрицательные, так и положительные значения. Это указывает на то, что цена закрытия акции была волатильной в течение всего периода наблюдения. Конечно, полученные вами результаты могут показать другую тенденцию.

Многомерные временные ряды

Многомерный временной ряд — это ряд с более чем одной переменной, изменяющейся во времени. Например, когда мы впервые получили данные о стоимости акции Tesla через библиотеку yfinance, они были представлены в виде многомерного временного ряда, поскольку включали не только цену закрытия на определенный день, но и цену открытия, самую высокую и самую низкую цену дня, а также несколько других точек данных. В данном случае многомерный временной ряд отслеживает несколько характеристик одного и того же объекта — отдельной акции. Кроме того, многомерные временные ряды могут отслеживать одну и ту же характеристику нескольких различных объектов, например цену закрытия нескольких акций за один период времени.

В следующем фрагменте кода мы создаем именно такой тип многомерного временного ряда, получая биржевые данные для нескольких тикеров за пять дней:

```
   import pandas as pd
   import yfinance as yf
❶  stocks = pd.DataFrame()
❷  tickers = ['MSFT','TSLA','GM','AAPL','ORCL','AMZN']
❸  for ticker in tickers:
     tkr = yf.Ticker(ticker)
     hist = tkr.history(period='5d')
   ❹ hist = pd.DataFrame(hist[['Close']].rename(columns={'Close': ticker}))
   ❺ if stocks.empty:
      ❻ stocks = hist
      else:
      ❼ stocks = stocks.join(hist)
```

Сначала мы определяем датафрейм `stocks` ❶, где будем хранить цены закрытия для нескольких тикеров. Затем определяем список тикеров ❷ и проходим по нему ❸, используя в теле цикла библиотеку yfinance для получения данных по каждому тикеру за последние пять дней. В цикле мы сокращаем датафрейм `hist`, полученный yfinance, до одного столбца с ценами закрытия данной акции и соответствующими метками времени в качестве индекса датафрейма ❹. Затем проверяем, пуст ли датафрейм `stocks` ❺. Если он пуст, это означает, что цикл первый, и мы инициализируем датафрейм `stocks` с помощью датафрейма `hist` ❻. На последующих итерациях `stocks` уже не будет пустым, поэтому мы присоединяем текущий датафрейм `hist` к `stocks`, добавляя цены закрытия другого тикера ❼. Структура `if/else` необходима, потому что мы не можем выполнить операцию объединения на пустом датафрейме.

Итоговый датафрейм `stocks`:

```
              MSFT      TSLA      GM      AAPL    ORCL      AMZN
Date
2022-01-10   314.26   1058.11   61.07   172.19   89.27   3229.71
2022-01-11   314.98   1064.40   61.45   175.08   88.48   3307.23
2022-01-12   318.26   1106.21   61.02   175.52   88.30   3304.13
2022-01-13   304.79   1031.56   61.77   172.19   87.79   3224.28
2022-01-14   310.20   1049.60   61.09   173.07   87.69   3242.76
```

Мы получили многомерный временной ряд. В столбцах этого набора данных отображены цены закрытия различных акций, в строках — единый промежуток времени.

Обработка многомерных временных рядов

Обработка многомерных временных рядов напоминает работу с одномерными временными рядами, за исключением того, что в первом случае придется иметь дело с несколькими переменными в каждой строке. Поэтому вычисления часто производятся внутри цикла, перебирающего столбцы серии. Предположим, что требуется отфильтровать датафрейм `stocks`, удалив тикеры, цены которых упали более чем на 3% по сравнению с ценой предыдущего дня хотя бы один раз за указанный период. Во фрагменте кода ниже мы перебираем столбцы и анализируем данные по каждому тикеру, чтобы определить, какие акции следует сохранить в датафрейме:

```
❶ stocks_to_keep = []
❷ for i in stocks.columns:
    if stocks[stocks[i]/stocks[i].shift(1)< .97].empty:
    stocks_to_keep.append(i)
  print(stocks_to_keep)
```

Сначала мы создаем список для хранения названий нужных столбцов ❶. Затем проходим по столбцам датафрейма `stocks` ❷, определяя, содержит ли столбец значения, которые более чем на 3% ниже, чем значение в предыдущей строке. В частности, мы используем оператор `[]` для фильтрации датафрейма и метод `shift()` для сравнения цены закрытия текущего дня с ценой закрытия предыдущего дня. Если столбец не содержит значений, удовлетворяющих условию фильтрации (то есть отфильтрованный столбец пуст), мы добавляем имя этого столбца в список `stocks_to_keep`.

Исходя из данных датафрейма `stocks`, полученный список `stocks_to_keep` будет выглядеть следующим образом:

```
['GM', 'AAPL', 'ORCL', 'AMZN']
```

Как видите, TSLA и MSFT отсутствуют в списке, потому что они содержат одно или несколько значений, которые более чем на 3% ниже цены закрытия в предыдущий день. Конечно, ваши результаты будут отличаться; ваш список может оказаться пустым или содержать сразу все тикеры. В этих случаях попробуйте поэкспериментировать с пороговым значением фильтрации. Если список пуст, попробуйте снизить порог с 0.97 до 0.96 или ниже. А если список, напротив, включает все тикеры, попробуйте увеличить порог.

Мы выводим на экран датафрейм `stocks`, фильтруя его таким образом, чтобы он включал только столбцы из списка `stocks_to_keep`:

```
print(stocks[stocks_to_keep])
```

В моем случае вывод выглядит так:

```
                GM      AAPL    ORCL       AMZN
Date
2022-01-10   61.07   172.19   89.27   3229.71
2022-01-11   61.45   175.08   88.48   3307.23
2022-01-12   61.02   175.52   88.30   3304.13
2022-01-13   61.77   172.19   87.79   3224.28
2022-01-14   61.09   173.07   87.69   3242.76
```

Как и ожидалось, столбцы `TSLA` и `MSFT` удалены, поскольку они содержат одно или несколько значений, превышающих 3-процентный порог волатильности.

Анализ зависимости между переменными

Одной из распространенных задач при анализе многомерных временных рядов является выявление взаимосвязей между переменными. Эта взаимосвязь может присутствовать, но не обязательно. Например, скорее всего, существует определенная зависимость между ценами открытия и закрытия акций, поскольку цена закрытия на конкретный день редко отличается от цены открытия более чем на несколько процентов. А зависимости между ценами закрытия двух акций из разных секторов экономики, наоборот, может не быть.

В этом разделе мы рассмотрим некоторые методы проверки существования взаимосвязи между переменными временного ряда. Для примера проверим, существует ли зависимость между изменением цены акции и объемом ее продаж. Для начала запустим следующий скрипт, чтобы получить данные об акциях за один месяц:

```
import yfinance as yf
import numpy as np
ticker = 'TSLA'
tkr = yf.Ticker(ticker)
df = tkr.history(period='1mo')
```

Как вы уже видели, yfinance создает многомерный временной ряд в виде датафрейма с несколькими столбцами. Для данного примера нам понадобятся только два из них: `Close` и `Volume`. Сокращаем датафрейм до нужных колонок и изменяем название столбца `Close` на `Price`:

```
df = df[['Close','Volume']].rename(columns={'Close': 'Price'})
```

Чтобы определить, существует ли взаимосвязь между столбцами `Price` и `Volume`, необходимо рассчитать процентное изменение в каждом столбце ото дня ко дню. Рассчитаем ежедневное изменение в столбце `Price` в процентах, используя `shift(1)` и функцию NumPy `log()`, как мы делали ранее, и сохраним результат в новом столбце `priceRise`:

```
df['priceRise'] = np.log(df['Price'] / df['Price'].shift(1))
```

Используем ту же технику для создания столбца `volumeRise`, который будет отображать процентное изменение объема продаж по сравнению с предыдущим днем:

```
df['volumeRise'] = np.log(df['Volume'] / df['Volume'].shift(1))
```

Как уже отмечалось, натуральный логарифм обеспечивает хорошее приближение процентного изменения в пределах +/–20%. Хотя некоторые значения в колонке `volumeRise` вполне могут выйти за пределы этого диапазона, мы все равно будем использовать `log()`, потому что высокая степень точности в этом примере не требуется; цель анализа фондового рынка — предсказание тенденций, а не поиск точных значений.

Если теперь вывести датафрейм `df` на экран, то получим следующее:

```
              Price      Volume   priceRise   volumeRise
Date
2021-12-15   975.98   25056400         NaN          NaN
2021-12-16   926.91   27590500   -0.051585     0.096342
2021-12-17   932.57   33479100    0.006077     0.193450
2021-12-20   899.94   18826700   -0.035616    -0.575645
2021-12-21   938.53   23839300    0.041987     0.236059
2021-12-22  1008.86   31211400    0.072271     0.269448
2021-12-23  1067.00   30904400    0.056020    -0.009885
2021-12-27  1093.93   23715300    0.024935    -0.264778
2021-12-28  1088.46   20108000   -0.005013    -0.165003
2021-12-29  1086.18   18718000   -0.002097    -0.071632
2021-12-30  1070.33   15680300   -0.014700    -0.177080
2021-12-31  1056.78   13528700   -0.012750    -0.147592
2022-01-03  1199.78   34643800    0.126912     0.940305
2022-01-04  1149.58   33416100   -0.042733    -0.036081
2022-01-05  1088.11   26706600   -0.054954    -0.224127
2022-01-06  1064.69   30112200   -0.021758     0.120020
2022-01-07  1026.95   27919000   -0.036090    -0.075623
2022-01-10  1058.11   30605000    0.029891     0.091856
2022-01-11  1064.40   22021100    0.005918    -0.329162
```

```
2022-01-12  1106.21   27913000    0.038537     0.237091
2022-01-13  1031.56   32403300   -0.069876     0.149168
2022-01-14  1049.60   24246600    0.017346    -0.289984
```

Если существует зависимость между ценой и объемом продаж, можно ожидать, что изменение цены выше среднего (то есть рост волатильности) будет коррелировать с изменением объема выше среднего. Чтобы проверить, так ли это, необходимо установить некоторый порог для столбца `priceRise` и просматривать только те строки, в которых процентное изменение цены этот порог превышает. Например, для значений в столбце `priceRise` данного вывода можно выбрать 5-процентный порог. Работая с другим датасетом, можно выбрать иной порог, например 3 или 7 %. Идея в том, что только несколько записей должны выходить за рамки порогового значения, поэтому, как правило, чем более волатильны акции, тем выше должен быть порог.

Выводим только те строки, в которых `priceRise` превышает порог:

```
print(df[abs(df['priceRise']) > .05])
```

Мы используем функцию `abs()` для получения абсолютного значения изменения, так, чтобы и `0.06`, и `-0.06` удовлетворяли указанному условию. С учетом наших данных получаем:

```
              Price     Volume   priceRise   volumeRise
Date
2021-12-16    926.91   27590500   -0.051585    0.096342
2021-12-22   1008.86   31211400    0.072271    0.269448
2021-12-23   1067.00   30904400    0.056020   -0.009885
2022-01-03   1199.78   34643800    0.126912    0.940305
2022-01-05   1088.11   26706600   -0.054954   -0.224127
2022-01-13   1031.56   32403300   -0.069876    0.149168
```

Далее вычисляем среднее изменение объема продаж по всей серии:

```
print(df['volumeRise'].mean().round(4))
```

Для данной конкретной серии получим:

```
-0.0016
```

Наконец, вычисляем среднее изменение объема продаж только для тех строк, в которых изменение цены выше среднего. Если в результате получим значение,

превышающее среднее изменение объема продаж по всей серии, значит, между повышенной волатильностью и ростом продаж существует связь:

```
print(df[abs(df['priceRise']) > .05]['volumeRise'].mean().round(4))
```

Для нашей серии получаем следующее:

```
0.2035
```

Как видите, среднее изменение объема продаж в отфильтрованной серии намного выше, чем этот же показатель, рассчитанный для всей серии. Это говорит о том, что между волатильностью цен и волатильностью объема продаж, вероятно, существует положительная корреляция.

УПРАЖНЕНИЕ № 17: ВВЕДЕНИЕ ДОПОЛНИТЕЛЬНЫХ МЕТРИК ДЛЯ АНАЛИЗА ЗАВИСИМОСТЕЙ

Продолжая рассматривать датафрейм из предыдущего раздела, можно заметить, что хотя между столбцами `priceRise` и `volumeRise`, скорее всего, есть связь, она не так однозначна. Например, 2022-12-16 цена упала примерно на 5%, а продажи выросли на 10%, но почти такое же снижение цены 2022-01-05 сопровождалось 22-процентным падением продаж.

Чтобы разобраться в этих расхождениях, необходимо изучить другие показатели, которые могут коррелировать с объемом продаж. Например, интересно рассчитать величину скользящего окна общих продаж за два предыдущих дня. Исходим из того, что если продажи за конкретный день превышают (или почти равны) сумме продаж за два предшествующих дня, то скорее всего, не стоит ожидать роста продаж на следующий день. Таким образом, скользящее окно помогает предсказать тенденции в продажах.

Проверьте, верно ли это предположение. Прежде чем выполнять расчет с помощью скользящего окна, добавьте в датафрейм столбец `volumeSum`:

```
df['volumeSum'] =
df['Volume'].shift(1).rolling(2).sum().fillna(0).astype(int)
```

С помощью строки кода выше мы сдвигаем точки данных на один день, чтобы не брать в расчет продажи текущего дня. Затем создаем скользящее

окно, включающее данные двух дней, и с помощью `sum()` вычисляем общий объем продаж в этом окне. По умолчанию значения в новом столбце будут иметь тип float, но их можно преобразовать в целые числа (int) с помощью функции `astype()`. Прежде чем выполнять это преобразование, необходимо заменить значения `NaN` нулями. Сделать это можно с помощью метода `fillna()`.

Теперь, имея метрику `volumeSum`, снова посмотрим на самые волатильные дни в серии:

```
print(df[abs(df['priceRise']) > .05].replace(0, np.nan).dropna())
```

Из нашего примера данных снова выводим дни, когда цена изменилась более чем на 5 % по сравнению с предыдущим днем, но теперь вместе с колонкой `volumeSum`:

```
              Price     Volume   priceRise  volumeRise  volumeSum
Date
2021-12-22   1008.86   31211400    0.072271    0.269448   42666000
2021-12-23   1067.00   30904400    0.056020   -0.009885   55050700
2022-01-03   1199.78   34643800    0.126912    0.940305   29209000
2022-01-05   1088.11   26706600   -0.054954   -0.224127   68059900
2022-01-13   1031.56   32403300   -0.069876    0.149168   49934100
```

Значения в колонке `volumeSum` говорят о том, что более низкий общий объем продаж за два предшествующих дня коррелирует с более высоким потенциалом роста или снижения продаж в текущий день, и наоборот. Посмотрите, например, на данные за 2022-01-03: значение в колонке `volumeRise` на эту дату является самым высоким во всем наборе данных, а значение `volumeSum` — самым низким. Фактически объем продаж за этот день почти равен сумме продаж за два предыдущих дня (2021-12-30 и 2021-12-31), демонстрируя тем самым значительный рост.

Однако напомним, что изначально предполагалось, что в дни, когда продажи превышают (или примерно соответствуют) сумме продаж за последние два дня, не стоит ожидать роста продаж на следующий день. Чтобы убедиться в этом, добавим колонку, показывающую объем продаж за следующий день:

```
df['nextVolume'] = df['Volume'].shift(-1).fillna(0).astype(int)
print(df[abs(df['priceRise']) > .05].replace(0, np.nan).dropna()
```

Создаем колонку `nextVolume`, сдвигая `Volume` на `-1`, то есть сдвигаем объем продаж за следующий день назад так, чтобы он совпал с текущим. Получим вывод:

```
              Price    Volume  priceRise  volumeRise  volumeSum  nextVolume
Date
2021-12-22  1008.86  31211400   0.072271    0.269448   42666000    30904400
2021-12-23  1067.00  30904400   0.056020   -0.009885   55050700    23715300
2022-01-03  1199.78  34643800   0.126912    0.940305   29209000    33416100
2022-01-05  1088.11  26706600  -0.054954   -0.224127   68059900    30112200
2022-01-13  1031.56  32403300  -0.069876    0.149168   49934100    24246600
```

Как видите, предположение верно для 2022-01-03: `nextVolume` меньше, чем `Volume`. Однако для точного анализа может понадобиться больше показателей. Попробуйте добавить еще одну метрику, которая суммирует значения `priceRise` за два предыдущих дня. Если значение положительное, это означает, что цены в целом находились в восходящем тренде в течение последних двух дней. Отрицательное же значение указывает на падение цен. Используйте эту новую метрику вместе с уже существующими метриками `priceRise` и `volumeSum`, чтобы выяснить, как они совместно влияют на значения в колонке `volumeRise`.

Выводы

Как вы узнали из этой главы, временной ряд — это набор данных, организованный в хронологическом порядке, в котором одна или несколько переменных изменяются во времени. На примере данных фондового рынка мы рассмотрели несколько методов применения pandas для анализа временных рядов с целью получения из них полезной статистики. Вы научились смещать точки данных во временном ряду, чтобы рассчитать изменения с течением времени, а также узнали, как выполнять вычисления в скользящем окне, то есть как агрегировать данные в пределах перемещающегося по всей серии временного интервала фиксированного размера. В совокупности эти методы помогают делать выводы о тенденциях в данных. Наконец, мы рассмотрели методы выявления зависимостей между различными переменными в многомерном временном ряду.

11

Получение инсайтов из данных

Ежедневно компании генерируют огромное количество данных в виде необработанных фактов, показателей и событий, но о чем вся эта информация говорит на самом деле? Чтобы извлечь ценные сведения и инсайты из данных, их необходимо преобразовать, проанализировать и представить в наглядной форме. Другими словами, нужно превратить необработанные данные в значимую информацию, которую можно использовать для принятия решений, ответов на вопросы и выполнения поставленных задач.

Рассмотрим сценарий с супермаркетом, который собирает большие объемы данных о покупках клиентов. Эти данные представляют интерес для аналитиков супермаркета, поскольку с их помощью можно получить представление о предпочтениях покупателей. В частности, для этого служит *анализ потребительской корзины* (market basket analysis) — метод интеллектуального анализа данных, который исследует совершенные транзакции и выявляет товары, которые обычно приобретаются вместе. Вооружившись этими знаниями, супермаркет сможет принимать более обоснованные бизнес-решения, например, о выкладке товаров в магазине или о том, как объединять товары в группы со скидками.

В этой главе мы подробно рассмотрим, как получить инсайты из данных о транзакциях с помощью анализа потребительской корзины на Python. Мы расскажем, как использовать библиотеку mlxtend и алгоритм Apriori для определения товаров, которые обычно приобретаются вместе, и как применять эти знания для принятия обоснованных бизнес-решений.

Хотя в центре внимания данной главы будет выявление предпочтений покупателей, анализ потребительской корзины можно применять не только с этой целью. Та же техника используется в таких областях, как телекоммуникации, анализ использования веб-ресурсов (web usage mining), банковское дело и здравоохранение. Например, при исследовании использования веб-ресурсов с помощью анализа потребительской корзины можно определить, на какую страницу пользователь, скорее всего, перейдет дальше, и создать ассоциации часто посещаемых страниц.

Ассоциативные правила

Анализ потребительской корзины — это измерение степени взаимосвязи между объектами на основе вероятности их совместного присутствия в одних и тех же транзакциях. Взаимосвязи между объектами представлены в виде *ассоциативных правил*, которые обозначаются следующим образом:

```
X->Y
```

`X` и `Y`, называемые *антецедентом* (antecedent) и *консеквентом* (consequent) правила соответственно, представляют собой отдельные *наборы товаров*, или группы из одного либо нескольких товаров, полученных из данных о транзакции. Например, ассоциативное правило, описывающее связь между товарами *творог* и *сметана*, будет таким:

```
творог -> сметана
```

В данном случае `творог` является антецедентом, а `сметана` — консеквентом. Правило утверждает, что люди, покупающие творог, скорее всего, купят и сметану.

Само по себе ассоциативное правило, подобное этому, на самом деле не очень информативно. Ключом к успешному анализу потребительской корзины является использование данных о транзакциях для оценки степени значимости ассоциативных правил на основе различных метрик. Возьмем простой пример. Предположим, у нас есть данные о 100 покупательских транзакциях, 25 из которых содержат творог и 30 — сметану. Среди 30 транзакций, содержащих сметану, 20 также содержат творог. В табл. 11.1 представлены эти показатели.

Учитывая эти данные, можно оценить значимость ассоциативного правила `творог -> сметана`, используя такие метрики, как поддержка (support), доверие (confidence) и лифт (lift). Эти метрики помогут определить, действительно ли существует связь между творогом и сметаной.

Таблица 11.1. Данные транзакций с творогом и сметаной

	Творог	Сметана	Творог и сметана	Общее количество
Транзакции	25	30	20	100

Поддержка

Поддержка (support) — это отношение количества транзакций, включающих один или более товаров, к общему количеству транзакций. Например, показатель поддержки творога в наших данных о сделке может быть рассчитан следующим образом:

```
поддержка(творог) = творог / общее количество = 25 / 100 = 0.25
```

В контексте ассоциативного правила поддержка — это отношение количества транзакций, включающих и антецедент, и консеквент, к общему количеству транзакций. Таким образом, поддержка ассоциативного правила `творог -> сметана` будет равна:

```
поддержка(творог -> сметана) = (творог & сметана) / общее количество =
20 / 100 = 0.2
```

Метрика поддержки имеет значение в диапазоне от 0 до 1 и говорит о том, в каком проценте случаев набор товаров появляется в транзакции вместе. В данном примере мы видим, что в 20% транзакций есть и творог, и сметана. Поддержка симметрична для любого ассоциативного правила, то есть поддержка для `творог -> сметана` такая же, как для `сметана -> творог`.

Доверие

Доверие (confidence) ассоциативного правила — это отношение транзакций, в которых есть и антецедент, и консеквент, к транзакциям, в которых присутствует только антецедент. Другими словами, доверие измеряет, какая доля транзакций, содержащих антецедент, также содержит консеквент. Доверие для ассоциативного правила `творог -> сметана` можно рассчитать следующим образом:

```
доверие(творог -> сметана) = (творог & сметана) / творог = 20 / 25 = 0.8
```

Этот показатель можно интерпретировать так: если клиент купил творог, то вероятность того, что он также купит сметану, составляет 80%.

Как и поддержка, доверие находится в диапазоне от 0 до 1, но, в отличие от поддержки, оно не симметрично. Это означает, что метрика доверия для правила `творог -> сметана` может отличаться от доверия для правила `сметана -> творог`:

```
доверие(сметана -> творог) = (творог & сметана) / сметана = 20 / 30 = 0.66
```

В данном сценарии значение доверия будет меньше, если антецедент и консеквент ассоциативного правила поменяются местами. Это говорит о том, что вероятность того, что человек, покупающий сметану, купит и творог, меньше, чем вероятность того, что человек, покупающий творог, купит и сметану.

Лифт

Лифт (lift) оценивает значимость ассоциативного правила для случая, когда элементы правила оказываются в одной транзакции случайно. Лифт ассоциативного правила `творог -> сметана` — это отношение наблюдаемой поддержки для `творог -> сметана` к ожидаемой, если бы покупка творога и покупка сметаны были независимы друг от друга. Рассчитать лифт можно следующим образом:

```
лифт(сметана -> творог) = поддержка(творог & сметана) / (поддержка (творог) *
поддержка(сметана)) = 0.2 / (0.25 * 0.3) = 2.66
```

Метрика лифта симметрична — если поменять местами антецедент и консеквент, значение метрики не изменится. Коэффициент лифта варьируется от 0 до бесконечности, и чем больше этот коэффициент, тем сильнее связь. В частности, коэффициент лифта, больший 1, указывает на то, что связь между антецедентом и консеквентом сильнее, чем можно было бы ожидать, если бы они были независимыми, то есть эти два товара часто покупают вместе. Коэффициент лифта, равный 1, указывает на отсутствие корреляции между антецедентом и консеквентом. Коэффициент лифта, меньший 1, говорит о наличии отрицательной корреляции между антецедентом и консеквентом. Это означает, что их вряд ли купят вместе. В данном случае коэффициент лифта 2.66 можно интерпретировать так: когда клиент покупает творог, ожидаемая вероятность того, что он также купит сметану, увеличивается на 166%.

Алгоритм Apriori

Теперь вы знаете, что собой представляют ассоциативные правила и некоторые метрики оценки их значимости, но как создавать ассоциативные правила для

анализа потребительской корзины? Один из способов — использовать *алгоритм Apriori* (Apriori algorithm), автоматизированный процесс анализа данных о транзакциях. В общих чертах этот алгоритм состоит из двух шагов:

1. Определение *всех часто встречающихся наборов* или группы из одного либо нескольких товаров, которые в рамках датасета присутствуют сразу во многих транзакциях. Алгоритм находит все товары или группы товаров, значение поддержки которых превышает определенный порог.
2. Генерирование ассоциативных правил для часто встречающихся наборов товаров путем рассмотрения всех возможных бинарных разбиений каждого набора товаров (то есть всех разбиений набора на группу антецедентов и группу консеквентов) и вычисления метрик ассоциативных правил для каждого разбиения.

После создания ассоциативных правил их значимость можно оценить с помощью метрик из предыдущего раздела.

Несколько сторонних библиотек Python поставляются с реализацией алгоритма Apriori. Одна из них — библиотека mlxtend (сокращение от *machine learning extensions*). Библиотека mlxtend включает инструменты для решения ряда общих задач в области data science. В этом разделе мы рассмотрим пример анализа потребительской корзины с помощью реализации алгоритма Apriori. Но сначала установим mlxtend с помощью `pip`:

```
$ pip install mlxtend
```

ПРИМЕЧАНИЕ

Чтобы узнать больше о mlxtend, обратитесь к документации библиотеки[1].

Создание датасета с транзакциями

Для проведения анализа потребительской корзины понадобятся данные о нескольких транзакциях. Для простоты можно использовать всего несколько транзакций, реализованных в виде списка списков, как показано ниже:

```
transactions = [
 ['curd', 'sour cream'], ['curd', 'orange', 'sour cream'],
 ['bread', 'cheese', 'butter'], ['bread', 'butter'], ['bread', 'milk'],
 ['apple', 'orange', 'pear'], ['bread', 'milk', 'eggs'], ['tea', 'lemon'],
```

[1] http://rasbt.github.io/mlxtend

```
 ['curd', 'sour cream', 'apple'], ['eggs', 'wheat flour', 'milk'],
 ['pasta', 'cheese'], ['bread', 'cheese'], ['pasta', 'olive oil', 'cheese'],
 ['curd', 'jam'], ['bread', 'cheese', 'butter'],
 ['bread', 'sour cream', 'butter'], ['strawberry', 'sour cream'],
 ['curd', 'sour cream'], ['bread', 'coffee'], ['onion', 'garlic']
] [1]
```

Каждый внутренний список содержит набор товаров одной транзакции. Внешний список `transactions` содержит в общей сложности 20 транзакций. Для сохранения количественных пропорций, определенных в исходном примере творог/сметана, датасет содержит пять операций с творогом (curd), шесть операций со сметаной (sour cream) и четыре операции, содержащие и творог, и сметану.

Чтобы пропустить данные транзакций через алгоритм Apriori библиотеки mlxtend, необходимо преобразовать их в *булев массив*, созданный с помощью *быстрого кодирования* (OHE, one-hot encoding), то есть структуру, где каждый столбец представляет доступный товар, а строка — транзакцию. Значения массива могут быть равны либо `True`, либо `False` (`True`, если транзакция включала данный конкретный товар, и `False` — если нет). Во фрагменте кода ниже мы выполняем необходимое преобразование, используя объект mlxtend `TransactionEncoder`:

```
   import pandas as pd
   from mlxtend.preprocessing import TransactionEncoder

❶ encoder = TransactionEncoder()
❷ encoded_array = encoder.fit(transactions).transform(transactions)
❸ df_itemsets = pd.DataFrame(encoded_array, columns=encoder.columns_)
```

Мы создаем объект `TransactionEncoder` ❶ и используем его для преобразования списка списков `transactions` в булев OHE-массив с названием `encoded_array` ❷. Затем преобразуем массив в pandas DataFrame `df_itemsets` ❸, фрагмент которого приведен ниже:

```
    apple   bread   butter   cheese   coffee    curd    eggs   ...
0   False   False   False     False    False    True   False   ...
1   False   False   False     False    False    True   False   ...
2   False    True    True      True    False   False   False   ...
3   False    True    True     False    False   False   False   ...
4   False    True   False     False    False   False   False   ...
```

[1] curd — творог, sour cream — сметана, orange — апельсин, bread — хлеб, cheese — сыр, butter — масло, milk — молоко, apple — яблоко, pear — груша, eggs — яйца, tea — чай, lemon — лимон, wheat flour — пшеничная мука, pasta — макароны, olive oil — оливковое масло, jam — джем, strawberry — клубника, coffee — кофе, onion — лук, garlic — чеснок. — *Примеч. пер.*

```
5    True   False   False     False    False   False   False   ...
6   False    True   False     False    False   False    True   ...
--фрагмент--

[20 rows x 20 columns]
```

Датафрейм состоит из 20 строк и 20 столбцов. Строки представляют собой транзакции, а столбцы — товары. Чтобы проверить, что в исходном списке списков действительно было 20 транзакций, основанных на 20 доступных товарах, используйте следующий код:

```
print('Number of transactions: ', len(transactions))
print('Number of unique items: ', len(set(sum(transactions, []))))
```

В обоих случаях должно получиться `20`.

Определение часто встречающихся наборов

Теперь, когда данные транзакции представлены в удобном формате, можно использовать функцию mlxtend `apriori()` для определения всех часто встречающихся наборов товаров в данных о транзакциях, то есть всех товаров или групп товаров с достаточно высокой метрикой поддержки. Вот как это реализовать:

```
from mlxtend.frequent_patterns import apriori
frequent_itemsets = apriori(df_itemsets, min_support=0.1, use_colnames=True)
```

Из модуля `mlxtend.frequent_patterns` импортируем функцию `apriori()`. Затем вызываем ее, передавая датафрейм с данными о транзакциях в качестве первого параметра. Также устанавливаем для параметра `min_support` значение `0.1`, чтобы возвращать наборы с поддержкой не менее 10% (помните, что метрика поддержки показывает, в каком проценте транзакций встречается товар или группа товаров). Для `use_colnames` мы устанавливаем значение `True`, чтобы определить столбцы, включаемые в каждый набор товаров, по их названию (например, `curd` или `sour cream`), а не по индексу. В результате `apriori()` возвращает следующий датафрейм:

```
    support                itemsets
0      0.10                 (apple)
1      0.40                 (bread)
2      0.20                (butter)
3      0.25                (cheese)
4      0.25                  (curd)
5      0.10                  (eggs)
```

```
6      0.15                    (milk)
7      0.10                  (orange)
8      0.10                   (pasta)
9      0.30              (sour cream)
10     0.20           (bread, butter)
11     0.15           (bread, cheese)
12     0.10             (bread, milk)
13     0.10          (cheese, butter)
14     0.10           (pasta, cheese)
15     0.20        (sour cream, curd)
16     0.10              (milk, eggs)
17     0.10   (bread, cheese, butter)
```

Как уже отмечалось, набор товаров может состоять из одного или нескольких позиций, и действительно, `apriori()` вернул несколько наборов с одним товаром. В конечном итоге mlxtend не будет учитывать наборы с одним товаром при составлении ассоциативных правил; тем не менее ему понадобятся данные обо *всех* часто встречающихся наборах (включая те, которые содержат один товар). Ради интереса можете выбрать только те наборы товаров, которые содержат несколько позиций. Для этого добавьте колонку `length` к датафрейму `frequent_itemsets`:

```
frequent_itemsets['length'] = frequent_itemsets['itemsets'].apply(lambda itemset:
len(itemset))
```

Затем, используя синтаксис pandas, отфильтруйте датафрейм так, чтобы остались только те строки, значение поля `length` которых равно или больше `2`:

```
print(frequent_itemsets[frequent_itemsets['length'] >= 2])
```

Вы получите датафрейм, который не содержит наборов с одним товаром:

```
10   0.20         (bread,   butter)   2
11   0.15         (bread,   cheese)   2
12   0.10             (bread, milk)   2
13   0.10          (cheese, butter)   2
14   0.10           (pasta, cheese)   2
15   0.20        (sour cream, curd)   2
16   0.10              (milk, eggs)   2
17   0.10   (bread, cheese, butter)   3
```

Повторимся, однако, что для генерации ассоциативных правил mlxtend требует информацию обо всех часто встречающихся наборах товаров. Поэтому убедитесь, что вы не удаляете ни одной строки из исходного датафрейма `frequent_itemsets`.

Генерирование ассоциативных правил

Мы определили все наборы товаров, которые соответствуют желаемому пороговому значению метрики поддержки. Второй шаг алгоритма Apriori заключается в генерации ассоциативных правил для этих наборов. Для этого используется функция `association_rules()` из модуля mlxtend `frequent_patterns`:

```
from mlxtend.frequent_patterns import association_rules
rules = association_rules(frequent_itemsets, metric="confidence",
min_threshold=0.5)
```

Во фрагменте кода выше мы вызываем функцию `association_rules()`, передавая в нее датафрейм `frequent_itemsets` в качестве первого параметра. Кроме того, выбираем метрику для оценки значимости правил и устанавливаем ее пороговое значение. В нашем конкретном случае мы указываем, что функция должна возвращать только те ассоциативные правила, метрика доверия которых равна или больше 0.5. Как отмечалось в предыдущем разделе, функция автоматически пропускает генерацию правил для наборов с одним товаром.

Функция `association_rules()` возвращает правила в виде датафрейма, где каждая строка представляет одно ассоциативное правило. В датафрейме несколько колонок: для антецедентов, консеквентов и различных метрик, включая поддержку, доверие и лифт. Выводим на экран выбранные столбцы:

```
print(rules.iloc[:,0:7])
```

Вот что мы получаем:

```
         antecedents    consequents  antecedent sup.  consequent sup.  support  confidence     lift
0            (bread)       (butter)             0.40             0.20     0.20    0.500000   2.500000
1           (butter)        (bread)             0.20             0.40     0.20    1.000000   2.500000
2           (cheese)        (bread)             0.25             0.40     0.15    0.600000   1.500000
3             (milk)        (bread)             0.15             0.40     0.10    0.666667   1.666667
4           (butter)       (cheese)             0.20             0.25     0.10    0.500000   2.000000
5            (pasta)       (cheese)             0.10             0.25     0.10    1.000000   4.000000
6       (sour cream)         (curd)             0.30             0.25     0.20    0.666667   2.666667
7             (curd)   (sour cream)             0.25             0.30     0.20    0.800000   2.666667
8             (milk)         (eggs)             0.15             0.10     0.10    0.666667   6.666667
9             (eggs)         (milk)             0.10             0.15     0.10    1.000000   6.666667
10   (bread, cheese)       (butter)             0.15             0.20     0.10    0.666667   3.333333
11   (bread, butter)       (cheese)             0.20             0.25     0.10    0.500000   2.000000
12  (cheese, butter)        (bread)             0.10             0.40     0.10    1.000000   2.500000
13          (butter)(bread, cheese)             0.20             0.15     0.10    0.500000   3.333333

[14 rows x 7 columns]
```

Некоторые из правил могут показаться излишними. Например, присутствует и правило `bread -> butter`, и правило `butter -> bread`. Аналогичным образом существует несколько правил, основанных на наборе элементов (`bread, cheese, butter`). Отчасти это связано с тем, что, как уже говорилось в этой главе, доверие несимметрично; если поменять местами антецедент и консеквент в правиле, значение метрики может измениться. Кроме того, для набора из трех товаров лифт может меняться в зависимости от того, какие элементы являются частью антецедента, а какие — частью консеквента. Таким образом, значение лифта для `(bread, cheese) -> butter` отличается от `(bread, butter) -> cheese`.

Визуализация ассоциативных правил

Как вы узнали из главы 8, визуализация — это простой, но мощный метод анализа данных. При анализе потребительской корзины визуализация — это удобный способ оценки значимости набора ассоциативных правил: можно просматривать метрики для различных пар антецедент/консеквент. В этом разделе мы будем использовать Matplotlib для визуализации ассоциативных правил, созданных в предыдущем разделе, в виде аннотированной тепловой карты (heatmap).

Тепловая карта — это график в виде сетки, где значения ячеек обозначаются цветом. В этом примере мы создадим тепловую карту метрики лифта различных ассоциативных правил. Мы расположим все антецеденты вдоль оси y, а консеквенты — вдоль оси x и заполним область их пересечения соответствующим цветом метрики лифта. Чем темнее цвет, тем выше значение лифта.

> **ПРИМЕЧАНИЕ**
>
> В этом примере мы визуализируем метрику лифта, поскольку она часто используется для оценки значимости ассоциативных правил. Однако вы можете визуализировать и другую метрику, например доверие.

Перед построением графика создадим пустой датафрейм, в который скопируем столбцы `antecedents`, `consequents` и `lift` из созданного ранее датафрейма `rules`:

```
rules_plot = pd.DataFrame()
rules_plot['antecedents']= rules['antecedents'].apply(lambda x:
','.join(list(x)))
rules_plot['consequents']= rules['consequents'].apply(lambda x:
','.join(list(x)))
rules_plot['lift']= rules['lift'].apply(lambda x: round(x, 2))
```

Используем лямбда-функции для преобразования значений столбцов `antecedents` и `consequents` из датафрейма `rules` в строки, чтобы их было удобнее использовать в качестве меток графика. Изначально значения имели тип frozenset (неизменяемые множества Python). Для округления значений лифта до двух знаков после запятой применяем еще одну лямбда-функцию.

Далее необходимо преобразовать вновь созданный датафрейм `rules_plot` в матрицу, которая будет использоваться для создания тепловой карты с консеквентами, расположенными по горизонтали, и антецедентами — по вертикали. Для этого можно изменить форму `rules_plot` так, чтобы уникальные значения в столбце `antecedents` стали индексами, а уникальные значения в столбце `consequents` — новыми столбцами. Значения столбца `lift` будут использоваться для заполнения ячеек преобразованного датафрейма. Для этой цели применим метод `pivot()` датафрейма `rules_plot`:

```
pivot = rules_plot.pivot(index = 'antecedents', columns = 'consequents', values=
'lift')
```

Выбираем столбцы `antecedents` и `consequents` для формирования осей итогового датафрейма `pivot`, а из столбца `lift` берем значения. Если вывести `pivot` на экран, получим:

consequents	bread	butter	cheese	cheese,bread	curd	eggs	milk	sour cream
antecedents								
bread	NaN	2.50	NaN	NaN	NaN	NaN	NaN	NaN
bread,butter	NaN	NaN	2.0	NaN	NaN	NaN	NaN	NaN
butter	2.50	NaN	2.0	3.33	NaN	NaN	NaN	NaN
cheese	1.50	NaN	NaN	NaN	NaN	NaN	NaN	NaN
cheese,bread	NaN	3.33	NaN	NaN	NaN	NaN	NaN	NaN
cheese,butter	2.50	NaN	NaN	NaN	NaN	NaN	NaN	NaN
curd	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.67
eggs	NaN	NaN	NaN	NaN	NaN	NaN	6.67	NaN
milk	1.67	NaN	NaN	NaN	NaN	6.67	NaN	NaN
pasta	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN
sour cream	NaN	NaN	NaN	NaN	2.67	NaN	NaN	NaN

Этот датафрейм содержит все необходимое для построения тепловой карты: значения индекса (antecedents) станут метками оси y, названия столбцов (consequents) — метками оси x, а сетка чисел и `NaN` — значениями для графика (в данном контексте `NaN` означает, что для данной пары антецедент/консеквент отсутствует ассоциативное правило). Извлекаем эти компоненты в отдельные переменные:

```
antecedents = list(pivot.index.values)
consequents = list(pivot.columns)
import numpy as np
pivot = pivot.to_numpy()
```

Теперь у нас есть метки оси y в списке `antecedents`, метки оси x в списке `consequents`, а также значения для графика в NumPy-массиве `pivot`. В скрипте ниже мы используем все эти компоненты для построения тепловой карты с помощью Matplotlib:

```
   import matplotlib
   import matplotlib.pyplot as plt
   import numpy as np
   fig, ax = plt.subplots()
❶  im = ax.imshow(pivot, cmap = 'Reds')
   ax.set_xticks(np.arange(len(consequents)))
   ax.set_yticks(np.arange(len(antecedents)))
   ax.set_xticklabels(consequents)
   ax.set_yticklabels(antecedents)
❷  plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
            rotation_mode="anchor")
❸  for i in range(len(antecedents)):
     for j in range(len(consequents)):
     ❹ if not np.isnan(pivot[i, j]):
        ❺ text = ax.text(j, i, pivot[i, j], ha="center", va="center")
   ax.set_title("Lift metric for frequent itemsets")
   fig.tight_layout()
   plt.show()
```

Основные моменты построения графиков с помощью Matplotlib были рассмотрены в главе 8. Сейчас мы разберем только специфичные для данного конкретного примера строки кода. Метод `imshow()` преобразует данные из массива `pivot` в двумерное изображение (image) с цветовой кодировкой ❶. С помощью параметра метода `cmap` указываем, каким цветам соответствуют числовые значения массива. В Matplotlib есть ряд встроенных цветовых палитр, из которых можно выбрать любую, в том числе `Reds`, используемую в нашем примере.

После создания меток осей используем метод `setp()` для поворота меток оси x на 45 градусов ❷. Это позволит уместить все метки в пространстве, отведенном для маркировки горизонтальной оси. Затем проходим по массиву `pivot` ❸ и создаем текстовые аннотации для каждого квадрата тепловой карты с помощью метода `text()` ❺. Первые два параметра, `j` и `i`, являются x- и y-координатами метки. Следующий параметр, `pivot[i, j]`, — текст метки, остальные параметры задают

другие настройки. Перед вызовом метода `text()` используется оператор `if` для отсеивания пар антецедент/консеквент, для которых отсутствует значение лифта ❹. В противном случае в каждом пустом квадрате тепловой карты появится метка `NaN`.

На рис. 11.1 показан полученный график.

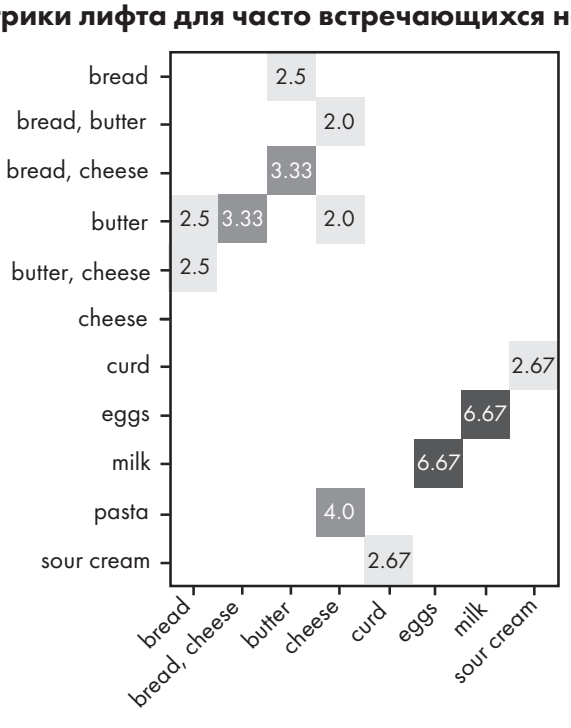

Рис. 11.1. Тепловая карта метрики лифта для выборки ассоциативных правил

Тепловая карта наглядно демонстрирует, для каких ассоциативных правил значения лифта самые высокие (чем темнее цвет ячейки, тем выше значение). Глядя на эту визуализацию, можно с высокой степенью уверенности утверждать, что покупатель, который приобретает молоко, скорее всего, купит и яйца. Точно так же можно быть вполне уверенными, что покупатель, приобретающий макароны, купит и сыр. Есть и другие ассоциативные пары, например масло и сыр, но как видим, они не так сильно подкреплены метрикой лифта.

Тепловая карта также иллюстрирует, что метрика симметрична. Посмотрите, например, на значения правил `bread -> buttero` и `butter -> bread`. Они одинаковы.

Однако можно заметить, что для некоторых пар антецедент/консеквент на графике значение лифта несимметрично. Например, лифт для правила `cheese -> bread` равен `1.5`, однако значения лифта для `bread -> cheese` на графике нет. Это связано с тем, что когда мы изначально создавали ассоциативные правила с помощью функции mlxtend `association_rules()`, мы установили 50-процентный порог доверия. Это решение исключило многие потенциальные ассоциативные правила, включая `bread -> cheese` с показателем доверия 37.5% против 60-процентого доверия пары `cheese -> bread`. Таким образом, данные правила `bread -> cheese` не учитывались при построении графика.

Получение полезных инсайтов из ассоциативных правил

Используя алгоритм Apriori, мы определили часто встречающиеся наборы товаров в выборке данных о транзакциях и на их основе создали ассоциативные правила. По сути, эти правила показывают вероятность того, что покупатель приобретет какой-то продукт, если он уже купил другой, а визуализация значений лифта на тепловой карте позволяет определить самые четкие закономерности. Следующий логичный вопрос — как эта информация может быть полезна бизнесу?

В этом разделе мы рассмотрим два способа, с помощью которых компания может получить полезные инсайты из набора ассоциативных правил. Мы исследуем, как генерировать рекомендации на основе товаров, которые покупатель уже приобрел, и как эффективно планировать скидки на основе часто встречающихся наборов товаров. Обе эти техники помогают увеличить доход компании и одновременно обеспечить лучшее качество клиентского сервиса.

Генерирование рекомендаций

После того как покупатель положил какой-то товар в корзину, какой следующий товар он, скорее всего, добавит? Конечно, это нельзя определить наверняка, однако можно сделать прогноз на основе ассоциативных правил, полученных из данных о транзакциях. Результаты такого прогнозирования могут стать основой для формирования набора рекомендаций тех товаров, которые часто приобретают вместе с товаром, который уже находится в корзине. Ритейлеры обычно используют такие рекомендации, чтобы показать покупателям другие товары, которые им потенциально понадобятся.

Вероятно, самый естественный способ создания рекомендаций такого типа — рассмотрение всех ассоциативных правил, в которых товар, находящийся

в корзине, выступает в качестве антецедента. Затем определяются самые значимые правила — например, три правила с самым высоким значением доверия — и извлекаются их консеквенты. Ниже показано, как выполнить этот алгоритм для товара *butter* (масло). Начинаем с поиска правил, в которых *butter* является антецедентом, используя возможности фильтрации библиотеки pandas:

```
butter_antecedent = rules[rules['antecedents'] == {'butter'}]
[['consequents','confidence']]
            .sort_values('confidence', ascending = False)
```

В коде выше мы сортируем правила по столбцу `confidence` так, чтобы правила с наивысшим рейтингом доверия оказались в начале датафрейма `butter_antecedent`. Далее используем списковое включение для извлечения трех основных консеквентов:

```
butter_consequents = [list(item) for item in butter_antecedent.iloc[0:3:,]
['consequents']]
```

В этом списковом включении мы проходим по столбцу `consequents` датафрейма `butter_antecedent`, выбирая первые три значения. Для списка `butter_consequents` можно сгенерировать рекомендации:

```
item = 'butter'
print('Items frequently bought together with', item, 'are:', butter_consequents)
```

Вот как они выглядят:

```
Items frequently bought together with butter are: [['bread'], ['cheese'],
['cheese', 'bread']]
```

Это указывает на то, что покупатели масла в дополнение к нему часто покупают либо хлеб или сыр, либо и то и другое.

Планирование скидок на основе ассоциативных правил

Ассоциативные правила, созданные для часто встречающихся наборов товаров, также применяются для выбора продуктов, на которые можно назначить скидки. В идеале продукт со скидкой должен быть в каждой значимой группе товаров, чтобы удовлетворить как можно больше покупателей. Другими словами, необходимо выбрать один товар для назначения скидки в каждом часто встречающемся наборе.

Для этого, прежде всего, такие наборы нужно найти. К сожалению, в датафрейме `rules`, созданном ранее функцией `association_rules()`, содержатся колонки с антецедентами и консеквентами, а не с наборами товаров целиком. Поэтому необходимо создать колонку `itemsets`, объединив столбцы `antecedents` и `consequents`, как показано ниже:

```
from functools import reduce
rules['itemsets'] = rules[['antecedents', 'consequents']].apply(lambda x:
                reduce(frozenset.union, x), axis=1)
```

Мы используем функцию `reduce()` из модуля `functools` Python для применения метода `frozenset.union()` к значениям колонок `antecedents` и `consequents`. При этом отдельные неизменяемые множества (frozenset) из этих колонок объединяются в одно.

Чтобы посмотреть, что получилось в результате, можно вывести на экран только что созданный столбец `itemsets` вместе с колонками `antecedents` и `consequents`:

```
print(rules[['antecedents','consequents','itemsets']])
```

Вывод:

```
          antecedents       consequents                itemsets
0            (butter)           (bread)          (butter, bread)
1             (bread)          (butter)          (butter, bread)
2            (cheese)           (bread)          (bread, cheese)
3              (milk)           (bread)            (milk, bread)
4            (butter)          (cheese)         (butter, cheese)
5             (pasta)          (cheese)          (pasta, cheese)
6        (sour cream)            (curd)       (sour cream, curd)
7              (curd)     (sour cream)       (sour cream, curd)
8              (milk)           (eggs)             (milk, eggs)
9              (eggs)           (milk)             (milk, eggs)
10   (butter, cheese)           (bread)  (bread, butter, cheese)
11    (butter, bread)          (cheese)  (butter, cheese, bread)
12    (bread, cheese)          (butter)  (bread, butter, cheese)
13           (butter)   (bread, cheese)  (butter, cheese, bread)
```

Обратите внимание, что в новом столбце `itemsets` есть несколько дубликатов. Как уже говорилось, один и тот же набор товаров может образовывать более одного ассоциативного правила, поскольку порядок товаров влияет на некоторые метрики. Порядок товаров в наборе не имеет значения для

текущей задачи, поэтому можно безопасно удалить дубли наборов, как показано ниже:

```
rules.drop_duplicates(subset=['itemsets'], keep='first', inplace=True)
```

Для этого используется метод датафрейма `drop_duplicates()`, осуществляющий поиск повторяющихся значений в столбце `itemsets`. Мы сохраняем первую строку множества дублирующихся строк и, задавая значение `True` для параметра `inplace`, удаляем дубликаты из существующего датафрейма вместо создания нового.

Выведем на экран колонку `itemsets`:

```
print(rules['itemsets'])
```

И получим:

```
0            (bread, butter)
2            (bread, cheese)
3              (bread, milk)
4           (butter, cheese)
5            (cheese, pasta)
6         (curd, sour cream)
8               (milk, eggs)
10   (bread, cheese, butter)
```

Затем из каждого набора выбираем по одному товару, который будет уценен:

```
   discounted = []
   others = []
❶  for itemset in rules['itemsets']:
   ❷ for i, item in enumerate(itemset):
      ❸ if item not in others:
         ❹ discounted.append(item)
            itemset = set(itemset)
            itemset.discard(item)
         ❺ others.extend(itemset)
            break
      ❻ if i == len(itemset)-1:
           discounted.append(item)
           itemset = set(itemset)
           itemset.discard(item)

         others.extend(itemset)
   print(discounted)
```

Сначала создается список `discounted` для сохранения товаров, выбранных для скидки, и список `others` для получения товаров из набора, на которых скидки не будет. Затем мы проходим по каждому набору товаров ❶ и по каждому товару внутри набора ❷. Мы ищем элемент, которого еще нет в списке `others`, поскольку такого элемента либо нет ни в одном из предыдущих наборов, либо он уже был выбран ранее в качестве уцененного товара для какого-то другого набора. Рационально выбрать его в качестве товара со скидкой и для текущего набора ❸. Мы отправляем выбранный товар в список `discounted` ❹, а остальные товары набора — в список `others` ❺. Если мы перебрали все товары набора и не нашли элемент, которого еще нет в списке `others`, то выбираем последний элемент набора и отправляем его в список `discounted` ❻.

Итоговый список `discounted` будет отличаться, поскольку множества frozenset, представляющие наборы товаров, не упорядочены. Вот как он будет выглядеть:

```
['bread', 'bread', 'bread', 'cheese', 'pasta', 'curd', 'eggs', 'bread']
```

Если сопоставить список с созданным ранее столбцом `itemsets`, можно заметить, что в каждом наборе есть один уцененный товар. Более того, благодаря тому, что мы грамотно распределили скидки, фактически уцененных товаров получилось значительно меньше, чем наборов. Это можно проверить, удалив дубликаты из списка `discounted`:

```
print(list(set(discounted)))
```

Как видно из результата, несмотря на то что у нас восемь наборов, сделать скидку пришлось лишь на пять товаров:

```
['cheese', 'eggs', 'bread', 'pasta', 'curd']
```

Таким образом, нам удалось уценить хотя бы один товар в каждом наборе (существенная выгода для многих покупателей) и не пришлось делать скидку на большее количество продуктов (существенная выгода для бизнеса).

УПРАЖНЕНИЕ № 18: ИЗВЛЕЧЕНИЕ ДАННЫХ О РЕАЛЬНЫХ ТРАНЗАКЦИЯХ

В этой главе мы работали с датасетом, содержащим всего 20 транзакций. Пришло время засучить рукава и поработать с большим набором данных.

В этом упражнении вы будете формировать часто встречающиеся наборы товаров на основе реальной коллекции данных, содержащей более полумиллиона покупок. Этот датасет можно найти в репозитории UCI Machine Learning Repository[1].

Загрузите файл *Online Retail.xlsx* на свой компьютер или воспользуйтесь прямой ссылкой[2] для доступа к файлу. Прежде чем приступить к обработке, прочитайте Excel-файл и преобразуйте его в pandas DataFrame. Для этого установите библиотеку openpyxl. Она позволяет обрабатывать Excel-файлы в Python-скрипте. Установить ее можно с помощью `pip` следующим образом:

```
$ pip install openpyxl
```

После этого используйте openpyxl для загрузки и преобразования датасета в объект DataFrame:

```
import pandas as pd
df_retail = pd.read_excel('path/to/Online Retail.xlsx', index_col=0,
engine='openpyxl')
```

В зависимости от компьютера процесс загрузки может занять несколько минут. Затем, чтобы убедиться, что загрузка прошла успешно, проверьте количество строк в датафрейме `df_retail` и выведите несколько первых строк:

```
print('The number of instances: ', len(df_retail))
print(df_retail.head())
```

Вы должны увидеть следующее:

```
The number of instances:   541909
         StockCode                Description  Quantity  InvoiceDate  UnitPrice  Country
InvoiceNo
536365      85123A  WHITE HANGING HEART T-LIG...       6   2010-12-01  2.55       UK
536365       71053         WHITE METAL LANTERN         6   2010-12-01  3.39       UK
536365      84406B    CREAM CUPID HEARTS COAT...       8   2010-12-01  2.75       UK
536365      84029G     KNITTED UNION FLAG HOT...       6   2010-12-01  3.39       UK
```

[1] https://archive.ics.uci.edu/ml/datasets/online+retail

[2] https://archive.ics.uci.edu/ml/machine-learning-databases/00352/Online%20Retail.xlsx

```
536365      84029E    RED WOOLLY HOTTIE WHITE...       6   2010-12-01  3.39       UK

[5 rows x 7 columns]
```

Реальные данные, подобные представленным, могут быть несовершенными, поэтому на этом этапе необходимо провести очистку, чтобы подготовить их к дальнейшей обработке. Для начала необходимо удалить все строки, имеющие значение `NaN` в поле `Description`. Это поле содержит названия товаров, поэтому значения `NaN` могут исказить процесс определения часто встречающихся наборов:

```
df_retail = df_retail.dropna(subset=['Description'])
```

Чтобы проверить, сработало ли это, выведем длину обновленного датафрейма `df_retail`:

```
print(len(df_retail))
```

Мы получили значение `540455`. Это означает, что теперь в датафрейме на 1454 строки меньше. Чтобы еще лучше подготовить данные, следует убедиться, что значения колонки `Description` приведены к типу данных `str`:

```
df_retail = df_retail.astype({"Description":'str'})
```

Теперь мы готовы преобразовать датасет в формат, который можно обработать с помощью mlxtend. Сначала сгруппируем данные о транзакциях по столбцу `InvoiceNo` и преобразуем датафрейм в список списков, где каждый список содержит набор товаров одной транзакции:

```
trans = df_retail.groupby(['InvoiceNo'])['Description'].apply(list).
to_list()
```

Теперь можно узнать количество фактических транзакций, представленных в датасете:

```
print(len(trans))
```

Должно получиться `24446`.

Следующие шаги анализа были подробно рассмотрены выше в этой главе. В общих чертах необходимо сделать следующее:

1. Преобразовать список списков `trans` в булев OHE-массив с помощью объекта `TransactionEncoder` библиотеки mlxtend.
2. Сформировать часто встречающиеся наборы товаров с помощью функции `apriori()` (для рассматриваемого примера используйте `min_support=0.025`).
3. Сгенерировать ассоциативные правила с помощью функции `association_rules()` (для рассматриваемого примера используйте `metric="confidence"`, `min_threshold=0.3`).

После выполнения этих шагов выведите на экран созданные правила:

```
print(rules.iloc[:,0:7])
```

Вывод должен получиться примерно таким:

```
                      antecedents                    consequents  ...confidence   lift
0         (ALARM CLOCK BAKELI...)    (ALARM CLOCK BAKELIKE GREEN) ... 0.5975   14.5942
1         (ALARM CLOCK BAKELI...)      (ALARM CLOCK BAKELIKE RED) ... 0.6453   14.5942
2   (GREEN REGENCY TEACUP AND...)    (PINK REGENCY TEACUP AND...) ... 0.6092   18.5945
3    (PINK REGENCY TEACUP AND...)   (GREEN REGENCY TEACUP AND...) ... 0.8039   18.5945
4   (GREEN REGENCY TEACUP AND...)   (ROSES REGENCY TEACUP AND...) ... 0.7417   16.1894
5   (ROSES REGENCY TEACUP AND...)   (GREEN REGENCY TEACUP AND...) ... 0.7000   16.1894
6     (JUMBO BAG PINK POLKADOT)         (JUMBO BAG RED RETROSPOT) ... 0.6766    7.7481
7     (JUMBO BAG RED RETROSPOT)         (JUMBO BAG PINK POLKADOT) ... 0.3901    7.7481
8  (JUMBO SHOPPER VINTAGE RED...)       (JUMBO BAG RED RETROSPOT) ... 0.5754    6.5883
9          (JUMBO BAG RED RET...)  (JUMBO SHOPPER VINTAGE RED...) ... 0.3199    6.5883
10       (JUMBO STORAGE BAG SUKI)       (JUMBO BAG RED RETROSPOT) ... 0.6103    6.9882
11          (JUMBO BAG RED RE...)          (JUMBO STORAGE BAG...) ... 0.3433    6.9882
12       (LUNCH BAG BLACK SKULL.)       (LUNCH BAG RED RETROSPOT) ... 0.5003    7.6119
13      (LUNCH BAG RED RETROSPOT)         (LUNCH BAG BLACK SKULL) ... 0.4032    7.6119
14      (LUNCH BAG PINK POLKADOT)       (LUNCH BAG RED RETROSPOT) ... 0.5522    8.4009
15      (LUNCH BAG RED RETROSPOT)       (LUNCH BAG PINK POLKADOT) ... 0.3814    8.4009
16   (PINK REGENCY TEACUP AND...)   (ROSES REGENCY TEACUP AND...) ... 0.7665   16.7311
17  (ROSES REGENCY TEACUP AND...)  (PINK REGENCY TEACUP AND...)   ... 0.5482   16.7311
```

Чтобы создать коллекцию ассоциативных правил большего или меньшего размера, поэкспериментируйте с параметром `min_support`, передаваемым функции `apriori()`, а также с параметрами `metric` и `threshold` внутри функции `association_rules()`.

Выводы

Анализ потребительской корзины — ценный инструмент для извлечения полезной информации из больших объемов транзакционных данных. Из этой главы вы узнали, как использовать алгоритм Apriori для формирования ассоциативных правил из данных о транзакциях, а также как оценивать значимость этих правил с помощью различных метрик. Таким образом можно получить представление о том, какие товары обычно приобретаются вместе. Эти знания мы успешно применили для создания товарных рекомендаций и рационального планирования скидок.

12

Машинное обучение для анализа данных

Машинное обучение (МО) — это метод анализа данных, при котором приложения используют имеющиеся данные для обнаружения закономерностей и принятия решений без четко запрограммированных указаний. Другими словами, приложения обучаются сами, без вмешательства человека. Будучи надежным методом анализа данных, МО используется во многих областях, включая классификацию, кластеризацию, предиктивную аналитику, ассоциативное обучение, обнаружение аномалий, анализ изображений и обработку естественного языка.

В этой главе представлен обзор некоторых фундаментальных концепций машинного обучения, а также подробно рассмотрены два практических примера его применения. Начнем с анализа тональности (sentiment analysis). Мы разработаем модель для прогнозирования количества звезд (от одной до пяти) в отзыве о продукте. Затем создадим еще одну модель для прогнозирования изменений цены акций.

Почему машинное обучение?

Машинное обучение позволяет компьютерам выполнять задачи, которые трудно или даже невозможно решить с помощью обычных методов программирования. Например, представьте, что вам нужно создать приложение для обработки

изображений, которое умеет различать виды животных на основе предоставленных фотографий. Для этого гипотетического сценария у нас уже есть библиотека кода, которая может идентифицировать границы объекта (например, животного) на изображении. Таким образом, животное на фотографии можно превратить в характерный набор линий. Но как с помощью кода различить линии, изображающие двух разных животных, например кошку и собаку?

Традиционный подход к программированию заключается в создании правил, которые сопоставляют каждую комбинацию характерных линий с животным. К сожалению, такое решение потребует большого объема кода, и приложение может не справиться, если границы объектов на какой-то фотографии не будут соответствовать ни одному из правил, заданных вручную. Напротив, приложения, в основе которых лежат алгоритмы машинного обучения, не опираются на заранее определенную логику, а зависят от способности приложения автоматически обучаться на ранее просмотренных данных. Таким образом, приложение для разметки фотографий на основе МО будет искать закономерности в комбинациях линий просмотренных фотографий, а затем, исходя из статистики вероятности, делать прогнозы относительно того, какие животные изображены на новых фотографиях.

Типы машинного обучения

Специалисты по работе с данными различают несколько типов машинного обучения. Наиболее распространенные — это обучение с учителем (supervised learning) и обучение без учителя (unsupervised learning). Эта глава в основном посвящена обучению с учителем, но в данном разделе представлен краткий обзор обоих типов.

Обучение с учителем

Обучение с учителем использует размеченный датасет (называемый *обучающим набором*), чтобы натренировать модель на выдачу желаемого результата при получении новых, ранее неизвестных данных. Строго говоря, обучение с учителем — это техника выведения функции, которая сопоставляет входные данные обучающего набора с выходными данными. Вы уже видели пример обучения такого типа в главе 3, где мы использовали выборку отзывов о товарах, чтобы обучить модель предсказывать, какими будут отзывы о новом товаре — положительными или отрицательными.

Входные данные для алгоритма обучения с учителем могут представлять собой характеристики реальных объектов или событий. Например, можно

использовать характеристики выставленных на продажу домов (площадь, количество спален и ванных комнат и т. д.) в качестве входных данных для алгоритма прогнозирования цен на недвижимость. Цены будут выходными данными алгоритма. На наборе пар входных и выходных данных — характеристик домов и связанных с ними цен — необходимо обучить алгоритм. Тогда можно будет подавать на вход алгоритма характеристики новых домов и на выходе получать их предполагаемую стоимость.

Кроме того, существуют алгоритмы обучения с учителем, которые предназначены для работы не с характеристиками, а с *данными наблюдений*, сведениями, собранными путем наблюдения за работой или поведением. В качестве примера рассмотрим временной ряд с информацией, поступающей с датчиков, которые контролируют уровень шума в аэропорту. Эти наблюдения об уровне шума можно передать алгоритму машинного обучения вместе со сведениями о времени суток и дне недели, чтобы научить его предсказывать уровень шума в ближайшие несколько часов. В данном примере время и дни недели являются входными данными, а уровень шума — выходными. Другими словами, алгоритм разрабатывается для прогнозирования данных будущих наблюдений.

ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ В МАШИННОМ ОБУЧЕНИИ

В программировании *входными данными* (input) обычно называют информацию, которую получает функция, скрипт или приложение. Впоследствии из входных данных получают *выходные данные* (output). Это значения, которые функция, скрипт или приложение возвращают. Однако в контексте машинного обучения с учителем *входные* и *выходные* данные имеют несколько иное значение. Когда модель обучается, она получает *пары* входных и выходных данных, например отзывы о продукте (входные данные) и их класс: положительный или отрицательный (выходные данные). Затем, после обучения модели, предоставляются только новые входные значения, и модель генерирует соответствующие выходные данные на основе того, чему она научилась на примерах пар «вход — выход».

Входные данные модели МО могут состоять из одной или нескольких переменных, которые называются *независимыми переменными* (independent variables), или *признаками* (features). А выходное значение обычно представляет собой одну переменную, известную как *целевая* (target), или *зависимая* (dependant), *переменная*, поскольку выходные данные *зависят* от входных.

Предсказание стоимости дома или уровня шума — типичный пример *регрессии*, распространенного метода обучения с учителем для прогнозирования непрерывных величин. Другой распространенный метод обучения

с учителем — *классификация*, когда модель присваивает набору входных значений одну из меток классов конечного множества. Определение положительных и отрицательных отзывов о продукте, подобно другим приложениям для *анализа тональности*, идентифицирующих фрагменты текста как позитивные или негативные, служат примерами классификации. Мы рассмотрим пример анализа тональности позже в этой главе.

Обучение без учителя

Обучение без учителя — это метод машинного обучения, в котором отсутствует этап обучения. Такие приложения получают входные данные без соответствующих им выходных значений, на которых можно обучаться. В этом смысле модели МО без учителя должны работать самостоятельно, обнаруживая скрытые закономерности во входных данных.

Отличным примером обучения без учителя является *анализ ассоциативных правил* (association analysis), когда приложение на базе МО определяет элементы в наборе, которые имеют сходство друг с другом. В главе 11 мы провели такой анализ на наборе данных о транзакциях, выявив товары, которые обычно приобретают вместе. Мы использовали алгоритм Apriori, которому не нужны выходные данные для обучения; вместо этого он принимает все данные о транзакциях в качестве входных значений и ищет в них часто встречающиеся наборы товаров, таким образом представляя собой обучение без учителя.

Как работает машинное обучение

Типичный пайплайн МО состоит из трех основных компонентов:

- данные для обучения;
- статистическая модель, применяемая к данным;
- новые, неизвестные данные для обработки.

В следующих разделах более подробно рассматривается каждый из этих компонентов.

Данные для обучения

Машинное обучение исходит из того, что компьютерные системы могут обучаться, а для обучения любому алгоритму МО требуются данные. Как мы уже обсуждали, природа этих данных варьируется в зависимости от того, с какой моделью мы имеем дело: с учителем или без учителя. В случае машинного

обучения с учителем данные принимают форму пар входных и выходных значений, которые обучают модель для последующего прогнозирования выходных данных на основе новых входных. А при обучении без учителя модель получает только входные данные и проверяет их на наличие закономерностей, на основе которых можно сформировать выходные значения.

Хотя для всех приложений на основе МО требуются данные, формат этих данных зависит от алгоритма. Множество алгоритмов обучается на основе датасета, данные в котором организованы в виде таблицы, где строки представляют собой экземпляры, например отдельные объекты или определенные моменты времени, а столбцы — признаки этих экземпляров. Классическим примером является датасет Iris[1]. Он включает 150 строк, каждая из которых содержит наблюдения об отдельном экземпляре цветка ириса. Вот так выглядят первые четыре строки датасета:

```
sepal length   sepal width   petal length   petal width   species
5.1            3.5           1.4            0.2           Iris-setosa
4.9            3.0           1.4            0.2           Iris-setosa
4.7            3.2           1.3            0.2           Iris-setosa
4.6            3.1           1.5            0.2           Iris-setosa
```

Первые четыре столбца представляют различные свойства, или признаки, экземпляров. Пятый столбец содержит метку для каждого экземпляра: точное видовое название ириса. Обучая модель классификации на этом датасете, мы бы использовали значения в первых четырех столбцах в качестве независимых переменных, или входных данных, а пятый столбец был бы зависимой переменной, или выходным значением. После такого обучения в идеале модель должна уметь классифицировать видовую принадлежность новых экземпляров ирисов.

Некоторые алгоритмы обучаются на нетабличных данных. Например, алгоритм Apriori, используемый для анализа ассоциативных правил, который мы обсуждали в предыдущей главе, принимает в качестве входных данных набор транзакций (или корзин) различных размеров. Вот простой пример такого набора транзакций:

```
(butter, cheese)
(cheese, pasta, bread, milk)
(milk, cheese, eggs, bread, butter)
(bread, cheese, butter)
```

[1] https://archive.ics.uci.edu/ml/datasets/Iris

Помимо вопроса о том, как структурированы данные машинного обучения, существует проблема зависимости типа используемых данных от алгоритма. Как видно из предыдущих примеров, некоторые алгоритмы МО работают с числовыми или текстовыми данными. Однако существуют и алгоритмы, предназначенные для работы с фото-, видео- и аудиоданными.

Статистическая модель

Какой бы формат данных ни требовался для алгоритма МО, входные данные необходимо привести к виду, пригодному для анализа и получения выходных данных. Именно здесь в игру вступает *статистическая модель*: статистика используется для создания представления данных, чтобы алгоритм мог выявлять взаимосвязи между переменными, находить инсайты, делать прогнозы новых данных, генерировать рекомендации и т. д. Статистические модели лежат в основе любого алгоритма МО.

Например, алгоритм Apriori использует метрику поддержки в качестве статистической модели для поиска часто встречающихся наборов элементов (как упоминалось в главе 11, поддержка — это процент транзакций, включающих некоторый набор товаров или любых других элементов). В частности, алгоритм определяет каждый возможный набор элементов и вычисляет соответствующую метрику поддержки, а затем выбирает наборы с высоким показателем поддержки. Вот простой пример, иллюстрирующий, как работает алгоритм:

```
Itemset          Support
--------------   -------
butter, cheese   0.75
bread, cheese    0.75
milk, bread      0.50
bread, butter    0.50
```

В данном примере показаны наборы, состоящие только из двух товаров. Фактически после вычисления метрики поддержки для каждого возможного набора из двух товаров алгоритм Apriori переходит к анализу наборов из трех позиций, затем из четырех и т. д. После этого алгоритм использует значения поддержки всех наборов товаров (любого размера), чтобы сформировать список часто встречающихся наборов.

Неизвестные данные

В машинном обучении с учителем после обучения модели на имеющихся данных можно применить ее к новым, неизвестным данным. Однако перед этим,

возможно, потребуется оценить производительность модели, поэтому обычно принято разделять исходный датасет на обучающий и тестовый наборы данных. Первый — это данные, на которых модель учится, а второй — это новые данные для тестирования, которые модель раньше не видела.

В тестовых данных тоже содержатся как входные, так и выходные данные, однако модели сообщаются только входные. Затем настоящие выходные данные сравниваются со значениями, предложенными моделью, чтобы оценить точность ее прогнозов. Убедившись, что точность модели приемлема, можно использовать свежие входные данные для проведения прогностического анализа.

В случае обучения без учителя различия между данными, на которых нужно учиться, и неизвестными данными нет. Все данные, по сути, являются новыми, и модель пытается обучиться на них, анализируя основные признаки.

Пример анализа тональности: классификация отзывов о товарах

Теперь, когда мы рассмотрели основы машинного обучения, пришло время попрактиковаться в анализе тональности. Как уже говорилось, этот метод обработки естественного языка позволяет с помощью кода определить, несет фрагмент текста позитивную или негативную эмоциональную окраску (в некоторых приложениях возможны и другие категории, например нейтральная, очень позитивная или очень негативная). По сути, анализ тональности — это форма классификации, метод машинного обучения с учителем, который сортирует данные по дискретным категориям.

В главе 3 мы использовали scikit-learn для проведения простого анализа тональности некоторых отзывов о товарах с сайта Amazon. Мы обучили модель определять, является отзыв хорошим или плохим. Сейчас мы пойдем дальше. Загрузим реальный набор отзывов о товарах непосредственно с Amazon и будем использовать его для обучения модели классификации. Цель модели — предсказать рейтинг отзывов (в звездах) по шкале от одного до пяти. Таким образом, модель будет сортировать отзывы на пять возможных категорий, а не только на две.

Получение отзывов о товарах

Первый шаг в построении модели — загрузка набора реальных отзывов о товарах с сайта Amazon. Один из простых способов сделать это — воспользоваться расширением Amazon Reviews Exporter для браузера Google Chrome. Оно позволяет

загружать отзывы о товарах Amazon в файл CSV. Это расширение можно установить в свой браузер Chrome одним щелчком мыши[1].

Установив расширение, откройте страницу товара Amazon в Chrome. Для этого примера мы используем страницу с книгой Эрика Мэтиза (Eric Matthes) «Python Crash Course»[2] издательства No Starch Press[3], которая на момент написания статьи имеет 445 отзывов. Чтобы загрузить отзывы о книге, найдите и нажмите кнопку Amazon Reviews Exporter на панели инструментов Chrome.

Загрузив отзывы в виде CSV-файла, считайте их и преобразуйте в датафрейм pandas:

```
import pandas as pd
df = pd.read_csv('reviews.csv')
```

Прежде чем продолжить, вам, возможно, будет интересно посмотреть на общее количество отзывов и на первые несколько строк датафрейма:

```
print('The number of reviews: ', len(df))
print(df[['title', 'rating']].head(10))
```

Вывод будет выглядеть примерно так:

```
The number of reviews:   445
                                                 title   rating
0   Great inner content! Not that great outer qual...        4
1                                Very enjoyable read        5
2                                The updated preface        5
3   Good for beginner but does not go too far or deep        4
4                                 Worth Every Penny!        5
5                                 Easy to understand        5
6                              Great book for python.        5
7                   Not bad, but some disappointment        4
8   Truely for the person that doesn't know how to...        3
9         Easy to Follow, Good Intro for Self Learner        5
```

[1] https://chrome.google.com/webstore/detail/amazon-reviews-exporter-c/njlppnciolcibljfdobcefcngiampidm

[2] https://www.amazon.com/Python-Crash-Course-2nd-Edition/dp/1593279280

[3] Мэтиз Э. «Изучаем Python: программирование игр, визуализация данных, веб-приложения». Санкт-Петербург, издательство «Питер».

Здесь для каждой записи представлены только поля `title` (заголовок) и `rating` (рейтинг). Будем рассматривать заголовки отзывов как независимые переменные модели (то есть входные данные), а рейтинги — как зависимые (выходные данные). Обратите внимание, что мы игнорируем полный текст отзыва и ориентируемся только на названия. Это решение целесообразно в рамках анализа тональности, поскольку заголовок обычно представляет собой краткое изложение отношения автора отзыва к товару. А вот полный текст отзыва часто включает другую «неэмоциональную» информацию, например обзор содержания книги.

Очистка данных

Прежде чем обрабатывать реальные данные, их почти всегда необходимо очистить. В рассматриваемом примере нам нужно отфильтровать отзывы, написанные не на английском языке. Для этого нам понадобится программно определить язык каждого отзыва. Существует несколько библиотек Python с возможностью определения языка; используем google_trans_new.

Установка google_trans_new

Установите google_trans_new library с помощью `pip`:

```
$ pip install google_trans_new
```

Прежде чем двигаться дальше, убедитесь, что в установленной версии google_trans_new исправлена ошибка, которая при определении языка вызывает исключение `JSONDecodeError`. Для этого запустите следующий тестовый фрагмент кода в сеансе Python:

```
$ from google_trans_new import google_translator
$ detector = google_translator()
$ detector.detect('Good')
```

Если тест проходит, можно продолжать работу, а если выдается исключение `JSONDecodeError`, потребуется внести небольшие изменения в исходный код библиотеки *google_trans_new.py*. Найдите расположение файла с помощью `pip`:

```
$ pip show google_trans_new
```

Эта команда покажет основную информацию о библиотеке, включая расположение ее исходного кода на локальной машине. Перейдите по указанному пути и откройте *google_trans_new.py* в текстовом редакторе. Затем найдите строки 151 и 233, которые будут выглядеть следующим образом:

```
response = (decoded_line + ']')
```

и измените их на:

```
response = decoded_line
```

Сохраните изменения, перезапустите сеанс Python и повторно запустите тест. Теперь библиотека должна правильно определить *good* как англоязычное слово:

```
$ from google_trans_new import google_translator
$ detector = google_translator()
$ detector.detect('Good)
['en', 'english']
```

ПРИМЕЧАНИЕ

Чтобы узнать больше о google_trans_new, посетите страницу библиотеки на сайте Pypi.org[1].

Удаление неанглоязычных отзывов

Теперь мы готовы определить язык каждого отзыва и отфильтровать отзывы, которые написаны не на английском языке. В следующем фрагменте кода мы используем модуль `google_translator` из google_trans_new для определения языка заголовка каждого отзыва, и сохраним результат в новом столбце датафрейма. Определение языка большого количества образцов может занять некоторое время, будьте готовы к этому:

```
from google_trans_new import google_translator
detector = google_translator()
df['lang'] = df['title'].apply(lambda x: detector.detect(x)[0])
```

Сначала создаем объект `google_translator`, затем с помощью лямбда-функции к каждому заголовку обзора применяем метод объекта `detect()`. Сохраняем

[1] https://pypi.org/project/google-trans-new

результат в новом столбце `lang`. Выводим на экран этот столбец вместе с `title` и `rating`:

```
print(df[['title', 'rating', 'lang']])
```

Вывод будет выглядеть следующим образом:

```
   title   rating   lang
0            Great inner content! Not that great outer qual...   4   en
1                                        Very enjoyable read    5   en
2                                        The updated preface    5   en
3            Good for beginner but does not go too far or deep   4   en
4                                         Worth Every Penny!    5   en
--фрагмент--
440                                                    Not bad   1   en
441                                                      Good    5   en
442                                                     Super    5   en
443                                      内容はとても良い、作りは×   4   ja
444                                                    非常实用   5   zh-CN
```

Следующий шаг — отфильтровать датасет, сохранив только те отзывы, которые написаны на английском языке:

```
df = df[df['lang'] == 'en']
```

Эта операция должна уменьшить общее количество строк в датасете. Чтобы проверить, что это сработало, подсчитайте количество строк в обновленном датафрейме:

```
print(len(df))
```

Количество строк должно быть меньше первоначального, поскольку все неанглоязычные отзывы удалены.

Разделение и преобразование данных

Прежде чем двигаться дальше, необходимо разделить обзоры на обучающий набор для создания модели и тестовый набор для оценки ее точности. Также необходимо преобразовать заголовки рецензий, написанные на естественном языке, в числовые данные, которые сможет понять модель. Как было показано в разделе «Преобразование текста в числовые векторы признаков» главы 3, для

этой цели можно использовать метод «мешок слов» (BoW, bag of words); чтобы вспомнить, как он работает, вернитесь к упомянутому разделу.

В следующем фрагменте кода используется библиотека scikit-learn для разделения и преобразования данных. Алгоритм строится по тому же принципу, что и код из главы 3:

```
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
reviews = df['title'].values
ratings = df['rating'].values
❶ reviews_train, reviews_test, y_train, y_test = train_test_split(reviews,
                ratings, test_size=0.2, random_state=1000)
vectorizer = CountVectorizer()
vectorizer.fit(reviews_train)
❷ x_train = vectorizer.transform(reviews_train)
x_test = vectorizer.transform(reviews_test)
```

Напомним, что функция scikit-learn `train_test_split()` случайным образом разбивает данные на обучающую и тестовую выборку ❶, а класс библиотеки `CountVectorizer` содержит методы для преобразования текстовых данных в числовые векторы признаков ❷. Код генерирует следующие структуры, реализуя обучающий и тестовый наборы данных как массивы NumPy и соответствующие им векторы признаков как разреженные матрицы SciPy:

`reviews_train` Массив, содержащий заголовки отзывов, выбранных для обучения.

`reviews_test` Массив, содержащий заголовки отзывов, выбранных для тестирования.

`y_train` Массив, содержащий рейтинги (количество звезд), соответствующие отзывам в `reviews_train`.

`y_test` Массив, содержащий рейтинги (количество звезд), соответствующие отзывам в `reviews_test`.

`x_train` Матрица, содержащая набор векторов признаков для заголовков отзывов из массива `reviews_train`.

`x_test` Матрица, содержащая набор векторов признаков для заголовков отзывов из массива `reviews_test`.

Нас больше всего интересуют `x_train` и `x_test`, числовые векторы признаков, которые scikit-learn сгенерировал из заголовков отзывов, используя метод BoW.

На каждый заголовок отзыва в матрице должна приходиться одна строка, которая представляет собой числовой вектор признаков. Чтобы проверить количество строк в матрице, сформированной из массива `reviews_train`, используйте следующую команду:

```
print(len(x_train.toarray()))
```

Полученное число должно составлять 80% от общего числа англоязычных отзывов, поскольку мы разделили данные на обучающий и тестовый наборы по схеме 80/20. Матрица `x_test` должна содержать остальные 20% векторов. Проверить это можно с помощью следующей функции:

```
print(len(x_test.toarray()))
```

Также можно проверить длину векторов признаков в обучающей матрице:

```
print(len(x_train.toarray()[0]))
```

Мы выводим на экран длину только первой строки матрицы, поскольку длина каждой строки одинакова. Получаем следующий результат:

```
442
```

Это означает, что в названиях отзывов обучающего набора встречается 442 уникальных слова. Такая коллекция слов называется *словарным составом* (vocabulary dictionary) набора данных.

Если интересно, можете вывести на экран всю матрицу:

```
print(x_train.toarray())
```

Вот что получится:

```
[[000...100]
 [000...000]
 [000...000]
 --фрагмент--
 [000...000]
 [000...000]
 [000...000]]
```

Каждый столбец матрицы соответствует одному из слов словарного состава набора данных, а значение указывает, сколько раз каждое слово встречается в том или ином заголовке отзыва. Как видите, матрица состоит в основном из нулей. Этого следовало ожидать: в среднем заголовок рецензии состоит всего из 5–10 слов, однако словарный состав всего набора включает 442 слова, это означает, что в обычной строке только 5–10 элементов из 442 будут иметь значение 1 или выше. Тем не менее такое представление данных — именно то, что нужно, чтобы обучить модель классифицировать тональности.

Обучение модели

Теперь мы готовы перейти к обучению модели. В частности, нам нужно обучить *классификатор* (classifier), модель машинного обучения, которая сортирует данные по категориям. Таким образом мы сможем предсказывать количество звезд в отзыве. Для этого можно использовать классификатор из scikit-learn — `LogisticRegression`:

```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression()
classifier.fit(x_train, y_train)
```

Импортируем класс `LogisticRegression` и создадим объект `classifier`. Затем обучим классификатор, передавая ему матрицу `x_train` (векторы признаков заголовков отзывов обучающего набора) и массив `y_train` (соответствующие рейтинги, выраженные в количестве звезд).

Оценка модели

Теперь, когда модель обучена, используем матрицу `x_test` для оценки ее точности, сравнивая предсказанные моделью рейтинги с фактическими рейтингами из массива `y_test`. В главе 3 мы использовали метод классификатора `score()` для оценки точности. На этот раз применим другой метод оценки, который позволяет добиться большей точности:

```
   import numpy as np
❶ predicted = classifier.predict(x_test)
   accuracy = ❷ np.mean(❸ predicted == y_test)
   print("Accuracy:", round(accuracy,2))
```

Мы применяем метод классификатора `predict()` для предсказания оценок на основе векторов признаков из `x_test` ❶. Затем проверяем равнозначность

предсказаний модели и реальных рейтингов ❸. Результатом этого сравнения будет булев массив, где значения `True` и `False` будут указывать, был ли прогноз точным. Взяв среднее арифметическое из массива ❷, получим общую оценку точности модели (при расчете среднего каждое значение `True` трактуется как `1`, а `False` — как `0`). Вывод результата на экран будет примерно таким:

```
Accuracy: 0.68
```

Это говорит о том, что точность модели составляет 68%, то есть в среднем примерно 7 прогнозов из 10 оказываются верными. Однако чтобы получить более тонкую оценку точности модели, необходимо использовать другие функции scikit-learn для исследования более конкретных метрик. Например, можно исследовать *матрицу ошибок* (confusion matrix) модели — сетку, которая сравнивает предсказанные классы с фактическими. Матрица ошибок помогает узнать точность модели в каждом отдельном классе, а также показывает вероятность того, что модель перепутает два класса (присвоит метку одного класса другому). Создать матрицу ошибок для модели классификации можно следующим образом:

```
from sklearn import metrics
print(metrics.confusion_matrix(y_test, predicted, labels = [1,2,3,4,5]))
```

Импортируем модуль scikit-learn `metrics`, а затем используем метод `confusion_matrix()` для создания матрицы. Методу передаются фактические оценки рейтинга тестового набора (`y_test`), оценки, предсказанные моделью (`predicted`), и метки классов, соответствующие этим оценкам. Полученная матрица будет выглядеть примерно так:

```
[[ 0,   0,   0,   1,   7],
 [ 0,   0,   1,   0,   1],
 [ 0,   0,   0,   4,   3],
 [ 0,   0,   0,   1,   6],
 [ 0,   0,   0,   3,  54]]
```

Здесь строки соответствуют фактическим значениям рейтинга, а столбцы — прогнозируемым. Например, если посмотреть на значения в первой строке, то можно сказать, что тестовый набор в реальности содержал 8 рейтингов с одной звездой, однако модель предсказала для одного из отзывов рейтинг с четырьмя звездами, а для остальных 7 — с пятью звездами.

Главная диагональ матрицы ошибок (с верхнего левого угла до правого нижнего) показывает количество верно предсказанных значений для каждого уровня

рейтинга. Взглянув на эту диагональ, можно увидеть, что модель сделала 54 правильных прогноза для пятизвездочных отзывов и только один правильный прогноз для отзывов с четырьмя звездами. Ни один отзыв с одной, двумя или тремя звездами не был определен правильно. В общей сложности из тестового набора, состоящего из 81 отзыва, 55 были определены верно.

Этот результат вызывает ряд вопросов. Например, почему модель хорошо работает только для пятизвездочных отзывов? Проблема может заключаться в том, что в датасете достаточно много примеров с пятизвездочным рейтингом. Чтобы проверить, так ли это, можно подсчитать количество строк в каждой группе рейтинга:

```
print(df.groupby('rating').size())
```

Группируем исходный датафрейм, содержащий данные для обучения и тестирования, по столбцу `rating` и используем метод `size()` для получения количества записей в каждой группе. Получим следующий вывод:

```
rating
1     25
2     15
3     23
4     51
5    290
```

Как видите, этот результат подтверждает нашу гипотезу: отзывов с пятью звездами гораздо больше, чем отзывов с любым другим рейтингом, что говорит о том, что у модели не было достаточно данных для эффективного изучения признаков отзывов с четырьмя звездами и ниже.

Для более тщательного исследования точности модели также можно рассмотреть основные метрики классификации, передав массивы `y_test` и `predicted` в функцию `classification_report()` из модуля `metrics` библиотеки scikit-learn:

```
print(metrics.classification_report(y_test, predicted, labels = [1,2,3,4,5]))
```

Сформированный отчет будет выглядеть следующим образом:

```
              precision    recall  f1-score   support
           1       0.00      0.00      0.00         8
           2       0.00      0.00      0.00         2
           3       0.00      0.00      0.00         7
           4       0.11      0.14      0.12         7
```

```
           5      0.76      0.95      0.84        57

    accuracy                           0.68        81
   macro avg      0.17      0.22      0.19        81
weighted avg      0.54      0.68      0.60        81
```

Этот отчет содержит сводку основных метрик классификации для каждого класса отзывов. Здесь нам важны метрики поддержки (support) и полноты (recall); дополнительную информацию о других метриках отчета можно найти в документации[1].

Метрика поддержки отображает количество отзывов для каждого класса оценок. В частности, она демонстрирует, что отзывы распределены по рейтинговым группам крайне неравномерно, причем тестовый набор данных выявляет ту же тенденцию, что и весь датасет. 57 отзывов из 81 имеет рейтинг из пяти звезд, и только 2 — из двух звезд.

Метрика полноты показывает отношение отзывов, для которых правильно предсказан рейтинг, ко всем отзывам с таким же рейтингом. Например, метрика полноты отзывов с пятью звездами равна 0.95, что означает, что модель с 95-процентной точностью делает прогнозы для отзывов с таким рейтингом, в то время как полнота для отзывов с четырьмя звездами составляет всего 0.14. Поскольку для отзывов с другими оценками правильных предсказаний нет, средневзвешенная полнота для всего тестового набора, которая отображена внизу в отчете, равна 0.68. Это та же оценка точности, которую мы получили в начале этого раздела.

Принимая во внимание все эти моменты, можно прийти к выводу, что проблема заключается в том, что в используемом наборе примеров отзывы крайне неравномерно распределены по рейтинговым группам.

УПРАЖНЕНИЕ № 19: РАСШИРЕНИЕ НАБОРА ПРИМЕРОВ

Как мы только что выяснили, общая точность модели классификации может быть обманчивой, если в наборе данных количество экземпляров каждого класса неравномерно. Попробуйте расширить набор данных, загрузив больше отзывов с Amazon. Постарайтесь получить примерно одинаковое (и достаточно большое) количество экземпляров для каждого показателя рейтинга (скажем, по 500 примеров на группу). Затем повторно обучите модель и снова протестируйте ее, чтобы узнать, повысилась ли точность.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html#sklearn.metrics.classification_report

Прогнозирование тенденций фондового рынка

Чтобы узнать, как еще можно применять машинное обучение для анализа данных, создадим модель для прогнозирования тенденций фондового рынка. Построим еще одну простую классификационную модель, которая предсказывает, будет ли завтра цена акции выше, ниже или такой же, как сегодня. Более сложный вариант модели может задействовать регрессию для прогнозирования изменения фактической стоимости акций по дням.

> **ПРЕДУПРЕЖДЕНИЕ**
>
> Рассматриваемая здесь модель носит исключительно иллюстративный характер и не предназначена для реального использования. Модели машинного обучения, используемые для торговли на бирже, как правило, гораздо сложнее. Любая попытка использовать модель этой книги для совершения реальных биржевых сделок может привести к убыткам, за которые ни автор, ни издательство не несут ответственности.

По сравнению с моделью для анализа тональности новая модель, прогнозирующая тенденции фондового рынка (как и многие другие модели, использующие нетекстовые данные), поднимает новый вопрос: как выбирать данные, используемые в качестве признаков или входных значений? Для анализа тональности мы использовали векторы признаков, сгенерированные из текста заголовков отзывов с помощью техники BoW. Содержание такого вектора сильно зависит от содержания соответствующего текста. В этом смысле содержание вектора предопределено, он формируется из признаков, извлеченных из текста по определенному правилу.

А когда модель, напротив, использует нетекстовые данные, например цены на акции, часто приходится выбирать и, возможно, даже вычислять набор признаков, который будет использован в качестве входных данных. Что это может быть: процентное изменение цены за день, средняя цена за последнюю неделю или общий объем торгов за два предыдущих дня? Или же процентное изменение цены за два дня, средняя цена за последний месяц и изменение объема торгов за день? В качестве исходных данных для моделей, прогнозирующих тенденции фонового рынка, финансовые аналитики используют всевозможные метрики, подобные этим, в различных комбинациях.

Из главы 10 вы узнали, как извлекать метрики из данных фондового рынка путем расчета процентного изменения цены во времени либо с помощью скользящих окон и т. п. Мы вернемся к некоторым из этих методов позже в этом разделе, когда будем создавать признаки для нашей прогностической модели. Но сначала получим данные.

Получение данных

Для обучения модели нам понадобятся данные за год по акциям отдельной компании. Для примера возьмем Apple (AAPL). Во фрагменте кода ниже мы используем библиотеку yfinance, чтобы получить данные об акциях компании за последний год:

```
import yfinance as yf
tkr = yf.Ticker('AAPL')
hist = tkr.history(period="1y")
```

Будем использовать полученный датафрейм `hist` для расчета метрик, например ежедневное изменение цены в процентах, и передавать эти метрики в модель. Однако разумно предположить, что существуют и внешние факторы (то есть информация, которую невозможно получить из данных о самих акциях), влияющие на цену акций Apple. Например, на показатели отдельных акций могут влиять общие тенденции фондового рынка. Таким образом, в рамках модели интересно также учесть данные об индексе всего фондового рынка.

Одним из наиболее известных индексов фондового рынка является S&P 500. Он содержит данные о динамике акций 500 крупных компаний. Как вы видели в главе 4, данные S&P 500 можно получить через Python с помощью библиотеки pandas-datareader. Ниже мы используем метод `get_data_stooq()` для получения данных S&P 500 за один год с сайта Stooq:

```
   import pandas_datareader.data as pdr
   from datetime import date, timedelta
   end = date.today()
❶  start = end - timedelta(days=365)
❷  index_data = pdr.get_data_stooq('^SPX', start, end)
```

Используя модуль Python `datetime`, мы определяем начальную и конечную дату запроса относительно текущей даты ❶. Затем вызываем метод `get_data_stooq()`, используя `'^SPX'` для запроса данных S&P 500, и сохраняем результат в датафрейме `index_data` ❷.

Теперь, когда у нас есть показатели акций Apple и индекса S&P 500 за один и тот же год, объединим эти данные в один датафрейм:

```
df = hist.join(index_data, rsuffix = '_idx')
```

В объединяемых датафреймах есть столбцы с одинаковыми именами. Чтобы избежать дублирования, используем параметр `rsuffix`. Он сообщает методу

`join()`, что нужно добавить суффикс `'_idx'` ко всем именам столбцов из датафрейма `index_data`.

В данном примере нас будут интересовать только ежедневные показатели цены закрытия и объема торгов как для Apple, так и для S&P 500. Фильтруем датафрейм так, чтобы остались только эти столбцы:

```
df = df[['Close','Volume','Close_idx','Volume_idx']]
```

Выведем датафрейм `df` на экран:

```
                Close      Volume   Close_idx   Volume_idx
Date
2021-01-15   126.361000   111598500     3768.25   2741656357
2021-01-19   127.046791    90757300     3798.91   2485142099
2021-01-20   131.221039   104319500     3851.85   2350471631
2021-01-21   136.031403   120150900     3853.07   2591055660
2021-01-22   138.217926   114459400     3841.47   2290691535
--snip --
2022-01-10   172.190002   106765600     4670.29   2668776356
2022-01-11   175.080002    76138300     4713.07   2238558923
2022-01-12   175.529999    74805200     4726.35   2122392627
2022-01-13   172.190002    84505800     4659.03   2392404427
2022-01-14   173.070007    80355000     4662.85   2520603472
```

Датафрейм содержит непрерывный многомерный временной ряд. Следующий шаг — извлечение из данных признаков, которые можно использовать в качестве входных данных для модели машинного обучения.

Извлечение признаков из непрерывных данных

Мы хотим обучить модель на информации об изменениях цены и объема акций по дням. Как упоминалось в главе 10, вычисление изменений в процентах для данных непрерывного временного ряда производится путем сдвига точек данных во времени, чтобы привести более ранние точки данных в соответствие с текущими для их сравнения. Во фрагменте кода ниже мы используем `shift(1)` для вычисления процентного изменения цены по дням для каждого столбца датафрейма и сохраняем результаты в новой партии столбцов:

```
import numpy as np
df['priceRise'] = np.log(df['Close'] / df['Close'].shift(1))
df['volumeRise'] = np.log(df['Volume'] / df['Volume'].shift(1))
df['priceRise_idx'] = np.log(df['Close_idx'] / df['Close_idx'].shift(1))
```

```
df['volumeRise_idx'] = np.log(df['Volume_idx'] / df['Volume_idx'].shift(1))
df = df.dropna()
```

Для каждого из четырех столбцов разделите текущую точку данных на точку данных в предшествующий день, затем вычислите натуральный логарифм полученного значения. Помните, что натуральный логарифм обеспечивает хорошее приближение процентного изменения. В итоге получаем несколько новых столбцов:

`priceRise` Процентное изменение цены акций Apple по дням.

`volumeRise` Процентное изменение объема торгов Apple по дням.

`priceRise_idx` Процентное изменение цены индекса S&P 500 по дням.

`volumeRise_idx` Процентное изменение объема торгов для S&P 500 по дням.

Теперь снова отфильтруем датафрейм, оставив только новые столбцы:

```
df = df[['priceRise','volumeRise','priceRise_idx','volumeRise_idx']]
```

Содержимое датафрейма будет выглядеть следующим образом:

```
            priceRise   volumeRise   priceRise_idx   volumeRise_idx
Date
2021-01-19   0.005413    -0.206719        0.008103        -0.098232
2021-01-20   0.032328     0.139269        0.013839        -0.055714
2021-01-21   0.036003     0.141290        0.000317         0.097449
2021-01-22   0.015946    -0.048528       -0.003015        -0.123212
2021-01-25   0.027308     0.319914        0.003609         0.199500
--snip--
2022-01-10   0.000116     0.209566       -0.001442         0.100199
2022-01-11   0.016644    -0.338084        0.009118        -0.175788
2022-01-12   0.002567    -0.017664        0.002814        -0.053288
2022-01-13  -0.019211     0.121933       -0.014346         0.119755
2022-01-14   0.005098    -0.050366        0.000820         0.052199
```

Эти столбцы станут признаками (или независимыми переменными) для модели.

Генерирование выходной переменной

Следующий шаг — вычисление выходной переменной (также называемой целевой или зависимой переменной) на основе имеющегося датасета. Эта переменная должна отражать, что произойдет с ценой акции на следующий день: вырастет

она, упадет или останется прежней. Выяснить это можно, изучив столбец `priceRise` в строке для следующего дня, который можно получить с помощью команды `df['priceRise'].shift(-1)`. Отрицательный сдвиг смещает будущие значения назад во времени. На основе этого сдвига можно создать новый столбец со значением `-1`, если цена падает, `0`, если цена остается прежней, и `1`, если цена растет. Вот как это делается:

```
❶ conditions = [
      (df['priceRise'].shift(-1) > 0.01),
      (df['priceRise'].shift(-1)< -0.01)
  ]
❷ choices = [1, -1]
  df['Pred'] = ❸ np.select(conditions, choices, default=0)
```

Алгоритм, реализованный выше, исходит из следующих предположений:

1. Увеличение цены более чем на 1% по отношению к стоимости акции в следующий день расценивается как повышение (`1`).
2. Снижение цены более чем на 1% по отношению к стоимости акции в следующий день расценивается как падение (`-1`).
3. Остальные случаи расцениваются как стагнация (`0`).

Для реализации алгоритма мы задаем список `conditions`, который проверяет данные в соответствии с пунктами 1 и 2 ❶, а также список `choices` со значениями `1` и `-1` для обозначения роста или падения цены ❷. Затем мы передаем эти два списка в функцию NumPy `select()` ❸, которая создает массив, выбирая значения из `choices` на основе значений в `conditions`. Если ни одно из условий не выполняется, в соответствии с пунктом 3 по умолчанию используется значение `0`. Сохраняем массив в новом столбце `Pred` датафрейма, который можно использовать в качестве выходных данных для обучения и тестирования модели. По сути, `-1`, `0` и `1` теперь являются классами, из которых модель будет делать выбор при классификации новых данных.

Обучение и оценка модели

Для обучения модели scikit-learn требуется представить входные и выходные данные в отдельных массивах NumPy. Создаем массивы из датафрейма `df`:

```
features =
df[['priceRise','volumeRise','priceRise_idx','volumeRise_idx']].to_numpy()
features = np.around(features, decimals=2)
target = df['Pred'].to_numpy()
```

Массив `features` теперь содержит четыре независимые переменные (входные данные), а массив `target` — одну зависимую переменную (выходные данные). Далее можно разделить данные на обучающий и тестовый наборы и обучить модель:

```
from sklearn.model_selection import train_test_split
rows_train, rows_test, y_train, y_test = train_test_split(features, target,
test_size=0.2)
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression()
clf.fit(rows_train, y_train)
```

Так же как в примере с анализом тональности, приведенном в начале главы, мы используем функцию scikit-learn `train_test_split()` для разделения датасета по схеме 80/20, а для обучения модели используем классификатор `LogisticRegression`. Далее передаем тестовую часть датасета в метод классификатора `score()` для оценки его точности:

```
print(clf.score(rows_test, y_test))
```

Результат будет приблизительно таким:

```
0.6274509803921569
```

Он означает, что примерно в 62% случаев модель верно предсказала тенденции для акций Apple на следующий день. Разумеется, у вас может получиться другая цифра.

УПРАЖНЕНИЕ № 20: ЭКСПЕРИМЕНТИРУЕМ С РАЗЛИЧНЫМИ АКЦИЯМИ И НОВЫМИ МЕТРИКАМИ

Продолжая наш пример, поэкспериментируйте с различными акциями и попробуйте использовать новые метрики, полученные из данных об акциях, в качестве дополнительных независимых переменных, чтобы повысить точность модели. Возможно, вам пригодятся метрики, выведенные в главе 10.

Выводы

Из этой главы вы узнали, как некоторые задачи анализа данных, такие как классификация, можно решить с помощью машинного обучения — метода, который позволяет компьютерным системам учиться на основе исторических данных или прошлого опыта. В частности, мы рассмотрели, как использовать алгоритмы МО для решения задачи по анализу тональности естественного языка. Мы преобразовали текстовые данные из отзывов о товарах Amazon в машиночитаемые числовые векторы признаков, затем обучили модель классифицировать отзывы в соответствии с их рейтингом. Вы также узнали, как генерировать признаки на основе числовых данных фондового рынка, и использовали эти признаки, чтобы обучить модель прогнозировать изменения цен акций.

В Python доступно множество способов объединения методов машинного обучения, статистических методов, общедоступных API и возможностей структур данных. Эта книга на разных примерах показала вам некоторые из них. Я надеюсь, что она вдохновила вас на поиск новых прогрессивных решений.

Юлий Васильев

Python для data science

Перевела с английского А. Алимова

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Е. Строганова*
Литературный редактор	*М. Трусковская*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 06.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 07.04.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 21,930. Тираж 1000. Заказ 0000.