PYTHON PARA CIÊNCIA DE DADOS

UMA INTRODUÇÃO PRÁTICA

YULI VASILIEV

Python para Ciência de Dados

Uma introdução prática

Yuli Vasiliev

Novatec

Editor: Rubens Prates
Tradução: Cibelle Ravaglia
Revisão gramatical: Tássia Carvalho

ISBN do impresso: 978-85-7522-848-7

ISBN do ebook: 978-85-7522-849-4

Histórico de impressões:

Maio/2023 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
E-mail: novatec@novatec.com.br
Site: https://novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec
GRA20230411

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Vasiliev, Yuli

Python para ciência de dados : uma introdução prática / Yuli Vasiliev ; tradução Cibelle Ravaglia. -- 1. ed. -- São Paulo : Novatec Editora, 2023.

Título original: Python for data science
ISBN 978-85-7522-848-7

1. Python (Linguagem de programação para computadores) I. Título.

23-151673 CDD-005.133

Índices para catálogo sistemático:

1. Python : Linguagem de programação : Computadores : Processamento de dados 005.133

Eliane de Freitas Leite - Bibliotecária - CRB 8/8415

Sumário

Sobre o autor

Yuli Vasiliev é programador, escritor e consultor especializado em desenvolvimento open source, criação de estruturas e modelos de dados e implementação de banco de dados back-end. É o autor do livro *Natural Language Processing with Python and spaCy* (Processamento de Linguagem Natural com Python e spaCy em tradução livre) (No Starch Press, 2020).

Sobre o revisor técnico

Dr. Daniel Zingaro é professor adjunto de ciência da computação e professor premiado da Universidade de Toronto. O foco de sua pesquisa é a compreensão e a melhoria da aprendizagem dos alunos de ciência da computação. É autor de dois livros recentes pela editora No Starch Press: *Algorithmic Thinking* (Pensamento Algorítmico, em tradução livre, de 2020), um guia pragmático e sem matemática para algoritmos e estruturas de dados; e *Learn to Code by Solving Problems* (Aprenda a Programar Resolvendo Problemas, em tradução livre, de 2021), introdução para aprender Python e pensamento computacional.

Introdução

Vivemos no mundo de tecnologia da informação (TI), em que sistemas computacionais coletam enormes quantidades de dados para processá-los e extrair informações úteis deles. Essa realidade orientada a dados impacta não somente como as empresas modernas operam, mas também nossa vida cotidiana. Sem os inúmeros dispositivos e sistemas que usam tecnologias com foco em dados, seria difícil para muitos de nós manter contato com a sociedade. Mapas e navegação móveis, compras online e dispositivos domésticos inteligentes são alguns exemplos comuns do uso diário da tecnologia com foco em dados.

No mundo corporativo, as empresas costumam usar sistemas tecnológicos para tomar decisões ao extrair informações acionáveis de volumes gigantescos de dados. Os dados podem se originar de variadas fontes, ter diferentes formatos e podem exigir transformação antes de estarem prontos para análise. Por exemplo, muitas empresas com negócios online usam a análise de dados a fim de impulsionar a aquisição e retenção de clientes, coletando e mensurando tudo o que conseguem para modelar e entender o comportamento de seus usuários. Em geral, elas combinam e analisam dados quantitativos e qualitativos de usuários e de fontes diferentes, como perfis de usuários, mídias sociais e sites de empresas. E, em muitos casos, realizam todas essas tarefas usando a linguagem de programação Python.

Neste livro, você será apresentado ao mundo pythônico de dados, sem jargões acadêmicos ou complexidade excessiva. Você também aprenderá como usar o Python para desenvolver aplicações orientadas a dados, escrever códigos para um serviço de compartilhamento de viagens, gerar recomendações de produtos, predizer tendências do mercado de ações e muito mais. A partir desses exemplos do mundo cotidiano, você será capaz de obter experiência prática com as principais bibliotecas de ciência de dados do Python.

Usando o Python para ciência de dados

A linguagem de programação Python, além de ser fácil de entender, é a escolha ideal para acessar, manipular e obter insights de dados de qualquer tipo. O Python tem um conjunto sofisticado de estruturas built-in de dados para operações básicas e um ecossistema robusto de bibliotecas open source para análise e manipulação de dados com qualquer nível de complexidade. Neste livro, exploraremos muitas dessas bibliotecas, como NumPy, pandas, scikit-learn, Matplotlib, entre outras.

Com o Python, você será capaz de escrever um código conciso e intuitivo com o mínimo de empenho e tempo, expressando a maioria dos conceitos em apenas algumas linhas de código. Na prática, a sintaxe ágil do Python possibilita implementar inúmeras operações de dados com uma única linha de código. Por exemplo, é possível escrever um one-liner que filtre, transforme e agregue dados de uma só vez.

Como é uma linguagem de uso geral, o Python é adequado para uma ampla variedade de tarefas. Ao trabalhar com Python, podemos integrar perfeitamente a ciência de dados com outras tarefas para criar aplicações completas e totalmente funcionais. Por exemplo, é possível desenvolver um aplicativo de bot que faz previsões do mercado de ações em resposta a solicitações de linguagem natural dos usuários. Para tal, seria necessária uma API bot, um modelo de aprendizado de máquina (machine learning) para efetuar predições e uma ferramenta de processamento de linguagem natural (PNL) para interagir com os usuários. O Python disponibiliza bibliotecas poderosas para criarmos tudo isso.

A quem este livro se destina?

Este livro se destina a desenvolvedores que buscam obter melhor compreensão dos recursos de análise e de processamento de dados do Python. Talvez você trabalhe para uma empresa que queira usar dados para melhorar os processos de negócios, tomar melhores decisões e segmentar mais clientes. Ou talvez queira desenvolver as próprias aplicações orientadas a dados, ou simplesmente queira ampliar seu conhecimento sobre Python para a área de

ciência de dados.

Neste livro, parto do princípio de que o leitor já tenha experiência básica com Python e que se sinta à vontade em seguir instruções para executar tarefas como instalar um banco de dados ou obter uma chave de API. No entanto, o livro aborda os conceitos de ciência de dados do Python com detalhes, por meio de exemplos práticos, minuciosamente explicados. Você aprenderá na prática, sem necessidade de experiência prévia em dados.

Qual é o conteúdo deste livro?

O livro começa com uma introdução conceitual ao processamento e análise de dados, explicando como um típico pipeline de processamento de dados funciona. Em seguida, abordaremos as estruturas built-in de dados do Python e algumas das bibliotecas Python de terceiros, amplamente usadas em aplicações de ciência de dados. Depois, exploraremos técnicas cada vez mais sofisticadas para obter, combinar, agregar, agrupar, analisar e visualizar conjuntos de dados de diferentes tamanhos e tipos de dados. À medida que avançamos com o conteúdo, aplicaremos técnicas de ciência de dados do Python a casos de uso reais do mundo do gerenciamento de negócios, marketing e finanças. No decorrer da obra, cada capítulo tem seções com "Exercício" para que você possa praticar e reforçar o que acabou de aprender.

Confira o panorama geral de cada capítulo:

- Capítulo 1: Conceitos básicos de dados – Fornece os conceitos necessários para entender os fundamentos do trabalho com dados. Você aprenderá que existem diferentes categorias de dados, incluindo dados estruturados, não estruturados e semiestruturados. Em seguida, estudará as etapas envolvidas em um típico processo de análise de dados.
- Capítulo 2: Estruturas de dados do Python – Apresenta quatro estruturas de dados incorporadas ao Python: listas, dicionários, tuplas e conjuntos. Você verá como usar cada estrutura e como combiná-las em estruturas mais complexas que podem representar objetos do mundo cotidiano.
- Capítulo 3: Bibliotecas de ciência de dados Python – Analisa o ecossistema robusto de bibliotecas de terceiros do Python para análise e manipulação de dados. Você conhecerá a biblioteca pandas e suas

estruturas primárias de dados, a Series e o DataFrame, que se tornaram o padrão de referência para aplicações Python orientadas a dados. Além disso, será apresentado à NumPy e à scikit-learn, duas outras bibliotecas usadas com frequência em ciência de dados.

- Capítulo 4: Acessando dados a partir de arquivos e de APIs – Mergulha nos detalhes da obtenção de dados e de como carregá-los para nossos scripts. Você aprenderá a carregar dados a partir de diferentes fontes, como arquivos e APIs, para estruturas de dados em seus scripts Python para processamento posterior.
- Capítulo 5: Trabalhando com bancos de dados – Continua a análise sobre a importação de dados para o Python, abordando como trabalhar com banco de dados. Você verá exemplos de acesso e manipulação de dados armazenados em diferentes tipos de bancos de dados, incluindo bancos de dados relacionais como MySQL e bancos de dados NoSQL como MongoDB.
- Capítulo 6: Agregação de dados – Aborda o problema de sintetização de dados ao ordená-los em grupos, efetuando cálculos agregados. Você aprenderá como usar a biblioteca pandas para agrupar dados e gerar subtotais, totais e outras agregações.
- Capítulo 7: Combinando conjuntos de dados – Abrange como combinar dados de diferentes fontes em um único conjunto de dados. Você aprenderá técnicas que os desenvolvedores SQL usam para fazer join de tabelas de banco de dados e aplicá-las a estruturas built-in de dados, arrays NumPy e DataFrames pandas.
- Capítulo 8: Criando visualizações – Apresenta as visualizações como forma mais natural de revelar padrões ocultos nos dados. Você aprenderá sobre diferentes tipos de visualizações, como gráficos de linhas, gráficos de barras e histogramas, e verá como criá-las com a Matplotlib, a principal biblioteca Python para plotagem. Também usará a biblioteca Cartopy para gerar mapas.
- Capítulo 9: Analisando dados geográficos – Explica como trabalhar com dados geográficos usando as bibliotecas geopy e Shapely. Você aprenderá como obter e usar coordenadas GPS para objetos estacionários e móveis, e explorará o exemplo real de como um serviço de compartilhamento de

viagens consegue identificar o melhor carro para determinada corrida.

- Capítulo 10: Analisando dados de séries temporais – Apresenta algumas técnicas de análise que podemos aplicar a dados de séries temporais para extrairmos dados estatísticos significativos. Em termos específicos, os exemplos deste capítulo ilustram como a análise de dados de séries temporais pode ser aplicada aos dados do mercado de ações.
- Capítulo 11: Obtenção de insights a partir de dados – Explora estratégias para obter insights a partir de dados, a fim de tomar decisões embasadas. Por exemplo, você aprenderá como identificar associações entre produtos vendidos em um supermercado para determinar quais grupos de itens são frequentemente comprados juntos em uma única transação (vantajoso para recomendações e promoções).
- Capítulo 12: Aprendizado de máquina para análise de dados – Apresenta o uso da scikit-learn para tarefas sofisticadas de análise de dados. Você treinará modelos de aprendizado de máquina para classificar avaliações de produtos de acordo com as respectivas classificações por estrelas e a fim de predizer tendências no preço de uma ação.

CAPÍTULO 1

Conceitos básicos de dados

Dados representam conceitos distintos para pessoas diferentes: talvez um operador de bolsa considere dados as cotações das ações em tempo real, enquanto um engenheiro da NASA pode considerar dados os sinais emitidos por um veículo rover em Marte. No entanto, quando se trata de processamento e de análise de dados, é possível usar abordagens e técnicas iguais ou semelhantes em uma variedade de conjuntos de dados, independentemente de sua origem. O mais importante de tudo é como os dados são estruturados.

Neste capítulo, veremos uma introdução conceitual ao processamento e análise de dados. Primeiro, estudaremos as principais categorias de dados com as quais possivelmente lidaremos e, depois, abordaremos as fontes de dados comuns. Em seguida, analisaremos as etapas de um típico pipeline de processamento de dados (ou seja, o processo real de obtenção, preparação e análise de dados). Por último, examinaremos as vantagens inigualáveis do Python como ferramenta de ciência de dados.

Categorias de dados

Os programadores dividem os dados em três categorias principais: não estruturados, estruturados e semiestruturados. Em um pipeline de processamento de dados, a fonte de dados normalmente contém dados não estruturados; a partir dessa fonte, é possível criar conjuntos de dados estruturados ou semiestruturados para processamento posterior. Apesar disso, alguns pipelines usam dados estruturados desde o início. Por exemplo, um aplicativo que processa localizações geográficas pode receber dados estruturados diretamente de sensores GPS. Nas seções a seguir, vamos

explorar as três principais categorias de dados, bem como dados de séries temporais, um tipo especial de dados, que podem ser estruturados ou semiestruturados.

Dados não estruturados

Dados não estruturados são dados sem nenhuma estrutura organizacional ou esquema predefinido. É a forma mais comum de dados, e os exemplos usuais incluem imagens, vídeos, áudio e texto em linguagem natural. Para ilustrar isso, vejamos o seguinte demonstrativo de resultados financeiros de uma empresa farmacêutica:

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

Considera-se esse texto um dado não estruturado, pois as informações nele contidas não estão organizadas com um esquema predefinido. Pelo contrário, estão distribuídas aleatoriamente no demonstrativo. É possível reescrevermos esse demonstrativo de diversas maneiras enquanto transmitimos a mesma informação. Por exemplo:

```
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%.
```

Apesar da falta de estrutura, os dados não estruturados podem conter informações importantes, que podemos extrair e converter em dados estruturados ou semiestruturados por meio de etapas adequadas de transformação e análise. Por exemplo, as ferramentas de reconhecimento de imagem primeiro convertem a coleção de pixels em uma imagem em um conjunto de dados de um formato predefinido e, em seguida, analisam esses dados para identificar o conteúdo da imagem. Do mesmo modo, na seção a seguir, veremos como estruturar os dados extraídos de nosso demonstrativo de resultados financeiros.

Dados estruturados

Dados estruturados têm um formato predefinido que especifica como os dados são organizados. Em geral, esses dados são armazenados em um

repositório como um banco de dados relacional ou apenas em um arquivo *.csv* (valores separados por vírgula). Chamamos os dados inseridos nesse repositório de *registro*, e as informações contidas neles são organizadas em *campos* e devem chegar em uma sequência correspondente à estrutura esperada. Em um banco de dados, os registros com a mesma estrutura são agrupados logicamente em um contêiner chamado *tabela*. Um banco de dados pode ter inúmeras tabelas, e cada tabela pode ter uma estrutura definida de campos.

Existem dois tipos básicos de dados estruturados: numéricos e categóricos. *Dados categóricos* são aqueles que podem ser categorizados com base em características semelhantes; por exemplo, é possível categorizar carros por marca e modelo. Por outro lado, *dados numéricos* expressam informações em forma numérica, possibilitando operações matemáticas.

Não se esqueça de que os dados categóricos às vezes podem assumir valores numéricos. Por exemplo, imagine códigos postais ou números de telefone. Embora expressos em números, não faria sentido realizar operações matemáticas com esses dados, como encontrar a mediana de um código postal ou de um número de telefone.

Como podemos organizar a amostra de texto apresentada na seção anterior em dados estruturados? Estamos interessados nas informações específicas desse texto, como nomes de empresas, datas e preços de ações. Queremos apresentar essas informações preparadas em campos com o formato adequado, assim podemos inseri-las em um banco de dados:

```
Company:    ABC
Date:       yyyy-mm-dd
Stock:      nnnnn
```

Ao usar técnicas de *processamento de linguagem natural (PLN)*, disciplina que treina máquinas para entender texto legível por humanos, podemos extrair as informações adequadas para esses campos. Por exemplo, procuramos o nome de uma empresa identificando uma variável de dados categóricos, podendo ser apenas um dos muitos valores predefinidos, como Google, Apple ou GoodComp. Além disso, podemos identificar uma data ao comparar sua ordenação explícita com um conjunto de formatos de ordenação explícita, como `yyyy-mm-dd`. Em nosso exemplo, identificamos,

extraímos e apresentamos nossos dados no seguinte formato predefinido:

```
Company:   GoodComp
Date:      2021-01-07
Stock:     +8.2%
```

Para armazenar esse registro em um banco de dados, é melhor apresentá-lo como uma sequência de campos semelhante a uma linha. Desse modo, podemos reorganizar o registro como um objeto retangular de dados ou uma matriz 2D:

```
Company  | Date       | Stock
--------------------------
GoodComp |2021-01-07 | +8.2%
```

As informações que escolhemos extrair da mesma fonte de dados não estruturados dependem de nossos requisitos. Nosso exemplo de demonstrativo não apenas contém a alteração no valor das ações da GoodComp em determinada data, como também indica o motivo dessa alteração, na frase "*the company announced positive early-stage trial results for its vaccine (a empresa anunciou resultados promissores em relação aos testes iniciais de sua vacina)*". Considerando o demonstrativo a partir dessa perspectiva, é possível criar um registro com os seguintes campos:

```
Company:   GoodComp
Date:      2021-01-07
Product:   vaccine
Stage:     early-stage trial
```

Vamos comparar essas informações com o primeiro registro que extraímos:

```
Company:   GoodComp
Date:      2021-01-07
Stock:     +8.2%
```

Repare que esses dois registros contêm campos diferentes e, consequentemente, estruturas diferentes. Como resultado, devem ser armazenados em duas tabelas diferentes de banco de dados.

Dados semiestruturados

Nos casos em que a identidade estrutural da informação não está em conformidade com os exigentes requisitos de formatação, talvez seja necessário processar formatos de dados semiestruturados, que nos

possibilitam ter registros de diferentes estruturas dentro do mesmo contêiner (tabela ou documento de banco de dados). Como os dados não estruturados, os dados semiestruturados não estão vinculados a um esquema organizacional predefinido; apesar disso, ao contrário dos dados não estruturados, as amostras de dados semiestruturados apresentam certo grau de estrutura, geralmente na forma de tags autodescritivas ou outros marcadores.

Os formatos mais comuns de dados semiestruturados são XML e JSON. Vejamos como nosso demonstrativo com resultados financeiros fica no formato JSON:

```
{
  "Company": "GoodComp",
  "Date":    "2021-01-07",
  "Stock":   8.2,
  "Details": "the company announced positive early-stage trial results for its
vaccine."
}
```

Podemos identificar as principais informações que extraímos anteriormente do demonstrativo. Cada informação é acompanhada de uma tag descritiva, como `"Company"` ou `"Date"`. Graças às tags, a informação é organizada de modo semelhante à seção anterior, mas, agora, temos uma quarta tag, `"Details"`, acompanhada de um fragmento inteiro do demonstrativo original, que parece não estruturado. Neste exemplo, vemos como os formatos de dados semiestruturados podem acomodar partes de dados estruturados e não estruturados em um único registro.

Além disso, é possível inserir múltiplos registros de estrutura desigual no mesmo contêiner. Aqui, armazenamos os dois registros diferentes derivados de nosso exemplo do demonstrativo de resultados financeiros no mesmo documento JSON:

```
[
  {
    "Company": "GoodComp",
    "Date":    "2021-01-07",
    "Stock":   8.2
  },
  {
    "Company": "GoodComp",
    "Date":    "2021-01-07",
```

```
    "Product": "vaccine",
    "Stage":   "early-stage trial"
  }
]
```

Lembre-se da análise na seção anterior de que um banco de dados relacional, sendo um repositório de dados rigidamente estruturado, não pode acomodar registros de estruturas variadas na mesma tabela.

Dados de séries temporais

Uma série temporal é um conjunto de pontos de dados indexados ou ordenados em ordem cronológica. Inúmeros conjuntos de dados financeiros são armazenados como séries temporais devido ao fato de que esses dados geralmente consistem em observações em um período específico.

Os dados de séries temporais podem ser estruturados ou semiestruturados. Imagine que você está recebendo dados geográficos, provenientes dos registros de um dispositivo de rastreamento GPS de um táxi em intervalos de tempo regulares. Os dados podem apresentar o seguinte formato:

```
[
  {
    "cab": "cab_238",
    "coord": (43.602508,39.715685),
    "tm": "14:47",
    "state": "available"
  },
  {
    "cab": "cab_238",
    "coord": (43.613744,39.705718),
    "tm": "14:48",
    "state": "available"
  }
  ...
]
```

Um registro novo de dados chega a cada minuto, com as últimas coordenadas geográficas (latitude/longitude) de `cab_238`. Cada registro tem a mesma sequência de campos, e cada campo tem estrutura consistente de um registro para o outro, possibilitando que armazenemos esses dados de séries temporais como dados estruturados regulares em uma tabela de banco de dados relacional.

Agora, imagine que os dados cheguem em intervalos irregulares, o que geralmente acontece na prática, e que recebemos mais de um conjunto de coordenadas por minuto. A estrutura de entrada pode ser mais ou menos assim:

```
[
  {
    "cab": "cab_238",
    "coord": [(43.602508,39.715685),(43.602402,39.709672)],
    "tm": "14:47",
    "state": "available"
  },
  {
    "cab": "cab_238",
    "coord": (43.613744,39.705718),
    "tm": "14:48",
    "state": "available"
  }
]
```

Observe que o primeiro campo `coord` inclui dois conjuntos de coordenadas, por isso não é consistente com o segundo campo `coord`. Esses dados são semiestruturados.

Fontes de dados

Agora que já sabemos quais são as principais categorias de dados, e quanto às fontes a partir das quais podemos receber esses dados? Em termos gerais, os dados se originam de muitas fontes diferentes, incluindo textos, vídeos, imagens, sensores de dispositivos, entre outros. No entanto, do ponto de vista dos scripts Python que escreveremos, as fontes de dados mais comuns são:

- Interfaces de programação de aplicações (APIs)
- Páginas web
- Banco de dados
- Arquivos

Não se trata de uma lista abrangente ou restritiva; existem muitas outras fontes de dados. No Capítulo 9, por exemplo, aprenderemos a usar um smartphone como provedor de dados GPS para nosso pipeline de processamento de dados. Vamos utilizar especificamente um bot aplicativo

como intermediário para conectar o smartphone e nosso script Python.

Em termos técnicos, todas as opções enumeradas aqui exigem uma biblioteca Python correspondente. Por exemplo, antes de obter dados de uma API, é necessário instalar um wrapper Python para a API ou usar a biblioteca Requests do Python para criar requisições HTTP diretamente à API. Da mesma forma, para acessar dados de um banco de dados, é necessário instalar um conector no código Python que possibilite acessar tipos específicos de banco de dados.

Ainda que seja necessário fazer o download e instalar muitas dessas bibliotecas, algumas delas, usadas para carregar dados, são distribuídas com o Python por padrão. Por exemplo, para carregar dados de um arquivo JSON, é possível utilizar o pacote built-in do Python `json`.

Nos capítulos 4 e 5, analisaremos as fontes de dados com mais detalhes. Em especial, aprenderemos como carregar dados específicos, oriundos de diferentes fontes, em estruturas de dados de nosso script Python para processamento posterior. Por ora, analisaremos brevemente cada um dos tipos de fonte comuns mencionados na lista anterior.

APIs

Atualmente, talvez a forma mais comum de aquisição de dados seja por meio de uma API (intermediário de software que possibilita que duas aplicações interajam entre si). Conforme mencionado, para utilizar uma API no Python, talvez seja necessário instalar um wrapper para essa API na forma de uma biblioteca Python. Hoje em dia, o jeito mais comum de fazer isso é com o comando `pip`.

Nem todas as APIs têm o próprio wrapper Python, mas não necessariamente isso significa que não podemos chamá-las a partir do Python. Se uma API atende a requisições HTTP, podemos interagir com ela a partir do Python usando a biblioteca Requests. Com isso, temos acesso a milhares de APIs que podemos usar em nosso código Python a fim de solicitar conjuntos de dados para processamento posterior.

Ao escolhermos uma API para determinada tarefa, devemos levar em consideração o seguinte:

- Funcionalidade – Como muitas APIs viabilizam funcionalidades semelhantes, é necessário entender meticulosamente nossos requisitos. Por exemplo, muitas APIs possibilitam pesquisa na web a partir de scripts Python. No entanto, somente algumas possibilitam delimitar os resultados da pesquisa por data de publicação.
- Custo – Muitas APIs possibilitam usar as conhecidas *chaves de desenvolvedor* (developer key) que, apesar de gratuitas, têm determinadas limitações, como um número limitado de chamadas por dia.
- Estabilidade – Graças ao repositório Python Package Index (PyPI) (`https://pypi.org`), é possível empacotar uma API em um pacote `pip` e disponibilizá-lo publicamente. Por isso, existe uma API (ou diversas) para praticamente qualquer tarefa que possamos imaginar, mas nem todas são totalmente confiáveis. Felizmente, o repositório PyPI rastreia o desempenho e o uso de pacotes.
- Documentação – Via de regra, as APIs populares têm um site de documentação correspondente, assim podemos conferir todos os comandos da API com exemplos de uso. Como um bom exemplo, confira a página de documentação da API Nasdaq Data Link (também conhecida como Quandl) (`https://docs.data.nasdaq.com/docs/python-time-series`). É possível encontrar exemplos de como criar diferentes chamadas de séries temporais.

Muitas APIs retornam resultados em um dos seguintes três formatos: JSON, XML ou CSV. Podemos facilmente converter os dados, em qualquer um desses três formatos, em estruturas de dados incorporadas ou mais usadas com Python. Por exemplo, a API do Yahoo Finance obtém e analisa dados de ações e, em seguida, retorna as informações já convertidas em um DataFrame do Pandas, estrutura amplamente usada que veremos no Capítulo 3.

Páginas web

Páginas web podem ser estáticas ou geradas em tempo real, em resposta à interação de um usuário, caso em que podem conter informações de muitas fontes diferentes. Em ambas as situações, um programa pode ler uma página web e extrair partes dela. É possível recorrer ao chamado *web scraping* (raspagem de informações), e isso é permitido, desde que a página esteja disponível publicamente.

No Python, um típico cenário de scraping envolve duas bibliotecas: a Requests e a BeautifulSoup. A Requests busca o código-fonte da página e, em seguida, a BeautifulSoup cria uma *árvore de análise sintática*, conhecida também como parse tree, para a página, sendo uma representação hierárquica do conteúdo da página. Podemos pesquisar a árvore de análise sintática e extrair dados dela usando as expressões idiomáticas pythônicas. Por exemplo, vejamos o seguinte fragmento de uma árvore de análise sintática:

```
[<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>]
```

Podemos transformar facilmente esse fragmento na seguinte lista de itens dentro de um loop `for` em nosso script Python:

```
[
  {'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'}
]
```

Acabamos de ver um exemplo de transformação de dados semiestruturados em dados estruturados.

Banco de dados

Outra fonte comum de dados é um banco de dados relacional, estrutura que fornece um mecanismo para armazenar, acessar e manipular com eficiência dados estruturados. É possível buscar ou enviar uma parte dos dados para tabelas no banco de dados usando uma requisição SQL (Linguagem de Consulta Estruturada). Por exemplo, a seguinte requisição executada para tabela `employees` no banco de dados obtém a lista apenas dos programadores que trabalham no departamento de TI, sendo desnecessário fazer a busca na tabela inteira:

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

O Python tem um engine built-in de banco de dados, o SQLite. Outra opção é usar qualquer outro banco de dados disponível. Antes de acessar um banco de dados, precisamos instalar o software cliente do banco de dados em nosso ambiente.

Além dos bancos de dados convencionais rigidamente estruturados, nos últimos anos, a necessidade de armazenar dados heterogêneos e não estruturados em contêineres semelhantes a bancos de dados vem aumentando cada vez mais. Isso levou ao surgimento dos chamados bancos de dados *NoSQL* (não *SQL* ou *não apenas SQL*). Os bancos de dados NoSQL usam modelos de dados flexíveis, possibilitando armazenar grandes volumes de dados com o método *chave-valor*, em que é possível acessar cada parte dos dados com uma chave associada. Vejamos como ficaria nosso exemplo anterior do demonstrativo de resultados financeiros quando armazenado em um banco de dados NoSQL:

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

O demonstrativo é associado a uma chave de identificação, `26`. Talvez, pareça estranho armazenar todo o demonstrativo em um banco de dados. No entanto, lembre-se de que muitos registros possíveis podem ser extraídos de um único demonstrativo. Quando armazenamos o demonstrativo, isso nos possibilita a flexibilidade de extrair diferentes informações em momento posterior.

Arquivos

Arquivos podem conter dados estruturados, semiestruturados e não estruturados. Com a função built-in `open()` do Python, é possível abrir um arquivo para que possamos usar os dados em nosso script. No entanto, dependendo do formato dos dados (por exemplo, CSV, JSON ou XML), talvez seja necessário importar uma biblioteca correspondente, assim conseguimos executar as operações de leitura, gravação e/ou anexação.

Arquivos de texto simples não exigem uma biblioteca para serem processados e são meramente considerados sequências de linhas em Python. Por exemplo, vejamos a seguinte mensagem que um roteador Cisco pode enviar para um arquivo de log:

```
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%'
```

É possível ler linha por linha, procurando as informações necessárias. Desse modo, caso sua tarefa seja procurar mensagens com informações sobre a utilização da CPU e extrair números específicos dela, seu script deve identificar a última linha do trecho como uma mensagem a ser selecionada. No Capítulo 2, veremos um exemplo de como extrair informações específicas de dados de texto com técnicas de processamento de texto.

Pipeline de processamento de dados

Nesta seção, exploraremos a visão conceitual das etapas envolvidas no processamento de dados, também conhecido como pipeline de processamento de dados. As etapas usuais em um pipeline de processamento de dados são:

1. Aquisição
2. Limpeza
3. Transformação
4. Análise
5. Armazenamento

Conforme veremos, essas etapas nem sempre são óbvias. Em algumas aplicações, é possível combinar múltiplas etapas em uma ou desconsiderar algumas delas.

Aquisição

Antes de fazer qualquer coisa com os dados, é necessário coletá-los. Por isso, a aquisição de dados é a primeira etapa em qualquer pipeline de processamento de dados. Na seção anterior, aprendemos sobre os tipos mais comuns de fontes de dados. Algumas dessas fontes possibilitam que

carreguemos somente a parte necessária dos dados, conforme requisição.

Por exemplo, uma requisição para a API do Yahoo Finance exige que especifiquemos o ticker (identificador de ações) de uma empresa e um período de tempo para obtermos os preços das ações dessa empresa. Da mesma forma, a News API, que possibilita obter artigos de notícias, pode processar um conjunto de parâmetros para restringir a lista de artigos solicitados, incluindo a fonte e a data de publicação. Apesar desses parâmetros de qualificação, talvez seja necessário filtrar ainda mais a lista obtida. Ou seja, os dados podem exigir limpeza.

Limpeza

Limpeza de dados é o processo de detecção e correção de dados corrompidos ou imprecisos, ou a remoção de dados desnecessários. Em alguns casos, essa etapa não é necessária, pois os dados obtidos estão imediatamente prontos para análise. Por exemplo, a biblioteca yfinance (um wrapper Python para a API do Yahoo Finance) retorna dados de ações como um objeto DataFrame do Pandas, facilmente utilizável. Em geral, isso nos possibilita desconsiderar as etapas de limpeza e transformação e irmos direto para a etapa de análise de dados.

Contudo, se recorrermos a uma ferramenta de aquisição web scraper, com certeza precisaremos limpar os dados, já que, provavelmente, teremos fragmentos de marcação HTML com os dados do payload, conforme vemos a seguir:

```
6.\tThe development shall comply with the requirements of DCCâ\x80\x99s Drainage
Division as follows\r\n\r\n
```

Após a limpeza, o fragmento fica assim:

```
6. The development shall comply with the requirements of DCC's Drainage Division
as follows
```

Além da marcação HTML, o texto raspado via web scraper pode conter outros textos indesejados, como no exemplo a seguir, em que a frase *A View full text* é simplesmente um texto de hiperlink. Talvez seja necessário abrir este link para acessar o texto dentro dele:

```
Permission for proposed amendments to planning permission received on the 30th A
View full text
```

Podemos também recorrer à etapa de limpeza de dados para filtrar entidades específicas. Após requisitar um conjunto de artigos via News API, por exemplo, talvez seja necessário selecionar apenas os artigos no período especificado cujos títulos incluam dinheiro ou uma frase de porcentagem. Podemos considerar esse filtro uma etapa de limpeza de dados porque o objetivo é remover dados desnecessários e prepará-los para as operações de transformação e de análise de dados.

Transformação

A transformação de dados é o processo de alterar o formato ou a estrutura dos dados a fim de prepará-los para análise. Por exemplo, para extrairmos as informações de nossos dados de texto não estruturados GoodComp, como fizemos em “Dados estruturados”, é possível fragmentá-los em palavras ou *tokens* individuais. Desse modo, uma ferramenta de Reconhecimento de Entidade Nomeada (NER) consegue procurar as informações desejadas. Na extração de informações, uma *entidade nomeada* comumente representa um objeto do mundo cotidiano, como uma pessoa, uma organização ou um produto, que pode ser identificado por um substantivo próprio. Além do mais, existem entidades nomeadas que representam datas, porcentagens, termos financeiros e muito mais.

Muitas ferramentas de PLN lidam automaticamente com esse tipo de transformação. Após essa transformação, os dados fragmentados GoodComp ficariam assim:

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

Outras formas de transformação de dados são mais aprofundadas, pois os dados de texto são convertidos em dados numéricos. Por exemplo, se reunirmos uma coleção de artigos de notícias, podemos transformá-los executando a *análise de sentimentos*, técnica de processamento de texto que gera um número que representa as emoções expressas em um texto.

A análise de sentimento pode ser implementada com ferramentas como a SentimentAnalyzer, que pode ser encontrada no pacote `nltk.sentiment`. Uma típica saída de análise seria mais ou menos assim:

```
Sentiment  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
```

Agora, em nosso conjunto de dados, cada entrada inclui um número, como `0.9313`, representando o sentimento expresso no artigo correspondente. Com o sentimento de cada artigo expresso numericamente, é possível calcular o sentimento médio de todo o conjunto de dados, possibilitando determinar o sentimento geral em relação a um objeto de interesse, como uma determinada empresa ou produto.

Análise

Análise é a etapa mais importante no pipeline de processamento de dados. Nessa etapa, interpretamos os dados brutos e chegamos a conclusões que não são imediatamente óbvias.

Recorrendo mais uma vez ao nosso exemplo de análise de sentimento, talvez você queira analisar o sentimento em relação a uma empresa durante um período específico e associado ao preço das ações dessa empresa. Ou talvez você queira comparar os índices do mercado de ações, como o índice S&P 500, usando o sentimento expresso em uma ampla amostragem de artigos de notícias desse mesmo período. O fragmento a seguir exemplifica como seria um conjunto de dados, com os dados do índice S&P 500 na terceira coluna, ao lado do sentimento geral das notícias de determinado dia:

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

Como os valores de sentimento e de ações são expressos em números, podemos plotar dois gráficos correspondentes no mesmo gráfico para análise visual, conforme ilustrado na Figura 1.1.

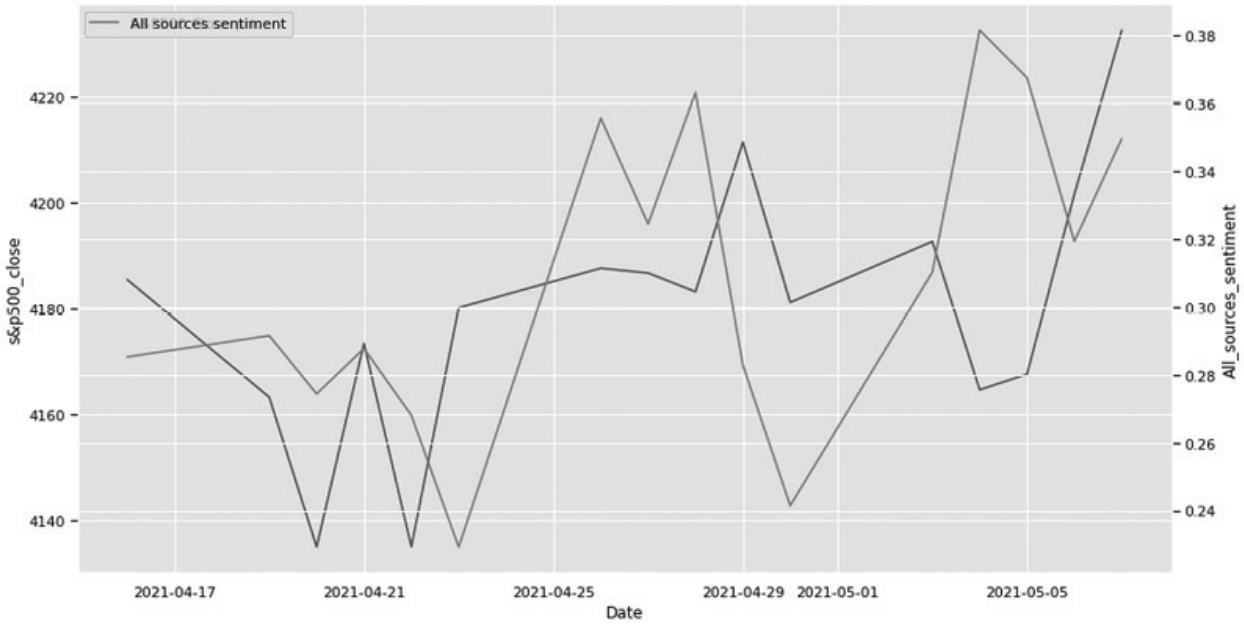


Figura 1.1: Exemplo de análise visual de dados.

A análise visual é um dos métodos comumente mais usados e eficientes para interpretar dados. No Capítulo 8, estudaremos em detalhes a análise visual.

Armazenamento

Na maioria dos casos, precisaremos armazenar os resultados gerados durante o processo de análise de dados a fim de disponibilizá-los para uso posterior. Via de regra, as opções de armazenamento incluem arquivos e banco de dados. Se for reutilizar seus dados com frequência, é preferível a última opção.

Abordagem pythônica

Quando se trata de ciência de dados com Python, espera-se que os códigos sejam escritos de modo *pythônico*. Ou seja: seus códigos devem ser concisos e eficientes. Não raro, códigos pythônicos são associados ao uso de *list comprehensions*, maneiras de implementar funcionalidades úteis de processamento de dados com uma única linha de código.

No Capítulo 2, abordaremos as list comprehensions mais detalhadamente. Por enquanto, vejamos o exemplo a seguir que ilustra como o conceito pythônico funciona na prática. Digamos que precisamos processar o seguinte fragmento

multissequencial de texto:

```
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'''
```

Em particular, precisamos dividir o texto em frases, criando uma lista de palavras individuais para cada frase, excluindo os sinais de pontuação. Graças ao recurso list comprehension do Python, é possível implementar tudo isso em uma única linha de código, o chamado *one-liner*:

```
word_lists = [[w.replace(',','') 1 for w in line.split() if w not in ['-']]
              2 for line in txt.replace('?','.').split('.')]
```

O loop `for line in txt` 2 divide o texto em frases e as armazena em uma lista. O loop `for w in line` 1 divide cada frase em palavras individuais e as armazena em uma lista dentro da lista maior. Como resultado, obtemos a seguinte lista de listas:

```
[['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"]]
```

Aqui, conseguimos executar duas etapas do pipeline de processamento de dados em uma única linha de código: limpeza e transformação. Ao removermos os sinais de pontuação do texto, limpamos os dados e, ao separarmos as palavras uma das outras a fim de criar uma lista de palavras para cada frase, executamos a etapa de transformação.

Caso programe em outra linguagem e esteja aprendendo Python, tente executar essa tarefa com a linguagem que sabe. Quantas linhas de código são necessárias?

Recapitulando

Após ler este capítulo, você será capaz de entender superficialmente as principais categorias de dados, as fontes de dados e como um típico pipeline de processamento de dados é organizado.

Como vimos, existem três categorias principais de dados: não estruturados, estruturados e semiestruturados. Normalmente, em um pipeline de processamento de dados, a matéria-prima de entrada são dados não estruturados, que passam por etapas de limpeza e de transformação para convertê-los em dados estruturados ou em dados semiestruturados a fim de que sejam usados na etapa de análise. Aprendemos também sobre pipelines de processamento de dados que usam dados estruturados ou semiestruturados desde o início, coletados a partir de uma API ou a partir de um banco de dados relacional.

CAPÍTULO 2

Estruturas de dados do Python

As estruturas de dados organizam e armazenam dados, facilitando acessá-los. O Python disponibiliza quatro estruturas de dados: listas, tuplas, dicionários e conjuntos. Como são fáceis de trabalhar, essas estruturas podem ser usadas com operações complexas de dados, o que faz do Python uma das linguagens mais populares para análise de dados.

Neste capítulo, veremos as quatro estruturas built-in de dados do Python, focando os recursos que possibilitam criar facilmente aplicações funcionais e centradas em dados com o mínimo de código. Aprenderemos também a combinar estruturas básicas com estruturas mais complexas, como uma lista de dicionários, para representar objetos cotidianos com mais acurácia. Aplicaremos esse conhecimento ao campo de processamento de linguagem natural e a uma breve introdução ao processamento de fotografias.

Listas

No Python, uma *lista* é uma coleção ordenada de objetos. Em uma lista, os elementos são separados por vírgulas, e toda lista é inserida entre colchetes, conforme mostrado a seguir:

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

Como as listas são mutáveis, é possível adicionar, remover ou modificar os elementos de uma lista. Ao contrário dos conjuntos, que analisaremos mais adiante neste capítulo, as listas podem ter elementos duplicados.

As listas contêm elementos que representam séries de coisas normalmente relacionadas e semelhantes que podem ser agrupadas de forma lógica. Uma típica lista contém apenas elementos pertencentes a uma única categoria (ou

seja, dados homogêneos, como nomes de pessoas, títulos de artigos ou números de participantes). Compreender esse conceito é essencial, ainda mais quando se trata de escolher a ferramenta adequada para a tarefa em questão. Caso precise de uma estrutura para objetos com propriedades diferentes, considere usar uma tupla ou um dicionário.

NOTA *Via de regra, mesmo que as listas sejam entendidas como homogêneas, o Python possibilita incluir elementos de diferentes tipos de dados na mesma lista. Por exemplo, a lista a seguir inclui strings e inteiros:*

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

Criando uma lista

Para criar uma lista básica, basta inserir uma sequência de elementos entre colchetes e atribuir a sequência a um nome de variável:

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

No entanto, na prática, as listas são normalmente preenchidas de modo dinâmico desde o início, muitas vezes com um loop que calcula um item por iteração. Nesses casos, o primeiro passo é criar uma lista vazia, conforme ilustrado a seguir:

```
regions = []
```

Após criarmos uma lista, podemos adicionar, remover e classificar itens dessa lista conforme necessário. É possível executar essas e outras tarefas com os diversos métodos de objeto de lista do Python.

Usando métodos comuns de objetos de lista

Métodos de objeto de lista são funções que implementam comportamentos específicos dentro de listas. Nesta seção, analisaremos alguns métodos comuns de objetos de lista, incluindo `append()`, `index()`, `insert()` e `count()`. Para praticar, vamos começar criando uma lista em branco. No decorrer do livro, conforme avançamos, vamos incrementá-la como uma lista de atividades com tarefas e organizá-la.

```
my_list = []
```

Talvez o método de objeto de lista mais comum seja o `append()`. Esse método

adiciona um item ao final da lista. É possível usar o `append()` para adicionar algumas tarefas à nossa lista de atividades, conforme mostrado a seguir:

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

Agora, a lista contém quatro itens, na ordem em que foram anexados:

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

Em uma lista, cada item contém uma chave numérica conhecida como *índice*. Esse recurso possibilita que a lista mantenha seus itens em uma ordem especificada. Como o Python usa indexação baseada em zero, o item inicial de uma sequência recebe o índice 0.

Para acessar um item individual de uma lista, especifique o nome da lista, seguido do índice do item desejado entre colchetes. Por exemplo, veja como exibir apenas o primeiro item de nossa lista de atividades:

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

A função `print()` gera a seguinte saída:

```
Pay bills
```

Podemos usar os índices de uma lista não somente para acessar um item necessário, como também para inserir um item novo em uma determinada posição na lista. Digamos que você queira adicionar uma tarefa nova à nossa lista de atividades entre passear com o cachorro e preparar o jantar. Para isso, primeiro use o método `index()` a fim de determinar o índice do item antes do qual quer inserir o item novo. Aqui, vamos armazená-lo na variável `i`:

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

Isso se tornará o índice para a tarefa nova, que agora podemos adicionar com o método `insert()`, como mostrado a seguir:

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

A tarefa nova é adicionada à lista no índice especificado, reduzindo todas as tarefas subsequentes em uma. Vejamos como a lista atualizada fica:

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

Como as listas permitem itens duplicados, talvez seja necessário verificar quantas vezes determinado item aparece em uma lista. Podemos fazer isso

com o método `count()`, conforme ilustrado no exemplo a seguir:

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

A função `print()` mostra apenas uma instância de `'Tidy up'` na lista, mas talvez seja uma boa ideia incluir esse item em nossa lista diária mais de uma vez!

NOTA *A lista completa de métodos de objeto de lista pode ser encontrada na documentação do Python em* https://docs.python.org/3/tutorial/datastructures.html.

Usando a notação de fatiamento

É possível acessar um intervalo de itens de um tipo sequencial de dados, como uma lista, usando o *fatiamento* ou slicing. Para obter a fatia de uma lista, especifique o índice da posição inicial e o índice da posição final mais 1. Separe os dois índices com dois-pontos e os insira entre colchetes. Por exemplo, podemos exibir os três primeiros itens de nossa lista de atividades assim:

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

O resultado é uma lista dos itens com índices de 0 a 2:

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

Os índices inicial e final em uma fatia são opcionais. Se omitirmos o índice inicial, a fatia começará no início da lista. Ou seja, a fatia no exemplo anterior pode ser alterada tranquilamente para:

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

Se omitirmos o índice final, a fatia continuará até o final da lista. Vejamos como exibir os itens com índices de 3 e maiores:

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

O resultado são os dois últimos itens da nossa lista de atividades:

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

Por último, podemos omitir os dois índices, caso em que obteremos uma cópia da lista inteira:

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

O resultado é:

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

A notação de fatiamento não se limita a extrair uma subsequência de uma lista. Em vez de usarmos os métodos `append()` e `insert()` para preencher uma lista com dados, podemos usar o fatiamento. Aqui, por exemplo, adicionamos dois itens ao final da nossa lista:

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

A função `len()` retorna o número de itens na lista, que também é o índice da primeira posição não utilizada fora da lista. É possível adicionar itens novos tranquilamente a partir desse índice. Vejamos como fica a lista agora:

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

Do mesmo jeito, podemos remover itens com o comando `del` e fatiando da seguinte maneira:

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

Esse comando remove os itens com índices 5 e maiores que 5, retornando a lista ao seu formato anterior:

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

Usando uma lista como uma fila

Uma *fila* é um tipo abstrato de dados que pode ser implementado com a estrutura de dados de lista. Uma extremidade de uma fila é sempre usada para inserir itens (*enqueue/enfileirar*) e a outra é utilizada para removê-los (*dequeue*/*desenfileirar*), seguindo assim a metodologia *first-in, first-out (FIFO: primeiro entrar, primeiro a sair)*. Na prática, a metodologia FIFO é bastante utilizada na armazenagem: os primeiros produtos que chegam ao armazém são os primeiros a sair. Organizar a venda de mercadorias dessa forma ajuda a evitar o vencimento dos produtos, garantindo que os mais antigos sejam os primeiros a serem vendidos.

É fácil transformar uma lista Python em uma fila com o objeto `deque` (abreviação de *double-ended queue [fila dupla]*). Nesta seção, exploraremos como esse objeto funciona usando nossa lista de atividades[1]. Para que uma lista funcione como uma fila, as tarefas concluídas devem ficar no início enquanto as tarefas novas devem aparecer no final da lista, conforme ilustrado na Figura 2.1.

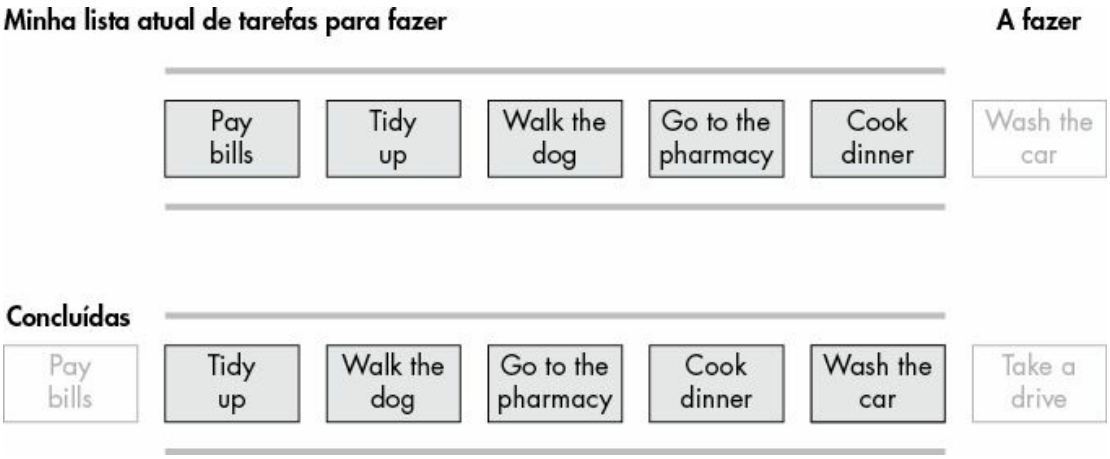


Figura 2.1: Exemplo de uma lista como uma fila.

Vejamos como implementar o processo mostrado na figura:

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

Neste script, primeiro transformamos o objeto `my_list` dos exemplos anteriores em um objeto `deque`, que faz parte do módulo `collections` do Python. O construtor de objeto `deque()` adiciona um conjunto de métodos ao objeto de lista que está sendo passado para ele, facilitando usar essa lista como uma fila. Neste exemplo específico, adicionamos um elemento novo ao lado direito da fila com o método `append()` e, em seguida, removemos um item do lado esquerdo da fila com o método `popleft()`. Esse método não apenas remove o item mais à esquerda, como também o retorna, fornecendo-o à nossa mensagem exibida. Devemos ver a seguinte mensagem como resultado:

```
Pay bills - Done!
```

Após ser reconvertido de um objeto `deque` para uma lista na última linha do script, a lista de atividades aparece assim:

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

Como podemos ver, o primeiro item foi removido da lista, enquanto um item novo foi anexado.

Usando uma lista como uma pilha

Assim como uma fila, uma *pilha* é uma estrutura abstrata de dados que podemos organizar na parte superior de uma lista. Uma pilha implementa a metodologia *last-in, first-out (LIFO, último a entrar, primeiro a sair),* em que o último item adicionado é o primeiro item acessado. Para que nossa lista de atividades funcione como uma pilha, devemos concluir as tarefas na ordem inversa, começando pela tarefa mais à direita. Vejamos como implementar esse conceito em 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')
```

No loop `for`, inserimos as tarefas da lista de atividades em uma pilha definida como outra lista, começando pela primeira tarefa. Este é um exemplo de uso do `append()` em um loop para preenchermos dinamicamente uma lista vazia. Assim, no loop `while`, removemos as tarefas da pilha, começando pela última. Fazemos isso com o método `pop()`, que remove o último item de uma lista e retorna o item removido. Vejamos a saída da pilha:

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

The stack is empty
```

Usando listas e pilhas para processamento de linguagem natural

Na realidade cotidiana, listas e pilhas têm muitos usos práticos, inclusive na área de PLN. Por exemplo, é possível usar listas e pilhas para extrair todos os chunks de substantivos de um texto. Um chunk de substantivo é formado de um substantivo e de seus filhos sintáticos à esquerda (ou seja, todas as palavras à esquerda do substantivo que são sintaticamente dependentes dele, como os termos determinantes da oração: artigos, adjetivos, numerais e

pronomes). Assim, para extrairmos os chunks de substantivo de um texto, devemos pesquisar no texto todos os substantivos e os filhos sintáticos à esquerda desses substantivos. Podemos implementar isso com um algoritmo baseado em pilha, conforme ilustrado na Figura 2.2.

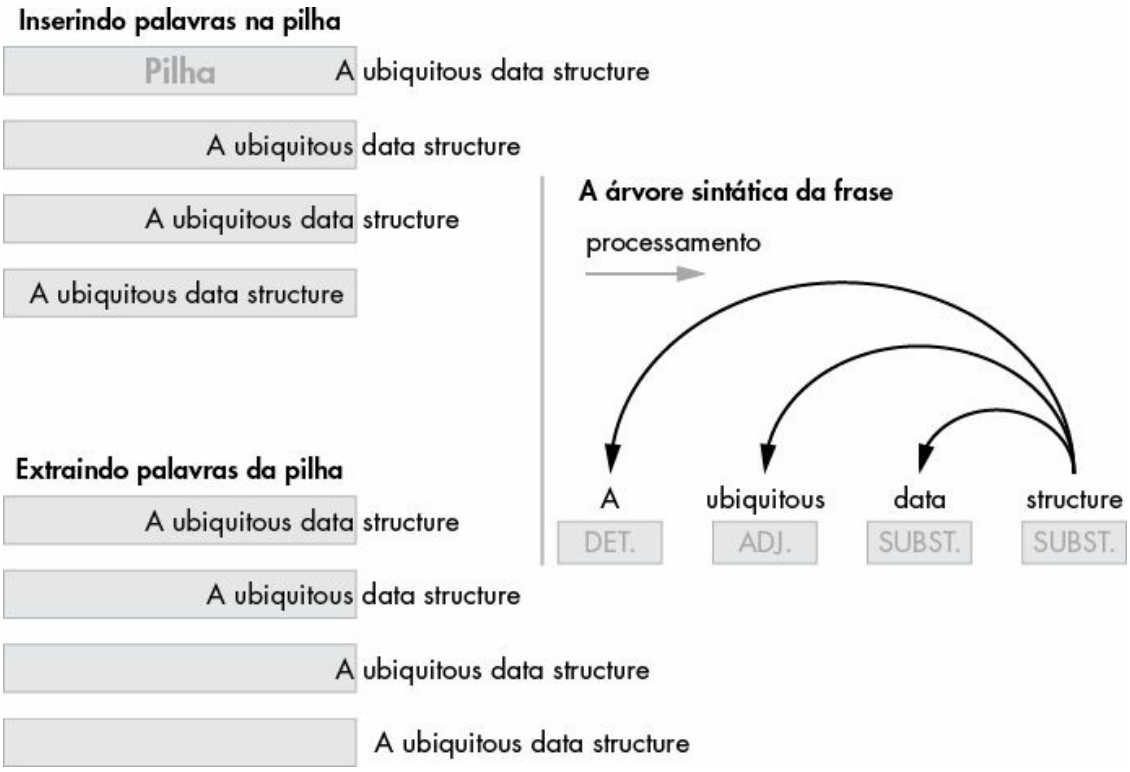


Figura 2.2: Exemplo de uso de uma lista como uma pilha.

A figura usa um único chunk de substantivo como exemplo, *A ubiquitous data structure*. Na árvore sintática, as setas à direita ilustram como as palavras *A*, *ubiquitous* e *data* são filhos sintáticos do substantivo *structure*, conhecido como *núcleo sintático* desses filhos sintáticos. O algoritmo analisa o texto da esquerda para a direita, uma palavra por vez, enviando a palavra para a pilha se for um substantivo ou o filho sintático à esquerda de um substantivo. Ao identificar uma palavra que não corresponde com essa descrição, ou se não houver palavras no texto, o algoritmo encontra um chunk inteiro de substantivo, que é extraído da pilha.

Para implementar esse algoritmo baseado em pilha a fim de extrair chunks de substantivos, é necessário instalarmos a spaCy, a principal biblioteca open source Python para processamento de linguagem natural, bem como um de seus modelos em língua inglesa. Usaremos os seguintes comandos:

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

O script a seguir implementa a spaCy:

```
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': 1
    stk.append(w.text)
  elif (w.head.pos_ == 'NOUN' or w.head.pos_ == 'PROPN') and (w in w.head.lefts):
2
    stk.append(w.text)
  elif stk: 3
    chunk = ''
    while stk:
      chunk = stk.pop() + ' ' + chunk 4
    print(chunk.strip())
```

As primeiras linhas do script passam pelo processo de configuração padrão visando analisar uma frase de texto com a spaCy. Basta importarmos a biblioteca spaCy, definirmos uma sentença a ser processada e carregar o modelo de língua inglesa da spaCy. Depois disso, aplicamos o pipeline `nlp` à sentença, instruindo a spaCy a gerar a estrutura sintática da sentença, necessária para tarefas como extração de chunks de substantivos.

NOTA *Para mais informações sobre a spaCy, confira a documentação em* https://space.io.

Em seguida, implementamos o algoritmo descrito anteriormente, iterando cada palavra do texto com um loop. Caso encontre um substantivo `1` ou um de seus filhos sintáticos à esquerda `2`, basta enviá-lo para a pilha com a operação `append()`. É possível criar essas definições com as propriedades *built-in da spaCy*, como `w.head.lefts`, que possibilita navegarmos pela estrutura sintática da sentença para encontrarmos as palavras desejadas nela. Assim, com `w in w.head.lefts`, procuramos o núcleo sintático de uma palavra (`w.head`), depois buscamos os filhos sintáticos à esquerda desse núcleo (`.lefts`) e determinamos se a palavra em questão é uma delas. Para exemplificar, ao

avaliar a palavra *ubiquitous*, `w.head` resultaria em *structure*, o núcleo sintático de *ubiquitous* e `.lefts` de *structure* geraria as palavras *a*, *ubiquitous* e *data*, demonstrando que *ubiquitous* é de fato um filho sintático à esquerda de *structure*.

Para finalizarmos o algoritmo, quando definimos que a próxima palavra no texto não faz parte do chunk de substantivo em questão (nem um substantivo nem um filho sintático à esquerda de um substantivo) `3`, temos um chunk completo de substantivos. Assim, podemos extrair as palavras da pilha `4`. O script a seguir encontra e gera os três chunks de substantivos:

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

Melhorias com list comprehensions

No Capítulo 1, vimos um exemplo de criação de uma lista com o recurso list comprehension. Nesta seção, recorreremos às list comprehensions para melhorar nosso algoritmo de extração de chunks de substantivos. Em geral, melhorar como uma solução exige aperfeiçoamentos significativos no código existente. No entanto, neste caso, como temos as list comprehensions, faremos aprimoramentos sucintos.

Ao analisarmos a árvore de dependência sintática da Figura 2.2, podemos perceber que cada elemento da frase ali representada está diretamente relacionado por um arco sintático ao substantivo *structure*. No entanto, um chunk de substantivo também pode seguir outro padrão, como algumas palavras não estarem interligadas ao substantivo da frase por uma relação sintática direta. A Figura 2.3 ilustra como seria a árvore de dependência de tal frase. Repare que o advérbio *most* é filho sintático do adjetivo *useful*, não do substantivo *type*, mas ainda faz parte do chunk de substantivo que tem *type* como núcleo sintático.

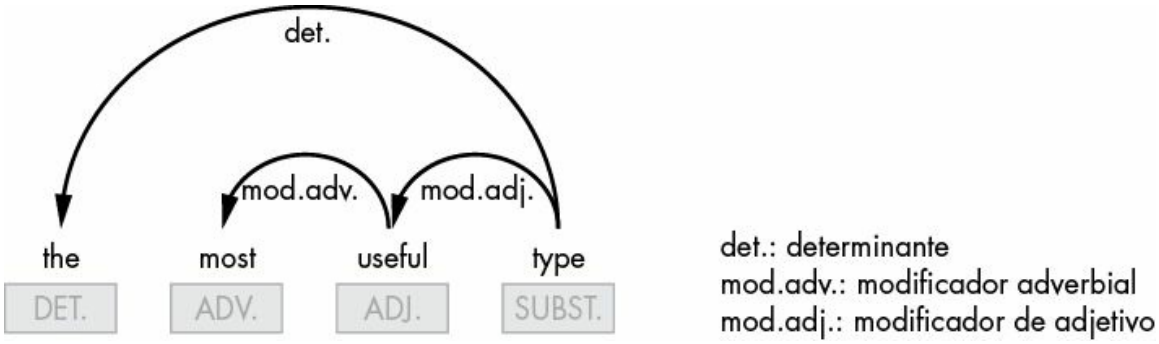


Figura 2.3: A árvore de dependência sintática de um chunk de substantivo mais complexo.

Precisamos melhorar o script da seção anterior para que também possamos extrair chunks de substantivos como mostrado na Figura 2-3, em que algumas palavras em um chunk não estão relacionadas diretamente ao substantivo da frase. Para refinar nosso algoritmo, primeiro vamos comparar as árvores de dependência sintática representadas nas figuras 2-2 e 2-3, assim podemos identificar o que têm em comum. Ambas as árvores têm uma semelhança importante: o núcleo sintático de cada palavra dependente em um chunk de substantivo pode ser encontrado à direita. No entanto, o substantivo que estrutura a frase pode não seguir esse padrão. Por exemplo, na frase “List is a ubiquitous data structure in the Python programming language (Lista é uma estrutura de dados onipresente na linguagem de programação Python)”, a palavra *structure* é o núcleo sintático do chunk de substantivo cujo núcleo, o verbo *is*, localiza-se à sua esquerda. Para ter certeza, executaremos o seguinte script que gera o núcleo sintático para cada palavra na frase:

```
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)
```

Nosso novo algoritmo precisa escanear um texto em busca de palavras cujos núcleos sintáticos estejam à direita, indicando, assim, possíveis chunks de substantivos. A ideia é criar uma espécie de matriz para uma sentença que indique se o núcleo sintático de uma palavra está à sua direita ou não. Para fins de legibilidade, podemos incluir na matriz as palavras como estão na sentença e cujos núcleos estão à direita, enquanto podemos substituir todas as

outras por zeros. Assim, para a seguinte frase:

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

Teríamos a seguinte matriz:

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

É possível criar essa matriz com uma list comprehension:

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

Aqui, iteramos as palavras da sentença enviada com um loop dentro da list comprehension, substituindo por zeros as palavras cujos núcleos sintáticos não estão à direita `1`.

Vejamos a lista gerada:

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

Podemos perceber que a lista contém um elemento a mais do que palavras na sentença. Na verdade, isso ocorre porque a spaCy divide o texto em tokens, que podem ser palavras ou sinais de pontuação. Na lista, o final `0` representa o ponto no final da sentença.

Agora, precisamos de uma forma de percorrer essa lista para localizarmos e extrairmos os chunks de substantivos. Será necessário criarmos uma série de fragmentos de texto, em que cada um deles começa em uma determinada posição e continua até o final do texto. No trecho a seguir, movemos palavra por palavra do início até o restante do texto, gerando uma matriz dos lados dos núcleos sintáticos em cada iteração:

```
for w in doc:
  head_lefts = [t.text if t in t.head.lefts else 0 for t in 1 doc[w.i:]]
  print(head_lefts)
```

No objeto `doc`, usamos a notação de fatiamento para obter o fragmento necessário `1`. Com esse mecanismo, podemos deslocar a posição mais à esquerda da fatia resultante uma palavra à direita e a cada iteração do loop

`for`. O código gera o seguinte conjunto de matrizes:

```
['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]
```

Devemos analisar cada fragmento a seguir, procurando o primeiro zero. As palavras até e incluindo zero podem ser potenciais chunks de substantivo. Vejamos o código:

```
for w in doc:
  head_lefts = [t.text if t in t.head.lefts else 0 for t in doc[w.i:]]
1 i0 = head_lefts.index(0)
  if i0 > 0:
  2 noun = [1 if t.pos_== 'NOUN' or t.pos_== 'PROPN' else 0 for t in
                  reversed(doc[w.i:w.i+i0 +1])]
    try:
    3 i1 = noun.index(1)+1
    except ValueError:
      pass
    print(head_lefts[:i0 +1])
  4 print(doc[w.i+i0 +1-i1])
```

Definimos `i0` igual a `head_lefts.index(0)` a fim de encontrarmos o índice do primeiro zero no fragmento `1`. Se houver múltiplos elementos zero, `head_lefts.index(0)` retorna o índice do primeiro elemento. Em seguida, verificamos se `i0 > 0` para eliminar fragmentos que não começam com um elemento head-left.

Usamos então outra list comprehension para processar os elementos dos chunks de substantivos a serem enviados à pilha. Nesta segunda list comprehension, procuramos um substantivo ou um nome próprio dentro de cada fragmento que poderia ser um possível chunk de substantivos.

Percorremos o fragmento com um loop na ordem inversa para, primeiro, coletar o substantivo ou o nome próprio que forma o chunk e que, portanto, deve aparecer na última posição do fragmento `2`. O que realmente enviamos para a lista quando encontramos qualquer substantivo ou nome próprio é `1`. Além disso, enviamos um `0` para todos os outros elementos. Desse modo, o primeiro `1` encontrado na lista indica a posição do substantivo principal no fragmento em relação ao final do fragmento `3`. Precisaremos dele quando calcularmos a fatia de texto que representa o chunk de substantivo `4`.

Por ora, basta gerarmos os fragmentos gerados com os substantivos encontrados dentro deles. Veremos a seguinte saída:

```
['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
```

Agora, podemos incorporar o código novo na solução apresentada na seção anterior. Juntando tudo, temos o seguinte script:

```
  txt = 'List is arguably the most useful type in the Python
                                        programming language.'
  import spacy
  nlp = spacy.load('en')
  doc = nlp(txt)
  stk = []
 1 for w in doc:
  2 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':
    3 stk.append(w.text)
    elif (i1 > 0):
    4 stk.append(w.text)
    elif stk:
      chunk = ''
      while stk:
      5 chunk = stk.pop() + ' ' + chunk
      print(chunk.strip())
```

Iteramos os tokens com um loop na sentença submetida `1`, gerando uma lista `head_lefts` em cada iteração `2`. Lembre-se de que essa lista é uma matriz contendo zeros para as palavras da sentença cujos núcleos sintáticos estão à esquerda. Usamos essas matrizes para identificar chunks de substantivos. Para cada chunk que identificamos, enviamos cada substantivo ou nome próprio à pilha `3`, assim como qualquer outra palavra que pertença ao chunk mas não seja um substantivo `4`. Ao chegarmos ao final do chunk, extraímos os tokens da pilha, formando uma frase `5`.

O script gera a seguinte saída:

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

NOTA *Se quiser aprender mais sobre processamento de linguagem natural, confira meu livro Natural Language Processing with Python and spaCy, também da No Starch Press (*https://nostarch.com/NLPPython*).*

Tuplas

Assim como uma lista, uma *tupla* é uma coleção ordenada de objetos. Ao contrário das listas, no entanto, as tuplas são imutáveis. Após criada, uma tupla não pode ser alterada. Os itens em uma tupla são separados por vírgulas e podem opcionalmente ser inseridos entre parênteses, conforme mostrado aqui:

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

Normalmente, as tuplas são utilizadas para armazenar coleções de dados heterogêneos; ou seja, dados de diferentes tipos, como marca, modelo e ano de um carro. Como exemplificado, as tuplas são bastante úteis quando precisamos de uma estrutura para armazenar as propriedades de um objeto do mundo cotidiano.

Uma lista de tuplas

No Python, é comum aninhar estruturas de dados umas dentro das outras. Por exemplo, é possível ter uma lista em que cada elemento é uma tupla, possibilitando atribuir mais de um atributo a cada elemento da lista. Digamos que você queira atribuir um horário de início para cada tarefa na lista de atividades que criamos anteriormente neste capítulo. Cada item da lista se tornará uma estrutura de dados por si só, composta de dois elementos: a descrição de uma tarefa e seu horário de início agendado.

Para implementar essa estrutura, as tuplas são a escolha ideal, uma vez que se destinam a coletar dados heterogêneos em uma única estrutura. Nossa lista de tuplas pode ser mais ou menos assim:

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

É possível criar uma lista de tuplas a partir das duas listas simples a seguir:

```
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']
```

Como podemos ver, a primeira lista é a `my_list` original e a segunda é uma lista contendo os horários de início correspondentes. O modo mais fácil de combiná-las em uma lista de tuplas é com uma list comprehension, conforme mostrado a seguir:

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

Dentro da list comprehension, utilizamos a função `zip()` do Python, que itera as duas listas simples de forma simultânea, combinando os horários e as tarefas correspondentes em tuplas.

Como nas listas, para acessar um item em uma tupla, basta especificarmos o índice do item entre colchetes após o nome da tupla. Contudo, perceba que as tuplas aninhadas em uma lista não recebem nomes. A fim de acessar um item

em uma tupla aninhada, primeiro é necessário especificar o nome da lista, depois o índice da tupla na lista e, por último, o índice do item na tupla. Por exemplo, para verificar o horário atribuído à segunda tarefa em nossa lista de atividades, usaríamos a seguinte sintaxe:

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

Essa sintaxe gera a seguinte saída:

```
8:30
```

Imutabilidade

Não podemos nos esquecer de um fato importante: as tuplas são imutáveis. Ou seja, não podemos modificá-las. Por exemplo, se tentarmos alterar o horário de início de uma de nossas tarefas:

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

Receberemos o seguinte erro:

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

Por serem imutáveis, as tuplas não são adequadas para armazenar valores de dados que precisam ser periodicamente atualizados.

Dicionários

No Python, *dicionário* é outra estrutura built-in de dados amplamente usada. Dicionários são coleções mutáveis e não ordenadas de *pares chave-valor*, em que cada *chave* é um nome exclusivo que identifica um item de dados, o *valor*. Um dicionário é delimitado por chaves. Cada chave é separada de seu valor por dois-pontos e os pares chave-valor são separados por vírgulas, conforme mostrado a seguir:

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

Como as tuplas, os dicionários são úteis para armazenar dados heterogêneos sobre objetos do mundo cotidiano. Conforme ilustrado pelo exemplo, os dicionários têm a vantagem adicional de atribuir um rótulo a cada item de dados.

Lista de dicionários

Como outras estruturas de dados, os dicionários podem ser aninhados dentro de outras estruturas. Ao implementarmos nossa lista de atividades como uma lista de dicionários, podemos ter o seguinte:

```
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'}
]
```

Ao contrário das tuplas, os dicionários são mutáveis. Ou seja, podemos alterar facilmente o valor em um par chave-valor:

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

Esse exemplo também ilustra como acessar valores em um dicionário: diferentemente de listas e de tuplas, usamos nomes de chave em vez de índices numéricos.

Adicionando dados a um dicionário com o método setdefault()

O método `setdefault()` viabiliza um modo prático de adicionar dados novos a um dicionário. Esse método aceita um par chave-valor como parâmetro. Se a chave especificada já existir, o método simplesmente retorna o valor atual dessa chave. Se a chave não existir, `setdefault()` insere a chave com o valor especificado. Vejamos um exemplo. Primeiro, criamos um dicionário chamado `car` que armazena o modelo `Jetta`:

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

Agora, tentaremos adicionar uma chave nova de `model` com um valor de `Passat` usando o método `setdefault()`:

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

Isso gera a seguinte saída, mostrando que o valor da chave de `model` permaneceu o mesmo:

```
Jetta
```

No entanto, se quisermos especificar uma chave nova, o método `setdefault()` insere o par chave-valor e retorna o valor:

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

A saída será a seguinte:

```
2022
```

Se exibirmos o dicionário inteiro:

```
print(car)
```

Veremos isto:

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

Como podemos verificar, o método `setdefault()` evita que tenhamos de verificar manualmente se a chave no par chave-valor que queremos inserir já está no dicionário. É possível tentar inserir facilmente um par chave-valor em um dicionário sem o risco de sobrescrever o valor de uma chave já existente.

Agora que sabemos como o método `setdefault()` funciona, vejamos um exemplo prático. Contar o número de ocorrências de cada palavra em uma frase de texto é uma tarefa comum em PLN. No exemplo a seguir, veremos como podemos fazer isso, com a ajuda de um dicionário, usando o método `setdefault()`. Segue o texto que vamos processar:

```
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.'''
```

A primeira etapa é remover a pontuação do texto. Sem essa etapa, `'languages'` e `'languages.'` seriam contabilizadas como duas palavras separadas. Aqui, removemos os pontos e vírgulas:

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

Em seguida, dividimos o texto em palavras e as inserimos em uma lista:

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

```
print(lst)
```

Segue a lista de palavras geradas:

```
['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']
```

Agora, podemos contar as ocorrências de cada palavra na lista. É possível fazer essa implementação com um dicionário usando o método `setdefault()` da seguinte forma:

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

Primeiro, criamos um dicionário vazio. Em seguida, adicionamos os pares chave-valor ao dicionário, usando as palavras da lista como as chaves. O método `setdefault()` define o valor inicial de cada chave como `0`. O valor é então acrescido em 1 para a primeira ocorrência de cada palavra, gerando uma contagem de 1. Nas ocorrências subsequentes dessa palavra, o método `setdefault()` manterá o valor da contagem anterior intacto, mas esse valor será incrementado em 1 com o operador `+=`, gerando, assim, uma contagem com maior acurácia.

Antes de gerar o dicionário, podemos ordenar as palavras pelo número de ocorrências:

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

Com o método `items()` do dicionário, é possível converter esse dicionário em uma lista de tuplas, em que cada tupla contém uma chave e seu valor. Assim, ao especificarmos `x[1]` em `lambda` para o parâmetro `key` da função `sorted()`, estamos ordenando conforme os itens nas tuplas com índice `1`, ou seja, os valores (contagem de palavras) do dicionário original. Vejamos o dicionário resultante:

```
{'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}
```

Carregando arquivos JSON em um dicionário

Com a ajuda de dicionários, é possível converter facilmente estruturas de dados Python em strings JSON e vice-versa. Vejamos como carregar uma string representando um documento JSON em um dicionário usando apenas o operador de atribuição:

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

Como podemos ver, esse dicionário tem uma estrutura complexa. O valor da chave `ShippingInstructions` propriamente dito é um dicionário, no qual o valor da chave `Address` é outro dicionário e o valor da chave `Phone` é uma lista de dicionários.

Podemos salvar o dicionário diretamente em um arquivo JSON com o módulo `json` do Python usando o método `json.dump()`:

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

Da mesma forma, é possível usar o método `json.load()` para carregar o conteúdo de um arquivo JSON diretamente em um dicionário Python:

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

Como resultado, obtemos o mesmo dicionário mostrado no início desta seção. No Capítulo 4, analisaremos em detalhes como trabalhar com arquivos.

Conjuntos

Um *conjunto* Python é uma coleção não ordenada de itens exclusivos. Em um conjunto, não são permitidos itens duplicados. Um conjunto é definido entre chaves e contém itens separados por vírgulas, conforme ilustrado a seguir:

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

Removendo duplicações de sequências

Como os itens de um conjunto devem ser exclusivos, os conjuntos são úteis quando precisamos remover itens duplicados de uma lista ou tupla. Suponha que uma empresa queira ver uma lista de seus clientes. Podemos obter essa lista derivando os nomes dos clientes a partir dos pedidos feitos por eles. Como um cliente pode ter feito mais de um pedido, a lista pode ter nomes duplicados. É possível remover as duplicações com a ajuda de um conjunto da seguinte forma:

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

Basta converter a lista original em um conjunto e depois voltar para uma lista. O construtor set exclui automaticamente as duplicações. A lista atualizada fica mais ou menos assim:

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

Essa abordagem tem um inconveniente: não preserva a ordem inicial dos elementos. Isso se deve ao fato de que um conjunto é uma coleção não ordenada de itens. Na verdade, se executarmos o código anterior duas ou três vezes, a ordem da saída poderá ser diferente a cada vez.

Para realizar a mesma operação sem perder a ordem inicial, use a função `sorted()` do Python, conforme mostrado a seguir:

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

Essa função ordena o conjunto pelos índices da lista original, preservando a ordem. A lista atualizada é a seguinte:

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

Executando operações comuns de conjunto

Objetos de conjunto apresentam métodos para executar operações matemáticas comuns em sequências, como uniões e interseções. Esses métodos possibilitam combinar facilmente conjuntos ou extrair os elementos compartilhados por múltiplos conjuntos.

Imagine que precise classificar uma enorme quantidade de fotos em grupos com base no que está nas fotos. Para automatizar essa tarefa, podemos começar com uma ferramenta de reconhecimento visual como a Clarifai API, que gera um conjunto de tags descritivas para cada foto. Os conjuntos de tags podem então ser comparados uns com os outros usando o método `intersection()`. Esse método compara dois conjuntos e cria um conjunto novo contendo todos os elementos presentes em ambos. Neste caso específico, quanto mais tags houver em ambos os conjuntos, mais semelhantes serão as duas imagens em relação ao seu tema.

Simplificando, o exemplo a seguir usa somente duas fotos. Ao utilizar os conjuntos correspondentes de tags descritivas, podemos determinar até que ponto o conteúdo das duas fotos coincide:

```
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.")
```

Nesse código, executamos a operação de interseção para localizar os itens compartilhados por ambos os conjuntos. Se o número de itens que os conjuntos têm em comum for igual ou superior a dois, pode-se concluir que as fotos têm conteúdo semelhante e, assim, podem ser agrupadas.

EXERCÍCIO #1: ANÁLISE APRIMORADA DE TAGS DE FOTOS

Nesta seção, você é incentivado a praticar o que aprendeu no capítulo. Ao continuar com o exemplo de conjuntos da seção anterior, você também precisará usar dicionários e listas.

No exemplo anterior, comparamos as tags descritivas de somente duas fotos, determinando suas tags comuns por interseção. Vamos aprimorar a funcionalidade do código para que seja possível processar um número arbitrário de fotos, agrupando-as em categorias geradas dinamicamente com base na interseção de tags.

Como entrada, suponha que você tenha a seguinte lista de dicionários, em que cada dicionário representa uma foto (claro que, se você quiser, pode criar a própria lista com muito mais itens). A lista de dicionários usados aqui está disponível para download no repositório GitHub que acompanha este livro, em https://github.com/pythondatabook/sources/blob/main/ch2/list_of_dicts.txt:

```
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'}
}
]
```

Sua tarefa é agrupar fotos com interseção de tags, salvando os resultados em um dicionário:

```
photo_groups = {}
```

Para tal, é necessário iterar com um loop todos os pares possíveis de fotos apresentadas na lista. É possível fazer essa implementação com um par aninhado de loops `for`, organizados da seguinte forma:

```
for i in range(1, len(l)):
  for j in range(i+1,len(l)+1):
    print(f"Intersecting photo {i} with photo {j}")
    # Implementa a interseção aqui, salvando os resultados em photo_groups
```

Você precisará implementar o corpo do loop interno para que execute a interseção entre `l[i]['tags']` e `l[j]['tags']` e crie um par novo chave-valor no dicionário `photo_groups` se o resultado da interseção não estiver vazio. A chave pode ser composta dos nomes dos tags na interseção, enquanto o valor é uma lista com os nomes dos arquivos correspondentes. Se a chave já existe para um determinado conjunto de tags de interseção, basta anexar os nomes dos arquivos correspondentes à lista de valores. Para implementar essa funcionalidade, você pode usar o método `setdefault()`.

Caso queira usar a lista de fotos anterior, acabará com os seguintes grupos:

```
{
'tableware_food': ['photo1.jpg', 'photo2.jpg'],
```

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

Se estiver usando o próprio conjunto de fotos com mais itens, poderá ver mais chaves e mais arquivos associados a cada chave no dicionário resultante.
A solução para este e todos os outros exercícios pode ser encontrada no repositório GitHub que acompanha este livro.

Recapitulando

Neste capítulo, vimos as quatro estruturas built-in de dados do Python: listas, tuplas, dicionários e conjuntos. Além disso, vimos uma ampla variedade de exemplos ilustrando como essas estruturas podem representar objetos do mundo cotidiano e aprendemos a combiná-las com estruturas aninhadas, incluindo uma lista de tuplas, uma lista de dicionários e um dicionário cujos valores são listas.

Exploramos também os recursos que possibilitam criar facilmente aplicações funcionais de análise de dados em Python. Por exemplo, aprendemos como usar as list comprehensions para criar listas novas a partir das existentes e como usar o método `setdefault()` para acessar e manipular com eficiência os dados em um dicionário. Por meio de exemplos, vimos como esses recursos são válidos para desafios comuns, como processamento de texto e análise de fotos.

---

1 N.T.: Segue a tradução das seguintes tarefas em nossa lista de atividades e na Figura 2.1: *Pay bills* (pagar as contas), *tidy up* (arrumação), *walk the dog* (passear com o cachorro), *go to the pharmacy* (ir à farmácia), *cook dinner* (preparar o jantar), *wash the car* (lavar o carro), *pay bills* (pagar as contas).

CAPÍTULO 3

Bibliotecas de ciência de dados do Python

O Python disponibiliza acesso a um ecossistema robusto de bibliotecas de terceiros, útil para análise e manipulação de dados. Neste capítulo, veremos três das bibliotecas de ciência de dados mais populares: NumPy, pandas e scikit-learn. Como veremos, muitas aplicações de análise de dados usam de modo extensivo essas bibliotecas, explícita ou implicitamente.

NumPy

A biblioteca NumPy, ou Numeric Python, é útil para trabalhar com *arrays*, estruturas de dados que armazenam valores do mesmo tipo de dados. Muitas outras bibliotecas do Python que executam cálculos numéricos dependem do NumPy.

Os arrays do NumPy, uma grade de elementos do mesmo tipo, são componentes fundamentais da biblioteca NumPy. Em um array do NumPy, os elementos são indexados por uma tupla de inteiros não negativos. Os arrays do NumPy são semelhantes às listas do Python, exceto que exigem menos memória e, em geral, são mais rápidos porque usam código C otimizado e pré-compilado.

Além disso, suportam *operações por elemento (element-wise)*, possibilitando executar operações aritméticas básicas em arrays inteiros com código compacto e legível. A operação por elemento é uma operação em dois arrays com as mesmas dimensões que gera outro array com as mesmas dimensões, em que cada elemento *i, j* é o resultado de um cálculo realizado nos elementos *i, j* dos dois arrays originais. A Figura 3.1 exemplifica uma

operação por elemento realizada com dois arrays do NumPy.

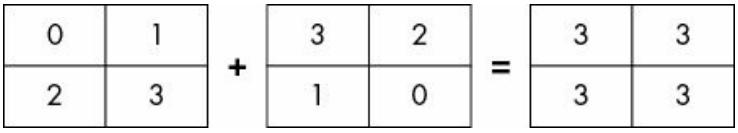


Figura 3.1: Adicionando dois arrays do NumPy.

Conforme se pode ver, o array resultante tem as mesmas dimensões que os dois arrays originais, e cada elemento novo é a soma dos elementos correspondentes nos arrays originais.

Instalando o NumPy

Como o NumPy é uma biblioteca de terceiros, não faz parte da biblioteca padrão do Python. A forma mais simples de instalá-lo é com o seguinte comando:

```
$ pip install NumPy
```

Como o Python considera o NumPy um módulo, é necessário importá-lo em nosso script antes de usá-lo.

Criando um array do NumPy

É possível criar um array do NumPy a partir dos dados de uma ou mais listas do Python. Suponha que tenha uma lista para cada funcionário de uma empresa contendo os pagamentos do piso salarial desses colaboradores nos últimos três meses. Podemos usar o seguinte código para obtermos todas as informações salariais em uma estrutura de dados:

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

Começamos importando a biblioteca NumPy `1`. Em seguida, definimos um conjunto de listas, em que cada lista contém os dados do piso salarial de um colaborador nos últimos três meses `2`. Por último, combinamos essas listas em um array do NumPy `3`. Vejamos como ficou o array:

```
[[2700 3000 3000]
```

```
[2600 2800 2800]
[2300 2500 2500]]
```

Temos um array 2D: tem dois eixos, indexados por números inteiros, começando com 0. Nas linhas do array, o eixo 0 é executado verticalmente para baixo, ao passo que, nas colunas, o eixo 1 é executado horizontalmente.

Podemos seguir o mesmo processo para criar um array contendo os bônus mensais dos colaboradores:

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

Executando operações por elemento

É fácil realizar operações por elementos em múltiplos arrays do NumPy com as mesmas dimensões. Por exemplo, podemos adicionar os arrays `base_salary` e `bonus` juntos a fim de determinarmos o valor total pago a cada mês para cada colaborador:

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

Como podemos ver, a operação de adição é um one-liner `1`. O conjunto resultante de dados também é um array do NumPy, no qual cada elemento é a soma dos elementos correspondentes nos arrays `base_salary` e `bonus`:

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

Usando funções estatísticas do NumPy

As funções estatísticas do NumPy possibilitam analisar o conteúdo de um array. Por exemplo, é possível encontrar o valor máximo de um array inteiro ou o valor máximo de um array ao longo de um determinado eixo.

Digamos que você queira encontrar o valor máximo no array `salary_bonus` que criamos na seção anterior. É possível fazer isso com a função `max()` do array do NumPy:

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

A função retorna o valor máximo pago nos últimos três meses a qualquer colaborador no conjunto de dados:

```
3400
```

O NumPy também pode encontrar o valor máximo de um array ao longo de um determinado eixo. Se quisermos determinar o valor máximo pago a cada colaborador nos últimos três meses, podemos usar a função `amax()` do NumPy, como mostrado a seguir:

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

Ao especificarmos `axis = 1`, instruímos `amax()` a pesquisar horizontalmente nas colunas por um máximo no array `salary_bonus`, aplicando, assim, a função em cada linha. Isso calcula o valor máximo mensal pago a cada colaborador nos últimos três meses:

```
[3400 3200 3000]
```

Da mesma forma, podemos calcular o valor máximo pago a cada mês a qualquer colaborador alterando o parâmetro `axis` para `0`:

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

O resultado é o seguinte:

```
[3200 3400 3400]
```

NOTA *Caso queira saber a lista de funções estatísticas suportadas pelo NumPy, confira a documentação do NumPy em* https://NumPy.org/doc/stable/reference/routines.statistics.html.

EXERCÍCIO #2: USANDO FUNÇÕES ESTATÍSTICAS DO NUMPY

Nos exemplos anteriores, cada conjunto de resultados retornado por `np.amax()`é um array do NumPy. Isso significa que podemos passar o conjunto de resultados obtidos para a função `amax()` novamente se quisermos encontrar o valor máximo no conjunto de resultados. Da mesma forma, é possível passar o conjunto de resultados para qualquer outra função estatística do NumPy, como `np.median()` ou `np.average()`. Faça um teste. Por exemplo, tente encontrar a média dos valores máximos pagos a cada mês a todos os colaboradores.

pandas

A biblioteca pandas é o padrão dominante para aplicações Python orientadas

a dados. (Caso esteja se perguntando, o nome é de alguma forma derivado de Python Data Analysis Library, Biblioteca de Análise de Dados do Python.) A biblioteca inclui duas estruturas de dados: a *Series*, que é 1D, e o *DataFrame*, que é 2D. Apesar de o DataFrame ser a principal estrutura de dados do pandas, na verdade, um DataFrame é uma coleção de objetos Series. Por isso, é importante entender as Series e os DataFrames.

Instalação do pandas

A distribuição padrão do Python não fornece o módulo pandas. Vamos instalá-lo com o seguinte comando:

```
$ pip install pandas
```

O comando `pip` também resolve as dependências da biblioteca, instalando implicitamente os pacotes NumPy, pytz e python-dateutil.

Assim como com o NumPy, é necessário importar o módulo pandas em nosso script antes de usá-lo.

Estrutura de dados Series do pandas

A Series do pandas é um array rotulado 1D. Por padrão, os elementos da Series são rotulados com números inteiros conforme sua posição, como em uma lista do Python. No entanto, podemos especificar rótulos personalizados. Por mais que esses rótulos não precisem ser exclusivos, devem ser do tipo hashable, como inteiros, floats, strings ou tuplas.

Os elementos de uma Series podem ser de qualquer tipo (inteiros, strings, floats, objetos Python e assim por diante), mas uma Serie funciona melhor se todos os seus elementos forem do mesmo tipo. Por fim, uma Serie pode se tornar uma coluna em um DataFrame maior, e é improvável que queiramos armazenar diferentes tipos de dados na mesma coluna.

Criando uma Series

Uma Series pode ser criada de diversas formas. Na maioria dos casos, fornecemos a uma Series algum tipo de conjunto de dados 1D. Vejamos como criar uma Series a partir de uma lista do Python:

```
1 import pandas as pd
```

```
2 data = ['Jeff Russell','Jane Boorman','Tom Heints']
3 emps_names = pd.Series(data)
  print(emps_names)
```

Começamos importando a biblioteca pandas e atribuindo-lhe um alias como `pd` 1. Em seguida, criamos uma lista de itens a serem usados como dados para a Series 2. Por último, criamos a Series, passando a lista para o método construtor `Series` 3.

Isso fornece uma única lista com índices numéricos definidos por padrão, a partir de 0:

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

O atributo `dtype` sinaliza o tipo dos dados subjacentes para a Series fornecida. Por padrão, o pandas usa o tipo de dados `object` para armazenar strings.

Podemos criar uma Series com índices definidos pelo usuário da seguinte forma:

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

Desta vez, os dados no objeto Series `emps_names` aparecem assim:

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

Acessando dados em uma Series

Para acessar um elemento em uma Series, basta especificarmos o nome da Series seguido pelo índice do elemento entre colchetes, como mostrado aqui:

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

Isso gera o elemento correspondente ao índice 9001:

```
Jeff Russell
```

Como alternativa, podemos usar a propriedade `loc` do objeto da Series:

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

Mesmo que usemos índices personalizados nesse objeto da Series, ainda é

possível acessarmos seus elementos por posição (ou seja, usar indexação baseada em localização de inteiros) por meio da propriedade `iloc`. Aqui, por exemplo, exibimos o primeiro elemento da Series:

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

Podemos acessar múltiplos elementos por seus índices com a operação de fatiamento, conforme analisado no Capítulo 2:

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

Isso gera a seguinte saída:

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

Repare que fatiar com `loc` inclui o ponto de extremidade correto (nesse caso, o índice 9002), ao passo que normalmente a sintaxe de fatia do Python não inclui.

É possível também usar o fatiamento para definir o intervalo de elementos por posição, e não por índice. Por exemplo, os resultados anteriores poderiam ser gerados com o seguinte código:

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

Ou simplesmente assim:

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

Como podemos verificar, ao contrário do fatiamento com `loc`, o fatiamento com `[]` ou `iloc` funciona do mesmo modo que o fatiamento usual do Python: a posição inicial é incluída, mas a parada não. Assim, `[0:2]` omite o elemento na posição 2 e retorna somente os dois primeiros elementos.

Combinando Series em um DataFrame

É possível combinar múltiplas Series para formar um DataFrame. Vamos tentar criar outra Series e combiná-la com a Series `emps_names`:

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

Para criar a Series nova, chamamos o construtor `Series()` `1`, passando os seguintes argumentos: a lista a ser convertida em uma Series, os índices da

Series e o nome da Series.

Além disso, precisamos nomear a Series antes de concatená-las em um DataFrame, porque esses nomes se tornarão os nomes das colunas correspondentes do DataFrame. Como não nomeamos a Series `emps_names` quando a criamos anteriormente, vamos nomeá-la aqui definindo sua propriedade `name` como `'names'` 2. Depois disso, podemos concatená-la com a Series `emps_emails` 3. Especificamos `axis=1` com o intuito de concatená-la ao longo das colunas.

Vejamos o DataFrame resultante:

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

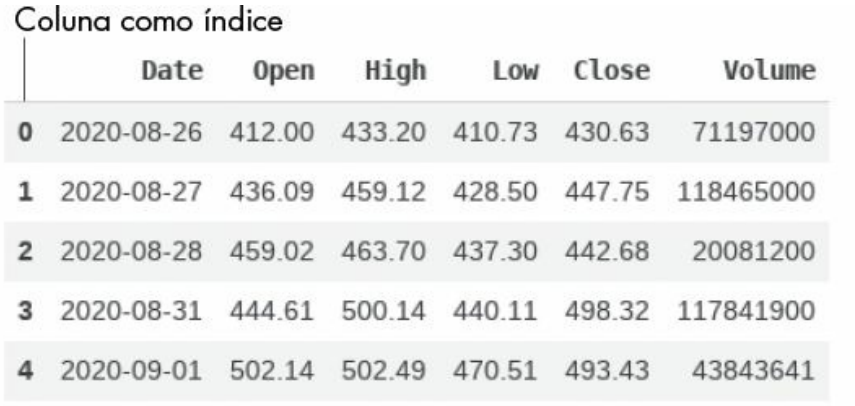
EXERCÍCIO #3: COMBINANDO TRÊS SERIES

Na seção anterior, criamos um DataFrame concatenando duas Series. Com essa mesma abordagem, tente criar um DataFrame a partir de três Series. Para tal, você precisará criar mais uma Series (digamos, `emps_phones`).

DataFrames do pandas

Um DataFrame do pandas é uma estrutura de dados rotulada em 2D com colunas que podem ser de diferentes tipos. Podemos considerar um DataFrame como um contêiner semelhante a um dicionário para objetos de Series, em que cada chave no dicionário é um rótulo de coluna e cada valor é uma Series.

Caso esteja familiarizado com bancos de dados relacionais, perceberá que um DataFrame do pandas é parecido com uma tabela comum do SQL. A Figura 3.2 ilustra um exemplo de um DataFrame do pandas.



	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

Figura 3.2: Exemplo de um DataFrame do pandas.

Observe que o DataFrame inclui uma coluna de índice. Como na Series, por padrão, o pandas usa a indexação numérica baseada em zero para DataFrames. No entanto, é possível substituir o índice padrão por uma ou mais colunas existentes. A Figura 3.3 mostra o mesmo DataFrame, mas com a coluna Data definida como índice.

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

Figura 3.3: DataFrame do pandas que usa uma coluna como índice.

Nesse exemplo específico, o índice é uma coluna do tipo `date`. Na verdade, o pandas possibilita que tenhamos índices DataFrame de qualquer tipo. Os tipos de índice mais usados são inteiros e strings. No entanto, não estamos limitados a usar apenas tipos simples. É possível definir um índice de um tipo de sequência, como uma lista ou uma tupla, ou até mesmo recorrer a um tipo de objeto que não seja do Python: pode ser um objeto de terceiros ou nosso próprio tipo de objeto.

Criando um DataFrame do pandas

Vimos como criar um DataFrame do pandas combinando múltiplos objetos da Series. Vimos também como criar um DataFrame carregando dados de um banco de dados, de um arquivo CSV, de uma requisição de API ou de outra fonte externa com um dos métodos *reader* da biblioteca pandas. Os readers nos possibilitam ler diferentes tipos de dados, como JSON e Excel, em um DataFrame.

Vejamos o DataFrame mostrado na Figura 3.2. Talvez tenha sido criado como resultado de uma requisição à API do Yahoo Finance por meio da biblioteca yfinance. Para criar o DataFrame, primeiro instale o yfinance com o `pip` da seguinte forma:

```
$ pip install yfinance
```

Em seguida, solicite os dados das ações, conforme a seguir:

```
  import yfinance as yf
1 tkr = yf.Ticker('TSLA')
2 hist = tkr.history(period="5d")
3 hist = hist.drop("Dividends", axis = 1)
  hist = hist.drop("Stock Splits", axis = 1)
4 hist = hist.reset_index()
```

Neste script, enviamos uma requisição à API a fim de obter os dados dos preços de ações para um determinado ticker `1` e usamos o método `history()` do yfinance a fim de especificar que queremos os dados do período de cinco dias `2`. Os dados resultantes, armazenados na variável `hist`, já estão na forma de um DataFrame do pandas. Não é necessário criar o DataFrame explicitamente; o yfinance faz isso nos bastidores. Após obter o DataFrame, removemos algumas de suas colunas `3` e mudamos para a indexação numérica `4`, gerando a estrutura mostrada na Figura 3.2.

Para definir o índice para a coluna Date, como mostrado na Figura 3.3, precisaremos executar a seguinte linha de código:

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

NOTA *O yfinance indexa automaticamente os DataFrames pela coluna Date. Nos exemplos anteriores, mudamos para indexação numérica e retomamos a indexação por data para ilustrar os métodos*

`reset_index()` e `set_index()`.

Agora, vamos tentar converter um documento JSON em um objeto pandas. Aqui, o conjunto de dados de amostra usado contém os dados salariais mensais de três colaboradores identificados por seus IDs na coluna `Empno`:

```
  import json
  import pandas as pd
  data = [
  {"Empno":9001,"Salary":3000},
  {"Empno":9002,"Salary":2800},
  {"Empno":9003,"Salary":2500}
  ]
1 json_data = json.dumps(data)
2 salary = pd.read_json(json_data)
3 salary = salary.set_index('Empno')
  print(salary)
```

Usamos o método reader do pandas `read_json()` a fim de passarmos uma string JSON para um DataFrame `2`. Simplificando, este exemplo usa uma string JSON convertida a partir de uma lista pelo método `json.dumps()` `1`. Como alternativa, podemos passar um objeto `path`, a fim de que o reader aponte para um arquivo JSON de interesse, ou um URL para uma API HTTP que publique dados no formato JSON. Por último, definimos a coluna `Empno` como índice do DataFrame `3`, substituindo, assim, o índice numérico padrão.

Vejamos o DataFrame resultante:

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

Outra prática comum é criar DataFrames do pandas a partir das estruturas padrão de dados do Python apresentadas no capítulo anterior. Por exemplo, vejamos como criar um DataFrame a partir de uma lista de listas:

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

```
4 emps = emps.set_index('Empno')
  print(emps)
```

Primeiro, inicializamos uma lista de listas com os dados a serem enviados para o DataFrame `1`. Cada lista aninhada se tornará uma linha no DataFrame. Em seguida, criamos explicitamente o DataFrame, definindo as colunas a serem utilizadas `2`. Depois, usamos um dicionário, `column_types`, para alterar os tipos de dados definidos para as colunas por padrão `3`. Por mais que seja opcional, talvez essa etapa seja imprescindível, caso esteja planejando fazer o join de um DataFrame com outro. Isso ocorre, pois é possível unir dois DataFrames apenas em colunas com o mesmo tipo de dados. Por último, definimos a coluna `Empno` como o índice do DataFrame `4`. Vejamos o DataFrame resultante:

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

Perceba que os DataFrames `emps` e `salary` usam `Empno` como coluna de índice para identificar exclusivamente cada linha. Em ambos os casos, definimos essa coluna como índice do DataFrame para simplificar o processo de merge dos dois DataFrames em um único, que analisaremos na próxima seção.

Combinando DataFrames

O pandas possibilita fazer o merge (ou o join) de DataFrames, do mesmo modo que podemos fazer o join de diferentes tabelas em um banco de dados relacional. Isso viabiliza reunir dados para análise. Os DataFrames suportam operações no estilo de banco de dados por meio de dois métodos: `merge()` e `join()`. Apesar de esses métodos terem alguns parâmetros diferentes, é possível usá-los mais ou menos de forma intercambiável.

Para começar, faremos o join dos DataFrames `emps` e `salary` definidos na seção anterior. Temos um exemplo de *join um-para-um*, pois uma linha em um DataFrame está associada a uma única linha no outro DataFrame. A Figura 3.4 ilustra como isso funciona.

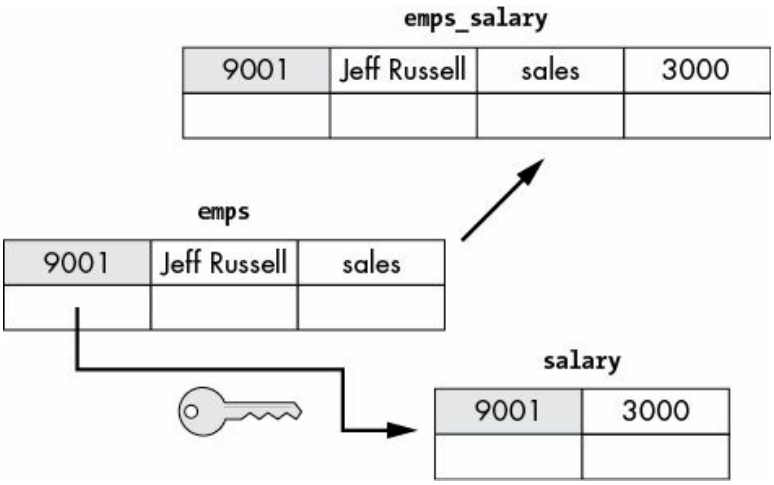


Figura 3.4: Join de dois DataFrames em um relacionamento um-para-um.

Aqui, vemos uma entrada do DataFrame `emps` e uma do DataFrame `salary`. As entradas compartilham o mesmo valor de índice, `9001`. Assim, podem ser unidas em uma única entrada no novo DataFrame `emps_salary`. Na terminologia de banco de dados relacional, as colunas por meio das quais as tabelas se relacionam são conhecidas como *colunas-chave*. Embora o pandas use o termo *índice* para essas colunas, a Figura 3.4 usa o ícone de chave para associação visual.

Com a ajuda do método `join()`, a implementação é bastante simples:

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

O método `join()` é desenvolvido para unirmos facilmente DataFrames com base em seus índices. Nesse exemplo específico, nem precisamos fornecer nenhum parâmetro adicional para unir esses dois DataFrames; o join deles em uma relação índice-a-índice é o comportamento padrão.

Vejamos como fica o conjunto resultante de dados:

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

Na prática, talvez seja necessário fazer o join de dois DataFrames, mesmo que um deles tenha linhas sem correspondências no outro DataFrame. Suponha que você tenha mais uma linha no DataFrames `emps` e que não haja

nenhuma linha correspondente no DataFrame `salary`:

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

Aqui, criamos um objeto pandas da Serie e o adicionamos ao DataFrame `emps` com o método `append()`. É uma forma comum de adicionar linhas novas a um DataFrame.

O DataFrame `emps` atualizado será o seguinte:

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

Se usarmos a operação de join novamente, teremos:

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

O resultado é o seguinte DataFrame:

```
               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
```

Repare que a linha adicionada ao DataFrame `emps` aparece no conjunto de dados resultante, mesmo que não tenha uma linha relacionada no DataFrame `salary`. A entrada `NaN` no campo Salary da última linha indica que o valor do salário está ausente. Em alguns casos, talvez você permita linhas incompletas como essas, mas, em outros casos, talvez queira excluir as linhas que não têm uma linha relacionada no outro DataFrame.

Por padrão, o método `join()` usa o índice do DataFrame chamado no DataFrame associado resultante, realizando, assim, um *left join*. Nesse exemplo, o DataFrame chamado é o `emps`. Como é considerado o DataFrame esquerdo na operação de join, todas as linhas dele são incluídas no conjunto resultante de dados. É possível alterarmos esse comportamento padrão passando o parâmetro `how` para o método `join()`. Esse parâmetro assume os

seguintes valores:

- `left` Usa o índice do DataFrame chamado (ou outra coluna se o parâmetro `on` for especificado), retornando todas as linhas do DataFrame (à esquerda) e somente as linhas correspondentes do outro DataFrame (à direita).
- `right` Usa o índice do outro DataFrame (à direita), retornando todas as linhas desse DataFrame e apenas as linhas correspondentes do DataFrame chamado (à esquerda).
- `outer` Forma a combinação do índice do DataFrame chamado (ou outra coluna se o parâmetro `on` for especificado) e o índice do outro DataFrame, retornando todas as linhas de ambos os DataFrames.
- `inner` Forma a interseção do índice do DataFrame chamado (ou outra coluna se o parâmetro `on` for especificado) com o índice do outro DataFrame, retornando somente as linhas cujos índices aparecem em ambos os DataFrames.

A Figura 3.5 exemplifica cada tipo de join.

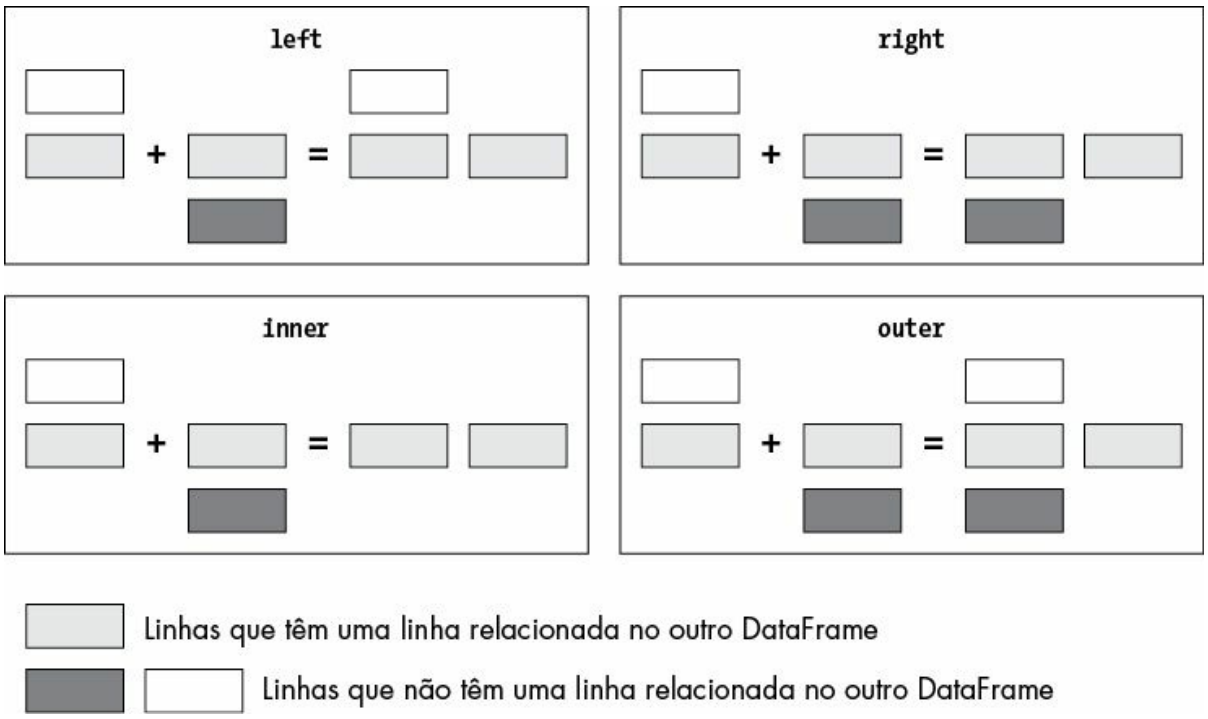


Figura 3.5: Os resultados dos diferentes tipos de join.

Caso queira que o DataFrame resultante inclua apenas as linhas de `emps` que têm uma linha relacionada no DataFrame `salary`, defina o parâmetro `how` de

`join()` como `inner`:

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

Vejamos o DataFrame resultante:

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

Outra forma de obter esse resultado é passar `right` como o parâmetro de `how`. Nesse caso, `join()` retorna todas as linhas do DataFrame `salary`, anexando a elas os campos das linhas correspondentes do DataFrame `emps`. Todavia, é importante observar que um right join não será igual a um inner join em diversas situações. Por exemplo, se adicionarmos uma linha ao DataFrame `salary` sem correspondência no DataFrame `emps`, o right join incluirá essa linha com as linhas que têm correspondências em `emps`.

NOTA *Para saber mais detalhes sobre como fazer joins em DataFrames, confira a documentação do pandas em* https://pandas.pydata.org/pandas-docs/stable/reference/frame.html#combining-comparing-joining-merging.

EXERCÍCIO #4: USANDO DIFERENTES JOINS

É importante compreender como diferentes valores do parâmetro `how` passado para o método `join()` afetam o DataFrame resultante.

Adicione uma linha nova ao DataFrame `salary` para que o valor de `Empno` nessa linha não possa ser encontrado no DataFrame `emps`. (Na seção anterior, vimos como adicionar uma linha a um DataFrame com um objeto da Series.) Depois disso, faça o join do DataFrame `emps` com o DataFrame `salary` para que o novo DataFrame inclua somente as linhas de `emps` com correspondências no DataFrame `salary`. Em seguida, faça o join de `emps` com o `salary` mais uma vez, de modo que o DataFrame novo inclua todas as linhas de ambos os DataFrames.

Joins de um-para-muitos

Em um *join de um-para-muitos*, uma linha de um DataFrame pode corresponder a múltiplas linhas de outro DataFrame. Imagine a situação em que cada vendedor no DataFrame `emps` processou uma série de pedidos. Podemos retratar esse cenário em um DataFrame `orders` da seguinte forma:

```
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)
```

Vejamos as `orders` do DataFrame:

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

Agora que temos um DataFrame de pedidos, podemos combiná-lo com o DataFrame de colaboradores definido anteriormente. Trata-se de um join um-para-muitos, pois um colaborador do DataFrame `emps` pode ser associado a múltiplas linhas no DataFrame `orders`:

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

Nesse código, recorremos ao método `merge()` para definir um join um-para-muitos, combinando os dados dos Dataframes `emps` e `orders`. O método `merge()` possibilita especificar as colunas a serem unidas em ambos os DataFrames, usando `left_on` para especificar a coluna no DataFrame chamado e `right_on` para a coluna no outro DataFrame. Com `join()`, apenas conseguimos especificar a coluna a ser unida com o DataFrame chamado. Para o outro DataFrame, o `join()` usa a coluna de índice.

Nesse exemplo, usamos o tipo inner join para incluir somente linhas relacionadas de ambos os DataFrames. Vejamos como fica o conjunto resultante de dados:

```
       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
```

A Figura 3.6 exemplifica como funciona o join de um-para-muitos.

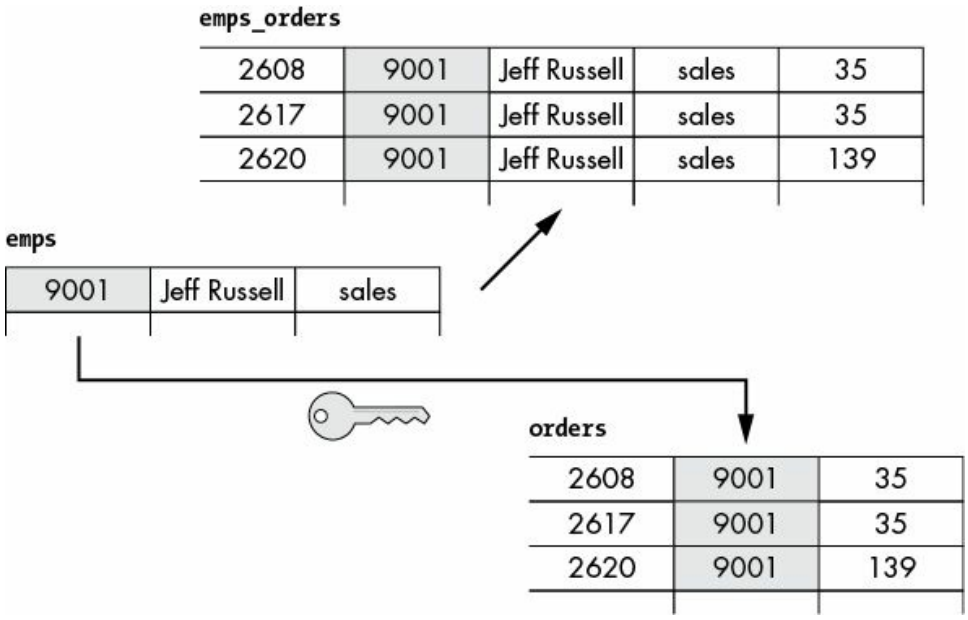


Figura 3.6: Join de dois DataFrames em um relacionamento um-para-um.

Na figura, como podemos notar, o join de um-para-muitos inclui uma linha para cada linha no conjunto de dados ao lado de *muitos* joins. Como estamos usando o tipo inner join, nenhuma outra linha será incluída. Mas, no caso de um left join ou de um outer join, o join também incluiria as linhas do conjunto de dados em *um* lado do join que não têm correspondências com *muitos* lados.

Além dos joins um-para-muitos e um-para-um, há também os *joins muitos-para-muitos*. Vejamos um exemplo desse relacionamento. Imagine dois conjuntos de dados: um que lista livros e um que lista autores. No conjunto de dados de autores, cada registro pode ser vinculado a um ou mais registros no conjunto de dados de livros e, no conjunto de dados de livros, cada registro pode ser vinculado a um ou mais registros no conjunto de dados de autores. No Capítulo 7, analisaremos esse tipo de relacionamento, pois veremos com mais detalhes as operações de concatenar, de merge e de join de dados.

Agregando dados com groupby()

A função `groupby()` do pandas possibilita agregar dados em múltiplas linhas de um DataFrame. Por exemplo, essa função pode encontrar a soma de uma coluna ou obter a média de um subconjunto de valores em uma coluna.

Imagine que precisa calcular o total médio para os pedidos processados por cada colaborador no DataFrame `orders` que criamos anteriormente. Podemos utilizar a função `groupby()` da seguinte maneira:

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

A função `groupby()` retorna um objeto GroupBy que suporta diversas funções de agregação. Nesse exemplo específico, usamos `mean()` para calcular um total médio para o grupo de pedidos associados a um colaborador. Para fazer isso, agrupamos as linhas no DataFrame `orders` pela coluna `Empno` e, em seguida, aplicamos a operação `mean()` à coluna `Total`. O conjunto de dados gerado é um objeto da Series:

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

Agora, vamos supor que você queira somar os totais dos pedidos dentro dos grupos. Neste momento, a função `sum()` do objeto GroupBy é bastante útil:

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

Vejamos os dados gerados no objeto da Series:

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

NOTA *Para saber mais sobre as funções suportadas pelo objeto GroupBy, confira a referência da API do pandas em* https://pandas.pydata.org/pandas-docs/stable/reference/groupby.html.

scikit-learn

A scikit-learn é um pacote Python desenvolvido para aplicações de aprendizado de máquina (machine learning). Com a NumPy e o pandas, é outro componente fundamental no ecossistema Python de ciência de dados. O scikit-learn fornece ferramentas eficientes e fáceis que podem ser usadas com problemas comuns de aprendizado de máquina, incluindo análise exploratória e preditiva de dados. No final do livro, nos aprofundaremos mais em aprendizado de máquina. Por ora, nesta seção, veremos uma pequena amostra

de como o Python pode ser usado na área de aprendizado de máquina, sobretudo para análise preditiva de dados.

A análise preditiva de dados é uma área do aprendizado de máquina que depende de algoritmos de classificação e de regressão. A classificação e a regressão usam dados históricos para efetuar predições em relação a dados novos, mas a classificação ordena os dados em categorias discretas, ao passo que a regressão pode gerar um intervalo consecutivo de valores numéricos. Nesta seção, analisaremos um exemplo de classificação implementada com o scikit-learn. Vamos criar um modelo preditivo para analisar as avaliações de produtos do cliente e ordená-las em duas classes: avaliações positivas e avaliações negativas. A partir de amostras já classificadas, o modelo aprenderá como predizer a classe de outras amostras. Após treinarmos o modelo, mostraremos algumas avaliações novas para classificação como positivas ou negativas.

Instalando o scikit-learn

Como o NumPy e o pandas, o scikit-learn é uma biblioteca Python de terceiros. É possível instalá-lo da seguinte forma:

```
$ pip install sklearn
```

O pacote scikit-learn tem muitos submódulos, cada um com a própria funcionalidade específica. Desse modo, é comum importar apenas o submódulo necessário para uma tarefa específica (digamos, `sklearn.model_selection`) em vez do módulo inteiro.

Obtendo um conjunto de dados de amostra

Para fins de acurácia, devemos treinar um modelo preditivo em um grande número de amostras rotuladas. Ou seja, nosso primeiro passo para criar um modelo que possa classificar as avaliações de produtos é obter um conjunto de avaliações que já estão rotuladas como positivas ou negativas. Isso evita que tenhamos de coletar avaliações e rotulá-las manualmente.

Existem inúmeras fontes online para conjuntos de dados rotulados. Uma das melhores é o Machine Learning Repository da UC Irvine, em https://archive.ics.uci.edu/ml/index.php. No repositório, basta pesquisar

“customer product reviews” e encontraremos um link para o Sentiment Labelled Sentences Data Set (Conjunto de Dados de Sentenças Rotuladas Com Análise de Sentimentos, em tradução livre) ou, caso prefira, basta acessar https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences). Faça o download e descompacte o arquivo *“sentiments.zip” rotulado como sentimento* na página do repositório.

> **NOTA** *O conjunto de dados Sentiment Labelled Sentences foi criado para o artigo “From Group to Individual Labels Using Deep Features (De Rótulos de Grupo a Rótulos Individuais Usando Features Profundas, em tradução livre)”, de Dimitrios Kotzias et al.*

O arquivo *.zip* contém avaliações do IMDb, da Amazon e do Yelp, em três arquivos *.txt* diferentes. As avaliações são rotuladas com sentimento positivo ou negativo (`1` ou `0`, respectivamente); há 500 avaliações positivas e 500 negativas de cada fonte, um total de 3.000 avaliações rotuladas em todo o conjunto de dados. Para simplificar, usaremos somente as instâncias da Amazon, que podem ser encontradas no arquivo *amazon_cells_labelled.txt*.

Carregando o conjunto de dados de amostra em um DataFrame do pandas

Para simplificar cálculos adicionais, é necessário carregar as avaliações do arquivo de texto em uma estrutura de dados mais gerenciável. É possível ler os dados do *amazon_cells_labelled.txt* em um DataFrame do pandas da seguinte forma:

```
import pandas as pd
df = pd.read_csv('/usr/Downloads/sentiment labelled
sentences/amazon_cells_labelled.txt',
                 names=[1 'review', 2 'sentiment'], 3 sep='\t')
```

Aqui, recorremos ao método reader do pandas `read_csv()` para carregar os dados em um DataFrame. Especificamos duas colunas: a primeira para armazenar as avaliações `1` e a segunda para as pontuações de sentimento correspondentes `2`. No arquivo original, como tabulações separam as avaliações de suas pontuações de sentimento correspondentes, especificamos `\t` como separador `3`.

Dividindo o conjunto de dados de amostra em um conjunto de treinamento e em um conjunto de teste

Agora que importamos o conjunto de dados, o próximo passo é dividi-lo em duas partes: uma para treinar o modelo preditivo e outra para testar sua acurácia. Ao usar o scikit-learn, fazemos isso com apenas algumas linhas de código:

```
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, 1 test_size=0.2, 2 random_state=500)
```

Dividimos o conjunto de dados com a função `train_test_split()` do módulo `sklearn.model_selection`. Os respectivos sentimentos e avaliações (as pontuações de sentimento) são passados para a função como arrays do NumPy, obtidos por meio da propriedade `values` dos objetos da Series correspondentes e extraídos do DataFrame. Passamos o parâmetro `test_size 1` para controlar como o conjunto de dados é dividido. O valor `0.2` significa que 20% das avaliações serão atribuídas aleatoriamente ao conjunto de teste. Assim, adotamos o padrão 80/20; os 80% restantes das avaliações vão compor o conjunto de treinamento. O parâmetro `random_state 2` inicializa o gerador interno de números aleatórios necessário para dividir aleatoriamente os dados.

Transformando texto em vetores de features numéricas

Para treinar e testar nosso modelo, precisamos de uma maneira de representar os dados de texto numericamente. É aí que o modelo *bag of words* (*BoW* ou saco de palavras) entra em cena. Esse modelo representa um texto como o conjunto (bag) de suas palavras, a fim de gerar dados numéricos sobre o texto. A feature numérica mais típica, gerada a partir do modelo BoW, é a frequência de palavras, o número de ocorrências de cada palavra no texto. Vejamos um simples exemplo que mostra como o modelo BoW transforma um texto em um vetor de feature numérica com base na frequência de palavras:

```
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]
```

Podemos usar a função `CountVectorizer()` do scikit-learn a fim de criar uma matriz BoW para os dados de texto. A `CountVectorizer()` converte dados de texto em vetores de features numéricas (vetores n-dimensionais de features numéricas representando algum objeto) e executa a tokenização (separa um texto em palavras individuais e em sinais de pontuação) com o tokenizador padrão ou personalizado. É possível implementar um tokenizador personalizado com uma ferramenta de processamento de linguagem natural como a spaCy, apresentada no capítulo anterior. Nesse exemplo, adotaremos a opção padrão para simplificar as coisas.

Vejamos como converter as avaliações em vetores de feature:

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

Primeiro, criamos um objeto `vectorize`. Em seguida, aplicamos o método `fit()` do vetorizador para criar o vocabulário de tokens encontrados no conjunto de dados `reviews`, que contém todas as avaliações dos conjuntos de treinamento e de teste. Depois disso, usamos o método `transform()` do objeto `vectorizer` para transformar os dados de texto nos conjuntos de treinamento e de teste em vetores de features numéricas.

Treinando e avaliando um modelo

Agora que temos os conjuntos de treinamento e de teste na forma de vetores numéricos, podemos treinar e testar o modelo. Primeiro, vamos treinar o classificador `LogisticRegression()` do scikit-learn para predizer o sentimento de uma avaliação. A *regressão logística* é um algoritmo básico, ainda que popular, para resolver problemas de classificação.

A seguir, criamos um classificador `LogisticRegression()` e usamos seu método `fit()` para treinar o modelo de acordo com os dados de treinamento fornecidos:

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

Agora, devemos avaliar com que acurácia o modelo consegue efetuar predições em relação a dados novos. Para tal, será necessário um conjunto de dados rotulados. Por isso, é comum dividir um conjunto de dados rotulados em um conjunto de treinamento e em um conjunto de testes, como fizemos antes. Aqui, avaliamos o modelo usando o conjunto de testes:

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

Normalmente, a classificação da acurácia aparece assim:

```
Accuracy: 0.81
```

Isso significa que o modelo tem 81% de acurácia. Na função `train_test_split()`, se tentarmos usar o parâmetro `random_state`, poderemos obter um valor um pouco diferente, já que as instâncias dos conjuntos de treinamento e de teste são escolhidas aleatoriamente a partir do conjunto original.

Efetuando predições com novos dados

Agora que treinamos e testamos nosso modelo, ele está pronto para analisar novos dados e não rotulados. Isso nos fornecerá um cenário mais completo de como o modelo está se saindo. Vamos testá-lo com algumas amostras de novas avaliações:

```
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))
```

Começamos criando uma lista de avaliações novas e transformando esse texto novo em vetores de features numéricas. Por último, predizemos os sentimentos de classe para as amostras novas. Os sentimentos são retornados como uma lista:

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

Lembre-se, `0` indica uma avaliação negativa e `1` indica uma avaliação positiva. Conforme podemos verificar, o modelo funcionou para essas amostras de avaliações, mostrando que a primeira é negativa e as outras duas são positivas.

Recapitulando

Neste capítulo, conhecemos algumas das bibliotecas Python de terceiros mais populares para aplicações de ciência de dados. Primeiro, exploramos a biblioteca NumPy e seus objetos de array multidimensional, depois analisamos a biblioteca pandas e suas estruturas de dados Series e DataFrame. Vimos como criar arrays do NumPy, objetos da Series e do DataFrame do pandas a partir de estruturas built-in do Python, como listas, e de fontes de dados armazenadas em formatos padrão, como JSON. Aprendemos também como acessar e manipular dados nesses objetos. Por último, usamos o scikit-learn, biblioteca popular do Python de aprendizado de máquina, a fim de criarmos um modelo preditivo para classificação.

CAPÍTULO 4

Acessando dados a partir de arquivos e de APIs

Em análise de dados, a primeira etapa é acessarmos os dados e inseri-los em nosso script. Neste capítulo, veremos diversas formas de importar dados a partir de arquivos e de outras fontes para nossas aplicações Python, assim como formas de exportar dados para arquivos. Veremos também como acessar o conteúdo de diferentes tipos de arquivos, incluindo aqueles armazenados localmente em sua máquina, e outros que podemos acessar de modo remoto por meio de requisições HTTP. Além disso, aprenderemos como obter dados enviando requisições para APIs acessíveis por meio de um URL. Por último, aprenderemos como carregar diferentes tipos de dados em DataFrames do pandas.

Importando dados com a função open() do Python

A função built-in `open()` do Python abre um arquivo de qualquer tipo para ser processado dentro de um script. A função retorna um objeto `file`, com métodos que possibilitam acessar e manipular o conteúdo encontrado no arquivo. No entanto, se o arquivo contiver dados em determinados formatos, como CSV, JSON ou HTML, será necessário importar uma biblioteca correspondente para acessar e para manipular os dados. O processamento de arquivos simples de texto não exige nenhuma biblioteca especial; basta invocar os métodos do objeto `file` retornado pela função `open()`.

Arquivos de texto

Talvez os arquivos de texto (*.txt*) sejam o tipo de arquivo mais comum que você encontrará. Para o Python, um arquivo de texto é uma sequência de objetos de string. Cada objeto de string é uma linha do arquivo de texto, ou seja, uma sequência de caracteres que termina em um caractere de nova linha não exibido (`\n`) ou quebra de linha incondicional (hard return).

NOTA *Na tela, é possível exibir uma única linha de um arquivo de texto como múltiplas linhas, dependendo da largura da janela de visualização, mas o Python ainda a entenderá como uma linha, desde que não seja dividida por caracteres de nova linha.*

O Python comporta funções built-in para manipular arquivos de texto, possibilitando que executemos operações de leitura, gravação e anexação neles. Nesta seção, vamos focar como ler dados de um arquivo de texto. Comece digitando a seguinte passagem em um editor de texto e salve-a como *excerpt.txt*. Não se esqueça de pressionar ENTER duas vezes no final do primeiro parágrafo a fim de criar a linha em branco entre os parágrafos (mas não pressione ENTER para quebrar as linhas extensas):

```
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.
```

Para os humanos, a passagem consiste em dois parágrafos com três sentenças no total. Para o Python, no entanto, a passagem inclui duas linhas não vazias e uma linha em branco entre elas. Vejamos como ler todo o conteúdo do arquivo em um script do Python e como exibi-lo:

```
1 path = "/path/to/excerpt.txt"
  with open(2 path, 3 "r") as 4 f:
  5 content = f.read()
  print(content)
```

Começamos especificando o path para o arquivo `1`. Será necessário substituir

`/path/to/excerpt.txt` por seu próprio path de arquivo com base em que diretório o arquivo é salvo. Passamos o path para a função `open()` como primeiro parâmetro `2`. O segundo parâmetro controla como o arquivo será usado. O parâmetro default é a leitura no modo texto. Ou seja, o conteúdo do arquivo será aberto apenas para leitura (não edição) e tratado como strings. Especificamos explicitamente `"r"` para a *reading* `3`, mas isso não é estritamente necessário. (Passar `"rt"` faria o modo explícito com a especificação da leitura.) A função `open()` retorna um objeto de arquivo no modo especificado `4`. Em seguida, utilizamos o método `read()` do objeto `file` para ler todo o conteúdo do arquivo `5`.

A palavra reservada `with` com a função `open()` garante que o objeto `file` seja fechado corretamente quando terminarmos de usá-lo, mesmo que uma exceção tenha sido lançada. Caso contrário, precisaremos chamar `f.close()` para fechar o objeto de arquivo e liberar os recursos do sistema consumidos.

O trecho de código a seguir lê no mesmo `/path/to/excerpt.txt` o conteúdo do arquivo linha por linha, exibindo apenas linhas não vazias:

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

Neste exemplo, adicionamos números de linha a cada linha com a função `enumerate()` `1`. Em seguida, filtramos as linhas vazias com o método `strip()` `2`, que remove qualquer espaço em branco do início e do fim do objeto da string em cada linha. A segunda linha em branco do arquivo de texto contém somente um caractere, uma linha nova, que o método `strip()` remove. Desse modo, a segunda linha se torna uma string vazia, que a instrução `if` avaliará como False e ignorará. Vejamos como fica nossa saída. Como podemos ver, não há `Line 1`:

```
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.
```

Em vez de exibirmos as linhas, podemos enviá-las para uma lista com uma list comprehension:

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

Na lista, cada linha não vazia será um item separado.

Arquivos de dados tabulares

Um arquivo de dados *tabulares* é um arquivo no qual os dados são estruturados em linhas. Normalmente, cada linha contém informações sobre alguém ou algo, como mostrado a seguir:

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

Esse é um exemplo de um arquivo simples (*flat file*), o tipo mais comum de arquivo de dados tabulares. Em inglês, o nome flat se origina da estrutura: os arquivos simples contêm registros simples estruturados (flat), significando que os registros não contêm estruturas aninhadas ou sub-registros. Um arquivo simples usualmente é um arquivo de texto em formato CSV ou valores separados por tabulação (TSV), contendo um registro por linha. Nos arquivos *.csv*, os valores em um registro são separados por vírgulas, enquanto os arquivos *.tsv* usam tabulações como separadores. Ambos os formatos são amplamente suportados e são bastante usados na troca de dados para mover dados tabulares entre diferentes aplicações.

Vejamos a seguir um exemplo de dados em formato CSV, em que a primeira linha contém cabeçalhos descrevendo o conteúdo das linhas abaixo dele. As descrições de cabeçalho são utilizadas como *chaves* para os dados nas linhas seguintes. Copie esses dados em um editor de texto e salve-os como *cars.csv*:

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

A função `open()` do Python pode abrir arquivos *.csv* no modo de texto. Depois,

podemos carregar os dados em um objeto Python com uma função reader do módulo `csv`, conforme ilustrado aqui:

```
import csv
path = "/path/to/cars.csv"
with open(path, "r") as 1 csv_file:
  csv_reader = 2 csv.DictReader(csv_file)
  cars = []
  for row in csv_reader:
  3 cars.append(dict(row))
print(cars)
```

A função `open()` retorna um objeto `file 1`, que passamos para um reader do módulo `csv`. Nesse caso, recorremos ao `DictReader() 2`, que mapeia os dados em cada linha para um dicionário usando os cabeçalhos correspondentes da primeira linha como chaves. Anexamos esses dicionários a uma lista `3`. Vejamos a lista resultante de dicionários:

```
[
{'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'}
]
```

Como alternativa, é possível usar o método `reader()` do módulo `csv` para transformar o arquivo *.csv* em uma lista de listas, em que cada lista interna representa uma linha, incluindo a linha de cabeçalho:

```
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)
```

Vamos conferir a saída:

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

Os métodos `csv.DictReader()` e `csv.reader()` têm um parâmetro opcional

`delimiter`, possibilitando que especifiquemos o caractere que separa os campos em nosso arquivo de dados tabular. Por padrão, esse parâmetro usa uma vírgula, perfeito para arquivos *.csv*. No entanto, ao definirmos o parâmetro como `delimiter = "\t"`, podemos ler os dados separados por tabulações dos arquivos *.tsv*.

EXERCÍCIO #5: ABRINDO ARQUIVOS JSON

É possível abrir arquivos JSON com a função `open()` no modo de texto e, em seguida, usar o módulo `json` para processamento posterior. Como o `CSV`, o `json` é um pacote built-in do Python, você não precisa instalá-lo separadamente. No Capítulo 3, você viu um exemplo com o módulo `json` 3 ao converter um documento JSON em um objeto do pandas. Neste exercício, usará o módulo `json` para salvar o seguinte texto em um arquivo *.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"}
]}
```

Abra o arquivo para leitura com a função `open()` e envie o objeto de arquivo obtido para o método `json.load()`, que desserializará o JSON em um objeto do Python. A partir desse objeto, extraia a parte que contém as linhas de carros. Com um loop, itere sobre essas linhas, gerando os valores da seguinte forma:

```
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
```

Arquivos binários

Não lidaremos somente com arquivos de texto. Existem também arquivos executáveis (*.exe*) e de imagem (*.jpeg*, *.bmp* e assim por diante), que contêm

dados em formato binário, representados como uma sequência de bytes. Em geral, esses bytes devem ser interpretados como algo diferente de caracteres de texto, por isso não podemos abrir um arquivo binário no modo de texto a fim de acessar e manipular seu conteúdo. Em vez disso, devemos usar o modo binário da função `open()`.

O exemplo a seguir mostra como abrir um arquivo de imagem no modo binário. Se tentássemos abri-lo no modo de texto, receberíamos um erro. É possível executar o seguinte código com qualquer arquivo *.jpg* em seu computador:

```
  image = "/path/to/file.jpg"
  with open(image, 1 "rb") as image_file:
    content = 2 image_file.read()
 3 print(len(content))
```

Instruímos a função `open()` a abrir um arquivo para leitura no modo binário passando `"rb"` como o segundo parâmetro `1`. O objeto obtido, como um objeto no modo de texto, tem o método `read()` para obter o conteúdo do arquivo `2`. Aqui, o conteúdo é acessado como um objeto `bytes`. Nesse exemplo, simplesmente determinamos o número de bytes lidos do arquivo `3`.

Exportando dados para arquivos

Após um pouco de processamento, talvez seja necessário armazenar dados em um arquivo para que possamos usá-los durante a próxima execução do script ou importá-los para outros scripts ou para outras aplicações. Além disso, é necessário armazenarmos as informações em um arquivo para que você ou outras pessoas possam visualizá-lo. Por exemplo, talvez você queira registrar informações sobre as exceções e os erros gerados por sua aplicação para análise posterior.

É possível criar um arquivo novo a partir de nosso script Python e gravar dados nele, ou podemos gravar os dados em um arquivo existente. Vamos explorar o último exemplo. Retomando o exemplo de “Arquivos de dados tabulares”, suponha que precisemos modificar uma linha no arquivo *cars.csv*, alterando o preço de um determinado carro. Lembre-se de que os dados foram lidos a partir do arquivo *cars.csv* em uma lista de dicionários

chamada `cars`. Nessa lista, para vermos os valores de cada dicionário, podemos executar o seguinte loop:

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

No corpo do loop, chamamos o método `values()` em cada dicionário da lista, convertendo, assim, os valores do dicionário em um objeto `dict_values` que pode ser facilmente convertido em uma lista. Cada lista representa uma linha do arquivo *.csv* original, como mostrado a seguir:

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

Suponha que precisemos atualizar o campo `Price`, na segunda linha (para o Chevy Venture), e armazenar essa alteração no arquivo *cars.csv* original. Podemos fazer essa alteração da seguinte forma:

```
1 to_update = ['1999', 'Chevy', 'Venture']
2 new_price = '4500.00'
3 with open('path/to/cars.csv', 'w') as csvfile:
 4 fieldnames = cars[0].keys()
 5 writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
   writer.writeheader()
 6 for row in cars:
     if set(to_update).issubset(set(row.values())):
       row['Price'] = new_price
     writer.writerow(row)
```

Primeiro, precisamos de uma maneira para identificar a linha a ser atualizada. Então, criamos uma lista chamada `to_update`, incluindo campos suficientes da linha para identificar exclusivamente a linha `1`. Em seguida, especificamos o valor novo para o campo a ser alterado como `new_price` `2`. Depois, abrimos o arquivo para gravação, passando a flag `'w'` à função `open()` `3`. Aqui, o modo `w` substituirá o conteúdo existente do arquivo. Assim sendo, devemos definir os nomes de campo a serem enviados para o arquivo `4`. São os nomes das chaves usadas em um dicionário que representa uma linha de carro.

Com a função `csv.DictWriter()` `5`, criamos um objeto writer que mapeará os dicionários da lista `cars` para as linhas de saída a serem enviadas ao *arquivo* cars.csv. Na lista `cars`, com um loop iterando os dicionários `6`, verificamos se cada linha corresponde ao seu identificador especificado. Em caso afirmativo,

atualizamos o campo `Price` da linha. Por último, ainda dentro do loop, escrevemos cada linha no arquivo usando o método `writer.writerow()`.

Vamos conferir o que veremos no arquivo *cars.csv* após executarmos o script:

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

Conforme podemos verificar, parece o conteúdo original, mas o valor do campo `Price` na segunda linha foi alterado.

Acessando arquivos remotos e APIs

No Python, diversas bibliotecas de terceiros, incluindo a urllib3 e a Requests, possibilitam obter dados de um arquivo remoto acessível por URL. Além do mais, é possível usar as bibliotecas a fim de enviar requisições para APIs HTTP (aquelas que usam HTTP como protocolo de transferência), muitas das quais retornam os dados solicitados em formato JSON. Tanto a urllib3 quanto a Requests funcionam formulando requisições HTTP personalizadas com base nas informações inseridas.

O *HTTP (Protocolo de Transferência de Hipertexto)*, protocolo cliente/servidor que constitui a base da troca de dados pela web, é estruturado como uma série de requisições e respostas. As mensagens HTTP enviadas por um cliente são *requisições*, ao passo que as mensagens de resposta retornadas pelo servidor são *respostas*. Por exemplo, sempre que clicamos em um link em nosso navegador, o navegador, atuando como cliente, envia uma requisição HTTP para buscar a página web desejada no servidor web adequado. Podemos fazer a mesma coisa a partir de um script do Python. O script, atuando como cliente, obtém os dados solicitados na forma de um documento JSON ou XML.

Como funcionam as requisições HTTP

Existem diversos tipos de requisições HTTP. Entre as mais comuns, temos `GET`, `POST`, `PUT` e `DELETE`. São conhecidas também como *métodos de requisição*

HTTP, *comandos HTTP* ou apenas *verbos HTTP*. O comando HTTP em qualquer requisição HTTP define a ação a ser executada para um recurso especificado. Por exemplo, uma requisição `GET` obtém dados de um recurso, enquanto uma requisição `POST` envia dados para um destino.

Uma requisição HTTP também inclui o *destino* da requisição, geralmente incluindo um URL e *cabeçalhos*, sendo estes últimos campos que passam informações adicionais com a requisição. Algumas requisições também incluem um *corpo*, que carrega dados de requisições reais, como um envio de formulário. Em geral, as requisições `POST` incluem um corpo, enquanto as requisições `GET` não incluem.

Como exemplo, vejamos a seguinte requisição HTTP:

```
1 GET 2 /api/books?bibkeys=ISBN%3A1718500521&format=json HTTP/1.1
3 Host: openlibrary.org
4 User-Agent: python-requests/2.24.0
5 Accept-Encoding: gzip, deflate
  Accept: */*
6 Connection: keep-alive
```

Essa requisição utiliza o comando HTTP `GET` `1` a fim de obter os dados do servidor fornecido (indicado como `Host` `3`) usando o URI especificado `2`. As linhas restantes incluem outros cabeçalhos especificando informações adicionais. O cabeçalho da requisição `User-Agent` identifica a aplicação que faz a requisição e sua versão `4`. Os cabeçalhos `Accept` informam quais tipos de conteúdo o cliente é capaz de entender `5`. O cabeçalho `Connection`, definido como `keep-alive` `6`, instrui o servidor a estabelecer uma conexão persistente com o cliente, possibilitando que requisições subsequentes sejam feitas.

No Python, não precisamos compreender com detalhes a estrutura interna das requisições HTTP para enviá-las e receber respostas. Conforme aprendermos nas seções a seguir, bibliotecas como a Requests e a urllib3 possibilitam manipular requisições HTTP com facilidade e com eficiência, apenas chamando um método adequado e passando os parâmetros necessários para ele.

Com a ajuda da biblioteca Requests, a requisição HTTP anterior pode ser gerada por um script simples do Python:

```
import requests
```

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

Em breve, analisaremos a biblioteca Requests em detalhes. Por ora, observe que, com essa biblioteca, não precisamos definir manualmente os cabeçalhos de uma requisição. A biblioteca se encarrega de definir os valores padrão nos bastidores, gerando de forma automática uma requisição HTTP totalmente formatada com apenas algumas linhas de código.

Biblioteca urllib3

A urllib3 é uma biblioteca de processamento de URL que viabiliza acessar e manipular recursos acessíveis por URL, como APIs HTTP, sites e arquivos. A biblioteca foi desenvolvida para manipular requisições HTTP com eficiência, usando o pool de conexão thread-safe para minimizar os recursos necessários no final do servidor. Comparada à biblioteca Requests, que veremos a seguir, a urllib3 exige mais trabalho manual, apesar de também fornecer controle mais direto sobre as requisições que preparamos, o que ajuda quando, por exemplo, precisamos personalizar o comportamento do pool ou decodificar explicitamente respostas HTTP.

Instalando a urllib3

Como a urllib3 depende de muitos pacotes populares do Python, como Requests e `pip`, é provável que já esteja instalada em seu ambiente Python. Vamos verificar, tentando importá-la em uma sessão Python. Caso receba um `ModuleNotFoundError`, poderá instalá-la explicitamente com o seguinte comando:

```
$ pip install urllib3
```

Acessando arquivos com a urllib3

Para vermos como carregar dados de um arquivo acessível por URL com a urllib3, podemos usar o arquivo *excerpt.txt* criado anteriormente. Para que esse arquivo seja acessível por meio de um URL, podemos inseri-lo na pasta de documentos de um servidor HTTP em execução em seu host local. Como alternativa, use o seguinte URL para obtê-lo no repositório do GitHub que acompanha este livro: https://github.com/pythondatabook/sources/blob/main/ch4/excerpt.txt.

Vamos executar o seguinte código, substituindo o URL, se necessário:

```
  import urllib3
1 http = urllib3.PoolManager()
2 r = http.request('GET', 'http://localhost/excerpt.txt')
  for i, line in enumerate(3 r.data.decode('utf-8').split('\n')):
    if line.strip():
    4 print("Line %i: " %(i), line.strip())
```

Primeiro, criamos uma instância `PoolManager` `1`, que é como a urllib3 cria requisições. Depois disso, criamos uma requisição HTTP para o URL especificado com o método `request()` do `PoolManager` `2`. O método `request()` retorna um objeto `HTTPResponse`. Acessamos os dados solicitados por meio do atributo `data` desse objeto `3`. Em seguida, geramos apenas linhas não vazias, enumerando-as no início de cada linha `4`.

Requisições de API com a urllib3

Podemos usar também a urllib3 a fim de criar requisições para APIs HTTP. No exemplo a seguir, criamos uma requisição a News API (https://newsapi.org), que busca artigos a partir de uma ampla variedade de fontes de notícias, encontrando aqueles que são mais relevantes a nossa requisição. Como muitas outras APIs atuais, essa API exige uma chave de API a cada requisição. É possível obter uma chave de API de desenvolvedor gratuitamente em https://newsapi.org/register, após preencher um formulário simples de cadastro. Em seguida, usamos esse código para pesquisar artigos sobre a linguagem de programação Python:

```
  import json
  import urllib3
  http = urllib3.PoolManager()
  r = http.request('GET', 'https://newsapi.org/v2/everything? 1 q=Python
          programming language& 2 apiKey=your_api_key_here& 3 pageSize=5')
4 articles = json.loads(r.data.decode('utf-8'))
  for article in articles['articles']:
    print(article['title'])
    print(article['publishedAt'])
    print(article['url'])
    print()
```

Passamos a frase de pesquisa como parâmetro `q` no URL da requisição `1`. No URL da requisição, o único outro parâmetro necessário a ser especificado é o

`apiKey` 2, ao qual passamos a chave de API. Existem também muitos outros parâmetros opcionais. Por exemplo, podemos especificar as fontes de notícias ou blogs a partir dos quais queremos artigos. Nesse exemplo específico, utilizamos `pageSize` para definir o número de artigos sendo obtidos, nesse caso cinco 3. A lista completa de parâmetros suportados pode ser encontrada na documentação da News API em https://newsapi.org/docs.

O atributo `data` do objeto `HTTPResponse` retornado por `request()` é um documento JSON na forma de um objeto `bytes`. Decodificamos o objeto em uma string, que passamos para o método `json.loads()` visando convertê-la em um dicionário 4. Nesse dicionário, para vermos como os dados são estruturados, podemos gerá-los como saída, mas essa etapa é omitida nessa listagem. Se analisássemos a saída, veríamos que as informações sobre os artigos podem ser encontradas na lista chamada `articles`, dentro do documento retornado e, nessa lista, cada registro tem os campos `title`, `publishedAt` e `url`.

Com essas informações, podemos exibir a lista obtida de artigos em um formato mais legível, gerando uma saída mais ou menos assim:

```
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
```

Esse exemplo ilustrou como integrar a API News em uma aplicação do Python usando requisições diretas HTTP por meio da biblioteca urllib3. Uma alternativa seria usar a biblioteca de cliente do Python não oficial em https://newsapi.org/docs/client-libraries/python.

Biblioteca Requests

Requests é outra biblioteca popular de processamento de URL que possibilita enviar facilmente requisições HTTP. A Requests usa a urllib3 nos bastidores, facilitando ainda mais a criação de requisições e obtenção de dados. Podemos instalar a biblioteca Requests com o comando `pip`:

```
$ pip install requests
```

Os verbos HTTP são implementados como os métodos da biblioteca (por exemplo, `request.get()` para uma requisição HTTP `GET`). Vejamos como acessar remotamente *excerpt.txt* com a Requests. Substitua o URL pelo link GitHub do arquivo, se necessário:

```
  import requests
1 r = requests.get('http://localhost/excerpt.txt')
  for i, line in enumerate(2 r.text.split('\n')):
    if line.strip():
    3 print("Line %i: " %(i), line.strip())
```

Criamos uma requisição HTTP `GET` com o método `requests.get()`, passando o URL do arquivo como parâmetro `1`. O método retorna um objeto `Response` que inclui o conteúdo obtido no atributo `text` `2`. A Requests decodifica automaticamente o conteúdo obtido, fazendo suposições fundamentadas sobre a codificação, para que não precisemos fazer manualmente essa operação. Assim como no exemplo da urllib3, geramos somente as linhas não vazias, adicionando um número de linha no início de cada uma `3`.

EXERCÍCIO #6: ACESSANDO UMA API COM A REQUESTS

Assim como a urllib3, a biblioteca Requests pode interagir com APIs HTTP. Tente reescrever o código que envia uma requisição `GET` a News API, de modo que essa requisição use a biblioteca Requests no lugar da biblioteca urllib3. Observe que, com a Requests, você não precisa adicionar manualmente os parâmetros de query ao URL passado. Em vez disso, é possível passar os parâmetros como um dicionário de strings.

Movendo dados a partir de e para um DataFrame

O pandas tem uma variedade de métodos reader, cada um deles desenvolvido para carregar dados em um determinado formato e/ou a partir de um determinado tipo de fonte. Esses métodos possibilitam carregar dados tabulares em um DataFrame com a ajuda de uma única chamada, fazendo com que o conjunto de dados importado fique imediatamente pronto para análise. O pandas também tem métodos para converter dados de DataFrame em outros formatos, como JSON. Nesta seção, exploramos exemplos de métodos para mover dados a partir de ou para um DataFrame. Veremos também a biblioteca pandas-datareader, conveniente para carregar dados a partir de diversas fontes online em DataFrames do pandas.

Importando estruturas aninhadas JSON

Visto que JSON se tornou o padrão dominante para a troca de dados entre aplicações, é importante aprender uma forma de importar rapidamente um documento JSON e convertê-lo em uma estrutura de dados do Python. No capítulo anterior, vimos um exemplo de como carregar uma estrutura simples JSON e não aninhada em um DataFrame com o método reader `read_json()`. Nesta seção, aprenderemos como carregar um documento JSON mais complexo e aninhado, como este:

```
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}
   ]
}]
```

Como podemos ver, no documento JSON, cada registro começa com um par de chave-valor estruturado e simples com a chave `Emp`, seguido por uma estrutura aninhada com a chave `POs`. É possível converter essa estrutura hierárquica do JSON em um DataFrame tabular do pandas com o método

reader `json_normalize()` também do pandas, que pega uma estrutura aninhada e a achata ou a *normaliza* em uma tabela simples. Vejamos como fica essa estrutura:

```
import json
import pandas as pd
df = pd.json_normalize(1 data, 2 "POs", 3 "Emp").set_index([4 "Emp","Pono"])
print(df)
```

Além da amostra JSON `1` a ser processada por `json_normalize()`, especificamos também `POs` como array aninhado a ser achatado `2` e `Emp` como o campo a ser usado como parte do índice complexo na tabela resultante `3`. Na mesma linha de código, definimos duas colunas como o índice: `Emp` e `Pono` `4`. Como resultado, veremos o seguinte DataFrame do pandas:

```
                   Total
Emp          Pono
Jeff Russell 2608     35
             2617     35
             2620    139
Jane Boorman 2621     95
             2626    218
```

NOTA *Usar um índice de duas colunas simplifica a agregação de dados em grupos. No Capítulo 6, abordaremos detalhadamente os DataFrames com índices multicolunas.*

Convertendo um DataFrame em JSON

Na prática, precisamos às vezes efetuar a operação inversa e converter um DataFrame do pandas em um arquivo JSON. O código a seguir converte nosso DataFrame em amostra JSON, a partir da qual foi originalmente gerado:

```
1 df = df.reset_index()
  json_doc = (2 df.groupby(['Emp'], as_index=True)
              3 .apply(lambda x: x[['Pono','Total']].to_dict('records'))
              4 .reset_index()
              5 .rename(columns={0:'POs'})
              6 .to_json(orient='records'))
```

Começamos descartando o índice de duas colunas do DataFrame para tornar as colunas `Emp` e `Pono` regulares `1`. Em seguida, utilizamos um one-liner

composto a fim de converter o DataFrame em um documento JSON. Primeiro, aplicamos uma operação `groupby` ao DataFrame, agrupando as linhas pela coluna `Emp` 2. Utilizamos um `groupby()` com `apply()` para aplicar uma função lambda a cada registro em cada grupo 3. Na expressão lambda, especificamos a lista de campos que queremos ver em uma linha do array aninhado, associado a cada registro `Emp`. Recorremos ao método `DataFrame.to_dict ()` com o parâmetro `records` para formatar os campos no array da seguinte forma: `[{coluna: valor}, ... , {coluna: valor}]`, e cada dicionário representa uma ordem associada a um determinado funcionário.

Nesse momento, temos um objeto Series com o índice `Emp` e uma coluna com um array de ordens associado a um funcionário. Para nomear essa coluna (nesse caso, `POs`), é necessário converter a Series em um DataFrame. Uma forma simples de fazer isso é com `reset_index()` 4. Além de converter a Series em um DataFrame, `reset_index()` altera `Emp` de um índice para uma coluna regular, o que será importante quando convertermos o DataFrame para o formato JSON. Por último, definimos explicitamente o nome da coluna que contém o array aninhado (`POs`) com o método `rename()` do DataFrame 5 e transformamos esse DataFrame em um arquivo JSON 6.

Vejamos o conteúdo do `json_doc`:

```
[{"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}
    ]
}]
```

Para melhorar a legibilidade, é possível exibi-lo com o seguinte comando:

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

EXERCÍCIO #7: MANIPULANDO ESTRUTURAS COMPLEXAS JSON

Na seção anterior, a amostra JSON usada tinha um único campo estruturado simples (`Emp`) no nível superior de cada registro. Em um documento JSON do mundo cotidiano, pode haver mais desses campos. Neste exemplo, os registros têm um segundo campo simples, `Emp_email`, no nível superior:

```
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}
   ]
}]
```

Para carregar esses dados em um DataFrame, é necessário passar uma lista de todos os campos estruturados simples de nível superior para o terceiro parâmetro de `json_normalize()`, como mostrado a seguir:

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

O conteúdo do DataFrame será o seguinte:

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

Tente converter esse DataFrame novamente no documento JSON original, modificando a operação `groupby` mostrada na seção anterior.

Carregando dados online em um DataFrame com a pandas-datareader

Diversas bibliotecas de terceiros vêm com métodos reader compatíveis com o pandas para acessar dados a partir de uma variedade de fontes online, como Quandl (https://data.nasdaq.com) e Stooq (https://stooq.com). A mais popular é a pandas-datareader. No momento em que eu escrevia este livro, essa biblioteca comportava 70 métodos, cada um desenvolvido para carregar dados a partir de uma determinada fonte para um DataFrame do pandas. Muitos desses métodos são wrappers para APIs financeiras, possibilitando que obtenhamos dados financeiros no formato pandas com facilidade.

Instalando a pandas-datareader

Digite o seguinte comando para instalar a pandas-datareader:

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

Para saber mais sobre os métodos reader da biblioteca, confira a documentação da pandas-datareader em https://pandas-datareader.readthedocs.io/en/latest/remote_data.html. Podemos também exibir uma lista dos métodos disponíveis com a função `dir()` do Python:

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

Obtendo dados do Stooq

No exemplo a seguir, recorremos ao método `get_data_stooq()` para obter dados do índice S&P500 para um período especificado:

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

O método `get_data_stooq()` obtém dados do Stooq, site gratuito que fornece informações sobre vários índices de mercado. Passe o ticker do índice de mercado que deseja como o primeiro parâmetro. As opções disponíveis podem ser encontradas em https://stooq.com/t.

Em geral, os dados do índice S&P 500 obtidos aparecerão neste formato:

```
              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
```

Por padrão, a coluna `Date` é definida como o índice do DataFrame.

Recapitulando

Neste capítulo, aprendemos como obter dados a partir de diferentes fontes e inseri-los em nossos scripts do Python para processamento posterior. Em especial, vimos como importar dados de arquivos com as funções built-in do Python, como enviar requisições HTTP de seus scripts do Python para APIs

online e como usar os métodos reader do pandas a fim de obter diferentes formatos de dados a partir de variadas fontes. Além do mais, aprendemos como exportar dados para arquivos e como converter dados de DataFrame em JSON.

CAPÍTULO 5

Trabalhando com bancos de dados

Um *banco de dados* é uma coleção organizada de dados, que podem ser facilmente acessados, gerenciados e atualizados. Na arquitetura inicial de seu projeto, ainda que não exista um banco de dados, em algum momento, os dados que passam pela aplicação provavelmente terão contato com um ou mais banco de dados.

Retomando a análise do capítulo anterior sobre a importação de dados para nossas aplicações Python, neste Capítulo, veremos como trabalhar com banco de dados. Aqui, veremos exemplos de como acessar e manipular dados armazenados em bancos de dados de diferentes tipos, incluindo aqueles que estabelecem o SQL como principal ferramenta para trabalhar com dados e aqueles que não. Exploraremos como usar o Python a fim de interagir com uma variedade de bancos de dados populares, incluindo o MySQL, o Redis o e MongoDB.

Os bancos de dados proporcionam muitas vantagens. Por exemplo, com a ajuda de um banco de dados, é possível persistir os dados entre invocações de um script e compartilhá-los de forma eficiente entre diferentes aplicações. As linguagens de banco de dados também podem nos ajudar a organizar e a responder sistematicamente a perguntas sobre nossos dados. Além disso, muitos sistemas de banco de dados possibilitam implementar código de programação dentro do próprio banco de dados, podendo melhorar o desempenho, a modularidade e a reutilização de uma aplicação. Por exemplo, podemos armazenar uma *trigger* em um banco de dados; trata-se de um pedaço de código invocado automaticamente sempre que um determinado evento ocorre: como toda vez que inserimos uma linha nova em uma tabela

específica.

Há duas categorias de bancos de dados: bancos de dados relacionais e bancos de dados não relacionais (NoSQL). Os bancos de dados relacionais apresentam uma estrutura rígida implementada na forma de um esquema para os dados que estão sendo armazenados. Essa abordagem ajuda a assegurar a integridade, a consistência e a acurácia geral dos dados. No entanto, a principal desvantagem dos bancos de dados relacionais é que não escalam bem à medida que os volumes de dados aumentam. Por outro lado, os bancos de dados NoSQL não estabelecem restrições à estrutura dos dados armazenados, possibilitando, assim, mais flexibilidade, mais adaptabilidade e mais escalabilidade. Neste capítulo, abordaremos o armazenamento e a obtenção de dados em bancos de dados relacionais e não relacionais.

Bancos de dados relacionais

Bancos de dados relacionais, também conhecidos como *bancos de dados* orientados a linhas ou a colunas, são o tipo mais comum de banco de dados atualmente em uso. Esse tipo de banco viabiliza uma forma estruturada de armazenar dados. Assim como uma lista de livros na Amazon tem uma estrutura definida para armazenar informações, com campos para títulos de livros, autores, descrições, classificações e assim por diante, em um banco de dados relacional, os dados armazenados devem se ajustar a um esquema formal predefinido. Para trabalhar com um banco de dados relacional, começamos criando este esquema formal: definimos uma coleção de tabelas, cada uma composta de um conjunto de campos ou colunas, e especificamos que tipo de dados cada campo armazenará. Estabelecemos também os relacionamentos entre as tabelas. Em seguida, podemos armazenar dados no banco de dados, acessar dados do banco de dados ou atualizar os dados conforme necessário.

Os bancos de dados relacionais são arquitetados para permitir a inserção, atualização e/ou exclusão eficientes de pequenas a enormes quantidades de dados estruturados. Existem inúmeras aplicações em que esse tipo de banco de dados pode ser bastante utilizado. Em particular, os bancos de dados relacionais são apropriados para *aplicações de processamento de transações*

online (OLTP), que processam um grande volume de transações para um grande número de usuários.

Alguns sistemas de banco de dados relacionais comuns são MySQL, MariaDB e PostgreSQL. Nesta seção, focaremos o MySQL, sem sombras de dúvidas o banco de dados open source mais popular do mundo, para exemplificar como interagir com um banco de dados. Aprenderemos como configurar o MySQL, criar um banco de dados novo, definir sua estrutura e escrever scripts Python para armazenar e acessar dados a partir de e para o banco de dados.

Entendendo instruções SQL

O *SQL (Structured Query Language/ Linguagem de Consulta Estruturada)* é a principal ferramenta para interagir com um banco de dados relacional. Aqui, embora nosso foco seja a interface com bancos de dados usando o Python, o próprio código Python deve conter instruções SQL para fazer isso. Mesmo que a análise abrangente do SQL fuja ao escopo deste livro, uma breve introdução a essa linguagem de consulta é necessária.

Instruções SQL são comandos de texto reconhecidos e executados por um engine de banco de dados como o MySQL. Por exemplo, a seguinte instrução SQL solicita a um banco de dados que acesse todas as linhas de uma tabela chamada `orders` cujo campo `status` está definido como `Shipped`:

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

Via de regra, as instruções SQL têm três componentes majoritários: uma *operação* a ser executada, um *destino* para essa operação e uma *condição* que restringe o escopo da operação. No exemplo anterior, `SELECT` é a operação SQL, significando que estamos acessando linhas do banco de dados. A tabela `orders` é o destino para a operação, conforme definido pela cláusula `FROM`, e a condição é especificada na cláusula `WHERE` da instrução. Todas as instruções SQL devem ter uma operação e um destino, mas a condição é opcional. Por exemplo, como a instrução a seguir não tem uma condição, acessa todas as linhas da tabela `orders`:

```
SELECT * FROM orders;
```

Podemos também refinar instruções SQL para afetar somente determinadas

colunas de uma tabela. Vejamos como acessar apenas as colunas `pono` e `date` de todas as linhas na tabela `orders`:

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

Por convenção, palavras reservadas na linguagem SQL, como `SELECT` e `FROM`, são escritas em letras maiúsculas. No entanto, o SQL é uma linguagem que não diferencia maiúsculas de minúsculas (case-insensitive). Assim, letras maiúsculas não são estritamente necessárias. Cada instrução SQL deve terminar com um ponto e vírgula.

As operações `SELECT`, como as que acabamos de ver, são exemplos de *instruções DML (Data Manipulation Language*/Linguagem de Manipulação de Dados), categoria de instruções SQL que usamos para acessar e manipular dados do banco de dados. Outras operações DML incluem `INSERT`, `UPDATE` e `DELETE`, que adicionam, alteram e removem registros de um banco de dados, respectivamente. As *instruções DDL (Data Definition Language*/Linguagem de Definição de Dados) são outra categoria comum de instruções SQL. Podemos utilizá-las para realmente definirmos a estrutura do banco de dados. As típicas operações DDL incluem `CREATE` para criar, `ALTER` para modificar e `DROP` para excluir contêineres de dados, sejam colunas, tabelas ou bancos de dados inteiros.

Introdução ao MySQL

O MySQL é disponibilizado na maioria dos sistemas operacionais modernos, incluindo Linux, Unix, Windows e macOS. Edições gratuitas e comerciais estão disponíveis. Neste capítulo, podemos usar o MySQL Community Edition (https://www.mysql.com/products/community), versão para download gratuito do MySQL, disponível sob a licença GPL. Para instruções detalhadas de instalação do MySQL em seu sistema operacional, confira o manual de referência da versão mais recente do MySQL, disponível em https://dev.mysql.com/doc.

Para inicializar o servidor MySQL após instalação, precisamos usar o comando que o guia de instalação especifica para nosso sistema operacional. Em seguida, é possível se conectar ao servidor MySQL a partir de um terminal do sistema usando o programa cliente *mysql*:

```
$ mysql -uroot -p
```

NOTA *No macOS, talvez seja necessário usar todo o path para o MySQL, como* `/usr/local/mysql/bin/mysql -uroot -p`.

Você será solicitado a inserir sua senha, definida durante o processo de instalação do servidor MySQL. Depois disso, veremos prompt MySQL:

```
mysql>
```

Caso queira, é possível escolher uma senha nova para o usuário root com o seguinte comando SQL:

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

Agora, podemos criar o banco de dados de que precisaremos para nossa aplicação. Digite este comando no prompt `mysql>`:

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

Esse comando cria um banco de dados chamado `sampledb`. Em seguida, devemos escolher esse banco de dados para uso:

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

Agora, quaisquer comandos subsequentes serão aplicados ao nosso banco de dados `sampledb`.

Definindo a estrutura do banco de dados

Um banco de dados relacional obtém sua estrutura a partir da composição de suas tabelas constituintes e das conexões entre essas tabelas. Os campos que relacionam tabelas diferentes são chamados de *chaves*. Existem dois tipos: *chaves primárias* e *chaves estrangeiras*. Uma chave primária identifica exclusivamente um registro em uma tabela. Uma chave estrangeira é um campo em outra tabela que corresponde à chave primária na primeira tabela. Em geral, a chave primária e sua chave estrangeira correspondente compartilham o mesmo nome em ambas as tabelas.

NOTA *Os termos campo e coluna são usados com frequência de forma intercambiável. A rigor, uma coluna se torna um campo quando a referenciamos no contexto de uma única linha.*

Agora que criamos nosso banco de dados `sampledb`, vamos criar algumas tabelas e definir sua estrutura. Para fins de demonstração, as tabelas terão a mesma estrutura de alguns dos DataFrames do pandas com os quais trabalhamos no Capítulo 3. Vejamos três estruturas de dados tabulares a serem implementadas em nosso banco de dados:

```
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
```

Para analisar quais relacionamentos podem ser estabelecidos entre essas estruturas, confira as figuras 3.4 e 3.6 do Capítulo 3. Conforme podemos ver na Figura 3.4, as linhas nas tabelas `emps` e `salary` apresentam um relacionamento de um-para-um. O relacionamento é estabelecido por meio do campo `empno`. As tabelas `emps` e `orders` também estão relacionadas por meio do campo `empno`. Trata-se de um relacionamento um-para-muitos, conforme ilustrado na Figura 3.6.

É possível adicionar essas estruturas de dados ao nosso banco de dados relacional com os comandos SQL no prompt `mysql>`. Vamos começar criando

a tabela `emps`:

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

Criamos a tabela com o comando `CREATE TABLE`, especificando cada coluna com o tipo e, opcionalmente, o tamanho dos dados que podemos armazenar nela. Por exemplo, a coluna `empno` é para inteiros (tipo `INT`), e a restrição `NOT NULL` aplicada a ela garante que não poderemos inserir uma linha com um campo `empno` vazio. Além disso, a coluna `empname` pode armazenar strings (do tipo `VARCHAR`) de até 50 caracteres, enquanto `job` pode armazenar strings de até 30 caracteres. Na tabela, especificamos também que `empno` é a coluna de chave primária. Ou seja, não devemos ter valores duplicados na tabela.

Na execução bem-sucedida desse comando, veremos a seguinte mensagem:

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

Da mesma forma, vejamos como criar a tabela `salary`:

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

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

Em seguida, vamos adicionar uma restrição de chave estrangeira à coluna `empno` da tabela `salary`, referenciando a coluna `empno` da tabela `emps`:

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

Esse comando cria o relacionamento entre as tabelas `salary` e `empno`. Isso estabelece que um número de funcionários na tabela `salary` deve corresponder a um número de funcionários na tabela `emps`. Com essa restrição, garantimos que não possamos inserir uma linha na tabela `salary`, caso não tenha uma linha correspondente na tabela `emps`.

Até agora, como a tabela `salary` não tem linhas, a operação `ALTER TABLE` não afeta as linhas, como pode ser visto na mensagem resultante:

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

Por último, criamos a tabela `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)
```

Desta vez, adicionamos uma restrição de chave estrangeira dentro do comando `CREATE TABLE`, definindo, assim, a chave estrangeira imediatamente após a criação da tabela.

Inserindo dados no banco de dados

Agora estamos prontos para inserir linhas em nossas tabelas recém-criadas. Apesar de podermos fazer isso com o prompt `mysql>`, esse tipo de operação geralmente é realizado a partir de uma aplicação. Vamos interagir com o banco de dados a partir de nosso código Python por meio do driver MySQL Connector/Python. É possível instalá-lo via `pip`, da seguinte forma:

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

Vamos executar o seguinte script para preencher as tabelas de banco de dados com dados:

```
import mysql.connector

try:
1 cnx = mysql.connector.connect(user='root', password='sua senha',
                                host='127.0.0.1',
                                database='sampledb')
2 cursor = cnx.cursor()
  # Definindo linhas de funcionários
3 emps = [
   (9001, "Jeff Russell", "sales"),
   (9002, "Jane Boorman", "sales"),
   (9003, "Tom Heints", "sales")
  ]
  # Definindo a query
```

```
 4 query_add_emp = ("""INSERT INTO emps (empno, empname, job)
                       VALUES (%s, %s, %s)""")
   # Inserindo as linhas de funcionários
   for emp in emps:
   5 cursor.execute(query_add_emp, emp)
   # Definindo e inserindo salários
   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)
   # Definindo e inserindo ordens
   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)
   # Tornando as inserções permanentes no banco de dados
 6 cnx.commit()
 7 except mysql.connector.Error as err:
   print("Error-Code:", err.errno)
   print("Error-Message: {}".format(err.msg))
 8 finally:
   cursor.close()
   cnx.close()
```

Nesse script, importamos o driver MySQL Connector/Python como `mysql.connector`. Em seguida, abrimos um bloco `try/except`, que fornece um modelo para todas as operações relacionadas ao banco de dados que precisamos executar em nosso script. No bloco `try`, escrevemos o código para a operação e, se ocorrer um erro quando a operação for realizada, a execução será transferida para o bloco `except`.

Dentro do bloco `try`, começamos estabelecendo uma conexão com o banco de

dados, especificando o nome de usuário e senha, o endereço IP do host (nesse caso, seu host local) e o nome do banco de dados `1`. Depois, obtemos um objeto `cursor` relacionado a essa conexão `2`. O objeto `cursor` fornece os meios para a execução da instrução, bem como a interface para buscar os resultados.

Definimos as linhas para a tabela `emps` como uma lista de tuplas `3`. Logo depois, definimos a instrução SQL a ser executada para inserir essas linhas na tabela `4`. Nesta instrução `INSERT`, especificamos os campos a serem preenchidos com dados, com os placeholders `%s` que mapeiam esses campos para os membros de cada tupla. Em um loop, executamos a instrução, inserindo as linhas uma de cada vez com o método `cursor.execute()` `5`. Do mesmo jeito, inserimos linhas nas tabelas `salary` e `orders`. No final do bloco `try`, tornamos todas as inserções do banco de dados permanentes com o método `commit()` da conexão `6`.

Se qualquer operação relacionada ao banco de dados falhar, o restante da cláusula `try` é ignorado e a cláusula `except` é executada `7`, exibindo um código de erro gerado pelo servidor MySQL com a mensagem de erro correspondente.

A cláusula `finally` é executada de qualquer forma `8`. Nessa cláusula, fechamos explicitamente o `cursor` e, depois, a conexão.

Consultando o banco de dados

Agora que já populamos as tabelas com dados, podemos consultar esses dados para usá-los no código Python. Digamos que você queira acessar todas as linhas na tabela `emps` em que `empno` é maior que `9001`. Para tal, usaremos o script da seção anterior como modelo, alterando apenas o bloco `try` da seguinte forma:

```
  --trecho de código omitido--
  try:
    cnx = mysql.connector.connect(user='root', password='sua_senha',
                                  host='127.0.0.1',
                                  database='sampledb')
    cursor = cnx.cursor()
    query = ("SELECT 1 * FROM emps WHERE 2 empno > %s")
  3 empno = 9001
  4 cursor.execute(query, (empno,))
```

```
5 for (empno, empname, job) in cursor:
    print("{}, {}, {}".format(
      empno, empname, job))
--trecho de código omitido--
```

Diferentemente da operação de inserção, selecionar linhas não exige que executemos múltiplas operações `cursor.execute()` em um loop, uma para cada linha. Em vez disso, escreveremos uma query especificando os critérios para as linhas que queremos selecionar e, em seguida, vamos acessá-las todas de uma só vez com uma única operação `cursor.execute()`.

Na instrução `SELECT` que compõe nossa query, especificamos símbolo de asterisco (`*`), significando que desejamos ver todos os campos nas linhas obtidas `1`. Na cláusula `WHERE`, especificamos a condição que uma linha deve atender para ser selecionada. Aqui, indicamos quais linhas devem ter um `empno` maior que o valor da variável vinculada ao placeholder `%s` `2`. A variável `empno` está vinculada ao placeholder durante a execução `3`. Ao executarmos a query com `cursor.execute()`, passamos a variável binding dentro de uma tupla como segundo parâmetro `4`. O método `execute()` exige que as variáveis binding sejam passadas dentro de uma tupla ou de um dicionário, mesmo que só precisemos passar uma única variável.

Acessamos as linhas obtidas por meio do objeto `cursor`, iterando-o com um loop. Cada linha é acessível como uma tupla cujos itens representam os valores dos campos da linha `5`. Aqui, simplesmente exibimos os valores dos campos, gerando os resultados linha por linha, deste modo:

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

É possível também escrever instruções `SELECT` que façam o join de linhas de tabelas diferentes. O join de tabelas de banco de dados relacional retrata o processo de join dos DataFrames do pandas, conforme analisado no Capítulo 3. Normalmente, fazemos o join de tabelas por meio do relacionamento de chave estrangeira que definimos quando configuramos o banco de dados.

Por exemplo, suponha que você queira efetuar o join das tabelas `salary` e `emps`, mantendo a condição de que `empno` seja maior que `9001`. Isso é possível por meio das colunas `empno` compartilhadas, já que, na tabela `salary`, definimos

empno como chave estrangeira que referencia empno na tabela emps. Podemos implementar esse join com outra modificação no bloco try de nosso script:

```
--trecho de código omitido--
try:
  cnx = mysql.connector.connect(user='root', password='sua_senha',
                                host='127.0.0.1',
                                database='sampledb')
  cursor = cnx.cursor()
  query = ("""SELECT 1 e.empno, e.empname, e.job, s.salary
              FROM 2 emps e JOIN salary s ON 3 e.empno = s.empno
              WHERE 4 e.empno > %s""")

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

Desta vez, a query contém uma instrução SELECT que faz o join das tabelas salary e emps. Na lista SELECT, especificamos as colunas de ambas as tabelas que queremos incluir no join 1. Na cláusula FROM, especificamos as duas tabelas, unindo-as com a palavra reservada JOIN, com os aliases e e s, necessários para diferenciar colunas com o mesmo nome em ambas as tabelas 2. Na cláusula ON, definimos a condição join, afirmando que os valores nas colunas empno de ambas as tabelas devem corresponder 3. Na cláusula WHERE, como no exemplo anterior, usamos o placeholder %s para definir o valor mínimo de empno 4.

O script gera as seguintes linhas, com o salário de cada funcionário associado ao seu registro a partir da tabela emps:

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

EXERCÍCIO #8: EXECUTANDO UM JOIN UM-PARA-MUITOS

Altere o código mostrado na seção anterior para que a query seja uma join da tabela emps e da tabela orders. Você pode manter a condição de que empno deve ser maior que 9001. Ajuste a chamada print() a fim de gerar as linhas do join modificado.

Usando ferramentas de análise de banco de dados

No MySql, quando persistimos dados, podemos utilizar as ferramentas built-

in de análise do banco de dados, como o analytical SQL, para reduzir significativamente o volume de dados enviados entre a aplicação e o banco de dados. O *Analytical SQL* é um conjunto especial de comandos desenvolvido para analisar com eficácia dados armazenados em um banco de dados, em vez de simplesmente armazenar, obter e atualizar dados. Por exemplo, suponha que você queira importar somente os dados do mercado de ações relacionados àquelas empresas cujos preços não caíram mais de 1% abaixo do preço do dia anterior em determinado período. É possível fazer essa análise preliminar com o analytical SQL, evitando ter de carregar um conjunto inteiro de dados de preços de ações do banco de dados para seu script do Python.

Para ver como isso funciona, basta obter os dados das ações por meio da biblioteca yfinance, apresentada no Capítulo 3 e os armazenar em uma tabela de banco de dados. Em seguida, consulte a tabela a partir de um script do Python, carregando somente a parte dos dados de ações que satisfaz a condição especificada. Para começar, é necessário criar uma tabela em nosso banco de dados `sampledb` para armazenar os dados de ações. A tabela deve conter três colunas: `ticker`, `date` e `price`. Digite o seguinte comando no prompt `mysql>`:

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

Agora, use este script para obter alguns dados de ações com a yfinance:

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

Primeiro, definimos uma lista vazia chamada `data` que será populada com os dados de ações `1`. Como vimos anteriormente no capítulo, o método `cursor.execute()` espera dados na forma de um objeto de lista quando executa uma instrução `INSERT`. Depois, definimos uma lista de tickers para os quais desejamos obter dados `2`. Em seguida, em um loop, passamos cada ticker da lista `tickers` para a função `Ticker()` da yfinance `3`. A função retorna um objeto `Ticker`, cujo método `history()` fornece os dados relacionados ao ticker correspondente. Neste exemplo, obtemos dados de ações para cada ticker nos últimos cinco dias úteis (`period='5d'`).

O método `history()` retorna os dados de ações como um DataFrame do pandas com a coluna `Date` como índice. Em última análise, queremos converter esse DataFrame em uma lista de tuplas para inserção no banco de dados. Como precisamos incluir a coluna `Date` em nosso conjunto de dados, removemos o índice com o método `reset_index()` do DataFrame, transformando `Date` em uma coluna regular `4`. Em seguida, pegamos somente as colunas `Date` e `Close` do DataFrame obtido, `Close` contém os preços das ações no final do dia, e as convertemos em um array de registro do NumPy, etapa intermediária no processo de conversão dos dados de entrada `5`. Então convertemos os dados em uma lista de tuplas `6`. Depois disso, ainda precisamos reformatar cada tupla para que possa ser inserida na tabela do banco de dados `stocks` como uma linha. Em termos específicos, cada campo `Date` contém muitas informações irrelevantes (horas, minutos, segundos e assim por diante). Ao pegar somente os primeiros 10 caracteres do campo 0 em cada tupla, extraímos o ano, o mês e o dia, tudo o que precisamos para nossa análise `7`. Por exemplo, `2022-01-06T00:00:00.000000000` se tornaria simplesmente `2022-01-06`. Por último, ainda dentro do loop, anexamos as tuplas relacionadas ao ticker à lista `data` `8`.

Como resultado, vejamos o conteúdo da lista de tuplas `data`:

```
[
('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)
]
```

Para inserir esse conjunto de dados na tabela `stocks` como um conjunto de linhas, anexamos o seguinte código ao script anterior e o executamos mais uma vez:

```
import mysql.connector
from mysql.connector import errorcode
try:
  cnx = mysql.connector.connect(user='root', password='sua_senha',
                                host='127.0.0.1',
                                database='sampledb')
  cursor = cnx.cursor()
  # Definindo a query
  query_add_stocks = ("""INSERT INTO stocks (ticker, date, price)
                         VALUES (%s, %s, %s)""")
  # Adicionando as linhas de preço das ações
1 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()
```

O código adota o mesmo modelo usado antes para inserir dados no banco de dados. Desta vez, no entanto, recorremos ao método `cursor.executemany()`, que possibilita executar eficientemente a instrução `INSERT` múltiplas vezes para cada tupla na lista de tuplas `data` 1.

Agora que temos os dados no banco de dados, podemos brincar com as queries usando o analytical SQL, tentando responder a perguntas. Por exemplo, para filtrar ações que caíram mais de 1% abaixo do preço do dia anterior, como sugerido no início desta seção, precisaremos de uma query que analise os preços para o mesmo ticker em múltiplos dias. Como primeira etapa, a query a seguir gera um conjunto de dados com o preço atual das ações e o preço do dia anterior na mesma linha. Vamos testá-la no prompt `mysql>`:

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

Na lista `SELECT`, a função `LAG()` é uma função do analytical SQL. Essa função possibilita que acessemos os dados a partir de uma linha anterior da linha atual. A cláusula `PARTITION BY` dentro da cláusula `OVER` divide o conjunto de dados em grupos, um para cada ticker. A função `LAG()` é aplicada separadamente dentro de cada grupo, assegurando que os dados não vazem de um ticker para o outro. O resultado gerado pela query será mais ou menos assim:

```
+------------+--------+---------+------------+
| 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)
```

A query gerou uma coluna nova, `prev_price`, contendo os preços das ações do dia anterior. Conforme podemos verificar, a `LAG()` basicamente fornece acesso a duas linhas de dados na mesma linha. Ou seja, podemos manipular dados de ambas as linhas dentro da mesma expressão matemática como parte de uma query. Por exemplo, podemos dividir um preço pelo outro a fim de calcular a variação percentual de um dia para o outro. Considerando isso, vejamos a seguir uma query para atender ao requisito original, selecionando apenas as linhas daqueles tickers cujos preços não caíram mais de 1% abaixo do preço do dia anterior durante o período especificado:

```
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;
```

A instrução SQL é um join entre duas queries diferentes geradas na mesma tabela: `stocks`. A primeira query do join obtém todas as linhas da tabela `stocks` `1`, enquanto a segunda query obtém apenas os nomes dos tickers cujos preços caíram 1% ou mais abaixo do preço do dia anterior, pelo menos, uma vez durante o período de análise `2`. A segunda query do join tem uma estrutura complexa: seleciona dados de uma subquery e não diretamente da tabela `stocks`. A subquery, que começa em `3`, obtém as linhas da tabela cujos valores no campo `price` são, pelo menos, 1% menores do que na linha anterior. Definimos isso dividindo `price` por `LAG(price)` `4` e verificando se o resultado é inferior a `0.99` `5`. Em seguida, na lista `SELECT` da query principal, aplicamos a função `DISTINCT()` ao campo `ticker` a fim de eliminarmos nomes de

ticker duplicados do conjunto de resultados `2`.

Criamos o join das queries na coluna `ticker` `6`. Na cláusula `WHERE`, instruímos o join a obter apenas as linhas em que nenhuma correspondência é encontrada entre o campo `a.ticker` (tickers cujo preço caiu mais de 1%) e o campo `s.ticker` (todos os tickers) `7`. Como temos um left join, somente as linhas correspondentes da primeira query são obtidas. Como resultado, o join retorna todas as linhas da tabela `stocks` com um ticker não encontrado entre os tickers obtidos pela segunda query.

Tendo em conta os dados de ações mostrados anteriormente, o conjunto de resultados gerado pela query é o seguinte:

```
+--------+------------+---------+
| 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)
```

Como podemos verificar, nem todas as linhas da tabela `stocks` foram obtidas. Em particular, não encontraremos as linhas relacionadas aos tickers FB e TSLA. Este último, por exemplo, foi excluído devido à seguinte linha encontrada na saída gerada pela query anterior:

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

Essa linha mostra uma queda de 3,54%, excedendo o limite de 1%.

No script a seguir, geramos a mesma query de dentro do código Python e

buscamos os resultados em um DataFrame do pandas:

```
import pandas as pd
import mysql.connector
from mysql.connector import errorcode
try:
  cnx = mysql.connector.connect(user='root', password='sua_senha",
                                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""")
1 df_stocks = pd.read_sql(query, con=cnx)
2 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()
```

O script se parece essencialmente com os mostrados anteriormente no capítulo. A principal diferença é que carregamos os dados do banco de dados diretamente para um DataFrame do pandas. Para isso, recorremos ao método pandas `read_sql()`, que aceita uma query SQL como string e como primeiro parâmetro e um objeto de conexão de banco de dados como segundo parâmetro `1`. Em seguida, definimos as colunas `ticker` e `data` como o índice do DataFrame `2`.

Considerando os dados de ações mostrados anteriormente, vejamos o DataFrame `df_stocks` resultante:

```
                    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
```

Agora que temos os dados em um DataFrame, podemos prosseguir com uma análise mais aprofundada dentro do Python. Por exemplo, suponha que você queira calcular o preço médio de cada ticker durante um período definido. No próximo capítulo, veremos como resolver esses problemas, aplicando uma função agregada e adequada em nível de grupo em um DataFrame.

Bancos de dados NoSQL

Os *bancos de dados NoSQL*, ou *bancos de dados não relacionais*, não exigem um esquema organizacional predeterminado para os dados armazenados e não oferecem suporte a operações padrão de banco de dados relacionais, como joins. Ao contrário, esses bancos possibilitam maneiras de armazenar dados com mais flexibilidade estrutural, facilitando a manipulação de grandes volumes de dados. Por exemplo, armazenamentos de chave-valor, um tipo de banco de dados NoSQL, possibilitam armazenar e obter dados pares de chave-valor como pares de tempo-evento. Os bancos de dados orientados a documentos, outro tipo de banco de dados NoSQL, são arquitetados para trabalhar com contêineres de dados estruturados de forma flexível, como documentos JSON. Isso possibilita armazenar todas as informações relacionadas a um determinado objeto como única entrada no banco de dados, em vez de dividir as informações em múltiplas tabelas, comum em bancos de dados relacionais.

Ainda que não existam há tanto tempo quanto seus equivalentes relacionais, os bancos de dados NoSQL mais do que depressa se tornaram populares, pois possibilitam que os desenvolvedores armazenem dados em formatos simples e diretos e não exigem conhecimentos avançados para acessar e manipular os dados. Essa flexibilidade os torna especialmente adequados para aplicações de big data em tempo real, como o Gmail do Google ou o LinkedIn.

NOTA *Não existe consenso sobre a origem do termo NoSQL. Alguns dizem*

que significa não SQL, ao passo que outros afirmam que representa não apenas banco de dados não SQL. Ambos são adequados: os bancos de dados NoSQL armazenam dados em um formato diferente das tabelas relacionais (SQL) e, ao mesmo tempo, muitos desses bancos de dados suportam queries do tipo SQL.

Armazenamentos de chave-valor

Um *armazenamento de chave-valor* é um banco de dados que contém pares de chaves-valor, parecido com um dicionário Python. Um bom exemplo de armazenamento de chave-valor é o Redis, que significa Remote Dictionary Service. O Redis suporta comandos como `GET`, `SET` e `DEL` para acessar e manipular pares chave-valor, como ilustrado no exemplo simples a seguir:

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

Aqui, usamos o comando `SET` para criar a chave `emp1` com o valor `Maya Silver` e, em seguida, utilizamos `GET` a fim de obter o valor por meio de sua chave.

Configurando o Redis

Para explorarmos o Redis, é necessário instalá-lo. Podemos encontrar detalhes na página Redis Quick Start em https://redis.io/topics/quickstart. Após instalar o servidor Redis em nosso sistema, precisaremos também instalar o redis-py, a biblioteca do Python que viabiliza interagir com Redis a partir do nosso código Python. Vamos fazer isso com o comando `pip`:

```
$ pip install redis
```

Em seguida, importamos o redis-py para o script com o comando `import redis`.

Acessando o Redis com o Python

Vejamos a seguir um exemplo simples de acesso ao servidor Redis a partir do Python por meio da biblioteca redis-py:

```
  > import redis
1 > r = redis.Redis()
2 > r.mset({"emp1": "Maya Silver", "emp2": "John Jamison"})
```

```
  True
3 > r.get("emp1")
  b'Maya Silver'
```

Usamos o método `redis.Redis()` a fim de definir uma conexão com o servidor Redis `1`. Como os parâmetros do método são omitidos, valores padrão serão adotados, pressupondo-se que o servidor esteja sendo executado em sua máquina local: `host='localhost'`, `port=6379` e `db=0`.

NOTA *Os bancos de dados do Redis usam números como indexação baseada em zero. Conexões novas usam o banco de dados 0 por padrão.*

Depois de estabelecer uma conexão, utilizamos o método `mset()` para definir múltiplos pares chave-valor `2` (*m* é a abreviação de *multiple*). O servidor retorna `True` quando os dados forem armazenados com sucesso. Podemos então obter o valor de qualquer uma das chaves armazenadas com o método `get()` 3.

Como qualquer outro banco de dados, o Redis possibilita persistirmos os dados que estão sendo inseridos. Assim, conseguimos obter um valor por sua chave em outra sessão ou script do Python. Além disso, o Redis possibilita definirmos uma *expire flag* (flag de expiração) em uma chave ao definirmos um par chave-valor, especificando por quanto tempo deve ser retido. Nas aplicações em tempo real, isso pode ajudar e muito, já que os dados de entrada se tornam irrelevantes após determinado período de tempo. Por exemplo, se tivermos um app de serviço de táxi, talvez devêssemos armazenar os dados sobre a disponibilidade de cada táxi individual. Como esses dados estariam sujeitos a alterações frequentes, queremos que expirem após um curto período de tempo. Vejamos como isso pode funcionar:

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

Recorremos ao método `setex()` para definir um par chave-valor que será removido automaticamente do banco de dados após um período de tempo especificado. Aqui, especificamos o tempo de expiração como um objeto `timedelta`. Outra alternativa é especificá-lo como um número em segundos.

Até agora, analisamos somente pares simples de chave-valor, mas também podemos armazenar múltiplas informações sobre o mesmo objeto com o Redis, conforme exemplificado a seguir:

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

Começamos definindo um dicionário do Python, que pode conter um número arbitrário de pares chave-valor. Em seguida, enviamos todo o dicionário para o banco de dados, armazenando-o com a chave `cab48` usando `hmset()` (*h* é abreviação de *hash*). Depois, utilizamos a função `hgetall()` para obter todos os pares chave-valor armazenados na chave `cab48`.

Bancos de dados orientados a documento

Um *banco de dados orientado a documento* armazena cada registro como um documento separado. Em vez de ter de se ajustar a um esquema predefinido, como os campos de uma tabela de banco de dados relacional, em um banco de dados orientado a documento, cada documento pode ter a própria estrutura. Essa flexibilidade faz dos bancos de dados orientados a documentos a categoria mais popular de bancos de dados NoSQL e, entre os bancos de dados orientados a documentos, o MongoDB é sem sombras de dúvidas o líder em uso. O MongoDB é arquitetado para gerenciar coleções de documentos semelhantes ao JSON. Nesta seção, exploraremos como trabalhar com o MongoDB.

Configurando o MongoDB

Existem diversas maneiras de testar o MongoDB. Uma delas é instalar o banco de dados MongoDB em seu sistema. Para mais detalhes, confira a documentação do MongoDB em https://docs.mongodb.com/manual/installation. Outra opção que não exige sobrecarga de instalação é criar um banco de dados MongoDB hospedado gratuito com o MongoDB Atlas. É necessário se registrar em https://www.mongodb.com/cloud/atlas/register.

Antes de começar a interagir com um banco de dados MongoDB a partir do Python, precisaremos instalar o PyMongo, o driver oficial do Python para o

MongoDB. Faremos essa instalação com o comando `pip`:

```
$ pip install pymongo
```

Como acessar o MongoDB com o Python

A primeira etapa para trabalhar com MongoDB usando o Python é estabelecer uma conexão com o servidor de banco de dados por meio de um objeto `MongoClient` do PyMongo, como mostrado a seguir:

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

A string de conexão pode ser um URI de conexão do MongoDB, como *mongodb://localhost:27017*. Essa string de conexão assume que instalamos o MongoDB em nosso sistema local. Caso esteja usando o MongoDB Atlas, precisará usar uma string de conexão fornecida pelo Atlas. Para obter mais detalhes, confira a página "Connect via Driver" na documentação do Atlas em https://docs.atlas.mongodb.com/driver-connection. Você também pode conferir a página "Connection String URI Format" da documentação do MongoDB em https://docs.mongodb.com/manual/reference/connection-string.

Em vez de usar uma string de conexão, é possível especificar o host e a porta como parâmetros separados do construtor `MongoClient()`:

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

Uma única instância do MongoDB pode suportar múltiplos bancos de dados. Assim sendo, uma vez que uma conexão com o servidor é estabelecida, precisamos especificar o banco de dados com o qual queremos trabalhar. Como o MongoDB não fornece um comando separado para criar um banco de dados, usamos a mesma sintaxe para criar um banco de dados novo e acessar um existente. Por exemplo, para criar um banco de dados chamado `sampledb` (ou acessá-lo, se já existir), podemos recorrer à seguinte sintaxe semelhante a um dicionário:

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

Ou podemos utilizar a sintaxe de acesso ao atributo:

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

Ao contrário dos bancos de dados relacionais, o MongoDB não armazena dados em tabelas. Em vez disso, os documentos são agrupados em *coleções*.

Criar ou acessar uma coleção é semelhante a criar ou acessar um banco de dados:

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

No banco de dados `sampleb`, esse comando criará a coleção `emps`, caso ainda não tenha sido criada. Depois, passamos o método `insert_one()` para inserir um documento na coleção. Neste exemplo, inserimos um documento `emp` formatado como um dicionário:

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

Ao inserir o documento, um campo `"_id"` é adicionado automaticamente a ele. O valor desse campo é gerado para ser exclusivo em toda a coleção. É possível acessar o ID por meio do campo `insert_id` do objeto retornado por `insert_one()`:

```
> result.inserted_id
ObjectId('69y67385ei0b650d867ef236')
```

Agora que temos alguns dados no banco de dados, como podemos consultá-los? O tipo mais comum de consulta pode ser realizado via query com o `find_one()`, que retorna um único documento correspondente aos critérios de pesquisa:

```
> emp = emps_collection.find_one({"empno": 9001})
> print(emp)
```

Conforme podemos observar, o `find_one()` não exige que usemos o ID do documento, adicionado automaticamente na inserção. Ao contrário, podemos consultar elementos específicos, supondo que o documento resultante corresponda a eles.

Vejamos o resultado:

```
{
u'empno': 9001,
u'_id': ObjectId('69y67385ei0b650d867ef236'),
u'empname': u'Jeff Russell',
u'orders': [2608, 2617, 2620]
}
```

EXERCÍCIO #9: INSERINDO E CONSULTANDO MÚLTIPLOS DOCUMENTOS

Na seção anterior, aprendemos a inserir ou obter um único documento em um banco de dados MongoDB. Continuando com a coleção `emps` criada no banco de dados `sampledb`, tente executar inserções em massa com o método `insert_many()` e, em seguida, consulte mais de um documento usando o método `find()` via query. Para mais detalhes sobre como usar esses métodos, confira a documentação do PyMongo em `https://pymongo.readthedocs.io/en/stable`.

Recapitulando

Neste capítulo, vimos exemplos de movimentação de dados a partir de e para bancos de dados de diferentes tipos, incluindo bancos de dados relacionais e NoSQL. Trabalhamos com o MySQL, um dos bancos de dados relacionais mais populares. Em seguida, analisamos o Redis, uma solução NoSQL que possibilita armazenar e obter eficientemente pares chave-valor. Exploramos também o MongoDB, indiscutivelmente o banco de dados NoSQL mais popular da atualidade, que possibilita trabalhar com documentos semelhantes ao JSON de maneira amigável ao Python.

CAPÍTULO 6
Agregando dados

Não raro, para obter o máximo valor de tomada de decisão a partir de nossos dados, precisamos agregá-los. A *agregação* é um processo de coleta de dados que nos possibilita apresentá-los de forma resumida: agrupados por subtotais, por totais, por médias e por outras métricas estatísticas. Neste capítulo, exploraremos técnicas de agregação do pandas e estudaremos como usá-las para analisar nossos dados.

Com a agregação, é possível ter uma ideia geral e eficiente de um grande conjunto de dados e, assim, responder às perguntas sobre os valores nos dados. Por exemplo, talvez uma empresa varejista de grande porte queira verificar o desempenho de um produto com base na marca ou queria analisar os totais de vendas em diferentes regiões. O proprietário de um site pode querer identificar recursos mais atrativos com base no número de visitantes. Talvez um climatologista precise determinar os lugares mais ensolarados de uma região com base na média anual de dias ensolarados.

Ao coletar valores de dados específicos e apresentá-los em um formato resumido, a agregação de dados pode responder a perguntas como essas. Como apresenta informações baseadas em conjuntos de dados relacionados, a agregação implica primeiro agrupar os dados por um ou mais atributos. No caso do varejista de grande porte, talvez seja necessário agrupar os dados por marca ou por região e data.

Nos exemplos a seguir, veremos como o agrupamento, com as *funções de agregação* aplicadas a cada grupo de linhas, pode ser implementado ao usar DataFrames do pandas. Uma função de agregação retorna uma única linha de resultado com base em um grupo inteiro de linhas, formando, assim, uma única linha resumida e agregada para cada grupo.

Dados para agregação

Para vermos como funciona a agregação, criaremos um conjunto de exemplos de DataFrames contendo dados de vendas para uma varejista online de moda outdoor. Os dados incluirão valores como números e datas de pedidos; detalhes sobre os itens comprados em cada pedido, como preço e quantidade; os funcionários que atendem a cada pedido; e os locais dos armazéns de fulfillment da empresa. Em uma aplicação real, esses dados provavelmente seriam armazenados em um banco de dados que poderíamos acessar a partir de nosso código Python, conforme descrito no Capítulo 5. Para simplificar as coisas, carregaremos os dados nos DataFrames a partir de listas de tuplas. Acesse o GitHub deste livro para fazer o download das listas de tuplas.

Vamos começar com alguns pedidos de amostra. A lista denominada `orders` contém diversas tuplas, cada uma representando um pedido. Cada tupla tem três campos: o número do pedido, a data e o ID do funcionário que atendeu o pedido, respectivamente:

```
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)
]
```

Importe o pandas e carregue a lista em um DataFrame da seguinte forma:

```
import pandas as pd
df_orders = pd.DataFrame(orders, columns =['OrderNo', 'Date', 'Empno'])
```

Em geral, os detalhes do pedido (também conhecidos como *linhas do pedido*) são armazenados em outro contêiner de dados. Nesse caso, temos uma lista de tuplas chamada `details` que carregaremos em outro DataFrame. Cada tupla representa a linha de um pedido, com campos correspondentes ao número do pedido, nome do item, marca, preço e quantidade:

```
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)
]
# Convertendo a lista em um DataFrame
df_details = pd.DataFrame(details, columns =['OrderNo', 'Item', 'Brand', 'Price',
'Quantity'])
```

Armazenaremos informações sobre os funcionários da empresa em um terceiro DataFrame. Vamos criá-lo a partir de outra lista de tuplas chamada `emps` com números de funcionários, nomes e locais:

```
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'])
```

Finalmente, temos a cidade e a região de cada armazém em uma lista de tuplas chamada `locations`, que armazenaremos em um quarto DataFrame:

```
locations = [
('LA', 'West'),
('San Francisco', 'West'),
('NYC', 'East'),
('Philadelphia', 'East')
]

df_locations = pd.DataFrame(locations, columns =['Location', 'Region'])
```

Agora que carregamos os dados em DataFrames, é possível agregá-los de muitas maneiras, possibilitando responder a todos os tipos de perguntas sobre a situação dos negócios da empresa. Por exemplo, talvez você queira analisar o desempenho de vendas em diferentes regiões, gerando subtotais por data.

Para tal, primeiro é necessário combinar os dados relevantes em um único DataFrame. Em seguida, podemos agrupar os dados e aplicar as funções de agregação aos grupos.

Combinando DataFrames

Muitas vezes, precisaremos coletar dados a partir de muitos contêineres diferentes antes de ter tudo o que queremos para gerar a agregação desejada. Nosso exemplo não foge à regra. Até mesmo os dados que representam os pedidos são distribuídos entre dois DataFrames diferentes: `df_orders` e `df_details`. Nosso objetivo é gerar as somas de vendas por região e por data. Precisamos combinar quais DataFrames e quais colunas de cada um devemos incluir?

Como precisamos somar os números de vendas, teremos de incluir as colunas `Price` e `Quantity` de `df_details`. Devemos incluir também a coluna `Date` de `df_orders` e a coluna `Region` de `df_locations`. Ou seja, teremos de fazer o join dos seguintes DataFrames: `df_orders`, `df_details` e `df_locations`.

É possível fazer o join dos DataFrames `df_orders` e `df_details` diretamente com uma única chamada ao método `merge()` do pandas, da seguinte forma:

```
df_sales = df_orders.merge(df_details)
```

NOTA *Para aperfeiçoar o método* `merge()`*, confira "Joins de um-para-muitos" no Capítulo 3.*

Fazemos o join dos DataFrames com base na coluna `OrderNo`. Desnecessário especificá-los explicitamente, porque a coluna `OrderNo` está presente em ambos os DataFrames e, portanto, é escolhida por padrão. Agora, o DataFrame recém-mesclado contém um registro para cada linha de pedido, como em `df_details`, mas com as informações adicionadas dos registros correspondentes de `df_orders`. Para vermos os registros no DataFrame que passou pelo merge, basta exibi-lo:

```
print(df_sales)
```

Vejamos o conteúdo de `df_sales`:

```
    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
```

Como podemos observar na coluna `Quantity`, pode haver mais de um item em uma única linha de pedido. Sendo assim, precisamos multiplicar os valores dos campos `Price` e `Quantity` para calcular o total de uma linha de pedido. É possível armazenar o resultado dessa multiplicação em um campo novo do DataFrame, assim:

```
df_sales['Total'] = df_sales['Price'] * df_sales['Quantity']
```

Isso adiciona uma coluna `Total` ao DataFrame, além das sete colunas existentes. Agora, temos a opção de remover as colunas de que não precisamos para gerar as somas de vendas por região e data. Nesta etapa, precisamos reter somente as colunas `Date`, `Total` e `Empno`. As duas primeiras, obviamente, serão essenciais para nossos cálculos. Falaremos sobre a necessidade da coluna `Empno` daqui a pouco.

Vamos filtrar o DataFrame para as colunas necessárias, passando uma lista dos nomes das colunas para o operador `[]` do DataFrame, conforme mostrado a seguir:

```
df_sales = df_sales[['Date','Empno','Total']]
```

Agora, é necessário fazermos o join do DataFrame `df_sales` que acabamos de criar com o DataFrame `df_regions`. No entanto, não podemos fazer o join diretamente, pois esses DataFrames não têm nenhuma coluna em comum. Teremos de fazer o join com o DataFrame `df_emps`, que compartilha uma coluna com `df_sales` e outra com `df_regions`. Em termos específicos, podemos fazer o join de `df_sales` e `df_emps` na coluna `Empno`. Por isso, mantivemos a coluna em `df_sales`, já que podemos fazer o join de `df_ emps` e de `df_locations`

na coluna `Location`. Vamos implementar esses joins com método `merge()`:

```
df_sales_emps = df_sales.merge(df_emps)
df_result = df_sales_emps.merge(df_locations)
```

Vamos exibir o DataFrame `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
```

Mais uma vez, talvez você queira remover as colunas desnecessárias e manter apenas aquelas de que realmente precisa. Desta vez, podemos descartar as colunas `Empno`, `Empname e Location`, ficando somente com as colunas `Date`, `Region` e `Total`:

```
df_result = df_result[['Date','Region','Total']]
```

Agora, vejamos como fica o conteúdo de `df_result`:

```
          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
```

Como não incluímos colunas desnecessárias, o DataFrame `df_result` agora está devidamente formatado para agregar os dados de vendas por região e por data.

Agrupamento e agregação dos dados

Para realizarmos cálculos agregados de nossos dados, primeiro devemos ordená-los em grupos relevantes. A função do pandas `groupby()` divide os dados de um DataFrame em subconjuntos que têm valores correspondentes para uma ou mais colunas. Para o nosso exemplo, é possível usar o `groupby()` para agrupar o DataFrame `df_result` por data e região. Em seguida, podemos aplicar a função do pandas de agregação `sum()` a cada grupo. É possível executar ambas as operações em uma única linha de código:

```
df_date_region = df_result.groupby(['Date','Region']).sum()
```

O primeiro agrupamento é baseado na coluna `Date`. Depois, dentro de cada data, agrupamos com base em `Region`. A função `groupby()` retorna um objeto ao qual aplicamos a função de agregação `sum()`. Essa função sintetiza os valores das colunas numéricas. Neste exemplo específico, `sum()` é aplicada somente à coluna `Total` porque se trata da única coluna numérica no DataFrame. (Se o DataFrame tivesse outras colunas numéricas, também aplicaríamos a função a essas colunas.) Teremos o seguinte DataFrame:

```
                   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
```

`Date` e `Region` são as colunas de índice do novo DataFrame. Juntas, formam um *índice hierárquico*, também conhecido como *índice multinível*, ou apenas *MultiIndex*.

Um MultiIndex possibilita trabalhar com dados com um número arbitrário de dimensões dentro da estrutura 2D de um DataFrame, usando múltiplas colunas para identificar unicamente cada linha. Em nosso caso, o DataFrame

`df_date_region` pode ser visto como um conjunto de dados 3D, com três eixos correspondentes a datas, a regiões e aos valores agregados (cada eixo representa a dimensão correspondente), conforme descrito na Tabela 6.1.

Tabela 6.1: As três dimensões do `dataFrame df_date_region`

Axis	Coordinates
Date	2022-02-04, 2022-02-05, 2022-02-06
Region	West, East
Aggregation	Total

NOTA *Neste contexto, as coordenadas são possíveis valores para determinado eixo.*

O MultiIndex do nosso DataFrame nos possibilita escrever queries que navegam pelas dimensões do DataFrame, acessando totais por data, região ou ambos. É possível identificar unicamente cada linha do DataFrame e acessar valores agregados selecionados em diferentes grupos de dados.

Visualizando agregações específicas por MultiIndex

A visualização de categorias específicas de informações em um DataFrame é requisito comum. Por exemplo, no caso do DataFrame `df_date_region` que acabamos de criar, talvez seja necessário obter os números de vendas agregados apenas de uma determinada data ou de uma região específica e de uma data particular ao mesmo tempo. Podemos usar o índice do DataFrame (ou MultiIndex) para encontrar a agregação necessária.

Para termos uma ideia de como trabalhar com um MultiIndex, será útil ver como cada valor MultiIndex é representado no Python. Para isso, podemos usar a propriedade `index` do DataFrame `df_date_region`:

```
print(df_date_region.index)
```

A propriedade `index` retorna todos os valores de índice, ou rótulos de linha, de um DataFrame, independentemente de esse DataFrame ter um índice simples ou um MultiIndex. Vejamos os valores MultiIndex para `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'])
```

Conforme podemos verificar, cada valor MultiIndex é uma tupla que podemos utilizar para acessar um valor de vendas correspondente no campo `Total`. Pensando nisso, vejamos como acessar os valores totais de uma determinada data e região:

```
df_date_region1 [df_date_region.index.isin(2 [('2022-02-05', 'West')])]
```

Inserimos a tupla que representa o MultiIndex desejado no operador `[] 2` e a passamos para o método `index.isin()` do pandas. O método exige que o parâmetro passado seja iterável (lista ou tupla), Series, DataFrame ou dicionário, por isso inserimos o MultiIndex desejado entre colchetes. O método retorna um array booleano indicando se os dados em cada um dos valores de índice do DataFrame correspondem ao(s) valor(es) de índice que especificamos: `True` para correspondências ou, caso contrário, `False`. Neste exemplo particular, o método `isin()` gera o array `[False, False, False, True, False, False]`, significando que o quarto valor do índice é uma correspondência.

Em seguida, passamos o array booleano para o DataFrame `df_date_region` no operador `[] 1`, resultando na seleção do valor de vendas correspondente, conforme mostrado a seguir:

```
                  Total
Date       Region
2022-02-05 West     35.0
```

Mas não precisamos nos limitar a obter somente uma única linha do DataFrame. É possível passar mais de um valor de índice para `index.isin()` a fim de obter um conjunto de números de vendas correspondentes, conforme mostrado a seguir:

```
df_date_region[df_date_region.index.isin([('2022-02-05', 'East'), ('2022-02-05',
'West')])]
```

Isso obterá as duas linhas de `df_date_region`:

```
                  Total
Date       Region
2022-02-05 East    160.0
           West     35.0
```

Apesar de esse exemplo específico usar dois índices vizinhos, na prática podemos passar qualquer índice para `index.isin()` em qualquer ordem, assim:

```
df_date_region[df_date_region.index.isin([('2022-02-06', 'East'),
               ('2022-02-04', 'East'), ('2022-02-05', 'West')])]
```

Vejamos como fica o conjunto de linhas obtidas:

```
                  Total
Date       Region
2022-02-04 East    97.0
2022-02-05 West    35.0
2022-02-06 East   110.0
```

Repare que a ordem dos registros obtidos corresponde à ordem dos registros no DataFrame, em vez da ordem em que especificamos os índices.

Fatiando um intervalo de valores agregados

Assim como é possível aplicar o fatiamento para obter um intervalo de valores de uma lista, podemos usá-lo para extrair um intervalo de valores agregados a partir de um DataFrame. Podemos fazer isso da seguinte forma: no DataFrame `df_date_region`, fornecemos duas tuplas especificando as chaves MultiIndex das posições inicial e final do intervalo da fatia. No exemplo a seguir, obtemos o intervalo de valores agregados das datas `2022-02-04` a `2022-02-05` para todas as regiões. Basta inserir as chaves MultiIndex inicial e final entre colchetes, separadas por dois-pontos:

```
df_date_region[('2022-02-04', 'East'):('2022-02-05', 'West')]
```

Como resultado, obtemos as seguintes linhas do DataFrame:

```
                  Total
Date       Region
2022-02-04 East    97.0
           West   243.0
2022-02-05 East   160.0
           West    35.0
```

Neste exemplo particular, como estamos obtendo os números de vendas de todas as regiões dentro de um intervalo de datas especificado, podemos desconsiderar os nomes das regiões e passar apenas as datas:

```
df_date_region['2022-02-04':'2022-02-05']
```

Isso deve retornar exatamente o mesmo resultado que o exemplo anterior.

Fatiamento dentro dos níveis de agregação

Talvez você queira fatiar as agregações dentro dos diferentes níveis de um índice hierárquico. Em nosso exemplo, o nível de agregação mais alto é o nível `Date`, dentro do qual temos o nível `Region`. Digamos que precisamos obter os números de vendas para uma fatia específica de datas, selecionando todo o conteúdo do nível `Region`. Podemos fazer isso com a função `slice()` do Python em conjunto com a propriedade `loc` do DataFrame, conforme ilustrado a seguir:

```
df_date_region.loc[(slice('2022-02-05', '2022-02-06'), slice(None)), :]
```

Aqui, usamos a `slice()` duas vezes. Na primeira instância, `slice()` define o intervalo fatiado para `Date`, o nível de agregação mais alto, gerando o objeto `slice` que especifica as datas inicial e final. Na segunda vez que invocarmos `slice()`, o faremos para o nível `Region` (o próximo nível mais baixo). Ao especificar `None`, selecionamos todo o conteúdo do nível `Region`. No operador `[]` da propriedade `loc`, incluímos também uma vírgula seguida de dois pontos (`:`). Essa sintaxe especifica que estamos utilizando rótulos de linha em vez de rótulos de coluna.

Vejamos o conjunto de resultados:

```
                    Total
Date       Region
2022-02-05 East    160.0
           West     35.0
2022-02-06 East    110.0
           West     86.0
```

No próximo exemplo, substituímos `slice(None)` por `slice('East')`, reduzindo, assim, os números de vendas obtidos para apenas linhas contendo `East`, retiradas de dentro do intervalo de datas especificado:

```
df_date_region.loc[(slice('2022-02-05', '2022-02-06'), slice('East')), :]
```

Isso obterá as seguintes linhas:

```
                   Total
Date       Region
2022-02-05 East    160.0
2022-02-06 East    110.0
```

É possível especificar um intervalo para o nível `Region` em vez de um único valor, assim como especificamos um intervalo para o nível `Date`. Neste

exemplo específico, no entanto, esse intervalo só poderia começar com `'East'` e terminar com `'West'`, implementado como `slice('East','West')`. Como se trata do intervalo máximo possível, uma chamada de `slice('East','West')` será igual a uma chamada `slice(None)`.

Adicionando um total geral

Quando se trata de agregar dados de vendas, talvez seja interessante calcular o total geral ou a soma de todos os outros totais de valor de vendas e adicioná-los ao DataFrame. No nosso exemplo, como todos os totais estão em um único DataFrame, `df_date_region`, podemos utilizar o método `sum()` do pandas para encontrar o total de vendas em todas as regiões e todas as datas. O método calcula a soma dos valores para um eixo especificado, conforme mostrado a seguir:

```
ps = df_date_region.sum(axis = 0)
print(ps)
```

Aqui, `sum()` retorna uma Series do pandas com a soma em relação à coluna `Total` no DataFrame `df_date_region`. Lembre-se de que não é necessário especificar a coluna `Total` na chamada para `sum()`, pois é aplicada automaticamente a quaisquer dados numéricos. Vejamos o conteúdo da Series `ps`:

```
Total    731.0
dtype: float64
```

Para anexar a Series recém-criada ao DataFrame `df_date_region`, primeiro devemos nomeá-la: Esse nome será usado como índice para a linha de total geral no DataFrame. Como as chaves de índice são tuplas no DataFrame `df_date_region`, usamos também uma tupla para o nome da Series:

```
ps.name=('All','All')
```

Na tupla, o primeiro `'All'` está relacionado ao componente `Date` da chave de índice, ao passo que o segundo está relacionado ao componente de chave de índice `Region`. Agora, podemos anexar a Series ao DataFrame:

```
df_date_region_total = df_date_region.append(ps)
```

Se exibirmos o DataFrame recém-criado, veremos o seguinte conteúdo:

```
                   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
```

Podemos acessar a linha de total geral por seu índice, como faríamos com qualquer outra linha em um DataFrame. Aqui, passamos a tupla que representa o índice da linha para o método `index.isin()`, conforme analisado anteriormente:

```
df_date_region_total[df_date_region_total.index.isin([('All', 'All')])]
```

Isso nos fornecerá a linha do total geral:

```
             Total
Date Region
All  All     731.0
```

Adicionando Subtotais

Além de somar o total geral, podemos também querer adicionar subtotais para cada data ao DataFrame. Vejamos o DataFrame resultante:

```
                  Total
Date       Region
2022-02-04 East     97.0
           West    243.0
           All     340.0
2022-02-05 East    160.0
           West     35.0
           All     195.0
2022-02-06 East    110.0
           West     86.0
           All     196.0
All        All     731.0
```

Gerar esse DataFrame exige algumas etapas. Primeiro, agrupe o DataFrame pelo nível `Date` do índice. Em seguida, itera sobre o objeto GroupBy resultante, acessando cada data com um conjunto de linhas (conhecido como *subframe*) contendo a região e o total de informações para essa data. Depois, selecione e anexe cada subframe a um DataFrame vazio, com uma linha de subtotal correspondente. Vejamos como fazer isso:

```
1 df_totals = pd.DataFrame()
  for date, date_df in 2 df_date_region.groupby(level=0):
  3 df_totals = df_totals.append(date_df)
  4 ps = date_df.sum(axis = 0)
  5 ps.name=(date,'All')
  6 df_totals = df_totals.append(ps)
```

Começamos criando um DataFrame vazio, `df_totals`, para receber os dados finais `1`. Em seguida, criamos um objeto GroupBy `2`, agrupando o DataFrame `df_date_region` pelo nível hierárquico superior (`level=0`) de seu índice (ou seja, a coluna `Date`) e inserimos um loop `for` para iterarmos o objeto GroupBy. A cada iteração, obtemos uma data e seu subframe correspondente. Anexamos o subframe ao DataFrame `3` `df_totals` e, depois, criamos a linha do subtotal na forma de uma Series com a soma das linhas do subframe `4`. Em seguida, nomeamos essa Series com sua data associada e `'All'` a fim de representar todas as regiões `5` e, então, anexamos a Series ao DataFrame `df_totals` `6`.

```
df_totals = df_totals.append(df_date_region_total.loc[('All','All')])
```

Como resultado, obtemos um DataFrame que tem a soma de vendas para cada data e o total geral para todas as datas.

EXERCÍCIO #10: EXCLUINDO O TOTAL DE LINHAS DO DATAFRAME

Em um DataFrame, as linhas para totais possibilita que você o use como um relatório sem ter de recorrer a outras etapas. No entanto, caso use o DataFrame em outras operações de agregação, talvez seja necessário excluir as linhas para totais.
Tente filtrar o DataFrame `df_totals` criado na seção anterior, excluindo o total geral e as linhas de subtotal. Use as técnicas de fatiamento analisadas neste capítulo.

Selecionando todas as linhas em um grupo

Além de auxiliar na agregação, a função `groupby()` também ajuda a selecionar todas as linhas pertencentes a um determinado grupo. Para isso, os objetos retornados por `groupby()` fornecem o método `get_group()`. Vejamos como isso funciona:

```
group = df_result.groupby(['Date','Region'])
group.get_group(('2022-02-04','West'))
```

Agrupamos o DataFrame `df_result` por `Date` e `Region`, passando os nomes das colunas como uma lista para `groupby()`, exatamente como fizemos antes. Em seguida, invocamos o método `get_group()` no objeto GroupBy resultante,

passando uma tupla com o índice desejado, retornando o seguinte DataFrame:

```
         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
```

Conforme podemos observar, o conjunto de resultados não é uma agregação. Ao contrário, inclui todas as linhas de pedidos relacionadas à data e à região especificadas.

Recapitulando

Neste capítulo, aprendemos que agregação é o processo de coletar dados e expressá-los em um formato resumido. Normalmente, esse processo envolve a divisão de dados em grupos e, em seguida, os cálculos dos resumos para cada grupo. Aqui, os exemplos mostraram como agregar dados contidos nos DataFrames do pandas, recorrendo aos métodos e às propriedades do DataFrame, como `merge()`, `groupby()`, `sum()`, `index` e `loc`. Aprendemos a tirar proveito do índice hierárquico de um DataFrame, ou MultiIndex, a fim de modelar relacionamentos multiníveis nos dados que estão sendo agregados. Aprendemos também a visualizar e dividir dados agregados seletivamente usando um MultiIndex.

CAPÍTULO 7

Combinando conjuntos de dados

É prática comum dividir os dados em múltiplos contêineres. Por isso, combinaremos com frequência diferentes conjuntos de dados em um. Nos capítulos anteriores, fizemos algumas combinações, mas, neste capítulo, usaremos técnicas mais avançadas para combinar conjuntos de dados.

Às vezes, para combinar conjuntos de dados, basta adicionar um conjunto ao final de outro. Por exemplo, ao receber um lote novo de dados de ações a cada semana, um analista financeiro precisa adicioná-lo a uma coleção existente de dados. Outras vezes, talvez seja necessário combinar mais seletivamente por meio de joins conjuntos de dados que compartilhem uma coluna comum, de modo que gerem um conjunto resumido de dados. Por exemplo, talvez uma empresa varejista queira fazer o merge dos dados gerais de pedidos online com os detalhes específicos de itens pedidos, conforme vimos no Capítulo 6. Em ambos os casos, após combinar os dados, podemos usá-los para uma análise mais aprofundada. Por exemplo, é possível executar uma variedade de operações de filtragem, de agrupamento ou de agregação no conjunto de dados combinado.

Como aprendemos nos capítulos anteriores, os conjuntos de dados em Python podem assumir a forma de estruturas built-in de dados, como listas, tuplas e dicionários, ou podem ser organizados usando estruturas de dados de terceiros, como arrays do NumPy ou DataFrames do pandas. Nesse último caso, temos um conjunto mais completo de ferramentas para combinar dados e, com isso, mais opções quando precisamos atender a determinas condições de join. No entanto, isso não significa que não seremos capazes de combinar efetivamente as estruturas built-in de dados do Python. Neste capítulo,

veremos como fazer isso, assim como combinar estruturas de dados de terceiros.

Combinando estruturas built-in de dados

A sintaxe para combinar as estruturas built-in de dados do Python é bastante simples. Nesta seção, veremos como combinar listas ou tuplas com o operador `+`. Em seguida, aprenderemos a combinar dicionários com o operador `**`. Vamos explorar também como executar joins, agregações e outras operações em listas de tuplas. Basicamente, vamos manipulá-las como se fossem tabelas de banco de dados, com cada tupla representando uma linha.

Combinando listas e tuplas com o operador +

O modo mais fácil de combinar duas ou mais listas ou duas ou mais tuplas é com o operador `+`. Basta escrever uma declaração com listas ou tuplas juntas, assim como faríamos para adicionar múltiplos números. Esse método funciona bem quando precisamos inserir os elementos das estruturas que estão sendo combinadas em uma única estrutura nova sem alterar os elementos propriamente ditos. Muitas vezes, esse processo é conhecido como *concatenação.*

Para demonstrá-lo, retomaremos o exemplo da varejista online de moda do capítulo anterior. Suponha que as informações sobre os pedidos feitos ao longo de determinado dia sejam coletadas em uma lista. Assim, temos uma lista para cada dia. Temos as três listas a seguir:

```
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)
]
```

Para fins de análise posterior, talvez seja necessário combiná-las em uma única lista. O operador `+` facilita isso; basta adicionar as três listas juntas:

```
orders = orders_2022_02_04 + orders_2022_02_05 + orders_2022_02_06
```

Vejamos a lista `orders` resultante:

```
[
(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)
]
```

Agora, como podemos verificar, os elementos das três listas originais aparecem em uma única lista, e sua ordem é determinada pela ordem escrita da declaração de concatenação. Nesse exemplo específico, os elementos das listas combinadas eram todos tuplas. No entanto, o operador `+` concatena listas cujos elementos são de qualquer tipo. Desse modo, podemos combinar facilmente listas de inteiros, strings, dicionários ou qualquer outra coisa.

Além disso, podemos usar a mesma sintaxe do operador `+` para combinar múltiplas tuplas. Contudo, se tentarmos usar `+` para combinar dicionários, receberemos o erro `unsupported operand type(s)`. Na seção a seguir, veremos a sintaxe adequada para combinar dicionários.

Combinando dicionários com o operador **

O operador `**` divide ou desempacota um dicionário em seus pares individuais de chave-valor. Para combinar dois dicionários em um, precisamos desempacotar ambos com o operador `**` e armazenar os resultados em um dicionário novo. Isso funciona mesmo que um ou ambos os dicionários tenham uma estrutura hierárquica. No contexto do nosso exemplo de varejista, considere o seguinte dicionário com alguns campos adicionais pertencentes a um pedido:

```
extra_fields_9423517 = {
```

```
  'ShippingInstructions' : { 'name'   : 'John Silver',
                             'Phone' :  [{ 'type' : 'Office', 'number' : '809-
123-9309' },
                                         { 'type' : 'Mobile', 'number' : '417-
123-4567' }
                                        ]}
}
```

Graças ao uso de nomes de chaves significativos, a estrutura aninhada do dicionário fica clara. Na prática, quando estamos lidando com estruturas hierárquicas de dados, a possibilidade de acessar dados por chaves em vez de acessá-los por posições torna os dicionários preferíveis às listas.

Agora, imagine que temos outros campos nessa mesma ordem em outro dicionário:

```
order_9423517 = {'OrderNo':9423517, 'Date':'2022-02-04', 'Empno':9001}
```

Nossa tarefa é concatenar esses dicionários em um único dicionário que inclua todos os pares chave-valor de ambos os originais. Aplicamos o operador `**` da seguinte forma:

```
order_9423517 = {**order_9423517, **extra_fields_9423517}
```

Inserimos os dicionários que queremos concatenar entre chaves, precedendo cada nome de dicionário com `**`. O operador `**` desempacota os dois dicionários em seus pares chave-valor e, em seguida, as chaves os empacotam de volta em um único dicionário. Vejamos como `order_9423517` ficou:

```
{
'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'}
                                  ]}
}
```

Conforme podemos observar, todos os elementos dos dicionários originais estão presentes e sua estrutura hierárquica foi preservada.

Combinando linhas correspondentes de duas estruturas

Já sabemos como combinar diversas listas em uma única lista sem alterar os

elementos delas. Na prática, teremos de criar muitas vezes o join de duas ou mais estruturas de dados que compartilham uma coluna comum em uma única estrutura, combinando linhas correspondentes dessas estruturas de dados em uma única linha. Caso suas estruturas de dados sejam DataFrames do pandas, é possível recorrer a métodos como `join()` e `merge()`, conforme vimos nos capítulos anteriores. No entanto, caso suas estruturas de dados sejam listas contendo "linhas" de tuplas, esses métodos estão indisponíveis. Em vez de usá-los, é necessário iterar com um loop nas listas e fazer o join de cada linha individualmente.

Para exemplificar, faremos o join da lista `orders`, criada na seção "Combinando listas e tuplas com o operador +" no início deste capítulo, com a lista `details`, apresentada na seção "Dados para agregação" do Capítulo 6. Vamos relembrar como era essa lista:

```
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)
]
```

Ambas as listas contêm tuplas cujo primeiro elemento é um número de pedido. Nosso objetivo é encontrar as tuplas com números de pedidos correspondentes, fazer o merge delas em uma única tupla e armazenar todas as tuplas em uma lista. Vejamos como isso é feito:

```
1 orders_details = []
2 for o in orders:
    for d in details:
    3 if d[0] == o[0]:
        orders_details.append(o + 4 d[1:])
```

Primeiro, criamos uma lista vazia para receber as tuplas mescladas 1. Em seguida, usamos um par de loops aninhados para iterar sobre as listas 2 e uma instrução `if` 3 para fazer o merge apenas das tuplas com números de pedidos correspondentes. Na tupla que está sendo combinada e anexada à lista `orders_details`, para evitar repetir o número de pedido duas vezes, fatiamos cada tupla de `details` 4, obtendo todos os seus campos, exceto o primeiro, que contém o número redundante de pedido.

Ao ver esse código, talvez você esteja se perguntando se poderíamos implementá-lo de forma mais elegante em uma única linha. Na verdade, com as list comprehensions, podemos chegar ao mesmo resultado da seguinte forma:

```
orders_details =  [[o for o in orders if d[0] == o[0]][0] + d[1:] for d in
details]
```

Na list comprehension externa, iteramos com um loop nas tuplas da lista `details`. Na list comprehension interna, encontramos uma tupla na lista `orders` cujo número de pedido corresponde à tupla atual de `details`. Visto que uma linha de pedido em `details` deve ter somente uma tupla correspondente em `orders`, a list comprehension interna deve gerar uma lista com um único elemento (uma tupla representando um pedido). Sendo assim, obtemos o primeiro elemento da list comprehension interna com o operador `[0]`, em seguida concatenamos a tupla desse pedido com a tupla correspondente de `details` usando o operador `+`, excluindo o número redundante de pedido com `[1:]`.

Caso tenha criado `orders_details` por meio de list comprehensions ou com a ajuda de dois loops `for`, como mostrado anteriormente, terá a seguinte lista resultante:

```
[
(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)
]
```

A lista contém todas as tuplas da lista `details`, e cada tupla também contém informações adicionais da tupla correspondente da lista `orders`.

Implementando diferentes tipos de joins para listas

A operação executada na seção anterior é um join padrão de um-para-muitos: na lista `details`, cada linha de pedido corresponde a um pedido na lista `orders`; e na lista `orders`, cada pedido corresponde a uma ou mais linhas na lista `details`. Mas, na prática, quando se trata de joins, pode acontecer de um ou ambos os conjuntos de dados não terem linhas correspondentes no outro conjunto. Para responder a essas situações, devemos executar operações equivalentes a múltiplos joins, no estilo banco de dados, que vimos no Capítulo 3: left joins, right joins, inner joins e outer joins.

Por exemplo, a lista `details` pode ter linhas de pedido para pedidos não encontrados na lista `orders`. Isso pode ocorrer se filtramos um determinado intervalo de dados na lista `orders`; como a lista `details` não tem um campo de data, não podemos filtrá-la adequadamente. Para simular esse comportamento, adicionaremos uma linha nova à lista `details` para referenciar um pedido que não está na lista `orders`:

```
details.append((4626592, 'Shorts', 'Protest', 48.0, 1))
```

Agora, quando tentamos gerar a lista `orders_details`,

```
orders_details = [[o for o in orders if d[0] == o][0] + d[1:] for d in details]
```

recebemos o seguinte erro:

```
IndexError: list index out of range
```

O problema ocorre quando obtemos o número do pedido não correspondente na lista `details` e tentamos acessar o primeiro elemento da list comprehension interna correspondente. Como esse elemento não existe, o número do pedido não está na lista `orders`. Podemos resolver isso adicionando uma cláusula `if` ao loop `for d in details` dentro da list comprehension externa, verificando se o

número do pedido em uma linha de `details` pode ser encontrado em qualquer linha de `orders`, conforme mostrado a seguir:

```
orders_details = [[o for o in orders if d[0] in o][0] + d[1:] for d in details
                1 if d[0] in [o[0] for o in orders]]
```

Resolvemos o problema excluindo quaisquer linhas da lista `details` que não tenham uma linha correspondente na lista `orders`, implementando a verificação na cláusula `if` que acompanha o loop `for d in details 1`. Desse modo, a list comprehension mostrada aqui resulta em um inner join.

Mas e se quisermos incluir todas as linhas da lista `details` na lista resultante `orders_details`? Por exemplo, podemos querer resumir os totais de todos os pedidos, não apenas dos pedidos da lista atual `orders` (que hipoteticamente foi filtrada por data). Podemos resumir os totais dos pedidos que *estão* na lista atual `orders` e comparar essas somas.

Nesse caso, precisamos implementar um right join, partindo do princípio de que a lista `orders` esteja no lado esquerdo do relacionamento, e a lista `details` esteja à direita. Lembre-se de que um right join retorna todas as linhas do conjunto de dados à direita e apenas as linhas correspondentes do conjunto de dados à esquerda. Vamos atualizar a list comprehension anterior:

```
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] 1 else (d[0], None, None) + d[1:] for d in
details]
```

Aqui, adicionamos uma cláusula `else 1` à cláusula `if` atribuída ao loop `for d in details`. Como funciona para qualquer linha de `details` que não tenha linha correspondente em `orders`, a cláusula `else` cria uma tupla nova com o número do pedido e com mais duas entradas `None` para substituir os campos ausentes de `orders` e concatenar essa tupla com a linha de `details`, gerando, assim, uma linha com a mesma estrutura de todas as outras. Desse modo, o conjunto de dados gerado incluirá a linha de `details` que não tem linha correspondente em `orders`, além de todas as linhas correspondentes:

```
[
--trecho de código omitido--
(4626490, '2022-02-06', 9004, 'Sweater', 'Dickies', 56.0, 1),
(4626592, None, None, 'Shorts', 'Protest', 48.0, 1)
]
```

Agora que temos a lista `orders_details_right` (o right join das listas `orders` e `details`), é possível somar todos os pedidos e comparar o resultado apenas com o total dos pedidos incluídos na lista `orders`. Somamos o total de todos os pedidos com a função built-in `sum()` do Python:

```
sum(pr*qt for _, _, _, _, _, pr, qt in orders_details_right)
```

Já que loop `for` passado como parâmetro para `sum()` é um pouco semelhante ao loop usado em uma list comprehension, isso possibilita obter somente os elementos necessários em cada iteração do loop. Neste exemplo particular, tudo que precisamos encontrar em cada iteração é `pr*qt`, a multiplicação dos valores de Price e Quantity da tupla em questão. Como não estamos interessados nos outros valores de cada tupla, usamos os placeholders _ na cláusula após a palavra reservada `for`.

Caso tenha seguido as etapas apresentadas neste capítulo até agora, a chamada anterior retornará:

```
779.0
```

Podemos calcular apenas os totais dos pedidos que estão na lista `orders` com uma versão modificada da chamada `sum()`:

```
sum(pr*qt for _, dt, _, _, _, pr, qt in orders_details_right 1 if dt != None)
```

Aqui, adicionamos uma cláusula `if` ao loop para filtrar pedidos que não estavam na lista `orders 1`. Ignoramos as linhas em que o campo Date (`dt`) contém `None`, indicando que as informações do pedido da linha não foram obtidas a partir de `orders`. A soma gerada será:

```
731.0
```

Concatenando arrays do NumPy

Ao contrário das listas, não podemos usar o operador `+` para concatenar arrays do NumPy. Isso ocorre porque, conforme analisado no Capítulo 3, o NumPy reserva o operador `+` para executar operações de adição por elemento em múltiplos arrays. Para concatenar dois arrays do NumPy, usamos a função `numpy.concatenate()`.

Para demonstrar, usaremos o array `base_salary` da seção “Criando um array do Numpy” do Capítulo 3, criado da seguinte forma (aqui, chamaremos de

`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])
```

Lembre-se de que cada linha no array contém três meses de dados de salário-base para determinado funcionário. Agora suponha que temos informações salariais para mais dois funcionários em outro array, `base_salary2`:

```
maya_salary = [2200,2400,2400]
john_salary = [2500,2700,2700]
base_salary2 = np.array([maya_salary, john_salary])
```

Queremos armazenar as informações salariais de todos os cinco funcionários no mesmo array. Para isso, concatenamos `base_salary1` e `base_salary2` com `numpy.concatenate()`, da seguinte forma:

```
base_salary = np.concatenate((base_salary1, base_salary2), axis=0)
```

O primeiro parâmetro é uma tupla contendo os arrays a serem concatenados. O segundo parâmetro, `axis`, é vital: especifica se os arrays devem ser concatenadas horizontal ou verticalmente, ou, em outras palavras, se o segundo array será adicionado como linhas ou colunas novas. O primeiro eixo (`axis=0`) é executado verticalmente. Assim, `axis=0` instrui a função `concatenate()` a anexar as linhas de `base_salary2` abaixo das linhas de `base_salary1`. Vejamos o array resultante:

```
[[2700 3000 3000]
[2600 2800 2800]
[2300 2500 2500],
[2200 2400 2400],
[2500 2700 2700]]
```

Agora imagine que as informações salariais para o próximo mês chegaram. Podemos inserir esses números novos em outro array do NumPy, deste modo:

```
new_month_salary = np.array([[3000],[2900],[2500],[2500],[2700]])
```

Se exibirmos o array, veremos a seguinte saída:

```
[[3000]
[2900]
[2500]
[2500]
```

```
[2700]]
```

Precisamos adicionar o array `new_month_salary` ao array `base_salary` como uma coluna extra. Presumindo que a ordem dos funcionários é a mesma em ambos os arrays, podemos usar `concatenate()` assim:

```
base_salary = np.concatenate((base_salary, new_month_salary), axis=1)
```

Como é executado horizontalmente nas colunas, `axis=1` instrui a função `concatenate()` a anexar o array `new_month_salary` como uma coluna à direita das colunas do array `base_salary`. Vejamos como fica `base_salary` agora:

```
[[2700 3000 3000 3000]
[2600 2800 2800 2900]
[2300 2500 2500 2500]
[2200 2400 2400 2500]
[2500 2700 2700 2700]]
```

EXERCÍCIO #11: ADICIONANDO NOVAS LINHAS/COLUNAS A UM ARRAY DO NUMPY

Retomando o exemplo anterior, crie um array novo do NumPy de duas colunas com informações salariais de mais dois meses para cada funcionário. Em seguida, concatene o array `base_salary` existente com o array recém-criado. Do mesmo modo, anexe uma linha nova ao array `base_salary`, adicionando as informações salariais para outro funcionário. Repare que, ao adicionar uma única linha ou coluna a um array do NumPy, você pode utilizar a função `numpy.append()` em vez de `numpy.concatenate()`.

Combinando estruturas de dados do pandas

No Capítulo 3, vimos algumas técnicas básicas para combinar estruturas de dados do pandas. Analisamos exemplos de como os objetos Series podem ser combinados em um DataFrame e como fazer o join de dois DataFrames pelos seus índices. Aprendemos também sobre os diferentes tipos de joins que podemos criar ao fazermos o merge de dois DataFrames em um único, passando o parâmetro `how` para os métodos `join()` ou `merge()` do pandas. Nesta seção, veremos mais exemplos de como usar esse parâmetro para criar joins não padrões de DataFrame, como um right join. Antes disso, no entanto, aprenderemos a concatenar dois DataFrames ao longo de um eixo específico.

Concatenando DataFrames

Assim como os arrays do NumPy, talvez seja necessário concatenar dois

DataFrames ao longo de um eixo específico, anexando as linhas ou as colunas de um DataFrame ao outro. Nesta seção, veremos exemplos de como fazer isso com a função `concat()` do pandas. Antes de passarmos aos exemplos, precisaremos criar dois DataFrames para serem concatenados.

Usando as listas `jeff_salary`, `nick_salary` e `tom_salary` do início deste capítulo, é possível criar um DataFrame com um dicionário, assim:

```
import pandas as pd
salary_df1 = pd.DataFrame(
    {'jeff': jeff_salary,
     'nick': nick_salary,
     'tom': tom_salary
    })
```

Cada lista se torna um valor no dicionário que, por sua vez, torna-se uma coluna no DataFrame novo. As chaves do dicionário, os nomes correspondentes dos funcionários, tornam-se os rótulos das colunas. Cada linha do DataFrame contém todos os dados salariais de um único mês. Por padrão, as linhas são indexadas numericamente, porém seria mais relevante indexá-las por mês. Podemos atualizar os índices da seguinte forma:

```
salary_df1.index = ['June', 'July', 'August']
```

Vejamos como fica o DataFrame `salary_df1` agora:

```
        jeff  nick   tom
June    2700  2600  2300
July    3000  2800  2500
August  3000  2800  2500
```

Talvez seja mais conveniente visualizar os dados salariais de um funcionário como uma linha em vez de uma coluna. Podemos fazer isso com a propriedade `T` do DataFrame, abreviação para o método `DataFrame.transpose()`:

```
salary_df1 = salary_df1.T
```

Essa instrução faz a *transposição* do DataFrame, transformando suas colunas em linhas e vice-versa. Agora, o DataFrame é indexado pelo nome do funcionário:

```
      June  July  August
jeff  2700  3000    3000
nick  2600  2800    2800
tom   2300  2500    2500
```

Agora, vamos criar outro DataFrame com as mesmas colunas para ser

concatenado com `salary_df1`. Aqui, seguindo o exemplo da concatenação de arrays do NumPy, criamos um DataFrame que contém dados salariais de mais dois funcionários:

```
salary_df2 = pd.DataFrame(
    {'maya': maya_salary,
     'john': john_salary
    },
    index = ['June', 'July', 'August']
).T
```

Basta criar o DataFrame, definir o índice e transpor as linhas e colunas em uma única instrução. Vejamos o DataFrame recém-criado:

```
       June  July  August
maya   2200  2400    2400
john   2500  2700    2700
```

Agora que criamos ambos os DataFrames, estamos prontos para concentrá-los.

Concatenação ao longo do eixo 0

A função `concat()` do pandas concatena objetos do pandas ao longo de um determinado eixo. Por padrão, essa função usa o eixo 0. Ou seja, as linhas do DataFrame que aparecem em segundo lugar na lista de argumentos serão anexadas abaixo das linhas do DataFrame que aparecem primeiro. Portanto, para concatenar os DataFrames `salary_df1` e `salary_df2`, podemos chamar `concat()` sem passar explicitamente o argumento `axis`. Tudo que precisamos fazer é especificar os nomes dos DataFrames entre colchetes:

```
salary_df = pd.concat([salary_df1, salary_df2])
```

Isso gerará o seguinte DataFrame:

```
       June  July  August
jeff   2700  3000    3000
nick   2600  2800    2800
tom    2300  2500    2500
maya   2200  2400    2400
john   2500  2700    2700
```

Como podemos ver, as linhas `maya` e `john` do segundo DataFrame foram adicionadas abaixo das linhas do primeiro DataFrame.

Concatenação ao longo do eixo 1

Ao concatenar ao longo do eixo 1, a função `concat()` anexará as colunas do segundo DataFrame à direita das colunas do primeiro. Para ilustrar isso, podemos usar `salary_df` da seção anterior como primeiro DataFrame. Para o segundo DataFrame, criamos a seguinte estrutura, que contém mais dois meses de dados salariais:

```
salary_df3 = pd.DataFrame(
    {'September': [3000,2800,2500,2400,2700],
     'October': [3200,3000,2700,2500,2900]
    },
    index = ['jeff', 'nick', 'tom', 'maya', 'john']
)
```

Agora chamamos `concat()`, passando os dois DataFrames e especificando `axis=1` a fim de garantir que a concatenação seja horizontal:

```
salary_df = pd.concat([salary_df, salary_df3], axis=1)
```

Vejamos o DataFrame resultante:

```
      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
```

Os dados salariais do segundo DataFrame aparecem como colunas novas à direita dos dados salariais do primeiro DataFrame.

Removendo colunas/linhas de um DataFrame

Após combinar os DataFrames, talvez seja necessário remover algumas linhas ou colunas desnecessárias. Digamos, por exemplo, que você queira remover as colunas `September` e `October` do DataFrame `salary_df`. Isso é possível com o método `DataFrame.drop()`:

```
salary_df = salary_df.drop(['September', 'October'], axis=1)
```

O primeiro argumento aceita os nomes das colunas ou das linhas a serem excluídas do DataFrame. Em seguida, usamos o argumento `axis` para especificar se são linhas ou colunas. Neste exemplo, estamos excluindo colunas, pois `axis` está definido como `1`.

Com o `drop()`, não ficamos limitados a excluir apenas as últimas colunas/linhas de um DataFrame. Podemos passar uma lista arbitrária de colunas ou de linhas a serem excluídas, como mostrado a seguir:

```
salary_df = salary_df.drop(['nick', 'maya'], axis=0)
```

Vejamos como fica `salary_df`, após executarmos as duas operações anteriores:

```
      June  July  August
jeff  2700  3000    3000
tom   2300  2500    2500
john  2500  2700    2700
```

Removemos as colunas September e October, e as linhas Nick e Maya.

Concatenando DataFrames com um índice hierárquico

Até o momento, vimos exemplos de concatenação de DataFrames com índices simples. Mas, agora, veremos como concatenar DataFrames com um MultiIndex. O exemplo a seguir usa o DataFrame `df_date_region`, apresentado na seção “Agrupamento e agregação dos dados” do Capítulo 6. Vejamos esse DataFrame criado como resultado de diversas operações sucessivas:

```
                   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
```

Para recriarmos esse DataFrame, não é necessário seguirmos as etapas do Capítulo 6. Em vez disso, vamos executar a seguinte instrução:

```
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'])
```

Agora, precisaremos de outro DataFrame que também seja indexado por `Date`

e `Region`. Vejamos como criá-lo:

```
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'])
```

O segundo DataFrame apresenta as mesmas três datas que o primeiro, mas tem dados de uma região nova, `South`. Ao concatenar esses dois DataFrames, o desafio é manter o resultado ordenado por data, em vez de simplesmente anexar o segundo DataFrame abaixo do primeiro. Vejamos como podemos fazer isso:

```
df_date_region = pd.concat([df_date_region1,
            df_date_region2]).sort_index(level=['Date','Region'])
```

Começamos com uma chamada `concat()`, que seria mais ou menos a mesma se estivéssemos concatenando DataFrames com índices de uma única coluna. Identificamos os DataFrames a serem combinados e, como omitimos o parâmetro `axis`, os DataFrames serão concatenados verticalmente por padrão. Para ordenar as linhas no DataFrame resultante por data e região, devemos chamar o método `sort_index()`. Como resultado, obteremos o seguinte DataFrame:

```
                   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
```

Como podemos verificar, as linhas do segundo DataFrame foram integradas entre as linhas do primeiro DataFrame, mantendo o agrupamento de nível superior por data.

Join de dois DataFrames

Ao fazer o join de dois DataFrames, combinamos cada linha de um conjunto de dados com a(s) linha(s) correspondente(s) do outro, em vez de simplesmente anexarmos as linhas ou colunas de um DataFrame abaixo ou ao lado do outro. Para relembrar os conceitos básicos sobre joins de DataFrames, confira a seção “Combinando DataFrames” do Capítulo 3, que abrange os diferentes tipos de joins que podemos implementar. Nesta seção, iremos além do que aprendemos no Capítulo 3: implementaremos um right join e um join baseado em um relacionamento muitos-para-muitos.

Implementando um right join

Um right join obtém todas as linhas de um segundo DataFrame e as combina com quaisquer linhas correspondentes de um primeiro DataFrame. Conforme veremos, esse tipo de join possibilita que algumas linhas no DataFrame resultante tenham campos indefinidos, o que pode ocasionar desafios inesperados.

Para ver como isso funciona, faremos o right join dos DataFrames `df_orders` e `df_details`, criados a partir das listas `orders` e `details` apresentadas no Capítulo 6, aquelas que usamos nos exemplos da seção de abertura deste capítulo. Vejamos como criar os DataFrames a partir dessas listas:

```
import pandas as pd
df_orders = pd.DataFrame(orders, columns =['OrderNo', 'Date', 'Empno'])
df_details = pd.DataFrame(details, columns =['OrderNo', 'Item', 'Brand',
                                             'Price', 'Quantity'])
```

Não se esqueça de que cada linha na lista original `details` tem uma linha correspondente na lista `orders`. Assim sendo, o mesmo vale para os DataFrames `df_details` e `df_orders`. Para exemplificar adequadamente um right join, é necessário adicionar uma ou mais linhas novas a `df_details` que não tenham correspondências em `df_orders`. Podemos adicionar uma linha com o método `DataFrame.append()`, que aceita a linha que está sendo anexada como um dicionário ou uma Series.

Caso tenha acompanhado o exemplo da seção anterior “Implementando diferentes tipos de joins para listas” deste capítulo, sabe que já adicionamos a seguinte linha à lista `details` e, portanto, ela já deve aparecer no DataFrame `df_details`. Nesse caso, pode ignorar a operação de anexação a seguir. Caso

contrário, anexe esta linha ao `df_details` como um dicionário. Perceba que o valor do campo `OrderNo` da linha nova não pode ser encontrado entre os valores da coluna `OrderNo` no DataFrame `df_orders`:

```
df_details = df_details.append(
  {'OrderNo': 4626592,
   'Item': 'Shorts',
   'Brand': 'Protest',
   'Price': 48.0,
   'Quantity': 1
  },
1 ignore_index = True
)
```

Devemos definir o parâmetro `ignore_index` como `True` `1` ou não conseguiremos anexar um dicionário a um DataFrame. Definir esse parâmetro como `True` também redefine o índice do DataFrame, mantendo valores de índice contínuos (`0`, `1`, ...) para as linhas.

Em seguida, faremos o join dos DataFrames `df_orders` e `df_details` com o método `merge()`. Conforme analisado no Capítulo 3, o método `merge()` proporciona uma forma conveniente de fazer o join de dois DataFrames com uma coluna comum:

```
df_orders_details_right = df_orders.merge(df_details, 1 how='right',
                  2 left_on='OrderNo', right_on='OrderNo')
```

Recorremos ao parâmetro `how` para especificar o tipo de join, neste exemplo, um right join `1`. Com os parâmetros `left_on` e `right_on`, especificamos as colunas para o join nos DataFrames `df_orders` e `df_details`, respectivamente `2`.

Vejamos o DataFrame resultante:

```
    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
```

Como a linha recém-anexada a `df_details` não tem linha correspondente em `df_orders`, a linha correspondente no DataFrame resultante contém `NaN` (o marcador de valor ausente padrão) nos campos `Date` e `Empno`. No entanto, isso ocasiona um problema: os `NaN`s não podem ser armazenados em colunas de inteiros. Por isso, o pandas converte automaticamente uma coluna de inteiro em uma coluna de float quando um `NaN` é inserido. Por esse motivo, os valores na coluna `Empno` foram convertidos em floats. É possível validarmos isso com a propriedade `dtypes` do DataFrame `df_orders_details_right`, que mostra o tipo de cada coluna:

```
print(df_orders_details_right.dtypes)
```

Veremos a seguinte saída:

```
OrderNo       int64
Date         object
Empno       float64
Item         object
Brand        object
Price       float64
Quantity      int64
dtype: object
```

Como podemos ver, a coluna `Empno` é do tipo `float64`. Se verificarmos de forma similar a propriedade `dtypes` de `df_orders`, veremos que a coluna `Empno` era originalmente do tipo `int64`.

Obviamente, essa conversão de inteiros para floats não é um comportamento desejável: os números dos funcionários não devem ter pontos decimais! Existe uma maneira de reconverter a coluna `Empno` para números inteiros? Uma solução alternativa é substituir `NaN`s dessas colunas por algum valor inteiro, digamos `0`. Desde que `0` não seja o ID de funcionário de alguém, essa substituição é perfeitamente aceitável. Vejamos como implementar essa mudança:

```
df_orders_details_right =
df_orders_details_right.fillna({'Empno':0}).astype({'Empno':'int64'})
```

Usamos o método `fillna()` do DataFrame, que substitui `NaN`s nas colunas

especificadas por um determinado valor. A coluna e o valor substituídos são definidos como um dicionário. Neste exemplo específico, substituímos NaNs na coluna `Empno` por `0`s. Em seguida, convertemos o tipo da coluna para `int64` com o método `astype()`. Mais uma vez, a coluna e o tipo novos são especificados como um par chave-valor em um dicionário.

Teremos o seguinte DataFrame:

```
     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
--trecho de código omitido--
14   4626592         NaN      0        Shorts         Protest   48.0         1
```

Na coluna `Empno`, o `NaN` se tornou um `0` e os números de funcionários aparecem novamente como números inteiros. Nada de pontos decimais desagradáveis!

Implementando um join muito-para-muitos

Podemos criar um join muitos-para-muitos com conjuntos de dados em que uma linha em cada conjunto pode estar relacionada a múltiplas linhas no outro. Por exemplo, suponha que tenhamos dois conjuntos de dados contendo livros e autores, respectivamente. No conjunto de dados de autores, cada registro pode ser vinculado a um ou mais registros no conjunto de dados de livros e, no conjunto de dados de livros, cada registro pode ser vinculado a um ou mais registros no conjunto de dados de autores.

Normalmente, fazemos um join de conjuntos de dados que compartilham um relacionamento muitos-para-muitos por meio de uma *tabela associativa*, também conhecida como uma *tabela de correspondência*. Essa tabela mapeia dois (ou mais) conjuntos de dados juntos, referenciando as chaves primárias de cada um. A tabela associativa tem um relacionamento um-para-muitos com cada um dos conjuntos de dados e funciona como um intermediário entre eles, possibilitando o merge.

Vejamos como implementar um join muitos-para-muitos por meio de uma tabela de correspondência, criando um DataFrame `books` e um DataFrame `authors`:

```
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']})
```

O DataFrame `books` inclui três livros, cada um com um `book_id` único, enquanto o DataFrame `authors` inclui três autores, cada um com um `author_id` exclusivo. Agora, criaremos um terceiro DataFrame, `matching`, que servirá como tabela associativa, correlacionando cada livro com seu(s) respectivo(s) autor(es) e vice-versa:

```
matching = pd.DataFrame({'author_id': ['jsn', 'jsn','tri', 'wsn'],
                         'book_id': ['b1', 'b2', 'b2', 'b3']})
```

O DataFrame `matching` tem duas colunas: uma correspondente aos IDs do autor e outra correspondente aos IDs do livro. Ao contrário dos outros dois DataFrames, cada linha em `matching` não representa somente um único autor ou um único livro. Em vez disso, contém informações sobre o *relacionamento entre* um autor específico e um livro particular. Vejamos esse DataFrame:

```
  author_id book_id
0       jsn      b1
1       jsn      b2
2       tri      b2
3       wsn      b3
```

A primeira e a segunda linhas referenciam `author_id` de `jsn`, determinando que Johnson é o autor de dois livros diferentes. Do mesmo modo, a segunda e a terceira referenciam `book_id` de `b2`, determinando que o livro *Python for Web Development* tem dois autores.

Agora podemos criar um join muitos-para-muitos dos conjuntos de dados `authors` e `books` por meio de `matching`, conforme mostrado a seguir:

```
authorship = books.merge(matching).merge(authors)[['title','topic','author']]
```

Na verdade, a operação consiste em dois joins separados com o método `merge()`. Primeiro, criamos o join dos DataFrames correspondentes `books` e `matching` em suas respectivas colunas `book_id`. Em seguida, criamos o join do resultado do primeiro join com o Dataframe `authors` por meio de suas colunas `author_id`. Ambos os joins são meros joins de um-para-muitos; porém, juntos, resultam em um DataFrame novo, exemplificando o relacionamento de

muitos-para-muitos entre os conjuntos de dados `books` e `authors`. Vejamos o DataFrame, já que o filtramos para incluir somente as colunas `title`, `topic` e `author`:

```
                         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
```

Conforme podemos observar, Johnson é listado duas vezes como autor de *Beautiful Coding* e *Python for Web Development*, ao passo que *Python for Web Development* é listado duas vezes, uma para cada um de seus autores.

Recapitulando

Neste capítulo, examinamos diferentes maneiras de combinar conjuntos de dados apresentados na forma de estruturas built-in de dados, como listas, ou estruturas de dados de terceiros, como arrays do NumPy e DataFrames do pandas. Aprendemos a concatenar conjuntos de dados, anexando as colunas ou linhas de um conjunto de dados a outro. Além do mais, aprendemos a criar join de conjuntos de dados, combinando suas linhas correspondentes.

CAPÍTULO 8

Criando visualizações

Os dados podem ser visualizados com mais clareza em formato visual do que em formato de números brutos. Por exemplo, podemos criar um gráfico de linhas para representar graficamente as mudanças no preço de uma ação ao longo do tempo. Ou podemos acompanhar o interesse nos artigos de um site com um histograma que mostre as visualizações diárias de cada um. Visualizações como essas nos ajudam a identificar imediatamente as tendências nos dados.

Neste capítulo, veremos um panorama geral dos tipos mais comuns de visualizações de dados e como criá-las usando a Matplotlib, popular biblioteca de plotagem do Python. Aprenderemos também como integrar a Matplotlib com o pandas e como criar mapas com a Matplotlib e com a biblioteca Cartopy.

Visualizações comuns

Diversos tipos de gráficos estão disponíveis para visualização de dados, incluindo gráficos de linhas, gráficos de barras, gráficos de pizza e histogramas. Nesta seção, analisaremos essas visualizações comuns e exploraremos os típicos casos de uso de cada uma delas.

Gráficos de linhas

Gráficos de linhas, também conhecidos como *gráficos de segmentos*, são úteis quando precisamos ilustrar tendências em dados ao longo de um período de tempo. Em um gráfico de linhas, inserimos a coluna de timestamp de um conjunto de dados ao longo do eixo x e uma ou mais colunas numéricas no eixo y.

Imagine um site em que os usuários podem visualizar diferentes artigos. É possível criar um gráfico para um artigo no qual o eixo x represente uma série de dias e o eixo y exiba quantas vezes o artigo foi visualizado por dia. Podemos ver esse gráfico na Figura 8.1.

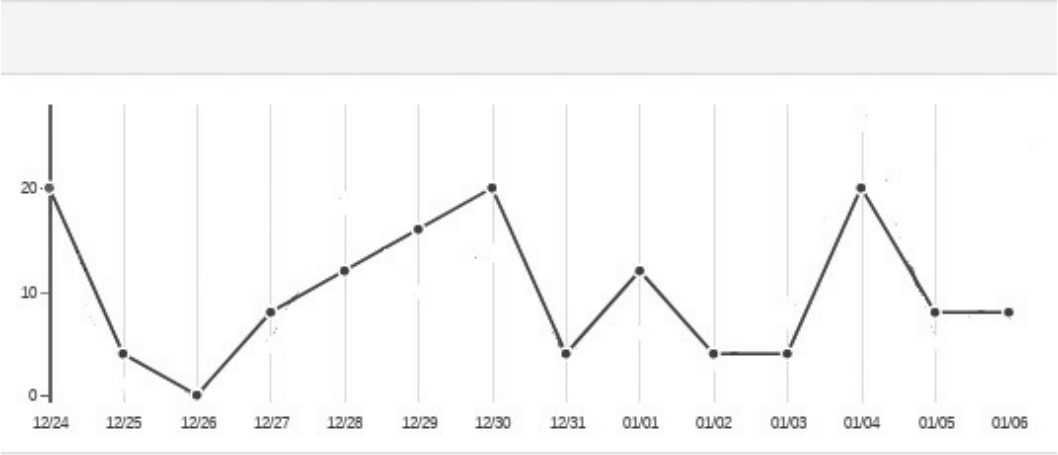


Figura 8.1: Gráfico de linhas que mostra as visualizações do artigo ao longo do tempo.

Em um gráfico de linhas, podemos sobrepor dados a partir de múltiplos parâmetros a fim de ilustrar a correlação entre eles, plotando os dados de cada parâmetro com uma linha de cor diferente. Por exemplo, a Figura 8.2 mostra o número diário de usuários únicos do site, além do número de visualizações de artigos.

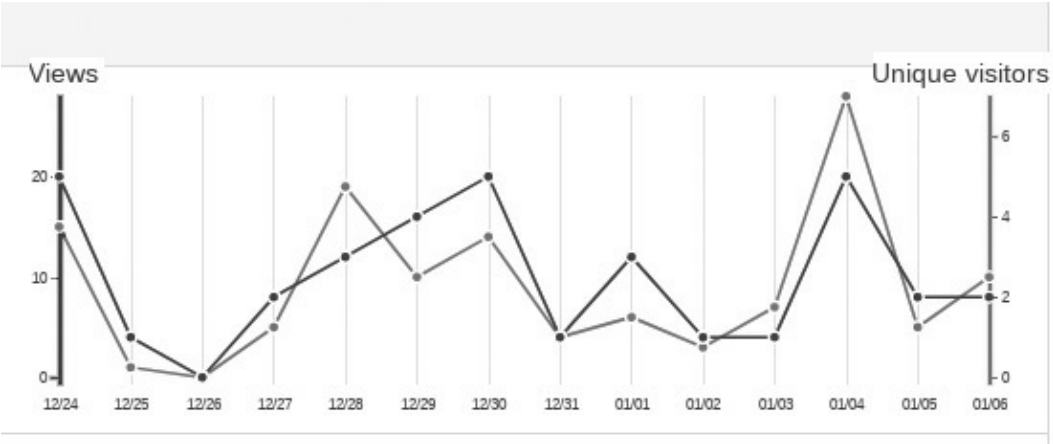


Figura 8.2: Gráfico de linhas que mostra o relacionamento de diversos parâmetros.

Nesse gráfico de linhas, o eixo y esquerdo mostra as visualizações do artigo,

enquanto o eixo y direito mostra os visitantes. Ao sobrepor os dados para ambos os parâmetros, fica visualmente claro que há uma correlação geral entre o número de visualizações de artigos e o número de visitantes.

NOTA *As visualizações de artigo podem ser plotadas como um histograma em vez de um gráfico de linhas. Em breve, analisaremos os histogramas nesta seção.*

Gráficos de barras

Gráficos de barras, também conhecidos como *gráficos de colunas*, mostram dados categóricos usando barras retangulares com alturas proporcionais aos valores que representam, possibilitando comparações entre categorias. Por exemplo, vejamos os seguintes valores, que representam as vendas anuais de uma empresa agregadas em base regional:

```
New England   $882,703
Mid-Atlantic  $532,648
Midwest       $714,406
```

A Figura 8.3 ilustra como esses dados de vendas ficam quando plotados em um gráfico de barras.

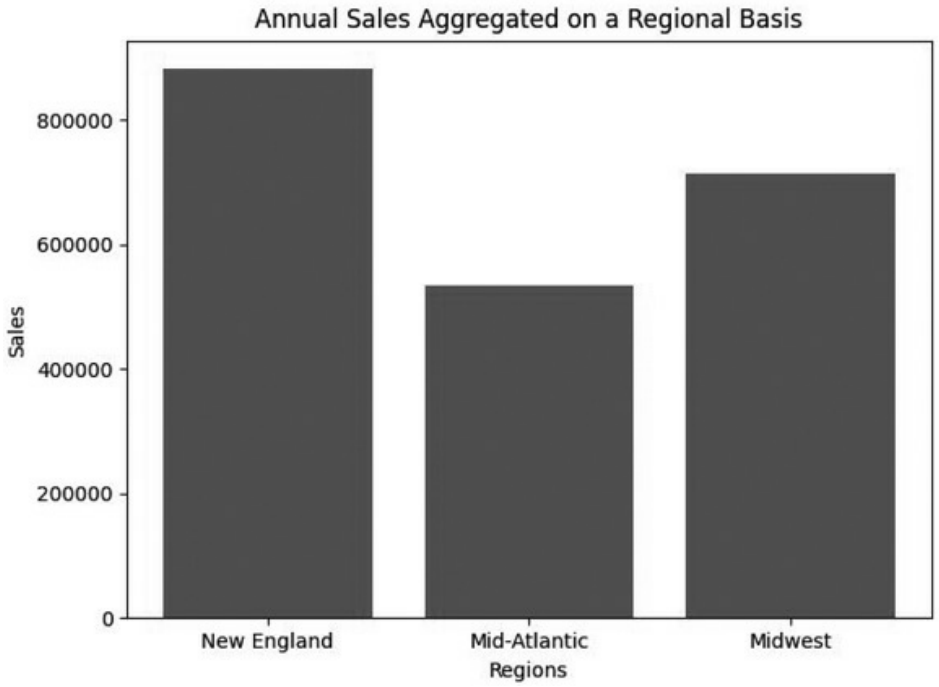


Figura 8.3: Gráfico de barras mostrando dados categóricos comparativos.

Neste gráfico, o eixo y exibe os valores de vendas (sales) comparativos para as regiões mostradas no eixo x.

Gráficos de pizza

Gráficos de pizza ilustram a proporção de cada categoria no conjunto de dados completo, expressa em porcentagem. A Figura 8.4 ilustra os valores de vendas do exemplo anterior quando plotados em um gráfico de pizza.

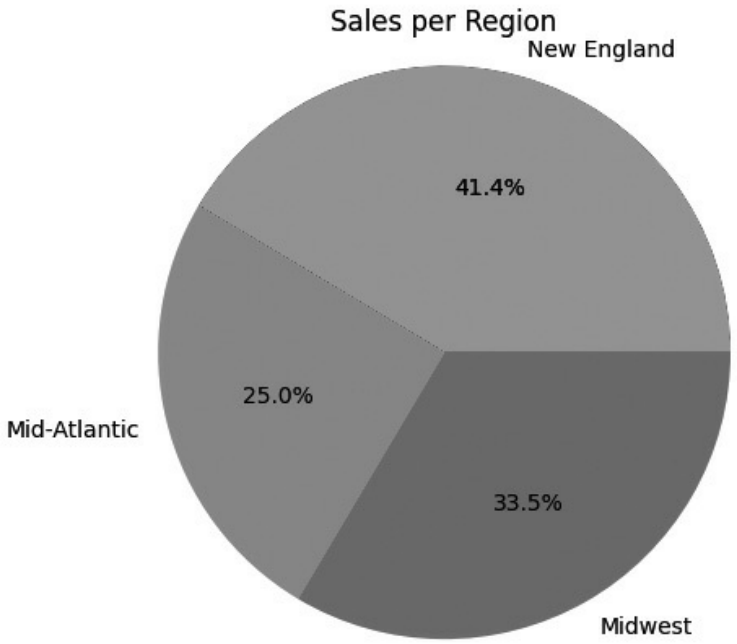


Figura 8.4: Os gráficos de pizza representam a porcentagem de cada categoria como uma fatia de um círculo.

Aqui, o tamanho de cada fatia fornece uma representação visual da proporção que cada categoria contribui para o todo. Podemos ver facilmente como as vendas de cada região (Sales per Region) se comparam entre si. Isso funciona bem quando cada fatia representa uma parte substancial da pizza, mas, como se pode imaginar, um gráfico de pizza não é a melhor escolha quando temos de representar partes muito pequenas. Por exemplo, uma fatia que representa 0,01% do todo pode nem ficar visível no gráfico.

Histogramas

Os *histogramas* mostram distribuições de frequência ou quantas vezes um determinado valor ou intervalo de valores aparece em um conjunto de dados. Cada valor, ou resultado, é representado por uma barra vertical cuja altura corresponde à frequência desse valor. Por exemplo, o histograma na Figura 8.5 evidencia de imediato a frequência de diferentes grupos salariais em um departamento de vendas.

Nesse histograma, os salários são agrupados em intervalos de US$50, e cada barra vertical representa o número de pessoas que têm salários dentro de um determinado intervalo. A visualização possibilita que visualizemos com rapidez quantos funcionários ganham entre US$1.200 e US$ 1.250, por exemplo, em comparação com outros intervalos, como US$1.250 a US$1.300.

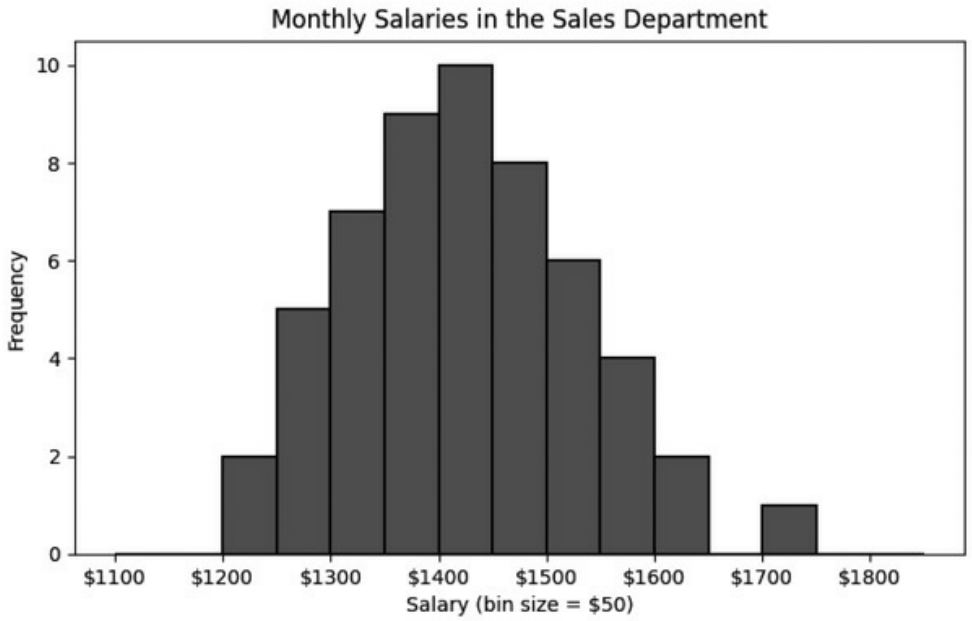


Figura 8.5: Histograma que mostra uma distribuição salarial.

Plotando gráficos com a Matplotlib

Agora que vimos os tipos mais comuns de gráficos, aprenderemos a criá-los com a Matplotlib, uma das bibliotecas do Python mais populares para visualização de dados. Aprenderemos a criar gráficos de linhas, gráficos de pizza, gráficos de barras e histogramas.

Cada visualização da Matplotlib, ou *figura*, é construída a partir de uma hierarquia de objetos aninhados. É possível trabalhar com esses objetos diretamente para criar visualizações bastante personalizáveis, ou podemos manipular os objetos indiretamente por meio de funções fornecidas no módulo `matplotlib.pyplot`. A última abordagem é mais simples e geralmente é suficiente para criar tabelas e gráficos básicos.

Instalando a Matplotlib

Verifique se a Matplotlib já está instalada tentando importá-la em uma sessão do interpretador Python:

```
> import matplotlib
```

Se receber um `ModuleNotFoundError`, instale a Matplotlib com o `pip` da seguinte forma:

```
$ python -m pip install -U matplotlib
```

Usando a matplotlib.pyplot

O módulo `matplotlib.pyplot`, normalmente sinalizado no código como `plt`, disponibiliza uma coleção de funções para a criação de figuras elegantes. O módulo viabiliza definir facilmente diversos aspectos de uma figura, como título, rótulos de eixo e assim por diante. Por exemplo, vejamos a seguir como criar um gráfico de linha plotando o preço da cotação final das ações da Tesla durante cinco dias consecutivos:

```
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('Date')
plt.ylabel('USD')
plt.show()
```

Primeiro, definimos o conjunto de dados como duas listas: `days`, contendo as datas que serão plotadas ao longo do eixo x, e `prices`, contendo os preços que serão plotados ao longo do eixo y. Em seguida, criamos um *plot*, a parte da figura que realmente mostra os dados, com a função `plt.plot()`, passando os

dados para os eixos x e y. Nas próximas três linhas de código, personalizamos a figura: adicionamos um título com a função `plt.title()` e rótulos para os eixos x e y com a função `plt.xlabel()` e com a função `plt.ylabel()`. Por último, exibimos a figura com a função `plt.show()`. A Figura 8.6 mostra o resultado.

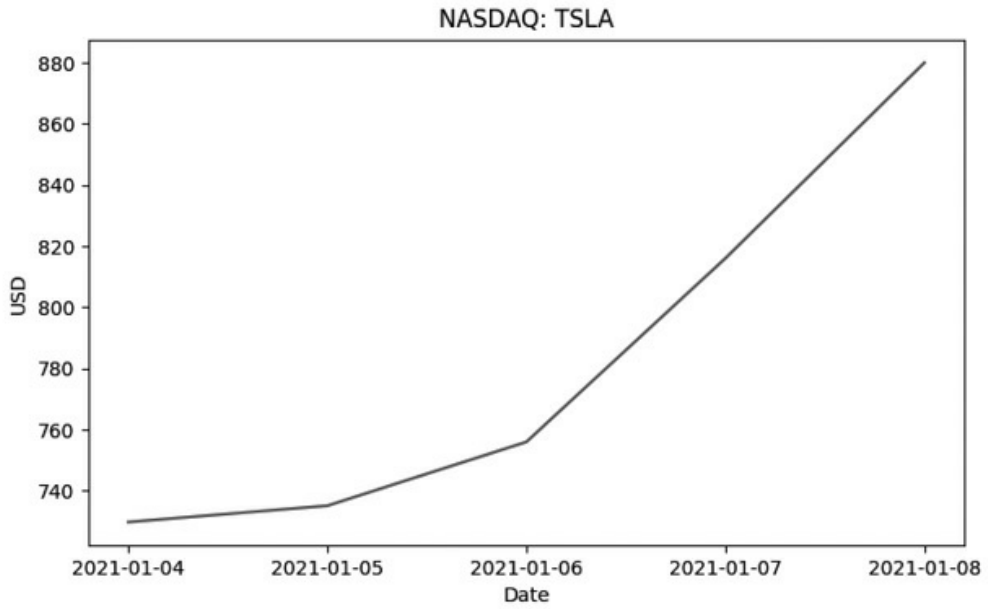


Figura 8.6: Gráfico de linhas simples gerado com o módulo `matplotlib.pyplot`.

Por padrão, `plt.plot()` gera uma visualização renderizada como uma série de linhas conectando os pontos de dados, que são plotados nos eixos x e y. A Matplotlib escolheu automaticamente um intervalo de 720 a 880 para o eixo y, demarcado em intervalos de 20, facilitando a visualização do preço das ações de cada dia.

Criar um gráfico de pizza básico é tão simples quanto criar um gráfico de linhas. Por exemplo, o código a seguir gera o gráfico de pizza mostrado anteriormente na Figura 8.4:

```
import matplotlib.pyplot as plt

regions = ['New England', 'Mid-Atlantic', 'Midwest']
sales = [882703, 532648, 714406]

plt.pie(sales, labels=regions, autopct='%1.1f%%')
plt.title('Sales per Region')
plt.show()
```

O script segue o mesmo padrão básico usado para criar o gráfico de linhas: definimos os dados a serem plotados, criamos um plot, personalizamos alguns de seus recursos e o exibimos. Desta vez, os dados consistem em uma lista de regiões, que servirão como rótulos para cada fatia do gráfico de pizza, e uma lista dos totais de vendas para cada região, a qual definirá o tamanho de cada fatia. Para transformar a plotagem em um gráfico de pizza em vez de um gráfico de linhas, basta chamarmos a função `plt.pie()`, passando `sales` como os dados a serem plotados e `regions` como rótulos para esses dados. Utilizamos também o parâmetro `autopct` a fim de exibir valores percentuais nas fatias de pizza, com a formatação de strings do Python para mostrar os valores até a casa decimal mais próxima de 1%.

Aqui, podemos visualizar os mesmos dados de entrada como um gráfico de barras, como o da Figura 8.3:

```
import matplotlib.pyplot as plt
regions = ['New England', 'Mid-Atlantic', 'Midwest']
sales = [882703, 532648, 714406]

plt.bar(regions, sales)
plt.xlabel("Regions")
plt.ylabel("Sales")
plt.title("Annual Sales Aggregated on a Regional Basis")

plt.show()
```

Passamos a lista `regions` para a função `plt.bar()` como rótulos do eixo x para as barras. O segundo argumento que passamos para `plt.bar()` é a lista com os valores de vendas correspondentes aos itens em `regions`. Tanto aqui quanto no exemplo do gráfico de pizza, é possível usar listas separadas para rótulos e valores de vendas, pois a ordem dos elementos em uma lista Python é persistente.

Trabalhando com os objetos Figure e Axes

Internamente, uma visualização Matplotlib é criada a partir de dois tipos principais de objetos: um objeto `Figure` e um ou mais objetos `Axes`. Nos exemplos anteriores, o `matplotlib.pyplot` serviu como interface para trabalhar indiretamente com esses objetos, possibilitando personalizar alguns elementos de uma visualização. No entanto, podemos controlar mais nossas

visualizações trabalhando diretamente com os próprios objetos `Figure` e `Axes`.

O objeto `Figure` é o contêiner externo de nível superior para uma visualização Matplotlib, podendo incluir um ou mais gráficos. Manipulamos um objeto `Figure` quando precisamos fazer algo com a visualização geral, como redimensioná-lo ou salvá-lo em um arquivo. Nesse ínterim, cada objeto `Axes` representa um gráfico na figura. Usamos um objeto `Axes` para personalizar um gráfico e definir seu layout. Por exemplo, podemos definir o sistema de coordenadas do gráfico e marcar posições em um eixo.

Acessamos os objetos `Figure` e `Axes` por meio da função `matplotlib.pyplot.subplots()`. Quando invocada sem argumentos, essa função retorna uma instância de `Figure` e uma única instância de `Axes` associada a `Figure`. Ao adicionar argumentos à função `subplots()`, podemos criar uma instância de `Figure` e múltiplas instâncias associadas de `Axes`. Dito de outro modo, criaremos uma figura com múltiplos gráficos. Por exemplo, uma chamada de `subplots(2,2)` cria uma figura com quatro gráficos, organizados em duas linhas de dois. Cada gráfico é representado por um objeto `Axes`.

NOTA *Para mais detalhes sobre o uso da função* `subplots()`, *confira a documentação da Matplotlib em* https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.subplots.html.

Criando um histograma com a função subplots()

No script a seguir, recorremos à função `subplots()` para criar um objeto `Figure` e um único objeto `Axes`. Em seguida, manipulamos os objetos a fim de gerar o histograma mostrado anteriormente na Figura 8.5, que ilustra a distribuição salarial de uma coleção de funcionários. Além de manipular os objetos `Figure` e `Axes`, trabalhamos também com um módulo da Matplotlib chamado `matplotlib.ticker` para formatar os ticks ao longo do eixo x do gráfico, bem como com o NumPy a fim de definir uma sequência de bins para o histograma em incrementos de US$50.

```
# Importando módulos
import numpy as np
from matplotlib import pyplot as plt
import matplotlib.ticker as ticker
```

```
 # Dados para plotar
1 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]

 # Preparando um histograma
2 fig, ax = plt.subplots()
3 fig.set_size_inches(5.6, 4.2)
4 ax.hist(salaries, bins=np.arange(1100, 1900, 50), edgecolor='black',
        linewidth=1.2)
5 formatter = ticker.FormatStrFormatter('$%1.0f')
6 ax.xaxis.set_major_formatter(formatter)
7 plt.title('Monthly Salaries in the Sales Department')
  plt.xlabel('Salary (bin size = $50)')
  plt.ylabel('Frequency')
  # Exibindo o histograma
  plt.show()
```

Começamos definindo uma lista `salaries` com os dados salariais que queremos visualizar `1`. Em seguida, invocamos a função `subplots()` sem os parâmetros `2`, e a instruímos a criar uma figura contendo um único gráfico. A função retorna uma tupla contendo dois objetos, `fig` e `ax`, representando a figura e o gráfico, respectivamente.

Agora que temos essas instâncias de `Figure` e `Axes`, podemos começar a personalizá-las. Primeiro, invocamos o método `set_size_inches()` do objeto `Figure` para redimensionar a figura geral `3`. Depois, invocamos o método `hist()` do objeto `Axes` para plotar um histograma `4`. Passamos o método da lista `salaries` como os dados de entrada para o histograma, bem como um array do NumPy definindo os pontos do eixo x para os bins do histograma. Geramos o array com a função `arange()` do NumPy, que cria um array de valores uniformemente espaçados dentro de um determinado intervalo (neste caso, incrementos de `50` entre `1100` e `1900`). Utilizamos o parâmetro `edgecolor` do método `hist()` a fim de desenhar limites com linhas pretas para os bins e o parâmetro `linewidth` para definir a largura desses limites.

Em seguida, usamos a função `FormatStrFormatter()` do módulo `matplotlib.ticker` para criar um formatador que prefixará um sinal de dólar a cada rótulo do eixo x `5`. Aplicamos o formatador aos rótulos do eixo x com o método

`set_major_formatter()` do objeto `ax.xaxis` [6]. Por último, definimos os aspectos gerais do gráfico, como título e rótulos do eixo principal, por meio da interface `matplotlib.pyplot` [7] e o exibimos.

Exibindo distribuições de frequência em um gráfico de pizza

Apesar de os histogramas serem adequados para visualizar distribuições de frequência, podemos usar também um gráfico de pizza a fim de apresentar distribuições de frequência como porcentagens. Para exemplificar, nesta seção, veremos como transformar o histograma de distribuição de salários que acabamos de criar em um gráfico de pizza, mostrando os salários distribuídos como partes de um todo.

Antes de criar um gráfico de pizza, é necessário extrair e organizar algumas informações importantes do histograma. Em especial, precisamos saber o valor de salários em cada intervalo de US$50. Podemos usar a função `histogram()` do NumPy para calcular um histograma sem exibi-lo:

```
import numpy as np
count, labels = np.histogram(salaries, bins=np.arange(1100, 1900, 50))
```

Aqui, chamamos a função `histogram()`, passando a mesma lista `salaries` que criamos anteriormente e mais uma vez recorremos à função `arange()` do NumPy para gerar bins uniformemente espaçados. Chamar `histogram()` retorna dois arrays do NumPy: `count` e `labels`. Vejamos o array `count` que representa o número de funcionários com salários em cada intervalo:

```
[0,  0,  2,  5,  7,  9, 10,  8,  6,  4,  2,  0,  1,  0,  0]
```

Nesse meio-tempo, o array `labels` contém as bordas dos intervalos do bin:

```
[1100, 1150, 1200, 1250, 1300, 1350, 1400, 1450, 1500, 1550, 1600, 1650,
1700, 1750, 1800, 1850]
```

Depois, é necessário combinar os elementos vizinhos do array `labels`, transformando-os nos rótulos das fatias do gráfico de pizza. Por exemplo, os elementos vizinhos `1100` e `1150` devem se tornar um único rótulo formatado como `'$1100-1150'`. Use a seguinte list comprehension:

```
labels = ['$'+str(labels[i])+'-'+str(labels[i+1]) for i, _ in
enumerate(labels[1:])]
```

Vejamos como ficará a lista `labels`:

```
['$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']
```

Na lista `labels`, cada elemento corresponde ao elemento no array `count` com o mesmo índice. No entanto, se analisarmos mais uma vez o array `count`, percebemos um problema: como o cálculo para alguns intervalos é `0`, não queremos incluir intervalos vazios no gráfico de pizza. Para excluí-los, é necessário gerar uma lista dos índices correspondentes a intervalos não vazios no array `count`:

```
non_zero_pos = [i for i, x in enumerate(count) if x != 0]
```

Agora, podemos usar `non_zero_pos` para filtrar `count` e `labels`, excluindo os elementos que representam intervalos vazios:

```
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]
```

Agora só falta criar e exibir o gráfico de pizza usando a interface `matplotlib.pyplot` e `plt.pie()`:

```
from matplotlib import pyplot as plt
plt.pie(count, labels=labels, autopct='%1.1f%%')
plt.title('Monthly Salaries in the Sales Department')
plt.show()
```

A Figura 8.7 mostra o resultado.

O gráfico de pizza apresenta os mesmos dados que o histograma na Figura 8.5, porém mostra cada bin como uma porcentagem do todo, em vez de sinalizar exatamente quantos funcionários têm salários que pertencem a esse bin.

EXERCÍCIO #12: COMBINANDO BINS EM “OUTRA” FATIA

Ao examinar o gráfico na Figura 8.7, percebemos que alguns intervalos são representados por uma fatia muito fina na pizza. São bins com o cálculo de um ou dois funcionários. Modifique o gráfico para que esses intervalos sejam mesclados em uma única fatia chamada `Other`. Para tal, você precisará alterar o array `count` e a lista `labels`. Em seguida, precisará recriar o gráfico.

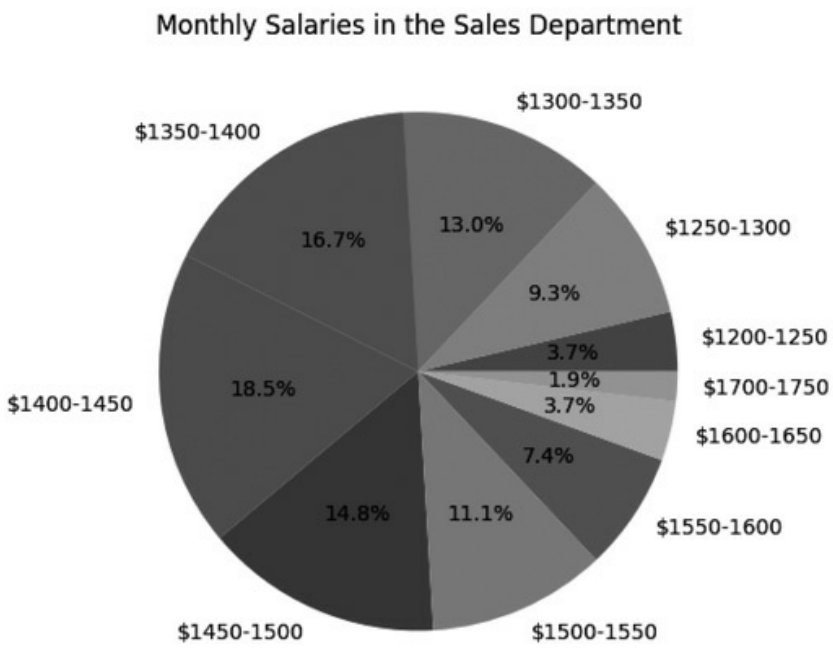


Figura 8.7: Gráfico de pizza de uma distribuição de frequência.

Usando outras bibliotecas com a Matplotlib

A Matplotlib pode interagir facilmente com outras bibliotecas do Python para plotar dados de diferentes fontes ou para criar outros tipos de visualizações. Por exemplo, é possível usar a Matplotlib em conjunto com o pandas para plotar os dados de um DataFrame ou, ainda, criar mapas combinando a Matplotlib com a Cartopy, biblioteca especializada na manipulação de dados geoespaciais.

Plotando dados do pandas

A biblioteca pandas está estritamente integrada à Matplotlib. Na realidade, cada Series ou DataFrame do pandas tem um método `plot()`, que na verdade é um wrapper em torno do método `matplotlib.pyplot.plot()`, possibilitando converter diretamente uma estrutura de dados do pandas em um gráfico da Matplotlib. Para ver como isso funciona, criaremos um gráfico de barras a partir de um DataFrame com dados populacionais de cidades dos EUA. Usaremos dados brutos do arquivo *us-cities-top-1k.csv* disponível em https://github.com/plotly/datasets. O gráfico de barras mostrará o número de megacidades (aquelas com população de 1 milhão ou mais) em cada estado

dos EUA. Vejamos como criar esse gráfico:

```
  import pandas as pd
  import matplotlib.pyplot as plt

  # Preparando o DataFrame
1 us_cities = pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/
                              master/us-cities-top-1k.csv")
2 top_us_cities = us_cities[us_cities.Population.ge(1000000)]
3 top_cities_count = top_us_cities.groupby(['State'], as_index = False)
                   .count().rename(columns={'City': 'cities_count'})
                      [['State','cities_count']]

  # Desenhando o gráfico
4 top_cities_count.plot.bar('State', 'cities_count', rot=0)
5 plt.xlabel("States")
  plt.ylabel("Top cities count")
  plt.title("Number of Megacities per US State")
6 plt.yticks(range(min(top_cities_count['cities_count']),
                 max(top_cities_count['cities_count'])+1 ))
  plt.show()
```

Primeiro, carregamos o conjunto de dados em um DataFrame com o método `read_csv()` do pandas `1`. O conjunto de dados contém a população, a latitude e a longitude das mil maiores cidades dos EUA. Para filtrar o conjunto de dados somente por megacidades, utilizamos o método `ge()` do DataFrame, abreviação em inglês de *greater than or equal to* (maior ou igual a), solicitando apenas as linhas cujo campo `Population` seja maior ou igual a `1000000` `2`. Em seguida, agrupamos os dados pela coluna `State` e usamos a função de agregação `count()` a fim de encontrar o número total de megacidades por estado `3`. Na operação `groupby`, definimos `as_index` como `False` para que a coluna `State` no índice do DataFrame resultante não seja convertida. Isso ocorre, porque, mais adiante, precisaremos referenciar a coluna `State` no script. Agora, renomeamos a coluna `City` como `cities_count` para que armazene as informações agregadas e inclua somente as colunas `State` e `cities_count` no DataFrame resultante `top_cities_count`.

Em seguida, desenhamos um gráfico de barras com o método `plot.bar()` do DataFrame `4`. Lembre-se, o `plot()` é na verdade um wrapper para o método `pyplot.plot()` da Matplotlib. Nessa chamada, especificamos os nomes das colunas do DataFrame que serão usados como eixos x e y do gráfico e

rotacionamos os rótulos de escala do eixo x para 0 graus. Após criar a figura, podemos personalizá-la com a interface `matplotlib.pyplot`, como fizemos nos exemplos anteriores. Definimos os rótulos dos eixos e o título da figura `5` e usamos `plt.yticks()` a fim de definir os rótulos numéricos para o eixo y, retratando o cálculo das principais cidades `6`. Por último, exibimos a figura com `plt.show()`. A Figura 8.8 mostra o resultado.

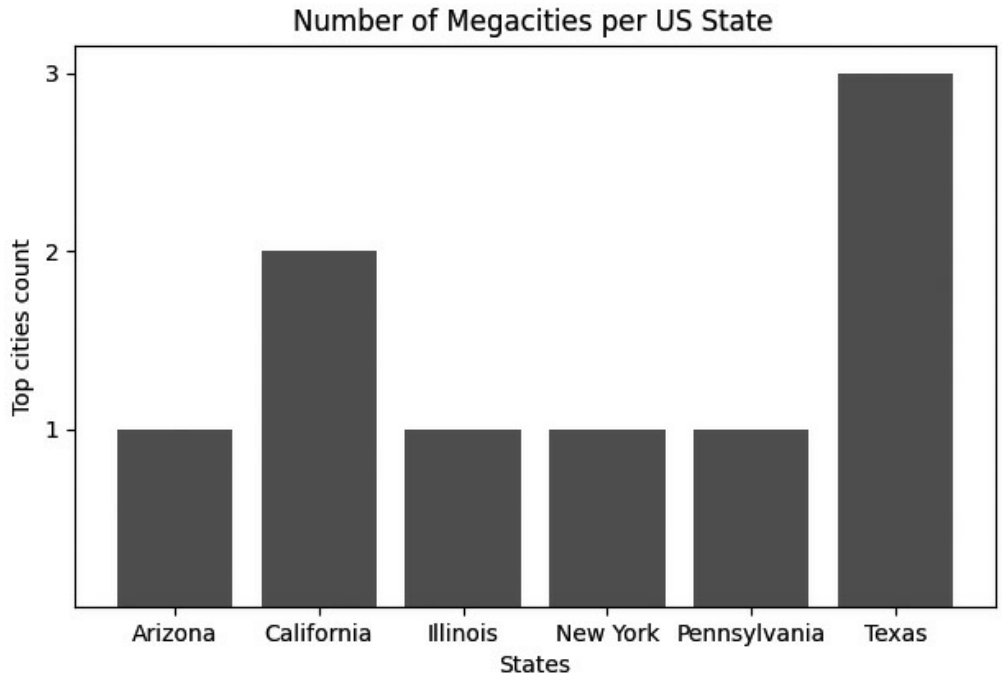


Figura 8.8: Gráfico de barras gerado a partir de um DataFrame do pandas.

Conforme podemos observar, a figura é parecida com as outras que criamos neste capítulo, precisamente com o gráfico de barras na Figura 8.3. Não é nenhuma surpresa, já que foi gerado pela mesma biblioteca Matplotlib, a qual o pandas usa nos bastidores.

Plotando dados geoespaciais com a Cartopy

A Cartopy é uma biblioteca Python para criar visualizações geoespaciais ou mapas. Inclui a interface programática `matplotlib.pyplot`, facilitando o desenho de mapas. Basicamente, para desenhar um mapa com a Cartopy, basta criar uma figura Matplotlib com coordenadas de longitude plotadas ao longo do eixo x e coordenadas de latitude plotadas ao longo do eixo y. A Cartopy se encarrega da complexidade de converter a forma esférica da Terra em um

plano 2D para o gráfico. Para exemplificar, usaremos o conjunto de dados *us-cities-top-1k.csv*, apresentado na seção anterior, para desenhar mapas do sul da Califórnia mostrando a localização de diversas cidades. Mas, primeiro, é necessário configurar a Cartopy.

Usando a Cartopy no Google Colab

Instalar a Cartopy pode ser complicado, e o processo varia dependendo do sistema de cada um. Assim sendo, nesta seção, veremos como usar a Cartopy por meio do Google Colab Web IDE, que possibilita escrever e executar código Python em um navegador.

Para carregar o Colab, acesse https://colab.research.google.com. Em seguida, clique em Novo notebook para iniciar um novo notebook do Colab, em que podemos criar, preencher e executar um número arbitrário de células de código. Em cada célula de código, é possível agrupar uma ou mais linhas de código Python e executá-las clicando no botão Executar célula, no canto superior esquerdo da célula. O Colab indica o estado de execução criado por qualquer célula executada anteriormente, semelhante a uma sessão do interpretador Python. Criamos células novas de código com o botão +Código no canto superior esquerdo da janela do Colab.

Na primeira célula de código, digitamos e executamos o seguinte comando para instalar a Cartopy em nosso notebook Colab:

```
!pip install cartopy
```

Após instalar a Cartopy, podemos seguir com exemplos na próxima seção, executando cada listagem separada na própria célula de código.

NOTA *Caso prefira instalar a Cartopy diretamente em seu sistema, confira a documentação em* https://scitools.org.uk/cartopy/docs/latest/installing.html.

Criando mapas

Nesta seção, usaremos a Cartopy para criar dois mapas do sul da Califórnia. Primeiro, vamos desenhar um mapa que mostre todas as cidades do sul da Califórnia incluídas no conjunto de dados *us-cities-top-1k.csv*. Começamos importando todos os módulos necessários:

```
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
from cartopy.mpl.ticker import LongitudeFormatter, LatitudeFormatter
```

Precisaremos do pandas, da interface `matplotlib.pyplot` e de alguns módulos diferentes da Cartopy: `cartopy.crs` para gerar mapas, `LongitudeFormatter` e `LatitudeFormatter` para formatar corretamente os rótulos de escala. O comando `%matplotlib inline` é necessário para incluir figuras da Matplotlib em um notebook do Google Colab, ao lado do código.

Em seguida, carregamos os dados necessários e desenhamos o mapa:

```
1 us_cities = pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/
                           master/us-cities-top-1k.csv")
2 calif_cities = us_cities[us_cities.State.eq('California')]
3 fig, ax = plt.subplots(figsize=(15,8))
4 ax = plt.axes(projection=ccrs.Mercator())
5 ax.coastlines('10m')
6 ax.set_yticks([32,33,34,35,36], crs=ccrs.PlateCarree())
  ax.set_xticks([-121, -120, -119, -118, -117, -116, -115],
                crs=ccrs.PlateCarree())
7 lon_formatter = LongitudeFormatter()
  lat_formatter = LatitudeFormatter()
  ax.xaxis.set_major_formatter(lon_formatter)
  ax.yaxis.set_major_formatter(lat_formatter)
8 ax.set_extent([-121, -115, 32, 36])
  X = calif_cities['lon']
  Y = calif_cities['lat']
9 ax.scatter(X, Y, color='red', marker='o', transform=ccrs.PlateCarree())
  plt.show()
```

Carregamos o conjunto de dados *us-cities-top-1k.csv* em um DataFrame `1`, como fizemos na seção anterior. Lembre-se, o conjunto contém dados geoespaciais na forma de coordenadas de latitude e de longitude, bem como dados populacionais. Depois, filtramos os dados para incluir somente as cidades da Califórnia com o método `eq()` do DataFrame `2`, abreviação em inglês de *equal to* (igual a).

Como desenhar um mapa exige mais personalização do que a `interface matplotlib.pyplot` permitirá, precisamos trabalhar diretamente com os objetos da

Matplotlib subjacentes da visualização. Por isso, chamamos a função `plt.subplots()` a fim de obter um objeto `Figure` e um único objeto `Axes`, definindo o tamanho da figura no processo `3`. Depois, chamamos `plt.axes()` para sobrescrever o objeto `Axes`, transformando-o em um mapa da Cartopy `4`. Fazemos isso instruindo a Matplotlib a usar a projeção Mercator da Cartopy ao plotar as coordenadas na superfície plana da figura. A projeção Mercator é uma técnica padrão de criação de mapas que converte a Terra de uma esfera em um cilindro e, em seguida, desenrola esse cilindro em um retângulo.

Depois, chamamos `ax.coastlines()` para mostrar os contornos das massas terrestres no mapa `5`. Os contornos costeiros são adicionados ao atual objeto `Axes` a partir da coleção Natural Earth de *faixas costeiras* com formato shapefile. Ao especificar `10m`, desenhamos as linhas costeiras em uma escala de 1 a 10 milhões; ou seja, no mapa, 1 centímetro equivale a 100 quilômetros do mundo cotidiano.

NOTA *Visite* `https://www.naturalearthdata.com` *para obter mais informações sobre os conjuntos de dados do Natural Earth.*

Para definir quais serão os ticks ao longo dos eixos y e x, recorremos aos métodos `set_yticks()` e `set_xticks()`, passando uma lista de latitudes e de longitudes para ambos `6`. Em termos específicos, passamos de `32` a `36` como y e `-121` a `-115` como x (ou seja, 32 N a 36 N e 121 W a 115 W), pois essas latitudes e longitudes abrangem a área sul da Califórnia. Em ambos os casos, temos de adicionar `crs=ccrs.PlateCarree()` para especificar como as informações de latitude e de longitude devem ser projetadas em uma superfície plana. Como a projeção Mercator, a Plate Carrée trata a Terra como um cilindro que foi achatado em um retângulo.

Em seguida, criamos formatadores com os objetos `LongitudeFormatter()` e `LatitudeFormatter()` da Cartopy e os aplicamos aos eixos x e y `7`. Usar esses formatadores garante que os valores de longitude e de latitude sejam mostrados com sinais de grau e um *W* ou *N* para *oeste* ou *norte*, respectivamente. Definimos também a extensão do gráfico, especificando as longitudes e as latitudes adequadas para limitar o mapa a mostrar somente o sul da Califórnia `8`. Depois, extraímos dois objetos Series do pandas do nosso DataFrame, `X` e `Y`, para os valores de longitude e latitude, respectivamente. Por

último, desenhamos o mapa com o método `scatter()` 9 da Matplotlib, passando os dados a fim de plotar os eixos x e y com instruções para exibir as cidades como pontos vermelhos. A Figura 8.9 mostra o resultado.

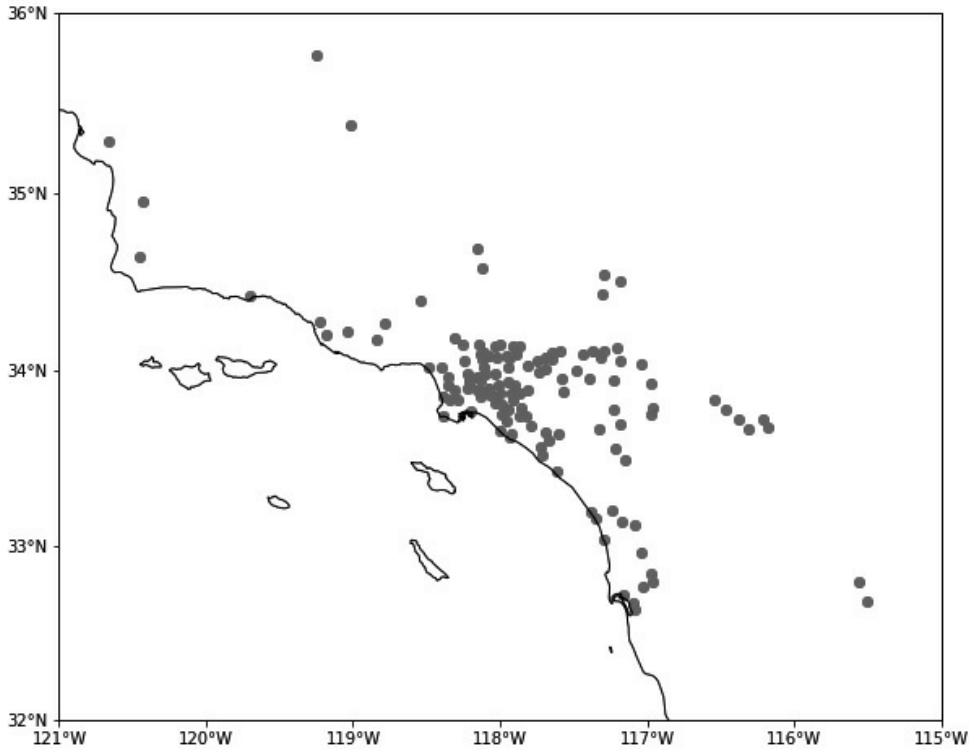


Figura 8.9: Mapa de esboço do sul da Califórnia com as cidades.

O mapa fornece uma ilustração clara das áreas com alta densidade populacional. Mas e se quisermos ver apenas as maiores cidades marcadas com seus nomes? Vejamos como podemos fazer isso:

```
1 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])
  X = top_calif_cities['lon']
  Y = top_calif_cities['lat']
2 cities = top_calif_cities['City']
```

```
  ax.scatter(X, Y, color='red', marker='o', transform=ccrs.PlateCarree())
3 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()
```

Filtramos o DataFrame gerado `calif_cities` na listagem anterior para incluir apenas as cidades com populações de 400.000 ou mais `1`. Em seguida, geramos o gráfico seguindo o mesmo processo de antes, com algumas etapas extras para adicionar rótulos de cidade. Armazenamos os nomes das cidades em uma Series do pandas chamada `cities` `2` e, em seguida, iteramos os nomes das cidades, atribuindo-os como rótulos centralizados nos pontos do mapa com o método `plt.text()` da Matplotlib `3`. Especificamos `transform=ccrs.Geodetic()` para fazer a Matplotlib usar o sistema de coordenadas geodésicas da Cartopy ao adicionar os rótulos. Esse sistema trata a Terra como uma esfera e especifica coordenadas como valores de latitude e de longitude. A Figura 8.10 mostra o resultado.

Agora, o mapa mostra os locais e os nomes das três cidades no sul da Califórnia com populações acima de 400.000.

EXERCÍCIO #13: DESENHANDO UM MAPA COM A CARTOPY E COM A MATPLOTLIB

Agora que você sabe como criar um mapa com as bibliotecas Cartopy e Matplotlib, crie um que mostre as cidades em outro local nos EUA. Por exemplo, é possível criar um mapa para o norte da Califórnia. Você precisará especificar diferentes valores de latitude e de longitude para os eixos y e x.

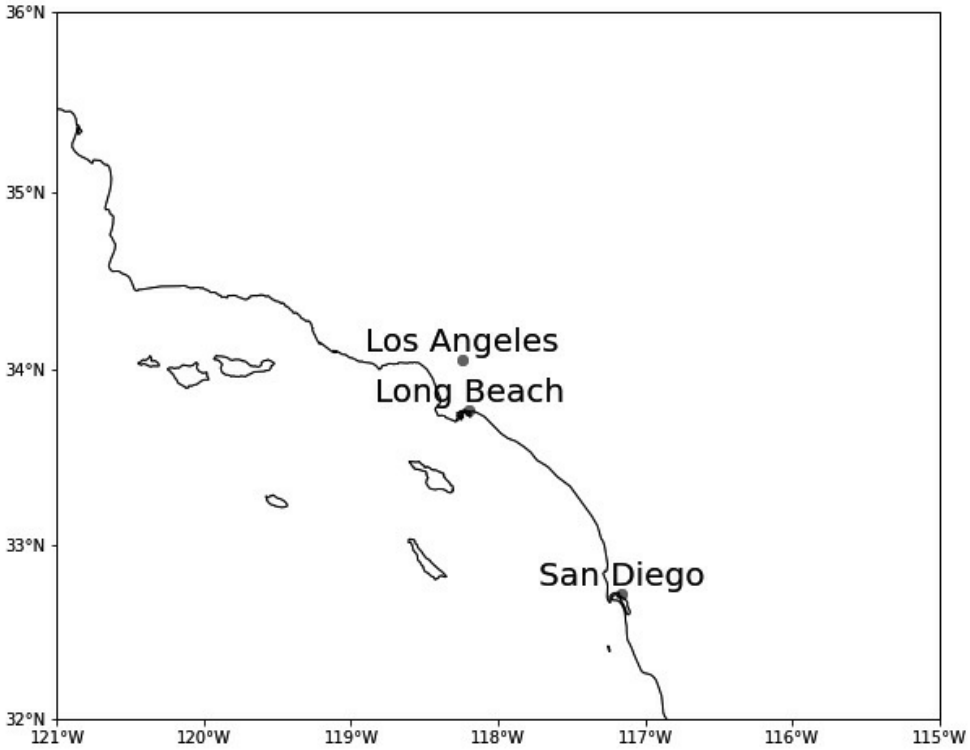


Figura 8.10: As maiores cidades do sul da Califórnia.

Recapitulando

Como vimos, as visualizações de dados são ferramentas poderosas para identificar tendências e para obter insights dos dados. Por exemplo, um gráfico de linhas revela de imediato tendências no preço de uma ação, enquanto um mapa pode demonstrar claramente áreas de alta densidade populacional. Neste capítulo, aprendemos a criar visualizações comuns como gráficos de linhas, gráficos de barras, gráficos de pizza e histogramas com a biblioteca Matplotlib. Vimos como criar visualizações simples, mas poderosas, com a interface `matplotlib.pyplot` e como exercer mais controle sobre o resultado manipulando diretamente os objetos subjacentes `Figure` e `Axes` de uma visualização. Além do mais, aprendemos a usar a Matplotlib com o pandas para visualizar dados do DataFrame e, como prática, criamos mapas com a Matplotlib e com a Cartopy, biblioteca de processamento de dados geoespaciais.

CAPÍTULO 9

Analisando dados geográficos

Tudo acontece em algum lugar. Por isso, a localização de um objeto pode ser tão importante quanto seus atributos não espaciais para fins de análise de dados. Na verdade, os dados espaciais e não espaciais normalmente são indissociáveis.

Por exemplo, imagine um aplicativo de serviço de compartilhamento de viagens. Após solicitar uma corrida, talvez você queira rastrear a localização do carro em um mapa e em tempo real enquanto o espera. Talvez também queira saber algumas informações básicas não espaciais a respeito do carro e do motorista atribuídos à sua solicitação de corrida: a marca e o modelo do carro, a classificação do motorista e assim por diante.

No capítulo anterior, vimos como trabalhar com dados geográficos para gerar mapas. Neste capítulo, nos aprofundaremos em como usar o Python para coletar e analisar dados geográficos e veremos como integrar dados espaciais e não espaciais em nossa análise. Ao longo de nosso percurso de aprendizagem, recorreremos ao exemplo de um serviço de gerenciamento de táxi e tentaremos responder à questão fundamental de qual táxi deve ser atribuído a determinada tarefa.

Obtendo os dados geográficos

A primeira etapa para realizar uma análise espacial é obter os dados geográficos para os objetos de interesse. Em termos específicos, esses dados geográficos devem assumir a forma de *coordenadas geográficas* (abreviação: *geo coordenadas*) ou valores de latitude e de longitude. Esse sistema de coordenadas viabiliza que todos os locais do planeta sejam especificados como um conjunto de números. Ou seja, os locais podem ser

programaticamente analisados. Nesta seção, veremos maneiras de obter as coordenadas geográficas de objetos estacionários e móveis. Isso demonstrará como nosso exemplo de serviço de táxi pode determinar o local de partida de um passageiro, bem como a localização em tempo real de seus múltiplos táxis.

Transformando um endereço legível por humanos em coordenadas geográficas

A maioria dos seres humanos pensa em nomes de ruas e em números de edifícios, em vez de pensar em coordenadas geográficas. Por isso, é comum que serviços de táxi, aplicativos de delivery de alimentos e afins permitam que os usuários especifiquem locais de partida e de destino na forma de endereços de rua. Contudo, nos bastidores, muitos desses serviços convertem endereços legíveis por humanos em coordenadas geográficas correspondentes. Assim, o aplicativo pode realizar cálculos com os dados geográficos, como determinar o taxista disponível mais próximo do local especificado de partida.

Como converter endereços de rua em coordenadas geográficas? Uma forma é usar a Geocoding, uma API fornecida pelo Google para esse propósito. Para interagir com a Geocoding API a partir de um script Python, precisaremos usar a biblioteca googlemaps. Vejamos como instalá-la com o comando `pip`:

```
$ pip install -U googlemaps
```

Além disso, precisaremos obter uma chave de API para a Geocoding API usando uma conta do Google Cloud. Para obter informações sobre como adquirir uma chave de API, confira https://developers.google.com/maps/documentation/geocoding/get-api-key. Os detalhes sobre a estrutura de custos da API estão disponíveis em https://cloud.google.com/maps-platform/pricing. No momento em que eu escrevia esta obra, o Google fornecia um crédito mensal de US$200 para os usuários da API, o suficiente para testar o código deste livro.

O script a seguir ilustra uma chamada de exemplo para a Geocoding API usando a googlemaps. Essa chamada obtém as coordenadas de latitude e de longitude correspondentes ao endereço 1600 Amphitheatre Parkway,

Mountain View, CA:

```
import googlemaps

gmaps = googlemaps.Client(key='INSIRA_SUA_CHAVE_DE_API_AQUI')
address = '1600 Amphitheatre Parkway, Mountain View, CA'
geocode_result = gmaps.geocode(address)

print(geocode_result[0]['geometry']['location'].values())
```

Nesse script, estabelecemos uma conexão com a API e enviamos o endereço que queremos converter. A API retorna um documento JSON com uma estrutura aninhada. As coordenadas geográficas são armazenadas na chave `location`, que é um subcampo de `geometry`. Na última linha, acessamos e exibimos as coordenadas, gerando a seguinte saída:

```
dict_values([37.422388, -122.0841883])
```

Obtendo as coordenadas geográficas de um objeto em movimento

Agora que sabemos como obter as coordenadas geográficas de uma localização fixa por meio de seu endereço, como podemos obter as coordenadas geográficas em tempo real de um objeto em movimento, como um táxi? Alguns serviços de táxi podem usar dispositivos GPS especializados, mas, aqui, focaremos uma solução de baixo custo e de fácil implementação. Tudo o que precisamos é de smartphone.

Os smartphones detectam nossa localização por meio de sensores GPS integrados e podemos configurá-los para compartilhar essas informações. Aqui, veremos como coletar as coordenadas do GPS do smartphone por meio do popular aplicativo de mensagens Telegram. Com a Telegram Bot API, criaremos um *bot*, um app executado no Telegram. Os bots são comumente usados para processamento de linguagem natural, mas o nosso coletará e registrará os dados de geolocalização dos usuários do Telegram que optarem por compartilhar seus dados com o bot.

Configurando um bot do Telegram

Para criar um bot, é necessário fazer o download do app Telegram e criar uma conta. Em seguida, siga a seguintes etapas usando um smartphone ou um

computador:

1. No app Telegram, procure por @BotFather. O BotFather é um bot do Telegram que gerencia todos os outros bots em sua conta.
2. Na página do BotFather, clique em Iniciar para ver a lista de comandos que podemos usar para configurar bots do Telegram.
3. Digite `/newbot` na caixa de mensagem. Você será solicitado a fornecer um nome e um nome de usuário para seu bot. Em seguida, receberá um token de autorização para o bot novo. Anote esse token; você precisará dele quando programar o bot.

Após concluir essas etapas, é possível implementar o bot com o Python usando a biblioteca python-telegram-bot. Vejamos como instalar essa biblioteca:

```
$ pip install python-telegram-bot -upgrade
```

As ferramentas necessárias para programar o bot estão no módulo `telegram.ext` da biblioteca, que foi desenvolvido com base na Telegram Bot API.

Programando o bot

Aqui, recorreremos ao módulo `telegram.ext` da biblioteca python-telegram-bot a fim de programar o bot para ouvir e registrar coordenadas do GPS:

```
from telegram.ext import Updater, MessageHandler, Filters
from datetime import datetime
import csv

1 def get_location(update, context):
    msg = None
    if update.edited_message:
      msg = update.edited_message
    else:
      msg = update.message
  2 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)
    3 writer.writerow([sender, gps.latitude, gps.longitude, tm])
    4 context.bot.send_message(chat_id=msg.chat_id, text=str(gps))

  def main():
```

```
5 updater = Updater('TOKEN', use_context=True)
6 updater.dispatcher.add_handler(MessageHandler(Filters.location,
                                 get_location))
 7 updater.start_polling()
 8 updater.idle()

if __name__ == '__main__':
    main()
```

A função `main()` contém as invocações comuns encontradas em um script que implementa um bot do Telegram. De início, criamos um objeto `Updater` `5`, passando o token de autorização do bot (gerado pelo BotFather). Esse objeto orquestra o processo de execução do bot em todo o script. Em seguida, utilizamos o objeto `Dispatcher` associado ao `Updater` a fim de adicionar uma função handler chamada `get_location()` para as mensagens recebidas `6`. Ao especificar `Filters.location`, adicionamos um filtro à função handler para que apenas seja chamada quando o bot receber mensagens que incluam os dados geográficos do remetente. Inicializamos o bot invocando o método `start_polling()` do objeto `Updater` `7`. Como `start_polling()` é um método non-blocking, devemos também chamar o método `idle()` do objeto `Updater` `8` para bloquear o script até que uma mensagem seja recebida.

No início do script, definimos a função handler `get_location()` `1`. Dentro da handler, armazenamos as mensagens recebidas como `msge`, em seguida extraímos os dados geográficos do remetente usando a propriedade `location` da mensagem `2`. Registramos também o nome de usuário do remetente e geramos uma string contendo a hora atual. Depois, com o módulo `csv` do Python, armazenamos todas essas informações como uma linha em um arquivo CSV `3` em um local de nossa escolha. Reenviamos também os dados geográficos ao remetente para que saiba que sua localização foi recebida `4`.

Obtendo dados do bot

Execute o script em uma máquina conectada à internet. Depois de executado, os usuários podem seguir algumas etapas simples para começar a compartilhar seus dados geográficos em tempo real com o bot:

1. Crie uma conta no Telegram.
2. No Telegram, toque no nome do bot.

3. Toque no ícone de clipe de papel e selecione Localização no menu.
4. Escolha Compartilhar Minha Localização Por e defina por quanto tempo o Telegram compartilhará dados geográficos em tempo real com o bot. As opções incluem 15 minutos, 1 hora ou 8 horas.

O screenshot na Figura 9.1 mostra como é fácil compartilhar a localização em tempo real no Telegram.

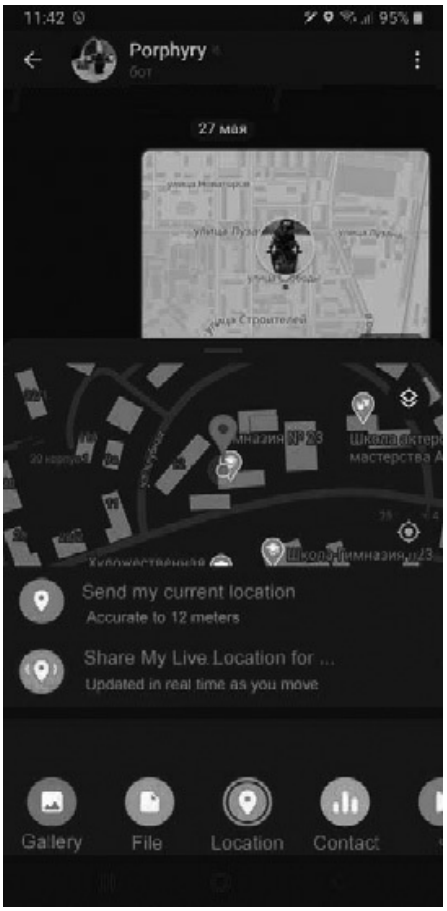


Figura 9.1: Compartilhando a localização em tempo real por meio de um smartphone no Telegram.

Depois que os usuários começarem a compartilhar seus dados geográficos, o bot começará a enviá-los para um arquivo CSV na forma de linhas, mais ou menos assim (lembre-se, no código, táxi é cab):

```
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
```

O primeiro campo de cada linha contém um nome de usuário, o segundo e o terceiro campos contêm a latitude e a longitude da localização do usuário e o quarto campo contém um timestamp. Para algumas tarefas, como encontrar o carro mais próximo de um determinado local de partida, precisamos somente da última linha para cada carro. No entanto, para outras tarefas, como calcular a distância total de uma corrida, seria vantajoso múltiplas linhas de dados para determinado carro, ordenadas por tempo.

Análise de dados espaciais com geopy e Shapely

A análise de dados espaciais se resume a responder a perguntas sobre relacionamentos: qual objeto está mais próximo de um determinado local? Dois objetos estão na mesma área? Nesta seção, responderemos a essas perguntas comuns de análise espacial com duas bibliotecas do Python, geopy e Shapely, tudo dentro do contexto de nosso exemplo de serviço de táxi.

Como foi desenvolvida para realizar cálculos com base em coordenadas geográficas, a geopy é especialmente adequada para responder a perguntas sobre distância. Ao mesmo tempo, a especialidade da Shapely é definir e analisar planos geométricos, por isso é ideal para determinar se um objeto está em uma área específica. Conforme veremos, ambas as bibliotecas podem desempenhar um papel na identificação do melhor táxi para determinada tarefa.

Mas, antes de prosseguirmos, vejamos como instalar essas bibliotecas:

```
$ pip install geopy
$ pip install shapely
```

Encontrando o objeto mais próximo

Continuando com nosso exemplo de serviço de táxi, veremos como usar os dados geográficos para identificar o táxi mais próximo de um local de partida. Para começar, precisaremos de alguns dados geográficos de amostra. Caso tenha implementado o bot do Telegram da seção anterior, talvez já tenha alguns dados na forma de um arquivo CSV. Aqui, carregamos os dados

em um DataFrame do pandas para que possamos ordená-los e filtrá-los facilmente:

```
import pandas as pd
df = pd.read_csv("HOME/PI/LOCATION_BOT/LOG.CSV", names=['cab', 'lat',
                                                        'long', 'tm'])
```

Caso não tenha implementado um bot do Telegram, pode criar uma lista de tuplas com alguns dados geográficos de amostra e carregá-los em um DataFrame da seguinte forma:

```
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'])
```

De qualquer forma, obteremos um DataFrame chamado `df` com colunas para o ID do táxi, latitude, longitude e timestamp.

NOTA *Se quiser criar o próprio conjunto de dados geográficos de amostra para manipular, um método simples é procurar coordenadas de latitude e de longitude usando o Google Maps. Ao clicar com o botão direito do mouse em um local em um mapa, as coordenadas de latitude e de longitude do local serão a primeira coisa a aparecer.*

O DataFrame tem múltiplas linhas para cada táxi, mas, para identificar o táxi mais próximo de um local de partida, precisamos somente da localização mais recente de cada táxi. Podemos filtrar as linhas desnecessárias da seguinte forma:

```
latestrows = df.sort_values(['cab','tm'],ascending=False).drop_duplicates('cab')
```

Aqui, ordenamos as linhas pelos campos `cab` e `tm` em ordem decrescente. Essa operação ordena o conjunto de dados pela coluna `cab` e, primeiro, insere a linha mais recente para cada táxi em seu grupo. Em seguida, usamos o método `drop_duplicates()` para eliminar tudo, exceto a primeira linha de cada táxi. Vejamos o DataFrame `latestrows` resultante:

```
       cab        lat       long        tm
5   cab_26  43.612203  39.720491  14:52:48
3  cab_112  43.574906  39.766325  14:51:53
```

Agora temos um DataFrame com apenas os dados geográficos mais recentes de cada táxi. Para a conveniência de futuros cálculos, convertemos o DataFrame em uma estrutura do Python mais simples, uma lista de listas. Assim, é possível anexar com mais facilidade campos novos a cada linha, como um campo para a distância entre o táxi e o local de partida:

```
latestrows = latestrows.values.tolist()
```

A propriedade `values` de `laterrows` retorna uma representação NumPy do DataFrame, que convertemos em uma lista de listas usando `tolist()`.

Agora, estamos prontos para calcular a distância entre cada táxi e um ponto de partida. Utilizaremos a biblioteca geopy, que consegue realizar essa tarefa com apenas algumas linhas de código. Aqui, recorremos à função `distance()` do módulo `distance` da geopy para efetuar os cálculos necessários:

```
from geopy.distance import distance
pick_up = 43.578854, 39.754995

for i,row in enumerate(latestrows):
1 dist = distance(pick_up, (row[1],row[2])).m
  print(row[0] + ':', round(dist))
  latestrows[i].append(round(dist))
```

Para simplificar as coisas, determinamos o local de partida ao definir manualmente as coordenadas de latitude e de longitude. Contudo, na prática, podemos usar a Geocoding API do Google para gerar as coordenadas automaticamente a partir de um endereço, conforme analisado anteriormente neste capítulo. Em seguida, iteramos com um loop em cada linha do conjunto de dados e calculamos a distância entre cada táxi e o local de partida chamando a função `distance()` `1`. Essa função recebe duas tuplas contendo as coordenadas de latitude/longitude como argumentos. Adicionando `.m`, obtemos a distância em metros. Para fins de demonstração, exibiremos o resultado de cada cálculo de distância; em seguida, o anexamos ao final da linha como um campo novo. O script gera a seguinte saída:

```
cab_112: 1015
cab_26: 4636
```

Fica evidente que `cab_112` está mais próximo, mas como determinar isso programaticamente? Com a função built-in do Python `min()`, conforme mostrado a seguir:

```
closest = min(latestrows, key=lambda x: x[4])
print('The closest cab is: ', closest[0], ' - the distance in meters: ',
closest[4])
```

Fornecemos os dados para `min()` e usamos uma função lambda para avaliar sua ordenação com base no item do índice `4` de cada linha. Agora, temos o cálculo de distância recém-anexado. Exibimos então o resultado em um formato legível por humanos, que gera a seguinte saída:

```
The closest cab is:  cab_112  - the distance in meters:  1015
```

Neste exemplo, calculamos a distância em linha reta entre cada táxi e o local de partida. Ainda que essas informações sejam certamente úteis, em nosso mundo cotidiano, os carros quase nunca seguem uma linha perfeitamente reta quando nos deslocamos de um local para o outro. Devido à estruturação das ruas, a distância real que um táxi deve percorrer para chegar a um local de partida será maior do que a distância em linha reta. Considerando isso, veremos a seguir uma forma mais confiável de fazer a correspondência dos locais de partida com os táxis.

Encontrando objetos em determinada área

Não raro, a pergunta certa a ser feita para determinar o melhor táxi não é “Qual é o táxi mais próximo?”, e sim “Qual táxi está em determinada área que abrange o local de partida?”. Isso não ocorre somente porque a distância percorrida entre dois pontos é quase sempre maior que a distância em linha reta entre eles. Na prática, barreiras como rios ou trilhos ferroviários geralmente dividem áreas geográficas em zonas separadas, conectadas somente em um número limitado de pontos por pontes, por túneis e similares. Assim, as distâncias em linha reta podem induzir e muito ao erro. Vejamos o exemplo da Figura 9.2.

Figura 9.2: Obstáculos como rios podem induzir os cálculos de distância ao erro.

Como podemos verificar, neste cenário, `cab_26` está espacialmente mais próximo do local de partida, mas, devido ao rio, `cab_112` provavelmente será capaz de chegar mais rápido. Quando olhamos um mapa, podemos perceber isso de imediato, mas como chegar à mesma conclusão com um script Python? Uma forma é dividir a área em diversos *polígonos* menores, ou áreas delimitadas por um conjunto de linhas retas conectadas e, em seguida, verificar quais táxis estão dentro do mesmo polígono do local de partida.

Neste exemplo específico, devemos definir um polígono que englobe o local de partida e tenha um perímetro ao longo do rio. No Google Maps, é possível identificar o perímetro do polígono manualmente: basta clicar com o botão direito do mouse em vários pontos que se conectam para formar um polígono fechado e anotar as coordenadas geográficas de cada ponto. Após obter as coordenadas, podemos definir o polígono no Python com a biblioteca Shapely.

Vejamos como criar um polígono com a Shapely e verificar se um determinado ponto está dentro desse polígono:

```
1 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)]
2 poly = Polygon(coords)
3 cab_26 = Point(46.073852, 38.991890)
  cab_112 = Point(46.078228, 39.003949)
  pick_up = Point(46.080074, 38.991289)

4 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))
```

Primeiro, importamos duas classes Shapely, `Point` e `Polygon` 1, e, depois, criamos um objeto `Polygon` usando uma lista de tuplas de latitude/longitude 2. Esse objeto representa a área ao norte do rio, incluindo o local de partida. Em seguida, criamos diversos objetos `Point` representando os locais de `cab_26`, `cab_112` e o local de partida, respectivamente 3. Por último, com o método `within()` da Shapely 4, executamos uma série de queries espaciais para detectar se um determinado ponto está dentro do polígono. Como resultado, o script deve gerar a seguinte saída:

```
cab_26 within the polygon: False
cab_112 within the polygon: True
pick_up within the polygon: True
```

EXERCÍCIO #14: DEFININDO DOIS OU MAIS POLÍGONOS

Na seção anterior, usamos um único polígono para englobar uma área no mapa. Agora, tente definir dois ou mais polígonos que englobem áreas urbanas adjacentes e divididas por um obstáculo, como um rio. Obtenha as coordenadas para esses polígonos por meio do mapa do Google de seu município ou de sua cidade, ou de qualquer outra área urbana do planeta. Você também precisará das coordenadas de diversos pontos dentro desses polígonos para simular a localização de alguns táxis e de um local de partida.

Em seu script, defina os polígonos com a Shapely e os agrupe em um dicionário e, em seguida, agrupe os pontos que representam os táxis em outro dicionário. Depois, divida os táxis em grupos com base no polígono em que estão localizados. É possível fazer isso com dois loops: um loop externo para iterar sobre os polígonos e um loop interno para iterar sobre os pontos que representam os táxis. Assim, você verifica se um ponto está dentro de um polígono em cada iteração do loop interno. O trecho de código a seguir exemplifica como isso pode ser implementado:

```
--trecho de código omitido--
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)
--trecho de código omitido--
```

Em seguida, você precisará determinar qual polígono contém o local de partida. Após determiná-lo, você pode selecionar a lista correspondente de táxis do dicionário `cabs_dict`, usando o nome do polígono como chave. Por último, use a geopy para determinar qual táxi dentro do polígono escolhido está mais próximo do local de partida.

Combinando ambas as abordagens

Até o momento, escolhemos o melhor táxi para um local de partida calculando distâncias lineares e encontrando aquele mais próximo dentro de uma determinada área. Na verdade, talvez a forma mais precisa de encontrar o táxi certo seja usar os elementos de ambas as abordagens. Isso ocorre porque não é necessariamente seguro excluir de forma arbitrária todos os táxis que não estão no mesmo polígono do local de partida. Em um polígono adjacente, um táxi ainda pode estar mais próximo em termos de distância real de percurso, mesmo considerando a possibilidade de que precise contornar um rio ou outro obstáculo. O importante é considerar os pontos de entrada entre um polígono e outro. A Figura 9.3 mostra como podemos considerá-los.

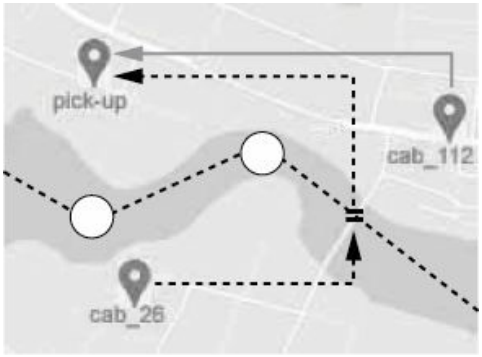


Figura 9.3: Usando pontos de entrada para conectar áreas adjacentes.

A linha pontilhada que atravessa o meio da figura representa o perímetro que divide a área em dois polígonos: o norte do rio e o sul do rio. Na ponte, o sinal de igual marca o ponto de entrada em que os táxis podem percorrer de um polígono para o outro. Para os táxis no polígono limítrofe ao ponto de partida, a distância até o ponto de partida consiste em dois intervalos: o intervalo entre a localização atual do táxi e o ponto de entrada, e o intervalo entre o ponto de entrada e o ponto de partida do local.

Para encontrar o táxi mais próximo, é necessário identificar em qual polígono cada táxi está, pois, assim, podemos decidir como calcular a distância desse táxi até o local de partida: a distância em linha reta, caso o táxi esteja no mesmo polígono do local de partida ou a distância até o ponto de entrada, caso esteja em um polígono adjacente. Abaixo, fazemos esse cálculo somente para `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)]
1 poly = Polygon(coords)
2 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):
  3 dist = distance((pick_up.x, pick_up.y), (cab_26.x,cab_26.y)).m
  else:
  4 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))
```

O script usa as duas bibliotecas, Shapely e geopy. Primeiro, definimos um objeto `Polygon` da Shapely, incluindo o local de partida, como antes `1`. Além disso, definimos objetos `Point` para o táxi, para o local de partida e para o ponto de entrada `2`. Depois, calculamos a distância em metros com a ajuda da função `distance()` da geopy. Se o táxi estiver dentro do polígono, encontramos a distância diretamente entre o táxi e o local de partida `3`. Caso contrário, calculamos primeiro a distância entre o táxi e o ponto de entrada e, em seguida, a distância entre o ponto de entrada e o local de partida, somando-as para obter a distância total `4`. Vejamos o resultado:

```
1544
```

EXERCÍCIO #15: MELHORANDO AINDA MAIS O ALGORITMO DO PONTO DE PARTIDA

No script que acabamos de analisar, processamos os dados geográficos relacionados a um único táxi para determinar a distância entre esse táxi e o local de partida de um passageiro. Modifique o script para que possa determinar as distâncias entre o local de partida e cada um dos diversos táxis. Será necessário agrupar os pontos que representam os táxis em uma lista (não se esqueça, táxi no código é cab) e, em seguida, processar essa lista em um loop, usando a instrução `if/else` do script anterior como o corpo do loop. Depois, identifique a táxi mais próximo do local de partida.

Combinando dados espaciais e não espaciais

Até agora, neste capítulo, trabalhamos exclusivamente com dados espaciais,

mas é importante perceber que as análises espaciais geralmente precisam considerar dados não espaciais também. Por exemplo, de que adianta saber que uma loja está a 16 km de sua localização atual se você não tem certeza de que o item que quer está disponível? Ou, retomando nosso exemplo de táxi, de que adianta determinar o táxi mais próximo de um local de partida se você não sabe se esse táxi está disponível ou atendendo a outra pessoa? Nesta seção, examinaremos o cálculo de dados não espaciais como parte da análise espacial.

Derivando atributos não espaciais

As informações sobre a disponibilidade atual de táxis podem ser derivadas de um conjunto de dados contendo solicitações de corrida. Depois que uma corrida é atribuída a um táxi, essas informações podem ser inseridas em uma estrutura de dados `orders`, em que as solicitações são listadas como abertas (em processo) ou como encerradas (concluídas). Segundo esse esquema, identificar apenas as solicitações em aberto informaria quais táxis não estão disponíveis para atender a uma solicitação nova. Vejamos como implementar essa lógica no 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)
```

Neste exemplo, a lista de tuplas `orders` pode ser derivada de um conjunto de dados mais completo, como uma coleção de todas as solicitações em aberto nas últimas duas horas, incluindo informações adicionais sobre cada uma delas (local de partida, local de destino, hora de início, hora de término e

assim por diante). Aqui, simplificando, o conjunto de dados já foi reduzido apenas aos campos necessários para a tarefa atual. Convertemos a lista em um DataFrame e a filtramos para incluir apenas as solicitações cujo status está `open`. Por último, convertemos o DataFrame em uma lista contendo apenas os valores da coluna `cab`. Vejamos a lista de táxis indisponíveis:

```
['cab_14', 'cab_79', 'cab_104', 'cab_45']
```

Com essa lista, precisamos verificar os outros táxis e determinar qual é o mais próximo do local de partida. Anexamos o seguinte código ao script anterior:

```
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]))
```

Para exemplificar, definimos manualmente as coordenadas geográficas do local de partida e todos os táxis como tuplas, e enviamos as coordenadas dos táxis para um dicionário, em que as chaves são os nomes dos táxis. Em seguida, iteramos com um loop sobre o dicionário e, para cada táxi que não esteja em `unavailable_list`, usamos a geopy para calcular a distância entre o táxi e o local de partida. Por fim, exibimos toda a lista de táxis disponíveis com suas distâncias até o local de partida, bem como apenas o táxi mais próximo, gerando a seguinte saída:

```
[('cab_26', 2165), ('cab_112', 2861)]
('cab_26', 2165)
```

Nesse caso, `cab_26` é o táxi disponível mais próximo.

EXERCÍCIO #16: FILTRANDO DADOS COM UMA LIST COMPREHENSION

Na seção anterior, filtramos a lista `orders` em uma lista de táxis indisponíveis, convertendo primeiro `orders` em um DataFrame. Agora tente gerar a lista `unavailable_list` sem o pandas, usando uma list comprehension. Com essa abordagem, você pode obter a lista de táxis atribuídos às solicitações atualmente em aberto com uma única linha de código:

```
unavailable_list = [x[2] for x in orders if x[1] == 'open']
```

Após essa substituição, você não precisará alterar mais nada no restante do script.

Join de conjuntos de dados espaciais e não espaciais

No exemplo anterior, mantivemos os dados espaciais (a localização de cada táxi) e os dados não espaciais (quais táxis estavam disponíveis) em estruturas de dados separadas. Às vezes, no entanto, talvez seja vantajoso combinar dados espaciais e não espaciais na mesma estrutura.

Considere que talvez um táxi precise satisfazer algumas outras condições além da disponibilidade para ser atribuído a uma solicitação. Por exemplo, um cliente pode precisar de um táxi com uma cadeira de segurança infantil. Para encontrar o táxi certo, precisaremos contar com um conjunto de dados que inclua informações não espaciais sobre os táxis, bem como a distância de cada táxi até o local de partida. Para o primeiro, podemos usar um conjunto de dados que contém apenas duas colunas: o nome do táxi e a cadeira infantil. Vejamos como criá-lo:

```
cabs_list = [
  ('cab_14',1),
  ('cab_79',0),
  ('cab_104',0),
  ('cab_45',1),
  ('cab_26',0),
  ('cab_112',1)
]
```

Os táxis com um `1` na segunda coluna têm cadeira infantil. Em seguida, convertemos a lista em um DataFrame. Criamos também um segundo DataFrame a partir de `dist_list`, a lista de táxis disponíveis e suas distâncias até o local de partida gerado na seção anterior:

```
df_cabs = pd.DataFrame(cabs_list, columns =['cab', 'seat'])
```

```
df_dist = pd.DataFrame(dist_list, columns =['cab', 'dist'])
```

Agora, fazemos o merge desses DataFrames com base na coluna `cab`:

```
df = pd.merge(df_cabs, df_dist, on='cab', how='inner')
```

Recorremos a um inner join. Ou seja, somente os táxis incluídos em `df_cabs` e `df_dist` fazem parte do novo DataFrame. Na prática, como o `df_dist` contém apenas os táxis disponíveis no momento, isso exclui táxis indisponíveis do conjunto de resultados. Agora, o DataFrame mesclado inclui dados espaciais (a distância de cada táxi até o local de partida) e dados não espaciais (se cada táxi tem ou não uma cadeira infantil):

```
        cab  seat  dist
0   cab_26     0  2165
1  cab_112     1  2861
```

Convertemos o DataFrame em uma lista de tuplas, que filtramos, deixando apenas as linhas em que o campo `seat` está definido como `1`:

```
result_list = list(df.itertuples(index=False,name=None))
result_list = [x for x in result_list if x[1] == 1]
```

Recorremos ao método `itertuples()` do DataFrame para converter cada linha em uma tupla e, em seguida, envolvemos as tuplas em um wrapper para uma lista com a função `list()`.

A etapa final é determinar a linha com o menor valor no campo distância, identificada pelo índice `2`:

```
print(min(result_list, key=lambda x: x[2]))
```

Vejamos o resultado:

```
('cab_112', 1, 2861)
```

Compare isso com o resultado mostrado no final da seção anterior. Como se pode ver, a necessidade de uma cadeira infantil nos levou a escolher um táxi diferente.

Recapitulando

Neste capítulo, partindo do exemplo cotidiano de um serviço de táxi, vimos como podemos realizar análises de dados espaciais. Começamos analisando um exemplo de transformar um endereço legível por humanos em coordenadas geográficas com a Geocoding API do Google e com a biblioteca

do Python googlemaps. Em seguida, aprendemos a usar um bot do Telegram para coletar dados geográficos de smartphones. Depois, usamos as bibliotecas geopy e Shapely para realizar operações geoespaciais fundamentais, como calcular a distância entre os pontos e estabelecer se os pontos estão dentro de determinada área. Com a ajuda dessas bibliotecas, das estruturas built-in de dados do Python e dos DataFrames do pandas, desenvolvemos um app para identificar o melhor táxi para determinado ponto de partida com base em variados critérios espaciais e não espaciais.

CAPÍTULO 10

Analisando dados de séries temporais

Dados de séries temporais, ou *dados de timestamp*, são um conjunto de pontos de dados indexados em ordem cronológica. Exemplos comuns incluem índices econômicos, registros meteorológicos e indicadores de saúde do paciente, todos registrados ao longo do tempo. Neste capítulo, veremos técnicas para analisar dados de séries temporais e extrair deles estatísticas significativas com a biblioteca pandas. Focaremos a análise de dados do mercado de ações, porém as mesmas técnicas podem ser usadas em todos os tipos de dados de séries temporais.

Série temporal regular vs. irregular

É possível criar uma série temporal para qualquer variável que mude ao longo do tempo, e essas mudanças podem ser registradas em intervalos de tempo regulares ou irregulares. Intervalos regulares são mais comuns. Na área de finanças, por exemplo, é normal que séries temporais acompanhem o preço de uma ação de um dia para o outro, conforme mostrado a seguir:

```
Date         Closing Price
-----------  -------------
16-FEB-2022  10.26
17-FEB-2022  10.34
18-FEB-2022  10.99
```

Como podemos observar, nessa série temporal, a coluna `Date` contém timestamps, organizados em ordem cronológica, para uma sequência de dias consecutivos. Os pontos de dados correspondentes, muitas vezes chamados de *observações*, são apresentados na coluna `Closing Price`. Consideram-se as séries temporais desse tipo como *regulares* ou como *contínuas*, visto que as

observações são feitas continuamente ao longo do tempo em intervalos regulares.

Outro exemplo de série temporal regular seria uma coleção de coordenadas de latitude e de longitude para um veículo, registradas a cada minuto, como mostrado a seguir:

```
Time      Coordinates
-------   ---------------
20:43:00  37.801618, -122.374308
20:44:00  37.796599, -122.379432
20:45:00  37.788443, -122.388526
```

Aqui, os timestamps são horários, não datas, mas ainda estão em ordem cronológica, minuto a minuto.

Ao contrário das séries temporais regulares, *séries temporais irregulares* são usadas para registrar sequências de eventos à medida que ocorrem ou estão planejadas para ocorrer, mesmo não sendo em intervalos regulares. Vejamos um exemplo simples da agenda de uma conferência:

```
Time      Event
-------   ---------------
8:00 AM   Registration
9:00 AM   Morning Sessions
12:10 PM  Lunch
12:30 PM  Afternoon Sessions
```

Os timestamps dessa série de pontos de dados são distribuídos irregularmente, com base no período de tempo que cada evento deve levar.

Via de regra, séries temporais irregulares são usadas em aplicações em que a origem dos dados é imprevisível. Para desenvolvedores de software, uma típica série temporal irregular seria um registro de erros identificados durante a execução de um servidor ou de uma aplicação. É difícil predizer a ocorrência de erros, pois quase certamente não ocorrerão em intervalos regulares. Outro exemplo seria um aplicativo que rastreia o consumo de energia elétrica e que pode usar uma série temporal irregular para registrar anomalias, como intermitências e falhas aleatoriamente ocorridas.

O que as séries temporais regulares e irregulares têm em comum é que os pontos de dados são listados em ordem cronológica. Na realidade, a análise de séries temporais depende dessa característica fundamental. A ordem

cronológica rigorosa possibilita comparar de forma consistente eventos ou valores dentro de uma série temporal, identificando estatísticas e tendências importantes.

No caso de dados do mercado de ações, por exemplo, a ordem cronológica possibilita acompanhar o desempenho das ações ao longo do tempo. No caso de coordenadas geográficas minuto a minuto de um veículo, é possível usar pares adjacentes de coordenadas a fim de calcular a distância percorrida a cada minuto e, em seguida, utilizar essa distância para comparar a velocidade média do veículo a cada minuto. Ao mesmo tempo, a ordem cronológica da agenda de conferência possibilita que identifiquemos de imediato a duração esperada de cada evento.

Em alguns casos, os timestamps podem não ser tão necessários para analisar as séries temporais; o importante mesmo é que os registros da série estejam em ordem cronológica. Vejamos as seguintes séries temporais irregulares de duas mensagens de erro consecutivas que nosso script pode retornar ao tentar se conectar a um banco de dados MySQL com a senha errada:

```
_mysql_connector.MySQLInterfaceError: Access denied for user
'root'@'localhost' (using password: YES)
NameError: name 'cursor' is not defined
```

A segunda mensagem de erro informa que a variável chamada `cursor` não foi definida. No entanto, só conseguimos entender a raiz do problema se verificarmos a mensagem anterior de erro: como a senha estava incorreta, nenhuma conexão poderia ser estabelecida com o banco de dados, portanto nenhum objeto `cursor` poderia ser criado.

Ainda que para os programadores analisar uma sequência de mensagens de erros seja comum, em geral, isso é feito manualmente, sem recorrer à programação. No resto deste capítulo, focaremos as séries temporais com pontos de dados numéricos, visto que podem ser facilmente analisadas com scripts Python. Em particular, veremos como extrair informações significativas de séries temporais regulares contendo dados do mercado de ações.

Técnicas comuns de análise de séries

temporais

Suponha que você queira analisar uma série temporal de preços diários de fechamento de ações para determinada ação durante determinado período de tempo. Nesta seção, aprenderemos algumas técnicas comuns para usar em análises, mas, primeiro, precisaremos dos dados do mercado de ações.

Conforme vimos nos capítulos 3 e 5, podemos obter dados do mercado de ações em um script Python com a biblioteca yfinance. Aqui, por exemplo, coletamos os dados de ações para o ticker TSLA (Tesla, Inc.) dos últimos cinco dias de negociação:

```
import yfinance as yf
ticker = 'TSLA'
tkr = yf.Ticker(ticker)
df = tkr.history(period='5d')
```

O resultado assume a forma de um DataFrame do pandas e será mais ou menos assim (as datas e os dados retornados sofrerão variação):

```
                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
```

Como visto, o DataFrame é indexado por data. Ou seja, os dados são uma série temporal adequada e cronologicamente organizada. Existem colunas para o preço de abertura e de fechamento, bem como para a alta e baixa dos preços diários. Nesse ínterim, a coluna `Volume` mostra o número total de ações negociadas naquele dia, e as duas colunas mais à direita fornecem detalhes sobre os dividendos e splits que a empresa concedeu aos seus acionistas.

Provavelmente, não precisaremos de todas essas colunas em nossa análise. Na verdade, por ora, precisamos apenas da coluna `Close`. A seguir, a exibimos como uma Series do pandas:

```
print(df['Close'])
```

Vejamos como fica mais ou menos a Series:

```
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
```

Agora estamos prontos para começar nossa análise de séries temporais. Focaremos duas técnicas comuns: calcular as mudanças percentuais ao longo do tempo e efetuar cálculos agregados dentro de uma janela rolante de tempo (rolling time window). Veremos como essas técnicas podem trabalhar juntas para revelar tendências nos dados.

Calculando mudanças percentuais

Talvez a técnica de análise de séries temporais mais comum seja acompanhar o quanto os dados observados mudam ao longo do tempo. No caso de dados de ações, isso pode envolver o cálculo da mudança percentual de um valor de ação durante determinado intervalo de tempo. Desse modo, é possível quantificar o desempenho das ações e desenvolver uma estratégia de investimento de curto prazo.

Tecnicamente falando, mudança percentual é a diferença (expressa como porcentagem) entre os valores de dois pontos distintos no tempo. Assim, para calcular essa mudança, precisamos mudar os pontos de dados no tempo. Ou seja, deslocamos o ponto de dados mais antigo no tempo para que se alinhe com o ponto de dados mais recente; em seguida, podemos comparar os pontos de dados e calcular a mudança percentual. Em ciência de dados, isso também é conhecido como shifting.

Quando uma série temporal é implementada como uma Series ou como um DataFrame do pandas, podemos usar o método `shift()` para mudar pontos de dados no tempo pelo número desejado de períodos. Continuando com o nosso exemplo de ticker TSLA, podemos querer saber quanto o preço de fechamento das ações mudou durante um período de dois dias. Nesse caso, usaríamos `shift(2)` a fim de obter o preço de fechamento de dois dias antes, de acordo com o preço de fechamento para determinado dia. Para se ter uma ideia de como o shifting funciona, aqui, deslocamos a coluna `Close` dois dias à frente, salvamos o resultado como `2DaysShift` e o concatenamos com a coluna `Close` original:

```
print(pd.concat([df['Close'], df['Close'].shift(2)], axis=1, keys= ['Close',
'2DaysShift']))
```

A saída deve ser mais ou menos assim:

```
             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
```

Como visto, os valores na coluna `Close` são refletidos na coluna `2DaysShift`, compensados por dois dias. Os dois primeiros valores na `2DaysShift` são `NAN` porque não temos os preços dois dias antes nos dois primeiros dias da série temporal.

Para encontrar a mudança percentual entre o preço de dois dias antes e o preço de um determinado dia, podemos pegar a diferença entre o valor de determinado dia e o valor anterior e dividi-lo pelo valor anterior:

```
(df['Close'] - df['Close'].shift(2))/ df['Close'].shift(2)
```

Na análise financeira, no entanto, é comum dividir o valor novo pelo valor antigo e obter o logaritmo natural do resultado. Esse cálculo fornece uma aproximação quase exata da mudança percentual quando dentro do intervalo +/– 5%, e permanece muito perto de +/– 20%. A seguir, calculamos a diferença percentual de dois dias usando o logaritmo natural e armazenamos o resultado como uma coluna nova chamada `2daysRise` no DataFrame `df`:

```
import numpy as np
df['2daysRise'] = np.log(df['Close'] / df['Close'].shift(2))
```

Obtemos o preço de fechamento de um dia e o dividimos pelo preço de fechamento dois dias antes, acessado com `shift(2)`. Em seguida, utilizamos a função `log()` do NumPy para obter o logaritmo natural do resultado. Agora, podemos exibir as colunas `Close` e `2daysRise` do DataFrame:

```
print(df[['Close','2daysRise']])
```

Vejamos a saída dessas séries temporais:

```
             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
```

A coluna `2daysRise` mostra a mudança percentual das ações em comparação com dois dias antes. Mais uma vez, os dois primeiros valores na coluna são `NaN` porque não temos os preços de dois dias antes para os dois primeiros dias da série temporal.

Cálculos de janela rolante

Outra técnica comum de análise de séries temporais é comparar cada valor com o valor médio ao longo de *n* períodos. Isso é chamado de *cálculo de janela rolante*: criamos uma janela de tempo de um tamanho fixo e efetuamos o cálculo agregado dos valores dentro dessa janela de tempo à medida que se desloca ou *desliza* pela série temporal. No caso de dados de ações, podemos recorrer ao cálculo de janela rolante para encontrar o preço médio de fechamento dos dois dias anteriores e comparar o preço de fechamento do dia atual com essa média. Isso nos daria uma noção da estabilidade do preço das ações ao longo do tempo.

Todo objeto do pandas tem um método `rolling()` para analisar uma janela rolante de valores. A seguir, o usamos com `shift()` e `mean()` para encontrar o preço médio das ações da Tesla nos dois dias anteriores:

```
df['2daysAvg'] = df['Close'].shift(1).rolling(2).mean()
print(df[['Close', '2daysAvg']])
```

Na primeira linha, usamos `shift(1)` para deslocar os pontos de dados da série em um dia. Fazemos isso porque queremos excluir o preço do dia atual ao calcular uma média que será comparada a ele. Em seguida, criamos a janela rolante com `rolling(2)`, indicando que queremos desenhar duas linhas consecutivas ao efetuar os cálculos. Por último, invocamos o método `mean()` a fim de calcular uma média para cada par de linhas consecutivas abrangidas pela janela rolante. Armazenamos os resultados em uma coluna nova chamada `2daysAvg`, que exibimos com a coluna `Close`. Vejamos o DataFrame resultante:

```
             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
```

Os preços na coluna `2daysAvg` são as médias dos dois dias de negociação anteriores. Por exemplo, o valor atribuído a `2022-01-12` é a média dos preços dos dias `2022-01-10` e `2022-01-11`.

Calculando a mudança percentual de uma média móvel

Dada uma média móvel dos preços de fechamento dos dois dias anteriores, o próximo passo lógico é calcular a mudança percentual entre o preço de cada dia e sua média móvel associada. Aqui, efetuamos esse cálculo, usando novamente o logaritmo natural para aproximar a mudança percentual:

```
df['2daysAvgRise'] = np.log(df['Close'] / df['2daysAvg'])
print(df[['Close','2daysRise','2daysAvgRise']])
```

Armazenamos os resultados em uma nova coluna chamada `2daysAvgRise`. Em seguida, exibimos as colunas `Close`, `2daysRise` e `2daysAvgRise` juntas. Vejamos como fica a saída:

```
             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
```

Para essa série temporal específica, ambas as métricas recém-criadas, `2daysRise` e `2daysAvgRise`, mostram valores negativos e positivos. Isso indica que o preço de fechamento das ações estava volátil durante todo o período de observação. Claro que os seus resultados podem revelar uma tendência diferente.

Séries temporais multivariadas

Uma *série temporal multivariada* é uma série temporal com mais de uma variável que muda ao longo do tempo. Por exemplo, ao obtermos os dados de ações da Tesla com a biblioteca yfinance, esses dados vieram como uma série

temporal multivariada, pois continham não somente o preço de fechamento das ações, como também o preço de abertura, preços máximos e mínimos e diversos outros pontos de dados para cada dia. Nesse caso, a série temporal multivariada encontrou múltiplas features do mesmo objeto, uma só ação. Outras séries temporais multivariadas podem encontrar a mesma feature de diversos objetos, como os preços de fechamento de múltiplas ações no mesmo período de tempo.

No script a seguir, criamos esse segundo tipo de série temporal multivariada, obtendo cinco dias de dados de ações para múltiplos tickers:

```
  import pandas as pd
  import yfinance as yf
 1 stocks = pd.DataFrame()
 2 tickers = ['MSFT','TSLA','GM','AAPL','ORCL','AMZN']
 3 for ticker in tickers:
    tkr = yf.Ticker(ticker)
    hist = tkr.history(period='5d')
  4 hist = pd.DataFrame(hist[['Close']].rename(columns={'Close': ticker}))
  5 if stocks.empty:
    6 stocks = hist
    else:
    7 stocks = stocks.join(hist)
```

Primeiro, definimos o DataFrame `stocks` `1`, em que concentraremos os preços de fechamento para múltiplos tickers. Em seguida, definimos uma lista de tickers `2` e iteramos com um loop sobre lista `3`, usando a biblioteca yfinance para obter os últimos cinco dias de dados para cada ticker. Dentro do loop, limitamos o DataFrame `hist` retornado pela yfinance a um DataFrame de coluna única, contendo os preços de fechamento das ações em questão com os timestamps correspondentes como índice `4`. Depois, verificamos se o DataFrame `stocks` está vazio `5`. Se estiver, como é a primeira vez dentro do loop, inicializamos o DataFrame `stocks` com o DataFrame `hist` `6`. Nas iterações subsequentes, como `stock` não estará vazio, fazemos o join do atual DataFrame `hist` com o DataFrame `stocks`, adicionando os preços de fechamento de outro ticker ao conjunto de dados `7`. A estrutura `if/else` é necessária, pois não podemos executar uma operação de join em um DataFrame vazio.

Vejamos como fica o DataFrame `stocks` resultante:

```
              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
```

Temos uma série temporal multivariada, com diferentes colunas mostrando os preços de fechamento de diferentes ações, ao longo do mesmo período de tempo.

Processando séries temporais multivariadas

Processar séries temporais multivariadas é semelhante a manipular séries temporais com uma única variável, exceto pelo fato de que precisamos lidar com diversas variáveis dentro de cada linha. Por isso, os cálculos geralmente ocorrem dentro de um loop que itera sobre as colunas das séries. Por exemplo, suponha que desejamos filtrar o DataFrame `stocks`, eliminando os tickers cujos preços caíram mais do que algum threshold (digamos, 3%), abaixo do preço do dia anterior e, pelo menos, uma vez no período determinado. Aqui, iteramos sobre as colunas e analisamos os dados para cada ticker visando determinar quais ações devem ser mantidas no DataFrame:

```
1 stocks_to_keep = []
2 for i in stocks.columns:
    if stocks[stocks[i]/stocks[i].shift(1)< .97].empty:
      stocks_to_keep.append(i)
  print(stocks_to_keep)
```

Primeiro, criamos uma lista para concentrar os nomes das colunas que queremos manter `1`. Em seguida, iteramos sobre as colunas do DataFrame `stocks` `2`, determinando se cada uma contém valores que sejam mais de 3% inferiores ao valor na linha anterior. Especificamente, usamos o operador `[]` para filtrar o DataFrame e o método `shift()` para comparar o preço de fechamento de cada dia com o do dia anterior. Se uma coluna não contiver nenhum valor que atenda à condição de filtragem (ou seja, se a coluna filtrada estiver vazia), basta anexar o nome da coluna à lista `stocks_to_keep`.

Considerando o DataFrame `stocks` mostrado anteriormente, vejamos como

fica a lista `stocks_to_keep` resultante:

```
['GM', 'AAPL', 'ORCL', 'AMZN']
```

Conforme podemos observar, TSLA e MSFT não estão na lista porque continham um ou mais valores que caíram mais de 3% abaixo do preço de fechamento do dia anterior. Claro que os resultados podem variar; você pode acabar com uma lista vazia ou uma lista que inclua todos os tickers. Nesses casos, tente experimentar o threshold de filtragem. Se a lista estiver vazia, tente diminuir o threshold de 0,97 para 0,96 ou inferior. Por outro lado, se a lista incluir todos os tickers, tente aumentar o threshold.

A seguir, exibimos o DataFrame `stocks` para que inclua somente as colunas da lista `stocks_to_keep`:

```
print(stocks[stocks_to_keep])
```

No meu caso, a saída ficou assim:

```
                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
```

Como esperado, as colunas `TSLA` e `MSFT` foram filtradas porque contêm um ou mais valores que excedem o threshold de volatilidade de 3%.

Analisando dependências entre variáveis

Ao analisar séries temporais multivariadas, uma tarefa habitual é identificar relacionamentos entre diferentes variáveis no conjunto de dados. Esses relacionamentos podem existir ou não. Por exemplo, é provável que haja algum grau de dependência entre os preços de abertura e de fechamento de uma ação, pois, em determinado dia, o preço de fechamento raramente diverge do preço de abertura em mais do que alguns por cento. Por outro lado, podemos não encontrar dependências entre os preços de fechamento de duas ações de diferentes setores econômicos.

Nesta seção, veremos algumas técnicas para verificar a existência de relacionamentos entre variáveis de séries temporais. E, para demonstrar,

examinaremos se existe dependência entre a mudança no preço de uma ação e seu volume de vendas. Para começar, executaremos o seguinte script a fim de obter um mês de dados de ações para nossa análise:

```
import yfinance as yf
import numpy as np
ticker = 'TSLA'
tkr = yf.Ticker(ticker)
df = tkr.history(period='1mo')
```

Conforme já observamos, a yfinance gera uma série temporal multivariada na forma de um DataFrame com muitas colunas. Para este exemplo, precisamos somente de duas delas: `Close` e `Volume`. Aqui, reduzimos o DataFrame devidamente e mudamos o nome da coluna `Close` para `Price`:

```
df = df[['Close','Volume']].rename(columns={'Close': 'Price'})
```

Para identificar se existe relacionamento entre as colunas `Price` e `Volume`, devemos calcular a mudança percentual diária em cada coluna. A seguir, calculamos a mudança percentual diária na coluna `Price` com `shift(1)` e com a função `log()` do NumPy, conforme visto anteriormente, e armazenamos o resultado em uma coluna nova chamada `priceRise`:

```
df['priceRise'] = np.log(df['Price'] / df['Price'].shift(1))
```

Utilizamos a mesma técnica para criar a coluna `volumeRise`, que mostra a mudança percentual no volume em comparação com o dia anterior:

```
df['volumeRise'] = np.log(df['Volume'] / df['Volume'].shift(1))
```

Como observado antes, o logaritmo natural fornece uma aproximação da mudança percentual dentro de um intervalo de +/– 20%. Na coluna `volumeRise`, embora alguns valores possam exceder esse intervalo, ainda podemos usar `log()` aqui, pois um alto grau de acurácia não é necessário neste exemplo; em geral, a análise do mercado de ações é mais focada em predizer tendências do que em identificar valores exatos.

Agora, vamos exibir e ver como fica o DataFrame `df`:

```
              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
```

Caso exista uma dependência entre preço e volume, espera-se que mudanças acima da média no preço (ou seja, maior volatilidade) se correlacionem com mudanças acima da média no volume. Para verificar se esse é o caso, devemos definir um threshold para a coluna `priceRise` e visualizar apenas as linhas em que a mudança percentual do preço está acima desse threshold. Por exemplo, nessa saída específica, se analisarmos os valores na coluna `priceRise`, podemos escolher um threshold de 5%. Talvez outro conjunto de dados sugira outro threshold, 3% ou 7%. A ideia é que apenas alguns registros excedam o threshold; portanto, como regra geral, quanto mais volátil a ação, maior deve ser o threshold.

Aqui, exibimos somente as linhas em que `priceRise` excede o threshold:

```
print(df[abs(df['priceRise']) > .05])
```

Recorremos à função `abs()` para obter o valor absoluto de uma mudança percentual de modo que, por exemplo, `0.06` e `-0.06` satisfaçam a condição especificada. Considerando os dados de amostra mostrados anteriormente, acabamos com o seguinte:

```
              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
```

Em seguida, calculamos a mudança média do volume ao longo de toda a série:

```
print(df['volumeRise'].mean().round(4))
```

Para essa série específica, o resultado é o seguinte:

```
-0.0016
```

Por último, calculamos a mudança média do volume apenas para as linhas com mudanças acima da média no preço. Se o resultado for maior do que a mudança média de volume em toda a série, saberemos que existe uma conexão entre o aumento da volatilidade e o aumento do volume:

```
print(df[abs(df['priceRise']) > .05]['volumeRise'].mean().round(4))
```

Vejamos a saída dessa série em particular:

```
0.2035
```

Como podemos verificar, a mudança de volume médio calculada ao longo da série filtrada é muito maior do que a mudança de volume médio calculada ao longo de toda a série. Isso sugere que pode existir uma correlação positiva entre a volatilidade dos preços e a volatilidade do volume de vendas.

EXERCÍCIO #17: ADICIONANDO MAIS MÉTRICAS PARA ANALISAR DEPENDÊNCIAS

Continuando com o DataFrame da seção anterior, é possível perceber que, embora exista uma conexão provável entre as coluna `priceRise` e `volumeRise`, isso não fica completamente óbvio. Por exemplo, em 2022-12-16, o preço caiu aproximadamente 5% e o volume de vendas subiu 10%, mas quase a mesma queda de preço em 2022-01-05 é acompanhada de uma queda de 22% no volume de vendas.

Para entender essas divergências, você precisa analisar mais métricas que possam se correlacionar com os volumes de vendas. Por exemplo, seria interessante um cálculo de janela rolante que mostrasse o volume total de vendas nos dois dias anteriores. A suposição é que, se o volume de vendas de um dia exceder (ou for quase igual a) a soma do volume dos últimos dois dias, provavelmente não devemos esperar mais crescimento no volume de vendas no dia seguinte. Ou seja, o cálculo de janela rolante pode ajudá-lo a predizer tendências no volume de vendas.

Para testar se essa suposição é verdadeira, primeiro adicione uma coluna `volumeSum` ao DataFrame para acomodar o cálculo de janela rolante.

```
df['volumeSum'] = df['Volume'].shift(1).rolling(2).sum().fillna(0).astype(int)
```

Desloque os pontos de dados por um dia para excluir o volume do dia atual da soma que

está sendo calculada. Em seguida, você cria uma janela rolante de dois dias e usa `sum()` para calcular o volume total de vendas dentro dessa janela. Na coluna nova, por padrão, os valores são floats, mas os converta para inteiros com `astype()`. Antes de fazer essa conversão, é necessário substituir os valores de `NaN` por zeros, basta recorrer ao método `fillna()` .

Agora, com a métrica `volumeSum` em mãos, talvez você queira analisar mais uma vez os dias mais voláteis de sua série:

```
print(df[abs(df['priceRise']) > .05].replace(0, np.nan).dropna())
```

Para os dados de amostra usados aqui, vejamos novamente os dias em que o preço mudou em mais de 5% em comparação com o dia anterior, agora com a coluna `volumeSum` adicionada:

	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

Os números na coluna `volumeSum` sugerem que um menor volume total de vendas nos dois dias anteriores está correlacionado com um maior potencial de crescimento ou queda no volume de vendas hoje e vice-versa. Veja, por exemplo, os números de 2022-01-03: o valor `volumeRise` desse dia é o mais alto no conjunto de resultados, enquanto o valor `volumeSum` é o mais baixo. Na verdade, o volume de vendas desse dia é quase igual à soma do volume de vendas dos dois dias anteriores (2021-12-30 e 2021-12-31), apresentando, assim, um crescimento significativo.

No entanto, lembre-se de que a suposição original era que, em dias assim, em que o volume de vendas excede (ou corresponde aproximadamente a) a soma do volume dos últimos dois dias, não devemos esperar mais crescimento em volume no dia seguinte. Para validar isso, você pode adicionar uma coluna mostrando o volume de vendas do dia seguinte:

```
df['nextVolume'] = df['Volume'].shift(-1).fillna(0).astype(int)
print(df[abs(df['priceRise']) > .05].replace(0, np.nan).dropna())
```

Você cria a coluna `nextVolume` deslocando `Volume` em `-1` unidade. Ou seja, desloca o volume do dia seguinte um passo atrás no tempo para se alinhar com o dia atual. Vejamos a saída:

	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

Como se pode observar, a suposição é verdadeira para 2022-01-03: `nextVolume` é menor que `Volume`. No entanto, são necessárias mais métricas para que sua análise tenha um

grau de acurácia maior. Tente adicionar outra métrica que some os valores de `priceRise` para os dois dias anteriores. Se mostrar um valor positivo, significa que os preços geralmente estavam em tendência ascendente nos últimos dois dias. Um valor negativo sinaliza preços em queda. Use essa métrica nova com as métricas `priceRise` e `volumeSum` já existentes para descobrir como, juntas, podem afetar os valores na coluna `volumeRise`.

Recapitulando

Conforme aprendido neste capítulo, uma série temporal é um conjunto organizado de dados em ordem cronológica, em que uma ou mais variáveis mudam ao longo do tempo. Com dados do mercado de ações como exemplo, examinamos algumas técnicas com o pandas para analisar dados de séries temporais a fim de obter estatísticas úteis deles. Aprendemos a mudar pontos de dados em uma série temporal para calcular as mudanças ao longo do tempo. Aprendemos também a executar cálculos de janela rolante ou agregações dentro de um intervalo de tempo de tamanho fixo que se move por toda a série. Juntas, essas técnicas o ajudam a avaliar as tendências nos dados. Por último, analisamos métodos para identificar dependências entre diferentes variáveis em uma série temporal multivariada.

CAPÍTULO 11

Obtendo insights a partir de dados

Todos os dias, as empresas geram quantidades colossais de dados na forma de fatos brutos, valores e eventos. Mas o que todos esses dados realmente nos dizem? Para extrair conhecimento e obter insights a partir de dados, é necessário transformá-los, analisá-los e visualizá-los. Dito de outro modo, precisamos transformar os dados brutos em informações relevantes que possamos usar para tomar decisões, responder a perguntas e solucionar problemas.

Imagine o caso de um supermercado que coleta grandes volumes de dados de transações de clientes. Os analistas do supermercado podem estar interessados em estudar esses dados para obter informações sobre as preferências de compra dos clientes. Em especial, talvez queiram realizar a análise *market basket* (análise de cesta de compras, em tradução livre), técnica de mineração de dados que analisa transações e identifica itens que são comumente comprados juntos. Com esse conhecimento, o supermercado poderia tomar decisões de negócios mais embasadas – por exemplo, sobre o layout dos itens na loja ou sobre como agrupar itens com descontos.

Neste capítulo, exploraremos detalhadamente esses exemplos, examinando como obter insights a partir de dados de transações por meio da análise market basket com Python. Aprenderemos a usar a biblioteca mlxtend e o algoritmo Apriori para identificar itens que são comumente comprados juntos e veremos como esse conhecimento pode ser empregado para tomar decisões inteligentes de negócios.

Ainda que a identificação das preferências do comprador seja o foco deste capítulo, não é a única aplicação que usa a análise market basket. A técnica

também é usada em setores como telecomunicações, mineração da web ou web mining, bancos e assistência médica. Na mineração da web, por exemplo, a análise market basket pode determinar para onde o usuário de uma página web provavelmente irá e gerar associações de páginas frequentemente visitadas juntas.

Regras de associação

A análise market basket mensura a força dos relacionamentos entre objetos com base em sua co-ocorrência nas mesmas transações. Os relacionamentos entre objetos são representados como *regras de associação*, denotadas da seguinte forma:

```
X -> Y
```

`X` e `Y`, referidos como *antecedente* e *consequente* da regra, respectivamente, representam conjuntos de itens *distintos*, ou grupos de um ou mais itens dos dados de transação sendo extraídos. Por exemplo, uma regra de associação que descreve um relacionamento entre os itens *curd* (coalhada) e *sour cream* (creme azedo) seria denotada assim:

```
curd -> sour cream
```

Nesse caso, `curd` é o antecedente e `sour cream` é o consequente. A regra é fazer uma asserção de que pessoas que compram coalhada também estão propensas a comprar creme azedo.

Por si só, uma regra de associação como essa não informa muita coisa. O segredo para uma análise market basket bem-sucedida é usar os dados da transação para avaliar a força das regras de associação com base em diversas métricas. Para demonstrar, vejamos um exemplo simples. Suponha que temos 100 transações de clientes, 25 das quais contêm curd e 30 das quais contêm sour cream. Das 30 transações que contêm sour cream, 20 também contêm curd. A Tabela 11.1 resume esses números.

Tabela 11.1: Números de transações para curd e sour cream

	Curd	Sour cream	Curd e sour cream	Total
Transações	25	30	20	100

Considerando esses dados de transação, podemos avaliar a força da regra de

associação `curd -> sour` com métricas como suporte, confiança e lift (alavancagem, em tradução livre). Essas métricas nos ajudarão a avaliar se existe ou não uma associação entre curd e sour cream.

Suporte

Suporte é a proporção de transações que incluem um ou mais itens para o número total de transações. Por exemplo, nos dados de transação de amostra, o suporte para curd pode ser calculado da seguinte forma:

```
support(curd) = curd/total = 25/100 = 0.25
```

No contexto de uma regra de associação, o suporte é a proporção de transações que incluem o antecedente e o consequente para o número total de transações. O suporte da regra de associação `curd -> sour` é, portanto:

```
support(curd -> sour cream) = (curd & sour cream)/total = 20/100 = 0.2
```

A métrica suporte fica no intervalo de 0 a 1 e informa em qual porcentagem de tempo um conjunto de itens aparece em uma transação. Nesse caso, podemos verificar que 20% das transações incluíam curd e sour cream. O suporte é simétrico para qualquer regra de associação; ou seja, o suporte para `curd -> sour cream` é o mesmo que o suporte para `sour cream -> curd`.

Confiança

A *confiança* de uma regra de associação é a proporção de transações em que tanto o antecedente quanto o consequente são comprados. Em outras palavras, a confiança calcula qual parcela das transações que contém o antecedente também contém o consequente. A confiança para a regra de associação `curd -> sour cream` pode ser calculada da seguinte forma:

```
confidence(curd -> sour cream) = (curd & sour cream)/curd = 20/25 = 0.8
```

Podemos interpretar essa regra de associação como: se um cliente comprou curd, há 80% de probabilidade de que ele também compre sour cream.

Como o suporte, a confiança está dentro de um intervalo de 0 a 1, mas, ao contrário do suporte, a confiança não é simétrica. Ou seja, a confiança para a regra `curd -> sour cream` pode ser diferente da confiança para a regra `sour cream -> curd`, conforme mostrado a seguir:

```
confidence(sour cream -> curd) = (curd & sour cream)/sour cream = 20/30 = 0.66
```

Nesse caso, obtivemos um valor de confiança mais baixo quando o antecedente e o consequente da regra de associação são invertidos. Isso indica que é menos provável que alguém que compra sour cream também compre curd do que alguém que compra curd também compre sour cream.

Lift

O *lift* avalia a força de uma regra de associação em comparação com a co-ocorrência aleatória dos itens que aparecem na regra. O lift da regra de associação `curd -> sour cream` é a proporção do suporte observado para `curd -> sour cream` com o esperado se curd e sour cream fossem independentes um do outro. Isso pode ser calculado da seguinte forma:

```
lift(sour cream -> curd) = support(curd & sour cream)/(support(curd)*support(sour cream))
                         = 0.2/(0.25*0.3) = 2.66
```

Os valores de lift são simétricos — se substituirmos o antecedente e o consequente, o lift permanece o mesmo. Os valores possíveis para o lift variam de 0 a infinito, e, quanto maior a proporção do lift, mais forte é a associação. Em especial, uma proporção de lift maior que 1 sinaliza que o relacionamento entre o antecedente e o consequente é mais forte do que seria esperado se fossem independentes. Ou seja, os dois itens são comprados juntos com frequência. Uma proporção de lift igual a 1 indica que não há correlação entre o antecedente e o consequente. Já uma proporção de lift menor que 1 sinaliza que há uma correlação negativa entre o antecedente e o consequente. Ou seja, é improvável que os dois itens sejam comprados juntos. Nesse caso, podemos interpretar a proporção do lift de 2,66 como: quando um cliente compra curd, o aumento da expectativa de que também compre sour cream é de 166%.

Algoritmo Apriori

Aprendemos as regras de associação e vimos algumas métricas que avaliam sua força. No entanto, como geramos regras de associação para uma análise market basket? Uma forma é recorrer ao *algoritmo Apriori*, processo automatizado para analisar dados de transações. Em termos gerais, o algoritmo consiste em duas etapas:

1. Identifica todos os *conjuntos de itens frequentes*, ou grupos de um ou mais itens que aparecem em muitas transações, no conjunto de dados. O algoritmo faz isso encontrando todos os itens ou grupos de itens cujo valor de suporte exceda determinado threshold.
2. Gera as regras de associação para esses conjuntos de itens frequentes considerando todas as partições binárias possíveis de cada conjunto de itens (ou seja, todas as divisões do conjunto de itens em um grupo antecedente e em um grupo consequente), calculando um conjunto de métricas de associação para cada partição.

Após as regras de associação serem geradas, podemos avaliá-las com base nas métricas analisadas na seção anterior.

Diversas bibliotecas do Python de terceiros vêm com implementações do algoritmo Apriori. Uma delas é a biblioteca mlxtend, abreviação em inglês de *machine learning extensions* (extensões de aprendizado de máquina). A mlxtend inclui ferramentas para realizar várias tarefas comuns de ciência de dados. Nesta seção, veremos um exemplo de análise market basket usando o algoritmo Apriori da mlxtend. Mas, primeiro, instale a biblioteca mlxtend com o `pip`, da seguinte forma:

```
$ pip install mlxtend
```

NOTA *Para obter mais informações sobre a mlxtend, confira a documentação da biblioteca em* http://rasbt.github.io/mlxtend.

Criando um conjunto de dados de transação

Para realizar nossa análise market basket, precisamos de alguns dados de transação de amostra. Simplificando, podemos usar somente algumas transações, definidas como uma lista de listas, como mostrado a seguir:

```
transactions = [
['curd', 'sour cream'], ['curd', 'orange', 'sour cream'],
['bread', 'cheese', 'butter'], ['bread', 'butter'], ['bread', 'milk'],
['apple', 'orange', 'pear'], ['bread', 'milk', 'eggs'], ['tea', 'lemon'],
['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']
```

```
]
```

Cada lista interna contém o conjunto de itens para uma única transação, e a lista `transacional` contém 20 transações no total. Para manter as proporções quantitativas definidas no exemplo original curd/sour cream, o conjunto de dados contém cinco transações com curd, seis transações com sour cream e quatro que contêm curd e sour cream.

Para executar os dados de transações por meio do algoritmo Apriori da mlxtend, é necessário transformá-los em um *array booleano via codificação one-hot*, estrutura em que cada coluna representa um item que pode ser comprado, cada linha representa uma transação e cada valor é `True` ou `False` (`True` se a transação incluiu esse item específico ou `False` se não incluiu). Aqui, executamos a transformação necessária com o objeto `TransactionEncoder` da mlxtend:

```
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder

encoder = TransactionEncoder()
encoded_array = encoder.fit(transactions).transform(transactions)
df_itemsets = pd.DataFrame(encoded_array, columns=encoder.columns_)
```

Criamos um objeto `TransactionEncoder` `1` e o usamos para transformar a lista de listas `transactions` em uma array booleano via codificação one-hot chamado `encomendará` `2`. Em seguida, convertemos o array em um DataFrame do pandas chamado `df_itemsets` `3`. Vejamos o trecho do código:

```
    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  ...
5    True  False   False   False   False  False  False  ...
6   False   True   False   False   False  False   True  ...
--trecho de código omitido--

[20 rows x 20 columns]
```

O DataFrame tem 20 linhas e 20 colunas, em que as linhas representam as transações e as colunas, os itens. Para confirmar que a lista original de listas inclui 20 transações com base em 20 itens possíveis, usamos o seguinte

código:

```
print('Number of transactions: ', len(transactions))
print('Number of unique items: ', len(set(sum(transactions, []))))
```

Em ambos os casos, devemos obter `20`.

Identificando conjuntos de itens frequentes

Agora que os dados de transações estão em formato utilizável, podemos usar a função `apriori()` da mlxtend a fim de identificar todos os conjuntos de itens frequentes nos dados de transações. Ou seja, todos os itens ou grupos de itens com uma métrica suporte alta o suficiente. Vejamos como:

```
from mlxtend.frequent_patterns import apriori
frequent_itemsets = apriori(df_itemsets, min_support=0.1, use_colnames=True)
```

Importamos a função `apriori()` do módulo `mlxtend.frequent_patterns`. Em seguida, chamamos a função, passando o DataFrame contendo os dados de transações como o primeiro parâmetro. Definimos também o parâmetro `min_support` como `0.1` para retornar os conjuntos de itens com, pelo menos, 10% de suporte. (Lembre-se, a métrica suporte indica em que porcentagem das transações um item ou grupo de itens ocorre.) Definimos `use_colnames` como `True` para identificar as colunas incluídas em cada conjunto de itens por nome (como `curd` ou `sour cream`), e não por número de índice. Como resultado, `apriori()` retorna o seguinte DataFrame:

```
    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)
```

Como observado anteriormente, um conjunto de itens pode consistir em um ou mais itens e, de fato, `apriori()` retornou diversos conjuntos de itens de item único. Em última análise, a mlxtend omitirá esses conjuntos de itens de item único quando definir as regras de associação; no entanto, precisará dos dados de *todos* os conjuntos de itens frequentes (incluindo aqueles com um item) para gerar as regras com sucesso. Ainda assim, por questão de interesse, por ora, talvez você queira visualizar somente os conjuntos de itens com múltiplos itens. Para fazer isso, primeiro adicionamos uma coluna `length` ao DataFrame `frequent_itemsets`, da seguinte forma:

```
frequent_itemsets['length'] = frequent_itemsets['itemsets'].apply(lambda
itemset: len(itemset))
```

Em seguida, use a sintaxe de seleção do pandas para filtrar o DataFrame apenas para as linhas com um campo `length` de `2` ou mais:

```
print(frequent_itemsets[frequent_itemsets['length'] >= 2])
```

Veremos o seguinte resultado, sem nenhum dos conjuntos de itens de item único:

```
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
```

Para reiterar, no entanto, a mlxtend exige informações sobre todos os conjuntos de itens frequentes ao gerar regras de associação. Assim sendo, lembre-se de não remover nenhuma linha do DataFrame `frequent_itemsets` original.

Gerando regras de associação

Identificamos todos os conjuntos de itens que atendem ao threshold de suporte desejado. A segunda etapa do algoritmo Apriori é gerar regras de associação para esses conjuntos de itens. Para isso, recorremos à função `association_rules()` do módulo `frequent_patterns` da mlxtend:

```
from mlxtend.frequent_patterns import association_rules
rules = association_rules(frequent_itemsets, metric="confidence",
min_threshold=0.5)
```

Aqui, invocamos a função `association_rules()`, passando o DataFrame `frequent_itemsets` como primeiro parâmetro. Além disso, escolhemos uma métrica para avaliar as regras e definimos um valor de threshold para essa métrica. Em termos específicos, instruímos que a função só deve retornar essas regras de associação com uma métrica de confiança de 0,5 ou mais. Conforme visto na seção anterior, a função ignorará automaticamente a geração de regras para conjuntos de itens de membro único.

A função `association_rules()` retorna as regras na forma de um DataFrame, em que cada linha representa uma única regra de associação. O DataFrame tem colunas para os antecedentes, consequentes e variadas métricas, incluindo suporte, confiança e lift. A seguir, exibimos uma seleção das colunas:

```
print(rules.iloc[:,0:7])
```

Veremos a seguinte saída:

```
      antecedents    consequents  antecedent sup. consequent
sup.  support  confidence    lift
0          (bread)       (butter)           0.40            0.20     0.20    0.50
2.500000
1         (butter)        (bread)           0.20            0.40     0.20    1.00
2.500000
2         (cheese)        (bread)           0.25            0.40     0.15    0.60
1.500000
3           (milk)        (bread)           0.15            0.40     0.10    0.66
1.666667
4         (butter)       (cheese)           0.20            0.25     0.10    0.50
2.000000
5          (pasta)       (cheese)           0.10            0.25     0.10    1.00
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    0.66
6.666667
9           (eggs)         (milk)           0.10            0.15     0.10    1.00
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]
```

Analisando essas regras, algumas parecem redundantes. Por exemplo, há uma regra `bread -> butter` e uma regra `butter -> bread`. Da mesma forma, existem diversas regras baseadas no conjunto de itens `(bread, cheese, butter)`. Em parte, isso ocorre porque, como observado anteriormente no capítulo, a confiança não é simétrica; se substituirmos o antecedente e o consequente em uma regra, o valor de confiança pode mudar. Além do mais, para um conjunto de itens de três membros, o lift pode mudar dependendo de quais itens fazem parte do antecedente e quais fazem parte do consequente. Assim, `(bread, cheese) -> butter` tem um lift diferente do que `(bread, butter) -> cheese`.

Visualizando regras de associação

Conforme aprendemos no Capítulo 8, a visualização é uma técnica simples, mas poderosa, para analisar dados. No contexto de análise market basket, a visualização fornece uma maneira conveniente de avaliar a força de um conjunto de regras de associação, visualizando as métricas para diferentes pares antecedentes/consequentes. Nesta seção, recorreremos à Matplotlib para visualizar as regras de associação geradas na seção anterior na forma de um heatmap ou mapa de calor.

Um *mapa de calor* é um gráfico em forma de grade em que as células assumem cores para indicar seus valores. Neste exemplo, criaremos um mapa de calor mostrando a métrica de lift das várias regras de associação. Organizaremos todos os antecedentes ao longo do eixo y e os consequentes ao longo do eixo x, e preencheremos a área em que os antecedentes e

consequentes de uma regra se intersectam com uma cor, sinalizando o valor de lift dessa regra. Quanto mais escura a cor, maior o lift.

NOTA *Neste exemplo, estamos visualizando o lift porque é a métrica popular para avaliar regras de associação. No entanto, podemos escolher visualizar uma métrica diferente, como a confiança.*

Para gerar a visualização, primeiro, criamos um DataFrame vazio, dentro do qual copiamos as colunas `antecedents`, `consequents` e `lift` do DataFrame `rules` criado anteriormente:

```
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))
```

Utilizamos funções lambda para converter os valores das colunas `antecedents` e `consequents` do DataFrame `rules` em strings, facilitando usá-las como rótulos na visualização. Originalmente, os valores eram frozensets, versões imutáveis de conjuntos Python. Usamos outra função lambda para arredondar os valores de lift em duas casas decimais.

Em seguida, é necessário transformar o DataFrame `rules_plot` recém-criado em uma matriz, que será utilizada para criar o mapa de calor, com os consequentes dispostos horizontalmente e os antecedentes dispostos verticalmente. Para isso, podemos remodelar `rules_plot`, de modo que os valores exclusivos na coluna `antecedents` formem o índice e os valores exclusivos na coluna `consequents` e se tornem colunas novas, enquanto os valores da coluna `lift` são usados para preencher os valores do DataFrame remodelado. Recorremos ao método `pivot()` do DataFrame `rules_plot`:

```
pivot = rules_plot.pivot(index = 'antecedents', columns = 'consequents', values=
'lift')
```

Especificamos as colunas `antecedents` e `consequents` para formar os eixos do DataFrame `pivot` resultante e desenhar os valores na coluna `lift`. Vejamos como fica o `pivot` se o exibirmos:

```
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
```

Esse DataFrame contém tudo de que precisamos para criar o mapa de calor: os valores do índice (os antecedentes) se tornarão os rótulos do eixo y, os nomes das colunas (os consequentes) se tornarão os rótulos do eixo x e a grade de números e `NaNs` se tornará os valores para o gráfico. (Nesse contexto, um `NaN` sinaliza que nenhuma regra de associação foi gerada para determinado par antecedente/consequente.) A seguir, extraímos esses componentes em variáveis separadas:

```
antecedents = list(pivot.index.values)
consequents = list(pivot.columns)
import numpy as np
pivot = pivot.to_numpy()
```

Agora, temos os rótulos do eixo y na lista `antecedents`, os rótulos do eixo x na lista `consequents` e os valores para o gráfico no array do NumPy `pivot`. Usamos todos esses componentes no script a seguir para criar o mapa de calor com a Matplotlib:

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
1 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)
2 plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
          rotation_mode="anchor")
3 for i in range(len(antecedents)):
   for j in range(len(consequents)):
   4 if not np.isnan(pivot[i, j]):
     5 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()
```

Os pontos-chave da plotagem com a Matplotlib foram abordados no Capítulo 8. Aqui, consideraremos somente as linhas específicas para esse exemplo particular. O método `imshow()` converte os dados do array `pivot` em uma imagem 2D codificada por cores `1`. Com o parâmetro `cmap` do método, especificamos como mapear os valores numéricos do array em cores. A Matplotlib tem uma série de mapeamentos built-in de cores que podemos escolher, incluindo o mapeamento `Reds` usado aqui.

Após criar os rótulos do eixo, utilizamos o método `setp()` para rotacionar os rótulos do eixo x em 45 graus `2`. Isso ajuda a acomodar os rótulos dentro do espaço horizontal alocado. Em seguida, iteramos com um loop sobre os dados na array `pivot` `3` e criamos anotações de texto para cada quadrado no mapa de calor com o método `text()` `5`. Os dois primeiros parâmetros, `j` e `i`, são as coordenadas x e y para o rótulo. O próximo parâmetro, `pivot[i, j]`, é o texto do rótulo, e os demais parâmetros definem a justificativa do rótulo. Antes de chamar o método `text()`, usamos uma instrução `if` para filtrar os pares antecedente/consequente sem nenhum dado da métrica lift `4`. Caso contrário, um rótulo `NaN` apareceria em cada quadrado vazio do mapa de calor.

A Figura 11.1 mostra a visualização resultante.

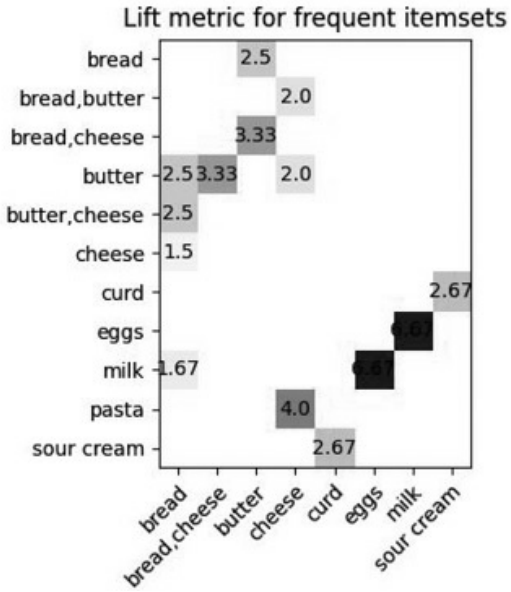


Figura 11.1: Mapa de calor da métrica lift para as regras de associação de amostra.

O mapa de calor nos ajudar a identificar de imediato quais regras de associação têm os maiores valores de lift com base nas cores escuras e sombreadas. Ao observar essa visualização, é possível afirmar com um alto grau de certeza que um cliente que compra milk (leite) também provavelmente comprará eggs (ovos). Da mesma forma, podemos ter certeza de que um cliente que compra pasta (macarrão) também comprará cheese (queijo). Outras associações, como butter (manteiga) com cheese, também existem, mas, como se pode observar, não são respaldadas tão fortemente pela métrica lift.

O mapa de calor também ilustra como a métrica lift é simétrica. Vejamos, por exemplo, os valores das regras `bread -> butter` e `butter -> bread`. São iguais. Contudo, é possível perceber que alguns pares antecedentes/consequentes no gráfico não têm um valor de lift simétrico. Por exemplo, o lift para a regra `cheese -> bread` é mostrado como `1.5`, mas, no gráfico, não existe um valor lift para `bread -> cheese`. Isso ocorre porque, de início, quando geramos as regras de associação com a função `association_rules()` da mlxtend, definimos um threshold de confiança de 50%. Isso excluiu muitas regras potenciais de associação, incluindo `bread -> cheese`, que tem um índice de confiança de 37,5% em comparação com o índice de confiança de 60% de `cheese -> bread`. Assim, nenhum dado para a regra `bread -> cheese` estava disponível para plotar.

Obtendo insights acionáveis a partir de regras de associação

Com o algoritmo Apriori, identificamos os conjuntos de itens frequentes em um lote de amostra de dados de transações e geramos regras de associação com base nesses conjuntos de itens. Basicamente, essas regras informam qual é a probabilidade de um cliente comprar um produto se tiver comprado outro e, visualizando as métricas lift das regras em um mapa de calor, vimos quais regras eram bastante convincentes. A próxima questão lógica a considerar é como uma empresa poderia se beneficiar dessas informações.

Nesta seção, veremos duas formas diferentes pelas quais uma empresa pode obter insights úteis a partir de uma coleção de regras de associação. Examinaremos como gerar recomendações de produtos com base nos itens que um cliente comprou e como planejar com eficiência descontos com base em conjuntos de itens frequentes. Ambas as aplicações têm o potencial de aumentar a receita da empresa e, ao mesmo tempo, proporcionar uma melhor experiência aos clientes.

Gerando recomendações

Após um cliente colocar um item em sua cesta ou carrinho de supermercado, qual é o próximo item que provavelmente será adicionado? Claro que não podemos afirmar nada com certeza, mas podemos efetuar uma predição com base nas regras de associação extraídas de nossos dados de transações. Os resultados dessa predição podem formar a base para um conjunto de recomendações de itens que são frequentemente comprados com o item atualmente no carrinho/cesta. Em geral, as empresas varejistas usam essas recomendações para mostrar aos clientes outros itens que podem querer comprar.

Talvez a maneira mais natural de gerar recomendações desse tipo seja analisar todas as regras de associação em que o atual item na cesta/carrinho se comporte como antecedente. Em seguida, identificamos as regras mais fortes – talvez as três regras com os maiores valores de confiança – e extraímos seus consequentes. O exemplo a seguir ilustra como fazer isso para o item *butter*. Começamos identificando as regras em que *butter* é o antecedente, usando os recursos de filtragem da biblioteca pandas:

```
butter_antecedent = rules[rules['antecedents'] == {'butter'}]
[['consequents','confidence']]
            .sort_values('confidence', ascending = False)
```

Aqui, ordenamos as regras pela coluna `confidence`, de modo que as regras com índice de confiança maior apareçam no início do DataFrame `butter_antecedent`. Depois, usamos uma list comprehension para extrair os três principais consequentes:

```
butter_consequents = [list(item) for item in butter_antecedent.iloc[0:3:,]
['consequents']]
```

Nessa list comprehension, iteramos com um loop sobre a coluna `consequents` no DataFrame `butter_antecedent`, extraindo os três primeiros valores. Com base na lista `butter_consequents`, é possível gerar uma recomendação:

```
item = 'butter'
print('Items frequently bought together with', item, 'are:', butter_consequents)
```

Vejamos qual será a recomendação:

```
Items frequently bought together with butter are: [['bread'], ['cheese'],
['cheese', 'bread']]
```

Isso sinaliza que os clientes que compram butter também costumam comprar bread (pão) ou cheese, ou ambos.

Planejando descontos com base nas regras da associação

As regras de associação geradas para conjuntos de itens frequentes também podem ser usadas para escolher quais produtos devem receber descontos. De preferência, devemos ter um item com desconto em cada grupo significativo de produtos a fim de satisfazer o maior número possível de clientes. Em outras palavras, devemos escolher um único item para receber desconto em cada conjunto de itens frequente.

Para tal, primeiro, precisamos de um conjunto de conjuntos de itens frequentes. Infelizmente, o DataFrame `rules` gerado anteriormente pela função `association_rules()` têm colunas para antecedentes e consequentes, mas não para os conjuntos de itens completos das regras. Por isso, é necessário criar uma coluna `itemsets` fazendo o merge das colunas `antecedents` e `consequents`, como mostrado a seguir:

```
from functools import reduce
rules['itemsets'] = rules[['antecedents', 'consequents']].apply(lambda x:
                reduce(frozenset.union, x), axis=1)
```

Usamos a função `reduce()` do módulo `functools` do Python para aplicar o método `frozenset.union()` aos valores das colunas `antecedents` e `consequents`. Isso combina os frozensets separados dessas colunas em um único.

Para ver o resultado, podemos exibir a coluna `itemsets` recém-criada com as colunas `antecedents` e `consequents`:

```
print(rules[['antecedents','consequents','itemsets']])
```

Veremos a seguinte saída:

```
         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)
```

Perceba que há algumas duplicatas na coluna nova `itemsets`. Como analisado antes, o mesmo conjunto de itens pode formar mais de uma regra de associação, pois a ordem dos itens influencia algumas métricas das regras. Como a ordem dos itens em um conjunto de itens não importa para a tarefa atual, podemos remover os conjuntos de itens duplicados com segurança, conforme mostrado a seguir:

```
rules.drop_duplicates(subset=['itemsets'], keep='first', inplace=True)
```

Recorremos ao método `drop_duplicates()` do DataFrame para fazer isso, especificando para procurar duplicatas na coluna `itemsets`. Mantivemos a primeira linha dentro de um conjunto de duplicatas e definimos `inplace` como `True`, excluindo as linhas duplicadas do DataFrame existente em vez de criar um DataFrame novo com as duplicatas removidas.

Vejamos como fica a coluna `itemsets` agora:

```
print(rules['itemsets'])
```

Veremos apenas o seguinte:

```
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)
```

Depois, escolhemos um item de cada conjunto de itens para receberem descontos:

```
  discounted = []
  others = []
1 for itemset in rules['itemsets']:
  2 for i, item in enumerate(itemset):
    3 if item not in others:
      4 discounted.append(item)
        itemset = set(itemset)
        itemset.discard(item)
      5 others.extend(itemset)
        break
    6 if i == len(itemset)-1:
        discounted.append(item)
        itemset = set(itemset)
        itemset.discard(item)
        others.extend(itemset)
  print(discounted)
```

Primeiro, criamos a lista `discounted` para armazenar os itens que estão sendo escolhidos para receber desconto e a lista `others` para receber os itens em um conjunto de itens que não são escolhidos para receber desconto. Em seguida, iteramos com um loop sobre cada conjunto de itens `1` e cada item nele `2`. Procuramos por um item que ainda não esteja incluído na lista `others`, pois tal item não estaria em nenhum dos conjuntos de itens anteriores, ou já tenha sido escolhido como item com desconto para um conjunto de itens anterior. Ou seja, seria eficiente escolhê-lo como item com desconto a partir desse conjunto de itens também `3`. Enviamos o item escolhido para a lista `discounted` `4` e, em seguida, enviamos os itens restantes do conjunto de itens para a lista `others` `5`. Caso tenha iterado com um loop sobre todos os itens em um conjunto de itens e não tenha conseguido encontrar um item que não esteja incluído na lista `others`, escolha o último item no conjunto de itens e o envie para a lista `discounted` `6`.

A lista `discounted` resultante sofrerá variação, pois os frozensets do Python que representam os conjuntos de itens não estão ordenados, mas veremos mais ou menos o seguinte:

```
['bread', 'bread', 'bread', 'cheese', 'pasta', 'curd', 'eggs', 'bread']
```

Comparamos o resultado com a coluna `itemsets` mostrada anteriormente e

vemos que cada conjunto de itens tem um item com desconto. Além disso, aplicamos os descontos de forma muito eficiente, de modo que o número real de itens com desconto é significativamente menor do que o número de conjuntos de itens. Podemos visualizar isso removendo as duplicatas da lista `discounted`:

```
print(list(set(discounted)))
```

Como mostra a saída, embora existam oito conjuntos de itens, só precisamos atribuir desconto a cinco itens:

```
['cheese', 'eggs', 'bread', 'pasta', 'curd']
```

Assim, foi possível atribuir desconto a um item em cada conjunto de itens (benefício substancial para muitos clientes) sem termos de aplicar desconto a muitos itens (benefício significativo para a empresa).

EXERCÍCIO #18: MINERAÇÃO DE DADOS DE TRANSAÇÕES REAIS

Neste capítulo, trabalhamos com um conjunto de dados de amostra contendo apenas 20 transações. Agora, finalmente chegou o momento de pôr a mão na massa: você trabalhará com um grande conjunto de dados. Neste exercício, será necessário obter conjuntos de itens frequentes a partir de uma coleção do mundo cotidiano com mais de meio milhão de itens comprados. É possível encontrar esse conjunto de dados no UCI Machine Learning Repository em `https://archive.ics.uci.edu/ml/datasets/online+retail`.

É necessário fazer o download do arquivo *Online Retail.xlsx* ou usar o link direto `https://archive.ics.uci.edu/ml/machine-learning-databases/00352/Online%20Retail.xlsx` para acessar o arquivo. Mas, antes de ler esse arquivo do Excel em um DataFrame do pandas para processamento posterior, você precisará instalar a biblioteca openpyxl. Essa biblioteca, que possibilita manipular arquivos do Excel a partir de scripts do Python, pode ser instalada com `pip` da seguinte forma:

```
$ pip install openpyxl
```

Depois disso, use a openpyxl para carregar o conjunto de dados em um DataFrame:

```
import pandas as pd
df_retail = pd.read_excel('path/to/Online Retail.xlsx', index_col=0,
engine='openpyxl')
```

Dependendo da sua máquina, o processo de carregar pode levar minutos para ser concluído. Em seguida, para garantir que a carga tenha sido bem-sucedida, verifique o número de linhas no DataFrame `df_retail` e gere as primeiras linhas:

```
print('The number of instances: ', len(df_retail))
print(df_retail.head())
```

Você verá a seguinte saída:

```
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
536365      84029E     RED WOOLLY HOTTIE WHITE...      6  2010-12-
01      3.39      UK

[5 rows x 7 columns]
```

Como dados do mundo cotidiano podem ser confusos, neste ponto, é necessário que você limpe um pouco os dados para prepará-los e processá-los. Para começar, é necessário remover todas as linhas que têm um `NaN` no campo `Description`. Como esse campo contém os nomes dos itens, os `NaNs` distorceriam o processo de identificação de conjuntos de itens frequentes:

```
df_retail = df_retail.dropna(subset=['Description'])
```

Para verificar se isso funcionou, exiba o tamanho do DataFrame `df_retail` atualizado:

```
print(len(df_retail))
```

Você verá `540455`, indicando que agora há 1.454 linhas a menos no DataFrame.

Para limpar ainda mais os dados, não se esqueça de definir explicitamente os valores em `Description` para o tipo de dados `str`:

```
df_retail = df_retail.astype({"Description":'str'})
```

Agora você está pronto para transformar o conjunto de dados em um formato que pode ser processado com a mlxtend. Primeiro, agrupe os dados em transações com a coluna `InvoiceNo` e transforme o DataFrame em uma lista de listas, em que cada lista contém o conjunto de itens atribuído a uma transação:

```
trans = df_retail.groupby(['InvoiceNo'])['Description'].apply(list).to_list()
```

Assim, você pode saber o número de transações reais representadas no conjunto de dados:

```
print(len(trans))
```

Você deve obter `24446`.

Os próximos passos da análise foram abordados em detalhes no início do capítulo. Em síntese, é necessário fazer o seguinte:

1. Transforme a lista de lista `trans` em um array booleano via codificação one-hot com o objeto `TransactionEncoder` da mlxtend.

2. Gere conjuntos de itens frequentes com a função `apriori()` (para este exemplo, use `min_support=0,025`).

3. Gere regras de associação com a função `association_rules()` (para este exemplo,

use `metric="confidence"`, `min_threshold=0.3`).

Após terminar, exiba as regras geradas:

```
print(rules.iloc[:,0:7])
```

O resultado deve ser parecido com o seguinte:

```
                        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
```

Para gerar uma coleção maior ou menor de regras de associação, teste o parâmetro `min_support` passado para a função `apriori()`, bem como os parâmetros `metric` e `threshold` passados para a função `association_rules()`.

Recapitulando

Como vimos, realizar a análise market basket é uma forma valiosa de extrair informações úteis de grandes volumes de dados de transações. Neste capítulo, aprendemos a usar o algoritmo Apriori a fim de extrair dados de transações para regras de associação e vimos como avaliar essas regras com diferentes métricas. Desse modo, conseguimos obter informações sobre quais itens são comumente comprados juntos. Empregamos esse conhecimento para fazer recomendações de produtos aos clientes e planejar descontos com eficiência.

CAPÍTULO 12

Aprendizado de máquina para análise de dados

O *aprendizado de máquina* é um método de análise de dados em que aplicações utilizam os dados existentes para identificar padrões e tomar decisões, sem serem explicitamente programadas para tal. Em outras palavras, as aplicações aprendem sozinhas, independentemente da interferência humana. Como técnica robusta de análise de dados, o aprendizado de máquina é usado em muitas áreas da atuação, incluindo, mas não se limitando a, classificação, clusterização, análise preditiva, aprendizado por associações, detecção de anomalias, análise de imagens e processamento de linguagem natural.

Neste capítulo, veremos um panorama geral de alguns conceitos fundamentais de aprendizado de máquina e, em seguida, exploraremos detalhadamente dois exemplos de aprendizado de máquina. Primeiro, faremos uma análise de sentimento, desenvolvendo um modelo para predizer o número de estrelas (de uma a cinco) associadas a uma avaliação do produto. Depois, desenvolveremos outro modelo visando predizer mudanças no preço de uma ação.

Por que aprendizado de máquina?

O aprendizado de máquina possibilita que os computadores executem tarefas que seriam difíceis, se não impossíveis, com o uso de técnicas convencionais de programação. Por exemplo, imagine que temos de criar uma aplicação de processamento de imagem que consiga distinguir entre diferentes tipos de animais com base nas fotos enviadas. Nesse cenário hipotético, já temos uma biblioteca de códigos que pode identificar as bordas de um objeto (como um

animal) em uma imagem. Assim sendo, é possível transformar o animal mostrado em uma foto em um conjunto característico de linhas. Mas como podemos distinguir programaticamente as linhas que representam dois animais diferentes — digamos, um gato e um cachorro?

Na programação tradicional, uma das abordagens seria criar manualmente regras que mapeiam todas as combinações de linhas características para um animal. Infelizmente, essa solução demandaria grande quantidade de código e poderia falhar por completo quando uma foto nova cujas bordas não se encaixassem em uma das regras definidas de forma manual fosse enviada. Em contrapartida, aplicações desenvolvidas com algoritmos de aprendizado de máquina não dependem de lógica predefinida, e sim da capacidade da aplicação de aprender de forma automática com os dados anteriormente observados. Assim, uma aplicação com tagueamento de fotos e baseada em aprendizado de máquina procuraria padrões nas combinações de linhas derivadas de fotos anteriores e, em seguida, com base em estatísticas de probabilidade, faria predições sobre os animais em fotos novas.

Tipos de aprendizado de máquina

Os cientistas de dados diferenciam diversos tipos de aprendizado de máquina. Os dois mais comuns são o aprendizado supervisionado e o aprendizado não supervisionado. Neste capítulo, trabalharemos principalmente com o aprendizado supervisionado, mas esta seção fornece um breve panorama geral de ambos.

Aprendizado supervisionado

O *aprendizado supervisionado* usa um conjunto rotulado de dados (referido como *conjunto de treinamento*) para ensinar um modelo a gerar a saída desejada quando dados novos e desconhecidos são fornecidos. Na prática, o aprendizado supervisionado é a técnica de inferir uma função que mapeia uma entrada para uma saída com base no conjunto de treinamento. No Capítulo 3, já vimos um exemplo de aprendizado supervisionado, em que usamos um conjunto de exemplos de análises de produtos para treinar um modelo visando predizer se as análises de produtos novos eram positivas ou

negativas.

Os dados de entrada para um algoritmo de aprendizado supervisionado podem representar características de objetos ou eventos do mundo cotidiano. Por exemplo, podemos usar as características de imóveis à venda (metros quadrados, número de quartos e banheiros e assim por diante) como entrada para um algoritmo desenvolvido que prediz os valores de imóveis. Esses valores seriam a saída do algoritmo. Treinaríamos o algoritmo em uma coleção de pares de entrada-saída, compostos das características de diversos imóveis e de seus valores associados, depois forneceríamos as características dos imóveis novos e receberíamos os valores estimados desses imóveis novos como saída.

Outros algoritmos de aprendizado supervisionado são desenvolvidos para trabalhar não com características, e sim com *dados observacionais*: dados coletados por meio da observação de uma atividade ou de um comportamento. Como exemplo, imagine uma série temporal gerada pelos sensores que monitoram os níveis de ruído em um aeroporto. Esses dados observacionais de ruído podem ser enviados a um algoritmo de aprendizado de máquina, com informações como hora do dia e dia da semana, para que esse algoritmo aprenda a predizer os níveis de ruído nas próximas horas. Nesse exemplo, os horários e dias da semana são a entrada e os níveis de ruído são a saída. Dito de outro modo, o algoritmo seria criado para predizer futuros dados observacionais.

ENTRADA E SAÍDA NO APRENDIZADO DE MÁQUINA

No mundo da programação, a *entrada (input)* normalmente se refere aos dados que uma função, script ou aplicação recebe. A entrada é então usada para gerar a *saída (output*), que são dados que a função, script ou aplicação retorna. Contudo, no contexto de aprendizado de máquina supervisionado, *entrada* e *saída* têm significados ligeiramente diferentes. Quando está sendo treinado, um modelo de aprendizado de máquina recebe *pares* de dados de entrada e saída, como avaliações de produtos (entrada) e suas classificações associadas como positivas ou negativas (saída). Assim sendo, após um modelo ser treinado, valores novos de entrada são fornecidos sozinhos, e o modelo gera valores adequados de saída com base no que aprendeu com os exemplos de pares de entrada-saída.

A entrada de um modelo de aprendizado de máquina pode consistir em uma ou mais variáveis, que são chamadas de *variáveis* ou *features* independentes. Ao mesmo tempo, a saída é normalmente uma única variável conhecida como variável *target* ou *dependente*, assim chamada porque a saída *depende* da entrada.

Tanto as predições de valor inicial quanto as predições de nível de ruído são exemplos de *regressão*, técnica comum de aprendizado supervisionado para predizer valores contínuos. Outra técnica comum de aprendizado supervisionado é a *classificação*, em que o modelo atribui um de um número finito de rótulos de classe a cada entrada. Distinguir entre análises de produtos favoráveis e desfavoráveis é um exemplo de classificação, assim como outras aplicações de *análise de sentimento*, em que fragmentos de texto são identificados como positivos ou negativos. Neste capítulo, mais adiante, vamos explorar um exemplo de análise de sentimento.

Aprendizado não supervisionado

O *aprendizado não supervisionado* é uma técnica de aprendizado de máquina na qual não há etapa de treinamento. Basta fornecer os dados de entrada da aplicação, sem nenhum valor de saída correspondente para aprender. Nesse aspecto, os modelos de aprendizado de máquina não supervisionados precisam trabalhar por conta própria, identificando padrões ocultos nos dados de entrada.

Um ótimo exemplo de aprendizado não supervisionado é a *análise de associação*, em que uma aplicação de aprendizado de máquina identifica itens que têm afinidade entre si dentro de um conjunto. No Capítulo 11, realizamos uma análise de associação em um conjunto de dados de transação, identificando itens comumente comprados juntos. Usamos o algoritmo Apriori, que não exige dados de saída de exemplo para aprender; ao contrário, o algoritmo coleta todos os dados da transação como uma entrada e pesquisa as transações por conjuntos de itens frequentes, representando, assim, o aprendizado sem treinamento.

Como funciona o aprendizado de máquina

Um típico pipeline de aprendizado de máquina depende de três componentes principais:

- Dados para aprender.
- Um modelo estatístico para aplicar aos dados.
- Dados novos e desconhecidos para processar.

Nas seções a seguir, examinaremos em detalhes esses componentes.

Dados para aprender

O aprendizado de máquina toma como base a ideia de que sistemas computacionais podem aprender. Ou seja, qualquer algoritmo de aprendizado de máquina exige dados para aprender. Como já analisamos, a natureza desses dados varia dependendo de o modelo de aprendizado de máquina ser supervisionado ou não supervisionado. No caso do aprendizado de máquina supervisionado, os dados para aprender assumem a forma de pares de entrada-saída, que treinam o modelo a fim de predizer saídas com base em entradas novas. Por outro lado, no aprendizado não supervisionado, o modelo recebe apenas dados de entrada, que extrai para padrões, a fim de gerar a saída.

Ainda que todas as aplicações de aprendizado de máquina exijam que os dados aprendam, o formato necessário desses dados pode variar de algoritmo para algoritmo. Muitos algoritmos aprendem a partir de um conjunto de dados organizado como uma tabela, em que as linhas representam diversas instâncias, como talvez objetos individuais ou momentos específicos no tempo, e as colunas representam atributos pertencentes a essas instâncias. Um clássico exemplo é o conjunto de dados de flores íris (https://archive.ics.uci.edu/ml/datasets/Iris). Esse conjunto tem 150 linhas, e cada uma contém observações sobre um espécime diferente de flor do tipo íris. Vejamos as primeiras quatro linhas do conjunto de dados:

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

As primeiras quatro colunas representam diferentes atributos, ou features, dos espécimes. A quinta coluna contém um rótulo para cada instância: o nome exato da espécie da flor do tipo íris. Caso tenha treinado um modelo de classificação com esse conjunto de dados, você usaria os valores das primeiras quatro colunas como variáveis independentes, ou entrada, enquanto a quinta coluna seria a variável dependente, ou saída. Após aprender com

esses dados, de preferência, o modelo seria capaz de classificar novos espécimes de flores íris.

Outros algoritmos de aprendizado de máquina aprendem com dados não tabulares. Por exemplo, o algoritmo Apriori usado para análise de associação, visto no capítulo anterior, aceita um conjunto de transações (ou cestas) de diferentes tamanhos como dados de entrada. Vejamos um exemplo simples desse conjunto de transações:

```
(butter, cheese)
(cheese, pasta, bread, milk)
(milk, cheese, eggs, bread, butter)
(bread, cheese, butter)
```

Além da questão de como os dados de aprendizado de máquina são estruturados, o tipo de dados usados também varia de algoritmo para algoritmo. Como os exemplos anteriores ilustram, alguns algoritmos de aprendizado de máquina trabalham com dados numéricos ou textuais. Existem também algoritmos desenvolvidos para trabalhar com dados de foto, de vídeo ou de áudio.

Modelo estatístico

Independentemente do formato de dados que o algoritmo de aprendizado de máquina exige, os dados de entrada devem ser transformados de forma que sejam analisados para gerar saídas. É aqui que um *modelo estatístico* entra em cena: as estatísticas são usadas para criar uma representação dos dados, de modo que o algoritmo possa identificar relacionamentos entre variáveis, detectar insights, efetuar predições sobre os dados novos, gerar recomendações e assim por diante. Os modelos estatísticos estão no cerne de qualquer algoritmo de aprendizado de máquina.

Por exemplo, o algoritmo Apriori usa a métrica suporte como modelo estatístico a fim de encontrar conjuntos de itens frequentes. (Conforme analisado no Capítulo 11, o suporte é a porcentagem de transações que incluem um conjunto de itens.) Em especial, o algoritmo identifica todos os conjuntos de itens possíveis e calcula a métrica suporte correspondente e, em seguida, seleciona apenas os conjuntos de itens com um suporte suficientemente alto. Vejamos um exemplo simples que ilustra como o

algoritmo funciona nos bastidores:

```
Itemset         Support
--------------  -------
butter, cheese  0.75
bread, cheese   0.75
milk, bread     0.50
bread, butter   0.50
```

Esse exemplo mostra apenas conjuntos de itens de dois itens. Na verdade, após calcular o suporte para cada conjunto de itens de dois itens possíveis, o algoritmo Apriori passa a analisar cada conjunto de itens de três itens, conjunto de itens de quatro itens e assim por diante. Em seguida, o algoritmo aplica os valores de suporte para os conjuntos de itens de todos os tamanhos, gerando uma lista de conjuntos de itens frequentes.

Dados desconhecidos

Em aprendizado de máquina supervisionado, depois de treinar um modelo em dados de exemplo, podemos aplicá-lo a dados novos e desconhecidos. No entanto, antes de fazer isso, talvez você queira avaliar o modelo. Por isso, é prática comum dividir o conjunto inicial de dados em conjuntos de treinamento e de teste. No conjunto de treinamento, estão os dados com os quais o modelo aprende. Já o conjunto de teste se torna os dados desconhecidos para fins de teste.

Os dados de teste ainda têm entrada e saída, mas apenas a entrada é mostrada ao modelo. Em seguida, a saída real é comparada com a saída sugerida pelo modelo a fim de avaliar a acurácia de suas predições. Depois de garantir que a acurácia do modelo seja aceitável, podemos usar dados novos de entrada para efetuar análises preditivas.

No caso de aprendizado não supervisionado, não há distinção entre os dados a serem aprendidos e os dados desconhecidos. Todos os dados são essencialmente desconhecidos, e o modelo tenta aprender com eles analisando suas features subjacentes.

Exemplo de análise de sentimentos: classificando avaliações de produtos

Agora que revisamos os conceitos básicos de aprendizado de máquina, estamos prontos para realizar uma análise de sentimento de amostra. Como explicado antes, trata-se de uma técnica de processamento de linguagem natural que possibilita determinar programaticamente se um fragmento de escrita é positivo ou negativo. (Mais categorias, como neutro, muito positivo ou muito negativo, também são possibilidades em algumas aplicações.) Basicamente, a análise de sentimentos é uma forma de classificação, técnica de aprendizado de máquina supervisionado que classifica os dados em categorias discretas.

No Capítulo 3, recorremos à scikit-learn para realizar uma análise básica de sentimentos em um conjunto de análises de produtos da Amazon. Treinamos um modelo para identificar se as avaliações eram boas ou ruins. Nesta seção, vamos incrementar o que já fizemos. Obteremos um conjunto real de análises de produtos diretamente da Amazon e o usaremos para treinar um modelo de classificação. O objetivo do modelo é predizer as classificações de estrelas das avaliações, em uma escala de uma a cinco. Desse modo, o modelo classificará as avaliações em cinco categorias possíveis, em vez de apenas duas.

Obtendo avaliações de produtos

A primeira etapa para criar um modelo é fazer o download de um conjunto de avaliações de algum produto da Amazon. Podemos fazer isso facilmente com o Amazon Reviews Exporter, extensão do navegador Google Chrome que faz o download das avaliações de um produto da Amazon em forma de arquivo CSV. É possível instalar essa extensão no navegador Chrome clicando nesta página: https://chrome.google.com/webstore/detail/amazon-reviews-exporter-c/njlppnciolcibljfdobcefcngiampidm.

Com a extensão instalada, abrimos a página de um produto da Amazon no Chrome. Nesse exemplo, usaremos a página da Amazon do *Curso Intensivo de Python* da No Starch Press, escrito por Eric Matthes (https://www.amazon.com/Python-Crash-Course-2nd-Edition/dp/1593279280)[1]. No momento em que escrevia este livro, a obra tinha 455 avaliações. Lembre-se, trata-se da versão em inglês do livro em questão. Para fazer o download das

avaliações do livro, localize o botão Amazon Reviews Exporter na barra de ferramentas do Chrome e clique nele.

Com as avaliações em um arquivo CSV, é possível lê-las em um DataFrame do pandas da seguinte forma:

```
import pandas as pd
df = pd.read_csv('reviews.csv')
```

Antes de prosseguir, podemos verificar o número total de avaliações e as primeiras avaliações que foram carregadas no DataFrame:

```
print('The number of reviews: ', len(df))
print(df[['title', 'rating']].head(10))
```

A saída deve ser mais ou menos assim:

```
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
```

Essa visualização mostra somente os campos `title` e `rating` para cada registro. No modelo, trataremos os títulos da avaliação como variável independente (ou seja, a entrada) e as classificações como variável dependente (a saída). Perceba que estamos ignorando o texto completo de cada avaliação e focando apenas os títulos. Parece razoável para treinamento de um modelo de classificação de sentimento, já que o título normalmente representa um resumo dos sentimentos do avaliador sobre o produto. Por outro lado, o texto completo da avaliação geralmente inclui outras informações não emocionais, como a descrição do conteúdo do livro.

Limpando os dados

Antes de processarmos dados do mundo cotidiano, quase sempre é necessário limpá-los. Nesse exemplo específico, precisaremos filtrar os comentários que

não estão escritos em inglês. Para isso, precisaremos determinar programaticamente o idioma de cada avaliação. Existem diversas bibliotecas do Python com recursos de detecção de linguagem; usaremos a google_trans_new.

Instalando a google_trans_new

Use `pip` para instalar a biblioteca google_trans_new, da seguinte forma:

```
$ pip install google_trans_new
```

Antes de continuar, é necessário verificar se a google_trans_new corrigiu o bug conhecido que lança uma exceção `JSONDecodeError` durante a detecção de idioma. Para isso, executamos o seguinte teste em uma sessão Python:

```
$ from google_trans_new import google_translator
$ detector = google_translator()
$ detector.detect('Good')
```

Se esse teste for executado sem erros, estamos prontos para seguir em frente. Se uma exceção `JSONDecodeError` for lançada, precisaremos fazer algumas pequenas alterações no código-fonte da biblioteca em *google_trans_new.py*. Localizamos o arquivo com `pip`:

```
$ pip show google_trans_new
```

O comando mostrará algumas informações básicas sobre a biblioteca, incluindo a localização do código-fonte em nossa máquina local. Acesse esse local e abra *google_trans_new.py* em um editor de texto. Em seguida, encontre as linhas 151 e 233:

```
response = (decoded_line + ']')
```

Vamos alterá-las para:

```
response = decoded_line
```

Salve as alterações, reinicie sua sessão Python e execute novamente o teste. Agora deve identificar corretamente *good* como uma palavra da língua inglesa:

```
$ from google_trans_new import google_translator
$ detector = google_translator()
$ detector.detect('Good')
['en', 'english']
```

NOTA *Para mais informações sobre a google_trans_new, visite*

https://pypi.org/project/google-trans-new.

Removendo avaliações em outros idiomas

Agora, estamos prontos para detectar o idioma de cada avaliação e filtrar as avaliações que não estão em inglês. No código a seguir, usamos o módulo `google_translator` da google_trans_new para identificar o idioma de cada título da avaliação e armazenamos o idioma em uma coluna nova do DataFrame. Talvez demore um pouco para detectar o idioma de um grande número de amostras, portanto seja paciente ao executar o código:

```
from google_trans_new import google_translator
detector = google_translator()
df['lang'] = df['title'].apply(lambda x: detector.detect(x)[0])
```

Primeiro, criamos um objeto `google_translator` e, em seguida, usamos uma expressão lambda para aplicar o método `detect()` do objeto a cada título de uma avaliação. Salvamos os resultados em uma coluna nova chamada `lang`. A seguir, exibimos essa coluna com `title` e `rating`:

```
print(df[['title', 'rating', 'lang']])
```

A saída deve ser mais ou menos assim:

```
                                                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
--trecho de código omitido--
440                                           Not bad       1     en
441                                              Good       5     en
442                                             Super       5     en
443                    内®eはとても¨}い¡B§@りは×       4     ja
444                                           非常用       5  zh-CN
```

Nossa próxima etapa é filtrar o conjunto de dados, mantendo apenas as avaliações escritas em inglês:

```
df = df[df['lang'] == 'en']
```

Essa operação deve reduzir o número total de linhas no conjunto de dados. Para validar se tudo funcionou, contamos o número de linhas no DataFrame atualizado:

```
print(len(df))
```

O número linhas deve ser menor do que era originalmente, já que todas as avaliações que não estão em língua inglesa foram removidas.

Dividindo e transformando os dados

Antes de prosseguir, é necessário dividir as avaliações em um conjunto de treinamento para desenvolver o modelo e em um conjunto de testes para avaliar sua acurácia. Além do mais, precisamos transformar a linguagem natural dos títulos das avaliações em dados numéricos que o modelo possa entender. Conforme visto na seção “Transformando texto em vetores de features numéricas”, do Capítulo 3, é possível usar a técnica bag of words (BoW). Confira mais uma vez essa seção para relembrar os conceitos.

O código a seguir usa a scikit-learn para dividir e transformar os dados. Ele segue o mesmo formato usado no Capítulo 3:

```
  from sklearn.model_selection import train_test_split
  from sklearn.feature_extraction.text import CountVectorizer
  reviews = df['title'].values
  ratings = df['rating'].values
1 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)
2 x_train = vectorizer.transform(reviews_train)
  x_test = vectorizer.transform(reviews_test)
```

Recapitulando, a função `train_test_split()` da scikit-learn divide aleatoriamente os dados em um conjunto de treinamento e em um conjunto de testes `1`. A classe `CountVectorizer` da biblioteca tem métodos para transformar dados textuais em vetores de features numéricas `2`. O código gera as seguintes estruturas, implementando os conjuntos de treinamento e de teste como arrays do NumPy e seus vetores de features correspondentes como matrizes esparsas da SciPy:

- `reviews_train` Array contendo os títulos das avaliações escolhidas para treinamento.
- `reviews_test` Array contendo os títulos das avaliações escolhidas para teste.
- `y_train` Array contendo as classificações por estrelas correspondentes às avaliações em `reviews_train`.

- `y_test` Array contendo as classificações por estrelas correspondentes às avaliações em `reviews_test`.
- `x_train` Matriz contendo o conjunto de vetores de features para os títulos de avaliações encontradas no array `reviews_train`.
- `x_test` Matriz contendo o conjunto de vetores de features para os títulos de avaliações encontradas no array `reviews_test`.

Estamos mais interessados em `x_train` e `x_test`, os vetores de features numéricas que a scikit-learn gerou a partir dos títulos de avaliações com a técnica BoW. Cada uma dessas matrizes deve incluir uma linha por título de avaliação, e essa linha deve representar o vetor de feature numérica do título. Para verificar o número de linhas na matriz gerada a partir do array `reviews_train`, usamos:

```
print(len(x_train.toarray()))
```

O número resultante deve ser 80% do número total de avaliações em inglês, visto que dividimos os dados em conjuntos de treinamento e de teste usando o padrão 80/20. A matriz `x_test` deve conter os outros 20% dos vetores de features, que podemos verificar com:

```
print(len(x_test.toarray()))
```

É possível também verificar o comprimento dos vetores de features na matriz de treinamento:

```
print(len(x_train.toarray()[0]))
```

Exibimos o comprimento apenas da primeira linha na matriz, mas o comprimento de cada linha é o mesmo. Vejamos o resultado:

```
442
```

No conjunto de treinamento, temos a ocorrência de 442 palavras nos títulos de avaliação. Essa coleção de palavras é chamada de *dicionário de vocabulário* do conjunto de dados.

Caso esteja curioso, vejamos como exibir a matriz completa:

```
print(x_train.toarray())
```

O resultado será mais ou menos assim:

```
[[0 0 0 ... 1 0 0]
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]
```

```
--trecho de código omitido--
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]]
```

Cada coluna da matriz corresponde a uma das palavras do dicionário de vocabulário do conjunto de dados, e os números informam o número de ocorrência de cada palavra em qualquer título de avaliação. Como podemos verificar, a matriz consiste principalmente de zeros. Isso é esperado: no conjunto de exemplos, a média do título de avaliação tem apenas de 5 a 10 palavras. No entanto, o dicionário de vocabulário de todo o conjunto tem 442 palavras. Ou seja, em uma típica linha, apenas 5 a 10 elementos de 442 serão definidos como 1 ou com um número superior. Mesmo assim, essa representação dos dados de exemplo é justamente o que precisamos a fim de treinar um modelo de classificação para análise de sentimentos.

Treinando o modelo

Já estamos prontos para treinarmos o modelo. Em termos específicos, é necessário treinar um *classificador*, um modelo de aprendizado de máquina que classifica os dados em categorias, de modo que consiga predizer o número de estrelas de uma avaliação. Para tal, usamos o classificador `LogisticRegression` da scikit-learn:

```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression()
classifier.fit(x_train, y_train)
```

Importamos a classe `LogisticRegression` e criamos um objeto `classifier`. Depois, treinamos o classificador passando a ele a matriz `x_train` (os vetores de features dos títulos de avaliações no conjunto de treinamento) e a array `y_train` (as classificações de estrelas correspondentes).

Avaliando o modelo

Como já treinamos o modelo, podemos usar a matriz `x_test` para avaliar sua acurácia, comparando as classificações preditas do modelo com as classificações reais no array `y_test`. No Capítulo 3, recorremos ao método `score()` do objeto `classifier` para avaliar sua acurácia. A seguir, usaremos um método de avaliação diferente, que permite maior precisão:

```
  import numpy as np
1 predicted = classifier.predict(x_test)
  accuracy = 2 np.mean(3 predicted == y_test)
  print("Accuracy:", round(accuracy,2))
```

Utilizamos o método `predict()` do classificador a fim de predizer as classificações com base nos vetores de features `x_test` `1`. Em seguida, testamos a equivalência entre as predições do modelo e as classificações de estrelas `3`. O resultado dessa comparação é um array booleano, em que `True` e `False` indicam predições acuradas e inadequadas. Ao calcular a média aritmética do array `2`, obtemos uma classificação de acurácia geral para o modelo. (Para fins de cálculo da média, cada `True` é tratado como `1` e cada `False` como `0`.) Vejamos o resultado:

```
  Accuracy: 0.68
```

Isso sinaliza que o modelo tem 68% de acurácia, ou seja, em média, aproximadamente 7 em cada 10 predições estão corretas. No entanto, visando obter uma compreensão mais refinada da acurácia do modelo, é necessário recorrer a outras features da scikit-learn para examinar métricas mais específicas. Por exemplo, podemos analisar a *matriz de confusão* do modelo, grade que compara classificações preditas com classificações reais. Uma matriz de confusão pode ajudar a revelar a acurácia do modelo dentro de cada classe individual, bem como mostrar se é provável que o modelo confunda duas classes (rotule incorretamente uma classe como outra). Vejamos como podemos criar a matriz de confusão para nosso modelo de classificação:

```
  from sklearn import metrics
  print(metrics.confusion_matrix(y_test, predicted, labels = [1,2,3,4,5]))
```

Importamos o módulo `metrics` da scikit-learn e usamos o método `confusion_matrix()` para gerar a matriz. Passamos ao método as classificações reais do conjunto de teste (`y_test`), as classificações preditas pelo modelo (`predicted`) e os rótulos correspondentes a essas classificações. A matriz será mais ou menos assim:

```
  [[ 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]]
```

Aqui, as linhas correspondem às classificações reais e as colunas correspondem às classificações preditas. Por exemplo, na primeira linha, os números informam que o conjunto de teste continha oito classificações reais de uma estrela, uma das quais foi predita para ser uma classificação de quatro estrelas e sete das quais foram preditas como classificações de cinco estrelas.

A diagonal principal da matriz de confusão (da esquerda para a direita) mostra o número de predições corretas para cada nível de classificação. Se analisarmos essa diagonal, podemos ver que o modelo fez 54 predições corretas para classificações de cinco estrelas e apenas 1 predição correta para classificações de quatro estrelas. Nenhuma avaliação de uma, duas ou três estrelas foi identificada de forma correta. No geral, de um conjunto de testes de 81 avaliações, 55 foram corretamente preditas.

Esse resultado levanta uma série de perguntas. Para início de conversa, por que o modelo só funciona bem para avaliações de cinco estrelas? Talvez o problema seja que o conjunto de dados de exemplo tenha avaliações cinco estrelas em quantidade suficiente. Para verificar se é o caso, podemos contar as linhas em cada grupo de classificação:

```
print(df.groupby('rating').size())
```

Agrupamos o DataFrame original contendo os dados de treinamento e de teste pela coluna `rating` e usamos o método `size()` a fim de obter o número de entradas em cada grupo. Vejamos a saída:

```
rating
1     25
2     15
3     23
4     51
5    290
```

Como podemos verificar, a contagem confirma nossa hipótese: há muito mais avaliações de cinco estrelas do que qualquer outra classificação, sugerindo que o modelo não tinha dados suficientes para aprender efetivamente as features das avaliações com quatro estrelas ou menos.

Para explorar ainda mais a acurácia do modelo, podemos também querer analisar suas principais métricas de classificação, comparando os arrays `y_test` e `predicted`. Faremos isso com a ajuda da função `classification_report()`

encontrada no módulo `metrics` da scikit-learn:

```
print(metrics.classification_report(y_test, predicted, labels = [1,2,3,4,5]))
```

Vamos conferir o relatório gerado:

```
              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
```

O relatório mostra o resumo das principais métricas de classificação para cada classe de avaliações. Aqui, focaremos as métricas suporte e revogação (recall); para mais informações sobre outras métricas do relatório, confira https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html#sklea

A métrica suporte mostra o número de avaliações para cada classe de classificações. Em particular, revela que as classificações são distribuídas de forma extremamente desigual entre os grupos de classificação: o conjunto de testes exibe a mesma tendência que todo o conjunto de dados. Do total de 81 avaliações, há 57 avaliações cinco estrelas e apenas 2 avaliações duas estrelas.

A métrica revogação mostra a proporção de avaliações corretamente preditas para todas as avaliações com determinada classificação. Por exemplo, a métrica revogação para avaliações de cinco estrelas é de 0,95, ou seja, o modelo teve 95% de acurácia na predição de avaliações de cinco estrelas, enquanto a mesma métrica para avaliações de quatro estrelas é de apenas 0,14. Já que as avaliações com as outras classificações não têm predições corretas, a revogação média ponderada para todo o conjunto de testes é mostrada como 0,68 na parte inferior do relatório. Trata-se da mesma classificação de acurácia que obtivemos no início desta seção.

Considerando todos esses pontos, podemos razoavelmente concluir que o problema do conjunto de exemplos que usamos é o número bastante desigual

de avaliações em cada grupo de classificação.

EXERCÍCIO #19: INCREMENTANDO O CONJUNTO DE EXEMPLOS

Como acabamos de aprender, a acurácia geral de um modelo de classificação pode ser equivocada se tivermos, no conjunto de dados, um número desigual de instâncias de cada classe. Tente expandir esse conjunto de dados fazendo o download de mais avaliações da Amazon. Procure ter um número aproximadamente igual e suficientemente grande de instâncias de cada classificação por estrelas (digamos, 500 por grupo). Em seguida, retreine o modelo e o teste mais uma vez para ver se a acurácia melhora.

Predizendo tendências de ações

Para explorar ainda mais como o aprendizado de máquina pode ser aplicado à análise de dados, criaremos um modelo para predizer as tendências do mercado de ações. Simplificando as coisas, criaremos outro modelo de classificação: um que prediz se o preço de uma ação amanhã será maior, menor ou o mesmo de hoje. Um modelo mais sofisticado pode usar a regressão a fim de predizer o preço real diário de uma ação.

AVISO *O modelo analisado aqui é apenas um exemplo, não é para usá-lo no dia a dia. Via de regra, os modelos de aprendizado de máquina do mundo cotidiano usados para negociação de ações são mais complexos. Qualquer tentativa de usar o modelo deste livro para fazer negociações reais com ações pode resultar em perdas. Nem o autor tampouco a editora se responsabilizam por essas perdas.*

Comparado ao exemplo de análise de sentimento deste capítulo, nosso modelo de predição de ações (e, de fato, muitos modelos que envolvem dados não textuais) suscita uma nova questão: como decidimos quais dados usar como features ou entradas do modelo? Para o modelo de análise de sentimento, utilizamos os vetores de features geradas a partir do texto dos títulos de avaliação por meio da técnica BoW. O conteúdo desse vetor depende estritamente do conteúdo do texto correspondente. Nesse sentido, o conteúdo do vetor é predefinido, sendo formado a partir das features extraídas do texto correspondente, conforme determinada regra.

Por outro lado, quando o modelo envolve dados não textuais, como preços de ações, não raro, cabe a nós decidir, e talvez até calcular, o conjunto de

features a serem usados como dados de entrada para o modelo. Quem sabe a mudança percentual no preço do dia anterior, preço médio da semana passada e volume total de negociação nos dois dias anteriores? Talvez. Ou a mudança percentual no preço de dois dias anteriores, preço médio no mês passado e mudança no volume desde ontem? Pode ser. Os analistas financeiros recorrem a todos os tipos de métricas, em diferentes combinações, como dados de entrada para seus modelos de predição de ações.

No Capítulo 10, aprendemos como derivar métricas a partir de dados do mercado de ações, calculando as mudanças percentuais ao longo do tempo, as médias de janelas rolantes e similares. Nesta seção, mais adiante, revisitaremos algumas dessas técnicas a fim de gerar as features para nosso modelo de predição. Mas, primeiro, precisamos obter alguns dados.

Obtendo os dados

Para treinar nosso modelo, precisaremos de um ano de dados, isso apenas de uma ação. Neste exemplo, usaremos a Apple (AAPL). A seguir, usamos a biblioteca yfinance para obter os dados de ações da empresa para o último ano:

```
import yfinance as yf
tkr = yf.Ticker('AAPL')
hist = tkr.history(period="1y")
```

Utilizaremos o DataFrame `hist` resultante para derivar métricas sobre as ações, como a mudança percentual diária do preço, e forneceremos ao nosso modelo essas métricas. No entanto, podemos razoavelmente presumir que existem fatores externos (ou seja, informações que não podem ser derivadas dos dados de ações) que influenciam o preço das ações da Apple. Por exemplo, o desempenho geral do mercado de ações pode afetar o desempenho de uma ação propriamente dita. Assim, seria interessante também considerar os dados de um índice mais abrangente do mercado de ações como parte do modelo.

Um dos índices mais conhecidos do mercado de ações é o S&P 500. Ele mensura o desempenho das ações das 500 maiores empresas. Conforme visto no Capítulo 4, é possível obter dados do S&P 500 no Python por meio da biblioteca pandas-datareader. Aqui, recorremos ao método `get_data_stooq()` da

biblioteca para obter um ano de dados do S&P 500 do site da Stooq:

```
  import pandas_datareader.data as pdr
  from datetime import date, timedelta
  end = date.today()
1 start = end - timedelta(days=365)
2 index_data = pdr.get_data_stooq('^SPX', start, end)
```

Com o módulo `datetime` do Python, definimos as datas de início e término de nossa query em relação à data atual `1`. Depois, chamamos o método `get_data_stooq()`, usando `'^SPX'` para solicitar os dados S&P 500 e armazenamos o resultado no DataFrame `index_data` `2`.

Agora que temos os valores das ações da Apple e os números do índice S&P 500 para o mesmo período de um ano, podemos combinar os dados em um único DataFrame:

```
  df = hist.join(index_data, rsuffix = '_idx')
```

Os DataFrames que passaram por join têm colunas com os mesmos nomes. Para evitar sobreposição, usamos o parâmetro `rsuffix`. Ele instrui o método `join()` a adicionar o sufixo `'_idx'` a todos os nomes da coluna do DataFrame `index_data`.

Aqui, o que nos interessa são os preços de fechamento diários e os volumes de negociação da Apple e do S&P 500. A seguir, filtramos o DataFrame somente para essas colunas:

```
  df = df[['Close','Volume','Close_idx','Volume_idx']]
```

Vejamos como ficou o DataFrame `df` agora:

```
                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
```

O DataFrame contém uma série temporal multivariada contínua. A próxima etapa é derivar as features dos dados que podem ser usadas como entrada para o modelo de aprendizado de máquina.

Derivando features de dados contínuos

Queremos treinar nosso modelo em informações relativas às mudanças diárias de preço e volume. Conforme aprendemos no Capítulo 10, calculamos as mudanças percentuais nos dados de séries temporais contínuas deslocando os pontos de dados no tempo, alinhando os pontos de dados passados com os pontos de dados atuais para compará-los. No código a seguir, usamos `shift(1)` para calcular a mudança percentual diária de cada coluna do DataFrame, salvando os resultados em um novo lote de colunas:

```
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()
```

Para cada uma das quatro colunas, dividimos cada ponto de dados pelo ponto de dados do dia anterior e, em seguida, obtivemos o logaritmo natural do resultado. Lembre-se, o logaritmo natural fornece uma aproximação próxima da mudança percentual. Vejamos as diversas colunas novas:

- `priceRise` Mudança percentual diária do preço das ações da Apple.
- `volumeRise` Mudança percentual do volume diário de negociação da Apple.
- `priceRise_idx` Mudança percentual diária do preço do índice S&P 500.
- `volumeRise_idx` Mudança percentual do volume diário de negociação do S&P 500.

Agora, podemos filtrar o DataFrame novamente para incluir apenas as colunas novas:

```
df = df[['priceRise','volumeRise','priceRise_idx','volumeRise_idx']]
```

Vejamos o conteúdo do DataFrame:

```
            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
--trecho de código omitido--
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
```

Essas colunas se tornarão as features, ou variáveis independentes, para o modelo.

Gerando a variável de saída

A próxima etapa é gerar a variável de saída (também chamada de variável target ou dependente) para o conjunto de dados existente. Essa variável deve repassar o que acontece com o preço das ações no dia seguinte: sobe, desce ou permanece o mesmo? Podemos identificar isso com a coluna `priceRise` do dia seguinte, que acessamos por meio de `df['priceRise'].shift(-1)`. A mudança negativa desloca os valores futuros um passo atrás no tempo. Com base nessa mudança, é possível gerar uma coluna nova com `-1` se o preço cair, `0` se o preço permanecer o mesmo ou `1` se o preço subir. Vejamos como:

```
conditions = [
    (df['priceRise'].shift(-1) > 0.01),
    (df['priceRise'].shift(-1)< -0.01)
]
choices = [1, -1]
df['Pred'] = np.select(conditions, choices, default=0)
```

O algoritmo implementado assume o seguinte:

1. Um aumento de preço de mais de 1% em relação ao dia seguinte é considerado um aumento (`1`).
2. Uma queda de preço de mais de 1% em relação ao dia seguinte é considerada uma queda (`-1`).
3. O restante é considerado estagnação (`0`).

Para implementar o algoritmo, definimos a lista `conditions` que verifica os dados de acordo com os pontos 1 e 2 `1`, bem como a lista `choices` com os valores `1` e `-1` para indicar um aumento ou queda no preço `2`. Em seguida,

fornecemos as duas listas para a função `select()` do NumPy [3], que cria um array selecionando valores de `choices` com base nos valores de `conditions`. Se nenhuma das condições for satisfeita, o valor padrão de `0` é atribuído, satisfazendo o ponto 3. Armazenamos o array em uma coluna nova do DataFrame, `Pred`, que pode ser usada como saída para treinar e testar nosso modelo. Basicamente, `-1`, `0 e 1` são agora as classes possíveis que o modelo pode escolher ao classificar dados novos.

Treinando e avaliando o modelo

Para treinar nosso modelo, a scikit-learn exige que apresentemos os dados de entrada e saída em arrays separados do NumPy. Geramos os arrays a partir do DataFrame `df`:

```
features =
df[['priceRise','volumeRise','priceRise_idx','volumeRise_idx']].to_numpy()
features = np.around(features, decimals=2)
target = df['Pred'].to_numpy()
```

O array `features` agora contém as quatro variáveis independentes (a entrada) e o array `target` contém a única variável dependente (a saída). Em seguida, podemos dividir os dados em conjuntos de treinamento e de teste e treinar o modelo:

```
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)
```

Assim como fizemos no exemplo de análise de sentimento no início do capítulo, recorremos à função `train_test_split()` da scikit-learn para dividir o conjunto de dados de acordo com o padrão 80/20 e usamos o classificador `LogisticRegression` para treinar o modelo. Depois, passamos a parte de teste do conjunto de dados para o método `score()` do classificador a fim de avaliar sua acurácia:

```
print(clf.score(rows_test, y_test))
```

Vejamos o resultado:

```
0.6274509803921569
```

Isso indica que o modelo previu acuradamente a trajetória diária das ações da Apple em cerca de 62% das vezes. Claro que você pode obter um valor diferente.

EXERCÍCIO #20: TESTANDO AÇÕES DIFERENTES E MÉTRICAS NOVAS

Continuando com o exemplo anterior, teste diferentes ações e considere métricas novas derivadas dos dados de ações para usar como variáveis independentes adicionais e tente melhorar a acurácia do modelo. Talvez você queira usar algumas das métricas derivadas do Capítulo 10.

Recapitulando

Neste capítulo, aprendemos como algumas tarefas de análise de dados, como a classificação, podem ser efetuadas com o aprendizado de máquina, método que possibilita que os sistemas computacionais aprendam com dados históricos ou experiências passadas. Em termos específicos, estudamos como os algoritmos de aprendizado de máquina podem ser usados para a tarefa de PLN de análise de sentimentos. Convertemos dados textuais de análises de produtos da Amazon em vetores de features numéricas legíveis por máquina e, em seguida, treinamos um modelo para classificar as avaliações de acordo com suas classificações por estrelas. Além do mais, aprendemos a gerar features com base em dados numéricos do mercado de ações e as utilizamos para treinar um modelo visando predizer as mudanças no preço de uma ação.

Há um mundo de possibilidades em que podemos combinar aprendizado de máquina, métodos estatísticos, APIs públicas e recursos de estruturas de dados disponíveis no Python. Este livro elencou algumas dessas possibilidades, abordando uma variedade de tópicos. Com isso, espero que os leitores se sintam inspirados a encontrar uma série de soluções inovadoras.

---

1 N.T.: Este livro foi traduzido pela Novatec Editora. https://novatec.com.br/livros/curso-intensivo-python-3ed/


Entendendo
algoritmos
Um guia *ilustrado* para programadores e outros curiosos
Aditya Y. Bhargava
novatec
MANNING

Entendendo Algoritmos

Bhargava, Aditya Y.

9788575226629

264 páginas

Um guia ilustrado para programadores e outros curiosos. Um algoritmo nada mais é do que um procedimento passo a passo para a resolução de um problema. Os algoritmos que você mais utilizará como um programador já foram descobertos, testados e provados. Se você quer entendê-los, mas se recusa a estudar páginas e mais páginas de provas, este é o livro certo. Este guia cativante e completamente ilustrado torna simples aprender como utilizar os principais algoritmos nos seus programas. O livro Entendendo Algoritmos apresenta uma abordagem agradável para esse tópico essencial da ciência da computação. Nele, você aprenderá como aplicar algoritmos comuns nos problemas de programação enfrentados diariamente. Você começará com tarefas básicas como a ordenação e a pesquisa. Com a prática, você enfrentará problemas mais complexos, como a compressão de dados e

a inteligência artificial. Cada exemplo é apresentado em detalhes e inclui diagramas e códigos completos em Python. Ao final deste livro, você terá dominado algoritmos amplamente aplicáveis e saberá quando e onde utilizá-los. O que este livro inclui A abordagem de algoritmos de pesquisa, ordenação e algoritmos gráficos Mais de 400 imagens com descrições detalhadas Comparações de desempenho entre algoritmos Exemplos de código em Python Este livro de fácil leitura e repleto de imagens é destinado a programadores autodidatas, engenheiros ou pessoas que gostariam de recordar o assunto.


Loiane Groner
Estruturas de dados e algoritmos com JavaScript
2ª Edição
Escreva um código JavaScript complexo e eficaz usando a mais recente ECMAScript
novatec
Packt

Estruturas de dados e algoritmos com JavaScript

Groner, Loiane

9788575227282

408 páginas

Uma estrutura de dados é uma maneira particular de organizar dados em um computador com o intuito de usar os recursos de modo eficaz. As estruturas de dados e os algoritmos são a base de todas as soluções para qualquer problema de programação. Com este livro, você aprenderá a escrever códigos complexos e eficazes usando os recursos mais recentes da ES 2017. O livro Estruturas de dados e algoritmos com JavaScript começa abordando o básico sobre JavaScript e apresenta a ECMAScript 2017, antes de passar gradualmente para as estruturas de dados mais importantes, como arrays, filas, pilhas e listas ligadas. Você adquirirá um conhecimento profundo sobre como as tabelas hash e as estruturas de dados para conjuntos funcionam, assim como de que modo as árvores e os mapas hash podem ser usados para buscar arquivos em um disco rígido

ou para representar um banco de dados. Este livro serve como um caminho para você mergulhar mais fundo no JavaScript. Você também terá uma melhor compreensão de como e por que os grafos – uma das estruturas de dados mais complexas que há – são amplamente usados em sistemas de navegação por GPS e em redes sociais. Próximo ao final do livro, você descobrirá como todas as teorias apresentadas podem ser aplicadas para solucionar problemas do mundo real, trabalhando com as próprias redes de computador e com pesquisas no Facebook. Você aprenderá a: • declarar, inicializar, adicionar e remover itens de arrays, pilhas e filas; • criar e usar listas ligadas, duplamente ligadas e ligadas circulares; • armazenar elementos únicos em tabelas hash, dicionários e conjuntos; • explorar o uso de árvores binárias e árvores binárias de busca; • ordenar estruturas de dados usando algoritmos como bubble sort, selection sort, insertion sort, merge sort e quick sort; • pesquisar elementos em estruturas de dados usando ordenação sequencial e busca binária

O'REILLY

Introdução à linguagem SQL

ABORDAGEM PRÁTICA
PARA INICIANTES

novatec

Thomas Nield

Introdução à Linguagem SQL

Nield, Thomas

9788575227466

144 páginas

Atualmente as empresas estão coletando dados a taxas exponenciais e mesmo assim poucas pessoas sabem como acessá-los de maneira relevante. Se você trabalha em uma empresa ou é profissional de TI, este curto guia prático lhe ensinará como obter e transformar dados com o SQL de maneira significativa. Você dominará rapidamente os aspectos básicos do SQL e aprenderá como criar seus próprios bancos de dados. O autor Thomas Nield fornece exercícios no decorrer de todo o livro para ajudá-lo a praticar em casa suas recém descobertas aptidões no uso do SQL, sem precisar empregar um ambiente de servidor de banco de dados. Além de aprender a usar instruções-chave do SQL para encontrar e manipular seus dados, você descobrirá como projetar e gerenciar eficientemente bancos de dados que atendam às suas necessidades. Também veremos como: •Explorar bancos de dados relacionais,

usando modelos leves e centralizados •Usar o SQLite e o SQLiteStudio para criar bancos de dados leves em minutos •Consultar e transformar dados de maneira significativa usando SELECT, WHERE, GROUP BY e ORDER BY •Associar tabelas para obter uma visualização mais completa dos dados da empresa •Construir nossas próprias tabelas e bancos de dados centralizados usando princípios de design normalizado •Gerenciar dados aprendendo como inserir, excluir e atualizar registros


Fundamentos de
HTML5 e CSS3
CSS3
HTML5
novatec
Maurício Samy Silva
www.maujor.com

Fundamentos de HTML5 e CSS3

Silva, Maurício Samy

9788575227084

304 páginas

Fundamentos de HTML5 e CSS3 tem o objetivo de fornecer aos iniciantes e estudantes da área de desenvolvimento web conceitos básicos e fundamentos da marcação HTML e estilização CSS, para a criação de sites, interfaces gráficas e aplicações para a web. Maujor aborda as funcionalidades da HTML5 e das CSS3 de forma clara, em linguagem didática, mostrando vários exemplos práticos em funcionamento no site do livro. Mesmo sem conhecimento prévio, com este livro o leitor será capaz de: •Criar um código totalmente semântico empregando os elementos da linguagem HTML5. •Usar os atributos da linguagem HTML5 para criar elementos gráficos ricos no desenvolvimento de aplicações web. •Inserir mídia sem dependência de plugins de terceiros ou extensões proprietárias. •Desenvolver formulários altamente interativos com validação no lado do cliente utilizando atributos criados especialmente para essas finalidades.

•Conhecer os mecanismos de aplicação de estilos, sua sintaxe, suas propriedades básicas, esquemas de posicionamento, valores e unidades CSS3. •Usar as propriedades avançadas das CSS3 para aplicação de fundos, bordas, sombras, cores e opacidade. •Desenvolver layouts simples com uso das CSS3.

A Linguagem de Programação Go

Alan A. A. Donovan
Brian W. Kernighan

novatec

A Linguagem de Programação Go

Donovan, Alan A. A.

9788575226551

480 páginas

A linguagem de programação Go é a fonte mais confiável para qualquer programador que queira conhecer Go. O livro mostra como escrever código claro e idiomático em Go para resolver problemas do mundo real. Esta obra não pressupõe conhecimentos prévios de Go nem experiência com qualquer linguagem específica, portanto você a achará acessível, independentemente de se sentir mais à vontade com JavaScript, Ruby, Python, Java ou C++.

Compre agora e leia