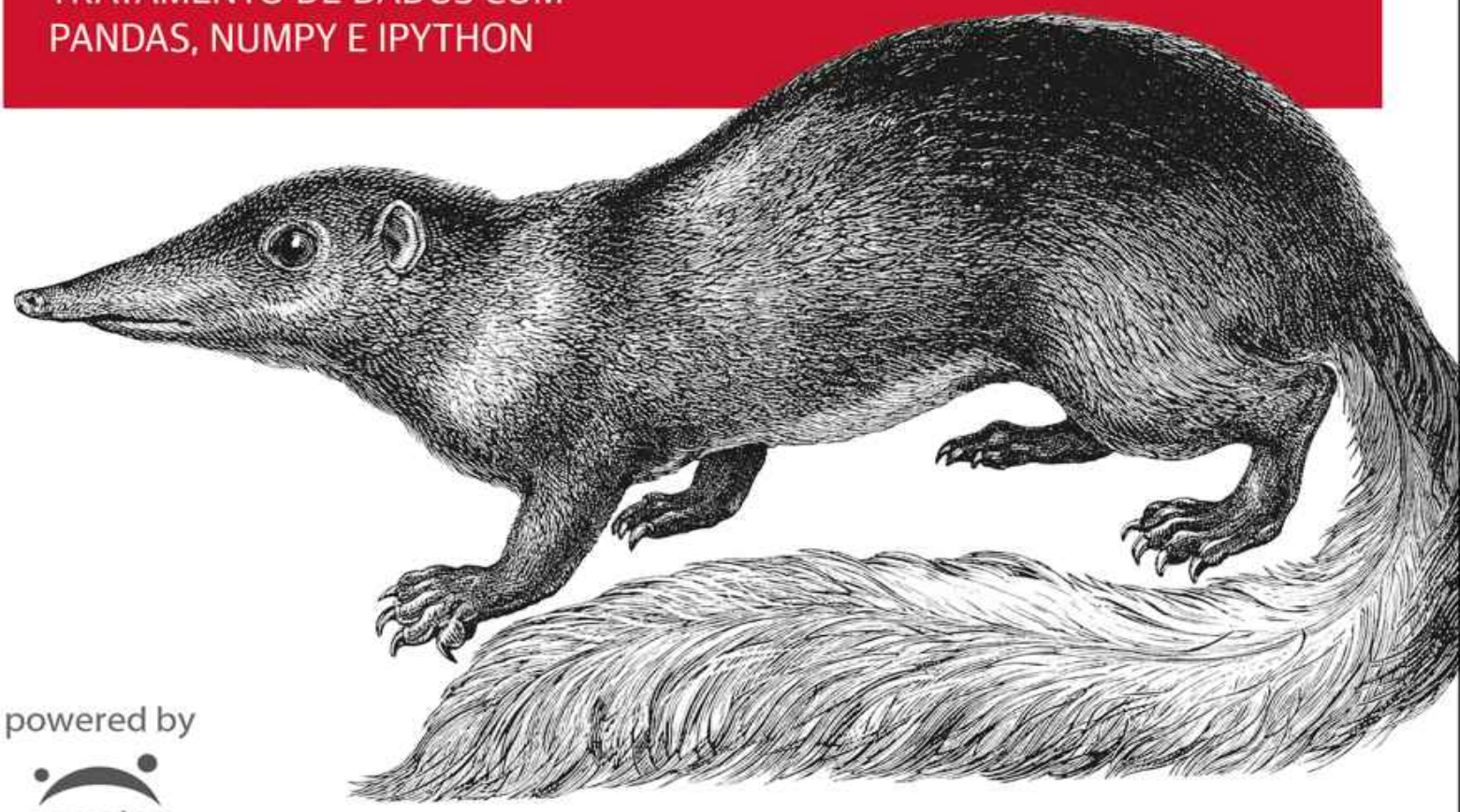
O'REILLY®

Python para Análise de Dados

TRATAMENTO DE DADOS COM PANDAS, NUMPY E IPYTHON



powered by

jupyter

novatec

Wes McKinney

Python para Análise de Dados

TRATAMENTO DE DADOS COM PANDAS, NUMPY E IPYTHON

Wes McKinney


O'REILLY
Novatec

Tradução em português autorizada da edição em inglês da obra Python for Data Analysis, 2E, ISBN 9781491957660 © 2018 William Wesley McKinney. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

Editor: Rubens Prates
Tradução: Lúcia A. Kinoshita
Revisão gramatical: Tássia Carvalho
Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-751-0

Histórico de edições impressas:

Fevereiro/2019 Segunda reimpressão
Agosto/2018 Primeira reimpressão
Janeiro/2018 Primeira edição

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

Sumário

Prefácio

Novidades na segunda edição

A primeira edição deste livro foi publicada em 2012, época em que as bibliotecas de análise de dados com código aberto para Python (como o pandas) eram muito novas e estavam se desenvolvendo rapidamente. Nesta segunda edição, atualizada e expandida, revisei os capítulos de modo a levar em conta tanto as mudanças incompatíveis e as informações obsoletas bem como os novos recursos que surgiram nos últimos cinco anos. Também acrescentei conteúdo novo a fim de apresentar as ferramentas que não existiam em 2012 ou que não haviam amadurecido suficientemente para estarem na primeira edição. Por fim, tentei evitar escrever sobre projetos de código aberto novos ou pioneiros que talvez não tenham tido a chance de amadurecer. Gostaria que os leitores desta edição achassem que o conteúdo ainda será quase tão relevante em 2020 ou em 2021 quanto o é em 2017.

As principais atualizações nesta segunda edição incluem:

- todo o código, englobando o tutorial de Python, atualizado para Python 3.6 (a primeira edição utilizava Python 2.7);
- instruções de instalação de Python atualizadas para a Anaconda Python Distribution e outros pacotes Python necessários;
- atualizações para as versões mais recentes da biblioteca pandas em 2017;
- um novo capítulo sobre algumas ferramentas mais sofisticadas do pandas, além de outras dicas de uso;
- uma rápida introdução ao uso do statsmodels e do scikit-learn.

Também reorganizei uma parte significativa do conteúdo da primeira edição a fim de tornar o livro mais acessível aos que estão

chegando.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica termos novos, URLs, endereços de email, nomes e extensões de arquivos.

Largura constante

Usada para listagens de programas, assim como em parágrafos para se referir a elementos de programas, como nomes de variáveis ou de funções, bancos de dados, tipos de dados, variáveis de ambiente, comandos e palavras-chave.

Largura constante em negrito

Mostra comandos ou outro texto que devam ser digitados literalmente pelo usuário.

Largura constante em itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou determinados pelo contexto.

 Este elemento significa uma dica ou sugestão.

 Este elemento significa uma observação geral.

 Este elemento significa um aviso ou uma precaução.

Uso de exemplos de código de acordo com a política da O’Reilly

Você encontrará os arquivos de dados e o material relacionado a

cada capítulo à sua disposição no repositório do GitHub para este livro em *http://github.com/wesm/pydata-book*.

Esta obra está aqui para ajudá-lo a fazer seu trabalho. De modo geral, se este livro incluir exemplos de código, você poderá usar o código em seus programas e em sua documentação. Não é necessário nos contatar para pedir permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use diversas partes de código deste livro não requer permissão, porém vender ou distribuir um CD-ROM de exemplos de livros da O'Reilly requer permissão. Responder a uma pergunta mencionando este livro e citar o código de exemplo não requer permissão. Em contrapartida, incluir uma quantidade significativa de código de exemplos deste livro na documentação de seu produto requer permissão.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: "*Python for Data Analysis* de Wes McKinney (O'Reilly). Copyright 2017 Wes McKinney, 978-1-491-95766-0".

Se você achar que o seu uso dos exemplos de código está além do razoável ou da permissão concedida, sinta-se à vontade para nos contatar em *permissions@oreilly.com*.

Como entrar em contato conosco

Envie seus comentários e suas dúvidas sobre este livro à editora pelo email: *novatec@novatec.com.br*.

Temos uma página web para este livro, na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição traduzida para o português

  http://www.novatec.com.br/livros/python-para-analise-de-dados

- Página da edição original em inglês

  http://bit.ly/python_data_analysis_2e

Para obter mais informações sobre os livros da Novatec, acesse nosso site em: *http://www.novatec.com.br*.

Agradecimentos

Esta obra é o produto de muitos anos de discussões produtivas, colaborações e da assistência mútua e de várias pessoas do mundo todo. Gostaria de agradecer a algumas delas.

In memoriam: John D. Hunter (1968–2012)

Nosso querido amigo e colega John D. Hunter faleceu em 28 de agosto de 2012 depois de batalhar contra um câncer de cólon. Isso ocorreu logo após a conclusão do manuscrito final para a primeira edição deste livro.

Nunca é demais falar do impacto de John e de seu legado às comunidades científica e de dados de Python. Além de ter desenvolvido a matplotlib no início dos anos 2000 (em uma época em que Python estava longe de ser tão popular), ele ajudou a moldar a cultura de uma geração crítica de desenvolvedores de código aberto que se transformou nos pilares do ecossistema de Python, o qual atualmente tomamos como dado.

Tive a sorte de ter contato com John no início de minha carreira em código aberto, em janeiro de 2010, logo após o lançamento do pandas 0.1. Sua inspiração e sua orientação como mentor me ajudaram a seguir em frente, mesmo nas horas mais difíceis, com minha visão sobre o pandas e sobre Python como uma linguagem de primeira classe para análise de dados.

John era muito próximo de Fernando Pérez e Brian Granger, pioneiros do IPython, do Jupyter e de muitas outras iniciativas na comunidade Python. Esperávamos trabalhar juntos, nós quatro, em um livro, mas o fato é que eu acabei sendo a pessoa com a maior parte do tempo livre. Tenho certeza de que ele estaria orgulhoso com o que realizamos, como indivíduos e como uma comunidade,

nos últimos cinco anos.

Agradecimentos para a segunda edição (2017)

Já faz quase cinco anos desde que concluí o manuscrito para a primeira edição deste livro em julho de 2012. Muita coisa mudou. A comunidade Python cresceu imensamente e o ecossistema de softwares de código aberto em torno dela floresceu.

Esta nova edição do livro não teria existido se não fossem os incansáveis esforços dos desenvolvedores do núcleo do pandas, que expandiram o projeto e a sua comunidade de usuários, transformando-o em um dos pilares do ecossistema de Python para ciência de dados. Aí se incluem as pessoas a seguir, sem que estejam limitadas a elas: Tom Augspurger, Joris van den Bossche, Chris Bartak, Phillip Cloud, gfyoung, Andy Hayden, Masaaki Horikoshi, Stephan Hoyer, Adam Klein, Wouter Overmeire, Jeff Reback, Chang She, Skipper Seabold, Jeff Tratner e y-p.

Sobre a escrita propriamente dita desta segunda edição, gostaria de agradecer à equipe da O'Reilly, que pacientemente me ajudou no processo. Incluem-se aí: Marie Beaugureau, Ben Lorica e Colleen Toporek. Novamente, tive revisores técnicos incríveis, com a colaboração de Tom Augpurger, Paul Barry, Hugh Brown, Jonathan Coe e Andreas Müller. Muito obrigado.

A primeira edição deste livro foi traduzida para várias línguas estrangeiras, incluindo chinês, francês, alemão, japonês, coreano e russo. Traduzir todo esse conteúdo e deixá-lo disponível a um público-alvo mais amplo é um esforço enorme e muitas vezes sem reconhecimento. Obrigado por ajudar mais pessoas no mundo a saber programar e usar ferramentas de análise de dados.

Também tive a sorte de ter recebido apoio para os meus esforços contínuos de desenvolvimento de código aberto da Cloudera e da Two Sigma Investments nos últimos anos. Com os projetos de software de código aberto com recursos mais escassos do que nunca, considerando o tamanho das bases de usuários, é cada vez

mais importante que os negócios deem apoio para o desenvolvimento de projetos essenciais de código aberto. É a atitude certa a ser tomada.

Agradecimentos para a primeira edição (2012)

Escrever este livro sem o apoio de um grande número de pessoas teria sido difícil para mim.

Na equipe da O'Reilly, sou muito grato às minhas editoras Meghan Blanchette e Julie Steele, que me orientaram durante o processo. Mike Loukides também trabalhou comigo nas fases de proposta e ajudou a tornar o livro uma realidade.

Recebi informações ricas na forma de revisões técnicas de um grupo enorme de personalidades. Em particular, Martin Blais e Hugh Brown foram incrivelmente prestativos na melhoria dos exemplos, na clareza e na organização do livro, de ponta a ponta. James Long, Drew Conway, Fernando Pérez, Brian Granger, Thomas Kluyver, Adam Klein, Josh Klein, Chang She e Stéfan van der Walt revisaram individualmente um ou mais capítulos, oferecendo feedbacks específicos de muitos pontos de vista diferentes.

Obtive muitas ideias ótimas para exemplos e conjuntos de dados de amigos e colegas da comunidade de dados, entre eles: Mike Dewar, Jeff Hammerbacher, James Johndrow, Kristian Lum, Adam Klein, Hilary Mason, Chang She e Ashley Williams.

Sem dúvida, devo muito a vários líderes da comunidade científica de código aberto de Python, que construíram as bases para o meu trabalho de desenvolvimento e me incentivaram enquanto eu escrevia este livro: o time do núcleo do IPython (Fernando Pérez, Brian Granger, Min Ragan-Kelly, Thomas Kluyver e outros), John Hunter, Skipper Seabold, Travis Oliphant, Peter Wang, Eric Jones, Robert Kern, Josef Perktold, Francesc Alted, Chris Fonnesbeck e vários outros a ponto de ser impossível mencionar todos. Muitas outras pessoas ofereceram uma boa dose de apoio, ideias e incentivos no caminho: Drew Conway, Sean Taylor, Giuseppe

Paleologo, Jared Lander, David Epstein, John Krowas, Joshua Bloom, Den Pilsworth, John Myles-White e muitos outros que devo ter esquecido.

Também gostaria de agradecer a algumas pessoas presentes em meus anos de formação. Inicialmente, aos meus antigos colegas da AQR, que me incutiram ânimo para o meu trabalho com o pandas ao longo dos anos: Alex Reyfman, Michael Wong, Tim Sargen, Oktay Kurbanov, Matthew Tschantz, Roni Israelov, Michael Katz, Chris Uga, Prasad Ramanan, Ted Square e Hoon Kim. Por fim, aos meus conselheiros acadêmicos Haynes Miller (MIT) e Mike West (Duke).

Recebi ajuda significativa de Phillip Cloud e Joris Van den Bossche em 2014 para atualizar os códigos de exemplos do livro e corrigir outras imprecisões em decorrência de alterações no pandas.

Do lado pessoal, Casey me ofereceu apoio de valor inestimável no dia a dia durante o processo de escrita, tolerando meus altos e baixos enquanto eu compunha a versão preliminar final em cima de um cronograma já sobrecarregado. Por fim, aos meus pais, Bill e Kim, que me ensinaram a correr atrás de meus sonhos e jamais me conformar com menos.

CAPÍTULO 1

Informações preliminares

1.1 De que se trata este livro?

Este livro diz respeito aos detalhes sobre manipulação, processamento, limpeza e extração de informações de dados em Python. Meu objetivo é oferecer um guia para as partes da linguagem de programação Python e o seu ecossistema de bibliotecas orientadas a dados e ferramentas que deixarão você equipado para se tornar um analista de dados eficaz. Embora "análise de dados" esteja no título do livro, o foco está especificamente na programação, nas bibliotecas e nas ferramentas Python, em oposição à metodologia de análise de dados. É a programação Python necessária *para* a análise de dados.

Quais tipos de dados?

Quando digo "dados", a que estou me referindo exatamente? O foco principal está nos *dados estruturados* – um termo propositalmente vago, que engloba muitos formatos diferentes e comuns de dados, por exemplo:

- Dados tabulares ou do tipo planilha, em que cada coluna pode ser de um tipo diferente (string, numérico, data ou outro tipo). Incluem a maior parte dos dados comumente armazenados em bancos de dados relacionais ou em arquivos-texto delimitados com tabulação ou vírgula.
- Arrays multidimensionais (matrizes).
- Várias tabelas de dados inter-relacionadas por colunas de chaves (que seriam as chaves primárias ou estrangeiras para um

usuário de SQL).

- Séries temporais uniformemente espaçadas ou não.

Sem dúvida, essa não é uma lista completa. Embora nem sempre pareça óbvio, uma grande porcentagem dos conjuntos de dados pode ser transformada em um formato estruturado, mais apropriado para análise e modelagem. Se não for, talvez seja possível extrair traços de um conjunto de dados, deixando-os em um formato estruturado. Como exemplo, uma coleção de artigos de notícias poderia ser processada de modo a gerar uma tabela de frequência de palavras, a qual poderia então ser usada para fazer uma análise de sentimentos (sentiment analysis).

Para a maioria dos usuários de programas de planilha, como o Microsoft Excel – talvez a ferramenta de análise de dados mais amplamente utilizada no mundo –, esses tipos de dados não serão desconhecidos.

1.2 Por que Python para análise de dados?

Para muitas pessoas, a linguagem de programação Python tem um forte apelo. Desde o seu surgimento em 1991, Python se tornou uma das linguagens de programação interpretadas mais populares, junto com Perl, Ruby e outras. Python e Ruby se tornaram especialmente populares por volta de 2005 para a construção de sites, com seus diversos frameworks web como Rails (Ruby) e Django (Python). Linguagens como essas com frequência são chamadas de linguagens de *scripting*, pois podem ser usadas para escrever rapidamente pequenos programas ou *scripts* para automatizar outras tarefas. Não gosto do termo “linguagem de scripting”, pois carrega uma conotação de que essas linguagens não poderiam ser usadas para a construção de softwares sérios. Entre as linguagens interpretadas, por diversos motivos históricos e culturais, Python desenvolveu uma comunidade grande e ativa de processamento científico e análise de dados. Nos últimos dez anos, Python passou de uma linguagem de computação científica

inovadora, ou para ser usada “por sua própria conta e risco”, para uma das linguagens mais importantes em ciência de dados, aprendizado de máquina (machine learning) e desenvolvimento de softwares em geral, no ambiente acadêmico e no mercado.

Para análise de dados, processamento interativo e visualização de dados, Python inevitavelmente atrairá comparações com outras linguagens de programação comerciais e de código aberto, e com ferramentas amplamente utilizadas, como R, MATLAB, SAS, Stata e outras. Nos últimos anos, o suporte melhorado de Python para bibliotecas (como o pandas e o scikit-learn) o transformou em uma opção popular para tarefas de análise de dados. Em conjunto com a robustez de Python para uma engenharia de software de propósito geral, é uma excelente opção como uma linguagem principal para a construção de aplicações de dados.

Python como aglutinador

Parte do sucesso de Python no processamento científico deve-se à facilidade de integração com códigos em C, C++ e FORTRAN. A maioria dos ambientes de processamento modernos compartilha um conjunto semelhante de bibliotecas FORTRAN e C legadas para álgebra linear, otimização, integração, transformadas rápidas de Fourier e outros algoritmos desses tipos. A mesma história vale para muitas empresas e laboratórios nacionais, que têm utilizado Python para aglutinar softwares legados existentes há décadas.

Muitos programas são constituídos de pequenas porções de código nas quais a maior parte do tempo é gasta, com grandes porções de “código aglutinador” que não são executadas com frequência. Em muitos casos, o tempo de execução do código aglutinador é insignificante; os esforços serão investidos de modo mais frutífero na otimização dos gargalos de processamento, às vezes passando o código para uma linguagem de nível mais baixo, como C.

Resolvendo o problema de “duas linguagens”

Em muitas empresas, é comum fazer pesquisas, gerar protótipos e testar novas ideias usando uma linguagem de computação mais especializada, como SAS e R, e então, posteriormente, portar essas ideias para que façam parte de um sistema maior de produção, escrito, digamos, em Java, C# ou C++. O que as pessoas percebem cada vez mais é que Python é uma linguagem apropriada não só para pesquisas e prototipagem, mas também para a construção de sistemas de produção. Por que manter dois ambientes de desenvolvimento quando um só seria suficiente? Acredito que mais e mais empresas adotarão esse caminho, pois geralmente há vantagens organizacionais significativas quando tanto os engenheiros pesquisadores quanto os engenheiros de software utilizam o mesmo conjunto de ferramentas de programação.

Por que não Python?

Embora Python seja um ambiente excelente para construir vários tipos de aplicações de análise e sistemas de propósito geral, há alguns usos para os quais Python talvez seja menos apropriado.

Pelo fato de Python ser uma linguagem de programação interpretada, em geral, a maior parte do código Python executará de forma substancialmente mais lenta do que um código escrito em uma linguagem compilada, como Java ou C++. Como o *tempo do programador* com frequência é mais valioso do que o *tempo de CPU*, muitas pessoas ficarão satisfeitas com essa negociação de custo-benefício. No entanto, em uma aplicação com uma latência muito baixa ou com requisitos para utilização de recursos (por exemplo, um sistema de HFT [High-Frequency Trading System, ou Sistema de Negociação de Alta Frequência]), o tempo gasto com programação em uma linguagem de nível mais baixo (mas também com mais baixa produtividade), como C++, para atingir o máximo possível de desempenho poderá ser um tempo bem gasto.

Python pode ser uma linguagem desafiadora para construir aplicações multithreaded (com várias threads) altamente

concorrentes, em particular aplicações com muitas threads limitadas por CPU (CPU-bound). O motivo para isso é ele ter o que é conhecido como GIL (*Global Interpreter Lock*, ou Trava Global do Interpretador) – um mecanismo que evita que o interpretador execute mais de uma instrução Python por vez. As razões técnicas para a existência da GIL estão além do escopo deste livro. Embora seja verdade que, em muitas aplicações de processamento de big data, um cluster de computadores talvez seja necessário para processar um conjunto de dados em um período de tempo razoável, ainda há situações em que um sistema multithreaded, com um único processo, é desejável.

Isso não quer dizer que Python não possa executar verdadeiramente um código multithreaded paralelo. As extensões C para Python que utilizam multithreading nativo (em C ou em C++) podem executar código em paralelo sem sofrer o impacto da GIL, desde que não precisem interagir regularmente com objetos Python.

1.3 Bibliotecas Python essenciais

Para aqueles com menos familiaridade com o ecossistema de dados de Python e as bibliotecas usadas ao longo do livro, apresentarei uma visão geral rápida de algumas delas.

NumPy

A NumPy (*http://www.numpy.org/*) – abreviatura de Numerical Python – tem sido a pedra angular do processamento numérico em Python há muito tempo. Ela oferece o código aglutinador para as estruturas de dados, os algoritmos e a biblioteca necessários à maioria das aplicações científicas que envolvam dados numéricos em Python. Entre outros recursos, a NumPy contém:

- um objeto array multidimensional *ndarray* rápido e eficiente;
- funções para efetuar processamentos em todos os elementos de arrays ou operações matemáticas entre arrays;

- ferramentas para ler e gravar conjuntos de dados baseados em arrays em disco;
- operações de álgebra linear, transformadas de Fourier e geração de números aleatórios;
- uma API C madura para permitir que extensões Python e códigos C ou C++ nativos acessem as estruturas de dados e os recursos de processamento da NumPy.

Além dos recursos de processamento rápido de arrays que a NumPy acrescenta ao Python, um de seus principais usos em análise de dados é como um contêiner para que dados sejam passados entre algoritmos e bibliotecas. Para dados numéricos, os arrays NumPy são mais eficientes para armazenar e manipular dados do que as outras estruturas de dados embutidas (built-in) de Python. Além do mais, as bibliotecas escritas em uma linguagem de mais baixo nível, como C ou Fortran, podem operar em dados armazenados em um array NumPy sem copiar dados para outra representação em memória. Assim, muitas ferramentas de processamento numérico para Python supõem os arrays NumPy como uma estrutura de dados principal ou têm como meta uma interoperabilidade suave com a NumPy.

pandas

O pandas (*http://pandas.pydata.org/*) oferece estruturas de dados de alto nível e funções, projetadas para fazer com que trabalhar com dados estruturados ou tabulares seja rápido, fácil e expressivo. Desde o seu surgimento em 2010, o pandas tem ajudado a viabilizar o Python como um ambiente eficaz e produtivo para análise de dados. Os principais objetos do pandas usados neste livro são o DataFrame – uma estrutura de dados tabular, orientada a colunas, com rótulos (labels) tanto para linhas quanto para colunas – e as Series – um objeto array unidimensional, com rótulo.

O pandas combina as ideias de processamento de alto desempenho de arrays da NumPy com os recursos flexíveis de manipulação de

dados das planilhas e dos bancos de dados relacionais (como o SQL). Ele disponibiliza uma funcionalidade sofisticada de indexação para facilitar a reformatação, a manipulação, as agregações e a seleção de subconjuntos de dados. Como a manipulação, a preparação e a limpeza de dados são habilidades importantes na análise de dados, o pandas será um dos focos principais deste livro.

Como um pequeno histórico, comecei a construir o pandas no início de 2008, durante minha permanência na AQR Capital Management – uma firma de gerenciamento de investimentos quantitativos. Naquela época, eu tinha um conjunto distinto de requisitos que não era devidamente tratado por nenhuma ferramenta única à minha disposição:

- estruturas de dados com eixos nomeados, com suporte para alinhamento automático ou explícito de dados – isso evita erros comuns resultantes de dados desalinhados e possibilita trabalhar com dados indexados de modo diferente, provenientes de origens distintas;
- funcionalidade para séries temporais integradas;
- mesmas estruturas de dados para lidar com séries de dados tanto temporais quanto não temporais;
- operações aritméticas e reduções que preservem metadados;
- tratamento flexível para dados ausentes;
- combinações (merge) e outras operações relacionais que se encontram em bancos de dados populares (baseados em SQL, por exemplo).

Eu queria ser capaz de fazer todas essas tarefas em um só local, de preferência em uma linguagem bem apropriada para um desenvolvimento de software de propósito geral. A linguagem Python era uma boa candidata para isso; no entanto, naquela época, não havia um conjunto integrado de estruturas de dados e ferramentas que oferecesse essas funcionalidades. Como resultado de ter sido inicialmente construído para resolver problemas

financeiros e de análise de negócios, o pandas contém recursos e ferramentas especialmente profundos para séries temporais, bem apropriados para trabalhar com dados indexados por tempo gerados por processos de negócios.

Para usuários da linguagem R para processamento estatístico, o nome DataFrame será conhecido, pois o objeto recebeu seu nome com base no objeto data.frame similar de R. De modo diferente de Python, os data frames estão embutidos na linguagem de programação R e em sua biblioteca-padrão. Como resultado, muitos dos recursos que se encontram no pandas geralmente fazem parte da implementação nuclear de R ou são disponibilizados por meio de pacotes add-on.

O próprio nome pandas é derivado de *panel data* (dados em painel), um termo de econometria para conjuntos de dados multidimensionais estruturados, e uma brincadeira com a própria expressão *Python data analysis* (análise de dados com Python).

matplotlib

A matplotlib (*http://matplotlib.org/*) é a biblioteca Python mais popular para fazer plotagens e gerar outras visualizações de dados bidimensionais. Foi originalmente criada por John D. Hunter e hoje em dia é mantida por uma grande equipe de desenvolvedores. A biblioteca foi projetada para criar plotagens apropriadas para publicação. Embora haja outras bibliotecas de visualização disponíveis aos programadores Python, a matplotlib é a mais amplamente utilizada e, desse modo, tem uma boa integração em geral com o restante do ecossistema. Julgo ser uma opção segura como uma ferramenta de visualização padrão.

IPython e Jupyter

O projeto IPython (*http://ipython.org/*) teve início em 2001 como um projeto secundário de Fernando Pérez para criar um interpretador Python interativo melhor. Nos dezesseis anos subsequentes, passou

a ser uma das ferramentas mais importantes da pilha moderna de dados Python. Embora não ofereça nenhuma ferramenta de processamento ou de análise de dados por si só, o IPython foi projetado desde o início para maximizar a sua produtividade, tanto em um processamento interativo, quanto no desenvolvimento de software. Ele incentiva um fluxo de trabalho de *execução-exploração*, em vez do fluxo de trabalho típico de *edição-compilação-execução* de muitas outras linguagens de programação. Também oferece acesso fácil ao shell de seu sistema operacional e ao sistema de arquivos. Como boa parte da programação na análise de dados envolve exploração, tentativa e erro e iteração, o IPython pode ajudar você a executar suas tarefas mais rapidamente.

Em 2014, Fernando e a equipe do IPython anunciaram o projeto Jupyter (*http://jupyter.org/*): uma iniciativa mais ampla para projetar ferramentas de processamento interativas, independentes da linguagem. O notebook web do IPython passou a ser o notebook Jupyter, com suporte hoje em dia para mais de 40 linguagens de programação. O sistema IPython agora pode ser utilizado como um *kernel* (um modo de linguagem de programação) para usar Python com o Jupyter.

O próprio IPython passou a ser um componente do projeto Jupyter de código aberto, muito mais amplo, que oferece um ambiente produtivo para processamento interativo e exploratório. Seu “modo” mais antigo e mais simples é como um shell Python aperfeiçoado, projetado para agilizar a escrita, os testes e a depuração de código Python. Você também pode usar o sistema IPython por meio do Jupyter Notebook – um código de “notebook” interativo, baseado em web, que oferece suporte para dezenas de linguagens de programação. O shell IPython e os notebooks Jupyter são particularmente convenientes para exploração e visualização de dados.

O sistema de notebooks Jupyter também permite criar conteúdo em Markdown e em HTML, oferecendo a você um meio para criar

documentos ricos, com código e texto. Outras linguagens de programação também implementaram kernels para o Jupyter a fim de possibilitar o uso de linguagens diferentes de Python no Jupyter.

Para mim, pessoalmente, o IPython em geral está envolvido na maioria de meus trabalhos com Python, incluindo execução, depuração e teste de código.

No material que acompanha este livro (*https://github.com/wesm/pydata-book*), você encontrará notebooks Jupyter contendo todos os exemplos de código de cada capítulo.

SciPy

O SciPy (*https://scipy.org/*) é uma coleção de pacotes voltada para uma série de diversos domínios de problemas padrões no processamento científico. Eis uma amostra dos pacotes incluídos:

scipy.integrate
: Rotinas de integração numérica e solucionadores de equações diferenciais.

scipy.linalg
: Rotinas para álgebra linear e decomposições de matrizes que se estendem para além dos recursos oferecidos por numpy.linalg.

scipy.optimize
: Otimizadores de funções (minimizadores) e algoritmos para encontrar raízes (root finding algorithms).

scipy.signal
: Ferramentas para processamento de sinais.

scipy.sparse
: Solucionadores de matrizes esparsas e sistemas lineares esparsos.

scipy.special
: Wrapper em torno da SPECFUN – uma biblioteca Fortran que

implementa muitas funções matemáticas comuns, por exemplo, a função gamma.

scipy.stats
: Distribuições padrões de probabilidade contínuas e discretas (funções de densidade, amostragens, funções de distribuição contínua), vários testes estatísticos e estatísticas mais descritivas.

Juntas, a NumPy e o SciPy compõem uma base de processamento razoavelmente completa e madura para muitas aplicações tradicionais de processamento científico.

scikit-learn

Desde a criação do projeto em 2010, o scikit-learn (*http://scikit-learn.org/stable/*) se transformou no principal kit de ferramentas de propósito geral para aprendizado de máquina dos programadores Python. Em apenas sete anos, teve mais de 1.500 colaboradores em todo o mundo. O scikit-learn inclui submódulos para módulos como:

- Classificação: SVM, vizinhos mais próximos (nearest neighbors), floresta aleatória (random forest), regressão logística etc.
- Regressão: regressão de Lasso, regressão de ridge etc.
- Clustering: *k*-means, clustering espectral etc.
- Redução de dimensionalidade: PCA, seleção de atributos, fatoração de matrizes etc.
- Seleção de modelos: grid search (busca em grade), validação cruzada, métricas.
- Pré-processamento: extração de atributos, normalização.

Junto com o pandas, o statsmodels e o IPython, o scikit-learn tem sido crucial para possibilitar que Python seja uma linguagem de programação produtiva para a ciência de dados. Embora não possa incluir um guia completo para o scikit-learn neste livro, farei uma rápida introdução a alguns de seus modelos e como usá-los com as

outras ferramentas apresentadas no livro.

statsmodels

O statsmodels (*http://statsmodels.org/*) é um pacote de análise estatística que se originou a partir do trabalho do professor de estatística Jonathan Taylor, da Universidade de Stanford; ele implementou uma série de modelos de análise de regressão populares na linguagem de programação R. Skipper Seabold e Josef Perktold criaram formalmente o novo projeto statsmodels em 2010, e, desde então, expandiram o projeto incluindo uma massa crítica de usuários e colaboradores engajados. Nathaniel Smith desenvolveu o projeto Patsy, que oferece um framework para especificação de fórmulas ou de modelos ao statsmodels, inspirado no sistema de fórmulas do R.

Quando comparado com o scikit-learn, o statsmodels contém algoritmos para estatística clássica (principalmente frequentista) e econometria. Inclui submódulos como:

- Modelos de regressão: regressão linear, modelos lineares generalizados, modelos lineares robustos, modelos lineares de efeitos mistos etc.;
- Análise de variância (ANOVA);
- Análise de séries temporais: AR, ARMA, ARIMA, VAR e outros modelos;
- Métodos não paramétricos: estimativa de densidade Kernel, regressão de kernel;
- Visualização de resultados de modelos estatísticos.

O statsmodels tem um foco maior em inferência estatística, oferecendo estimativas de incerteza e valores *p* para parâmetros. Em oposição, o scikit-learn está mais centrado em previsões.

Como no caso do scikit-learn, farei uma introdução rápida ao statsmodels e mostrarei como usá-lo com a NumPy e o pandas.

1.4 Instalação e configuração

Como todos utilizam Python para aplicações diferentes, não há uma única solução para configurá-lo e configurar os pacotes add-on necessários. Muitos leitores não terão um ambiente de desenvolvimento Python completo apropriado para acompanhar este livro, portanto darei aqui instruções detalhadas para efetuar a configuração em cada sistema operacional. Recomendo o uso da distribuição gratuita Anaconda. Atualmente (quando escrevi este livro), o Anaconda é oferecido nas versões Python 2.7 e 3.6, embora isso possa mudar em algum momento no futuro. Este livro utiliza Python 3.6, e incentivo você a usar essa versão ou uma mais recente.

Windows

Para começar a trabalhar no Windows, faça o download do instalador do Anaconda (*https://www.anaconda.com/download/*). Recomendo seguir as instruções de instalação para Windows disponíveis na página de download do Anaconda, as quais podem ter sido alteradas entre o momento em que este livro foi publicado e a sua leitura.

Vamos agora verificar se tudo está configurado corretamente. Para abrir a aplicação Command Prompt (também conhecida como *cmd.exe*), clique com o botão direito do mouse no menu Start (Iniciar) e selecione Command Prompt. Experimente iniciar o interpretador Python digitando **python**. Você deverá ver uma mensagem correspondente à versão do Anaconda instalada:

```
C:\Users\wesm>python
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 5 2016, 11:41:13)
[MSC v.1900 64 bit (AMD64)] on win32
>>>
```

Para sair do shell, pressione Ctrl-D (no Linux ou no MacOS), Ctrl-Z (no Windows) ou digite o comando **exit()** e tecle Enter.

Apple (OS X, MacOS)

Faça download do instalador do Anaconda para OS X, que deverá ter um nome como *Anaconda3-4.1.0-MacOSX-x86_64.pkg*. Dê um clique duplo no arquivo *.pkg* para executar o instalador. Ao executar, ele concatenará automaticamente o path do executável do Anaconda em seu arquivo *.bash_profile*. Ele está localizado em */Users/$USER/.bash_profile*.

Para conferir se tudo está funcionando, experimente iniciar o IPython no shell do sistema (abra a aplicação de Terminal a fim de obter um prompt de comandos):

```
$ ipython
```

Para sair do shell, pressione Ctrl-D ou digite **exit()** e tecle Enter.

GNU/Linux

Os detalhes sobre o Linux variarão um pouco conforme a sua variante de Linux, mas apresentarei nesta seção os detalhes de distribuições como Debian, Ubuntu, CentOS e Fedora. A configuração é semelhante à do OS X, com exceção do modo como o Anaconda é instalado. O instalador é um shell script que deve ser executado no terminal. Dependendo do fato de ter um sistema de 32 bits ou de 64 bits, você terá que utilizar o instalador x86 (para 32 bits) ou o x86_64 (64 bits). Então terá um arquivo cujo nome será algo como *Anaconda3-4.1.0-Linux-x86_64.sh*. Para instalá-lo, execute o script a seguir com o bash:

```
$ bash Anaconda3-4.1.0-Linux-x86_64.sh
```

Algumas distribuições de Linux têm versões com todos os pacotes Python necessários em seus gerenciadores de pacotes e poderão ser instaladas usando uma ferramenta como o apt. A configuração descrita nesta seção utiliza o Anaconda, pois ela pode ser facilmente reproduzida entre as distribuições, e é mais fácil fazer upgrade dos pacotes para suas versões mais recentes.

Depois de aceitar a licença, você verá uma opção para escolher o

local em que os arquivos do Anaconda serão colocados. Recomendo instalar os arquivos no local default em seu diretório home – por exemplo, */home/$USER/anaconda* (com o seu nome de usuário, naturalmente).

O instalador do Anaconda poderá perguntar se você deseja inserir o seu diretório *bin/* antes de sua variável *$PATH*. Se tiver algum problema após a instalação, poderá fazer isso por conta própria modificando o seu *.bashrc* (ou o *.zshrc*, caso esteja usando o shell zsh) com algo semelhante a:

```
export PATH=/home/$USER/anaconda/bin:$PATH
```

Feito isso, você poderá iniciar um novo processo de terminal ou executar o seu *.bashrc* novamente com *source ~/.bashrc*.

Instalando ou atualizando pacotes Python

Em algum ponto da leitura, talvez você queira instalar pacotes Python adicionais, não incluídos na distribuição do Anaconda. Em geral, é possível instalá-los com o comando a seguir:

```
conda install nome_do_pacote
```

Se isso não funcionar, talvez você possa também instalar o pacote usando a ferramenta pip de gerenciamento de pacotes:

```
pip install nome_do_pacote
```

Os pacotes podem ser atualizados com o comando conda update:

```
conda update nome_do_pacote
```

O pip também trata upgrades por meio da flag --upgrade:

```
pip install --upgrade nome_do_pacote
```

Você terá várias oportunidades para experimentar esses comandos ao longo do livro.

Embora seja possível usar tanto o conda quanto o pip para instalar pacotes, você não deverá tentar atualizar pacotes conda com o pip, pois fazer isso talvez resulte em problemas no ambiente. Quando usar o Anaconda ou o Miniconda, será melhor tentar inicialmente a atualização com o conda.

Python 2 e Python 3

A primeira versão de interpretadores da linha Python 3.x foi lançada no final de 2008. Ela incluía uma série de mudanças que deixaram alguns códigos anteriormente escritos com Python 2.x incompatíveis. Como já haviam se passado 17 anos desde a primeira versão de Python em 1991, criar uma versão Python 3 que provocasse uma “ruptura” era visto com bons olhos pela maioria, considerando as lições aprendidas naquele período.

Em 2012, boa parte da comunidade de análise de dados e da comunidade científica continuava usando Python 2.x, pois muitos pacotes ainda não estavam totalmente compatíveis com Python 3. Assim, a primeira edição deste livro utilizava Python 2.7. Atualmente os usuários têm liberdade para escolher entre Python 2.x e 3.x e, em geral, têm suporte completo das bibliotecas para qualquer que seja a variante.

No entanto, Python 2.x alcançará o final de sua vida de desenvolvimento em 2020 (incluindo patches críticos de segurança); portanto, não é uma boa ideia iniciar novos projetos com Python 2.7. Desse modo, este livro utiliza Python 3.6 – uma versão estável, amplamente implantada, com bom suporte. Começamos a chamar Python 2.x de “Python Legado”, e Python 3.x simplesmente de “Python”. Incentivo você a fazer o mesmo.

Este livro utiliza Python 3.6 como base. Sua versão de Python talvez seja mais recente que a versão 3.6, mas os códigos de exemplo deverão ser compatíveis com versões mais recentes. Alguns códigos de exemplo poderão funcionar de modo diferente ou poderão nem funcionar com Python 2.7.

Ambientes de desenvolvimento integrado (IDEs) e editores de texto

Quando me perguntam sobre o meu ambiente de desenvolvimento padrão, quase sempre digo: “IPython mais um editor de texto”. Geralmente escrevo um programa e testo e depuro cada parte dele de forma iterativa no IPython ou em notebooks Jupyter. Também é conveniente ser capaz de lidar com os dados interativamente e conferir visualmente se um conjunto em particular de manipulações de dados está fazendo a tarefa correta. Bibliotecas como pandas e NumPy foram projetadas para serem fáceis de usar no shell.

Ao construir um software, porém, alguns usuários talvez prefiram utilizar um IDE mais rico em recursos, em vez de um editor de texto comparativamente primitivo, como o Emacs ou o Vim. Eis alguns que você poderá explorar:

- PyDev (gratuito): um IDE incluído na plataforma Eclipse;
- PyCharm da JetBrains (exige inscrição para usuários comerciais e é gratuito para desenvolvedores de código aberto);
- Python Tools para Visual Studio (para usuários Windows);
- Spyder (gratuito): um IDE que atualmente acompanha o Anaconda;
- Komodo IDE (comercial).

Por causa da popularidade de Python, a maioria dos editores de texto, como Atom e Sublime Text 2, tem excelente suporte para essa linguagem.

1.5 Comunidade e conferências

Além das pesquisas na internet, as diversas listas de discussão científicas e relacionadas a dados para Python geralmente são úteis e oferecem respostas às perguntas. Algumas que vale a pena conferir incluem:

- pydata: uma lista do Google Group para perguntas relacionadas

a Python para análise de dados e ao pandas;

- pystatsmodels: para perguntas relacionadas ao statsmodels ou ao pandas;
- lista de discussão para o scikit-learn (*scikit-learn@python.org*) e aprendizado de máquina em Python, de modo geral;
- numpy-discussion: para perguntas relacionadas à NumPy;
- scipy-user: para perguntas gerais sobre o SciPy ou sobre processamento científico com Python.

Não postei propositalmente os URLs dessas listas para o caso de eles mudarem. As listas podem ser facilmente localizadas com uma pesquisa na internet.

Todo ano, muitas conferências para programadores Python são realizadas em todo o mundo. Se quiser entrar em contato com outros programadores Python que compartilhem de seus interesses, incentivo você a participar de uma, se for possível. Muitas conferências têm ajuda financeira disponível para aqueles que não podem pagar a inscrição ou viajar até o local da conferência.

Eis algumas a serem consideradas:

- PyCon e EuroPython: são as duas principais conferências Python de caráter geral, na América do Norte e na Europa, respectivamente;
- SciPy e EuroSciPy: conferências voltadas ao processamento científico, na América do Norte e na Europa, respectivamente;
- PyData: uma série de conferências regionais no mundo todo, cujo alvo são os casos de uso relacionados à ciência de dados e à análise de dados;
- Conferências PyCon internacionais e regionais (acesse *http://pycon.org* para ver uma lista completa).

1.6 Navegando pelo livro

Se você nunca programou em Python antes, vai querer investir um

tempo nos Capítulos 2 e 3, nos quais apresento um tutorial condensado sobre os recursos da linguagem Python, o shell IPython e os notebooks Jupyter. São informações consideradas como pré-requisitos para o restante do livro. Se já tem experiência com Python, talvez opte por passar os olhos rapidamente por esses capítulos ou ignorá-los.

A seguir, farei uma rápida introdução aos recursos essenciais da NumPy, deixando o seu uso mais avançado para o Apêndice A. Então apresentarei o pandas e dedicarei o restante do livro aos tópicos de análise de dados aplicando o pandas, a NumPy e a matplotlib (para visualização). Estruturei o material do modo mais incremental possível, embora, ocasionalmente, haja um pouco de cruzamento entre os capítulos, com alguns casos isolados em que os conceitos são usados sem que tenham sido necessariamente apresentados antes.

Embora os leitores possam ter muitos objetivos finais diferentes para seus trabalhos, as tarefas exigidas geralmente se enquadram em uma série de grupos amplos distintos:

Interação com o mundo externo
: Ler e escrever usando uma variedade de formatos de arquivos e repositórios de dados.

Preparação
: Limpar, manipular, combinar, normalizar, reformatar, tratar e transformar os dados para análise.

Transformação
: Aplicar operações matemáticas e estatísticas em grupos de conjuntos de dados a fim de obter novos conjuntos de dados (por exemplo, agregando uma tabela grande por meio de variáveis de grupo).

Modelagem e processamento
: Conectar seus dados a modelos estatísticos, algoritmos de

aprendizado de máquina ou outras ferramentas de processamento.

Apresentação

Criar visualizações gráficas interativas ou estáticas, ou sínteses textuais.

Exemplos de código

A maior parte dos códigos de exemplo do livro é exibida com entradas e saídas, como apareceriam se fossem executados no shell IPython ou em notebooks Jupyter:

```
In [5]: CÓDIGO DE EXEMPLO
Out[5]: SAÍDA
```

Quando vir um código de exemplo como esse, a intenção é que você digite o código do bloco `In` em seu ambiente de programação e execute-o pressionando a tecla Enter (ou Shift-Enter no Jupyter). Você deverá ver uma saída semelhante àquela mostrada no bloco `Out`.

Dados para os exemplos

Os conjuntos de dados dos exemplos de cada capítulo estão em um repositório do GitHub (*https://github.com/wesm/pydata-book)*. É possível fazer o download desses dados usando o sistema de controle de versões Git na linha de comando ou fazendo o download de um arquivo zip do repositório a partir do site. Se encontrar problemas, acesse o meu site (*http://wesmckinney.com/*) a fim de ver instruções atualizadas sobre como obter o material associado ao livro.

Investi todos os esforços possíveis para garantir que tudo que for necessário para reproduzir os exemplos fosse contemplado, mas posso ter cometido alguns erros ou feito algumas omissões. Se for esse o caso, por favor, envie um email para mim (em inglês): *book@wesmckinney.com*. A melhor forma de informar erros sobre o livro é na página de errata no site da O'Reilly (*http://bit.ly/pyDataAnalysis_errata*) ou na página do livro no site da

Novatec.

Convenções de importação

A comunidade Python adotou uma série de convenções de nomenclatura para os módulos comumente utilizados:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import statsmodels as sm
```

Isso significa que quando você vir `np.arange`, essa será uma referência à função `arange` da NumPy. Fazemos isso porque, no desenvolvimento de software em Python, importar tudo (`from numpy import *`) de um pacote grande como no caso da NumPy é considerado uma prática ruim.

Jargão

Utilizarei alguns termos comuns, tanto para programação quanto para a ciência de dados, com os quais talvez você não tenha familiaridade. Portanto eis algumas definições rápidas:

Manipular/manipulação/tratamento (Munge/munging/wrangling)

Descreve o processo geral de manipulação de dados não estruturados e/ou desorganizados, convertendo-os em um formato estruturado ou limpo. A palavra entrou para o jargão de muitos hackers de dados modernos. “Munge” rima com “grunge”.

Pseudocódigo

Descrição de um algoritmo ou processo que assume um formato semelhante ao de um código, embora, provavelmente, não seja um código-fonte de fato válido.

Açúcar sintático (syntactic sugar)

Sintaxe de programação que não acrescenta novas funcionalidades, mas deixa o código mais conveniente ou mais

fácil de digitar.

CAPÍTULO 2

Básico da linguagem Python, IPython e notebooks Jupyter

Quando escrevi a primeira edição deste livro em 2011 e 2012, havia menos recursos disponíveis para aprender a fazer análise de dados em Python. Era parcialmente um problema do ovo e da galinha; muitas bibliotecas que atualmente tomamos como dadas, incluindo, por exemplo, pandas, scikit-learn e statsmodels, eram imaturas na época, se comparadas com as versões de hoje em dia. Atualmente, em 2017, há uma literatura cada vez mais ampla sobre ciência de dados, análise de dados e aprendizado de máquina (machine learning), suplementando os trabalhos anteriores sobre processamento científico de propósito geral voltados para cientistas de computação, físicos e profissionais de outras áreas de pesquisa. Há também livros excelentes para conhecer a própria linguagem de programação Python e tornar-se um engenheiro de software eficaz.

Como o propósito deste livro é ser um texto introdutório para trabalhar com dados em Python, acho importante apresentar uma visão geral autocontida de alguns dos recursos mais importantes das estruturas de dados embutidas de Python e das bibliotecas, do ponto de vista da manipulação de dados. Assim, apresentarei, de modo geral, neste capítulo e no Capítulo 3, apenas informações suficientes para permitir que você acompanhe o restante do livro.

Em minha opinião, *não* é necessário tornar-se proficiente em construir bons softwares em Python para ser capaz de fazer análises de dados de forma produtiva. Incentivo você a utilizar o shell IPython e os notebooks Jupyter para fazer experimentos com

os códigos de exemplos e explorar a documentação dos diversos tipos, funções e métodos. Embora eu tenha aplicado meus melhores esforços para apresentar o material do livro de forma incremental, ocasionalmente você poderá deparar com informações que ainda não tenham sido totalmente apresentadas.

Boa parte deste livro foca nas ferramentas de análise de dados baseados em tabelas e de preparação de dados para trabalhar com conjuntos de dados grandes. Para utilizar essas ferramentas, com frequência você deverá fazer alguma preparação dos dados a fim de converter os dados desorganizados em um formato tabular (ou *estruturado*) mais elegante. Felizmente Python é uma linguagem ideal para deixar seus dados em forma com rapidez. Quanto maior a sua facilidade com a linguagem Python, mais fácil será para você preparar novos conjuntos de dados para análise.

Algumas das ferramentas deste livro serão mais bem exploradas a partir de uma sessão ativa de IPython ou do Jupyter. Depois que aprender a iniciar o IPython e o Jupyter, recomendo que você acompanhe os exemplos para que possa fazer experimentos e tentar opções diferentes. Como ocorre com qualquer ambiente do tipo console, guiado pelo teclado, treinar a memória para usar comandos comuns também faz parte da curva de aprendizado.

Há conceitos introdutórios de Python que este capítulo não aborda, como classes e programação orientada a objetos, os quais talvez você ache convenientes em suas incursões pela análise de dados em Python.

Para aprofundar seu conhecimento sobre a linguagem Python, recomendo que suplemente este capítulo com o tutorial oficial de Python (*https://docs.python.org*) e, possivelmente, com um dos muitos livros excelentes sobre programação Python de propósito geral. Algumas recomendações para você começar incluem:

- *Python Cookbook, terceira edição, de David Beazley e Brian K. Jones (Novatec);*
- *Python fluente, de Luciano Ramalho (Novatec);*
- *Python eficaz, de Brett Slatkin (Novatec).*

2.1 Interpretador Python

Python é uma linguagem *interpretada*. O interpretador Python roda um programa executando uma instrução por vez. O interpretador Python interativo padrão pode ser chamado na linha de comando com o comando python:

```
$ python
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print(a)
5
```

O >>> acima é o *prompt* no qual você digitará as expressões de código. Para sair do interpretador Python e voltar ao prompt de comandos, digite exit() ou pressione Ctrl-D.

Executar programas Python é simples e basta chamar python com um arquivo *.py* como seu primeiro argumento. Suponha que tivéssemos criado *hello_world.py* com o conteúdo a seguir:

```
print('Hello world')
```

É possível executá-lo com o seguinte comando (o arquivo *hello_world.py* deve estar no seu diretório atual de trabalho do terminal):

```
$ python hello_world.py
Hello world
```

Embora alguns programadores Python executem todos os seus códigos Python dessa forma, aqueles que efetuam análise de dados ou processamentos científicos utilizam o IPython, que é um interpretador Python melhorado, ou os notebooks Jupyter, que são notebooks de código baseados em web, originalmente criados no projeto IPython. Apresentarei uma introdução ao uso de IPython e de Jupyter neste capítulo, e incluí uma descrição mais detalhada das funcionalidades do IPython no Apêndice B. Quando o comando %run for utilizado, o IPython executará o código do arquivo

especificado no mesmo processo, permitindo que você explore os resultados interativamente quando estiverem prontos:

```
$ ipython
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]: %run hello_world.py
Hello world

In [2]:
```

O prompt default do IPython adota o estilo numerado `In [2]:`, em comparação com o prompt padrão `>>>`.

2.2 Básico sobre o IPython

Nesta seção prepararemos você para usar o shell IPython e o notebook Jupyter, e apresentaremos alguns dos conceitos essenciais.

Executando o shell IPython

Você pode iniciar o shell IPython na linha de comando, exatamente como iniciaria o interpretador Python usual, porém isso é feito com o comando `ipython`:

```
$ ipython
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
```

```
object? -> Details about 'object', use 'object??' for extra details.

In [1]: a = 5

In [2]: a
Out[2]: 5
```

É possível executar instruções Python arbitrárias digitando-as e teclando Return (ou Enter). Se você digitar somente uma variável no IPython, uma representação em string do objeto será apresentada:

```
In [5]: import numpy as np

In [6]: data = {i : np.random.randn() for i in range(7)}

In [7]: data
Out[7]:
{0: -0.20470765948471295,
 1: 0.47894333805754824,
 2: -0.5194387150567381,
 3: -0.55573030434749,
 4: 1.9657805725027142,
 5: 1.3934058329729904,
 6: 0.09290787674371767}
```

As duas primeiras linhas são instruções de código Python; a segunda instrução cria uma variável chamada `data` que referencia um dicionário Python recém-criado. A última linha exibe o valor de `data` no console.

Muitos tipos de objetos Python são formatados para que sejam mais legíveis, ou são exibidos de forma elegante (*pretty-printed*), de modo distinto da exibição usual com `print`. Se você exibir a variável anterior `data` no interpretador Python padrão, ela será bem menos legível:

```
>>> from numpy.random import randn
>>> data = {i : randn() for i in range(7)}
>>> print(data)
{0: -1.5948255432744511, 1: 0.10569006472787983, 2: 1.972367135977295,
3: 0.15455217573074576, 4: -0.24058577449429575, 5: -1.2904897053651216,
6: 0.3308507317325902}
```

O IPython também disponibiliza recursos para executar blocos de

código arbitrários (por meio de uma abordagem, de certo modo glorificada, de copiar e colar) e scripts Python inteiros. O notebook Jupyter também pode ser usado para trabalhar com blocos de código maiores, como veremos em breve.

Executando o notebook Jupyter

Um dos principais componentes do projeto Jupyter é o *notebook*: um tipo de documento interativo para código, texto (com ou sem marcação), visualizações de dados e outras saídas. O notebook Jupyter interage com *kernels*, que são implementações do protocolo de processamento interativo do Jupyter em várias linguagens de programação. O kernel do Jupyter para Python utiliza o sistema IPython para o seu comportamento subjacente.

Para iniciar o Jupyter, execute o comando `jupyter notebook` em um terminal:

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from local directory:
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server and shut down
all kernels (twice to skip confirmation).
Created new window in existing browser session.
```

Em muitas plataformas, o Jupyter será aberto automaticamente em seu navegador web default (a menos que você o inicie com `--no-browser`). Caso contrário, você poderá navegar para o endereço HTTP exibido quando o notebook foi iniciado, o qual, nesse caso, é `http://localhost:8888/`. Veja a Figura 2.1 para saber como é a aparência disso no Google Chrome.

Muitas pessoas usam o Jupyter como um ambiente de processamento local, mas ele também pode ser implantado em servidores e acessado remotamente. Não discutirei esses detalhes neste capítulo, mas incentivo você a explorar esse assunto na internet caso seja relevante para suas necessidades.

Figura 2.1 – Página de entrada do notebook Jupyter.

Para criar um novo notebook, clique no botão New (Novo) e selecione a opção “Python 3” ou “conda [default]”. Você deverá ver algo semelhante à Figura 2.2. Se essa é a sua primeira vez, experimente clicar na “célula” de código vazia e insira uma linha de código Python. Em seguida, pressione Shift-Enter para executá-la.

Quando salvar o notebook (veja “Save and Checkpoint” (Salvar e ponto de verificação) no menu File (Arquivo) do notebook), um arquivo com a extensão *.ipynb* será criado. É um formato de arquivo autocontido, com todo o conteúdo (incluindo qualquer saída de código avaliada) que estiver no notebook no momento. O arquivo pode ser carregado e editado por outros usuários do Jupyter. Para carregar um notebook existente, coloque o arquivo no mesmo diretório em que você iniciou o processo do notebook (ou em uma subpasta dentro dele) e, em seguida, dê um clique duplo em seu

nome na página de entrada. Você pode fazer testes com os notebooks de meu repositório *wesm/pydata-book* no GitHub. Veja a Figura 2.3.

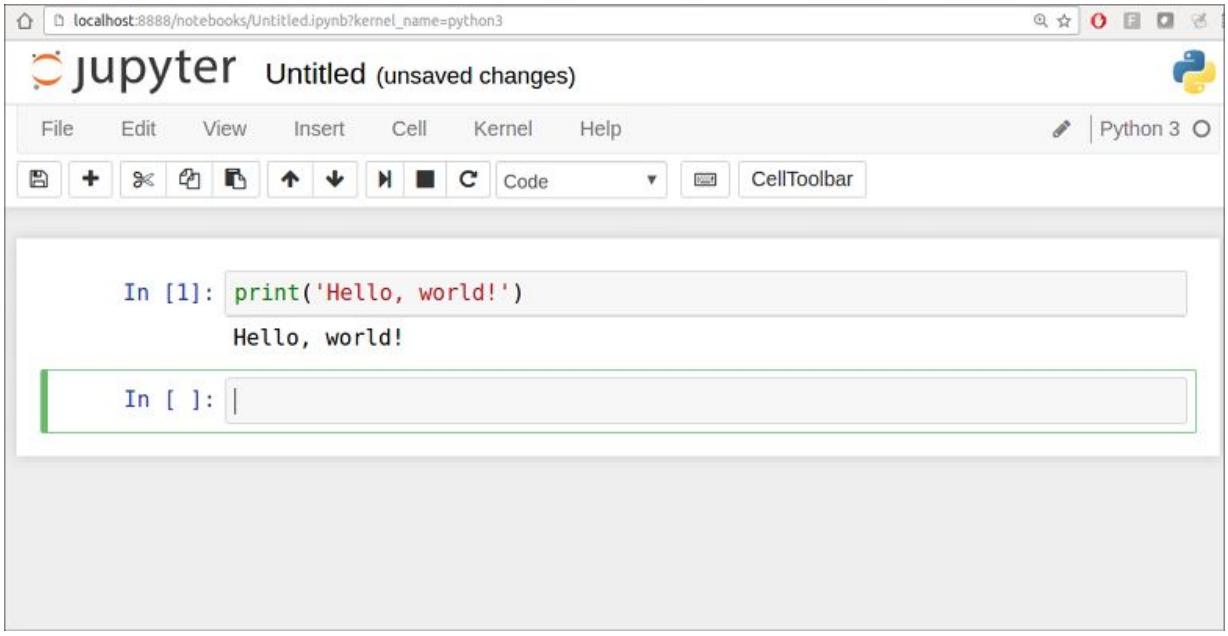


Figura 2.2 – Visão de um novo notebook Jupyter.

Embora o notebook Jupyter possa dar a impressão de ser uma experiência distinta do shell IPython, quase todos os comandos e ferramentas deste capítulo poderão ser usados em qualquer um dos ambientes.

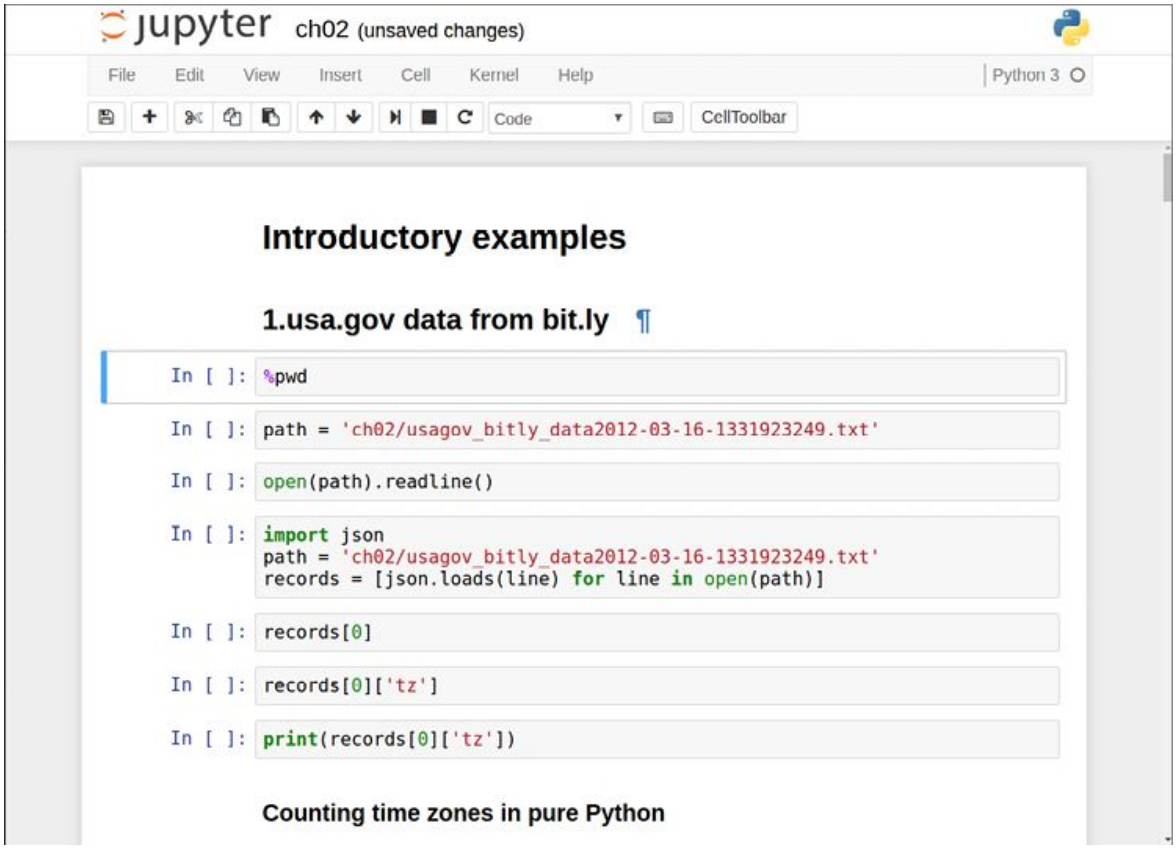


Figura 2.3 – Visualização de um exemplo do Jupyter para um notebook existente.

Preenchimento automático com tabulação

Superficialmente o shell IPython se parece com uma versão cosmeticamente diferente do interpretador Python padrão de terminal (chamado com python). Uma das principais melhorias em relação ao shell Python padrão é o *preenchimento automático com tabulação*, que se encontra em muitos IDEs ou em outros ambientes interativos de análise de processamento. Enquanto estiver inserindo expressões no shell, pressionar a tecla Tab fará uma busca no namespace para quaisquer variáveis (objetos, funções etc.) que correspondam aos caracteres que você tenha digitado até então:

```
In [1]: an_apple = 27

In [2]: an_example = 42

In [3]: an<Tab>
an_apple and an_example any
```

Nesse exemplo, observe que o IPython exibiu tanto as duas variáveis que defini como também a palavra reservada and e a função embutida any de Python. Naturalmente você também pode completar métodos e atributos em qualquer objeto depois de digitar um ponto:

```
In [3]: b = [1, 2, 3]

In [4]: b.<Tab>
b.append b.count b.insert b.reverse
b.clear b.extend b.pop b.sort
b.copy b.index b.remove
```

O mesmo vale para os módulos:

```
In [1]: import datetime

In [2]: datetime.<Tab>
datetime.date datetime.MAXYEAR datetime.timedelta
datetime.datetime datetime.MINYEAR datetime.timezone
datetime.datetime_CAPI datetime.time datetime.tzinfo
```

No notebook Jupyter e em versões mais recentes do IPython (versões 5.0 e mais recentes), os dados para preenchimento automático são exibidos em uma caixa suspensa em vez de serem uma saída textual.

Observe que o IPython, por padrão, oculta métodos e atributos que comecem com underscores, como os métodos mágicos e os métodos "privados" internos e atributos, a fim de evitar sobrecarregar o display (e confundir os usuários iniciantes!). Você também poderá ver esses dados preenchidos com a tecla de tabulação, mas deve antes digitar um underscore para vê-los. Se preferir ver sempre esses métodos no preenchimento com tabulação, essa configuração poderá ser alterada no IPython. Consulte a documentação do IPython para descobrir como fazer isso.

O preenchimento automático com tabulação funciona em muitos contextos além da pesquisa no namespace interativo ou do preenchimento de atributos de objetos ou de módulos. Ao digitar qualquer informação que se pareça com um path de arquivo

(mesmo em uma string Python), pressionar a tecla Tab resultará em um preenchimento com qualquer dado correspondente ao que você digitou, e que esteja no sistema de arquivos de seu computador.

```
In [7]: datasets/movielens/<Tab>
datasets/movielens/movies.dat datasets/movielens/README
datasets/movielens/ratings.dat datasets/movielens/users.dat

In [7]: path = 'datasets/movielens/<Tab>
datasets/movielens/movies.dat datasets/movielens/README
datasets/movielens/ratings.dat datasets/movielens/users.dat
```

Em conjunto com o comando %run (veja “Comando %run” a seguir), essa funcionalidade poderá fazer você economizar muitos pressionamentos de tecla.

Outra área em que o preenchimento com tabulação economiza tempo é no preenchimento de argumentos nomeados (keyword arguments) de funções (incluindo o sinal =). Veja a Figura 2.4.

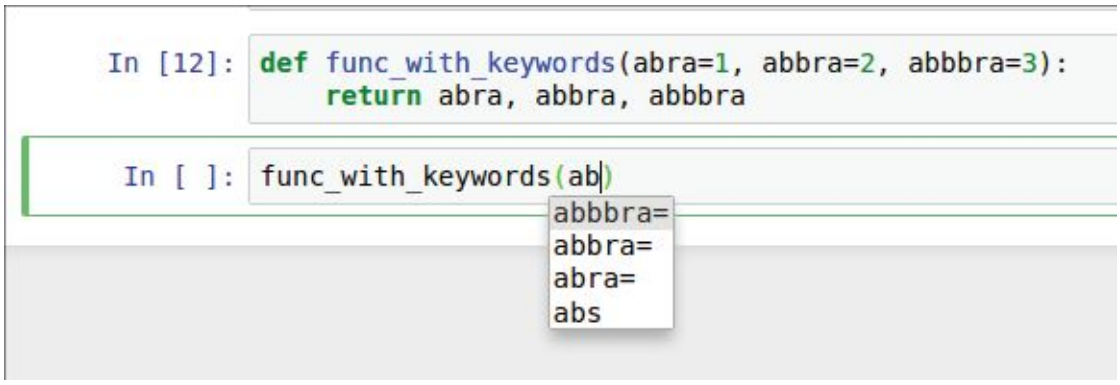


Figura 2.4 – Preenchimento automático de argumentos nomeados de função no notebook Jupyter.

Veremos as funções com mais detalhes em breve.

Introspecção

Usar um ponto de interrogação (?) antes ou depois de uma variável exibirá algumas informações gerais sobre o objeto:

```
In [8]: b = [1, 2, 3]

In [9]: b?
```

```
Type: list
String Form:[1, 2, 3]
Length: 3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items

In [10]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type: builtin_function_or_method
```

Isso é chamado de *introspecção de objeto* (object introspection). Se o objeto for uma função ou um método de instância, a docstring, se estiver definida, também será exibida. Suponha que tenhamos escrito a função a seguir (que você poderá reproduzir no IPython ou no Jupyter):

```
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
```

Então usar ? fará a docstring ser mostrada:

```
In [11]: add_numbers?
Signature: add_numbers(a, b)
Docstring:
Add two numbers together
```

```
Returns
-------
the_sum : type of arguments
File: <ipython-input-9-6a548a216e27>
Type: function
```

Usar ?? fará o código-fonte da função ser mostrado também, se for possível:

```
In [12]: add_numbers??
Signature: add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
File: <ipython-input-9-6a548a216e27>
Type: function
```

? tem um último uso, que é pesquisar o namespace IPython, de modo semelhante à linha de comando padrão do Unix ou do Windows. Uma série de caracteres combinada com o caractere-curinga (*) mostrará todos os nomes que correspondam à expressão com o caractere-curinga. Por exemplo, poderíamos obter uma lista de todas as funções no namespace de mais alto nível do NumPy que contenham load:

```
In [13]: np.*load*?
np.__loader__
np.load
np.loads
np.loadtxt
np.pkgload
```

Comando %run

Você pode executar qualquer arquivo como um programa Python no

ambiente de sua sessão IPython usando o comando %run. Suponha que tenha o script simples a seguir armazenado em *ipython_script_test.py*:

```
def f(x, y, z):
    return (x + y) / z

a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

Você pode executar esse código passando o nome do arquivo para %run:

```
In [14]: %run ipython_script_test.py
```

O script é executado em um *namespace vazio* (sem importações nem outras variáveis definidas), de modo que o comportamento deverá ser idêntico à execução do programa na linha de comando usando *python script.py*. Todas as variáveis (importações, funções e globais) definidas no arquivo (até que uma exceção, se houver, seja gerada) serão então acessíveis no shell IPython:

```
In [15]: c
Out [15]: 7.5

In [16]: result
Out[16]: 1.4666666666666666
```

Se um script Python espera argumentos de linha de comando (a serem encontrados em sys.argv), esses poderão ser passados após o path do arquivo, como executado na linha de comando.

Caso você queira dar acesso a variáveis já definidas no namespace interativo do IPython a um script, utilize %run -i em vez de simplesmente %run.

No notebook Jupyter, é possível também usar a função mágica relacionada %load, que importa um script para uma célula de código:

```
>>> %load ipython_script_test.py

    def f(x, y, z):
        return (x + y) / z

    a = 5
    b = 6
    c = 7.5
    result = f(a, b, c)
```

Interrompendo um código em execução

Pressionar Ctrl-C enquanto um código qualquer estiver executando, seja um script com %run ou um comando com execução demorada, vai gerar uma KeyboardInterrupt. Isso fará com que quase todos os programas Python parem imediatamente, exceto em determinados casos incomuns.

Quando uma porção de código Python tiver sido chamada em alguns módulos de extensão compilados, pressionar Ctrl-C nem sempre vai interromper a execução do programa de imediato. Em casos como esse, você terá que esperar até que o controle seja devolvido ao interpretador Python ou, em circunstâncias mais lamentáveis, terá que encerrar o processo Python de maneira forçada.

Executando código da área de transferência

Se você estiver usando o notebook Jupyter, poderá copiar e colar código para qualquer célula e executá-lo. Também é possível executar o código da área de transferência (clipboard) no shell IPython. Suponha que você tivesse o código a seguir em outra aplicação:

```
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
```

Os métodos mais simples são usar as funções mágicas %paste e %cpaste. %paste toma qualquer texto que esteja na área de transferência e executa-o como um único bloco no shell:

```
In [17]: %paste
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
## -- Final do texto colado --
```

%cpaste é semelhante, porém oferece um prompt especial no qual você pode colar o código:

```
In [18]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:x = 5
:y = 7
:if x > 5:
: x += 1
:
: y = 8
:--
```

Com o bloco %cpaste, você tem a liberdade de colar qualquer quantidade de código que quiser antes de executá-lo. É possível decidir utilizar %cpaste para ver o código colado antes de sua execução. Caso cole o código incorreto acidentalmente, poderá sair do prompt de %cpaste pressionando Ctrl-C.

Atalhos de teclado no terminal

O IPython tem muitos atalhos de teclado para navegar no prompt (os quais serão conhecidos dos usuários do editor de texto Emacs ou do shell bash do Unix) e interagir com o histórico de comandos do shell. A Tabela 2.1 sintetiza alguns dos atalhos mais comumente utilizados. Veja a Figura 2.5, que mostra alguns desses atalhos, por exemplo, um movimento de cursor.

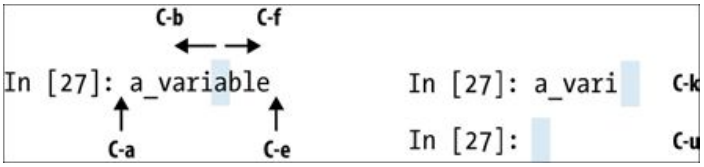


Figura 2.5 – Ilustração de alguns atalhos de teclado no shell IPython.

Tabela 2.1 – Atalhos de teclado padrões do IPython

Atalhos de teclado	Descrição
Ctrl-P ou seta para cima	Busca para trás no histórico de comandos por comandos que comecem com o texto inserido no momento
Ctrl-N ou seta para baixo	Busca para a frente no histórico de comandos por comandos que comecem com o texto inserido no momento
Ctrl-R	Busca reversa no histórico, no estilo de leitura de linha (correspondência parcial)
Ctrl-Shift-V	Cola o texto da área de transferência (clipboard)
Ctrl-C	Interrompe o código executado no momento
Ctrl-A	Move o cursor para o início da linha
Ctrl-E	Move o cursor para o final da linha
Ctrl-K	Apaga o texto a partir do cursor até o final da linha
Ctrl-U	Descarta todo o texto da linha atual
Ctrl-F	Move o cursor um caractere para a frente
Ctrl-B	Move o cursor um caractere para trás
Ctrl-L	Limpa a tela

Observe que os notebooks Jupyter têm um conjunto bem distinto de atalhos de teclado para navegação e edição. Como esses atalhos evoluíram mais rapidamente que os do IPython, incentivo você a explorar o sistema integrado de ajuda nos menus do notebook Jupyter.

Sobre os comandos mágicos

Os comandos especiais do IPython (que não estão incluídos no Python) são conhecidos como comandos “mágicos”. Foram concebidos para facilitar tarefas comuns e permitir que você controle facilmente o comportamento do sistema IPython. Um comando

mágico é qualquer comando prefixado com o símbolo de porcentagem %. Por exemplo, você pode conferir o tempo de execução de qualquer instrução Python, como uma multiplicação de matrizes, usando a função mágica %timeit (isso será discutido com mais detalhes posteriormente):

```
In [20]: a = np.random.randn(100, 100)

In [20]: %timeit np.dot(a, a)
10000 loops, best of 3: 20.9 µs per loop
```

Os comandos mágicos podem ser vistos como programas de linha de comando a serem executados no sistema IPython. Muitos deles têm opções adicionais de “linha de comando”, que podem ser todos vistos (como seria de se esperar) usando ?:

```
In [21]: %debug?
Docstring:
::

  %debug [--breakpoint FILE:LINE] [statement [statement ...]]

Activate the interactive debugger.
This magic command support two ways of activating debugger.
One is to activate debugger before executing code. This way, you can set a
break point, to step through the code from the point.
You can use this mode by giving statements to execute and optionally a
breakpoint.

The other one is to activate debugger in post-mortem mode. You can activate
this mode simply running %debug without any argument.
If an exception has just occurred, this lets you inspect its stack frames
interactively. Note that this will always work only on the last traceback that
occurred, so you must call this quickly after an exception that you wish to inspect
has fired, because if another one occurs, it clobbers the previous one.

If you want IPython to automatically do this on every exception, see the %pdb
magic for more details.

positional arguments:
  statement Code to run in debugger. You can omit this in cell magic mode.
```

```
optional arguments:
  --breakpoint <FILE:LINE>, -b <FILE:LINE>
                        Set break point at LINE in FILE.
```

As funções mágicas podem ser usadas por padrão, sem o sinal de porcentagem, desde que nenhuma variável esteja definida com o mesmo nome da função mágica em questão. Esse recurso se chama *automagic* e pode ser ativado ou desativado com %automagic.

Algumas funções mágicas se comportam como funções Python, e sua saída pode ser atribuída a uma variável:

```
In [22]: %pwd
Out[22]: '/home/wesm/code/pydata-book

In [23]: foo = %pwd

In [24]: foo
Out[24]: '/home/wesm/code/pydata-book'
```

Como a documentação do IPython é acessível de dentro do sistema, incentivo você a explorar todos os comandos especiais disponíveis digitando %quickref ou %magic.

A Tabela 2.2 destaca algumas das funções mais importantes para ser produtivo no processamento interativo e no desenvolvimento Python no IPython.

Tabela 2.2 – Alguns comandos mágicos do IPython usados com frequência

Comando	Descrição
%quickref	Exibe o Quick Reference Card (Cartão de Referência Rápida) do IPython
%magic	Exibe a documentação detalhada para todos os comandos mágicos disponíveis
%debug	Entra no debugger interativo no final do traceback da última exceção
%hist	Exibe o histórico de entrada (e opcionalmente de saída) dos comandos
%pdb	Entra automaticamente no debugger após qualquer exceção
%paste	Executa código Python pré-formatado da área de transferência
%cpaste	Abre um prompt especial para colar manualmente um código Python a ser executado

Comando	Descrição
%reset	Apaga todas as variáveis/nomes definidos no namespace interativo
%page *OBJETO*	Faz um pretty-print do objeto e exibe-o por meio de um paginador
%run *script.py*	Executa um script Python no IPython
%prun *instrução*	Executa *instrução* com cProfile e informa a saída do profiler
%time *instrução*	Informa o tempo de execução de uma única instrução
%timeit *instrução*	Executa uma instrução várias vezes para calcular um tempo médio de execução do conjunto; é útil para medir o tempo de um código cujo tempo de execução é muito rápido
%who, %who_ls, %whos	Exibe variáveis definidas no namespace interativo, com níveis variados de informações/verbosidade
%xdel *variável*	Apaga uma variável e tenta limpar qualquer referência ao objeto no interior do IPython

Integração com a matplotlib

Um dos motivos para a popularidade do IPython no processamento analítico é que ele se integra bem com bibliotecas de visualizaºção de dados e outras para interface de usuário, como a matplotlib. Não se preocupe se você não usou a matplotlib antes; ela será discutida com mais detalhes posteriormente neste livro. A função mágica %matplotlib configura a sua integração com o shell IPython ou o notebook Jupyter. Isso é importante, pois, do contrário, as plotagens que você criar não aparecerão (notebook) ou assumirão o controle da sessão até que essa seja fechada (shell).

No shell IPython, executar %matplotlib configura a integração, de modo que você poderá criar várias janelas de plotagens sem interferir na sessão do console:

```
In [26]: %matplotlib
Using matplotlib backend: Qt4Agg
```

No Jupyter, o comando é um pouco diferente (Figura 2.6):

In [26]: %matplotlib inline

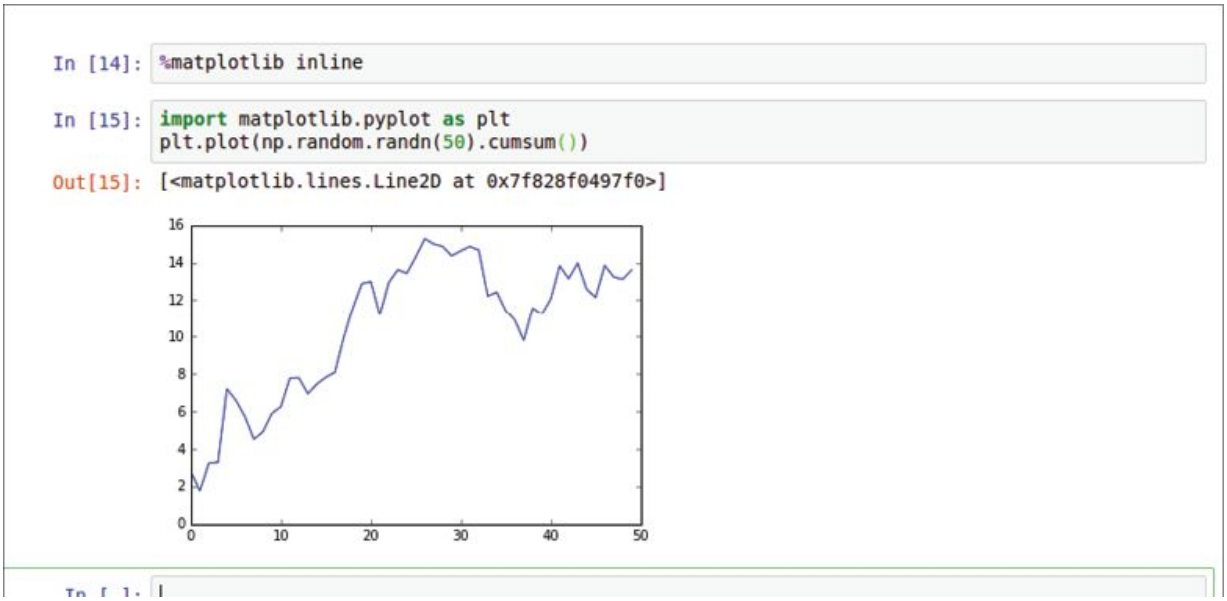


Figura 2.6 – Plotagem inline da matplotlib no Jupyter.

2.3 Básico da linguagem Python

Nesta seção, apresentarei uma visão geral dos conceitos essenciais de programação Python e do funcionamento da linguagem. No próximo capítulo, darei mais detalhes sobre as estruturas de dados de Python, as funções e outras ferramentas embutidas.

Semântica da linguagem

O design da linguagem Python destaca-se por sua ênfase na legibilidade, na simplicidade e em seu caráter explícito. Algumas pessoas vão mais longe, a ponto de compará-la com um "pseudocódigo executável".

Indentação no lugar de chaves

Python utiliza espaços em branco (tabulações ou espaços) para estruturar o código, em vez de usar chaves, como fazem muitas outras linguagens como R, C++, Java e Perl. Considere um laço for de um algoritmo de ordenação:

for x in array:

```
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

Dois-pontos indicam o início de um bloco de código indentado, após o qual todo o código deve estar indentado com o mesmo espaço, até o final do bloco.

Não importa se você ama ou detesta esse recurso, os espaços em branco significativos são um fato da vida para programadores Python e, em minha experiência, conseguem deixar o código Python mais legível do que outras linguagens que já usei. Embora possa parecer estranho à primeira vista, espero que, com o tempo, você se acostume.

Recomendo fortemente usar *quatro espaços* como sua indentação padrão e substituir as tabulações por quatro espaços. Muitos editores de texto têm uma configuração para substituir tabulações por espaços automaticamente (faça isso!). Algumas pessoas utilizam tabulações ou um número diferente de espaços, com dois espaços não sendo terrivelmente incomum. De modo geral, quatro espaços são o padrão adotado pela grande maioria dos programadores Python, portanto recomendo fazer isso, na falta de outro motivo convincente.

Como podemos ver a essa altura, instruções Python também não precisam terminar com pontos e vírgulas. Esses podem ser usados, porém, para separar diversas instruções em uma única linha:

```
a = 5; b = 6; c = 7
```

Colocar várias instruções em uma só linha geralmente não é incentivado em Python, pois, com frequência, deixa o código menos legível.

Tudo é um objeto

Uma característica importante da linguagem Python é a consistência de seu *modelo de objetos*. Todo número, string, estrutura de dados, função, classe, módulo e assim por diante existe no interpretador

Python em sua própria “caixa”, que é referenciada como um *objeto Python*. Todo objeto tem um *tipo* associado (por exemplo, *string* ou *função*) e dados internos. Na prática, isso deixa a linguagem muito flexível, pois até mesmo as funções podem ser tratadas como qualquer outro objeto.

Comentários

Qualquer texto precedido pelo sinal de sustenido (#) será ignorado pelo interpretador Python. Com frequência, ele é usado para adicionar comentários ao código. Ocasionalmente talvez você queira também excluir determinados blocos de código sem apagá-los. Uma solução simples é *comentar* o código:

```
results = []
for line in file_handle:
    # mantém as linhas em branco por enquanto
    # if len(line) == 0:
    # continue
    results.append(line.replace('foo', 'bar'))
```

Comentários também podem ocorrer após uma linha de código executado. Embora alguns programadores prefiram colocar os comentários na linha que antecede uma linha de código em particular, colocá-los depois às vezes pode ser conveniente:

```
print("Reached this line") # Simples informação sobre o status
```

Chamadas de função e de métodos de objeto

Chame funções usando parênteses e passando zero ou mais argumentos, opcionalmente atribuindo o valor devolvido a uma variável:

```
result = f(x, y, z)
g()
```

Quase todo objeto em Python tem funções associadas, conhecidas como *métodos*, que têm acesso ao conteúdo interno do objeto. É possível chamá-las usando a sintaxe a seguir:

```
obj.some_method(x, y, z)
```

As funções podem aceitar argumentos tanto *posicionais* (positional arguments) quanto *nomeados* (keyword arguments):

```
result = f(a, b, c, d=5, e='foo')
```

Mais informações sobre isso serão dadas posteriormente.

Variáveis e passagem de argumentos

Quando fazemos uma atribuição a uma variável (ou a um *nome*) em Python, estamos criando uma *referência* ao objeto do lado direito do sinal de igualdade. Em termos práticos, considere uma lista de inteiros:

```
In [8]: a = [1, 2, 3]
```

Suponha que vamos atribuir `a` a uma nova variável `b`:

```
In [9]: b = a
```

Em algumas linguagens, essa atribuição faria os dados `[1, 2, 3]` serem copiados. Em Python, `a` e `b`, na verdade, agora se referem ao mesmo objeto: a lista original `[1, 2, 3]` (veja a Figura 2.7 para uma representação). Você pode comprovar isso por conta própria concatenando um elemento a `a` e então examinando `b`:

```
In [10]: a.append(4)

In [11]: b
Out[11]: [1, 2, 3, 4]
```

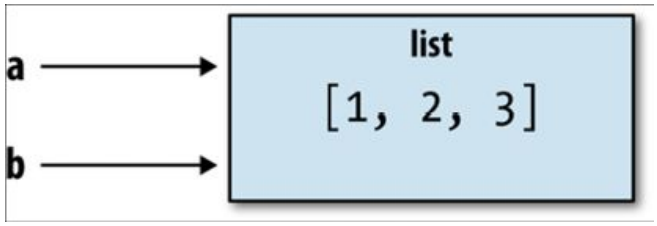


Figura 2.7 – Duas referências ao mesmo objeto.

Compreender a semântica das referências em Python, e quando, como e por que os dados são copiados, é particularmente crucial quando estivermos trabalhando com conjuntos de dados maiores em Python.

A atribuição também é chamada de *vinculação* (binding), pois estamos vinculando um nome a um objeto. Os nomes de variáveis que receberam uma atribuição ocasionalmente podem ser chamados de variáveis vinculadas.

Quando passamos objetos como argumentos para uma função, novas variáveis locais são criadas para referenciar os objetos originais, sem qualquer cópia. Se você vincular um novo objeto a uma variável em uma função, essa alteração não se refletirá no escopo-pai. Assim, é possível alterar o conteúdo interno de um argumento mutável. Suponha que tivéssemos a seguinte função:

```
def append_element(some_list, element):
    some_list.append(element)
```

Então teríamos:

```
In [27]: data = [1, 2, 3]

In [28]: append_element(data, 4)

In [29]: data
Out[29]: [1, 2, 3, 4]
```

Referências dinâmicas, tipos fortes

Em contraste com muitas linguagens compiladas, como Java e C++, *referências* a objetos em Python não têm nenhum tipo associado a elas. Não há problemas com as instruções a seguir:

```
In [12]: a = 5

In [13]: type(a)
Out[13]: int

In [14]: a = 'foo'
In [15]: type(a)
Out[15]: str
```

Variáveis são nomes para objetos em um namespace em particular; a informação sobre tipo é armazenada no próprio objeto. Alguns

observadores poderão apressadamente concluir que Python não é uma “linguagem tipada”. Isso não é verdade; considere o exemplo a seguir:

```
In [16]: '5' + 5
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-16-4dd8efb5fac1> in <module>()
----> 1 '5' + 5
TypeError: must be str, not int
```

Em algumas linguagens, por exemplo, Visual Basic, a string `'5'` pode ser implicitamente convertida (ou pode ocorrer um *cast*) para um inteiro, resultando, assim, em 10. Todavia, em outras linguagens, como JavaScript, pode haver um cast do inteiro `5` para uma string, resultando na string concatenada `'55'`. Quanto a esse aspecto, Python é considerada uma linguagem *fortemente tipada*, o que significa que todo objeto tem um tipo (ou classe) específico, e conversões implícitas ocorrerão somente em determinadas circunstâncias óbvias, como a seguir:

```
In [17]: a = 4.5

In [18]: b = 2

# Formatação de string, que será explicada mais adiante
In [19]: print('a is {0}, b is {1}'.format(type(a), type(b)))
a is <class 'float'>, b is <class 'int'>

In [20]: a / b
Out[20]: 2.25
```

Conhecer o tipo de um objeto é importante, e ser capaz de escrever funções que lidem com vários tipos diferentes de entrada é conveniente. Você pode verificar se um objeto é uma instância de um tipo particular usando a função `isinstance`:

```
In [21]: a = 5

In [22]: isinstance(a, int)
Out[22]: True
```

isinstance pode aceitar uma tupla de tipos se você quiser verificar se o tipo de um objeto está entre aqueles presentes na tupla:

```
In [23]: a = 5; b = 4.5

In [24]: isinstance(a, (int, float))
Out[24]: True

In [25]: isinstance(b, (int, float))
Out[25]: True
```

Atributos e métodos

Objetos em Python geralmente têm tanto atributos (outros objetos Python armazenados “dentro” do objeto) quanto métodos (funções associadas a um objeto, que podem ter acesso aos seus dados internos). Ambos são acessados por meio da sintaxe *obj.nome_do_atributo*:

```
In [1]: a = 'foo'

In [2]: a.<Pressione Tab>
a.capitalize a.format a.isupper a.rindex a.strip
a.center a.index a.join a.rjust a.swapcase
a.count a.isalnum a.ljust a.rpartition a.title
a.decode a.isalpha a.lower a.rsplit a.translate
a.encode a.isdigit a.lstrip a.rstrip a.upper
a.endswith a.islower a.partition a.split a.zfill
a.expandtabs a.isspace a.replace a.splitlines
a.find a.istitle a.rfind a.startswith
```

Atributos e métodos também podem ser acessados pelo nome usando a função getattr:

```
In [27]: getattr(a, 'split')
Out[27]: <function str.split>
```

Em outras linguagens, acessar objetos pelo nome muitas vezes é chamado de “reflexão”. Embora não usemos intensivamente as funções getattr e as funções relacionadas hasattr e setattr neste livro, elas podem ser utilizadas com muita eficiência para escrever códigos genéricos e reutilizáveis.

Duck typing

Com frequência, talvez você não se importe com o tipo de um objeto, mas apenas com o fato de ele ter determinados métodos ou comportamentos. Às vezes isso é chamado de “duck typing” (tipagem pato), com base no ditado: “Se anda como pato e faz quack como um pato, então é um pato”. Por exemplo, você pode verificar se um objeto é iterável se ele implementar o *protocolo iterador.* Para muitos objetos, isso significa que ele tem um “método mágico” __iter__, embora uma alternativa para verificar, e que é uma opção melhor, seja tentar usar a função iter:

```
def isiterable(obj):
    try:
        iter(obj)
        return True
    except TypeError: # não é iterável
        return False
```

Essa função devolveria True para strings, assim como para a maioria dos tipos Python para coleção:

```
In [29]: isiterable('a string')
Out[29]: True

In [30]: isiterable([1, 2, 3])
Out[30]: True

In [31]: isiterable(5)
Out[31]: False
```

Uma ocasião em que uso essa funcionalidade o tempo todo é quando escrevo funções que podem aceitar vários tipos de entrada. Um caso comum é escrever uma função que aceite qualquer tipo de sequência (lista, tupla, ndarray), ou até mesmo um iterador. Você pode inicialmente verificar se o objeto é uma lista (ou um array NumPy) e, se não for, convertê-lo em uma:

```
if not isinstance(x, list) and isiterable(x):
    x = list(x)
```

Importações

Em Python, um *módulo* é simplesmente um arquivo com a extensão *.py* contendo código Python. Suponha que tivéssemos o módulo a seguir:

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

def g(a, b):
    return a + b
```

Se quiséssemos acessar as variáveis e funções definidas em *some_module.py* a partir de outro arquivo no mesmo diretório, poderíamos fazer o seguinte:

```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

Ou, de modo equivalente:

```
from some_module import f, g, PI
result = g(5, PI)
```

Ao usar a palavra reservada as, podemos dar diferentes nomes de variáveis às importações:

```
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

Operadores binários e comparações

A maior parte das operações matemáticas binárias e comparações está de acordo com o que você esperaria:

```
In [32]: 5 - 7
Out[32]: -2
```

```
In [33]: 12 + 21.5
Out[33]: 33.5

In [34]: 5 <= 2
Out[34]: False
```

Veja a Tabela 2.3 que contém todos os operadores binários disponíveis.

Para verificar se duas referências apontam para o mesmo objeto, utilize a palavra reservada `is`. `is not` também é perfeitamente válido caso você queira verificar se dois objetos não são os mesmos:

```
In [35]: a = [1, 2, 3]

In [36]: b = a

In [37]: c = list(a)

In [38]: a is b
Out[38]: True

In [39]: a is not c
Out[39]: True
```

Como `list` sempre cria uma nova lista Python (isto é, uma cópia), podemos ter certeza de que `c` é distinto de `a`. Comparar com `is` não faz o mesmo que o operador `==`, pois, nesse caso, temos:

```
In [40]: a == c
Out[40]: True
```

Um uso bem comum de `is` e de `is not` é para verificar se uma variável é `None`, pois há somente uma instância de `None`:

```
In [41]: a = None

In [42]: a is None
Out[42]: True
```

Tabela 2.3 – Operadores binários

Operação	Descrição
a + b	Soma a e b

Operação	Descrição
a - b	Subtrai b de a
a * b	Multiplica a por b
a / b	Divide a por b
a // b	Faz a divisão pelo piso (floor division) de a por b, descartando qualquer resto fracionário
a ** b	Eleva a à potência de b
a & b	True se tanto a quanto b forem True; para inteiros, é a operação bit a bit (bitwise) AND
a | b	True se a ou b for True; para inteiros, é a operação bit a bit OR
a ^ b	Para booleanos, True se a ou b for True, mas não ambos; para inteiros, é a operação bit a bit EXCLUSIVE-OR
a == b	True se a for igual a b
a != b	True se a não for igual a b
a < b, a <= b	True se a for menor que (menor ou igual a) b
a > b, a >= b	True se a for maior que (maior ou igual a) b
a is b	True se a e b referenciarem o mesmo objeto Python
a is not b	True se a e b referenciarem objetos Python diferentes

Objetos mutáveis e imutáveis

A maioria dos objetos em Python, como listas, dicionários, arrays NumPy e a maior parte dos tipos definidos pelo usuário (classes), é mutável. Isso significa que o objeto ou os valores que eles contêm podem ser modificados:

```
In [43]: a_list = ['foo', 2, [4, 5]]

In [44]: a_list[2] = (3, 4)

In [45]: a_list
Out[45]: ['foo', 2, (3, 4)]
```

Outros objetos, como strings e tuplas, são imutáveis:

```
In [46]: a_tuple = (3, 5, (4, 5))
```

```
In [47]: a_tuple[1] = 'four'
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-47-23fe12da1ba6> in <module>()
----> 1 a_tuple[1] = 'four'
TypeError: 'tuple' object does not support item assignment
```

Lembre-se de que somente porque você *pode* modificar um objeto isso não significa que você sempre *deva* fazê-lo. Ações como essas são conhecidas como *efeitos colaterais.* Por exemplo, quando estiver escrevendo uma função, qualquer efeito colateral deve ser explicitamente informado ao usuário na documentação da função ou nos comentários. Se for possível, recomendo tentar evitar efeitos colaterais *e favorecer a imutabilidade, embora possa haver objetos mutáveis envolvidos.*

Tipos escalares

Python, junto com sua biblioteca-padrão, tem um pequeno conjunto de tipos embutidos (built-in types) para tratar dados numéricos, strings, valores booleanos (True ou False), datas e horas. Esses tipos de "valores únicos" às vezes são chamados de *tipos escalares, e nós nos referiremos a eles neste livro como escalares.* Veja a Tabela 2.4 que contém uma lista dos principais tipos escalares. O tratamento de data e hora será discutido separadamente, pois são feitos pelo módulo datetime da biblioteca-padrão.

Tabela 2.4 – Tipos escalares padrões de Python

Tipo	Descrição
None	O valor "null" de Python (há somente uma instância do objeto None)
str	Tipo string; armazena strings Unicode (codificadas com UTF-8)
bytes	Bytes ASCII puros (ou Unicode codificado como bytes)
float	Número de ponto flutuante com dupla precisão (64 bits; observe que não há um tipo double separado)
bool	Um valor True ou False
int	Inteiro com sinal, com precisão arbitrária

Tipos numéricos

Os principais tipos Python para números são int e float. Um int pode armazenar números arbitrariamente grandes:

```
In [48]: ival = 17239871

In [49]: ival ** 6
Out[49]: 26254519291092456596965462913230729701102721
```

Números de ponto flutuante são representados com o tipo Python float. Internamente cada um é um valor de dupla precisão (64 bits). Eles também podem ser expressos com notação científica:

```
In [50]: fval = 7.243

In [51]: fval2 = 6.78e-5
```

A divisão de inteiros que não tenha como resultado um número inteiro sempre resultará em um número de ponto flutuante:

```
In [52]: 3 / 2
Out[52]: 1.5
```

Para ter uma divisão de inteiros no estilo C (que descarte a parte fracionária se o resultado não for um número inteiro), utilize o operador de divisão pelo piso (floor division) //:

```
In [53]: 3 // 2
Out[53]: 1
```

Strings

Muitas pessoas utilizam Python por causa de seus recursos embutidos eficazes e flexíveis para processamento de strings. Podemos escrever *strings literais usando aspas simples* ' *ou aspas duplas* ":

```
a = 'one way of writing a string'
b = "another way"
```

Para strings multilinhas (multiline strings) com quebras de linha, podemos usar aspas triplas, que podem ser ''' ou """:

```
c = """
This is a longer string that
```

```
spans multiple lines
"""
```

Você poderá se surpreender com o fato de essa string `c` na verdade conter quatro linhas de texto; as quebras de linha após o """ e após a palavra `lines` estão incluídas na string. Podemos contar os caracteres de quebra de linha com o método `count` em `c`:

```
In [55]: c.count('\n')
Out[55]: 3
```

As strings Python são imutáveis; você não pode modificar uma string:

```
In [56]: a = 'this is a string'

In [57]: a[10] = 'f'
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-57-2151a30ed055> in <module>()
----> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment

In [58]: b = a.replace('string', 'longer string')

In [59]: b
Out[59]: 'this is a longer string'
```

Depois dessa operação, a variável `a` não foi modificada:

```
In [60]: a
Out[60]: 'this is a string'
```

Muitos objetos Python podem ser convertidos em uma string com a função `str`:

```
In [61]: a = 5.6

In [62]: s = str(a)

In [63]: print(s)
5.6
```

As strings são uma sequência de caracteres Unicode e, desse modo, podem ser tratadas como outras sequências, como listas e

tuplas (que exploraremos com mais detalhes no próximo capítulo):

```
In [64]: s = 'python'

In [65]: list(s)
Out[65]: ['p', 'y', 't', 'h', 'o', 'n']

In [66]: s[:3]
Out[66]: 'pyt'
```

A sintaxe `s[:3]` se chama *fatiamento* (slicing) e *está implementada para muitos tipos de sequências* Python. Será explicada com mais detalhes posteriormente, pois é usada com muita frequência neste livro.

O caractere de barra invertida `\` é um *caractere de escape*, o que significa que é utilizado para especificar caracteres especiais, como quebra de linha `\n` ou caracteres Unicode. Para escrever uma string literal com barras invertidas, você precisa escapá-las:

```
In [67]: s = '12\\34'

In [68]: print(s)
12\34
```

Se tiver uma string com muitas barras invertidas, sem caracteres especiais, talvez ache isso um pouco irritante. Felizmente podemos usar um prefixo `r` na aspa de abertura da string, indicando que os caracteres devem ser interpretados como estão:

```
In [69]: s = r'this\has\no\special\characters'
In [70]: s
Out[70]: 'this\\has\\no\\special\\characters'
```

`r` significa *raw* (puro).

Somar duas strings faz com que sejam concatenadas, e uma nova string é gerada:

```
In [71]: a = 'this is the first half '

In [72]: b = 'and this is the second half'

In [73]: a + b
Out[73]: 'this is the first half and this is the second half'
```

Templating ou formatação de strings é outro assunto importante. O número de maneiras de fazer isso se expandiu após o advento de Python 3 e, nesta seção, descreverei rapidamente o funcionamento de uma das principais interfaces. Os objetos string têm um método format que pode ser usado para substituir argumentos formatados na string, gerando uma nova string:

```
In [74]: template = '{0:.2f} {1:s} are worth US${2:d}'
```

Nessa string:

- {0:.2f} significa formatar o primeiro argumento como um número de ponto flutuante com duas casas decimais.
- {1:s} significa formatar o segundo argumento como uma string.
- {2:d} significa formatar o terceiro argumento como um inteiro exato.

A fim de substituir os argumentos para esses parâmetros de formato, passamos uma sequência de argumentos ao método format:

```
In [75]: template.format(4.5560, 'Argentine Pesos', 1)
Out[75]: '4.56 Argentine Pesos are worth US$1'
```

Formatação de strings é um tópico profundo; há vários métodos e diversas opções e ajustes disponíveis para controlar o modo como os valores são formatados na string resultante. Para saber mais, recomendo consultar a documentação oficial de Python (*https://docs.python.org/3.6/library/string.html*).

Discutirei o processamento geral de strings com mais detalhes, conforme se relaciona com a análise de dados, no Capítulo 8.

Bytes e Unicode

Em Python moderno (isto é, Python 3.0 e em versões mais recentes), o Unicode se tornou o tipo string de primeira classe para permitir um tratamento mais consistente de textos ASCII e não ASCII. Em versões mais antigas de Python, as strings eram todas em bytes, sem nenhuma codificação Unicode explícita. Você poderia converter para Unicode supondo que conhecesse a codificação dos

caracteres. Vamos ver um exemplo:

```
In [76]: val = "español"

In [77]: val
Out[77]: 'español'
```

Podemos converter essa string Unicode para a sua representação em bytes UTF-8 usando o método `encode`:

```
In [78]: val_utf8 = val.encode('utf-8')

In [79]: val_utf8
Out[79]: b'espa\xc3\xb1ol'

In [80]: type(val_utf8)
Out[80]: bytes
```

Supondo que você conheça a codificação Unicode de um objeto `bytes`, é possível fazer o inverso usando o método `decode`:

```
In [81]: val_utf8.decode('utf-8')
Out[81]: 'español'
```

Embora a preferência seja usar UTF-8 para qualquer codificação, por motivos históricos você poderá encontrar dados em qualquer variedade de codificações diferentes:

```
In [82]: val.encode('latin1')
Out[82]: b'espa\xf1ol'

In [83]: val.encode('utf-16')
Out[83]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'

In [84]: val.encode('utf-16le')
Out[84]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

É mais comum encontrar objetos `bytes` no contexto de trabalho com arquivos, em que decodificar implicitamente todos os dados em strings Unicode pode não ser desejável.

Embora raramente você vá precisar fazer isso, é possível definir seus próprios literais de bytes prefixando uma string com `b`:

```
In [85]: bytes_val = b'this is bytes'
```

```
In [86]: bytes_val
Out[86]: b'this is bytes'

In [87]: decoded = bytes_val.decode('utf8')

In [88]: decoded # é str (Unicode) agora
Out[88]: 'this is bytes'
```

Booleanos

Os dois valores booleanos em Python são escritos como True e False. Comparações e outras expressões condicionais são avaliadas como True ou False. Valores booleanos são combinados com as palavras reservadas and e or.

```
In [89]: True and True
Out[89]: True

In [90]: False or True
Out[90]: True
```

Casting de tipos

Os tipos str, bool, int e float também são funções que podem ser usadas para cast de valores para esses tipos:

```
In [91]: s = '3.14159'

In [92]: fval = float(s)

In [93]: type(fval)
Out[93]: float

In [94]: int(fval)
Out[94]: 3

In [95]: bool(fval)
Out[95]: True

In [96]: bool(0)
Out[96]: False
```

None

None é o valor do tipo nulo em Python. Se uma função não devolver explicitamente um valor, ela devolverá implicitamente None:

```
In [97]: a = None

In [98]: a is None
Out[98]: True

In [99]: b = 5

In [100]: b is not None
Out[100]: True
```

None também é um valor default comum para argumentos de função:

```
def add_and_maybe_multiply(a, b, c=None):
    result = a + b

    if c is not None:
        result = result * c

    return result
```

Embora seja uma questão técnica, vale a pena ter em mente que None não é apenas uma palavra reservada, mas também uma instância única de NoneType:

```
In [101]: type(None)
Out[101]: NoneType
```

Datas e horas

O módulo embutido datetime de Python disponibiliza os tipos datetime, date e time. O tipo datetime, como você pode imaginar, combina as informações armazenadas em date e em time, e é o tipo mais comumente utilizado:

```
In [102]: from datetime import datetime, date, time

In [103]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [104]: dt.day
```

```
Out[104]: 29

In [105]: dt.minute
Out[105]: 30
```

Dada uma instância de datetime, você pode extrair os objetos date e time equivalentes chamando métodos no datetime do mesmo nome:

```
In [106]: dt.date()
Out[106]: datetime.date(2011, 10, 29)

In [107]: dt.time()
Out[107]: datetime.time(20, 30, 21)
```

O método strftime formata um datetime como uma string:

```
In [108]: dt.strftime('%m/%d/%Y %H:%M')
Out[108]: '10/29/2011 20:30'
```

Strings podem ser convertidas (um parse é feito) em objetos datetime com a função strptime:

```
In [109]: datetime.strptime('20091031', '%Y%m%d')
Out[109]: datetime.datetime(2009, 10, 31, 0, 0)
```

Veja a Tabela 2.5 que contém uma lista completa das especificações de formato.

Quando você estiver agregando ou agrupando dados de séries temporais, ocasionalmente será conveniente substituir os campos de horário de uma série de datetimes – por exemplo, substituindo os campos de minuto e de segundo por zero:

```
In [110]: dt.replace(minute=0, second=0)
Out[110]: datetime.datetime(2011, 10, 29, 20, 0)
```

Como datetime.datetime é um tipo imutável, métodos como esses sempre geram novos objetos.

A diferença entre dois objetos datetime gera um tipo datetime.timedelta:

```
In [111]: dt2 = datetime(2011, 11, 15, 22, 30)

In [112]: delta = dt2 - dt

In [113]: delta
```

```
Out[113]: datetime.timedelta(17, 7179)

In [114]: type(delta)
Out[114]: datetime.timedelta
```

A saída timedelta(17, 7179) indica que o timedelta codifica um offset de 17 dias e 7.179 segundos.

Somar um timedelta a um datetime gera um novo datetime deslocado:

```
In [115]: dt

Out[115]: datetime.datetime(2011, 10, 29, 20, 30, 21)

In [116]: dt + delta
Out[116]: datetime.datetime(2011, 11, 15, 22, 30)
```

Tabela 2.5 – Especificação de formatos de datetime *(compatível com ISO C89)*

Tipo	Descrição
%Y	Ano com quatro dígitos
%y	Ano com dois dígitos
%m	Mês com dois dígitos [01, 12]
%d	Dia com dois dígitos [01, 31]
%H	Hora (relógio com 24 horas) [00, 23]
%I	Hora (relógio com 12 horas) [01, 12]
%M	Minuto com dois dígitos [00, 59]
%S	Segundos [00, 61] (segundos 60, 61 são usados para segundos intercalares (leap seconds))
%w	Dia da semana como inteiro [0 (Domingo), 6]
%U	Número da semana no ano [00, 53]; domingo é considerado o primeiro dia da semana, e os dias antes do primeiro domingo do ano são a “semana 0”
%W	Número da semana no ano [00, 53]; segunda-feira é considerada o primeiro dia da semana, e os dias antes da primeira segunda-feira do ano são a “semana 0”
%z	Deslocamento do fuso horário UTC como +HHMM ou -HHMM; vazio se não for considerado o fuso horário
%F	Atalho para %Y-%m-%d (por exemplo, 2012-4-18)
%D	Atalho para %m/%d/%y (por exemplo, 04/18/12)

Controle de fluxo

Python tem várias palavras reservadas embutidas para lógica condicional, laços e outros conceitos padrões de *controle de fluxo* encontrados em outras linguagens de programação.

if, elif e else

A instrução `if` é um dos tipos de instrução de controle de fluxo mais conhecidos. Ela verifica uma condição que, se for `True`, fará o código do bloco que vem a seguir ser avaliado:

```
if x < 0:
    print('It's negative')
```

Uma instrução `if` pode ser opcionalmente seguida de um ou mais blocos `elif`, e um bloco `else` que capturará tudo mais se todas as condições forem `False`:

```
if x < 0:
    print('It's negative')
elif x == 0:
    print('Equal to zero')
elif 0 < x < 5:
    print('Positive but smaller than 5')
else:
    print('Positive and larger than or equal to 5')
```

Se alguma das condições for `True`, nenhum bloco `elif` ou `else` mais adiante será alcançado. Com uma condição composta usando `and` ou `or`, as condições são avaliadas da esquerda para a direita e estarão em curto-circuito:

```
In [117]: a = 5; b = 7

In [118]: c = 8; d = 4

In [119]: if a < b or c > d:
   .....: print('Made it')
Made it
```

Nesse exemplo, a comparação `c > d` jamais será avaliada, pois a primeira comparação foi `True`.

Também é possível encadear comparações:

```
In [120]: 4 > 3 > 2 > 1
Out[120]: True
```

Laços for

Os laços for servem para iterar por uma coleção (como uma lista ou uma tupla) ou um iterador. Eis a sintaxe padrão de um laço for:

```
for value in collection:
    # faz algo com value
```

Você pode fazer um laço for avançar para a próxima iteração, ignorando o restante do bloco, usando a palavra reservada continue. Considere o código a seguir, que soma inteiros em uma lista e ignora valores None:

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
    if value is None:
        continue
    total += value
```

Podemos sair totalmente de um laço for com a palavra reservada break. O código a seguir soma elementos da lista até que um 5 seja alcançado:

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
    if value == 5:
        break
    total_until_5 += value
```

A palavra reservada break encerra somente o laço for mais interno; qualquer laço for mais externo continuará sendo executado:

```
In [121]: for i in range(4):
    .....:     for j in range(4):
    .....:         if j > i:
    .....:             break
    .....:         print((i, j))
```

```
    .....:
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

Conforme veremos com mais detalhes, se os elementos da coleção ou do iterador forem sequências (digamos, tuplas ou listas), eles poderão ser convenientemente *desempacotados* em variáveis na instrução do laço for:

```
for a, b, c in iterator:
    # faz algo
```

Laços while

Um laço while especifica uma condição e um bloco de código que deverá ser executado até que a condição seja avaliada como False ou o laço seja explicitamente encerrado com break:

```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```

pass

pass é a instrução “no-op” em Python. Pode ser usada em blocos em que nenhuma ação deva ser executada (ou como um placeholder para um código ainda não implementado); é necessário somente porque Python utiliza espaços em branco para delimitar blocos:

```
if x < 0:
```

```
    print('negative!')
elif x == 0:
    # TODO: insira algo inteligente aqui
    pass
else:
    print('positive!')
```

range

A função range devolve um iterador que produz uma sequência de inteiros uniformemente espaçados:

```
In [122]: range(10)
Out[122]: range(0, 10)

In [123]: list(range(10))
Out[123]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Podemos especificar tanto o início quanto o fim e o passo (que pode ser negativo):

```
In [124]: list(range(0, 20, 2))
Out[124]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [125]: list(range(5, 0, -1))
Out[125]: [5, 4, 3, 2, 1]
```

Como podemos ver, range gera inteiros até o ponto final, porém sem incluí-lo. Um uso comum de range é na iteração em sequências usando um índice:

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
    val = seq[i]
```

Embora possamos usar funções como list para armazenar todos os inteiros gerados por range em outra estrutura de dados, com frequência, o formato do iterador default será o que você quiser. O trecho de código a seguir soma todos os números de 0 a 99.999 que são múltiplos de 3 ou de 5:

```
sum = 0
for i in range(100000):
    # % é o operador de módulo
```

```
    if i % 3 == 0 or i % 5 == 0:
        sum += i
```

Embora seja possível o intervalo gerado ser arbitrariamente grande, o uso de memória em qualquer dado instante pode ser bem baixo.

Expressões ternárias

Uma *expressão ternária* em Python permite combinar um bloco if-else que gera um valor em uma única linha ou expressão. A sintaxe para isso em Python é:

```
value = true-expr if condition else false-expr
```

Nesse caso, *true-expr* e *false-expr* podem ser qualquer expressão Python. Isso tem o mesmo efeito do código mais extenso a seguir:

```
if condition:
    value = true-expr
else:
    value = false-expr
```

Eis um exemplo mais concreto:

```
In [126]: x = 5

In [127]: 'Non-negative' if x >= 0 else 'Negative'
Out[127]: 'Non-negative'
```

Como no caso dos blocos if-else, somente uma das expressões será executada. Assim, os lados “if ” e “else” da expressão ternária poderiam conter processamentos custosos, mas apenas o ramo verdadeiro será avaliado.

Embora seja tentador sempre utilizar expressões ternárias para condensar o seu código, note que a legibilidade poderá ser sacrificada se a condição assim como as expressões verdadeira e falsa forem muito complexas.

CAPÍTULO 3

Estruturas de dados embutidas, funções e arquivos

Este capítulo discute recursos embutidos na linguagem Python, os quais serão usados em todos os lugares no livro. Embora bibliotecas add-on, como pandas e NumPy, acrescentem funcionalidades avançadas de processamento para conjuntos maiores de dados, elas foram projetadas para serem usadas em conjunto com as ferramentas embutidas de manipulação de dados de Python.

Começaremos com as estruturas de dados que são a força de trabalho de Python: tuplas, listas, dicionários e conjuntos. Então discutiremos a criação de suas próprias funções Python reutilizáveis. Por fim, veremos o funcionamento dos objetos de arquivo em Python e a interação com o disco rígido local.

3.1 Estruturas de dados e sequências

As estruturas de dados de Python são simples, porém eficazes. Dominar seu uso é uma parte crucial para se tornar um programador Python proficiente.

Tupla

Uma tupla é uma sequência imutável, de tamanho fixo, de objetos Python. A forma mais fácil de criar uma tupla é com uma sequência de valores separados por vírgula:

```
In [1]: tup = 4, 5, 6

In [2]: tup
```

```
Out[2]: (4, 5, 6)
```

Quando definimos tuplas em expressões mais complicadas, geralmente é necessário colocar os valores entre parênteses, como no exemplo a seguir, em que uma tupla de tuplas é criada:

```
In [3]: nested_tup = (4, 5, 6), (7, 8)

In [4]: nested_tup
Out[4]: ((4, 5, 6), (7, 8))
```

Podemos converter qualquer sequência ou iterador em uma tupla chamando `tuple`:

```
In [5]: tuple([4, 0, 2])
Out[5]: (4, 0, 2)

In [6]: tup = tuple('string')

In [7]: tup
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')
```

Os elementos podem ser acessados com colchetes `[]`, como na maioria dos demais tipos de sequência. Como em C, C++, Java e em muitas outras linguagens, as sequências são indexadas a partir de 0 em Python:

```
In [8]: tup[0]
Out[8]: 's'
```

Já que os objetos armazenados em uma tupla podem ser, eles próprios, mutáveis, depois que a tupla é criada, não será possível modificar qualquer objeto armazenado em cada posição:

```
In [9]: tup = tuple(['foo', [1, 2], True])

In [10]: tup[2] = False
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-10-b89d0c4ae599> in <module>()
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

Se um objeto em uma tupla for mutável, por exemplo, uma lista,

você poderá modificá-lo in-place:

```
In [11]: tup[1].append(3)

In [12]: tup
Out[12]: ('foo', [1, 2, 3], True)
```

Podemos concatenar tuplas usando o operador + para gerar tuplas mais longas:

```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[13]: (4, None, 'foo', 6, 0, 'bar')
```

Multiplicar uma tupla por um inteiro, como ocorre com as listas, tem o efeito de concatenar essa quantidade de cópias da tupla:

```
In [14]: ('foo', 'bar') * 4
Out[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Observe que os próprios objetos não são copiados, apenas as referências a eles.

Desempacotando tuplas

Se você tentar fazer uma *atribuição* a uma expressão de variáveis do tipo tupla, Python tentará desempacotar o valor do lado direito do sinal de igualdade:

```
In [15]: tup = (4, 5, 6)

In [16]: a, b, c = tup

In [17]: b
Out[17]: 5
```

Até mesmo sequências com tuplas aninhadas podem ser desempacotadas:

```
In [18]: tup = 4, 5, (6, 7)

In [19]: a, b, (c, d) = tup

In [20]: d
Out[20]: 7
```

Usando essa funcionalidade, você poderá facilmente trocar nomes

de variáveis (fazer swap) – uma tarefa que, em muitas linguagens, pode ter o aspecto a seguir:

```
tmp = a
a = b
b = tmp
```

Em Python, porém, a troca pode ser feita assim:

```
In [21]: a, b = 1, 2

In [22]: a
Out[22]: 1

In [23]: b
Out[23]: 2

In [24]: b, a = a, b

In [25]: a
Out[25]: 2

In [26]: b
Out[26]: 1
```

Um uso comum do desempacotamento de variáveis é na iteração por sequências de tuplas ou listas:

```
In [27]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [28]: for a, b, c in seq:
....: print('a={0}, b={1}, c={2}'.format(a, b, c))
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

Outro uso comum é devolver múltiplos valores de uma função. Falarei sobre esse assunto com mais detalhes posteriormente.

A linguagem Python recentemente adquiriu um desempacotamento de tuplas um pouco mais sofisticado para ajudar em situações em que você possa querer “arrancar” alguns elementos do início da tupla. A sintaxe especial `*rest` é utilizada, e é também usada em

assinaturas de função para capturar uma lista arbitrariamente longa de argumentos posicionais:

```
In [29]: values = 1, 2, 3, 4, 5

In [30]: a, b, *rest = values

In [31]: a, b
Out[31]: (1, 2)

In [32]: rest
Out[32]: [3, 4, 5]
```

A parte com `rest` às vezes é algo que você vai querer descartar; não há nada de especial acerca do nome `rest`. Por questão de convenção, muitos programadores Python usarão o underscore (_) para variáveis indesejadas:

```
In [33]: a, b, *_ = values
```

Métodos de tupla

Como o tamanho e o conteúdo de uma tupla não podem ser modificados, ela é bem leve no que concerne aos métodos de instância. Um método particularmente útil (também disponível em listas) é `count`, que conta o número de ocorrências de um valor:

```
In [34]: a = (1, 2, 2, 2, 3, 4, 2)

In [35]: a.count(2)
Out[35]: 4
```

Lista

Em oposição às tuplas, as listas têm tamanhos variáveis e seu conteúdo pode ser modificado in-place. Podemos defini-las usando colchetes `[]` ou com a função de tipo `list`:

```
In [36]: a_list = [2, 3, 7, None]

In [37]: tup = ('foo', 'bar', 'baz')
```

```
In [38]: b_list = list(tup)

In [39]: b_list
Out[39]: ['foo', 'bar', 'baz']

In [40]: b_list[1] = 'peekaboo'

In [41]: b_list
Out[41]: ['foo', 'peekaboo', 'baz']
```

Listas e tuplas são semanticamente semelhantes (embora não seja possível modificar as tuplas), e podem ser usadas de forma intercambiável em muitas funções.

A função list é frequentemente usada em processamento de dados como uma maneira de materializar um iterador ou uma expressão geradora:

```
In [42]: gen = range(10)

In [43]: gen
Out[43]: range(0, 10)

In [44]: list(gen)
Out[44]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Adicionando e removendo elementos

Elementos podem ser concatenados no final da lista com o método append:

```
In [45]: b_list.append('dwarf')

In [46]: b_list
Out[46]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

Ao usar insert, podemos inserir um elemento em um local específico da lista:

```
In [47]: b_list.insert(1, 'red')

In [48]: b_list
Out[48]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

O índice de inserção deve estar entre 0 e o tamanho da lista, inclusive.

insert é custosa do ponto de vista de processamento, se comparada com append, pois as referências aos elementos subsequentes têm que ser deslocadas internamente a fim de criar espaço para o novo elemento. Se precisar inserir elementos tanto no início quanto no final de uma sequência, talvez você queira explorar collections.deque, uma fila de dupla extremidade para esse propósito.

A operação inversa de insert é pop, que remove e devolve um elemento de um índice em particular:

```
In [49]: b_list.pop(2)
Out[49]: 'peekaboo'

In [50]: b_list
Out[50]: ['foo', 'red', 'baz', 'dwarf']
```

Elementos podem ser removidos pelo valor com remove, que localiza o primeiro valor desse tipo e remove-o da lista:

```
In [51]: b_list.append('foo')

In [52]: b_list
Out[52]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [53]: b_list.remove('foo')

In [54]: b_list
Out[54]: ['red', 'baz', 'dwarf', 'foo']
```

Se o desempenho não for uma preocupação, ao usar append e remove, você poderá usar uma lista Python como uma estrutura de dados “multiset” perfeitamente apropriada.

Verifique se uma lista contém um valor utilizando a palavra reservada in:

```
In [55]: 'dwarf' in b_list
Out[55]: True
```

A palavra reservada not pode ser usada para negar in:

```
In [56]: 'dwarf' not in b_list
Out[56]: False
```

Verificar se uma lista contém um valor é muito mais lento do que fazer isso com dicionários e conjuntos (que serão apresentados em breve), pois Python faz uma verificação linear nos valores de lista, enquanto é capaz de verificar os outros tipos (com base em tabelas hash) em um tempo constante.

Concatenando e combinando listas

De modo similar às tuplas, somar duas listas com + as concatena:

```
In [57]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[57]: [4, None, 'foo', 7, 8, (2, 3)]
```

Se você já tiver uma lista definida, poderá concatenar vários elementos a ela utilizando o método extend:

```
In [58]: x = [4, None, 'foo']

In [59]: x.extend([7, 8, (2, 3)])

In [60]: x
Out[60]: [4, None, 'foo', 7, 8, (2, 3)]
```

Observe que a concatenação de lista por adição é uma operação comparativamente custosa, pois uma nova lista deve ser criada e os objetos devem ser copiados. Usar extend para concatenar elementos em uma lista existente, especialmente se você estiver construindo uma lista grande, em geral é preferível.

Assim:

```
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)
```

é mais rápido que a alternativa por concatenação:

```
everything = []
for chunk in list_of_lists:
    everything = everything + chunk
```

Ordenação

Você pode ordenar uma lista in-place (sem criar um novo objeto), chamando a sua função sort:

```
In [61]: a = [7, 2, 5, 1, 3]

In [62]: a.sort()

In [63]: a
Out[63]: [1, 2, 3, 5, 7]
```

sort tem algumas opções que ocasionalmente serão convenientes. Uma delas é a capacidade de passar uma *chave de ordenação* secundária – isto é, uma função que gera um valor a ser usado para ordenar os objetos. Por exemplo, poderíamos ordenar uma coleção de strings de acordo com seus tamanhos:

```
In [64]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [65]: b.sort(key=len)

In [66]: b
Out[66]: ['He', 'saw', 'six', 'small', 'foxes']
```

Em breve, veremos a função sorted, que é capaz de gerar uma cópia ordenada de uma sequência genérica.

Busca binária e manutenção de uma lista ordenada

O módulo embutido bisect implementa a busca binária e a inserção em uma lista ordenada. bisect.bisect encontra o local em que um elemento deve ser inserido para manter a lista ordenada, enquanto bisect.insort insere o elemento nesse local:

```
In [67]: import bisect

In [68]: c = [1, 2, 2, 2, 3, 4, 7]

In [69]: bisect.bisect(c, 2)
Out[69]: 4
```

```
In [70]: bisect.bisect(c, 5)
Out[70]: 6

In [71]: bisect.insort(c, 6)

In [72]: c
Out[72]: [1, 2, 2, 2, 3, 4, 6, 7]
```

 As funções do módulo bisect não verificam se a lista está ordenada, pois fazer isso seria custoso do ponto de vista de processamento. Desse modo, usá-las com uma lista não ordenada será uma operação bem-sucedida, sem erros, mas poderá levar a resultados incorretos.

Fatiamento

Podemos selecionar seções da maioria dos tipos de sequência usando a notação de fatias (slices), que, em seu formato básico, é constituída de start:stop passado ao operador de indexação []:

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [74]: seq[1:5]
Out[74]: [2, 3, 7, 5]
```

As fatias também podem receber uma atribuição com uma sequência:

```
In [75]: seq[3:4] = [6, 3]

In [76]: seq
Out[76]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

Embora o elemento no índice start esteja incluído, o índice stop *não está*, de modo que o número de elementos no resultado será stop - start.

Tanto start quanto stop podem ser omitidos, caso em que os valores default serão o início e o final da sequência, respectivamente:

```
In [77]: seq[:5]
Out[77]: [7, 2, 3, 6, 3]
```

```
In [78]: seq[3:]
Out[78]: [6, 3, 5, 6, 0, 1]
```

Índices negativos fatiam a sequência em relação ao final:

```
In [79]: seq[-4:]
Out[79]: [5, 6, 0, 1]

In [80]: seq[-6:-2]
Out[80]: [6, 3, 5, 6]
```

A semântica de fatiamento exige um pouco para que nos acostumemos com ela, especialmente se sua experiência anterior for com R ou com o MATLAB. Veja a Figura 3.1 que apresenta uma ilustração conveniente do fatiamento com inteiros positivos e negativos. Na figura, os índices são exibidos nas “bordas das áreas de armazenagem” para ajudar a mostrar em que ponto as seleções de fatias começam e terminam usando índices positivos e negativos.

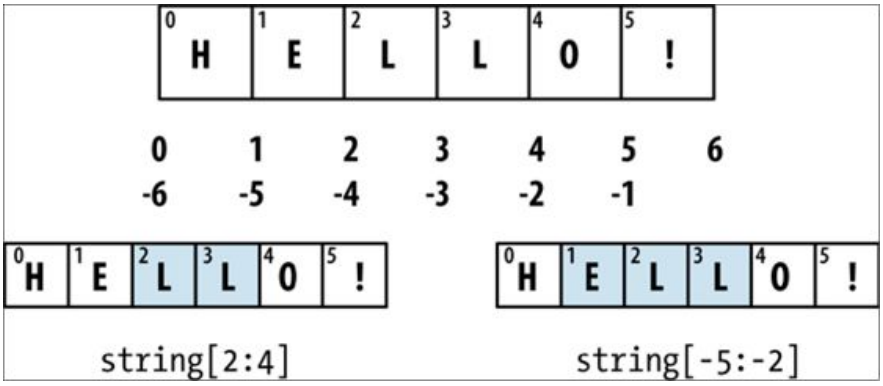


Figura 3.1 – Ilustração das convenções de fatiamento de Python.

Um step também pode ser usado depois de um segundo dois-pontos para, por exemplo, obter elementos alternados:

```
In [81]: seq[::2]
Out[81]: [7, 3, 3, 6, 1]
```

Um uso inteligente desse recurso consiste em passar -1, que tem o efeito conveniente de inverter uma lista ou uma tupla:

```
In [82]: seq[::-1]
Out[82]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

Funções embutidas para sequências

Python tem um punhado de funções de sequência úteis com as quais você deve ter familiaridade, usando-as em qualquer oportunidade.

enumerate

Querer manter o controle do índice do item atual quando iteramos por uma sequência é comum. Uma abordagem de implementação por conta própria seria assim:

```
i = 0
for value in collection:
   # faz algo com value
   i += 1
```

Como isso é bem comum, Python tem uma função embutida, enumerate, que devolve uma sequência de tuplas (i, value):

```
for i, value in enumerate(collection):
   # faz algo com value
```

Quando estiver indexando dados, um padrão conveniente, que utiliza enumerate, é calcular um dict que mapeie os valores (supostamente únicos) aos seus locais na sequência:

```
In [83]: some_list = ['foo', 'bar', 'baz']

In [84]: mapping = {}

In [85]: for i, v in enumerate(some_list):
....: mapping[v] = i

In [86]: mapping
Out[86]: {'bar': 1, 'baz': 2, 'foo': 0}
```

sorted

A função sorted devolve uma nova lista ordenada a partir dos elementos de qualquer sequência:

```
In [87]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[87]: [0, 1, 2, 2, 3, 6, 7]
```

```
In [88]: sorted('horse race')
Out[88]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

A função sorted aceita os mesmos argumentos que o método sort em listas.

zip

zip "pareia" os elementos de uma série de listas, tuplas ou outras sequências para criar uma lista de tuplas:

```
In [89]: seq1 = ['foo', 'bar', 'baz']

In [90]: seq2 = ['one', 'two', 'three']

In [91]: zipped = zip(seq1, seq2)

In [92]: list(zipped)
Out[92]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

zip pode aceitar um número arbitrário de sequências, e o número de elementos que ele gera é determinado pela sequência *mais curta*:

```
In [93]: seq3 = [False, True]

In [94]: list(zip(seq1, seq2, seq3))
Out[94]: [('foo', 'one', False), ('bar', 'two', True)]
```

Um uso bem comum de zip é quando iteramos simultaneamente por várias sequências, possivelmente combinando também com enumerate:

```
In [95]: for i, (a, b) in enumerate(zip(seq1, seq2)):
   ....: print('{0}: {1}, {2}'.format(i, a, b))
   ....:
0: foo, one
1: bar, two
2: baz, three
```

Dada uma sequência "pareada" (zipped), zip pode ser aplicado de forma inteligente para "desparear" (unzip) a sequência. Outra forma de pensar nisso é converter uma lista de *linhas* em uma lista de *colunas.* Eis a sintaxe, que parece um pouco mágica:

```
In [96]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
   ....: ('Schilling', 'Curt')]

In [97]: first_names, last_names = zip(*pitchers)

In [98]: first_names
Out[98]: ('Nolan', 'Roger', 'Schilling')

In [99]: last_names
Out[99]: ('Ryan', 'Clemens', 'Curt')
```

reversed

reversed itera pelos elementos de uma sequência na ordem inversa:

```
In [100]: list(reversed(range(10)))
Out[100]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Tenha em mente que reversed é um gerador (será discutido com um pouco mais de detalhes posteriormente), portanto ele não criará a sequência invertida até que ela seja materializada (por exemplo, com list ou com um laço for).

dict

dict provavelmente é a estrutura de dados embutida mais importante de Python. Um nome mais comum para ele é *hash map* ou *array associativo*. Consiste em uma coleção de pares *chave-valor de* tamanho flexível, em que *chave* e *valor* são objetos Python. Uma abordagem para criar um dicionário é usar chaves {} e dois-pontos para separar as chaves e os valores:

```
In [101]: empty_dict = {}

In [102]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}

In [103]: d1
Out[103]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

Você pode acessar, inserir ou definir elementos usando a mesma sintaxe utilizada para acessar elementos de uma lista ou de uma tupla:

```
In [104]: d1[7] = 'an integer'

In [105]: d1
Out[105]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}

In [106]: d1['b']
Out[106]: [1, 2, 3, 4]
```

Podemos verificar se um dicionário contém uma chave usando a mesma sintaxe utilizada para verificar se uma lista ou tupla contém um valor:

```
In [107]: 'b' in d1
Out[107]: True
```

É possível apagar valores usando a palavra reservada `del` ou com o método `pop` (que simultaneamente devolve o valor e apaga a chave):

```
In [108]: d1[5] = 'some value'

In [109]: d1
Out[109]:
{5: 'some value',
 7: 'an integer',
 'a': 'some value',
 'b': [1, 2, 3, 4]}

In [110]: d1['dummy'] = 'another value'
In [111]: d1
Out[111]:
{5: 'some value',
 7: 'an integer',
 'a': 'some value',
 'b': [1, 2, 3, 4],
'dummy': 'another value'}

In [112]: del d1[5]

In [113]: d1
Out[113]:
{7: 'an integer',
 'a': 'some value',
 'b': [1, 2, 3, 4],
```

```
 'dummy': 'another value'}

In [114]: ret = d1.pop('dummy')

In [115]: ret
Out[115]: 'another value'

In [116]: d1
Out[116]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}
```

Os métodos key e values oferecem iteradores para as chaves e valores do dicionário, respectivamente. Embora os pares chave-valor não estejam em nenhuma ordem em particular, essas funções devolvem as chaves e os valores na mesma ordem:

```
In [117]: list(d1.keys())
Out[117]: ['a', 'b', 7]

In [118]: list(d1.values())
Out[118]: ['some value', [1, 2, 3, 4], 'an integer']
```

Podemos combinar um dicionário com outro usando o método update:

```
In [119]: d1.update({'b' : 'foo', 'c' : 12})

In [120]: d1
Out[120]: {7: 'an integer', 'a': 'some value', 'b': 'foo', 'c': 12}
```

O método update altera os dicionários in-place, portanto qualquer chave existente nos dados passados para update terão seus valores antigos descartados.

Criando dicionários a partir de sequências

É comum ocasionalmente acabar com duas sequências cujos elementos você queira parear em um dicionário. Como primeira tentativa, poderíamos escrever um código assim:

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

Considerando que um dicionário é essencialmente uma coleção de

tuplas de 2, a função dict aceita uma lista de tuplas de 2:

```
In [121]: mapping = dict(zip(range(5), reversed(range(5))))
In [122]: mapping
Out[122]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Mais adiante discutiremos as *dict comprehensions* (abrangências de dicionário), que são outra forma elegante de construir dicionários.

Valores default

É muito comum ter uma lógica como esta:

```
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value
```

Assim, os métodos de dicionário get e pop podem aceitar um valor default a ser devolvido, de modo que é possível escrever o bloco if-else anterior simplesmente como:

```
value = some_dict.get(key, default_value)
```

Por padrão, get devolverá None se a chave não estiver presente, enquanto pop lançará uma exceção. Com valores a serem *definidos*, um caso comum é aquele em que os valores de um dicionário são outras coleções, como listas. Por exemplo, poderíamos imaginar a classificação de uma lista de palavras de acordo com suas primeiras letras como um dicionário de listas:

```
In [123]: words = ['apple', 'bat', 'bar', 'atom', 'book']

In [124]: by_letter = {}

In [125]: for word in words:
   .....: letter = word[0]
   .....: if letter not in by_letter:
   .....: by_letter[letter] = [word]
   .....: else:
   .....: by_letter[letter].append(word)
   .....:
```

```
In [126]: by_letter
Out[126]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

O método de dicionário setdefault serve exatamente para essa finalidade. O laço for anterior pode ser reescrito assim:

```
for word in words:
    letter = word[0]
    by_letter.setdefault(letter, []).append(word)
```

O módulo embutido collections tem uma classe útil, defaultdict, que torna isso mais fácil ainda. Para criar um dicionário, passe um tipo ou uma função para gerar o valor default de cada posição no dicionário:

```
from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
    by_letter[word[0]].append(word)
```

Tipos de chave válidos para dicionários

Enquanto os valores de um dicionário podem ser qualquer objeto Python, as chaves geralmente têm que ser objetos imutáveis, como tipos escalares (int, float, string) ou tuplas (todos os objetos na tupla devem ser imutáveis também). O termo técnico, nesse caso, é *hashability* (possibilidade de hashing). É possível verificar se um objeto é hashable (pode ser usado como chave em um dicionário) usando a função hash:

```
In [127]: hash('string')
Out[127]: -305472831944956388

In [128]: hash((1, 2, (2, 3)))
Out[128]: 1097636502276347782

In [129]: hash((1, 2, [2, 3])) # falha porque as listas são mutáveis
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-129-473c35a62c0b> in <module>()
----> 1 hash((1, 2, [2, 3])) # falha porque as listas são mutáveis
TypeError: unhashable type: 'list'
```

Para usar uma lista como chave, uma opção é convertê-la em uma tupla, que é hashable, desde que seus elementos também o sejam:

```
In [130]: d = {}

In [131]: d[tuple([1, 2, 3])] = 5

In [132]: d
Out[132]: {(1, 2, 3): 5}
```

set

Um conjunto (set) é uma coleção não ordenada de elementos únicos. Podemos pensar neles como dicionários, mas somente com as chaves, sem os valores. Um conjunto pode ser criado de duas maneiras: com a função `set` ou por meio de um *conjunto literal*, com chaves:

```
In [133]: set([2, 2, 2, 1, 3, 3])
Out[133]: {1, 2, 3}

In [134]: {2, 2, 2, 1, 3, 3}
Out[134]: {1, 2, 3}
```

Os conjuntos aceitam *operações matemáticas de conjunto*, como união, intersecção, diferença e diferença simétrica. Considere os dois exemplos de conjuntos a seguir:

```
In [135]: a = {1, 2, 3, 4, 5}

In [136]: b = {3, 4, 5, 6, 7, 8}
```

A união desses dois conjuntos é o conjunto de elementos distintos presentes em qualquer conjunto. Esse conjunto pode ser calculado com o método `union` ou com o operador binário `|`:

```
In [137]: a.union(b)
Out[137]: {1, 2, 3, 4, 5, 6, 7, 8}

In [138]: a | b
Out[138]: {1, 2, 3, 4, 5, 6, 7, 8}
```

A intersecção contém os elementos presentes nos dois conjuntos. O

operador & ou o método intersection podem ser usados:

```
In [139]: a.intersection(b)
Out[139]: {3, 4, 5}

In [140]: a & b
Out[140]: {3, 4, 5}
```

Veja a Tabela 3.1 que contém uma lista dos métodos de conjunto comumente utilizados.

Tabela 3.1 – Operações de conjunto de Python

Função	Sintaxe alternativa	Descrição
a.add(x)	N/A	Adiciona o elemento x ao conjunto a
a.clear()	N/A	Reinicia o conjunto a, deixando-o em um estado vazio, descartando todos os seus elementos
a.remove(x)	N/A	Remove o elemento x do conjunto a
a.pop()	N/A	Remove um elemento arbitrário do conjunto a, gerando um KeyError se o conjunto estiver vazio
a.union(b)	a | b	Todos os elementos únicos em a e b
a.update(b)	a |= b	Define o conteúdo de a como a união dos elementos em a e b
a.intersection(b)	a & b	Todos os elementos *tanto* em a *quanto* em b
a.intersection_update(b)	a &= b	Define o conteúdo de a como a intersecção dos elementos em a e em b
a.difference(b)	a - b	Os elementos em a que não estão em b
a.difference_update(b)	a -= b	Define a com os elementos que estão em a, mas que não estão em b

Função	Sintaxe alternativa	Descrição
a.symmetric_difference(b)	a ^ b	Todos os elementos que estão em a ou em b, mas *não em ambos*
a.symmetric_difference_update(b)	a ^= b	Define a para que contenha os elementos que estão em a ou em b, mas *não em ambos*
a.issubset(b)	N/A	True se os elementos de a estiverem todos contidos em b
a.issuperset(b)	N/A	True se os elementos de b estiverem todos contidos em a
a.isdisjoint(b)	N/A	True se a e b não tiverem elementos em comum

Todas as operações lógicas de conjunto têm contrapartidas in-place, o que permite a você substituir o conteúdo do conjunto à esquerda da operação com o resultado. Para conjuntos muito grandes, isso talvez seja eficiente:

```
In [141]: c = a.copy()

In [142]: c |= b

In [143]: c
Out[143]: {1, 2, 3, 4, 5, 6, 7, 8}

In [144]: d = a.copy()

In [145]: d &= b

In [146]: d
Out[146]: {3, 4, 5}
```

Assim como nos dicionários, os elementos de conjuntos geralmente devem ser imutáveis. Para ter elementos do tipo lista, você deverá fazer a conversão para uma tupla:

```
In [147]: my_data = [1, 2, 3, 4]

In [148]: my_set = {tuple(my_data)}
```

```
In [149]: my_set
Out[149]: {(1, 2, 3, 4)}
```

Também podemos verificar se um conjunto é um subconjunto (está contido) ou é um superconjunto (contém todos os elementos) de outro conjunto:

```
In [150]: a_set = {1, 2, 3, 4, 5}

In [151]: {1, 2, 3}.issubset(a_set)
Out[151]: True

In [152]: a_set.issuperset({1, 2, 3})
Out[152]: True
```

Os conjuntos serão iguais se, e somente se, seus conteúdos forem iguais:

```
In [153]: {1, 2, 3} == {3, 2, 1}
Out[153]: True
```

List, set e dict comprehensions

List comprehensions (abrangências de lista) são um dos recursos mais amados da linguagem Python. Elas permitem que você componha uma nova lista de modo conciso, filtrando os elementos de uma coleção, transformando os elementos ao passar o filtro, com uma expressão concisa. As list comprehensions assumem o seguinte formato básico:

```
[expr for val in collection if condition]
```

Isso é equivalente ao laço for a seguir:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

A condição de filtro pode ser omitida, restando somente a expressão. Por exemplo, dada uma lista de strings, poderíamos filtrar, eliminando aquelas de tamanho 2 ou menores, e também

converter as strings para letras maiúsculas, assim:

```
In [154]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']

In [155]: [x.upper() for x in strings if len(x) > 2]
Out[155]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Set e dict comprehensions são uma extensão natural, gerando conjuntos e dicionários de modo semelhante e idiomático, em vez de listas. Uma dict comprehension tem o seguinte aspecto:

```
dict_comp = {expr-chave : expr-valor for value in collection
             if condition}
```

Uma set comprehension se assemelha a uma list comprehension equivalente, exceto pelas chaves no lugar dos colchetes:

```
set_comp = {expr for value in collection if condition}
```

Assim como as list comprehensions, as set e dict comprehensions são, em sua maior parte, conveniências, mas, de modo semelhante, podem deixar o código mais fácil tanto para escrever quanto para ler. Considere a lista de strings anterior. Suponha que quiséssemos um conjunto contendo apenas os tamanhos das strings contidas na coleção; poderíamos facilmente calcular isso utilizando uma set comprehension:

```
In [156]: unique_lengths = {len(x) for x in strings}

In [157]: unique_lengths
Out[157]: {1, 2, 3, 4, 6}
```

Também poderíamos expressar isso de modo mais funcional usando a função map, que será apresentada em breve:

```
In [158]: set(map(len, strings))
Out[158]: {1, 2, 3, 4, 6}
```

Como um exemplo simples de dict comprehension, poderíamos criar um mapa de consulta dessas strings para seus locais na lista:

```
In [159]: loc_mapping = {val : index for index, val in enumerate(strings)}

In [160]: loc_mapping
Out[160]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

List comprehensions aninhadas

Suponha que tenhamos uma lista de listas contendo alguns nomes em inglês e em espanhol:

```
In [161]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
   .....: ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar']]
```

Você poderia ter obtido esses nomes de dois arquivos e decidiu organizá-los por idioma. Suponha agora que quiséssemos obter uma única lista contendo todos os nomes com dois ou mais e's. Certamente poderíamos fazer isso com um laço for simples:

```
names_of_interest = []
for names in all_data:
    enough_es = [name for name in names if name.count('e') >= 2]
    names_of_interest.extend(enough_es)
```

Na verdade, podemos encapsular toda essa operação em uma única *list comprehension aninhada*, que terá o seguinte aspecto:

```
In [162]: result = [name for names in all_data for name in names
   .....: if name.count('e') >= 2]


In [163]: result
Out[163]: ['Steven']
```

À primeira vista, list comprehensions aninhadas são um pouco difíceis de compreender. As partes com for da list comprehension são organizadas de acordo com a ordem em que estão aninhadas, e qualquer condição de filtragem será colocada no final, como antes. Eis um outro exemplo em que "linearizamos" uma lista de tuplas de inteiros, convertendo-a em uma lista simples:

```
In [164]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [165]: flattened = [x for tup in some_tuples for x in tup]

In [166]: flattened
Out[166]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Tenha em mente que a ordem das expressões for seria a mesma se você escrevesse um laço for aninhado em vez de uma list

comprehension:

```
flattened = []

for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```

Podemos ter arbitrariamente muitos níveis de código aninhado, embora, caso haja mais de dois ou três níveis, provavelmente você comece a se questionar se isso faz sentido do ponto de vista da legibilidade do código. É importante distinguir a sintaxe que acabamos de mostrar de uma list comprehension dentro de uma list comprehension, que é também perfeitamente válida:

```
In [167]: [[x for x in tup] for tup in some_tuples]
Out[167]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Isso gera uma lista de listas, em vez de gerar uma lista linearizada com todos os elementos mais internos.

3.2 Funções

As funções são o principal e mais importante método de organização de código e reutilização em Python. Como regra geral, se você prevê que precisará repetir o mesmo código ou um código muito semelhante mais de uma vez, talvez valha a pena escrever uma função reutilizável. As funções também podem ajudar você a deixar o seu código mais legível ao dar um nome a um grupo de instruções Python.

As funções são declaradas com a palavra reservada `def`, e o retorno é feito com a palavra reservada `return`:

```
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

Não há problemas em ter várias instruções `return`. Se Python alcançar o final de uma função sem encontrar uma instrução `return`,

None será devolvido automaticamente.

Toda função pode ter argumentos *posicionais (positional arguments) e* argumentos *nomeados* (keyword arguments). Argumentos nomeados são mais comumente utilizados para especificar valores default ou argumentos opcionais. Na função anterior, x e y são argumentos posicionais, enquanto z é um argumento nomeado. Isso significa que a função pode ser chamada de qualquer uma destas maneiras:

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
my_function(10, 20)
```

A principal restrição para os argumentos de função é que os argumentos nomeados *devem* vir depois dos argumentos posicionais (se houver). Você pode especificar argumentos nomeados em qualquer ordem; isso evita que precise se lembrar da ordem em que os argumentos da função foram especificados, sendo necessário se lembrar somente de seus nomes.

É possível usar nomes para passar argumentos posicionais também. No exemplo anterior, poderíamos ter escrito o seguinte:

```
my_function(x=5, y=6, z=7)
my_function(y=6, x=5, z=7)
```

Em alguns casos, isso ajudará na legibilidade.

Namespaces, escopo e funções locais

As funções podem acessar variáveis em dois escopos diferentes: *global* e *local*. Um nome alternativo e mais descritivo para o escopo de uma variável em Python é *namespace.* Quaisquer variáveis que recebam uma atribuição em uma função, por padrão são associadas ao namespace local, criado quando a função é chamada e imediatamente preenchido com os argumentos da função. Depois que a função termina, o namespace local é destruído (com algumas exceções que estão além da abrangência deste capítulo). Considere a função a seguir:

```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

Quando `func()` é chamada, a lista vazia `a` é criada, cinco elementos são concatenados e então `a` é destruída quando a função termina. Suponha que, em vez disso, tivéssemos declarado `a` assim:

```
a = []
def func():
    for i in range(5):
        a.append(i)
```

Fazer uma atribuição a uma variável fora do escopo da função é possível, porém essas variáveis devem ser declaradas como globais com a palavra reservada `global`:

```
In [168]: a = None

In [169]: def bind_a_variable():
   .....: global a
   .....: a = []
   .....: bind_a_variable()
   .....:

In [170]: print(a)
[]
```

Geralmente não incentivo o uso da palavra reservada global. As variáveis globais são comumente usadas para armazenar algum tipo de estado em um sistema. Se você se vir usando muitas delas, talvez isso indique a necessidade de utilizar programação orientada a objetos (com classes).

Devolvendo diversos valores

Quando comecei a programar em Python depois de ter programado em Java e em C++, um de meus recursos favoritos era a capacidade de devolver diversos valores de uma função usando uma sintaxe simples. Eis um exemplo:

```
def f():
    a = 5
    b = 6
    c = 7
    return a, b, c

a, b, c = f()
```

Em análise de dados e em outras aplicações científicas, você poderá se ver fazendo isso com frequência. O que está acontecendo nesse exemplo é que a função, na verdade, devolve apenas *um* objeto, uma tupla no caso, que está então sendo desempacotado nas variáveis de resultado. No exemplo anterior, poderíamos ter feito o seguinte, como alternativa:

```
return_value = f()
```

Nesse caso, `return_value` seria uma tupla de três, com as três variáveis devolvidas. Uma alternativa possivelmente atraente a devolver diversos valores como fizemos antes pode ser devolver um dicionário em seu lugar:

```
def f():
    a = 5
    b = 6
    c = 7
    return {'a' : a, 'b' : b, 'c' : c}
```

Essa técnica alternativa pode ser útil dependendo do que você estiver tentando fazer.

Funções são objetos

Como as funções Python são objetos, muitas construções podem ser facilmente expressas, as quais seriam difíceis em outras linguagens. Suponha que estejamos fazendo uma limpeza de dados e precisemos aplicar uma série de transformações na lista de strings a seguir:

```
In [171]: states = [' Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda',
   .....: 'south carolina##', 'West virginia?']
```

Qualquer pessoa que já tenha trabalhado alguma vez com dados de pesquisa submetidos por usuários já viu resultados desorganizados como esses. Muitas tarefas precisam ser feitas para deixar essa lista de strings uniforme e pronta para a análise: remover espaços em branco e símbolos de pontuação e criar um padrão para ter um uso adequado de letras maiúsculas ou não no início das palavras. Uma forma de fazer isso é usar métodos embutidos de string, junto com o módulo re da biblioteca-padrão para expressões regulares:

```
import re

def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub('[!#?]', '', value)
        value = value.title()
        result.append(value)
    return result
```

O resultado tem o seguinte aspecto:

```
In [173]: clean_strings(states)
Out[173]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Uma abordagem alternativa, que talvez lhe pareça conveniente, é criar uma lista das operações que você deseja aplicar em um conjunto particular de strings:

```
def remove_punctuation(value):
    return re.sub('[!#?]', '', value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
```

```
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
```

Então teremos o seguinte:

```
In [175]: clean_strings(states, clean_ops)
Out[175]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Um padrão mais *funcional* como esse permite modificar facilmente o modo como as strings são transformadas em um nível bem alto. A função clean_strings agora também está mais reutilizável e genérica.

Podemos usar funções como argumentos para outras funções, como a função embutida map, que aplica uma função a uma sequência de algum tipo:

```
In [176]: for x in map(remove_punctuation, states):
   .....: print(x)
Alabama
Georgia
Georgia
georgia
FlOrIda
south carolina
West virginia
```

Funções anônimas (lambdas)

Python oferece suporte para o que chamamos de funções *anônimas* ou *lambdas* – uma forma de escrever funções constituídas de uma única instrução, cujo resultado é o valor de retorno. Elas são

definidas com a palavra reservada lambda, que não tem outro significado a não ser dizer que “estamos declarando uma função anônima”:

```
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

Em geral, eu me referirei a elas como funções lambda no restante do livro. Essas funções são particularmente convenientes na análise de dados, pois, como veremos, há muitos casos em que as funções de transformação de dados aceitarão funções como argumentos. Geralmente, exige-se menos digitação (e é mais claro) passar uma função lambda, em oposição a escrever uma declaração de função completa ou até mesmo atribuir a função lambda a uma variável local. Por exemplo, considere o exemplo simplório a seguir:

```
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

Você poderia também ter escrito [x * 2 for x in ints], mas, nesse caso, pudemos passar sucintamente um operador personalizado para a função apply_to_list.

Como outro exemplo, suponha que quiséssemos ordenar uma coleção de strings de acordo com o número de letras distintas em cada string:

```
In [177]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

Nesse caso, poderíamos passar uma função lambda para o método sort da lista:

```
In [178]: strings.sort(key=lambda x: len(set(list(x))))

In [179]: strings
Out[179]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

 Um dos motivos para as funções lambda serem chamadas de funções anônimas é que, de modo diferente das funções declaradas com a palavra reservada def, o objeto função propriamente dito jamais recebe um atributo __name__ explícito.

Currying: aplicação parcial dos argumentos

Currying faz parte do jargão da ciência da computação (proveniente do nome do matemático Haskell Curry) e significa derivar novas funções a partir de funções existentes por meio da *aplicação parcial de argumentos*. Por exemplo, suponha que tenhamos uma função trivial que some dois números:

```
def add_numbers(x, y):
    return x + y
```

Usando essa função, poderíamos derivar uma nova função de uma variável, add_five, que some 5 ao seu argumento:

```
add_five = lambda y: add_numbers(5, y)
```

Dizemos que o segundo argumento de add_numbers sofreu *currying*. Não há nada muito sofisticado nesse caso, pois tudo que realmente fizemos foi definir uma nova função que chama uma função existente. O módulo embutido functools pode simplificar esse processo usando a função partial:

```
from functools import partial
add_five = partial(add_numbers, 5)
```

Geradores

Ter uma forma consistente de iterar por sequências, como objetos em uma lista ou linhas em um arquivo, é um recurso importante de Python. Isso é feito por meio do *protocolo iterador* – um modo genérico de deixar os objetos iteráveis. Por exemplo, iterar por um dicionário produz as suas chaves:

```
In [180]: some_dict = {'a': 1, 'b': 2, 'c': 3}
```

```
In [181]: for key in some_dict:
   .....: print(key)
a
b
c
```

Quando escrevemos `for key in some_dict`, o interpretador Python inicialmente tenta criar um iterador a partir de `some_dict`:

```
In [182]: dict_iterator = iter(some_dict)

In [183]: dict_iterator
Out[183]: <dict_keyiterator at 0x7ff84e90ee58>
```

Um iterador é qualquer objeto que produzirá objetos ao interpretador Python quando usado em um contexto como um laço `for`. A maioria dos métodos que espera uma lista ou um objeto do tipo lista também aceitará qualquer objeto iterável. Isso inclui métodos embutidos como `min`, `max` e `sum`, além de construtores de tipo como `list` e `tuple`:

```
In [184]: list(dict_iterator)
Out[184]: ['a', 'b', 'c']
```

Um *gerador* é uma forma concisa de construir um novo objeto iterável. Enquanto funções usuais executam e devolvem um único resultado de cada vez, os geradores devolvem uma sequência de vários resultados em modo lazy, fazendo uma pausa após cada um, até que o próximo resultado seja solicitado. Para criar um gerador, utilize a palavra reservada `yield` em vez de `return` em uma função:

```
def squares(n=10):
    print('Generating squares from 1 to {0}'.format(n ** 2))
    for i in range(1, n + 1):
        yield i ** 2
```

Quando você realmente chamar o gerador, nenhum código será executado imediatamente:

```
In [186]: gen = squares()

In [187]: gen
Out[187]: <generator object squares at 0x7ff84e92bbf8>
```

O gerador só passará a executar o seu código quando você começar a lhe solicitar elementos:

```
In [188]: for x in gen:
   .....: print(x, end=' ')
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

Expressões geradoras

Outra forma mais concisa ainda de criar um gerador é usando uma *expressão geradora.* É um gerador análogo a list, dict e set comprehensions; para criar um, coloque o que seria uma list comprehension entre parênteses em vez de usar colchetes:

```
In [189]: gen = (x ** 2 for x in range(100))

In [190]: gen
Out[190]: <generator object <genexpr> at 0x7ff84e92b150>
```

Isso é totalmente equivalente ao gerador a seguir, mais extenso:

```
def _make_gen():
    for x in range(100):
        yield x ** 2
gen = _make_gen()
```

Expressões geradoras podem ser usadas no lugar de list comprehensions como argumentos de função em muitos casos:

```
In [191]: sum(x ** 2 for x in range(100))
Out[191]: 328350

In [192]: dict((i, i **2) for i in range(5))
Out[192]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Módulo itertools

O módulo itertools da biblioteca-padrão tem uma coleção de geradores para muitos algoritmos comuns de dados. Por exemplo, groupby aceita qualquer sequência e uma função, agrupando elementos consecutivos da sequência de acordo com o valor de retorno da função. Eis um exemplo:

```
In [193]: import itertools

In [194]: first_letter = lambda x: x[0]

In [195]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']

In [196]: for letter, names in itertools.groupby(names, first_letter):
   .....: print(letter, list(names)) # names é um gerador
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

Veja a Tabela 3.2 que mostra uma lista de algumas das demais funções de itertools que, com frequência, acho úteis. Talvez você queira dar uma olhada na documentação oficial de Python (*https://docs.python.org/3/library/itertools.html*) para ver mais informações sobre esse conveniente módulo embutido utilitário.

Tabela 3.2 – Algumas funções úteis de itertools

Função	Descrição
combinations(iterable, k)	Gera uma sequência de todas as tuplas possíveis de k elementos no iterável, ignorando a ordem e sem substituição (veja também a função companheira combinations_with_replacement)
permutations(iterable, k)	Gera uma sequência de todas as tuplas possíveis de k elementos no iterável, respeitando a ordem
groupby(iterable[, keyfunc])	Gera (key, sub-iterator) para cada chave única
product(*iterables, repeat=1)	Gera o produto cartesiano dos iteráveis de entrada como tuplas, de modo semelhante a um laço for aninhado

Erros e tratamento de exceção

Tratar erros ou *exceções* em Python de modo elegante é uma parte importante na construção de programas robustos. Em aplicações de análise de dados, muitas funções servem somente para determinados tipos de entrada. Como exemplo, a função float de Python é capaz de fazer cast de uma string para um número de

ponto flutuante, porém falha com ValueError se os dados de entrada forem inadequados:

```
In [197]: float('1.2345')
Out[197]: 1.2345

In [198]: float('something')
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)
<ipython-input-198-2649e4ade0e6> in <module>()
----> 1 float('something')
ValueError: could not convert string to float: 'something'
```

Suponha que quiséssemos uma versão de float que falhasse de modo elegante, devolvendo o argumento de entrada. Podemos fazer isso escrevendo uma função que encapsule a chamada a float em um bloco try/except:

```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x
```

O código na parte except do bloco será executado somente se float(x) gerar uma exceção:

```
In [200]: attempt_float('1.2345')
Out[200]: 1.2345

In [201]: attempt_float('something')
Out[201]: 'something'
```

Note que float é capaz de gerar outras exceções além de ValueError:

```
In [202]: float((1, 2))
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-202-82f777b0e564> in <module>()
----> 1 float((1, 2))
TypeError: float() argument must be a string or a number, not 'tuple'
```

Talvez você queira suprimir apenas ValueError, pois um TypeError (a entrada não era um valor de string nem um valor numérico) pode

indicar um bug legítimo em seu programa. Para fazer isso, escreva o tipo da exceção após except:

```
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x
```

Então temos:

```
In [204]: attempt_float((1, 2))
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-204-8b0026e9e6b7> in <module>()
----> 1 attempt_float((1, 2))
<ipython-input-203-d99a2a135508> in attempt_float(x)
      1 def attempt_float(x):
      2 try:
----> 3 return float(x)
      4 except ValueError:
      5 return x
TypeError: float() argument must be a string or a number, not 'tuple'
```

Você pode capturar vários tipos de exceção escrevendo uma tupla de tipos de exceção (os parênteses são necessários):

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

Em alguns casos, talvez você não queira suprimir uma exceção, mas queira que algum código seja executado, independentemente de o código no bloco try ter sido bem-sucedido ou não. Para fazer isso, utilize finally:

```
f = open(path, 'w')

try:
    write_to_file(f)
finally:
    f.close()
```

Nesse caso, o handle do arquivo f *sempre* será fechado. De modo semelhante, podemos ter um código que execute somente se o bloco try: for bem-sucedido usando else:

```
f = open(path, 'w')

try:
    write_to_file(f)
except:
    print('Failed')
else:
    print('Succeeded')
finally:
    f.close()
```

Exceções no IPython

Se uma exceção for gerada enquanto você estiver executando um script com %run ou executando qualquer instrução, o IPython, por padrão, exibirá uma pilha de chamadas completa (um traceback), com algumas linhas de contexto em torno da posição em cada ponto na pilha:

```
In [10]: %run examples/ipython_bug.py
---------------------------------------------------------------------------
AssertionError Traceback (most recent call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
    13 throws_an_exception()
    14
---> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_things()
    11 def calling_things():
    12 works_fine()
---> 13 throws_an_exception()
    14
    15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in
throws_an_exception()
     7 a = 5
```

```
      8 b = 6
----> 9 assert(a + b == 10)
     10
     11 def calling_things():
AssertionError:
```

Ter um contexto adicional por si só é uma grande vantagem em relação ao interpretador Python padrão (que não provê nenhum contexto adicional). Você pode controlar o volume de contexto exibido usando o comando mágico *%xmode*, de Plain (mesmo que o interpretador Python padrão) a Verbose (que deixa os valores dos argumentos de função inline, além de apresentar outras informações). Como veremos mais adiante no capítulo, podemos executar passo a passo *dentro da pilha* (usando as mágicas %debug ou %pdb) depois que um erro tiver ocorrido, para uma depuração interativa *post-mortem*.

3.3 Arquivos e o sistema operacional

A maior parte deste livro utiliza ferramentas de alto nível, como pandas.read_csv, a fim de ler arquivos de dados de disco para estruturas de dados Python. No entanto, é importante compreender o básico sobre como trabalhar com arquivos em Python. Felizmente é bem simples, o que constitui um dos motivos pelos quais Python é tão popular para lidar com textos e arquivos.

Para abrir um arquivo para leitura ou escrita, utilize a função embutida open com um path de arquivo relativo ou absoluto:

```
In [207]: path = 'examples/segismundo.txt'

In [208]: f = open(path)
```

Por padrão, o arquivo é aberto em modo somente de leitura 'r'. Podemos então tratar o handle de arquivo f como uma lista e iterar pelas linhas, assim:

```
for line in f:
    pass
```

As linhas são extraídas do arquivo com os marcadores de fim de

linha (EOL) intactos, portanto, com frequência, você verá um código para obter uma lista de linhas de um arquivo livres de EOL, da seguinte maneira:

```
In [209]: lines = [x.rstrip() for x in open(path)]

In [210]: lines
Out[210]:
['Sueña el rico en su riqueza,',
 'que más cuidados le ofrece;',
 '',
 'sueña el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sueña el que a medrar empieza,',
 'sueña el que afana y pretende,',
 'sueña el que agravia y ofende,',
 '',
 'y en el mundo, en conclusión,',
 'todos sueñan lo que son,',
 'aunque ninguno lo entiende.',
 '']
```

Quando se usa `open` para criar objetos de arquivo, é importante fechar explicitamente o arquivo ao acabar de utilizá-lo. Fechar o arquivo libera seus recursos, devolvendo-os para o sistema operacional:

```
In [211]: f.close()
```

Uma das maneiras de facilitar a limpeza de arquivos abertos é usar a instrução `with`:

```
In [212]: with open(path) as f:
   .....: lines = [x.rstrip() for x in f]
```

Isso fará com que o arquivo `f` seja automaticamente fechado quando saímos do bloco `with`.

Se tivéssemos digitado `f = open(path, 'w')`, um *novo arquivo* em *examples/segismundo.txt* teria sido criado (tome cuidado!), sobrescrevendo qualquer um em seu lugar. Há também o modo de arquivo `'x'`, que cria um arquivo para escrita, porém falha se o path

do arquivo já existir. Veja a Tabela 3.3 que tem uma lista de todos os modos de arquivo válidos para leitura/escrita.

Para arquivos de leitura, alguns dos métodos mais comumente utilizados são read, seek e tell. read devolve um determinado número de caracteres do arquivo. O que constitui um "caractere" é determinado pela codificação do arquivo (por exemplo, UTF-8), ou simplesmente bytes puros se o arquivo for aberto em modo binário:

```
In [213]: f = open(path)

In [214]: f.read(10)
Out[214]: 'Sueña el r'

In [215]: f2 = open(path, 'rb') # Modo binário

In [216]: f2.read(10)
Out[216]: b'Sue\xc3\xb1a el '
```

O método read faz a posição do handle do arquivo avançar de acordo com o número de bytes lidos. tell devolve a posição atual:

```
In [217]: f.tell()
Out[217]: 11

In [218]: f2.tell()
Out[218]: 10
```

Apesar de termos lido 10 caracteres do arquivo, a posição é 11 porque foi necessária essa quantidade de bytes para decodificar 10 caracteres usando a codificação default. Podemos verificar a codificação default no módulo sys:

```
In [219]: import sys

In [220]: sys.getdefaultencoding()
Out[220]: 'utf-8'
```

seek altera a posição do arquivo para o byte aí indicado:

```
In [221]: f.seek(3)
Out[221]: 3

In [222]: f.read(1)
```

```
Out[222]: 'ñ'
```

Por fim, lembre-se de fechar os arquivos:

```
In [223]: f.close()
In [224]: f2.close()
```

Tabela 3.3 – Modos de arquivo em Python

Modo	Descrição
r	Modo somente de leitura
w	Modo somente de escrita; cria um novo arquivo (apagando os dados de qualquer arquivo com o mesmo nome)
x	Modo somente de escrita; cria um novo arquivo, mas falha se o path do arquivo já existir
a	Concatena no arquivo existente (cria o arquivo caso ele ainda não exista)
r+	Leitura e escrita
b	Adicione ao modo para arquivos binários (isto é, 'rb' ou 'wb')
t	Modo texto para arquivos (decodificando bytes automaticamente para Unicode). É o default se o modo não for especificado. Adicione t a outros modos para utilizar esse modo (isto é, 'rt' ou 'xt')

Para escrever texto em um arquivo, use os métodos de arquivo write ou writelines. Por exemplo, poderíamos criar uma versão de *prof_mod.py* sem linhas em branco, assim:

```
In [225]: with open('tmp.txt', 'w') as handle:
   .....: handle.writelines(x for x in open(path) if len(x) > 1)

In [226]: with open('tmp.txt') as f:
   .....: lines = f.readlines()

In [227]: lines
Out[227]:
['Sueña el rico en su riqueza,\n',
 'que más cuidados le ofrece;\n',
 'sueña el pobre que padece\n',
 'su miseria y su pobreza;\n',
 'sueña el que a medrar empieza,\n',
 'sueña el que afana y pretende,\n',
 'sueña el que agravia y ofende,\n',
 'y en el mundo, en conclusión,\n',
```

```
'todos sueñan lo que son,\n',
 'aunque ninguno lo entiende.\n']
```

Veja a Tabela 3.4 que tem vários dos métodos de arquivo comumente utilizados.

Tabela 3.4 – Métodos ou atributos de arquivo importantes em Python

Método	Descrição
read([size])	Devolve dados do arquivo como uma string, com o argumento opcional size indicando o número de bytes a ser lido
readlines([size])	Devolve uma lista de linhas do arquivo, com o argumento size opcional
write(str)	Escreve a string passada em um arquivo
writelines(strings)	Escreve a sequência de strings passada no arquivo
close()	Fecha o handle
flush()	Faz um flush do buffer de E/S interno (descarrega-o) no disco
seek(pos)	Move para a posição do arquivo indicada (inteiro)
tell()	Devolve a posição atual no arquivo como um inteiro
closed	True se o arquivo estiver fechado

Bytes e Unicode com arquivos

O comportamento padrão para arquivos Python (sejam de leitura ou de escrita) é o *modo texto*, que significa que você pretende trabalhar com strings Python (isto é, Unicode). Isso contrasta com o *modo binário*, que pode ser obtido concatenando b no modo do arquivo. Vamos observar o arquivo (que contém caracteres não ASCII com codificação UTF-8) da seção anterior:

```
In [230]: with open(path) as f:
   .....: chars = f.read(10)

In [231]: chars
Out[231]: 'Sueña el r'
```

O UTF-8 é uma codificação Unicode de tamanho variável, portanto, quando requisito certo número de caracteres do arquivo, Python lê bytes suficientes (que poderiam ser poucos como 10 ou muitos como 40 bytes) do arquivo para decodificar essa quantidade de

caracteres. Se, por outro lado, abrirmos o arquivo em modo 'rb', read requisitará números exatos de bytes:

```
In [232]: with open(path, 'rb') as f:
   .....: data = f.read(10)

In [233]: data
Out[233]: b'Sue\xc3\xb1a el '
```

Dependendo da codificação do texto, você poderá decodificar os bytes para um objeto str por conta própria, mas apenas se cada um dos caracteres Unicode codificados estiver totalmente formado:

```
In [234]: data.decode('utf8')
Out[234]: 'Sueña el '
In [235]: data[:4].decode('utf8')
---------------------------------------------------------------------------
UnicodeDecodeError Traceback (most recent call last)
<ipython-input-235-0ad9ad6a11bd> in <module>()
----> 1 data[:4].decode('utf8')
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 3:
unexpecte d end of data
```

O modo texto, combinado com a opção encoding de open, oferece uma forma conveniente de converter de uma codificação Unicode para outra:

```
In [236]: sink_path = 'sink.txt'

In [237]: with open(path) as source:
   .....: with open(sink_path, 'xt', encoding='iso-8859-1') as sink:
   .....: sink.write(source.read())

In [238]: with open(sink_path, encoding='iso-8859-1') as f:
   .....: print(f.read(10))
Sueña el r
```

Tome cuidado ao usar seek quando abrir arquivos em outro modo que não o binário. Se a posição do arquivo acabar ficando no meio dos bytes que definem um caractere Unicode, então leituras subsequentes resultarão em um erro:

```
In [240]: f = open(path)
```

```
In [241]: f.read(5)
Out[241]: 'Sueña'

In [242]: f.seek(4)
Out[242]: 4

In [243]: f.read(1)
---------------------------------------------------------------------------
UnicodeDecodeError Traceback (most recent call last)
<ipython-input-243-5a354f952aa4> in <module>()
----> 1 f.read(1)
/miniconda/envs/book-env/lib/python3.6/codecs.py in decode(self, input, final)
    319 # decodifica input (levando o buffer em consideração)
    320 data = self.buffer + input
--> 321 (result, consumed) = self._buffer_decode(data, self.errors, final
)
    322 # mantém o input não codificado até a próxima chamada
    323 self.buffer = data[consumed:]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position 0: invalid
start byte
In [244]: f.close()
```

Se você se vir regularmente fazendo análise de dados em texto não ASCII, dominar a funcionalidade de Unicode de Python se mostrará importante. Consulte a documentação online de Python (*https://docs.python.org*) para obter muito mais informações.

3.4 Conclusão

De posse agora de parte do básico e do ambiente e da linguagem Python, é hora de prosseguir e conhecer o NumPy e o processamento orientado a arrays em Python.

CAPÍTULO 4

Básico sobre o NumPy: arrays e processamento vetorizado

O NumPy, abreviatura de Numerical Python (Python Numérico), é um dos pacotes básicos mais importantes para processamento numérico em Python. A maioria dos pacotes de processamento com funcionalidades científicas utiliza objetos array do NumPy como a *lingua franca* para troca de dados.

Eis alguns recursos que você encontrará no NumPy:

- ndarray: um array multidimensional eficaz que oferece operações aritméticas rápidas, orientadas a arrays, e recursos flexíveis de *broadcasting*;
- funções matemáticas para operações rápidas em arrays de dados inteiros, sem que seja necessário escrever laços;
- ferramentas para ler/escrever dados de array em disco e trabalhar com arquivos mapeados em memória;
- recursos para álgebra linear, geração de números aleatórios e transformadas de Fourier;
- uma API C para conectar o NumPy a bibliotecas escritas em C, C++ ou FORTRAN.

Pelo fato de o NumPy oferecer uma API C fácil de usar, passar dados para bibliotecas externas escritas em uma linguagem de baixo nível é simples, e o mesmo vale para bibliotecas externas devolverem dados ao Python como arrays NumPy. Esse recurso fez de Python uma linguagem preferida para encapsular bases de código legadas em C/C++/Fortran, oferecendo-lhes uma interface

dinâmica e fácil de usar.

Embora o NumPy por si só não ofereça funcionalidades científicas nem de modelagem, compreender os arrays NumPy e o processamento orientado a arrays ajudará você a utilizar ferramentas com semântica orientada a arrays, como o pandas, de modo muito mais eficaz. Como o NumPy é um assunto bem amplo, discutirei vários de seus recursos avançados, por exemplo, o broadcasting, de modo mais detalhado, posteriormente (veja o Apêndice A).

Para a maioria das aplicações de análise de dados, as principais áreas de funcionalidades em que manterei o foco são:

- operações rápidas em arrays vetorizados para tratamento e limpeza de dados, geração de subconjuntos e filtragem, transformações e outros tipos de processamentos;
- algoritmos comuns para arrays como ordenação, unicidade e operações de conjunto;
- estatísticas descritivas eficazes e agregação/sintetização de dados;
- alinhamento de dados e manipulações de dados relacionais para combinar e juntar conjuntos de dados heterogêneos;
- expressão de lógica condicional na forma de expressões de array em vez de laços com ramos `if-elif-else`;
- manipulações de dados em grupos (agregação, transformação e aplicação de função).

Embora o NumPy ofereça os fundamentos para processamento de dados numéricos em geral, muitos leitores vão querer utilizar o pandas como base para a maior parte dos tipos de estatísticas e análises, especialmente em dados tabulares. O pandas também oferece algumas funcionalidades mais específicas de domínios, como manipulação de séries temporais, que não estão presentes no NumPy.

O processamento orientado a arrays em Python tem suas raízes nos idos de 1995, quando Jim Hugunin criou a biblioteca Numeric. Nos dez anos seguintes, muitas comunidades de programação científica começaram a fazer programação com arrays em Python, mas o ecossistema de bibliotecas se tornou fragmentado no início dos anos 2000. Em 2005, Travis Oliphant conseguiu desenvolver o projeto NumPy a partir dos projetos Numeric e Numarray da época, de modo a reunir a comunidade em torno de um único framework de processamento de arrays.

Um dos motivos para o NumPy ser tão importante para processamentos numéricos em Python é o fato de ele ter sido projetado para ser eficaz em arrays de dados grandes. Há uma série de motivos para isso:

- Internamente, o NumPy armazena dados em um bloco contíguo de memória, independentemente de outros objetos Python embutidos. A biblioteca do NumPy de algoritmos escritos na linguagem C é capaz de atuar nessa memória sem qualquer verificação de tipo ou outro overhead. Os arrays NumPy também utilizam muito menos memória que as sequências embutidas de Python.
- As operações do NumPy realizam processamentos complexos em arrays inteiros sem a necessidade de laços for de Python.

Para dar uma ideia da diferença no desempenho, considere um array NumPy com um milhão de inteiros e a lista equivalente em Python:

```
In [7]: import numpy as np

In [8]: my_arr = np.arange(1000000)

In [9]: my_list = list(range(1000000))
```

Vamos agora multiplicar cada sequência por 2:

```
In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 20 ms, sys: 10 ms, total: 30 ms
Wall time: 31.3 ms
```

```
In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
CPU times: user 680 ms, sys: 180 ms, total: 860 ms
Wall time: 861 ms
```

Os algoritmos baseados no NumPy geralmente são de 10 a 100 vezes mais rápidos (ou mais) do que suas contrapartidas em Python puro, além de utilizarem significativamente menos memória.

4.1 O ndarray do NumPy: um objeto array multidimensional

Um dos principais recursos do NumPy é seu objeto array N-dimensional, ou ndarray, que é um contêiner rápido e flexível para conjuntos de dados grandes em Python. Os arrays permitem realizar operações matemáticas em blocos inteiros de dados usando uma sintaxe semelhante às operações equivalentes entre elementos escalares.

Para lhe dar uma amostra de como o NumPy possibilita fazer processamentos em lote com uma sintaxe semelhante àquela usada por valores escalares em objetos embutidos de Python, inicialmente importarei o NumPy e gerarei um pequeno array de dados aleatórios:

```
In [12]: import numpy as np

# Gera alguns dados aleatórios
In [13]: data = np.random.randn(2, 3)

In [14]: data
Out[14]:
array([[-0.2047, 0.4789, -0.5194],
       [-0.5557, 1.9658, 1.3934]])
```

Em seguida, escreverei operações matemáticas com data:

```
In [15]: data * 10
Out[15]:
array([[ -2.0471, 4.7894, -5.1944],
       [ -5.5573, 19.6578, 13.9341]])
```

```
In [16]: data + data
Out[16]:
array([[-0.4094, 0.9579, -1.0389],
       [-1.1115, 3.9316, 2.7868]])
```

No primeiro exemplo, todos os elementos foram multiplicados por 10. No segundo, os valores correspondentes em cada "célula" do array foram somados uns aos outros.

 Neste capítulo, e ao longo do livro, utilizo a convenção padrão do NumPy de sempre usar import numpy as np. É claro que você é bem-vindo para utilizar from numpy import * em seu código a fim de evitar ter que escrever np., mas aconselho a não fazer disso um hábito. O namespace numpy é extenso e contém uma série de funções cujos nomes entram em conflito com funções embutidas de Python (como min e max).

Um ndarray é um contêiner genérico multidimensional para dados homogêneos; isso significa que todos os elementos devem ser do mesmo tipo. Todo array tem um shape, isto é, uma tupla que indica o tamanho de cada dimensão, e um dtype, que é um objeto que descreve o *tipo de dado* do array:

```
In [17]: data.shape
Out[17]: (2, 3)

In [18]: data.dtype
Out[18]: dtype('float64')
```

Este capítulo apresentará o básico sobre o uso de arrays NumPy, e deverá ser suficiente para você acompanhar o restante do livro. Embora não seja necessário ter um profundo conhecimento do NumPy para muitas aplicações de análise de dados, tornar-se proficiente em programação e raciocínio orientados a arrays é um passo essencial no caminho de se tornar um guru científico em Python.

 Sempre que você vir "array," "NumPy array" ou "ndarray" no texto, com poucas exceções, todos eles se referirão ao mesmo item: o objeto ndarray.

Criando ndarrays

A maneira mais fácil de criar um array é usar a função array. Ela aceita qualquer objeto do tipo sequência (incluindo outros arrays) e gera um novo array NumPy contendo os dados recebidos. Por exemplo, uma lista é uma boa candidata para a conversão:

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([ 6. , 7.5, 8. , 0. , 1. ])
```

Sequências aninhadas, como uma lista de listas de mesmo tamanho, serão convertidas em um array multidimensional:

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [23]: arr2 = np.array(data2)

In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Como data2 era uma lista de listas, o array NumPy arr2 tem duas dimensões, com o formato inferido a partir dos dados. Podemos confirmar isso inspecionando os atributos ndim e shape:

```
In [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```

A menos que esteja explicitamente especificado (mais sobre assunto posteriormente), np.array tentará inferir um bom tipo de dado para o array que ele criar. O tipo de dado é armazenado em um objeto especial de metadados dtype; por exemplo, nos dois exemplos anteriores temos:

```
In [27]: arr1.dtype
```

```
Out[27]: dtype('float64')

In [28]: arr2.dtype
Out[28]: dtype('int64')
```

Além de np.array, há uma série de outras funções para criar novos arrays. Como exemplos, zeros e ones criam arrays de 0s ou de 1s respectivamente, com um dado tamanho ou formato. empty cria um array sem inicializar seus valores com qualquer valor em particular. Para criar um array com dimensões maiores usando esses métodos, passe uma tupla para o formato:

```
In [29]: np.zeros(10)
Out[29]: array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [30]: np.zeros((3, 6))
Out[30]:
array([[ 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0.]])

In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[ 0., 0.],
        [ 0., 0.],
        [ 0., 0.]],
       [[ 0., 0.],
        [ 0., 0.],
        [ 0., 0.]]])
```

Não é seguro supor que np.empty devolverá um array somente com zeros. Em alguns casos, valores “lixo” não inicializados poderão ser devolvidos.

arange é uma versão da função embutida range de Python com valor de array:

```
In [32]: np.arange(15)
Out[32]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

Veja a Tabela 4.1 que contém uma pequena lista das funções

padrões para criação de arrays. Já que o NumPy tem como foco o processamento numérico, o tipo de dado, em muitos casos, caso não seja especificado, será float64 (ponto flutuante).

Tabela 4.1 – Funções para criação de arrays

Função	Descrição
array	Converte os dados de entrada (lista, tupla, array ou outro tipo de sequência) em um ndarray, seja inferindo um dtype ou especificando-o explicitamente; copia os dados de entrada, por padrão
asarray	Converte a entrada para um ndarray, mas não copia se a entrada já for um ndarray
arange	Como a função embutida range, porém devolve um ndarray em vez de uma lista
ones, ones_like	Gera um array somente com 1s, com o formato e o dtype especificados; ones_like aceita outro array e gera um array de uns com o mesmo formato e o mesmo dtype
zeros, zeros_like	Como ones e ones_like, porém gerando arrays com 0s
empty, empty_like	Cria novos arrays alocando nova memória, mas não preenche com nenhum valor, como ones e zeros
full, full_like	Gera um array com o formato e o dtype especificados, com todos os valores definidos com o “valor de preenchimento” indicado. full_like aceita outro array e gera um array preenchido com o mesmo formato e o mesmo dtype
eye, identity	Cria uma matriz-identidade quadrada N x N (1s na diagonal e 0s nas demais posições)

Tipos de dados para ndarrays

O *tipo de dado* ou dtype é um objeto especial contendo as informações (ou metadados, isto é, dados sobre dados) que o ndarray precisa para interpretar uma porção de memória como um tipo de dado particular:

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
```

```
Out[35]: dtype('float64')

In [36]: arr2.dtype
Out[36]: dtype('int32')
```

Os dtypes são um dos motivos para a flexibilidade do NumPy em interagir com dados provenientes de outros sistemas. Na maioria dos casos, eles oferecem um mapeamento direto para um disco subjacente ou uma representação em memória, o que facilita ler e escrever streams de dados binários em disco, e também se conectar com um código escrito em uma linguagem de baixo nível como C ou Fortran. Os dtypes numéricos recebem o nome da mesma maneira: um nome de tipo, como float ou int, seguido de um número que indica a quantidade de bits por elemento. Um valor padrão de ponto flutuante com dupla precisão (que é usado internamente no objeto float de Python) ocupa 8 bytes, isto é, 64 bits. Desse modo, esse tipo é conhecido no NumPy como float64. Veja a Tabela 4.2 que contém uma lista completa dos tipos de dados aceitos pelo NumPy.

Não se preocupe em memorizar os dtypes do NumPy, especialmente se você for um novo usuário. Geralmente será necessário se preocupar apenas com o *tipo* de dado geral com o qual você estiver lidando, seja ponto flutuante, complexo, inteiro, booleano, string ou um objeto Python genérico. Quando você precisar ter mais controle sobre como os dados são armazenados na memória e em disco, especialmente no caso de conjuntos grandes de dados, será bom saber que você tem controle sobre o tipo de armazenagem.

Tabela 4.2 – Tipos de dados do NumPy

Tipo	Código do tipo	Descrição
int8, uint8	i1, u1	Tipos inteiros de 8 bits (1 byte) com e sem sinal
int16, uint16	i2, u2	Tipos inteiros de 16 bits com e sem sinal
int32, uint32	i4, u4	Tipos inteiros de 32 bits com e sem sinal
int64, uint64	i8, u8	Tipos inteiros de 64 bits com e sem sinal
float16	f2	Ponto flutuante com metade da precisão

Tipo	Código do tipo	Descrição
float32	f4 ou f	Ponto flutuante padrão com precisão única; compatível com o float de C
float64	f8 ou d	Ponto flutuante padrão com dupla precisão; compatível com o double de C e o objeto float de Python
float128	f16 ou g	Ponto flutuante com precisão estendida
complex64, complex128, complex256	c8, c16, c32	Números complexos representados por dois floats de 32, 64 ou 128, respectivamente
bool	?	Tipo booleano que armazena os valores True e False
object	O	Tipo objeto de Python; um valor pode ser qualquer objeto Python
string_	S	Tipo string ASCII de tamanho fixo (1 byte por caractere); por exemplo, para criar um dtype string com tamanho 10, utilize 'S10'
unicode_	U	Tipo Unicode de tamanho fixo (número de bytes é específico de cada plataforma); a mesma semântica de especificação de string_ (por exemplo, 'U10')

Você pode converter explicitamente ou fazer *cast* de um array de um dtype para outro usando o método astype de ndarray:

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr.dtype
Out[40]: dtype('float64')
```

Nesse exemplo, os inteiros receberam cast para ponto flutuante. Se eu fizer cast de alguns números de ponto flutuante para que tenham um dtype do tipo inteiro, a parte decimal será truncada:

```
In [41]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [42]: arr
```

```
Out[42]: array([ 3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [43]: arr.astype(np.int32)
Out[43]: array([ 3, -1, -2, 0, 12, 10], dtype=int32)
```

Se você tiver um array de strings que representem números, poderá usar astype para convertê-lo em um formato numérico:

```
In [44]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

In [45]: numeric_strings.astype(float)
Out[45]: array([ 1.25, -9.6 , 42. ])
```

É importante ser cauteloso ao usar o tipo numpy.string_, pois dados de string no Numpy têm tamanho fixo e podem truncar os dados de entrada sem avisos. O pandas tem um comportamento próprio mais intuitivo para dados não numéricos.

Se, por algum motivo, o casting falhar (como no caso de uma string que não possa ser convertida para float64), um ValueError será gerado. Fui um pouco preguiçoso nesse caso, e escrevi float em vez de np.float64; o NumPy tem aliases para os tipos Python, para os seus próprios dtypes equivalentes.

O atributo dtype de outro array também pode ser utilizado:

```
In [46]: int_array = np.arange(10)

In [47]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [48]: int_array.astype(calibers.dtype)
Out[48]: array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

Há strings de atalho para os códigos de tipo, e que podem ser usadas para se referir a um dtype:

```
In [49]: empty_uint32 = np.empty(8, dtype='u4')

In [50]: empty_uint32
Out[50]:
array([ 0, 1075314688, 0, 1075707904, 0,
       1075838976, 0, 1072693248], dtype=uint32)
```

 Chamar astype *sempre* cria um novo array (uma cópia dos dados), mesmo que o novo dtype seja igual ao dtype antigo.

Aritmética com arrays NumPy

Os arrays são importantes porque permitem expressar operações em lote nos dados, sem escrever qualquer laço for. Os usuários do NumPy chamam isso de *vetorização* (vectorization). Qualquer operação aritmética entre arrays de mesmo tamanho faz a operação ser aplicada em todos os elementos:

```
In [51]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [52]: arr
Out[52]:
array([[ 1., 2., 3.],
       [ 4., 5., 6.]])

In [53]: arr * arr
Out[53]:
array([[ 1., 4., 9.],
       [ 16., 25., 36.]])

In [54]: arr - arr
Out[54]:
array([[ 0., 0., 0.],
       [ 0., 0., 0.]])
```

As operações aritméticas com escalares fazem o argumento escalar ser propagado a cada elemento do array:

```
In [55]: 1 / arr
Out[55]:
array([[ 1. , 0.5 , 0.3333],
       [ 0.25 , 0.2 , 0.1667]])

In [56]: arr ** 0.5
Out[56]:
array([[ 1. , 1.4142, 1.7321],
       [ 2. , 2.2361, 2.4495]])
```

As comparações entre arrays de mesmo tamanho resultam em arrays booleanos:

```
In [57]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [58]: arr2
Out[58]:
array([[ 0., 4., 1.],
       [ 7., 2., 12.]])

In [59]: arr2 > arr
Out[59]:
array([[False, True, False],
       [ True, False, True]], dtype=bool)
```

As operações entre arrays de tamanhos distintos são chamadas de *broadcasting* e serão discutidas com mais detalhes no Apêndice A. Ter um conhecimento profundo de broadcasting não é necessário na maior parte deste livro.

Indexação básica e fatiamento

A indexação de arrays NumPy é um assunto rico, pois há muitos modos com os quais você pode querer selecionar um subconjunto de seus dados ou elementos individuais. Arrays unidimensionais são simples; superficialmente, eles se comportam de modo semelhante às listas de Python:

```
In [60]: arr = np.arange(10)

In [61]: arr
Out[61]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [62]: arr[5]
Out[62]: 5

In [63]: arr[5:8]
Out[63]: array([5, 6, 7])

In [64]: arr[5:8] = 12
```

```
In [65]: arr
Out[65]: array([ 0, 1, 2, 3, 4, 12, 12, 12, 8, 9])
```

Como podemos ver, se você atribuir um valor escalar a uma fatia (slice), como em `arr[5:8] = 12`, o valor será propagado (sofrerá *broadcast* a partir daí) para toda a seleção. Uma primeira distinção importante em relação às listas embutidas de Python é que as fatias de arrays são *visualizações* (views) do array original. Isso significa que os dados não são copiados, e qualquer modificação na visualização se refletirá no array original.

Para dar um exemplo disso, criei inicialmente uma fatia de `arr`:

```
In [66]: arr_slice = arr[5:8]

In [67]: arr_slice
Out[67]: array([12, 12, 12])
```

Agora, quando altero valores em `arr_slice`, as mudanças se refletem no array `arr` original:

```
In [68]: arr_slice[1] = 12345

In [69]: arr
Out[69]: array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])
```

A fatia “nua” `[:]` fará uma atribuição a todos os valores em um array:

```
In [70]: arr_slice[:] = 64

In [71]: arr
Out[71]: array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])
```

Se o NumPy for novidade para você, talvez se surpreenda com isso, especialmente se já utilizou outras linguagens de programação de arrays que copiam dados mais avidamente. Como o NumPy foi projetado para ser capaz de trabalhar com arrays bem grandes, você poderia pensar em problemas de desempenho e de memória se o NumPy insistisse em sempre copiar os dados.

Se você quiser uma cópia de uma fatia de um ndarray em vez de ter uma visualização, será necessário copiar explicitamente o array – por exemplo,

arr[5:8].copy().

Com arrays de dimensões maiores, você tem muito mais opções. Em um array bidimensional, os elementos em cada índice não são mais escalares, mas são arrays unidimensionais:

```
In [72]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [73]: arr2d[2]
Out[73]: array([7, 8, 9])
```

Desse modo, os elementos individuais podem ser acessados recursivamente. Entretanto, isso exige um pouco de trabalho, portanto podemos passar uma lista de índices separados por vírgula para selecionar elementos individuais. Assim, as instruções a seguir são equivalentes:

```
In [74]: arr2d[0][2]
Out[74]: 3

In [75]: arr2d[0, 2]
Out[75]: 3
```

Veja a Figura 4.1, que apresenta uma ilustração da indexação em um array bidimensional. Pensar no eixo 0 como as “linhas” do array e no eixo 1 como as “colunas” pode ajudar.

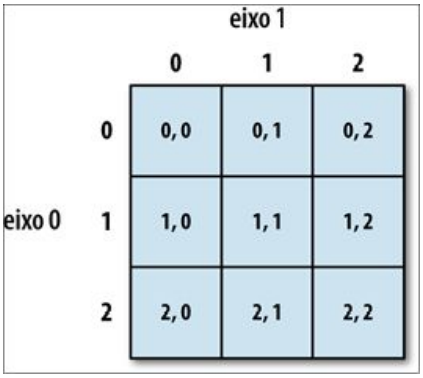


Figura 4.1 – Indexando elementos em um array NumPy.

Em arrays multidimensionais, se você omitir os índices finais, o objeto devolvido será um ndarray de dimensões menores, constituído de todos os dados nas dimensões mais altas. Assim, no array arr3d 2 × 2 × 3:

```
In [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [77]: arr3d
Out[77]:
array([[[ 1, 2, 3],
        [ 4, 5, 6]],
       [[ 7, 8, 9],
        [10, 11, 12]]])
```

`arr3d[0]` é um array 2 × 3:

```
In [78]: arr3d[0]
Out[78]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Tanto valores escalares quanto arrays podem ser atribuídos a `arr3d[0]`:

```
In [79]: old_values = arr3d[0].copy()

In [80]: arr3d[0] = 42

In [81]: arr3d
Out[81]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7, 8, 9],
        [10, 11, 12]]])

In [82]: arr3d[0] = old_values

In [83]: arr3d
Out[83]:
array([[[ 1, 2, 3],
        [ 4, 5, 6]],
       [[ 7, 8, 9],
        [10, 11, 12]]])
```

De modo semelhante, `arr3d[1, 0]` contém todos os valores cujos índices começam com `(1, 0)`, formando um array unidimensional:

```
In [84]: arr3d[1, 0]
Out[84]: array([7, 8, 9])
```

Essa expressão é equivalente a uma indexação em dois passos:

```
In [85]: x = arr3d[1]

In [86]: x
Out[86]:
array([[ 7, 8, 9],
       [10, 11, 12]])

In [87]: x[0]
Out[87]: array([7, 8, 9])
```

Observe que, em todos esses casos em que subseções do array foram selecionadas, os arrays devolvidos são visualizações.

Indexando com fatias

Assim como os objetos unidimensionais, por exemplo, as listas Python, os ndarrays podem ser fatiados com a sintaxe conhecida:

```
In [88]: arr
Out[88]: array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])

In [89]: arr[1:6]
Out[89]: array([ 1, 2, 3, 4, 64])
```

Considere o array bidimensional anterior, `arr2d`. Fatiar esse array é um pouco diferente:

```
In [90]: arr2d
Out[90]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [91]: arr2d[:2]
Out[91]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Como podemos ver, ele foi fatiado ao longo do eixo 0, que é o primeiro eixo. Uma fatia, portanto, seleciona um intervalo de elementos ao longo de um eixo. Ler a expressão `arr2d[:2]` como

“selecione as duas primeiras linhas de arr2d” talvez possa ajudar.

Podemos passar várias fatias, do mesmo modo como podemos passar vários índices:

```
In [92]: arr2d[:2, 1:]
Out[92]:
array([[2, 3],
       [5, 6]])
```

Ao fatiar dessa maneira, sempre vamos obter visualizações do array com o mesmo número de dimensões. Ao combinar índices inteiros com fatias, teremos fatias de dimensões menores.

Por exemplo, posso selecionar a segunda linha, mas somente as duas primeiras colunas, assim:

```
In [93]: arr2d[1, :2]
Out[93]: array([4, 5])
```

De modo semelhante, podemos selecionar a terceira coluna, mas somente as duas primeiras linhas, da seguinte maneira:

```
In [94]: arr2d[:2, 2]
Out[94]: array([3, 6])
```

Veja a Figura 4.2 que apresenta uma ilustração. Observe que os dois-pontos sozinhos significam que todo o eixo deve ser considerado, portanto podemos fatiar somente os eixos de dimensões mais altas fazendo o seguinte:

```
In [95]: arr2d[:, :1]
Out[95]:
array([[1],
       [4],
       [7]])
```

É claro que fazer uma atribuição a uma expressão de fatia faz a atribuição para toda a seleção:

```
In [96]: arr2d[:2, 1:] = 0

In [97]: arr2d
Out[97]:
array([[1, 0, 0],
```

```
[4, 0, 0],
[7, 8, 9]])
```

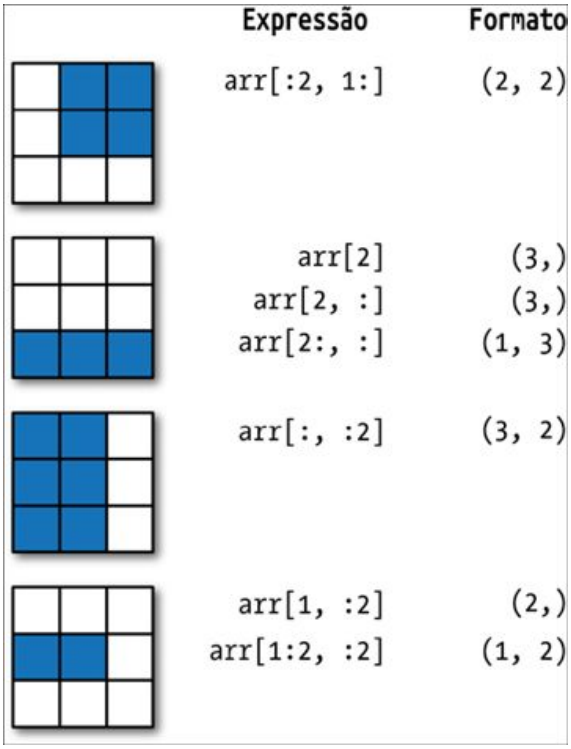


Figura 4.2 – Fatiamento de arrays bidimensionais.

Indexação booleana

Vamos considerar um exemplo em que temos alguns dados em um array e um array de nomes com duplicatas. Usarei aqui a função randn de numpy.random para gerar alguns dados aleatórios normalmente distribuídos:

```
In [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [99]: data = np.random.randn(7, 4)

In [100]: names
Out[100]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='<U4')

In [101]: data
Out[101]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
```

```
       [ 1.0072, -1.2962, 0.275 , 0.2289],
       [ 1.3529, 0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397, 0.477 ],
       [ 3.2489, -1.0212, -0.5771, 0.1241],
       [ 0.3026, 0.5238, 0.0009, 1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Suponha que cada nome corresponda a uma linha do array data e que queremos selecionar todas as linhas com o nome correspondente 'Bob'. Assim como nas operações aritméticas, as comparações (como ==) com arrays também são vetorizadas. Assim, comparar names com a string 'Bob' produzirá um array booleano:

```
In [102]: names == 'Bob'
Out[102]: array([ True, False, False, True, False, False, False], dtype=bool)
```

Esse array booleano pode ser passado quando o array é indexado:

```
In [103]: data[names == 'Bob']
Out[103]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
       [ 1.669 , -0.4386, -0.5397, 0.477 ]])
```

O array booleano deve ter o mesmo tamanho do eixo do array que ele está indexando. Você pode até mesmo misturar e fazer a correspondência entre arrays booleanos e fatias ou inteiros (ou sequências de inteiros; discutiremos melhor esse assunto depois).


A seleção booleana não falhará se o array booleano não tiver o tamanho correto, portanto recomendo tomar cuidado quando utilizar esse recurso.

Nesses exemplos, faço a seleção a partir das linhas em que names == 'Bob' e indexo as colunas também:

```
In [104]: data[names == 'Bob', 2:]
Out[104]:
array([[ 0.769 , 1.2464],
       [-0.5397, 0.477 ]])

In [105]: data[names == 'Bob', 3]
Out[105]: array([ 1.2464, 0.477 ])
```

Para selecionar tudo exceto 'Bob', podemos usar != ou negar a condição usando ~:

```
In [106]: names != 'Bob'
Out[106]: array([False, True, True, False, True, True, True], dtype=bool)

In [107]: data[~(names == 'Bob')]
Out[107]:
array([[ 1.0072, -1.2962, 0.275 , 0.2289],
       [ 1.3529, 0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771, 0.1241],
       [ 0.3026, 0.5238, 0.0009, 1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

O operador ~ pode ser útil se quisermos inverter uma condição genérica:

```
In [108]: cond = names == 'Bob'

In [109]: data[~cond]
Out[109]:
array([[ 1.0072, -1.2962, 0.275 , 0.2289],
       [ 1.3529, 0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771, 0.1241],
       [ 0.3026, 0.5238, 0.0009, 1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Selecionando dois dos três nomes para combinar várias condições booleanas, utilize operadores aritméticos booleanos como & (and) e | (or):

```
In [110]: mask = (names == 'Bob') | (names == 'Will')

In [111]: mask
Out[111]: array([ True, False, True, True, True, False, False], dtype=bool)

In [112]: data[mask]
Out[112]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
       [ 1.3529, 0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397, 0.477 ],
       [ 3.2489, -1.0212, -0.5771, 0.1241]])
```

Selecionar dados de um array com indexação booleana *sempre* criará uma cópia dos dados, mesmo que o array devolvido não seja alterado.


As palavras reservadas and e or de Python não funcionam com arrays booleanos. Utilize & (and) e | (or) em seu lugar.

Definir valores com arrays booleanos funciona de acordo com o senso comum. Para definir todos os valores negativos em data com 0, basta fazer o seguinte:

```
In [113]: data[data < 0] = 0

In [114]: data
Out[114]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
       [ 1.0072, 0. , 0.275 , 0.2289],
       [ 1.3529, 0.8864, 0. , 0. ],
       [ 1.669 , 0. , 0. , 0.477 ],
       [ 3.2489, 0. , 0. , 0.1241],
       [ 0.3026, 0.5238, 0.0009, 1.3438],
       [ 0. , 0. , 0. , 0. ]])
```

Definir linhas ou colunas inteiras usando um array booleano unidimensional também é fácil:

```
In [115]: data[names != 'Joe'] = 7

In [116]: data
Out[116]:
array([[ 7. , 7. , 7. , 7. ],
       [ 1.0072, 0. , 0.275 , 0.2289],
       [ 7. , 7. , 7. , 7. ],
       [ 7. , 7. , 7. , 7. ],
       [ 7. , 7. , 7. , 7. ],
       [ 0.3026, 0.5238, 0.0009, 1.3438],
       [ 0. , 0. , 0. , 0. ]])
```

Como veremos mais adiante, esses tipos de operações em dados bidimensionais são convenientes para serem executados com o pandas.

Indexação sofisticada

Indexação sofisticada (fancy indexing) é o termo adotado pelo NumPy para descrever a indexação usando arrays de inteiros. Suponha que tivéssemos um array 8 × 4:

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
   .....: arr[i] = i

In [119]: arr
Out[119]:
array([[ 0., 0., 0., 0.],
       [ 1., 1., 1., 1.],
       [ 2., 2., 2., 2.],
       [ 3., 3., 3., 3.],
       [ 4., 4., 4., 4.],
       [ 5., 5., 5., 5.],
       [ 6., 6., 6., 6.],
       [ 7., 7., 7., 7.]])
```

Para selecionar um subconjunto das linhas em uma ordem em particular, podemos simplesmente passar uma lista ou um ndarray de inteiros especificando a ordem desejada:

```
In [120]: arr[[4, 3, 0, 6]]
Out[120]:
array([[ 4., 4., 4., 4.],
       [ 3., 3., 3., 3.],
       [ 0., 0., 0., 0.],
       [ 6., 6., 6., 6.]])
```

Espero que esse código tenha feito o que você esperava! Usar índices negativos seleciona as linhas a partir do final:

```
In [121]: arr[[-3, -5, -7]]
Out[121]:
array([[ 5., 5., 5., 5.],
       [ 3., 3., 3., 3.],
       [ 1., 1., 1., 1.]])
```

Passar vários índices de array faz algo um pouco diferente; a

instrução seleciona um array unidimensional de elementos correspondentes a cada tupla de índices:

```
In [122]: arr = np.arange(32).reshape((8, 4))

In [123]: arr
Out[123]:
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])

In [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[124]: array([ 4, 23, 29, 10])
```

Veremos o método `reshape` com mais detalhes no Apêndice A.

Nesse caso, os elementos `(1, 0)`, `(5, 3)`, `(7, 1)` e `(2, 2)` foram selecionados. Independentemente de quantas dimensões o array tiver (no exemplo, apenas 2), o resultado da indexação sofisticada sempre será unidimensional.

O comportamento da indexação sofisticada nesse caso é um pouco diferente do que alguns usuários poderiam esperar (me incluo aqui), que é a região retangular formada pela seleção de um subconjunto das linhas e colunas da matriz. Eis uma forma de obter isso:

```
In [125]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[125]:
array([[ 4, 7, 5, 6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11, 9, 10]])
```

Tenha em mente que a indexação sofisticada, de modo diferente do fatiamento, sempre copia os dados para um novo array.

Transposição de arrays e troca de eixos

A transposição é uma forma especial de reformatação que, de modo semelhante, devolve uma visualização dos dados subjacentes, sem copiar nada. Os arrays têm o método transpose, além do atributo especial T:

```
In [126]: arr = np.arange(15).reshape((3, 5))

In [127]: arr
Out[127]:
array([[ 0, 1, 2, 3, 4],
       [ 5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])

In [128]: arr.T
Out[128]:
array([[ 0, 5, 10],
       [ 1, 6, 11],
       [ 2, 7, 12],
       [ 3, 8, 13],
       [ 4, 9, 14]])
```

Ao fazer processamentos de matrizes, talvez você realize isto com muita frequência – por exemplo, quando calcular o produto da matriz interna usando np.dot:

```
In [129]: arr = np.random.randn(6, 3)

In [130]: arr
Out[130]:
array([[-0.8608, 0.5601, -1.2659],
       [ 0.1198, -1.0635, 0.3329],
       [-2.3594, -0.1995, -1.542 ],
       [-0.9707, -1.307 , 0.2863],
       [ 0.378 , -0.7539, 0.3313],
       [ 1.3497, 0.0699, 0.2467]])

In [131]: np.dot(arr.T, arr)
Out[131]:
array([[ 9.2291, 0.9394, 4.948 ],
       [ 0.9394, 3.7662, -1.3622],
       [ 4.948 , -1.3622, 4.3437]])
```

Para arrays de dimensões maiores, transpose aceitará uma tupla de números de eixos para permutá-los (para dar mais um nó na cabeça):

```
In [132]: arr = np.arange(16).reshape((2, 2, 4))

In [133]: arr
Out[133]:
array([[[ 0, 1, 2, 3],
        [ 4, 5, 6, 7]],
       [[ 8, 9, 10, 11],
        [12, 13, 14, 15]]])

In [134]: arr.transpose((1, 0, 2))
Out[134]:
array([[[ 0, 1, 2, 3],
        [ 8, 9, 10, 11]],
       [[ 4, 5, 6, 7],
        [12, 13, 14, 15]]])
```

Nesse caso, os eixos foram reordenados com o segundo eixo em primeiro lugar, o primeiro eixo em segundo e o último eixo permaneceu inalterado.

Uma transposição simples com .T é um caso especial de troca de eixos. O ndarray tem o método swapaxes, que aceita um par de números de eixos e troca os eixos indicados para reorganizar os dados:

```
In [135]: arr
Out[135]:
array([[[ 0, 1, 2, 3],
        [ 4, 5, 6, 7]],
       [[ 8, 9, 10, 11],
        [12, 13, 14, 15]]])

In [136]: arr.swapaxes(1, 2)
Out[136]:
array([[[ 0, 4],
        [ 1, 5],
        [ 2, 6],
        [ 3, 7]],
```

```
[[ 8, 12],
 [ 9, 13],
 [10, 14],
 [11, 15]]])
```

De modo semelhante, swapaxes devolve uma visualização dos dados, sem criar uma cópia.

4.2 Funções universais: funções rápidas de arrays para todos os elementos

Uma função universal (universal function), ou *ufunc*, é uma função que executa operações em todos os elementos nos dados de ndarrays. Podemos pensar nelas como wrappers vetorizados rápidos para funções simples que aceitam um ou mais valores escalares e geram um ou mais resultados escalares.

Muitas ufuncs são transformações simples em todos os elementos, como sqrt ou exp:

```
In [137]: arr = np.arange(10)

In [138]: arr
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [139]: np.sqrt(arr)
Out[139]:
array([ 0. , 1. , 1.4142, 1.7321, 2. , 2.2361, 2.4495,
       2.6458, 2.8284, 3. ])

In [140]: np.exp(arr)
Out[140]:
array([ 1. , 2.7183, 7.3891, 20.0855, 54.5982,
        148.4132, 403.4288, 1096.6332, 2980.958 , 8103.0839])
```

Essas funções são chamadas de ufuncs *unárias*. Outras, como add ou maximum, aceitam dois arrays (portanto são ufuncs *binárias*) e devolvem um único array como resultado:

```
In [141]: x = np.random.randn(8)
```

```
In [142]: y = np.random.randn(8)

In [143]: x
Out[143]:
array([-0.0119, 1.0048, 1.3272, -0.9193, -1.5491, 0.0222, 0.7584, -0.6605])

In [144]: y
Out[144]:
array([ 0.8626, -0.01 , 0.05 , 0.6702, 0.853 , -0.9559, -0.0235, -2.3042])

In [145]: np.maximum(x, y)
Out[145]:
array([ 0.8626, 1.0048, 1.3272, 0.6702, 0.853 , 0.0222, 0.7584, -0.6605])
```

Nesse caso, numpy.maximum calculou o máximo entre os elementos de x e de y para todos os elementos.

Embora não seja comum, uma ufunc pode devolver vários arrays. modf é um exemplo; ela é uma versão vetorizada da função embutida divmod de Python e devolve as partes fracionária e inteira de um array de ponto flutuante:

```
In [146]: arr = np.random.randn(7) * 5

In [147]: arr
Out[147]: array([-3.2623, -6.0915, -6.663 , 5.3731, 3.6182, 3.45 , 5.0077])

In [148]: remainder, whole_part = np.modf(arr)

In [149]: remainder
Out[149]: array([-0.2623, -0.0915, -0.663 , 0.3731, 0.6182, 0.45 , 0.0077])

In [150]: whole_part
Out[150]: array([-3., -6., -6., 5., 3., 3., 5.])
```

As ufuncs aceitam um argumento opcional out que lhes permite atuar in-place nos arrays:

```
In [151]: arr
Out[151]: array([-3.2623, -6.0915, -6.663 , 5.3731, 3.6182, 3.45 , 5.0077])

In [152]: np.sqrt(arr)
Out[152]: array([ nan, nan, nan, 2.318 , 1.9022, 1.8574, 2.2378])
```

```
In [153]: np.sqrt(arr, arr)
Out[153]: array([ nan, nan, nan, 2.318 , 1.9022, 1.8574, 2.2378])

In [154]: arr
Out[154]: array([ nan, nan, nan, 2.318 , 1.9022, 1.8574, 2.2378])
```

Veja as Tabelas 4.3 e 4.4 que apresentam uma lista das ufuncs disponíveis.

Tabela 4.3 – Ufuncs unárias

Função	Descrição
abs, fabs	Calcula o valor absoluto de inteiros, números de ponto flutuante e valores complexos para todos os elementos
sqrt	Calcula a raiz quadrada de cada elemento (equivalente a arr ** 0.5)
square	Calcula o quadrado de cada elemento (equivalente a arr ** 2)
exp	Calcula o exponencial ex de cada elemento
log, log10, log2, log1p	Logaritmo natural (base *e*), log na base 10, log na base 2 e log (1 + x), respectivamente
sign	Calcula o sinal de cada elemento: 1 (positivo), 0 (zero) ou –1 (negativo)
ceil	Calcula o teto de cada elemento (isto é, o menor inteiro maior ou igual ao número)
floor	Calcula o piso de cada elemento (isto é, o maior inteiro menor ou igual ao elemento)
rint	Arredonda os elementos para o inteiro mais próximo, preservando o dtype
modf	Devolve as partes fracionária e inteira do array como um array separado
isnan	Devolve um array booleano indicando se cada valor é NaN (Not a Number)
isfinite, isinf	Devolve um array booleano indicando se cada elemento é finito (não inf, não NaN) ou infinito, respectivamente
cos, cosh, sin, sinh, tan, tanh	Funções trigonométricas regulares e hiperbólicas
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Funções trigonométricas inversas

Função	Descrição
logical_not	Calcula o valor-verdade de not x para todos os elementos (equivalente a ~arr).

Tabela 4.4 – Funções universais binárias

Função	Descrição
add	Soma elementos correspondentes em arrays
subtract	Subtrai elementos do segundo array do primeiro
multiply	Multiplica elementos do array
divide, floor_divide	Faz a divisão ou a divisão pelo piso (truncando o resto)
power	Eleva os elementos do primeiro array às potências indicadas no segundo array
maximum, fmax	Máximo para todos os elementos; fmax ignora NaN
minimum, fmin	Mínimo para todos os elementos; fmin ignora NaN
mod	Módulo para todos os elementos (resto da divisão)
copysign	Copia o sinal dos valores do segundo argumento para os valores do primeiro argumento
greater, greater_equal, less, less_equal, equal, not_equal	Faz uma comparação para todos os elementos, produzindo um array booleano (equivalente aos operadores infixos >, >=, <, <=, ==, !=)
logical_and, logical_or, logical_xor	Calcula o valor-verdade da operação lógica (equivalente aos operadores infixos &, |, ^) para todos os elementos

4.3 Programação orientada a arrays

Usar arrays NumPy permite expressar vários tipos de tarefas de processamento de dados na forma de expressões concisas de arrays que poderiam, do contrário, exigir a escrita de laços. Essa prática de substituir laços explícitos por expressões de arrays é comumente chamada de *vetorização* (vectorization). Em geral, operações vetorizadas em arrays com frequência serão mais rápidas em uma ou duas (ou mais) ordens de grandeza do que seus equivalentes em Python puro, com o maior impacto sendo em qualquer tipo de processamentos numéricos. Mais adiante, no Apêndice A, explicarei o que é o *broadcasting*: um método eficaz para processamentos com vetorização.

Como um exemplo simples, suponha que quiséssemos avaliar a função sqrt(x^2 + y^2) para uma grade regular de valores. A função np.meshgrid aceita dois arrays 1D e gera duas matrizes 2D correspondentes a todos os pares (x, y) nos dois arrays:

```
In [155]: points = np.arange(-5, 5, 0.01) # 1000 pontos igualmente espaçados

In [156]: xs, ys = np.meshgrid(points, points)

In [157]: ys
Out[157]:
array([[-5. , -5. , -5. , ..., -5. , -5. , -5. ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97, 4.97, 4.97, ..., 4.97, 4.97, 4.97],
       [ 4.98, 4.98, 4.98, ..., 4.98, 4.98, 4.98],
       [ 4.99, 4.99, 4.99, ..., 4.99, 4.99, 4.99]])
```

Avaliar a função é uma questão de escrever a mesma expressão que você escreveria com dois pontos:

```
In [158]: z = np.sqrt(xs ** 2 + ys ** 2)

In [159]: z
Out[159]:
array([[ 7.0711, 7.064 , 7.0569, ..., 7.0499, 7.0569, 7.064 ],
       [ 7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569],
       [ 7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       ...,
       [ 7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428],
       [ 7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       [ 7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569]])
```

Como uma prévia do Capítulo 9, utilizarei a matplotlib para criar visualizações desse array bidimensional:

```
In [160]: import matplotlib.pyplot as plt

In [161]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[161]: <matplotlib.colorbar.Colorbar at 0x7fa37d95c5f8>

In [162]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
```

```
Out[162]: Text(0.5,1,'Image plot of $\\sqrt{x^2 + y^2}$ for a grid of values')
```

Veja a Figura 4.3. Nesse caso, usei a função imshow da matplotlib para criar a imagem de uma plotagem de um array bidimensional dos valores de uma função.

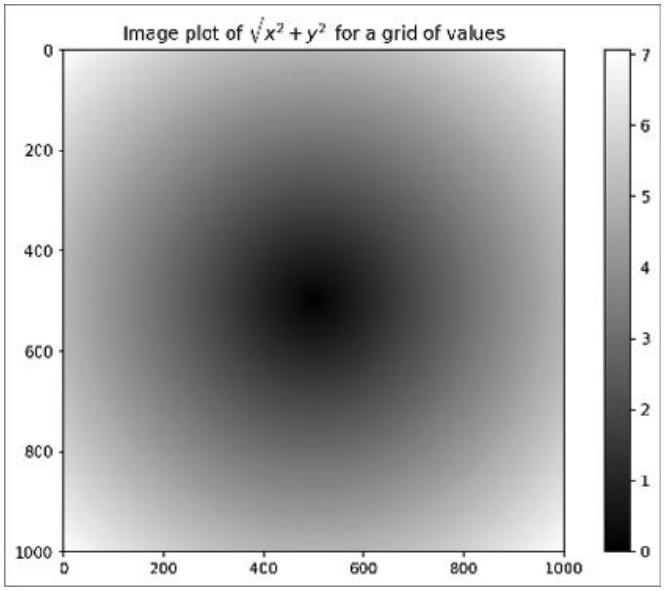


Figura 4.3 – Plotagem da função avaliada na grade.

Expressando uma lógica condicional como operações de array

A função numpy.where é uma versão vetorizada da expressão ternária x if condition else y. Suponha que tivéssemos um array booleano e dois arrays de valores:

```
In [165]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [166]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [167]: cond = np.array([True, False, True, True, False])
```

Suponha que queremos tomar um valor de xarr sempre que o valor correspondente em cond seja True e, caso contrário, tomar o valor de yarr. Uma list comprehension que faça isso pode ter o seguinte aspecto:

```
In [168]: result = [(x if c else y)
   .....: for x, y, c in zip(xarr, yarr, cond)]
```

```
In [169]: result
Out[169]: [1.1000000000000001, 2.2000000000000002, 1.3,
1.3999999999999999, 2.5]
```

Esse código apresenta vários problemas. Em primeiro lugar, ele não será muito rápido para arrays grandes (porque todo o trabalho está sendo feito em código Python interpretado). Em segundo, ele não funcionará para arrays multidimensionais. Com np.where, podemos escrever esse código de modo bem conciso:

```
In [170]: result = np.where(cond, xarr, yarr)

In [171]: result
Out[171]: array([ 1.1, 2.2, 1.3, 1.4, 2.5])
```

O segundo e o terceiro argumentos de np.where não precisam ser arrays; um deles, ou ambos, podem ser escalares. Um uso típico de where em análise de dados é aquele em que geramos um novo array de valores com base em outro array. Suponha que você tivesse uma matriz de dados gerados aleatoriamente e quisesse substituir todos os valores positivos por 2 e todos os valores negativos por –2. É muito fácil fazer isso usando np.where:

```
In [172]: arr = np.random.randn(4, 4)

In [173]: arr
Out[173]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 0.2229, 0.0513, -1.1577, 0.8167],
       [ 0.4336, 1.0107, 1.8249, -0.9975],
       [ 0.8506, -0.1316, 0.9124, 0.1882]])

In [174]: arr > 0
Out[174]:
array([[False, False, False, False],
       [ True, True, False, True],
       [ True, True, True, False],
       [ True, False, True, True]], dtype=bool)

In [175]: np.where(arr > 0, 2, -2)
```

```
Out[175]:
array([[-2, -2, -2, -2],
       [ 2, 2, -2, 2],
       [ 2, 2, 2, -2],
       [ 2, -2, 2, 2]])
```

Podemos combinar escalares e arrays quando usamos np.where. Por exemplo, posso substituir todos os valores positivos em arr pela constante 2, assim:

```
In [176]: np.where(arr > 0, 2, arr) # define somente os valores positivos com 2
Out[176]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 2. , 2. , -1.1577, 2. ],
       [ 2. , 2. , 2. , -0.9975],
       [ 2. , -0.1316, 2. , 2. ]])
```

Os arrays passados para np.where podem ser mais que apenas arrays de tamanhos iguais ou escalares.

Métodos matemáticos e estatísticos

Um conjunto de funções matemáticas que calcula estatísticas sobre um array inteiro ou sobre os dados ao longo de um eixo é acessível por meio de métodos da classe array. Você pode usar agregações (com frequência chamadas de *reduções*), como sum, mean e std (desvio-padrão), seja chamando o método da instância do array ou usando a função de nível superior do NumPy.

Neste exemplo, gerarei alguns dados aleatórios normalmente distribuídos e calcularei algumas estatísticas de agregação:

```
In [177]: arr = np.random.randn(5, 4)

In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149, 2.0037, 0.0296],
       [ 0.7953, 0.1181, -0.7485, 0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363, 1.0954],
       [ 0.9809, -0.5895, 1.5817, -0.5287]])
```

```
In [179]: arr.mean()
Out[179]: 0.19607051119998253

In [180]: np.mean(arr)
Out[180]: 0.19607051119998253

In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

Funções como `mean` e `sum` aceitam um argumento opcional `axis` que calcula a estatística no eixo dado, resultando em um array com uma dimensão a menos:

```
In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 , 0.1875, -0.502 , -0.0881, 0.3611])

In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345, 2.2381, 1.1486])
```

Nesse caso, `arr.mean(1)` significa “calcule a média pelas colunas”, enquanto `arr.sum(0)` significa “calcule a soma pelas linhas”.

Outros métodos como `cumsum` e `cumprod` não fazem agregações, gerando um array de resultados intermediários em seu lugar:

```
In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [185]: arr.cumsum()
Out[185]: array([ 0, 1, 3, 6, 10, 15, 21, 28])
```

Em arrays multidimensionais, funções de acumulação como `cumsum` devolvem um array de mesmo tamanho, porém com as agregações parciais calculadas ao longo do eixo indicado, de acordo com cada fatia de dimensão menor:

```
In [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [187]: arr
Out[187]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [188]: arr.cumsum(axis=0)
```

```
Out[188]:
array([[ 0, 1, 2],
       [ 3, 5, 7],
       [ 9, 12, 15]])

In [189]: arr.cumprod(axis=1)
Out[189]:
array([[ 0, 0, 0],
       [ 3, 12, 60],
       [ 6, 42, 336]])
```

Veja a Tabela 4.5 que apresenta uma lista completa. Veremos muitos exemplos desses métodos em ação em capítulos mais adiante.

Tabela 4.5 – Métodos estatísticos básicos de arrays

Método	Descrição
sum	Soma de todos os elementos do array ou ao longo de um eixo; arrays de tamanho zero têm soma igual a 0
mean	Média aritmética; arrays de tamanho zero têm média NaN
std, var	Desvio-padrão e variância, respectivamente, com graus opcionais de ajuste de liberdade (denominador default n)
min, max	Mínimo e máximo
argmin, argmax	Índices dos elementos mínimo e máximo, respectivamente
cumsum	Soma cumulativa dos elementos, começando de 0
cumprod	Produto cumulativo dos elementos, começando de 1

Métodos para arrays booleanos

Valores booleanos sofrem coerção para 1 (True) e 0 (False) nos métodos anteriores. Assim, sum com frequência é usado como uma forma de contar valores True em um array booleano:

```
In [190]: arr = np.random.randn(100)

In [191]: (arr > 0).sum() # Número de valores positivos
Out[191]: 42
```

Há dois métodos adicionais, any e all, particularmente úteis para

arrays booleanos. any testa se um ou mais valores em um array são True, enquanto all verifica se todos os valores são True:

```
In [192]: bools = np.array([False, False, True, False])

In [193]: bools.any()
Out[193]: True

In [194]: bools.all()
Out[194]: False
```

Esses métodos também funcionam com arrays não booleanos, em que elementos diferentes de zero são avaliados como True.

Ordenação

Assim como o tipo embutido lista de Python, os arrays NumPy podem ser ordenados in-place com o método sort:

```
In [195]: arr = np.random.randn(6)

In [196]: arr
Out[196]: array([ 0.6095, -0.4938, 1.24 , -0.1357, 1.43 , -0.8469])

In [197]: arr.sort()

In [198]: arr
Out[198]: array([-0.8469, -0.4938, -0.1357, 0.6095, 1.24 , 1.43 ])
```

Podemos ordenar cada seção unidimensional de valores em um array multidimensional in-place ao longo de um eixo, passando o número desse eixo para sort:

```
In [199]: arr = np.random.randn(5, 3)

In [200]: arr
Out[200]:
array([[ 0.6033, 1.2636, -0.2555],
       [-0.4457, 0.4684, -0.9616],
       [-1.8245, 0.6254, 1.0229],
       [ 1.1074, 0.0909, -0.3501],
       [ 0.218 , -0.8948, -1.7415]])
```

```
In [201]: arr.sort(1)

In [202]: arr
Out[202]:
array([[-0.2555, 0.6033, 1.2636],
       [-0.9616, -0.4457, 0.4684],
       [-1.8245, 0.6254, 1.0229],
       [-0.3501, 0.0909, 1.1074],
       [-1.7415, -0.8948, 0.218 ]])
```

O método de nível superior np.sort devolve uma cópia ordenada de um array, em vez de modificá-lo in-place. Uma forma rápida e fácil de calcular os quantis de um array é ordená-lo e selecionar o valor em uma posição em particular:

```
In [203]: large_arr = np.random.randn(1000)

In [204]: large_arr.sort()

In [205]: large_arr[int(0.05 * len(large_arr))] # quantil de 5%
Out[205]: -1.5311513550102103
```

Para ver mais detalhes sobre o uso dos métodos de ordenação do NumPy, além de técnicas mais avançadas como ordenações indiretas, consulte o Apêndice A. Vários outros tipos de manipulações de dados relacionados à ordenação (por exemplo, ordenar uma tabela de dados de acordo com uma ou mais colunas) também podem ser encontrados no pandas.

Unicidade e outras lógicas de conjuntos

O NumPy tem algumas operações básicas de conjunto para ndarrays unidimensionais. Uma operação comumente utilizada é np.unique, que devolve os valores únicos ordenados de um array:

```
In [206]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [207]: np.unique(names)
Out[207]:
array(['Bob', 'Joe', 'Will'], dtype='<U4')
```

```
In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [209]: np.unique(ints)
Out[209]: array([1, 2, 3, 4])
```

Compare `np.unique` com a alternativa em Python puro:

```
In [210]: sorted(set(names))
Out[210]: ['Bob', 'Joe', 'Will']
```

Outra função, `np.in1d`, testa a pertinência dos valores de um array em outro, devolvendo um array booleano:

```
In [211]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [212]: np.in1d(values, [2, 3, 6])
Out[212]: array([ True, False, False, True, True, False, True], dtype=bool)
```

Veja a Tabela 4.6 que contém uma lista das funções de conjunto do NumPy.

Tabela 4.6 – Operações de conjunto em arrays

Método	Descrição
unique(x)	Calcula os elementos únicos ordenados de x
intersect1d(x, y)	Calcula os elementos comuns ordenados em x e y
union1d(x, y)	Calcula a união ordenada dos elementos
in1d(x, y)	Calcula um array booleano indicando se cada elemento de x está contido em y
setdiff1d(x, y)	Diferença entre conjuntos, isto é, elementos em x que não estão em y
setxor1d(x, y)	Diferença simétrica entre conjuntos: elementos que estão em apenas um dos arrays, mas não em ambos

4.4 Entrada e saída de arquivos com arrays

O NumPy é capaz de salvar e carregar dados de e para o disco, seja em formato-texto ou em formato binário. Nesta seção, discutirei somente o formato binário embutido do NumPy, pois a maioria dos usuários preferirá o pandas e outras ferramentas para carregar texto ou dados tabulares (veja o Capítulo 6 para obter muito mais

informações).

np.save e np.load são as duas funções que representam a força de trabalho para salvar e carregar dados de array em disco, de modo eficiente. Os arrays são salvos por padrão em um formato binário puro, não compactado, com a extensão de arquivo *.npy*:

```
In [213]: arr = np.arange(10)

In [214]: np.save('some_array', arr)
```

Se o path do arquivo ainda não terminar com *.npy*, a extensão será concatenada. O array em disco poderá então ser carregado com np.load:

```
In [215]: np.load('some_array.npy')
Out[215]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Você pode salvar vários arrays em um arquivo não compactado usando np.savez, passando os arrays como argumentos nomeados:

```
In [216]: np.savez('array_archive.npz', a=arr, b=arr)
```

Ao carregar um arquivo *.npz*, você obterá um objeto do tipo dicionário que carregará os arrays individuais em modo lazy (preguiçoso):

```
In [217]: arch = np.load('array_archive.npz')

In [218]: arch['b']
Out[218]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Se seus dados tiverem um bom nível de compactação, você pode usar numpy.savez_compressed:

```
In [219]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

4.5 Álgebra linear

A álgebra linear, assim como a multiplicação de matrizes, as decomposições, os determinantes e outras operações matemáticas em matrizes quadradas, é uma parte importante de qualquer biblioteca de arrays. De modo diferente de algumas linguagens como o MATLAB, multiplicar dois arrays bidimensionais com *

corresponde ao produto de todos os elementos, e não a um produto escalar de matrizes. Desse modo, há uma função dot, que é tanto um método de array quanto uma função no namespace numpy, para multiplicação de matrizes:

```
In [223]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [224]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [225]: x
Out[225]:
array([[ 1., 2., 3.],
       [ 4., 5., 6.]])

In [226]: y
Out[226]:
array([[ 6., 23.],
       [ -1., 7.],
       [ 8., 9.]])

In [227]: x.dot(y)
Out[227]:
array([[ 28., 64.],
       [ 67., 181.]])
```

x.dot(y) é equivalente a np.dot(x, y):

```
In [228]: np.dot(x, y)
Out[228]:
array([[ 28., 64.],
       [ 67., 181.]])
```

Um produto de matrizes entre um array bidimensional e um array unidimensional de tamanho apropriado resulta em um array unidimensional:

```
In [229]: np.dot(x, np.ones(3))
Out[229]: array([ 6., 15.])
```

O símbolo @ (conforme usado em Python 3.5) também funciona como um operador infixo que efetua uma multiplicação de matrizes:

```
In [230]: x @ np.ones(3)
Out[230]: array([ 6., 15.])
```

numpy.linalg tem um conjunto padrão de decomposições de matrizes, além de operações como inverso e determinante. Elas estão implementadas internamente por meio das mesmas bibliotecas de álgebra linear que são padrões de mercado em outras linguagens como MATLAB e R, por exemplo, BLAS, LAPACK ou, possivelmente (dependendo de sua versão de NumPy), o Intel MKL (Math Kernel Library) proprietário:

```
In [231]: from numpy.linalg import inv, qr

In [232]: X = np.random.randn(5, 5)

In [233]: mat = X.T.dot(X)

In [234]: inv(mat)
Out[234]:
array([[ 933.1189, 871.8258, -1417.6902, -1460.4005, 1782.1391],
       [ 871.8258, 815.3929, -1325.9965, -1365.9242, 1666.9347],
       [-1417.6902, -1325.9965, 2158.4424, 2222.0191, -2711.6822],
       [-1460.4005, -1365.9242, 2222.0191, 2289.0575, -2793.422 ],
       [ 1782.1391, 1666.9347, -2711.6822, -2793.422 , 3409.5128]])

In [235]: mat.dot(inv(mat))
Out[235]:
array([[ 1., 0., -0., -0., -0.],
       [-0., 1., 0., 0., 0.],
       [ 0., 0., 1., 0., 0.],
       [-0., 0., 0., 1., -0.],
       [-0., 0., 0., 0., 1.]])

In [236]: q, r = qr(mat)

In [237]: r
Out[237]:
array([[-1.6914, 4.38 , 0.1757, 0.4075, -0.7838],
       [ 0. , -2.6436, 0.1939, -3.072 , -1.0702],
       [ 0. , 0. , -0.8138, 1.5414, 0.6155],
       [ 0. , 0. , 0. , -2.6445, -2.1669],
       [ 0. , 0. , 0. , 0. , 0.0002]])
```

A expressão X.T.dot(X) calcula o produto escalar de X com sua

transposta X.T.

Veja a Tabela 4.7 que contém uma lista de algumas das funções mais comumente utilizadas de álgebra linear.

Tabela 4.7 – Funções comumente usadas de numpy.linalg

Função	Descrição
diag	Devolve os elementos diagonais (ou fora da diagonal) de uma matriz quadrada como um array 1D, ou converte um array 1D em uma matriz quadrada, com zeros fora da diagonal
dot	Multiplicação de matrizes
trace	Calcula a soma dos elementos da diagonal
det	Calcula o determinante da matriz
eig	Calcula os autovalores (valores próprios) e os autovetores de uma matriz quadrada
inv	Calcula a inversa de uma matriz quadrada
pinv	Calcula a pseudoinversa de Moore-Penrose de uma matriz
qr	Calcula a decomposição QR
svd	Calcula a SVD (Singular Value Decomposition, ou Decomposição de Valor Singular)
solve	Resolve o sistema linear Ax = b para x, em que A é uma matriz quadrada
lstsq	Calcula a solução de quadrados mínimos para Ax = b

4.6 Geração de números pseudoaleatórios

O módulo numpy.random suplementa o módulo embutido random de Python com funções para gerar arrays inteiros de valores de amostras, de modo eficaz, a partir de vários tipos de distribuições de probabilidade. Por exemplo, podemos obter um array 4 × 4 de amostras da distribuição normal padrão usando normal:

```
In [238]: samples = np.random.normal(size=(4, 4))

In [239]: samples
Out[239]:
array([[ 0.5732, 0.1933, 0.4429, 1.2796],
       [ 0.575 , 0.4339, -0.7658, -1.237 ],
```

```
       [-0.5367, 1.8545, -0.92 , -0.1082],
       [ 0.1525, 0.9435, -1.0953, -0.144 ]])
```

O módulo embutido random de Python, em oposição, mostra somente um valor de cada vez. Como podemos ver com base nesse benchmark, numpy.random é mais rápido, em muito mais que uma ordem de grandeza, para gerar amostras bem grandes:

```
In [240]: from random import normalvariate

In [241]: N = 1000000

In [242]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.54 s +- 81.9 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [243]: %timeit np.random.normal(size=N)
71.4 ms +- 8.53 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Chamamos esses números de *pseudoaleatórios* porque são gerados por um algoritmo com um comportamento determinístico, baseado na semente (seed) do gerador de números aleatórios. Podemos alterar a semente para a geração de números aleatórios no NumPy usando np.random.seed:

```
In [244]: np.random.seed(1234)
```

As funções de geração de dados em numpy.random utilizam uma semente aleatória global. A fim de evitar um estado global, podemos usar numpy.random.RandomState para criar um gerador de números aleatórios isolado de outros:

```
In [245]: rng = np.random.RandomState(1234)

In [246]: rng.randn(10)
Out[246]:
array([ 0.4714, -1.191 , 1.4327, -0.3127, -0.7206, 0.8872, 0.8596,
       -0.6365, 0.0157, -2.2427])
```

Veja a Tabela 4.8 que apresenta uma lista parcial das funções disponíveis em numpy.random. Darei alguns exemplos de como tirar proveito da capacidade dessas funções para gerar arrays grandes de amostras, tudo de uma só vez na próxima seção.

Tabela 4.8 – Lista parcial de funções de numpy.random

Função	Descrição
seed	Fornece uma semente ao gerador de números aleatórios
permutation	Devolve uma permutação aleatória de uma sequência ou um intervalo permutado
shuffle	Permuta aleatoriamente uma sequência in-place
rand	Sorteia amostras de uma distribuição uniforme
randint	Sorteia inteiros aleatórios de um dado intervalo de valores menores para maiores
randn	Sorteia amostras de uma distribuição normal com média 0 e desvio-padrão 1 (interface do tipo MATLAB)
binomial	Sorteia amostras de uma distribuição binomial
normal	Sorteia amostras de uma distribuição normal (gaussiana)
beta	Sorteia amostras de uma distribuição beta
chisquare	Sorteia amostras de uma distribuição qui-quadrada
gamma	Sorteia amostras de uma distribuição gama
uniform	Sorteia amostras de uma distribuição uniforme [0, 1)

4.7 Exemplo: passeios aleatórios

A simulação de passeios aleatórios, ou random walks (*https://en.wikipedia.org/wiki/Random_walk*), é uma aplicação ilustrativa de como utilizar operações de arrays. Inicialmente vamos considerar um passeio aleatório simples que comece em 0, com passos de 1 e –1 ocorrendo com a mesma probabilidade.

Eis uma maneira em Python puro de implementar um único passeio aleatório com mil passos usando o módulo embutido random:

```
In [247]: import random
    .....: position = 0
    .....: walk = [position]
    .....: steps = 1000
    .....: for i in range(steps):
    .....: step = 1 if random.randint(0, 1) else -1
    .....: position += step
    .....: walk.append(position)
    .....:
```

Veja a Figura 4.4 que apresenta uma plotagem de exemplo dos cem primeiros valores em um desses passeios aleatórios:

```
In [249]: plt.plot(walk[:100])
```

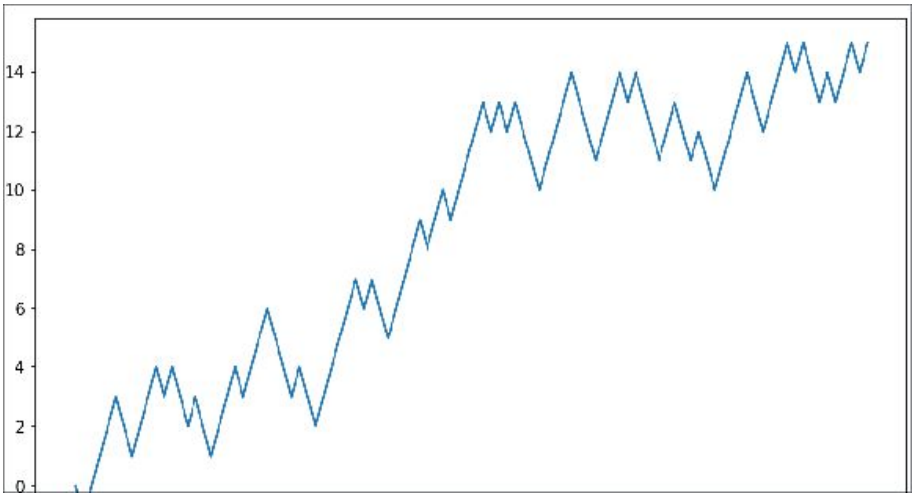


Figura 4.4 – Um passeio aleatório simples.

Você poderia perceber que walk é simplesmente a soma cumulativa dos passos aleatórios, e poderia ser avaliada como uma expressão de array. Assim, utilizarei o módulo np.random para sortear mil jogadas de moeda de uma só vez, definirei essas jogadas com 1 e –1 e calcularei a soma cumulativa:

```
In [251]: nsteps = 1000

In [252]: draws = np.random.randint(0, 2, size=nsteps)

In [253]: steps = np.where(draws > 0, 1, -1)

In [254]: walk = steps.cumsum()
```

A partir daí, podemos começar a extrair dados estatísticos, como o valor mínimo e o valor máximo na trajetória do passeio:

```
In [255]: walk.min()
Out[255]: -3

In [256]: walk.max()
Out[256]: 31
```

Uma estatística mais complicada é o *instante da primeira cruzada* (first

crossing time) – o passo em que o passeio aleatório alcança um determinado valor. Nesse caso, podemos querer saber quanto tempo demorou para que o passeio aleatório chegasse pelo menos a dez passos de distância da origem 0 em qualquer direção. `np.abs(walk) >= 10` nos dá um array booleano indicando em que ponto o passeio alcançou ou excedeu 10, mas queremos o índice do *primeiro* 10 ou –10. O fato é que podemos calcular isso usando `argmax`, que devolve o primeiro índice do valor máximo no array booleano (`True` é o valor máximo):

```
In [257]: (np.abs(walk) >= 10).argmax()
Out[257]: 37
```

Observe que usar `argmax` aqui nem sempre é eficaz, pois ele faz uma varredura completa do array. Nesse caso especial, quando um `True` é observado, sabemos que ele é o valor máximo.

Simulando vários passeios aleatórios de uma só vez

Se a sua meta é simular vários passeios aleatórios, por exemplo, 5 mil deles, pode gerar todos os passeios aleatórios com pequenas modificações no código anterior. Se receberem uma tupla de 2, as funções de `numpy.random` gerarão um array bidimensional de sorteios, e podemos calcular a soma cumulativa nas linhas a fim de calcular todos os 5 mil passeios aleatórios de uma só vez:

```
In [258]: nwalks = 5000

In [259]: nsteps = 1000

In [260]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1

In [261]: steps = np.where(draws > 0, 1, -1)

In [262]: walks = steps.cumsum(1)

In [263]: walks
Out[263]:
```

```
array([[ 1, 0, 1, ..., 8, 7, 8],
       [ 1, 0, -1, ..., 34, 33, 32],
       [ 1, 0, -1, ..., 4, 5, 4],
       ...,
       [ 1, 2, 1, ..., 24, 25, 26],
       [ 1, 2, 3, ..., 14, 13, 14],
       [ -1, -2, -3, ..., -24, -23, -22]])
```

Agora podemos calcular os valores máximo e mínimo obtidos em todos os passeios:

```
In [264]: walks.max()
Out[264]: 138

In [265]: walks.min()
Out[265]: -133
```

Entre esses passeios, vamos calcular o tempo mínimo de cruzada para 30 ou –30. É um pouco complicado, pois nem todos os 5 mil alcançam 30. Podemos verificar isso usando o método any:

```
In [266]: hits30 = (np.abs(walks) >= 30).any(1)

In [267]: hits30
Out[267]: array([False, True, False, ..., False, True, False], dtype=bool)

In [268]: hits30.sum() # Número que atinge 30 ou -30
Out[268]: 3410
```

Podemos utilizar esse array booleano para selecionar as linhas de walks que realmente cruzaram o nível absoluto 30 e chamar argmax no eixo 1 para obter os instantes de cruzada:

```
In [269]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)

In [270]: crossing_times.mean()
Out[270]: 498.88973607038122
```

Sinta-se à vontade para fazer experimentos com outras distribuições para passos que não sejam jogadas de moeda de mesma probabilidade. Basta usar uma função geradora de números aleatórios distinta, como normal, para gerar passos normalmente distribuídos, com alguma média e algum desvio-padrão:

```
In [271]: steps = np.random.normal(loc=0, scale=0.25,
   .....: size=(nwalks, nsteps))
```

4.8 Conclusão

Embora boa parte do restante do livro mantenha o foco no desenvolvimento de habilidades para tratamento de dados com o pandas, continuaremos a trabalhar com um estilo semelhante, baseado em arrays. No Apêndice A, exploraremos mais profundamente os recursos do NumPy a fim de ajudar você a desenvolver melhor as habilidades de processamento com arrays.

CAPÍTULO 5

Introdução ao pandas

O pandas será uma ferramenta principal de interesse em boa parte do restante do livro. Ele contém estruturas de dados e ferramentas para manipulação de dados, projetadas para agilizar e facilitar a limpeza e a análise de dados em Python. Com frequência, o pandas é usado em conjunto com ferramentas de processamento numérico como NumPy e SciPy, bibliotecas de análise como statsmodels e scikit-learn e bibliotecas de visualização de dados como a matplotlib. O pandas adota partes significativas do estilo idiomático do NumPy para processamento baseado em arrays, especialmente funções baseadas em arrays e uma preferência por processamento de dados sem laços for.

Embora o pandas adote muitos idioms de programação do NumPy, a principal diferença é que o pandas foi projetado para trabalhar com dados tabulares e heterogêneos. O NumPy, em comparação, é mais apropriado para trabalhar com dados numéricos homogêneos em arrays.

Desde que se tornou um projeto de código aberto em 2010, o pandas amadureceu e passou a ser uma biblioteca bem grande, aplicável a um amplo conjunto de casos de uso do mundo real. A comunidade de desenvolvedores cresceu, atingindo mais de 800 colaboradores distintos, que têm ajudado a construir o projeto à medida que o têm usado para resolver seus problemas de dados no cotidiano.

No restante do livro, usarei a seguinte convenção de importação para o pandas:

```
In [1]: import pandas as pd
```

Assim, sempre que você vir pd. no código, essa será uma referência ao pandas. Talvez também ache mais fácil importar Series e DataFrame para o namespace local, pois eles são utilizados com muita frequência:

```
In [2]: from pandas import Series, DataFrame
```

5.1 Introdução às estruturas de dados do pandas

Para começar a trabalhar com o pandas, você precisará se sentir à vontade com as duas estruturas de dados que são a sua força de trabalho: *Series* e *DataFrame.* Embora não sejam uma solução universal para todos os problemas, eles oferecem uma base sólida e fácil de usar para a maioria das aplicações.

Series

Uma Series é um objeto do tipo array unidimensional contendo uma sequência de valores (de tipos semelhantes aos tipos do NumPy) e um array associado de rótulos (labels) de dados, chamado de *índice*. A Series mais simples é composta de apenas um array de dados:

```
In [11]: obj = pd.Series([4, 7, -5, 3])

In [12]: obj
Out[12]:
0 4
1 7
2 -5
3 3
dtype: int64
```

A representação em string de uma Series exibida interativamente mostra o índice à esquerda e os valores à direita. Como não especificamos um índice para os dados, um índice default constituído dos inteiros de 0 a N - 1 (em que N é o tamanho dos dados) é criado. Podemos obter a representação do array e o objeto de índice de Series por meio de seus atributos de valores (values) e de índice (index), respectivamente:

```
In [13]: obj.values
Out[13]: array([ 4, 7, -5, 3])

In [14]: obj.index # como range(4)
Out[14]: RangeIndex(start=0, stop=4, step=1)
```

Com frequência, será desejável criar uma Series com um índice que identifique cada ponto de dado com um rótulo:

```
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [16]: obj2
Out[16]:
d 4
b 7
a -5
c 3
dtype: int64

In [17]: obj2.index
Out[17]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Em comparação com os arrays NumPy, podemos usar rótulos no índice quando selecionamos valores únicos ou um conjunto de valores:

```
In [18]: obj2['a']
Out[18]: -5

In [19]: obj2['d'] = 6

In [20]: obj2[['c', 'a', 'd']]
Out[20]:
c 3
a -5
d 6
dtype: int64
```

Nesse caso, `['c', 'a', 'd']` é interpretado como uma lista de índices, apesar de conter strings em vez de inteiros.

Usar funções NumPy ou operações do tipo NumPy, como filtragem com um array booleano, multiplicação escalar ou aplicação de

funções matemáticas, preservará a ligação entre índice e valor:

```
In [21]: obj2[obj2 > 0]
Out[21]:
d 6
b 7
c 3
dtype: int64

In [22]: obj2 * 2
Out[22]:
d 12
b 14
a -10
c 6
dtype: int64

In [23]: np.exp(obj2)
Out[23]:
d 403.428793
b 1096.633158
a 0.006738
c 20.085537
dtype: float64
```

Outra forma de pensar em uma Series é como um dicionário ordenado de tamanho fixo, como se fosse um mapeamento entre valores de índices e valores de dados. Ela pode ser utilizada em muitos contextos em que um dicionário poderia ser usado:

```
In [24]: 'b' in obj2
Out[24]: True

In [25]: 'e' in obj2
Out[25]: False
```

Se você tiver dados contidos em um dicionário Python, uma Series poderá ser criada a partir dele, passando-lhe o dicionário:

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [27]: obj3 = pd.Series(sdata)
```

```
In [28]: obj3
Out[28]:
Ohio 35000
Oregon 16000
Texas 71000
Utah 5000
dtype: int64
```

Se você passar somente um dicionário, o índice na Series resultante terá as chaves do dicionário ordenadas. É possível sobrescrever isso passando as chaves do dicionário na ordem que você quiser que elas apareçam na Series resultante:

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [30]: obj4 = pd.Series(sdata, index=states)

In [31]: obj4
Out[31]:
California NaN
Ohio 35000.0
Oregon 16000.0
Texas 71000.0
dtype: float64
```

Nesse caso, três valores encontrados em sdata foram colocados nos locais apropriados, mas, como nenhum valor para 'California' foi encontrado, ele aparece como NaN (not a number), que é considerado um marcador de valores ausentes ou *NA* no pandas. Como 'Utah' não estava incluído em states, ele foi excluído do objeto resultante.

Usarei os termos “ausente” ou “NA” indistintamente para me referir aos dados que estejam faltando. As funções isnull e notnull no pandas devem ser utilizadas para detectar dados ausentes:

```
In [32]: pd.isnull(obj4)
Out[32]:
California True
Ohio False
Oregon False
Texas False
```

```
dtype: bool

In [33]: pd.notnull(obj4)
Out[33]:
California False
Ohio True
Oregon True
Texas True
dtype: bool
```

Uma Series também tem esses métodos como métodos de instância:

```
In [34]: obj4.isnull()
Out[34]:
California True
Ohio False
Oregon False
Texas False
dtype: bool
```

Discutirei como trabalhar com dados ausentes de modo mais detalhado no Capítulo 7.

Um recurso útil de Series para muitas aplicações é que um alinhamento automático pelo rótulo do índice é feito nas operações aritméticas:

```
In [35]: obj3
Out[35]:
Ohio 35000
Oregon 16000
Texas 71000
Utah 5000
dtype: int64

In [36]: obj4
Out[36]:
California NaN
Ohio 35000.0
Oregon 16000.0
Texas 71000.0
dtype: float64
```

```
In [37]: obj3 + obj4
Out[37]:
California NaN
Ohio 70000.0
Oregon 32000.0
Texas 142000.0
Utah NaN
dtype: float64
```

Os recursos de alinhamento de dados serão tratados com mais detalhes posteriormente. Se você tem experiência com bancos de dados, poderá pensar nisso como semelhante a uma operação de junção (join).

Tanto o próprio objeto Series quanto seu índice têm um atributo `name`, que se integra com outras áreas essenciais de funcionalidades do pandas:

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'

In [40]: obj4
Out[40]:
state
California NaN
Ohio 35000.0
Oregon 16000.0
Texas 71000.0
Name: population, dtype: float64
```

Um índice de Series pode ser alterado in-place por atribuição:

```
In [41]: obj
Out[41]:
0 4
1 7
2 -5
3 3
dtype: int64

In [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

```
In [43]: obj
Out[43]:
Bob 4
Steve 7
Jeff -5
Ryan 3
dtype: int64
```

DataFrame

Um DataFrame representa uma tabela de dados retangular e contém uma coleção ordenada de colunas, em que cada uma pode ter um tipo de valor diferente (numérico, string, booleano etc.). O DataFrame tem índice tanto para linha quanto para coluna; pode ser imaginado como um dicionário de Series, todos compartilhando o mesmo índice. Internamente, os dados são armazenados como um ou mais blocos bidimensionais em vez de serem armazenados como uma lista, um dicionário ou outra coleção de arrays unidimensionais. Os detalhes exatos do funcionamento interno do DataFrame estão fora do escopo deste livro.

Embora um DataFrame seja fisicamente bidimensional, podemos usá-lo para representar dados de dimensões maiores em um formato tabular usando indexação hierárquica – um assunto que será discutido no Capítulo 8 e é um ingrediente de alguns dos recursos mais sofisticados de manipulação de dados do pandas.

Há várias formas de construir um DataFrame, embora uma das mais comuns seja a partir de um dicionário de listas de mesmo tamanho ou de arrays NumPy:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

O DataFrame resultante terá seu índice atribuído automaticamente como ocorre com Series, e as colunas estarão ordenadas:

```
In [45]: frame
Out[45]:
   pop state year
0 1.5 Ohio 2000
1 1.7 Ohio 2001
2 3.6 Ohio 2002
3 2.4 Nevada 2001
4 2.9 Nevada 2002
5 3.2 Nevada 2003
```

Se você estiver usando o notebook Jupyter, os objetos DataFrame do pandas serão exibidos como uma tabela HTML, mais apropriada a um navegador.

Para DataFrames grandes, o método `head` selecionará somente as cinco primeiras linhas:

```
In [46]: frame.head()
Out[46]:
   pop state year
0 1.5 Ohio 2000
1 1.7 Ohio 2001
2 3.6 Ohio 2002
3 2.4 Nevada 2001
4 2.9 Nevada 2002
```

Se você especificar uma sequência de colunas, as colunas do DataFrame serão organizadas nesta ordem:

```
In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[47]:
   year state pop
0 2000 Ohio 1.5
1 2001 Ohio 1.7
2 2002 Ohio 3.6
3 2001 Nevada 2.4
4 2002 Nevada 2.9
5 2003 Nevada 3.2
```

Se você passar uma coluna que não esteja contida no dicionário, ela aparecerá com valores ausentes no resultado:

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
   ....: index=['one', 'two', 'three', 'four',
```

```
....: 'five', 'six'])

In [49]: frame2
Out[49]:
       year state pop debt
one 2000 Ohio 1.5 NaN
two 2001 Ohio 1.7 NaN
three 2002 Ohio 3.6 NaN
four 2001 Nevada 2.4 NaN
five 2002 Nevada 2.9 NaN
six 2003 Nevada 3.2 NaN

In [50]: frame2.columns
Out[50]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Uma coluna em um DataFrame pode ser obtida como uma Series, seja usando uma notação do tipo dicionário ou por meio de atributo:

```
In [51]: frame2['state']
Out[51]:
one Ohio
two Ohio
three Ohio
four Nevada
five Nevada
six Nevada
Name: state, dtype: object

In [52]: frame2.year
Out[52]:
one 2000
two 2001
three 2002
four 2001
five 2002
six 2003
Name: year, dtype: int64
```

 Um acesso na forma de atributo (por exemplo, frame2.year) e um preenchimento automático de nomes de coluna com tabulação no IPython são oferecidos como conveniência.

frame2[column] funciona para qualquer nome de coluna, mas frame2.column só funcionará quando o nome da coluna for um nome de variável válido em Python.

Observe que a Series devolvida tem o mesmo índice que o DataFrame, e o seu atributo name foi definido de modo apropriado.

As linhas também podem ser obtidas com base na posição ou no nome, com o atributo especial loc (mais detalhes sobre esse assunto posteriormente):

```
In [53]: frame2.loc['three']
Out[53]:
year 2002
state Ohio
pop 3.6
debt NaN
Name: three, dtype: object
```

As colunas podem ser modificadas por atribuição. Por exemplo, a coluna vazia 'debt' poderia receber um valor escalar ou um array de valores:

```
In [54]: frame2['debt'] = 16.5

In [55]: frame2
Out[55]:
      year state pop debt
one 2000 Ohio 1.5 16.5
two 2001 Ohio 1.7 16.5
three 2002 Ohio 3.6 16.5
four 2001 Nevada 2.4 16.5
five 2002 Nevada 2.9 16.5
six 2003 Nevada 3.2 16.5

In [56]: frame2['debt'] = np.arange(6.)

In [57]: frame2
Out[57]:
      year state pop debt
one 2000 Ohio 1.5 0.0
two 2001 Ohio 1.7 1.0
three 2002 Ohio 3.6 2.0
```

```
four 2001 Nevada 2.4 3.0
five 2002 Nevada 2.9 4.0
six 2003 Nevada 3.2 5.0
```

Quando você estiver atribuindo listas ou arrays para uma coluna, o tamanho do valor deve coincidir com o tamanho do DataFrame. Se você atribuir uma Series, seus rótulos serão realinhados exatamente com o índice do DataFrame, e valores indicando ausência serão inseridos em quaisquer lacunas:

```
In [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [59]: frame2['debt'] = val

In [60]: frame2
Out[60]:
      year state pop debt
one 2000 Ohio 1.5 NaN
two 2001 Ohio 1.7 -1.2
three 2002 Ohio 3.6 NaN
four 2001 Nevada 2.4 -1.5
five 2002 Nevada 2.9 -1.7
six 2003 Nevada 3.2 NaN
```

Fazer uma atribuição a uma coluna que não exista fará uma nova coluna ser criada. A palavra reservada `del` apagará colunas, como ocorre em um dicionário.

Como um exemplo de `del`, inicialmente adicionarei uma nova coluna de valores booleanos, em que a coluna `state` seja igual a `'Ohio'`:

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'

In [62]: frame2
Out[62]:
      year state pop debt eastern
one 2000 Ohio 1.5 NaN True
two 2001 Ohio 1.7 -1.2 True
three 2002 Ohio 3.6 NaN True
four 2001 Nevada 2.4 -1.5 False
five 2002 Nevada 2.9 -1.7 False
six 2003 Nevada 3.2 NaN False
```


Novas colunas não podem ser criadas com a sintaxe frame2.eastern.

O método del pode então ser usado para remover essa coluna:

```
In [63]: del frame2['eastern']

In [64]: frame2.columns
Out[64]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```


A coluna devolvida pela indexação de um DataFrame é uma *visualização (view) dos dados subjacentes, e não uma cópia.* Assim, qualquer modificação in-place em Series se refletirá no DataFrame. A coluna pode ser explicitamente copiada com o método copy de Series.

Outro formato de dados comum é um dicionário de dicionários aninhados:

```
In [65]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
   ....: 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

Se o dicionário aninhado for passado para o DataFrame, o pandas interpretará as chaves do dicionário mais externo como as colunas e as chaves mais internas como os índices das linhas:

```
In [66]: frame3 = pd.DataFrame(pop)

In [67]: frame3
Out[67]:
     Nevada Ohio
2000 NaN 1.5
2001 2.4 1.7
2002 2.9 3.6
```

Podemos fazer a transposição do DataFrame (trocar linhas e colunas) com uma sintaxe semelhante àquela usada em um array NumPy:

```
In [68]: frame3.T
Out[68]:
       2000 2001 2002
Nevada NaN 2.4 2.9
Ohio 1.5 1.7 3.6
```

As chaves dos dicionários mais internos são combinadas e ordenadas para compor o índice no resultado. Isso não será verdade se um índice explícito for especificado:

```
In [69]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[69]:
     Nevada Ohio
2001 2.4 1.7
2002 2.9 3.6
2003 NaN NaN
```

Dicionários de Series são tratados praticamente do mesmo modo:

```
In [70]: pdata = {'Ohio': frame3['Ohio'][:-1],
   ....: 'Nevada': frame3['Nevada'][:2]}

In [71]: pd.DataFrame(pdata)
Out[71]:
     Nevada Ohio
2000 NaN 1.5
2001 2.4 1.7
```

Para ver uma lista completa dos dados que podemos passar para o construtor do DataFrame, consulte a Tabela 5.1.

Se index e columns de um DataFrame tiverem seus atributos name definidos, esses também serão exibidos:

```
In [72]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [73]: frame3
Out[73]:
state Nevada Ohio
year
2000 NaN 1.5
2001 2.4 1.7
2002 2.9 3.6
```

Tabela 5.1 – Possíveis entradas de dados para o construtor de DataFrame

Tipo	Observações
ndarray 2D	Uma matriz de dados, passando rótulos opcionais para linhas e colunas
Dicionário de	Cada sequência se transforma em uma coluna no DataFrame;

arrays, listas ou tuplas	todas as sequências devem ter o mesmo tamanho
Array NumPy estruturado/de registros	Tratado como o caso de “dicionário de arrays”
Dicionário de Series	Cada valor se transforma em uma coluna; os índices de cada Series são unidos para compor o índice das linhas do resultado caso nenhum índice explícito seja passado
Dicionário de dicionários	Cada dicionário interno se transforma em uma coluna; as chaves são unidas para compor o índice das linhas, como no caso do “dicionário de Series”
Lista de dicionários ou de Series	Cada item se transforma em uma linha no DataFrame; a união das chaves do dicionário ou dos índices de Series se transforma nos rótulos das colunas do DataFrame
Lista de listas ou de tuplas	Tratado como o caso de “ndarray 2D”
Outro DataFrame	Os índices do DataFrame são usados, a menos que índices diferentes sejam passados
MaskedArray do NumPy	Como no caso do “ndarray 2D”, exceto que valores com máscara passam a ser NA/ausentes no DataFrame resultante

Como ocorre com Series, o atributo values devolve os dados contidos no DataFrame como um ndarray bidimensional:

```
In [74]: frame3.values
Out[74]:
array([[ nan, 1.5],
       [ 2.4, 1.7],
       [ 2.9, 3.6]])
```

Se as colunas do DataFrame tiverem dtypes distintos, o dtype do array de valores será escolhido de modo a acomodar todas as colunas:

```
In [75]: frame2.values
Out[75]:
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

Objetos Index

Os objetos Index do pandas são responsáveis por armazenar os rótulos dos eixos e outros metadados (como o nome ou os nomes dos eixos). Qualquer array ou outra sequência de rótulos que você usar ao construir uma Series ou um DataFrame será internamente convertido em um Index:

```
In [76]: obj = pd.Series(range(3), index=['a', 'b', 'c'])

In [77]: index = obj.index

In [78]: index
Out[78]: Index(['a', 'b', 'c'], dtype='object')

In [79]: index[1:]
Out[79]: Index(['b', 'c'], dtype='object')
```

Os objetos Index são imutáveis e, desse modo, não podem ser modificados pelo usuário:

```
index[1] = 'd' # TypeError
```

A imutabilidade faz com que seja mais seguro compartilhar objetos Index entre estruturas de dados:

```
In [80]: labels = pd.Index(np.arange(3))

In [81]: labels
Out[81]: Int64Index([0, 1, 2], dtype='int64')

In [82]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [83]: obj2
Out[83]:
0 1.5
1 -2.5
2 0.0
dtype: float64

In [84]: obj2.index is labels
Out[84]: True
```

Alguns usuários não tirarão frequentemente proveito dos recursos oferecidos pelos índices; entretanto, como algumas operações produzirão resultados contendo dados indexados, é importante compreender como eles funcionam.

Além de ser como os arrays, um Index também se comporta como um conjunto de tamanho fixo:

```
In [85]: frame3
Out[85]:
state Nevada Ohio
year
2000 NaN 1.5
2001 2.4 1.7
2002 2.9 3.6

In [86]: frame3.columns
Out[86]: Index(['Nevada', 'Ohio'], dtype='object', name='state')

In [87]: 'Ohio' in frame3.columns
Out[87]: True

In [88]: 2003 in frame3.index
Out[88]: False
```

De modo diferente dos conjuntos Python, um Index do pandas pode conter rótulos duplicados:

```
In [89]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])

In [90]: dup_labels
Out[90]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

As seleções com rótulos duplicados conterão todas as ocorrências desse rótulo.

Cada Index tem uma série de métodos e propriedades para lógica de conjuntos, o que responde a outras perguntas comuns sobre os dados que ele contém. Alguns métodos e propriedades úteis estão sintetizados na Tabela 5.2.

Tabela 5.2 – Alguns métodos e propriedades de Index

Método	Descrição
append	Concatena com objetos Index adicionais, gerando um novo Index
difference	Calcula a diferença entre conjuntos como um Index
intersection	Calcula a intersecção entre conjuntos
union	Calcula a união entre conjuntos
isin	Calcula um array booleano indicando se cada valor está contido na coleção recebida
delete	Calcula um novo Index com o elemento no índice i apagado
drop	Calcula um novo Index apagando os valores recebidos
insert	Calcula um novo Index inserindo um elemento no índice i
is_monotonic	Devolve True se cada elemento for maior ou igual ao elemento anterior
is_unique	Devolve True se o Index não tiver valores duplicados
unique	Calcula o array de valores únicos no Index

5.2 Funcionalidades essenciais

Esta seção descreverá o mecanismo fundamental da interação com os dados contidos em uma Series ou um DataFrame. Nos próximos capítulos, mergulharemos mais profundamente nos assuntos referentes à análise e à manipulação de dados usando o pandas. O propósito deste livro não é servir como uma documentação exaustiva da biblioteca pandas; em vez disso, manteremos o foco nos recursos mais importantes, deixando os aspectos menos comuns (ou seja, os mais exotéricos) para você explorar por conta própria.

Reindexação

Um método importante dos objetos do pandas é reindex, que implica criar um novo objeto com os dados *de acordo com* um novo índice. Vamos considerar um exemplo:

```
In [91]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [92]: obj
```

```
Out[92]:
d 4.5
b 7.2
a -5.3
c 3.6
dtype: float64
```

Chamar reindex nessa Series reorganiza os dados de acordo com o novo índice, introduzindo valores indicativos de ausência se algum valor de índice não estava presente antes:

```
In [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [94]: obj2
Out[94]:
a -5.3
b 7.2
c 3.6
d 4.5
e NaN
dtype: float64
```

Para dados ordenados, como as séries temporais, talvez seja desejável fazer alguma interpolação ou preenchimento de valores na reindexação. A opção method nos permite fazer isso, usando um método como ffill, que faz um preenchimento para a frente (forward-fill) dos valores:

```
In [95]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [96]: obj3
Out[96]:
0 blue
2 purple
4 yellow
dtype: object

In [97]: obj3.reindex(range(6), method='ffill')
Out[97]:
0 blue
1 blue
2 purple
```

```
3 purple
4 yellow
5 yellow
dtype: object
```

Com DataFrame, reindex pode alterar o índice (linha), as colunas, ou ambos. Se apenas uma sequência for passada, as linhas serão reindexadas no resultado:

```
In [98]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   ....: index=['a', 'c', 'd'],
   ....: columns=['Ohio', 'Texas', 'California'])

In [99]: frame
Out[99]:
   Ohio Texas California
a 0 1 2
c 3 4 5
d 6 7 8

In [100]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [101]: frame2
Out[101]:
   Ohio Texas California
a 0.0 1.0 2.0
b NaN NaN NaN
c 3.0 4.0 5.0
d 6.0 7.0 8.0
```

As colunas podem ser reindexadas com a palavra reservada columns:

```
In [102]: states = ['Texas', 'Utah', 'California']

In [103]: frame.reindex(columns=states)
Out[103]:
   Texas Utah California
a 1 NaN 2
c 4 NaN 5
d 7 NaN 8
```

Veja a Tabela 5.3 para saber mais sobre os argumentos de reindex.

Conforme exploraremos de modo mais detalhado, podemos reindexar de forma mais sucinta, indexando por rótulos com loc, e muitos usuários preferem usar exclusivamente essa opção:

```
In [104]: frame.loc[['a', 'b', 'c', 'd'], states]
Out[104]:
   Texas Utah California
a 1.0 NaN 2.0
b NaN NaN NaN
c 4.0 NaN 5.0
d 7.0 NaN 8.0
```

Tabela 5.3 – Argumentos da função reindex

Argumento	Descrição
index	Nova sequência para ser usada como índice. Pode ser uma instância de Index ou qualquer outra estrutura de dados Python do tipo sequência. Um Index será usado exatamente como está, sem qualquer cópia.
method	Método de interpolação (preenchimento); 'ffill' preenche para a frente, enquanto 'bfill' preenche para trás.
fill_value	Valor de substituição a ser usado quando dados ausentes forem introduzidos pela reindexação.
limit	Quando fizer o preenchimento para a frente ou para trás, é o tamanho máximo da lacuna (em número de elementos) a ser preenchido.
tolerance	Quando fizer o preenchimento para a frente ou para trás, é o tamanho máximo da lacuna (em distância numérica absoluta) para preencher no caso de correspondências inexatas.
level	Faz a correspondência entre um Index simples e um nível de MultiIndex; caso contrário, seleciona um subconjunto.
copy	Se for True, sempre copia os dados subjacentes, mesmo que o novo índice seja equivalente ao antigo; se for False, não copia os dados quando os índices forem equivalentes.

Descartando entradas de um eixo

Descartar uma ou mais entradas de um eixo é fácil se você já tiver um array ou uma lista de índices sem essas entradas. Como isso pode exigir um pouco de manipulação de dados e lógica de conjuntos, o método drop devolverá um novo objeto com o valor ou

os valores indicados apagados de um eixo:

```
In [105]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])

In [106]: obj
Out[106]:
a 0.0
b 1.0
c 2.0
d 3.0
e 4.0
dtype: float64

In [107]: new_obj = obj.drop('c')

In [108]: new_obj
Out[108]:
a 0.0
b 1.0
d 3.0
e 4.0
dtype: float64

In [109]: obj.drop(['d', 'c'])
Out[109]:
a 0.0
b 1.0
e 4.0
dtype: float64
```

Com DataFrame, os valores dos índices podem ser apagados de qualquer eixo. Para demonstrar isso, inicialmente criaremos um DataFrame de exemplo:

```
In [110]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....: index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....: columns=['one', 'two', 'three', 'four'])

In [111]: data
Out[111]:
          one two three four
Ohio 0 1 2 3
```

```
Colorado 4 5 6 7
Utah 8 9 10 11
New York 12 13 14 15
```

Chamar drop com uma sequência de rótulos fará valores serem descartados das linhas com esses rótulos (eixo 0):

```
In [112]: data.drop(['Colorado', 'Ohio'])
Out[112]:
        one two three four
Utah 8 9 10 11
New York 12 13 14 15
```

Podemos descartar valores das colunas passando axis=1 ou axis='columns':

```
In [113]: data.drop('two', axis=1)
Out[113]:
        one three four
Ohio 0 2 3
Colorado 4 6 7
Utah 8 10 11
New York 12 14 15

In [114]: data.drop(['two', 'four'], axis='columns')
Out[114]:
        one three
Ohio 0 2
Colorado 4 6
Utah 8 10
New York 12 14
```

Muitas funções, como drop, que modificam o tamanho ou o formato de uma Series ou de um DataFrame, são capazes de manipular um objeto *in-place, sem devolver um novo objeto:*

```
In [115]: obj.drop('c', inplace=True)

In [116]: obj
Out[116]:
a 0.0
b 1.0
d 3.0
e 4.0
```

```
dtype: float64
```

Tome cuidado com o `inplace`, pois ele destruirá qualquer dado descartado.

Indexação, seleção e filtragem

A indexação de séries (`obj[...]`) funciona de modo análogo à indexação de arrays NumPy, exceto que você pode usar os valores de índice da Series em vez de utilizar somente inteiros. Eis alguns exemplos disso:

```
In [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [118]: obj
Out[118]:
a 0.0
b 1.0
c 2.0
d 3.0
dtype: float64

In [119]: obj['b']
Out[119]: 1.0

In [120]: obj[1]
Out[120]: 1.0

In [121]: obj[2:4]
Out[121]:
c 2.0
d 3.0
dtype: float64

In [122]: obj[['b', 'a', 'd']]
Out[122]:
b 1.0
a 0.0
d 3.0
dtype: float64
```

```
In [123]: obj[[1, 3]]
Out[123]:
b 1.0
d 3.0
dtype: float64

In [124]: obj[obj < 2]
Out[124]:
a 0.0
b 1.0
dtype: float64
```

O fatiamento com rótulos comporta-se de modo diferente do fatiamento usual de Python, pois o ponto final estará incluído:

```
In [125]: obj['b':'c']
Out[125]:
b 1.0
c 2.0
dtype: float64
```

Uma definição usando esses métodos modificará a seção correspondente da Series:

```
In [126]: obj['b':'c'] = 5

In [127]: obj
Out[127]:
a 0.0
b 5.0
c 5.0
d 3.0
dtype: float64
```

A indexação em um DataFrame serve para obter uma ou mais colunas, seja com um único valor ou com uma sequência:

```
In [128]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....: index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....: columns=['one', 'two', 'three', 'four'])

In [129]: data
Out[129]:
          one two three four
```

```
Ohio 0 1 2 3
Colorado 4 5 6 7
Utah 8 9 10 11
New York 12 13 14 15

In [130]: data['two']
Out[130]:
Ohio 1
Colorado 5
Utah 9
New York 13
Name: two, dtype: int64

In [131]: data[['three', 'one']]
Out[131]:
          three one
Ohio 2 0
Colorado 6 4
Utah 10 8
New York 14 12
```

Uma indexação como essa tem alguns casos especiais. Em primeiro lugar, temos o fatiamento e a seleção de dados com um array booleano:

```
In [132]: data[:2]
Out[132]:
          one two three four
Ohio 0 1 2 3
Colorado 4 5 6 7

In [133]: data[data['three'] > 5]
Out[133]:
          one two three four
Colorado 4 5 6 7
Utah 8 9 10 11
New York 12 13 14 15
```

A sintaxe de seleção de linhas `data[:2]` é disponibilizada como uma conveniência. Passar um único elemento ou uma lista para o operador `[]` seleciona colunas.

Outro caso de uso está na indexação com um DataFrame booleano, como aquele gerado por uma comparação escalar:

```
In [134]: data < 5
Out[134]:
          one two three four
Ohio True True True True
Colorado True False False False
Utah False False False False
New York False False False False

In [135]: data[data < 5] = 0

In [136]: data
Out[136]:
         one two three four
Ohio 0 0 0 0
Colorado 0 5 6 7
Utah 8 9 10 11
New York 12 13 14 15
```

Isso faz com que o DataFrame seja sintaticamente mais semelhante a um array NumPy bidimensional, nesse caso em particular.

Seleção com loc e iloc

Para indexação nas linhas do DataFrame com rótulos, apresentarei os operadores especiais de indexação loc e iloc. Eles permitem selecionar um subconjunto de linhas e colunas de um DataFrame com uma notação semelhante àquela do NumPy, usando rótulos de eixo (loc) ou inteiros(iloc).

Como exemplo preliminar, vamos selecionar uma única linha e várias colunas pelo rótulo:

```
In [137]: data.loc['Colorado', ['two', 'three']]
Out[137]:
two 5
three 6
Name: Colorado, dtype: int64
```

Então, faremos algumas seleções semelhantes com inteiros usando

iloc:

```
In [138]: data.iloc[2, [3, 0, 1]]
Out[138]:
four 11
one 8
two 9
Name: Utah, dtype: int64

In [139]: data.iloc[2]
Out[139]:
one 8
two 9
three 10
four 11
Name: Utah, dtype: int64

In [140]: data.iloc[[1, 2], [3, 0, 1]]
Out[140]:
          four one two
Colorado 7 0 5
Utah 11 8 9
```

As duas funções de indexação trabalham com fatias, além de rótulos únicos ou listas de rótulos:

```
In [141]: data.loc[:'Utah', 'two']
Out[141]:
Ohio 0
Colorado 5
Utah 9
Name: two, dtype: int64

In [142]: data.iloc[:, :3][data.three > 5]
Out[142]:
          one two three
Colorado 0 5 6
Utah 8 9 10
New York 12 13 14
```

Portanto, há muitas maneiras de selecionar e reorganizar os dados contidos em um objeto do pandas. Para um DataFrame, a Tabela

5.4 apresenta um breve resumo de muitas delas. Como veremos mais adiante, há uma série de opções adicionais para trabalhar com índices hierárquicos.

 Quando fiz inicialmente o design do pandas, achei que precisar digitar frame[:, col] para selecionar uma coluna era muito verboso (e suscetível a erros), pois a seleção de colunas é uma das operações mais comuns. Fiz uma negociação de custo-benefício no design, de modo a colocar todo o comportamento da indexação sofisticada (tanto rótulos quanto inteiros) no operador ix. Na prática, isso resultou em muitos casos inusitados em dados cujos rótulos eram inteiros nos eixos, de modo que a equipe do pandas decidiu criar os operadores loc e iloc para lidar com a indexação baseada estritamente em rótulos e em inteiros, respectivamente.

O operador de indexação ix continua existindo, mas é considerado obsoleto. Não recomendo usá-lo.

Tabela 5.4 – Opções de indexação com DataFrame

Tipo	Observações
df[val]	Seleciona uma única coluna ou uma sequência de colunas do DataFrame; conveniências para casos especiais: array booleano (filtra linhas), fatia (fatia linhas) ou DataFrame booleano (define valores com base em algum critério)
df.loc[val]	Seleciona uma única linha ou um subconjunto de linhas do DataFrame pelo rótulo
df.loc[:, val]	Seleciona uma única coluna ou um subconjunto de colunas pelo rótulo
df.loc[val1, val2]	Seleciona tanto linhas quanto colunas pelo rótulo
df.iloc[where]	Seleciona uma única linha ou um subconjunto de linhas do DataFrame pela posição com um inteiro
df.iloc[:, where]	Seleciona uma única coluna ou um subconjunto de colunas pela posição com um inteiro
df.iloc[where_i, where_j]	Seleciona tanto linhas quanto colunas pela posição com um inteiro
df.at[label_i, label_j]	Seleciona um único valor escalar pelo rótulo da linha e da coluna
df.iat[i, j]	Seleciona um único valor escalar pela posição (inteiros) da linha e da coluna

Tipo	Observações
método reindex	Seleciona linhas ou colunas pelos rótulos
métodos get_value, set_value	Seleciona um único valor pelo rótulo da linha e da coluna

Índices inteiros

Trabalhar com objetos do pandas indexados por inteiros é algo que, com frequência, confunde os novos usuários por causa de algumas diferenças com a semântica de indexação em estruturas de dados embutidas de Python, como listas e tuplas. Por exemplo, talvez você não esperasse que o código a seguir gerasse um erro:

```
ser = pd.Series(np.arange(3.))
ser
ser[-1]
```

Nesse caso, o pandas poderia “recorrer” à indexação com inteiros, mas é difícil fazer isso de modo geral, sem introduzir bugs sutis. Temos aqui um índice contendo 0, 1, 2, mas inferir o que o usuário quer (indexação baseada em rótulo ou baseada em posição) é difícil:

```
In [144]: ser
Out[144]:
0 0.0
1 1.0
2 2.0
dtype: float64
```

Por outro lado, com um índice que não seja inteiro, não há potencial para ambiguidades:

```
In [145]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])

In [146]: ser2[-1]
Out[146]: 2.0
```

Para manter o código consistente, se você tiver um índice de eixo contendo inteiros, a seleção de dados sempre será orientada a

rótulos. Para um tratamento mais preciso, utilize loc (para rótulos) ou iloc (para inteiros):

```
In [147]: ser[:1]
Out[147]:
0 0.0
dtype: float64

In [148]: ser.loc[:1]
Out[148]:
0 0.0
1 1.0
dtype: float64

In [149]: ser.iloc[:1]
Out[149]:
0 0.0
dtype: float64
```

Aritmética e alinhamento de dados

Um recurso importante do pandas para algumas aplicações é o comportamento da aritmética entre objetos com índices diferentes. Quando objetos estiverem sendo somados, se algum par de índices não for igual, o respectivo índice no resultado será a união dos pares de índices. Para usuários com experiência em banco de dados, isso é semelhante a uma outer join (junção externa) automática nos rótulos do índice. Vamos ver um exemplo:

```
In [150]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [151]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
   .....: index=['a', 'c', 'e', 'f', 'g'])

In [152]: s1
Out[152]:
a 7.3
c -2.5
d 3.4
e 1.5
```

```
dtype: float64

In [153]: s2
Out[153]:
a -2.1
c 3.6
e -1.5
f 4.0
g 3.1
dtype: float64
```

Somar isso resulta em:

```
In [154]: s1 + s2
Out[154]:
a 5.2
c 1.1
d NaN
e 0.0
f NaN
g NaN
dtype: float64
```

O alinhamento de dados interno introduz valores indicativos de ausência nos locais dos rótulos que não se sobrepõem. Esses valores então se propagarão para outros cálculos aritméticos.

No caso do DataFrame, o alinhamento é feito tanto nas linhas quanto nas colunas:

```
In [155]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
   .....: index=['Ohio', 'Texas', 'Colorado'])

In [156]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [157]: df1
Out[157]:
          b c d
Ohio 0.0 1.0 2.0
Texas 3.0 4.0 5.0
Colorado 6.0 7.0 8.0
```

```
In [158]: df2
Out[158]:
          b d e
Utah 0.0 1.0 2.0
Ohio 3.0 4.0 5.0
Texas 6.0 7.0 8.0
Oregon 9.0 10.0 11.0
```

Somar esses dados devolve um DataFrame cujos índice e colunas são as uniões dos dados de cada DataFrame:

```
In [159]: df1 + df2
Out[159]:
            b c d e
Colorado NaN NaN NaN NaN
Ohio 3.0 NaN 6.0 NaN
Oregon NaN NaN NaN NaN
Texas 9.0 NaN 12.0 NaN
Utah NaN NaN NaN NaN
```

Como as colunas 'c' e 'e' não se encontram nos dois objetos DataFrame, eles aparecerão como dados ausentes no resultado. O mesmo vale para linhas cujos rótulos não são comuns aos dois objetos.

Se você somar objetos DataFrame sem rótulos para colunas ou linhas em comum, o resultado conterá somente nulos:

```
In [160]: df1 = pd.DataFrame({'A': [1, 2]})

In [161]: df2 = pd.DataFrame({'B': [3, 4]})

In [162]: df1
Out[162]:
   A
0 1
1 2

In [163]: df2
Out[163]:
   B
0 3
1 4
```

```
In [164]: df1 - df2
Out[164]:
   A B
0 NaN NaN
1 NaN NaN
```

Métodos aritméticos com valores para preenchimento

Em operações aritméticas entre objetos indexados de modo diferente, talvez você queira fazer um preenchimento com um valor especial, como 0, quando um rótulo de eixo for encontrado em um objeto, mas não no outro:

```
In [165]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
   .....: columns=list('abcd'))

In [166]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
   .....: columns=list('abcde'))

In [167]: df2.loc[1, 'b'] = np.nan

In [168]: df1
Out[168]:
     a b c d
0 0.0 1.0 2.0 3.0
1 4.0 5.0 6.0 7.0
2 8.0 9.0 10.0 11.0

In [169]: df2
Out[169]:
      a b c d e
0 0.0 1.0 2.0 3.0 4.0
1 5.0 NaN 7.0 8.0 9.0
2 10.0 11.0 12.0 13.0 14.0
3 15.0 16.0 17.0 18.0 19.0
```

Somar esses dadis resulta em valores NA nas posições que não se sobrepõem:

```
In [170]: df1 + df2
```

```
Out[170]:
    a b c d e
0 0.0 2.0 4.0 6.0 NaN
1 9.0 NaN 13.0 15.0 NaN
2 18.0 20.0 22.0 24.0 NaN
3 NaN NaN NaN NaN NaN
```

Usando o método add em df1, passarei df2 e um argumento para fill_value:

```
In [171]: df1.add(df2, fill_value=0)
Out[171]:
    a b c d e
0 0.0 2.0 4.0 6.0 4.0
1 9.0 5.0 13.0 15.0 9.0
2 18.0 20.0 22.0 24.0 14.0
3 15.0 16.0 17.0 18.0 19.0
```

Veja a Tabela 5.5, que contém uma lista de métodos de Series e DataFrame para operações aritméticas. Cada um deles tem uma contrapartida que começa com a letra r, com argumentos invertidos. Assim, as duas instruções a seguir são equivalentes:

```
In [172]: 1 / df1
Out[172]:
      a b c d
0 inf 1.000000 0.500000 0.333333
1 0.250000 0.200000 0.166667 0.142857
2 0.125000 0.111111 0.100000 0.090909

In [173]: df1.rdiv(1)
Out[173]:
      a b c d
0 inf 1.000000 0.500000 0.333333
1 0.250000 0.200000 0.166667 0.142857
2 0.125000 0.111111 0.100000 0.090909
```

Relacionado a esse caso, quando reindexamos uma Series ou um DataFrame, podemos também especificar um valor diferente para preenchimento:

```
In [174]: df1.reindex(columns=df2.columns, fill_value=0)
Out[174]:
```

```
   a b c d e
0 0.0 1.0 2.0 3.0 0
1 4.0 5.0 6.0 7.0 0
2 8.0 9.0 10.0 11.0 0
```

Tabela 5.5 – Métodos aritméticos flexíveis

Método	Descrição
add, radd	Métodos para adição (+)
sub, rsub	Métodos para subtração (-)
div, rdiv	Métodos para divisão (/)
floordiv, rfloordiv	Métodos para divisão pelo piso (//)
mul, rmul	Métodos para multiplicação (*)
pow, rpow	Métodos para exponencial (**)

Operações entre DataFrame e Series

Como ocorre com os arrays NumPy de dimensões diferentes, a aritmética entre DataFrame e Series também está definida. Inicialmente, como um exemplo motivador, considere a diferença entre um array bidimensional e uma de suas linhas:

```
In [175]: arr = np.arange(12.).reshape((3, 4))

In [176]: arr
Out[176]:
array([[ 0., 1., 2., 3.],
       [ 4., 5., 6., 7.],
       [ 8., 9., 10., 11.]])

In [177]: arr[0]
Out[177]: array([ 0., 1., 2., 3.])

In [178]: arr - arr[0]
Out[178]:
array([[ 0., 0., 0., 0.],
       [ 4., 4., 4., 4.],
       [ 8., 8., 8., 8.]])
```

Quando subtraímos `arr[0]` de `arr`, a subtração é realizada uma vez para cada linha. Isso é chamado de *broadcasting*, e será explicado

com mais detalhes, conforme se relaciona aos arrays NumPy genéricos, no Apêndice A. As operações entre um DataFrame e uma Series são semelhantes:

```
In [179]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
   .....: columns=list('bde'),
   .....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [180]: series = frame.iloc[0]

In [181]: frame
Out[181]:
          b d e
Utah 0.0 1.0 2.0
Ohio 3.0 4.0 5.0
Texas 6.0 7.0 8.0
Oregon 9.0 10.0 11.0

In [182]: series
Out[182]:
b 0.0
d 1.0
e 2.0
Name: Utah, dtype: float64
```

Por padrão, a aritmética entre DataFrame e Series realiza a correspondência entre o índice da Series e as colunas do DataFrame, fazendo broadcasting pelas linhas:

```
In [183]: frame - series
Out[183]:
          b d e
Utah 0.0 0.0 0.0
Ohio 3.0 3.0 3.0
Texas 6.0 6.0 6.0
Oregon 9.0 9.0 9.0
```

Se o valor de um índice não for encontrado nas colunas do DataFrame nem no índice de Series, os objetos serão reindexados para formar a união:

```
In [184]: series2 = pd.Series(range(3), index=['b', 'e', 'f'])
```

```
In [185]: frame + series2
Out[185]:
          b d e f
Utah 0.0 NaN 3.0 NaN
Ohio 3.0 NaN 6.0 NaN
Texas 6.0 NaN 9.0 NaN
Oregon 9.0 NaN 12.0 NaN
```

Se, por outro lado, você quiser fazer broadcast pelas colunas, fazendo correspondências nas linhas, terá que usar um dos métodos aritméticos. Por exemplo:

```
In [186]: series3 = frame['d']

In [187]: frame
Out[187]:
          b d e
Utah 0.0 1.0 2.0
Ohio 3.0 4.0 5.0
Texas 6.0 7.0 8.0
Oregon 9.0 10.0 11.0

In [188]: series3
Out[188]:
Utah 1.0
Ohio 4.0
Texas 7.0
Oregon 10.0
Name: d, dtype: float64

In [189]: frame.sub(series3, axis='index')
Out[189]:
          b d e
Utah -1.0 0.0 1.0
Ohio -1.0 0.0 1.0
Texas -1.0 0.0 1.0
Oregon -1.0 0.0 1.0
```

O número do eixo que você passar é o *eixo a ser correspondido*. Nesse caso, pretendemos fazer a correspondência do índice da linha do DataFrame (axis='index' ou axis=0) e fazer o broadcast.

Aplicação de funções e mapeamento

As ufuncs do NumPy (métodos de array para todos os elementos) também funcionam com objetos do pandas:

```
In [190]: frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
   .....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [191]: frame
Out[191]:
               b d e
Utah -0.204708 0.478943 -0.519439
Ohio -0.555730 1.965781 1.393406
Texas 0.092908 0.281746 0.769023
Oregon 1.246435 1.007189 -1.296221

In [192]: np.abs(frame)
Out[192]:
               b d e
Utah 0.204708 0.478943 0.519439
Ohio 0.555730 1.965781 1.393406
Texas 0.092908 0.281746 0.769023
Oregon 1.246435 1.007189 1.296221
```

Outra operação frequente consiste em aplicar uma função em arrays unidimensionais para cada coluna ou linha. O método `apply` de DataFrame faz exatamente isso:

```
In [193]: f = lambda x: x.max() - x.min()

In [194]: frame.apply(f)
Out[194]:
b 1.802165
d 1.684034
e 2.689627
dtype: float64
```

Nesse caso, a função `f`, que calcula a diferença entre o máximo e o mínimo de uma Series, é chamada uma vez em cada coluna de `frame`. O resultado é uma Series com as colunas de `frame` como seu índice.

Por outro lado, se você passar `axis='columns'` para `apply`, a função será

chamada uma vez por linha:

```
In [195]: frame.apply(f, axis='columns')
Out[195]:
Utah 0.998382
Ohio 2.521511
Texas 0.676115
Oregon 2.542656
dtype: float64
```

Muitas das estatísticas mais comuns em arrays (como sum e mean) são métodos de DataFrame, portanto usar apply não será necessário.

A função passada para apply não precisa devolver um valor escalar; ela também pode devolver uma Series com múltiplos valores:

```
In [196]: def f(x):
   .....: return pd.Series([x.min(), x.max()], index=['min', 'max'])

In [197]: frame.apply(f)
Out[197]:
          b d e
min -0.555730 0.281746 -1.296221
max 1.246435 1.965781 1.393406
```

Funções Python para todos os elementos também podem ser usadas. Suponha que você quisesse calcular uma string formatada para cada valor de ponto flutuante em frame. Isso poderia ser feito com applymap:

```
In [198]: format = lambda x: '%.2f' % x

In [199]: frame.applymap(format)
Out[199]:
          b d e
Utah -0.20 0.48 -0.52
Ohio -0.56 1.97 1.39
Texas 0.09 0.28 0.77
Oregon 1.25 1.01 -1.30
```

O motivo para o nome applymap está no fato de Series ter um método map para aplicar uma função em todos os elementos:

```
In [200]: frame['e'].map(format)
Out[200]:
Utah -0.52
Ohio 1.39
Texas 0.77
Oregon -1.30
Name: e, dtype: object
```

Ordenação e classificação

Ordenar um conjunto de dados de acordo com algum critério é outra operação embutida importante. Para ordenar de modo lexicográfico pelo índice da linha ou da coluna, utilize o método sort_index, que devolve um novo objeto ordenado:

```
In [201]: obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])

In [202]: obj.sort_index()
Out[202]:
a 1
b 2
c 3
d 0
dtype: int64
```

Com um DataFrame, você pode ordenar pelo índice em qualquer eixo:

```
In [203]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
   .....: index=['three', 'one'],
   .....: columns=['d', 'a', 'b', 'c'])

In [204]: frame.sort_index()
Out[204]:
       d a b c
one 4 5 6 7
three 0 1 2 3

In [205]: frame.sort_index(axis=1)
Out[205]:
       a b c d
three 1 2 3 0
```

```
one 5 6 7 4
```

Os dados são ordenados em ordem crescente por padrão, mas podem ser ordenados também em ordem decrescente:

```
In [206]: frame.sort_index(axis=1, ascending=False)
Out[206]:
      d c b a
three 0 3 2 1
one 4 7 6 5
```

Para ordenar uma Series de acordo com seus valores, utilize o seu método sort_values:

```
In [207]: obj = pd.Series([4, 7, -3, 2])

In [208]: obj.sort_values()
Out[208]:
2 -3
3 2
0 4
1 7
dtype: int64
```

Qualquer valor indicativo de ausência será ordenado no final da Series, por padrão:

```
In [209]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [210]: obj.sort_values()
Out[210]:
4 -3.0
5 2.0
0 4.0
2 7.0
1 NaN
3 NaN
dtype: float64
```

Quando ordenar um DataFrame, você poderá usar os dados de uma ou mais colunas como chaves de ordenação. Para isso, passe um ou mais nomes de coluna para a opção by de sort_values:

```
In [211]: frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
```

```
In [212]: frame
Out[212]:
   a b
0 0 4
1 1 7
2 0 -3
3 1 2

In [213]: frame.sort_values(by='b')
Out[213]:
   a b
2 0 -3
3 1 2
0 0 4
1 1 7
```

Para ordenar de acordo com várias colunas, passe uma lista de nomes:

```
In [214]: frame.sort_values(by=['a', 'b'])
Out[214]:
   a b
2 0 -3
0 0 4
3 1 2
1 1 7
```

A *classificação* (ranking) atribui posições de um até o número de pontos de dados válidos em um array. Os métodos rank de Series e de DataFrame são aqueles a serem observados; por padrão, rank resolve empates atribuindo a cada grupo a classificação média:

```
In [215]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [216]: obj.rank()
Out[216]:
0 6.5
1 1.0
2 6.5
3 4.5
4 3.0
5 2.0
```

```
6 4.5
dtype: float64
```

As classificações também podem ser atribuídas de acordo com a ordem em que são observadas nos dados:

```
In [217]: obj.rank(method='first')
Out[217]:
0 6.0
1 1.0
2 7.0
3 4.0
4 3.0
5 2.0
6 5.0
dtype: float64
```

Nesse caso, em vez de usar a classificação média 6.5 para as entradas 0 e 2, elas foram definidas com 6 e 7 porque o rótulo 0 antecede o rótulo 2 nos dados.

Você pode classificar em ordem decrescente também:

```
# Atribui a classificação máxima no grupo em caso de empates
In [218]: obj.rank(ascending=False, method='max')
Out[218]:
0 2.0
1 7.0
2 2.0
3 4.0
4 5.0
5 6.0
6 4.0
dtype: float64
```

Veja a Tabela 5.6 que apresenta uma lista de métodos de desempate disponíveis.

O DataFrame pode calcular classificações nas linhas ou nas colunas:

```
In [219]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
   .....: 'c': [-2, 5, 8, -2.5]})
```

```
In [220]: frame
Out[220]:
   a b c
0 0 4.3 -2.0
1 1 7.0 5.0
2 0 -3.0 8.0
3 1 2.0 -2.5

In [221]: frame.rank(axis='columns')
Out[221]:
     a b c
0 2.0 3.0 1.0
1 1.0 3.0 2.0
2 2.0 1.0 3.0
3 2.0 3.0 1.0
```

Tabela 5.6 – Métodos de desempate com rank

Método	Descrição
'average'	Default: atribui a classificação média para cada entrada no mesmo grupo
'min'	Utiliza a classificação mínima do grupo todo
'max'	Utiliza a classificação máxima do grupo todo
'first'	Atribui classificações na ordem em que os valores aparecem nos dados
'dense'	Como method='min', porém as classificações sempre aumentam de 1 entre grupos, em vez do número de elementos iguais em um grupo

Índices de eixos com rótulos duplicados

Até agora, todos os exemplos que vimos tinham rótulos (valores de índice) únicos nos eixos. Embora muitas funções do pandas (como reindex) exijam que os rótulos sejam únicos, isso não é obrigatório. Vamos considerar uma pequena Series com índices duplicados:

```
In [222]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [223]: obj
Out[223]:
a 0
a 1
```

```
b 2
b 3
c 4
dtype: int64
```

A propriedade is_unique do índice pode dizer se seus rótulos são únicos ou não:

```
In [224]: obj.index.is_unique
Out[224]: False
```

A seleção de dados é uma das principais tarefas que se comporta de modo diferente com duplicatas. Indexar um rótulo com várias entradas devolve uma Series, enquanto entradas únicas devolvem um valor escalar:

```
In [225]: obj['a']
Out[225]:
a 0
a 1
dtype: int64

In [226]: obj['c']
Out[226]: 4
```

Isso pode deixar seu código mais complicado, pois o tipo da saída da indexação pode variar conforme um rótulo esteja repetido ou não.

A mesma lógica se estende para a indexação de linhas em um DataFrame:

```
In [227]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [228]: df
Out[228]:
          0 1 2
a 0.274992 0.228913 1.352917
a 0.886429 -2.001637 -0.371843
b 1.669025 -0.438570 -0.539741
b 0.476985 3.248944 -1.021228

In [229]: df.loc['b']
Out[229]:
```

```
          0 1 2
b 1.669025 -0.438570 -0.539741
b 0.476985 3.248944 -1.021228
```

5.3 Resumindo e calculando estatísticas descritivas

Os objetos do pandas estão equipados com um conjunto de métodos matemáticos e estatísticos comuns. A maior parte deles se enquadra na categoria de *reduções* ou de *estatísticas de resumo*: são métodos que extraem um único valor (como a soma ou a média) de uma Series ou uma Series de valores das linhas ou colunas de um DataFrame. Em comparação com métodos similares que se encontram em arrays NumPy, eles têm tratamento embutido para dados ausentes. Considere um pequeno DataFrame:

```
In [230]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
   .....: [np.nan, np.nan], [0.75, -1.3]],
   .....: index=['a', 'b', 'c', 'd'],
   .....: columns=['one', 'two'])

In [231]: df
Out[231]:
    one two
a 1.40 NaN
b 7.10 -4.5
c NaN NaN
d 0.75 -1.3
```

Chamar o método `sum` de DataFrame devolve uma Series contendo as somas das colunas:

```
In [232]: df.sum()
Out[232]:
one 9.25
two -5.80
dtype: float64
```

Por outro lado, passar `axis='columns'` ou `axis=1` faz a soma pelas colunas:

```
In [233]: df.sum(axis='columns')
Out[233]:
a 1.40
b 2.60
c NaN
d -0.55
dtype: float64
```

Valores NA são excluídos, a menos que a fatia inteira (linha ou coluna, nesse caso) seja NA. Isso pode ser desativado com a opção skipna:

```
In [234]: df.mean(axis='columns', skipna=False)
Out[234]:
a NaN
b 1.300
c NaN
d -0.275
dtype: float64
```

Veja a Tabela 5.7 que apresenta uma lista de opções comuns para cada método de redução.

Tabela 5.7 – Opções para os métodos de redução

Método	Descrição
axis	Eixo sobre o qual ocorrerá a redução; 0 para as linhas do DataFrame e 1 para as colunas
skipna	Exclui valores ausentes; True por padrão
level	Redução agrupada por nível se o eixo estiver hierarquicamente indexado (MultiIndex)

Alguns métodos como idxmin e idxmax devolvem estatísticas indiretas como o valor do índice em que os valores mínimo e máximo são encontrados:

```
In [235]: df.idxmax()
Out[235]:
one b
two d
dtype: object
```

Outros métodos são de *acúmulo*:

```
In [236]: df.cumsum()
Out[236]:
   one two
a 1.40 NaN
b 8.50 -4.5
c NaN NaN
d 9.25 -5.8
```

Há outro tipo de método que não é nem de redução nem de acúmulo. describe é um exemplo desse tipo, e gera vários dados estatísticos de resumo de uma só vez:

```
In [237]: df.describe()
Out[237]:
          one two
count 3.000000 2.000000
mean 3.083333 -2.900000
std 3.493685 2.262742
min 0.750000 -4.500000
25% 1.075000 -3.700000
50% 1.400000 -2.900000
75% 4.250000 -2.100000
max 7.100000 -1.300000
```

Em dados não numéricos, describe gera estatísticas de resumo alternativas:

```
In [238]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)

In [239]: obj.describe()
Out[239]:
count 16
unique 3
top a
freq 8
dtype: object
```

Veja a Tabela 5.8 que apresenta uma lista completa de estatísticas de resumo e os métodos relacionados.

Tabela 5.8 – Estatísticas descritivas e de resumo

Método	Descrição
count	Número de valores diferentes de NA

Método	Descrição
describe	Calcula o conjunto de dados estatísticos de resumo para Series ou cada coluna de DataFrame
min, max	Calcula os valores mínimo e máximo
argmin, argmax	Calcula as posições dos índices (inteiros) nas quais os valores mínimo ou máximo são obtidos, respectivamente
idxmin, idxmax	Calcula os rótulos dos índices nos quais os valores mínimo ou máximo são obtidos, respectivamente
quantile	Calcula o quantil da amostragem variando de 0 a 1
sum	Soma dos valores
mean	Média dos valores
median	Mediana aritmética (quantil 50%) dos valores
mad	Desvio absoluto médio do valor médio
prod	Produto de todos os valores
var	Variância dos valores da amostra
std	Desvio-padrão dos valores da amostra
skew	Assimetria ou obliquidade (skewness, ou terceiro momento) dos valores da amostra
kurt	Curtose (quarto momento) dos valores da amostra
cumsum	Soma cumulativa dos valores
cummin, cummax	Mínimo ou máximo cumulativo dos valores, respectivamente
cumprod	Produto cumulativo dos valores
diff	Calcula a primeira diferença aritmética (útil para séries temporais)
pct_change	Calcula mudanças percentuais

Correlação e covariância

Algumas estatísticas de resumo, como correlação e covariância, são calculadas a partir de pares de argumentos. Vamos considerar alguns DataFrames de preços e volumes de ações obtidos do Yahoo! Finance usando o pacote add-on pandas-datareader. Caso não o tenha instalado, ele poderá ser obtido usando o conda ou o pip:

```
conda install pandas-datareader
```

Utilizarei o módulo pandas_datareader para fazer download de alguns

dados de algumas listas de ações:

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
            for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}

price = pd.DataFrame({ticker: data['Adj Close']
                      for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume']
                       for ticker, data in all_data.items()})
```

É possível que, quando você estiver lendo este livro, o Yahoo! Finance não exista mais, pois o Yahoo! foi adquirido pela Verizon em 2017. Consulte a documentação online do pandas-datareader para ver as funcionalidades mais recentes.

Vamos agora calcular as mudanças percentuais nos preços – uma operação de série temporal que será mais bem explorada no Capítulo 11:

```
In [242]: returns = price.pct_change()

In [243]: returns.tail()
Out[243]:
                AAPL GOOG IBM MSFT
Date
2016-10-17 -0.000680 0.001837 0.002072 -0.003483
2016-10-18 -0.000681 0.019616 -0.026168 0.007690
2016-10-19 -0.002979 0.007846 0.003583 -0.002255
2016-10-20 -0.000512 -0.005652 0.001719 -0.004867
2016-10-21 -0.003930 0.003011 -0.012474 0.042096
```

O método corr de Series calcula a correlação entre os valores diferentes de NA de duas Series, alinhados pelo índice e que se sobrepõem. De forma relacionada, cov calcula a covariância:

```
In [244]: returns['MSFT'].corr(returns['IBM'])
Out[244]: 0.49976361144151144

In [245]: returns['MSFT'].cov(returns['IBM'])
Out[245]: 8.8706554797035462e-05
```

Como MSFT é um atributo Python válido, podemos também selecionar essas colunas usando uma sintaxe mais concisa:

```
In [246]: returns.MSFT.corr(returns.IBM)
Out[246]: 0.49976361144151144
```

Os métodos corr e cov de DataFrame, por outro lado, devolvem uma matriz completa de correlação ou de covariância como um DataFrame, respectivamente:

```
In [247]: returns.corr()
Out[247]:
          AAPL GOOG IBM MSFT
AAPL 1.000000 0.407919 0.386817 0.389695
GOOG 0.407919 1.000000 0.405099 0.465919
IBM 0.386817 0.405099 1.000000 0.499764
MSFT 0.389695 0.465919 0.499764 1.000000

In [248]: returns.cov()
Out[248]:
          AAPL GOOG IBM MSFT
AAPL 0.000277 0.000107 0.000078 0.000095
GOOG 0.000107 0.000251 0.000078 0.000108
IBM 0.000078 0.000078 0.000146 0.000089
MSFT 0.000095 0.000108 0.000089 0.000215
```

Ao usar o método corrwith de DataFrame, podemos calcular correlações de pares entre as colunas ou linhas de um DataFrame com outra Series ou um DataFrame. Passar uma Series devolve uma Series com o valor das correlações calculado para cada coluna:

```
In [249]: returns.corrwith(returns.IBM)
Out[249]:
AAPL 0.386817
GOOG 0.405099
IBM 1.000000
MSFT 0.499764
dtype: float64
```

Passar um DataFrame calcula as correlações entre os nomes de coluna correspondentes. Nesse caso, calculei as correlações entre

as mudanças percentuais e o volume:

```
In [250]: returns.corrwith(volume)
Out[250]:
AAPL -0.075565
GOOG -0.007067
IBM -0.204849
MSFT -0.092950
dtype: float64
```

Passar axis='columns' faz as operações serem linha a linha. Em todos os casos, os pontos de dados serão alinhados pelo rótulo antes de a correlação ser calculada.

Valores únicos, contadores de valores e pertinência

Outra classe de métodos relacionados extrai informações sobre os valores contidos em uma Series unidimensional. Para ilustrar isso, considere o exemplo a seguir:

```
In [251]: obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

A primeira função é unique, que devolve um array de valores únicos em uma Series:

```
In [252]: uniques = obj.unique()


In [253]: uniques
Out[253]: array(['c', 'a', 'd', 'b'], dtype=object)
```

Os valores únicos não são necessariamente devolvidos de forma ordenada, mas poderão ser ordenados posteriormente se for preciso (uniques.sort()).De modo relacionado, value_counts calcula uma Series contendo as frequências dos valores:

```
In [254]: obj.value_counts()
Out[254]:
c 3
a 3
b 2
d 1
dtype: int64
```

A Series é ordenada pelo valor em ordem decrescente como conveniência. value_counts também está disponível como um método de nível superior do pandas, e pode ser usado com qualquer array ou sequência:

```
In [255]: pd.value_counts(obj.values, sort=False)
Out[255]:
b 2
a 3
c 3
d 1
dtype: int64
```

isin executa uma verificação vetorizada de pertinência a conjuntos, e pode ser útil para filtrar um conjunto de dados e obter um subconjunto de valores em uma Series ou coluna de um DataFrame:

```
In [256]: obj
Out[256]:
0 c
1 a
2 d
3 a
4 a
5 b
6 b
7 c
8 c
dtype: object

In [257]: mask = obj.isin(['b', 'c'])

In [258]: mask
Out[258]:
0 True
1 False
2 False
3 False
4 False
5 True
6 True
```

```
7 True
8 True
dtype: bool

In [259]: obj[mask]
Out[259]:
0 c
5 b
6 b
7 c
8 c
dtype: object
```

Relacionado a isin, temos o método Index.get_indexer, que nos dá um array de índices de um array de valores possivelmente não distintos para outro array de valores distintos:

```
In [260]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])

In [261]: unique_vals = pd.Series(['c', 'b', 'a'])

In [262]: pd.Index(unique_vals).get_indexer(to_match)
Out[262]: array([0, 2, 1, 1, 0, 2])
```

Veja a Tabela 5.9, que contém uma referência a esses métodos.

Tabela 5.9 – Métodos para unicidade, contadores de valores e pertinência a conjuntos

Método	Descrição
Isin	Calcula um array booleano indicando se cada valor de uma Series está contido na sequência de valores recebida
match	Calcula índices inteiros para cada valor de um array em outro array de valores distintos; é útil para alinhamento de dados e operações do tipo junção (join)
unique	Calcula um array de valores únicos em uma Series, devolvido na ordem observada
value_counts	Devolve uma Series contendo valores únicos como seu índice e as frequências como seus valores; a ordem dos contadores é decrescente

Em alguns casos, talvez você queira calcular um histograma de várias colunas relacionadas em um DataFrame. Veja um exemplo:

```
In [263]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
   .....: 'Qu2': [2, 3, 1, 2, 3],
   .....: 'Qu3': [1, 5, 2, 4, 4]})

In [264]: data
Out[264]:
   Qu1 Qu2 Qu3
0 1 2 1
1 3 3 5
2 4 1 2
3 3 2 4
4 4 3 4
```

Passar pandas.value_counts para a função apply desse DataFrame resulta em:

```
In [265]: result = data.apply(pd.value_counts).fillna(0)

In [266]: result
Out[266]:
   Qu1 Qu2 Qu3
1 1.0 1.0 1.0
2 0.0 2.0 1.0
3 2.0 2.0 0.0
4 2.0 0.0 2.0
5 0.0 0.0 1.0
```

Nesse caso, os rótulos das linhas no resultado são os valores distintos que ocorrem em todas as colunas. Os valores são os respectivos contadores desses valores em cada coluna.

5.4 Conclusão

No próximo capítulo, discutiremos as ferramentas para leitura (ou *carga*) e escrita de conjuntos de dados com o pandas. Depois disso, exploraremos mais profundamente as ferramentas para limpeza, tratamento, análise e visualização de dados usando o pandas.

CAPÍTULO 6

Carga de dados, armazenagem e formatos de arquivo

Acessar dados é um primeiro passo necessário para usar a maior parte das ferramentas neste livro. Manterei o foco na entrada e na saída de dados usando o pandas, embora haja várias ferramentas em outras bibliotecas para ajudar na leitura e na escrita de dados em diversos formatos.

A entrada e a saída de dados geralmente se enquadram em algumas categorias principais: leitura de arquivos-texto e outros formatos mais eficientes em disco, carga de dados de bancos de dados e interação com fontes de dados da rede, como APIs web.

6.1 Lendo e escrevendo dados em formato-texto

O pandas tem uma série de funções para ler dados tabulares na forma de um objeto DataFrame. A Tabela 6.1 sintetiza algumas delas, embora read_csv e read_table provavelmente sejam aquelas que você usará com mais frequência.

Tabela 6.1 – Funções de parsing do pandas

Função	Descrição
read_csv	Carrega dados delimitados de um arquivo, um URL ou um objeto do tipo arquivo; utiliza vírgula como delimitador default
read_table	Carrega dados delimitados de um arquivo, um URL ou um objeto do tipo arquivo; utiliza tabulação ('\t') como delimitador default
read_fwf	Lê dados em formato de coluna com tamanho fixo (isto é, sem delimitadores)

Função	Descrição
read_clipboard	Versão de read_table que lê dados da área de transferência (clipboard); é útil para converter tabelas de páginas web
read_excel	Lê dados tabulares de um arquivo Excel XLS ou XLSX
read_hdf	Lê arquivos HDF5 escritos pelo pandas
read_html	Lê todas as tabelas que se encontram no documento HTML especificado
read_json	Lê dados de uma representação em string JSON (JavaScript Object Notation, ou Notação de Objetos JavaScript)
read_msgpack	Lê dados codificados pelo pandas no formato binário MessagePack
read_pickle	Lê um objeto arbitrário armazenado no formato pickle de Python
read_sas	Lê um conjunto de dados SAS armazenado em um dos formatos personalizados do sistema SAS
read_sql	Lê o resultado de uma consulta SQL (usando SQLAlchemy) na forma de um DataFrame do pandas
read_stata	Lê um conjunto de dados no formato de arquivo Stata
read_feather	Lê o formato de arquivo binário Feather

Apresentarei uma visão geral de como atuam essas funções, que foram criadas com o objetivo de converter dados de texto em um DataFrame. Os argumentos opcionais dessas funções podem se enquadrar em algumas categorias:

Indexação

Para tratar uma ou mais colunas como o DataFrame devolvido, e se os nomes das colunas devem ser obtidos do arquivo, do usuário ou de nenhum deles.

Inferência de tipos e conversão de dados

Inclui as conversões de valores definidas pelo usuário e uma lista personalizada de marcadores de valores ausentes.

Parsing de data e hora

Inclui recursos de combinação, entre eles, combinação de informações de data e hora espalhadas em várias colunas em uma única coluna no resultado.

Iteração

Suporte para iteração em partes de arquivos bem grandes.

Problemas com dados sujos

Pular linhas ou um rodapé, comentários ou outras pequenas informações como dados numéricos com vírgulas para separar milhares.

Por causa do modo como os dados podem estar desorganizados no mundo real, algumas das funções de carga de dados (particularmente read_csv) com o passar do tempo se tornaram muito complexas no que concerne às suas opções. É normal se sentir apreensivo por causa da quantidade de parâmetros diferentes (read_csv tem mais de 50 atualmente, quando escrevi este livro). A documentação online do pandas apresenta muitos exemplos de como cada um deles funciona, portanto, se você estiver enfrentando dificuldades para ler um arquivo em particular, deve haver um exemplo suficientemente semelhante para ajudar você a encontrar os parâmetros corretos.

Algumas dessas funções, como pandas.read_csv, fazem *inferência de tipos*, pois os tipos de dados das colunas não fazem parte do formato. Isso significa que você não precisa necessariamente especificar quais colunas são numéricas, inteiras, booleanas ou string. Outros formatos de dados, como HDF5, Feather e msgpack, têm os tipos de dados armazenados no formato.

Lidar com datas e outros tipos personalizados pode exigir esforço extra. Vamos começar com um pequeno arquivo-texto CSV (Comma-Separated Values, ou Valores Separados por Vírgula):

```
In [8]: !cat examples/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

 Nesse caso, usei o comando de shell cat para exibir o conteúdo puro do arquivo na tela. Se você estiver no Windows, poderá utilizar type em vez de cat para conseguir o mesmo efeito.

Considerando que os dados estão delimitados por vírgula, podemos usar read_csv para lê-los em um DataFrame:

```
In [9]: df = pd.read_csv('examples/ex1.csv')

In [10]: df
Out[10]:
   a b c d message
0 1 2 3 4 hello
1 5 6 7 8 world
2 9 10 11 12 foo
```

Também poderíamos ter usado read_table e especificado o delimitador:

```
In [11]: pd.read_table('examples/ex1.csv', sep=',')
Out[11]:
   a b c d message
0 1 2 3 4 hello
1 5 6 7 8 world
2 9 10 11 12 foo
```

Um arquivo nem sempre terá uma linha de cabeçalho. Considere o arquivo a seguir:

```
In [12]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Para ler esse arquivo, temos duas opções. Podemos permitir que o pandas atribua nomes default para as colunas ou podemos, nós mesmos, especificar os nomes:

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
   0 1 2 3 4
0 1 2 3 4 hello
1 5 6 7 8 world
```

```
2 9 10 11 12 foo

In [14]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[14]:
   a b c d message
0 1 2 3 4 hello
1 5 6 7 8 world
2 9 10 11 12 foo
```

Suponha que quiséssemos que a coluna message fosse o índice do DataFrame devolvido. Podemos informar que queremos a coluna no índice 4 ou de nome 'message' utilizando o argumento index_col:

```
In [15]: names = ['a', 'b', 'c', 'd', 'message']

In [16]: pd.read_csv('examples/ex2.csv', names=names, index_col='message')
Out[16]:
         a b c d
message
hello 1 2 3 4
world 5 6 7 8
foo 9 10 11 12
```

Caso você queira compor um índice hierárquico a partir de várias colunas, passe uma lista de números ou de nomes de colunas:

```
In [17]: !cat examples/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [18]: parsed = pd.read_csv('examples/csv_mindex.csv',
   ....: index_col=['key1', 'key2'])

In [19]: parsed
Out[19]:
           value1 value2
```

```
key1 key2
one a 1 2
    b 3 4
    c 5 6
    d 7 8
two a 9 10
    b 11 12
    c 13 14
    d 15 16
```

Em alguns casos, uma tabela pode não ter um delimitador fixo, usando espaço em branco ou outro padrão para separar os campos. Considere um arquivo-texto com a aparência a seguir:

```
In [20]: list(open('examples/ex3.txt'))
Out[20]:
[' A B C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb 0.927272 0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382 1.100491\n']
```

Embora você possa fazer algumas manipulações manualmente, os campos, nesse caso, estão separados por uma quantidade variável de espaços em branco. Em situações como essa, é possível passar uma expressão regular como um delimitador para read_table. Nesse exemplo, podemos usar a expressão regular \s+, de modo que teríamos:

```
In [21]: result = pd.read_table('examples/ex3.txt', sep='\s+')

In [22]: result
Out[22]:
          A B C
aaa -0.264438 -1.026059 -0.619500
bbb 0.927272 0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382 1.100491
```

Como havia um nome de coluna a menos que o número de linhas de dados, read_table infere que a primeira coluna deve ser o índice do DataFrame nesse caso especial.

As funções de parser têm muitos argumentos adicionais para ajudar você a lidar com a grande variedade de possíveis formatos de arquivo excepcionais (veja uma lista parcial na Tabela 6.2). Por exemplo, podemos ignorar a primeira, a terceira e a quarta linhas de um arquivo usando skiprows:

```
In [23]: !cat examples/ex4.csv
# ei!
a,b,c,d,message
# só queria deixar a situação um pouco mais difícil para você
# quem lê arquivos CSV com computadores, de qualquer modo?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
In [24]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[24]:
   a b c d message
0 1 2 3 4 hello
1 5 6 7 8 world
2 9 10 11 12 foo
```

Lidar com valores ausentes é uma parte importante e, em geral, com certas nuances, no processo de parsing do arquivo. Dados ausentes geralmente não estão presentes (são strings vazias) ou estão marcados com algum valor de *sentinela*. Por padrão, o pandas utiliza um conjunto de sentinelas que ocorrem usualmente, como NA e NULL:

```
In [25]: !cat examples/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
In [26]: result = pd.read_csv('examples/ex5.csv')

In [27]: result
Out[27]:
  something a b c d message
0 one 1 2 3.0 4 NaN
1 two 5 6 NaN 8 world
2 three 9 10 11.0 12 foo
```

```
In [28]: pd.isnull(result)
Out[28]:
   something a b c d message
0 False False False False False True
1 False False False True False False
2 False False False False False False
```

A opção na_values pode aceitar uma lista ou um conjunto de strings a serem considerados como valores ausentes:

```
In [29]: result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])

In [30]: result
Out[30]:
  something a b c d message
0 one 1 2 3.0 4 NaN
1 two 5 6 NaN 8 world
2 three 9 10 11.0 12 foo
```

Sentinelas NA diferentes podem ser especificadas para cada coluna em um dicionário:

```
In [31]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

In [32]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
Out[32]:
  something a b c d message
0 one 1 2 3.0 4 NaN
1 NaN 5 6 NaN 8 world
2 three 9 10 11.0 12 NaN
```

A Tabela 6.2 lista algumas das opções usadas com frequência no pandas.read_csv e no pandas.read_table.

Tabela 6.2 – Alguns argumentos das funções read_csv/read_table

Argumento	Descrição
Path	String que indica o local no sistema de arquivos, o URL ou um objeto do tipo arquivo
sep ou delimiter	Sequência de caracteres ou uma expressão regular a ser usada para separar campos em cada linha

Argumento	Descrição
header	Número da linha a ser usada como nomes de coluna; o default é 0 (primeira linha), mas deverá ser None se não houver linha de cabeçalho
index_col	Números ou nomes de coluna a serem usados como índice das linhas no resultado; pode ser um único nome/número ou uma lista deles se for um índice hierárquico
names	Lista de nomes de colunas para o resultado; pode ser combinado com header=None
skiprows	Número de linhas a serem ignoradas no início do arquivo ou lista de números de linhas (começando de 0) a serem ignoradas.
na_values	Sequência de valores a serem substituídos por NA.
comment	Caractere(s) para separar comentários no final das linhas
parse_dates	Tenta fazer parse de dados para datetime; o default é False. Se for True, tentará fazer parse de todas as colunas. Caso contrário, poderá especificar uma lista de números ou nomes de colunas para o parse. Se o elemento da lista for uma tupla ou uma lista, combinará várias colunas e fará o parse para data (por exemplo, se a data/hora estiverem separadas em duas colunas).
keep_date_col	Se as colunas forem reunidas para parse de data, mantém as colunas unidas; o default é False.
converters	Dicionário contendo o número ou o nome de colunas mapeados para funções (por exemplo, {'foo': f} aplica a função f em todos os valores da coluna 'foo').
dayfirst	Ao fazer parsing de datas potencialmente ambíguas, trata-as como estando no formato internacional (por exemplo, 7/6/2012 -> 7 de junho de 2012); o default é False.
date_parser	Função a ser usada para parse de datas.
nrows	Número de linhas a serem lidas no início do arquivo.
iterator	Devolve um objeto TextParser para ler o arquivo aos poucos.
chunksize	Para iteração, é o tamanho das partes do arquivo.
skip_footer	Número de linhas a serem ignoradas no final do arquivo.
verbose	Exibe várias informações de saída do parser, como o número de valores ausentes em colunas não numéricas.
encoding	Codificação de texto para Unicode (por exemplo, 'utf-8' para texto codificado em UTF-8).
squeeze	Se os dados sujeitos a parsing contiverem apenas uma coluna, devolve uma Series.

Argumento	Descrição
thousands	Separador de milhar (por exemplo, ',' ou '.').

Lendo arquivos-texto em partes

Ao processar arquivos bem grandes ou descobrir o conjunto certo de argumentos para processar corretamente um arquivo grande, talvez você queira ler somente uma pequena parte ou iterar por porções menores do arquivo.

Antes de observar um arquivo grande, vamos alterar as configurações do pandas a fim de que a exibição dos dados seja mais compacta:

```
In [33]: pd.options.display.max_rows = 10
```

Agora temos:

```
In [34]: result = pd.read_csv('examples/ex6.csv')

In [35]: result
Out[35]:
        one two three four key
0 0.467976 -0.038649 -0.295344 -1.824726 L
1 -0.358893 1.404453 0.704965 -0.200638 B
2 -0.501840 0.659254 -0.421691 -0.057688 G
3 0.204886 1.074134 1.388361 -0.982404 R
4 0.354628 -0.133116 0.283763 -0.837063 Q
... ... ... ... ... ..
9995 2.311896 -0.417070 -1.409599 -0.515821 L
9996 -0.479893 -0.650419 0.745152 -0.646038 E
9997 0.523331 0.787112 0.486066 1.093156 K
9998 -0.362559 0.598894 -1.843201 0.887292 G
9999 -0.096376 -1.012999 -0.657431 -0.573315 0
[10000 rows x 5 columns]
```

Se você quiser ler apenas uma quantidade pequena de linhas (evitando ler o arquivo todo), especifique isso usando nrows:

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[36]:
      one two three four key
0 0.467976 -0.038649 -0.295344 -1.824726 L
```

```
1 -0.358893 1.404453 0.704965 -0.200638 B
2 -0.501840 0.659254 -0.421691 -0.057688 G
3 0.204886 1.074134 1.388361 -0.982404 R
4 0.354628 -0.133116 0.283763 -0.837063 Q
```

Para ler um arquivo em partes, especifique uma quantidade de linhas para chunksize:

```
In [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

In [38]: chunker
Out[38]: <pandas.io.parsers.TextFileReader at 0x7fb741fe7160>
```

O objeto TextParser devolvido por read_csv permite iterar pelas partes do arquivo de acordo com o chunksize. Por exemplo, podemos iterar por ex6.csv, agregando os contadores de valores na coluna 'key', assim:

```
chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

tot = pd.Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
```

Temos então:

```
In [40]: tot[:10]
Out[40]:
E 368.0
X 364.0
L 346.0
O 343.0
Q 340.0
M 338.0
J 337.0
F 335.0
K 334.0
H 330.0
dtype: float64
```

TextParser também oferece um método get_chunk que permite ler partes de tamanho arbitrário.

Escrevendo dados em formato-texto

Os dados também podem ser exportados para um formato com delimitador. Vamos considerar um dos arquivos CSV que lemos antes:

```
In [41]: data = pd.read_csv('examples/ex5.csv')

In [42]: data
Out[42]:
  something a b c d message
0 one 1 2 3.0 4 NaN
1 two 5 6 NaN 8 world
2 three 9 10 11.0 12 foo
```

Usando o método to_csv de DataFrame, podemos escrever os dados separados por vírgula em um arquivo:

```
In [43]: data.to_csv('examples/out.csv')

In [44]: !cat examples/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

Outros delimitadores podem ser usados, é claro (escreveremos em sys.stdout para que o texto resultante seja exibido no console):

```
In [45]: import sys

In [46]: data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

Valores ausentes aparecem como strings vazias na saída. Você pode representá-las com outro valor de sentinela:

```
In [47]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
```

```
2,three,9,10,11.0,12,foo
```

Sem outras opções especificadas, os rótulos tanto das linhas quanto das colunas são escritos. Ambos podem ser desativados:

```
In [48]: data.to_csv(sys.stdout, index=False, header=False)
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

Também podemos escrever somente um subconjunto das colunas, em uma ordem de sua preferência:

```
In [49]: data.to_csv(sys.stdout, index=False, columns=['a', 'b', 'c'])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Uma Series também tem um método to_csv:

```
In [50]: dates = pd.date_range('1/1/2000', periods=7)

In [51]: ts = pd.Series(np.arange(7), index=dates)

In [52]: ts.to_csv('examples/tseries.csv')

In [53]: !cat examples/tseries.csv
2000-01-01,0
2000-01-02,1
2000-01-03,2
2000-01-04,3
2000-01-05,4
2000-01-06,5
2000-01-07,6
```

Trabalhando com formatos delimitados

É possível carregar a maioria dos formatos de dados tabulares de disco usando funções como pandas.read_table. Em alguns casos, porém, um pouco de processamento manual talvez seja necessário. Não é incomum receber um arquivo com uma ou mais linhas malformadas que poderão confundir read_table. Para demonstrar o

uso das ferramentas básicas, considere um pequeno arquivo CSV:

```
In [54]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3"
```

Para qualquer arquivo com um único caractere como delimitador, podemos usar o módulo embutido `csv` de Python. Para usá-lo, passe qualquer arquivo aberto ou um objeto do tipo arquivo para `csv.reader`:

```
import csv
f = open('examples/ex7.csv')

reader = csv.reader(f)
```

Iterar pelo reader como um arquivo produz tuplas de valores com qualquer caractere de aspas removido:

```
In [56]: for line in reader:
   ....: print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

A partir daí, cabe a você fazer a manipulação necessária para deixar os dados no formato em que precisar. Vamos executar essa tarefa passo a passo. Inicialmente vamos ler o arquivo em uma lista de linhas:

```
In [57]: with open('examples/ex7.csv') as f:
   ....: lines = list(csv.reader(f))
```

Em seguida, separamos as linhas em linha de cabeçalho e linhas de dados:

```
In [58]: header, values = lines[0], lines[1:]
```

Então podemos criar um dicionário de colunas de dados usando uma dictionary comprehension (abrangência de dicionário) e a expressão `zip(*values)`, que faz a transposição das linhas para as colunas:

```
In [59]: data_dict = {h: v for h, v in zip(header, zip(*values))}
```

```
In [60]: data_dict
Out[60]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

Os arquivos CSV têm muitas variantes distintas. Para definir um novo formato com um delimitador, uma convenção de aspas para strings ou um finalizador de linha diferentes, definimos uma subclasse simples de csv.Dialect:

```
class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimiter = ';'
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL
reader = csv.reader(f, dialect=my_dialect)
```

Também podemos especificar parâmetros individuais de dialetos de CSV como parâmetros nomeados para csv.reader, sem a necessidade de definir uma subclasse:

```
reader = csv.reader(f, delimiter='|')
```

As possíveis opções (atributos de csv.Dialect) e o que elas fazem podem ser encontrados na Tabela 6.3.

Tabela 6.3 – Opções de dialetos de CSV

Argumento	Descrição
delimiter	String de um caractere para separar os campos; o default é ','.
lineterminator	Finalizador de linha para escrita; o default é '\r\n'. O reader ignora isso e reconhece finalizadores de linha em plataformas diferentes.
quotechar	Caractere de aspas para campos com caracteres especiais (como um delimitador); o default é '"'.
quoting	Convenção para aspas. As opções incluem csv.QUOTE_ALL (aspas em todos os campos), csv.QUOTE_MINIMAL (somente campos com caracteres especiais, como o delimitador), csv.QUOTE_NONNUMERIC e csv.QUOTE_NONE (sem aspas). Consulte a documentação de Python para ver todos os detalhes. O default é QUOTE_MINIMAL.
skipinitialspace	Ignora espaços em branco depois de cada delimitador; o default é False.
doublequote	Como lidar com o caractere de aspas em um campo; se for True, é duplo (consulte a documentação online para ver todos os

	detalhes e o comportamento).
escapechar	String para escapar o delimitador se quoting estiver definido com csv.QUOTE_NONE; o default é desativado.

 Para arquivos com delimitadores contendo vários caracteres fixos ou mais complexos, não será possível utilizar o módulo csv. Nesses casos, será necessário fazer a separação de linhas e outras tarefas de limpeza usando o método split de strings ou o método de expressão regular re.split.

Para *escrever* arquivos com delimitadores manualmente, podemos usar csv.writer. Ele aceita um objeto de arquivo aberto, com permissão para escrita, além das mesmas opções de dialeto e de formato de csv.reader:

```
with open('mydata.csv', 'w') as f:
    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(('one', 'two', 'three'))
    writer.writerow(('1', '2', '3'))
    writer.writerow(('4', '5', '6'))
    writer.writerow(('7', '8', '9'))
```

Dados JSON

O JSON (abreviatura de JavaScript Object Notation, ou Notação de Objetos JavaScript) tornou-se um dos formatos padrões para envio de dados em requisições HTTP entre navegadores web e outras aplicações. É um formato de dados muito mais livre que um formato de texto tabular como o CSV. Eis um exemplo:

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]},
              {"name": "Katie", "age": 38,
               "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
```

O JSON é quase um código Python válido, com a exceção de seu

valor nulo null e outras nuances (como não permitir vírgulas no final das listas). Os tipos básicos são objetos (dicionários), arrays (listas), strings, números, booleanos e nulls. Todas as chaves em um objeto devem ser strings. Há várias bibliotecas Python para ler e escrever dados JSON. Usarei json nesta seção, pois ela está incluída na biblioteca-padrão de Python. Para converter uma string JSON em formato Python, utilize json.loads:

```
In [62]: import json

In [63]: result = json.loads(obj)

In [64]: result
Out[64]:
{'name': 'Wes',
 'pet': None,
 'places_lived': ['United States', 'Spain', 'Germany'],
 'siblings': [{'age': 30, 'name': 'Scott', 'pets': ['Zeus', 'Zuko']},
  {'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Stache', 'Cisco']}]}
```

json.dumps, por outro lado, converte um objeto Python de volta para JSON:

```
In [65]: asjson = json.dumps(result)
```

O modo como você converte um objeto JSON ou uma lista de objetos em um DataFrame ou em outra estrutura de dados para análise é uma tarefa que caberá a você. De forma conveniente, é possível passar uma lista de dicionários (que eram anteriormente objetos JSON) para o construtor de DataFrame e selecionar um subconjunto dos campos de dados:

```
In [66]: siblings = pd.DataFrame(result['siblings'], columns=['name', 'age'])

In [67]: siblings
Out[67]:
    name age
0 Scott 30
1 Katie 38
```

pandas.read_json pode converter automaticamente conjuntos de dados JSON organizados de modo específico em uma Series ou um

DataFrame. Por exemplo:

```
In [68]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

As opções default para pandas.read_json supõem que cada objeto no array JSON seja uma linha da tabela:

```
In [69]: data = pd.read_json('examples/example.json')

In [70]: data
Out[70]:
   a b c
0 1 2 3
1 4 5 6
2 7 8 9
```

Para um exemplo mais amplo de leitura e manipulação de dados JSON (incluindo registros aninhados), veja o exemplo do USDA Food Database no Capítulo 7.

Se precisar exportar dados do pandas para JSON, uma maneira é usar os métodos to_json em Series e em DataFrame:

```
In [71]: print(data.to_json())
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}

In [72]: print(data.to_json(orient='records'))
[{"a":1,"b":2, "c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]
```

XML e HTML: web scraping

Python tem muitas bibliotecas para ler e escrever dados nos formatos HTML e XML presentes em todos os lugares. Exemplos incluem lxml (*http://lxml.de/*), Beautiful Soup e html5lib. Embora o lxml, em geral, seja comparativamente muito mais rápido, as outras bibliotecas podem lidar melhor com arquivos HTML ou XML malformados.

O pandas tem uma função embutida read_html que utiliza bibliotecas como lxml e Beautiful Soup para fazer parse automaticamente de

tabelas de arquivos HTML em objetos DataFrame. Para mostrar como isso funciona, fiz o download de um arquivo HTML (usado na documentação do pandas) da agência governamental FDIC dos Estados Unidos, que mostra falências de bancos.[1] Inicialmente você deve instalar algumas bibliotecas adicionais usadas por read_html:

```
conda install lxml
pip install beautifulsoup4 html5lib
```

Se você não estiver usando o conda, pip install lxml provavelmente funcionará também.

A função pandas.read_html tem uma série de opções, mas, por padrão, ela procura e tenta fazer parse de todos os dados tabulares contidos em tags <table>. O resultado é uma lista de objetos DataFrame:

```
In [73]: tables = pd.read_html('examples/fdic_failed_bank_list.html')

In [74]: len(tables)
Out[74]: 1

In [75]: failures = tables[0]

In [76]: failures.head()
Out[76]:
                    Bank Name City ST CERT \
0 Allied Bank Mulberry AR 91
1 The Woodbury Banking Company Woodbury GA 11297
2 First CornerStone Bank King of Prussia PA 35312
3 Trust Company Bank Memphis TN 9956
4 North Milwaukee State Bank Milwaukee WI 20364
                Acquiring Institution Closing Date Updated Date
0 Today's Bank September 23, 2016 November 17, 2016
1 United Bank August 19, 2016 November 17, 2016
2 First-Citizens Bank & Trust Company May 6, 2016 September 6, 2016
3 The Bank of Fayette County April 29, 2016 September 6, 2016
4 First-Citizens Bank & Trust Company March 11, 2016 June 16, 2016
```

Como failures tem muitas colunas, o pandas insere um caractere de quebra de linha \.

Como você verá em capítulos mais adiante, a partir daqui

poderíamos prosseguir fazendo uma limpeza e a análise de dados, por exemplo, calculando o número de falências bancárias por ano:

```
In [77]: close_timestamps = pd.to_datetime(failures['Closing Date'])

In [78]: close_timestamps.dt.year.value_counts()
Out[78]:
2010 157
2009 140
2011 92
2012 51
2008 25
      ...
2004 4
2001 4
2007 3
2003 3
2000 2
Name: Closing Date, Length: 15, dtype: int64
```

Fazendo parse de XML com lxml.objectify

O XML (eXtensible Markup Language, ou Linguagem de Marcação Extensível) é outro formato comum de dados estruturados que aceita dados hierárquicos e aninhados com metadados. O livro que você está lendo no momento na verdade foi criado a partir de uma série de documentos XML longos.

Anteriormente mostrei a função pandas.read_html, que utiliza o lxml ou o Beautiful Soup internamente para fazer parse de dados de HTML. XML e HTML são estruturalmente semelhantes, porém o XML é mais genérico. Mostrarei aqui um exemplo de como usar o lxml para fazer parse de dados a partir de um formato XML mais genérico.

O MTA (Metropolitan Transportation Authority) de Nova York publica uma série de dados sobre seus serviços de ônibus e de trem (*http://www.mta.info/developers/download.html*). Nesse exemplo, observaremos os dados de desempenho, que estão contidos em um conjunto de arquivos XML. Cada serviço de trem ou de ônibus tem um arquivo diferente (como *Performance_MNR.xml* para a Metro-North

Railroad) contendo dados mensais na forma de uma série de registros XML com o seguinte aspecto:

```
<INDICATOR>
  <INDICATOR_SEQ>373889</INDICATOR_SEQ>
  <PARENT_SEQ></PARENT_SEQ>
  <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
  <INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
  <DESCRIPTION>Percent of the time that escalators are operational
  systemwide. The availability rate is based on physical observations performed
  the morning of regular business days only. This is a new indicator the agency
  began reporting in 2009.</DESCRIPTION>
  <PERIOD_YEAR>2011</PERIOD_YEAR>
  <PERIOD_MONTH>12</PERIOD_MONTH>
  <CATEGORY>Service Indicators</CATEGORY>
  <FREQUENCY>M</FREQUENCY>
  <DESIRED_CHANGE>U</DESIRED_CHANGE>
  <INDICATOR_UNIT>%</INDICATOR_UNIT>
  <DECIMAL_PLACES>1</DECIMAL_PLACES>
  <YTD_TARGET>97.00</YTD_TARGET>
  <YTD_ACTUAL></YTD_ACTUAL>
  <MONTHLY_TARGET>97.00</MONTHLY_TARGET>
  <MONTHLY_ACTUAL></MONTHLY_ACTUAL>
</INDICATOR>
```

Usando lxml.objectify, fazemos o parse do arquivo e obtemos uma referência para o nó raiz do arquivo XML com getroot:

```
from lxml import objectify

path = 'datasets/mta_perf/Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
```

root.INDICATOR devolve um gerador que produz cada elemento XML <INDICATOR>. Para cada registro, podemos preencher um dicionário de nomes de tags (como YTD_ACTUAL) para valores de dados (excluindo algumas tags):

```
data = []

skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
               'DESIRED_CHANGE', 'DECIMAL_PLACES']
```

```
for elt in root.INDICATOR:
    el_data = {}
    for child in elt.getchildren():
        if child.tag in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)
```

Por fim, convertemos essa lista de dicionários em um DataFrame:

```
In [81]: perf = pd.DataFrame(data)

In [82]: perf.head()
Out[82]:
          AGENCY_NAME CATEGORY \
0 Metro-North Railroad Service Indicators
1 Metro-North Railroad Service Indicators
2 Metro-North Railroad Service Indicators
3 Metro-North Railroad Service Indicators
4 Metro-North Railroad Service Indicators
                                DESCRIPTION FREQUENCY \
0 Percent of commuter trains that arrive at thei... M
1 Percent of commuter trains that arrive at thei... M
2 Percent of commuter trains that arrive at thei... M
3 Percent of commuter trains that arrive at thei... M
4 Percent of commuter trains that arrive at thei... M
                    INDICATOR_NAME INDICATOR_UNIT MONTHLY_ACTUAL \
0 On-Time Performance (West of Hudson) % 96.9
1 On-Time Performance (West of Hudson) % 95
2 On-Time Performance (West of Hudson) % 96.9
3 On-Time Performance (West of Hudson) % 98.3
4 On-Time Performance (West of Hudson) % 95.8
  MONTHLY_TARGET PERIOD_MONTH PERIOD_YEAR YTD_ACTUAL
YTD_TARGET
0 95 1 2008 96.9 95
1 95 2 2008 96 95
2 95 3 2008 96.3 95
3 95 4 2008 96.8 95
4 95 5 2008 96.6 95
```

Dados XML podem se tornar muito mais complexos que os dados

desse exemplo. Cada tag pode ter metadados também. Considere uma tag de link HTML, que também é um XML válido:

```
from io import StringIO
tag = '<a href="http://www.google.com">Google</a>'
root = objectify.parse(StringIO(tag)).getroot()
```

Agora podemos ter acesso a qualquer um dos campos (como href) da tag ou do texto de link:

```
In [84]: root
Out[84]: <Element a at 0x7fb73962a088>

In [85]: root.get('href')
Out[85]: 'http://www.google.com'

In [86]: root.text
Out[86]: 'Google'
```

6.2 Formatos de dados binários

Uma das formas mais simples de armazenar dados (também conhecida como *serialização*) de modo eficiente em formato binário é usando a serialização embutida pickle de Python. Todos os objetos do pandas têm um método to_pickle que escreve os dados em disco em formato pickle:

```
In [87]: frame = pd.read_csv('examples/ex1.csv')

In [88]: frame
Out[88]:
   a b c d message
0 1 2 3 4 hello
1 5 6 7 8 world
2 9 10 11 12 foo

In [89]: frame.to_pickle('examples/frame_pickle')
```

Podemos ler qualquer objeto armazenado em um arquivo em formato “pickle” utilizando diretamente a função embutida pickle ou, de modo mais conveniente ainda, usando pandas.read_pickle:

```
In [90]: pd.read_pickle('examples/frame_pickle')
Out[90]:
   a b c d message
0 1 2 3 4 hello
1 5 6 7 8 world
2 9 10 11 12 foo
```

pickle é recomendado apenas como um formato de armazenagem de curta duração. O problema está no fato de ser difícil garantir que o formato permanecerá estável com o passar do tempo; um objeto serializado com pickle hoje poderá não ser desserializado com uma versão mais recente de uma biblioteca. Tentamos manter a compatibilidade com versões anteriores quando possível, mas, em algum momento no futuro, talvez vá ser necessário "romper" com o formato pickle.

O pandas tem suporte incluído para dois outros formatos de dados binários: HDF5 e MessagePack. Darei alguns exemplos de HDF5 na próxima seção, mas incentivo você a explorar diferentes formatos de arquivo para ver quão rápidos eles são e se funcionarão bem para suas análises. Outros formatos de armazenagem para dados do pandas ou do NumPy incluem:

bcolz (http://bcolz.blosc.org)

Um formato binário passível de compactação, orientado a colunas, baseado na biblioteca de compactação Blosc.

Feather (http://github.com/wesm/feather)

Um formato de arquivo para várias linguagens, orientado a colunas, que projetei com Hadley Wickham (*http://hadley.nz/*) da comunidade de programação R. O Feather utiliza o formato de memória em colunas do Apache Arrow (*http://arrow.apache.org/*).

Usando o formato HDF5

O HDF5 é um formato de arquivo bem-visto, cujo propósito é armazenar grandes quantidades de dados científicos em arrays. Está disponível na forma de uma biblioteca C, e tem interfaces disponíveis em várias outras linguagens, incluindo Java, Julia,

MATLAB e Python. O “HDF” em HDF5 quer dizer *Hierarchical Data Format* (Formato de Dados Hierárquico). Cada arquivo HDF5 é capaz de armazenar vários conjuntos de dados e pode aceitar metadados. Se comparado com formatos mais simples, o HDF5 aceita compactação durante a execução, com uma variedade de modos de compactação, permitindo que dados com padrões repetidos sejam armazenados de modo mais eficiente. O HDF5 pode ser uma boa opção para trabalhar com conjuntos bem grandes de dados que não caibam na memória, pois você poderá ler e escrever pequenas seções de arrays muito maiores, de modo eficaz.

Embora seja possível acessar diretamente arquivos HDF5 usando as bibliotecas PyTables ou h5py, o pandas oferece uma interface de alto nível que simplifica o armazenamento de objetos Series e DataFrame. A classe HDFStore funciona como um dicionário e cuida dos detalhes de baixo nível:

```
In [92]: frame = pd.DataFrame({'a': np.random.randn(100)})

In [93]: store = pd.HDFStore('mydata.h5')

In [94]: store['obj1'] = frame

In [95]: store['obj1_col'] = frame['a']

In [96]: store
Out[96]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
/obj1 frame (shape->[100,1])
/obj1_col series (shape->[100])
/obj2 frame_table (typ->appendable,nrows->100,ncols->1,indexers->
[index])
/obj3 frame_table (typ->appendable,nrows->100,ncols->1,indexers->
[index])
```

Objetos contidos no arquivo HDF5 podem então ser recuperados com a mesma API do tipo dicionário:

```
In [97]: store['obj1']
```

```
Out[97]:
          a
0 -0.204708
1 0.478943
2 -0.519439
3 -0.555730
4 1.965781
.. ...
95 0.795253
96 0.118110
97 -0.748532
98 0.584970
99 0.152677
[100 rows x 1 columns]
```

HDFStore aceita dois esquemas de armazenagem: 'fixed' e 'table'. O último em geral é mais lento, porém aceita operações de consulta usando uma sintaxe especial:

```
In [98]: store.put('obj2', frame, format='table')

In [99]: store.select('obj2', where=['index >= 10 and index <= 15'])
Out[99]:
          a
10 1.007189
11 -1.296221
12 0.274992
13 0.228913
14 1.352917
15 0.886429

In [100]: store.close()
```

O put é uma versão explícita do método store['obj2'] = frame, mas permite definir outras opções como o formato de armazenagem.

A função pandas.read_hdf oferece um atalho para essas ferramentas:

```
In [101]: frame.to_hdf('mydata.h5', 'obj3', format='table')

In [102]: pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])
Out[102]:
          a
```

```
0 -0.204708
1 0.478943
2 -0.519439
3 -0.555730
4 1.965781
```

 Se você estiver processando dados armazenados em servidores remotos, como Amazon S3 ou HDFS, usar um formato binário diferente projetado para armazenagem distribuída, como o Apache Parquet (*http://parquet.apache.org/*), talvez seja mais apropriado. Python para Parquet e outros formatos de armazenagem desse tipo continuam em desenvolvimento, portanto não escreverei sobre eles neste livro.

Se você trabalha com grandes quantidades de dados localmente, incentivaria você a explorar o PyTables e o h5py para ver como eles podem atender às suas necessidades. Como muitos problemas de análise de dados são limitados por E/S (são I/O-bound em vez de CPU-bound), usar uma ferramenta como o HDF5 pode agilizar significativamente as suas aplicações.

 O HDF5 *não* é um banco de dados. É mais apropriado para conjuntos de dados em que há uma escrita e várias leituras (write-once, read-many). Embora dados possam ser adicionados em um arquivo a qualquer momento, se vários writers fizerem isso simultaneamente, o arquivo poderá ser corrompido.

Lendo arquivos do Microsoft Excel

O pandas também oferece suporte para ler dados tabulares armazenados em arquivos do Excel 2003 (e versões mais recentes) usando a classe `ExcelFile` ou a função `pandas.read_excel`. Internamente essas ferramentas utilizam os pacotes add-on `xlrd` e `openpyxl` para ler arquivos XLS e XLSX, respectivamente. Talvez você precise instalá-los manualmente usando o pip ou o conda.

Para usar `ExcelFile`, crie uma instância passando um path para um arquivo `xls` ou `xlsx`:

```
In [104]: xlsx = pd.ExcelFile('examples/ex1.xlsx')
```

Dados armazenados em uma planilha poderão então ser lidos em um DataFrame utilizando parse:

```
In [105]: pd.read_excel(xlsx, 'Sheet1')
Out[105]:
   a b c d message
0 1 2 3 4 hello
1 5 6 7 8 world
2 9 10 11 12 foo
```

Se você estiver lendo várias planilhas de um arquivo, será mais rápido criar o ExcelFile, mas também pode simplesmente passar o nome do arquivo para pandas.read_excel:

```
In [106]: frame = pd.read_excel('examples/ex1.xlsx', 'Sheet1')

In [107]: frame
Out[107]:
   a b c d message
0 1 2 3 4 hello
1 5 6 7 8 world
2 9 10 11 12 foo
```

Para escrever dados do pandas em formato Excel, você deve inicialmente criar um ExcelWriter e então escrever os dados aí usando o método to_excel de objetos do pandas:

```
In [108]: writer = pd.ExcelWriter('examples/ex2.xlsx')

In [109]: frame.to_excel(writer, 'Sheet1')

In [110]: writer.save()
```

Um path de arquivo também pode ser passado para to_excel, evitando o ExcelWriter:

```
In [111]: frame.to_excel('examples/ex2.xlsx')
```

6.3 Interagindo com APIs web

Muitos sites têm APIs públicas que oferecem feeds de dados usando JSON ou outro formato. Há várias maneiras de acessar

essas APIs a partir de Python; um método fácil de usar, que recomendo, é lançar mão do pacote requests (*http://docs.python-requests.org*).

Para encontrar os últimos 30 problemas do pandas no GitHub, podemos fazer uma requisição HTTP GET usando a biblioteca add-on requests:

```
In [113]: import requests

In [114]: url = 'https://api.github.com/repos/pandas-dev/pandas/issues'

In [115]: resp = requests.get(url)

In [116]: resp
Out[116]: <Response [200]>
```

O método json do objeto Response devolverá um dicionário contendo o parse dos dados JSON em objetos Python nativos:

```
In [117]: data = resp.json()

In [118]: data[0]['title']
Out[118]: 'BUG: rank with +-inf, #6945'
```

Cada elemento em data é um dicionário contendo todos os dados encontrados em uma página de problemas do GitHub (exceto os comentários). Podemos passar data diretamente para DataFrame e extrair os campos de interesse:

```
In [119]: issues = pd.DataFrame(data, columns=['number', 'title',
   .....: 'labels', 'state'])

In [120]: issues
Out[120]:
    number title \
0 17903 BUG: rank with +-inf, #6945
1 17902 Revert "ERR: Raise ValueError when setting sca...
2 17901 Wrong orientation of operations between DataFr...
3 17900 added 'infer' option to compression in _get_ha...
4 17898 Last day of month should group with that month
.. ... ...
25 17854 Adding an integer-location based "get" method
```

```
26 17853 BUG: adds validation for boolean keywords in D...
27 17851 BUG: duplicate indexing with embedded non-orde...
28 17850 ImportError: No module named 'pandas.plotting'
29 17846 BUG: Ignore division by 0 when merging empty d...
                                                labels state
0 [] open
1 [{'id': 35818298, 'url': 'https://api.github.c... open
2 [] open
3 [] open
4 [{'id': 76811, 'url': 'https://api.github.com/... open
.. ... ...
25 [{'id': 35818298, 'url': 'https://api.github.c... open
26 [{'id': 42670965, 'url': 'https://api.github.c... open
27 [{'id': 76811, 'url': 'https://api.github.com/... open
28 [{'id': 31932467, 'url': 'https://api.github.c... open
29 [{'id': 76865106, 'url': 'https://api.github.c... open
[30 rows x 4 columns]
```

Com um pouco de jogo de cintura, podemos criar algumas interfaces de nível mais alto para APIs web comuns que devolvam objetos DataFrame a fim de facilitar a análise.

6.4 Interagindo com bancos de dados

Em um ambiente de negócios, a maior parte dos dados talvez não esteja armazenada em arquivos-texto nem em arquivos Excel. Bancos de dados relacionais baseados em SQL (como SQL Server, PostgreSQL e MySQL) são amplamente usados, e muitos bancos de dados alternativos têm se tornado bem populares. A escolha do banco de dados em geral depende das necessidades de desempenho, de integridade dos dados e de escalabilidade de uma aplicação.

Carregar dados de SQL para um DataFrame é razoavelmente simples, e o pandas tem algumas funções para simplificar o processo. Como exemplo, criarei um banco de dados SQLite usando o driver embutido sqlite3 de Python:

```
In [121]: import sqlite3
```

```
In [122]: query = """
   .....: CREATE TABLE test
   .....: (a VARCHAR(20), b VARCHAR(20),
   .....: c REAL, d INTEGER
   .....: );"""

In [123]: con = sqlite3.connect('mydata.sqlite')

In [124]: con.execute(query)
Out[124]: <sqlite3.Cursor at 0x7fb7361b4b90>

In [125]: con.commit()
```

Em seguida, inserimos algumas linhas de dados:

```
In [126]: data = [('Atlanta', 'Georgia', 1.25, 6),
   .....: ('Tallahassee', 'Florida', 2.6, 3),
   .....: ('Sacramento', 'California', 1.7, 5)]

In [127]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

In [128]: con.executemany(stmt, data)
Out[128]: <sqlite3.Cursor at 0x7fb7396d25e0>

In [129]: con.commit()
```

A maioria dos drivers de SQL de Python (PyODBC, psycopg2, MySQLdb, pymssql etc.) devolve uma lista de tuplas ao selecionar dados de uma tabela:

```
In [130]: cursor = con.execute('select * from test')

In [131]: rows = cursor.fetchall()

In [132]: rows
Out[132]:
[('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', 'California', 1.7, 5)]
```

Podemos passar a lista de tuplas para o construtor de DataFrame, mas também precisaremos dos nomes das colunas, contidos no atributo description do cursor:

```
In [133]: cursor.description
Out[133]:
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [134]: pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
Out[134]:
          a b c d
0 Atlanta Georgia 1.25 6
1 Tallahassee Florida 2.60 3
2 Sacramento California 1.70 5
```

Há uma boa dose de manipulação que seria melhor não repetir sempre que você consultar o banco de dados. O projeto SQLAlchemy (*http://www.sqlalchemy.org/*) é um kit de ferramentas SQL popular para Python, que abstrai muitas das diferenças comuns entre os bancos de dados SQL. O pandas tem uma função read_sql que permite ler dados facilmente de uma conexão SQLAlchemy genérica. Neste exemplo, faremos a conexão com o mesmo banco de dados SQLite usando o SQLAlchemy e leremos dados da tabela criada anteriormente:

```
In [135]: import sqlalchemy as sqla

In [136]: db = sqla.create_engine('sqlite:///mydata.sqlite')

In [137]: pd.read_sql('select * from test', db)
Out[137]:
          a b c d
0 Atlanta Georgia 1.25 6
1 Tallahassee Florida 2.60 3
2 Sacramento California 1.70 5
```

6.5 Conclusão

Ter acesso aos dados geralmente é o primeiro passo no processo de análise de dados. Neste capítulo, vimos uma série de ferramentas úteis que deverão ajudar você a começar o seu

trabalho. Nos próximos capítulos, exploraremos com mais detalhes o tratamento e a visualização dos dados, a análise de séries temporais e outros assuntos.

---

1 Para ver a lista completa, acesse *https://www.fdic.gov/bank/individual/failed/banklist.html*.

CAPÍTULO 7

Limpeza e preparação dos dados

Durante a análise e a modelagem dos dados, um período significativo de tempo é gasto em sua preparação: carga, limpeza, transformação e reorganização. Sabe-se que essas tarefas em geral ocupam 80% ou mais do tempo de um analista. Às vezes, o modo como os dados são armazenados em arquivos ou em bancos de dados não constituem o formato correto para uma tarefa em particular. Muitos pesquisadores preferem fazer um processamento *ad hoc* dos dados de um formato para outro usando uma linguagem de programação de propósito geral como Python, Perl, R ou Java, ou ferramentas de processamento de texto do Unix como sed ou awk. Felizmente o pandas, junto com os recursos embutidos da linguagem Python, oferecem um conjunto de ferramentas de alto nível, rápido e flexível, para permitir que você manipule os dados, deixando-os no formato correto.

Se você identificar um tipo de manipulação de dados que não esteja em nenhum lugar neste livro nem em outro local da biblioteca pandas, sinta-se à vontade para compartilhar o seu caso de uso em uma das listas de discussão de Python ou no site do pandas no GitHub. Na verdade, boa parte do design e da implementação do pandas tem sido direcionada com base nas necessidades das aplicações do mundo real.

Neste capítulo, discutirei as ferramentas para dados ausentes, dados duplicados, manipulação de strings e outras transformações de dados para análise. No próximo capítulo, manterei o foco nas

várias maneiras de combinar e reorganizar os conjuntos de dados.

7.1 Tratando dados ausentes

Dados ausentes são comuns em muitas aplicações de análise de dados. Um dos objetivos do pandas é deixar o trabalho com dados ausentes o menos problemático possível. Por exemplo, todas as estatísticas descritivas em objetos do pandas, por padrão, excluem dados ausentes.

A forma como os dados ausentes são representados em objetos do pandas, de certo modo, não é perfeita, porém é funcional para muitos usuários. Para dados numéricos, o pandas utiliza o valor de ponto flutuante NaN (Not a Number) para representá-los. Esse valor é chamado de *valor de sentinela*, e pode ser facilmente detectado:

```
In [10]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [11]: string_data
Out[11]:
0 aardvark
1 artichoke
2 NaN
3 avocado
dtype: object

In [12]: string_data.isnull()
Out[12]:
0 False
1 False
2 True
3 False
dtype: bool
```

No pandas, adotamos uma convenção usada na linguagem de programação R, referenciando os dados ausentes como NA, que quer dizer *Not Available* (indisponível). Em aplicações estatísticas, dados NA podem ser dados inexistentes ou dados que existem, porém não foram observados (por causa de problemas com a coleta

de dados, por exemplo). Ao limpar os dados para análise, em geral é importante fazer a análise nos próprios dados ausentes a fim de identificar problemas em sua coleta ou possíveis distorções provocadas por dados ausentes.

O valor embutido None de Python também é tratado como NA em arrays de objetos:

```
In [13]: string_data[0] = None

In [14]: string_data.isnull()
Out[14]:
0 True
1 False
2 True
3 False
dtype: bool
```

Há um trabalho em andamento no projeto pandas cujo objetivo é melhorar os detalhes internos do tratamento de dados ausentes, mas as funções de API dos usuários, como pandas.isnull, abstraem muitos dos detalhes incômodos. Veja a Tabela 7.1 que apresenta uma lista de algumas funções relacionadas ao tratamento de dados ausentes.

Tabela 7.1 – Métodos para tratamento de NA

Argumento	Descrição
dropna	Filtra rótulos de eixos, baseado no fato de os valores para cada rótulo terem dados ausentes, com limites variados para a quantidade de dados ausentes a ser tolerada.
fillna	Preenche os dados ausentes com algum valor ou utilizando um método de interpolação como 'ffill' ou 'bfill'.
isnull	Devolve valores booleanos informando quais valores estão ausentes/são NA.
notnull	Negação de isnull.

Filtrando dados ausentes

Há algumas maneiras de filtrar dados ausentes. Embora sempre

haja a opção de fazer isso manualmente usando pandas.isnull e uma indexação booleana, o método dropna pode ser útil. Em uma Series, ele devolve a Series somente com os dados diferentes de null e os valores dos índices:

```
In [15]: from numpy import nan as NA

In [16]: data = pd.Series([1, NA, 3.5, NA, 7])

In [17]: data.dropna()
Out[17]:
0 1.0
2 3.5
4 7.0
dtype: float64
```

Essa instrução é equivalente a:

```
In [18]: data[data.notnull()]
Out[18]:
0 1.0
2 3.5
4 7.0
dtype: float64
```

Com objetos DataFrame, a situação é um pouco mais complexa. Talvez você queira descartar linhas ou colunas que contenham somente NA ou apenas aquelas que contenham algum NA. Por padrão, dropna descarta qualquer linha contendo um valor ausente:

```
In [19]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
   ....: [NA, NA, NA], [NA, 6.5, 3.]])

In [20]: cleaned = data.dropna()

In [21]: data
Out[21]:
     0 1 2
0 1.0 6.5 3.0
1 1.0 NaN NaN
2 NaN NaN NaN
3 NaN 6.5 3.0
```

```
In [22]: cleaned
Out[22]:
    0 1 2
0 1.0 6.5 3.0
```

Passar how='all' descartará apenas as colunas que contenham somente NAs:

```
In [23]: data.dropna(how='all')
Out[23]:
    0 1 2
0 1.0 6.5 3.0
1 1.0 NaN NaN
3 NaN 6.5 3.0
```

Para descartar colunas do mesmo modo, passe axis=1:

```
In [24]: data[4] = NA

In [25]: data
Out[25]:
    0 1 2 4
0 1.0 6.5 3.0 NaN
1 1.0 NaN NaN NaN
2 NaN NaN NaN NaN
3 NaN 6.5 3.0 NaN

In [26]: data.dropna(axis=1, how='all')
Out[26]:
    0 1 2
0 1.0 6.5 3.0
1 1.0 NaN NaN
2 NaN NaN NaN
3 NaN 6.5 3.0
```

Um modo relacionado de filtrar linhas de DataFrame diz respeito a dados de séries temporais. Suponha que queremos manter somente as linhas contendo determinado número de observações. Podemos representar isso com o argumento thresh:

```
In [27]: df = pd.DataFrame(np.random.randn(7, 3))

In [28]: df.iloc[:4, 1] = NA
```

```
In [29]: df.iloc[:2, 2] = NA

In [30]: df
Out[30]:
          0 1 2
0 -0.204708 NaN NaN
1 -0.555730 NaN NaN
2 0.092908 NaN 0.769023
3 1.246435 NaN -1.296221
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741

In [31]: df.dropna()
Out[31]:
          0 1 2
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741

In [32]: df.dropna(thresh=2)
Out[32]:
          0 1 2
2 0.092908 NaN 0.769023
3 1.246435 NaN -1.296221
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741
```

Preenchendo dados ausentes

Em vez de filtrar dados ausentes (e possivelmente descartar outros dados junto com esses), você poderá preencher as “lacunas” de várias maneiras. Na maioria dos casos, o método fillna será a função que representa a força de trabalho a ser utilizada. Chamar fillna com uma constante substitui valores ausentes por esse valor:

```
In [33]: df.fillna(0)
Out[33]:
          0 1 2
```

```
0 -0.204708 0.000000 0.000000
1 -0.555730 0.000000 0.000000
2 0.092908 0.000000 0.769023
3 1.246435 0.000000 -1.296221
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741
```

Ao chamar fillna com um dicionário, podemos usar um valor de preenchimento diferente para cada coluna:

```
In [34]: df.fillna({1: 0.5, 2: 0})
Out[34]:
          0 1 2
0 -0.204708 0.500000 0.000000
1 -0.555730 0.500000 0.000000
2 0.092908 0.500000 0.769023
3 1.246435 0.500000 -1.296221
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741
```

fillna devolve um novo objeto, mas o objeto existente pode ser alterado in-place:

```
In [35]: _ = df.fillna(0, inplace=True)

In [36]: df
Out[36]:
          0 1 2
0 -0.204708 0.000000 0.000000
1 -0.555730 0.000000 0.000000
2 0.092908 0.000000 0.769023
3 1.246435 0.000000 -1.296221
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741
```

Os mesmos métodos de interpolação disponíveis para reindexação podem ser usados com fillna:

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA
```

```
In [39]: df.iloc[4:, 2] = NA

In [40]: df
Out[40]:
          0 1 2
0 0.476985 3.248944 -1.021228
1 -0.577087 0.124121 0.302614
2 0.523772 NaN 1.343810
3 -0.713544 NaN -2.370232
4 -1.860761 NaN NaN
5 -1.265934 NaN NaN

In [41]: df.fillna(method='ffill')
Out[41]:
          0 1 2
0 0.476985 3.248944 -1.021228
1 -0.577087 0.124121 0.302614
2 0.523772 0.124121 1.343810
3 -0.713544 0.124121 -2.370232
4 -1.860761 0.124121 -2.370232
5 -1.265934 0.124121 -2.370232

In [42]: df.fillna(method='ffill', limit=2)
Out[42]:
          0 1 2
0 0.476985 3.248944 -1.021228
1 -0.577087 0.124121 0.302614
2 0.523772 0.124121 1.343810
3 -0.713544 0.124121 -2.370232
4 -1.860761 NaN -2.370232
5 -1.265934 NaN -2.370232
```

Com fillna, podemos executar várias outras tarefas com um pouco de criatividade. Por exemplo, podemos passar o valor da média ou da mediana de uma Series:

```
In [43]: data = pd.Series([1., NA, 3.5, NA, 7])

In [44]: data.fillna(data.mean())
Out[44]:
0 1.000000
```

```
1 3.833333
2 3.500000
3 3.833333
4 7.000000
dtype: float64
```

Veja a Tabela 7.2 que contém uma referência para fillna.

Tabela 7.2 – Argumentos da função fillna

Argumento	Descrição
value	Valor escalar ou um objeto do tipo dicionário a ser usado para preencher valores ausentes
method	Interpolação; por padrão, será 'ffill' se a função for chamada sem outros argumentos
axis	Eixo a ser preenchido; o default é axis=0
inplace	Modifica o objeto que faz a chamada, sem gerar uma cópia
limit	Para preenchimento para a frente (forward) e para trás (backward), é o número máximo de valores consecutivos a serem preenchidos

7.2 Transformação de dados

Até agora neste capítulo, estivemos preocupados com a reorganização dos dados. Filtragem, limpeza e outras transformações constituem outra classe de operações importantes.

Removendo duplicatas

Linhas duplicadas podem ser encontradas em um DataFrame por diversos motivos. Eis um exemplo:

```
In [45]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
   ....: 'k2': [1, 1, 2, 3, 3, 4, 4]})

In [46]: data
Out[46]:
    k1 k2
0 one 1
1 two 1
2 one 2
3 two 3
```

```
4 one 3
5 two 4
6 two 4
```

O método duplicated de DataFrame devolve uma Series booleana informando se cada linha é uma duplicata (foi observada em uma linha anterior) ou não:

```
In [47]: data.duplicated()
Out[47]:
0 False
1 False
2 False
3 False
4 False
5 False
6 True
dtype: bool
```

Relacionado a esse caso, temos drop_duplicates, que devolve um DataFrame com dados em que o array duplicated é False:

```
In [48]: data.drop_duplicates()
Out[48]:
   k1 k2
0 one 1
1 two 1
2 one 2
3 two 3
4 one 3
5 two 4
```

Por padrão, esses dois métodos consideram todas as colunas; de forma alternativa, podemos especificar qualquer subconjunto delas na detecção de duplicatas. Suponha que tivéssemos uma coluna adicional de valores e quiséssemos filtrar as duplicatas somente com base na coluna 'k1':

```
In [49]: data['v1'] = range(7)

In [50]: data.drop_duplicates(['k1'])
Out[50]:
   k1 k2 v1
```

```
0 one 1 0
1 two 1 1
```

duplicated e drop_duplicates, por padrão, mantêm a primeira combinação de valores observados. Passar keep='last' devolverá a última:

```
In [51]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[51]:
   k1 k2 v1
0 one 1 0
1 two 1 1
2 one 2 2
3 two 3 3
4 one 3 4
6 two 4 6
```

Transformando dados usando uma função ou um mapeamento

Para muitos conjuntos de dados, talvez você queira fazer algumas transformações com base nos valores de um array, uma Series ou uma coluna de um DataFrame. Considere os seguintes dados hipotéticos coletados acerca de vários tipos de carnes:

```
In [52]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
   ....: 'Pastrami', 'corned beef', 'Bacon',
   ....: 'pastrami', 'honey ham', 'nova lox'],
   ....: 'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

In [53]: data
Out[53]:
         food ounces
0 bacon 4.0
1 pulled pork 3.0
2 bacon 12.0
3 Pastrami 6.0
4 corned beef 7.5
5 Bacon 8.0
6 pastrami 3.0
7 honey ham 5.0
```

```
8 nova lox 6.0
```

Suponha que quiséssemos adicionar uma coluna informando o tipo do animal do qual cada alimento é proveniente. Vamos criar um mapeamento entre cada tipo distinto de carne e o tipo do animal:

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

O método map em uma Series aceita uma função ou um objeto do tipo dicionário contendo um mapeamento; nesse caso, porém, temos um pequeno problema, pois algumas das carnes utilizam letra inicial maiúscula enquanto outras não. Desse modo, precisamos converter todos os valores para letras minúsculas utilizando o método str.lower de Series:

```
In [55]: lowercased = data['food'].str.lower()

In [56]: lowercased
Out[56]:
0 bacon
1 pulled pork
2 bacon
3 pastrami
4 corned beef
5 bacon
6 pastrami
7 honey ham
8 nova lox
Name: food, dtype: object

In [57]: data['animal'] = lowercased.map(meat_to_animal)

In [58]: data
Out[58]:
        food ounces animal
```

```
0 bacon 4.0 pig
1 pulled pork 3.0 pig
2 bacon 12.0 pig
3 Pastrami 6.0 cow
4 corned beef 7.5 cow
5 Bacon 8.0 pig
6 pastrami 3.0 cow
7 honey ham 5.0 pig
8 nova lox 6.0 salmon
```

Poderíamos também ter passado uma função que fizesse todo o trabalho:

```
In [59]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[59]:
0 pig
1 pig
2 pig
3 cow
4 cow
5 pig
6 cow
7 pig
8 salmon
Name: food, dtype: object
```

Usar map é uma forma conveniente de fazer transformações em todos os elementos e executar outras operações relacionadas à limpeza de dados.

Substituindo valores

Preencher dados ausentes com o método fillna é um caso especial da substituição mais genérica de valores. Conforme já vimos, map pode ser usado para modificar um subconjunto de valores em um objeto, porém replace oferece uma forma mais simples e mais flexível de fazer isso. Considere a Series a seguir:

```
In [60]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [61]: data
Out[61]:
```

```
0 1.0
1 -999.0
2 2.0
3 -999.0
4 -1000.0
5 3.0
dtype: float64
```

Os valores -999 podem ser valores de sentinela para dados ausentes. Para substituí-los por valores NA, compreensíveis pelo pandas, podemos usar replace, gerando uma nova Series (a menos que você passe inplace=True):

```
In [62]: data.replace(-999, np.nan)
Out[62]:
0 1.0
1 NaN
2 2.0
3 NaN
4 -1000.0
5 3.0
dtype: float64
```

Se você quiser substituir diversos valores de uma só vez, passe uma lista e, em seguida, o valor para substituição:

```
In [63]: data.replace([-999, -1000], np.nan)
Out[63]:
0 1.0
1 NaN
2 2.0
3 NaN
4 NaN
5 3.0
dtype: float64
```

Para usar um substituto diferente para cada valor, passe uma lista deles:

```
In [64]: data.replace([-999, -1000], [np.nan, 0])
Out[64]:
0 1.0
1 NaN
```

```
2 2.0
3 NaN
4 0.0
5 3.0
dtype: float64
```

O argumento especificado também pode ser um dicionário:

```
In [65]: data.replace({-999: np.nan, -1000: 0})
Out[65]:
0 1.0
1 NaN
2 2.0
3 NaN
4 0.0
5 3.0
dtype: float64
```

O método data.replace é diferente de data.str.replace, que faz uma substituição de string em todos os elementos. Veremos esses métodos de string em Series mais adiante neste capítulo.

Renomeando os índices dos eixos

Assim como os valores em uma Series, os rótulos dos eixos podem ser transformados de modo semelhante por uma função ou alguma forma de mapeamento, a fim de gerar objetos novos com rótulos diferentes. Também podemos modificar os eixos in-place, sem criar uma nova estrutura de dados. Eis um exemplo simples:

```
In [66]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
   ....: index=['Ohio', 'Colorado', 'New York'],
   ....: columns=['one', 'two', 'three', 'four'])
```

Como em uma Series, os índices de um eixo têm um método map:

```
In [67]: transform = lambda x: x[:4].upper()


In [68]: data.index.map(transform)
Out[68]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

Podemos fazer uma atribuição para index, modificando o DataFrame

in-place:

```
In [69]: data.index = data.index.map(transform)

In [70]: data
Out[70]:
      one two three four
OHIO 0 1 2 3
COLO 4 5 6 7
NEW 8 9 10 11
```

Se você quiser criar uma versão transformada de um conjunto de dados sem modificar os dados originais, um método útil será rename:

```
In [71]: data.rename(index=str.title, columns=str.upper)
Out[71]:
      ONE TWO THREE FOUR
Ohio 0 1 2 3
Colo 4 5 6 7
New 8 9 10 11
```

Observe que rename pode ser usado em conjunto com um objeto do tipo dicionário fornecendo novos valores para um subconjunto dos rótulos de um eixo:

```
In [72]: data.rename(index={'OHIO': 'INDIANA'},
   ....: columns={'three': 'peekaboo'})
Out[72]:
         one two peekaboo four
INDIANA 0 1 2 3
COLO 4 5 6 7
NEW 8 9 10 11
```

rename evita que você tenha o trabalho de copiar o DataFrame manualmente e definir seus atributos index e columns. Caso você queira modificar um conjunto de dados in-place, passe inplace=True:

```
In [73]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

In [74]: data
Out[74]:
         one two three four
INDIANA 0 1 2 3
COLO 4 5 6 7
```

NEW 8 9 10 11

Discretização e compartimentalização (binning)

Dados contínuos com frequência são discretizados ou, de modo alternativo, separados em “compartimentos” (bins) para análise. Suponha que tenhamos dados sobre um grupo de pessoas em um estudo e queremos agrupá-las em conjuntos de idades discretas:

```
In [75]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Vamos dividir esses dados em compartimentos de 18 a 25, 26 a 35, 36 a 60 e, por fim, 61 anos ou mais. Para isso, utilize cut, uma função do pandas:

```
In [76]: bins = [18, 25, 35, 60, 100]

In [77]: cats = pd.cut(ages, bins)

In [78]: cats
Out[78]:
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35,
 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

O objeto devolvido pelo pandas é um objeto Categorical especial. A saída que vemos descreve os compartimentos calculados pelo pandas.cut. Podemos tratá-la como um array de strings informando o nome do compartimento; internamente, ela contém um array categories que especifica os nomes distintos das categorias, junto com rótulos para os dados de ages no atributo codes:

```
In [79]: cats.codes
Out[79]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [80]: cats.categories
Out[80]:
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]]
              closed='right',
              dtype='interval[int64]')
```

```
In [81]: pd.value_counts(cats)
Out[81]:
(18, 25] 5
(35, 60] 3
(25, 35] 3
(60, 100] 1
dtype: int64
```

Observe que pd.value_counts(cats) são os contadores de compartimentos para o resultado de pandas.cut.

De forma consistente com a notação matemática para intervalos, um parêntese significa que o lado está *aberto*, enquanto o colchete indica que está *fechado* (é inclusivo). Podemos alterar o lado que está fechado passando right=False:

```
In [82]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
Out[82]:
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36,
 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

Também podemos especificar nossos próprios nomes de compartimentos passando uma lista ou um array para a opção labels:

```
In [83]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']


In [84]: pd.cut(ages, bins, labels=group_names)
Out[84]:
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged,
Mid
dleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

Se passarmos um número inteiro de compartimentos para cut, em vez de passar fronteiras explícitas, ele calculará compartimentos de tamanhos iguais com base nos valores mínimo e máximo dos dados. Considere o caso de alguns dados uniformemente distribuídos, divididos em quartos:

```
In [85]: data = np.random.rand(20)
```

```
In [86]: pd.cut(data, 4, precision=2)
Out[86]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55], ..., (0.34
, 0.55], (0.34, 0.55], (0.55, 0.76], (0.34, 0.55], (0.12, 0.34]]
Length: 20
Categories (4, interval[float64]): [(0.12, 0.34] < (0.34, 0.55] < (0.55, 0.76] <
(0.76, 0.97]]
```

A opção precision=2 limita a precisão decimal em dois dígitos.

Uma função intimamente relacionada, qcut, compartimenta os dados com base nos quantis da amostra. Conforme a distribuição dos dados, usar cut em geral não resultará em cada compartimento com o mesmo número de pontos de dados. Como qcut utiliza quantis da amostra, por definição, você obterá compartimentos grosseiramente de mesmo tamanho:

```
In [87]: data = np.random.randn(1000) # Normalmente distribuídos

In [88]: cats = pd.qcut(data, 4) # Separa em quantis

In [89]: cats
Out[89]:
[(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0.62, 3.928], (-0.0265, 0.62]
, ..., (-0.68, -0.0265], (-0.68, -0.0265], (-2.95, -0.68], (0.62, 3.928], (-0.68,
-0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.68] < (-0.68, -0.0265] < (-0.0265,
 0.62] <
                                (0.62, 3.928]]

In [90]: pd.value_counts(cats)
Out[90]:
(0.62, 3.928] 250
(-0.0265, 0.62] 250
(-0.68, -0.0265] 250
(-2.95, -0.68] 250
dtype: int64
```

De modo semelhante a cut, podemos passar nossos próprios quantis (números entre 0 e 1, inclusive):

```
In [91]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[91]:
[(-0.0265, 1.286], (-0.0265, 1.286], (-1.187, -0.0265], (-0.0265, 1.286], (-0.026
5, 1.286], ..., (-1.187, -0.0265], (-1.187, -0.0265], (-2.95, -1.187], (-0.0265,
1.286], (-1.187, -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -1.187] < (-1.187, -0.0265] < (-0.026
5, 1.286] <
                                (1.286, 3.928]]
```

Voltaremos a discutir cut e qcut mais adiante neste capítulo, durante a nossa discussão sobre agregação e operações de grupo, pois essas funções de discretização são particularmente úteis para análise de quantis e de grupos.

Detectando e filtrando valores discrepantes

Filtrar ou transformar valores discrepantes (outliers) é, em boa medida, uma questão de aplicar operações de array. Considere um DataFrame com alguns dados normalmente distribuídos:

```
In [92]: data = pd.DataFrame(np.random.randn(1000, 4))

In [93]: data.describe()
Out[93]:
                0 1 2 3
count 1000.000000 1000.000000 1000.000000 1000.000000
mean 0.049091 0.026112 -0.002544 -0.051827
std 0.996947 1.007458 0.995232 0.998311
min -3.645860 -3.184377 -3.745356 -3.428254
25% -0.599807 -0.612162 -0.687373 -0.747478
50% 0.047101 -0.013609 -0.022158 -0.088274
75% 0.756646 0.695298 0.699046 0.623331
max 2.653656 3.525865 2.735527 3.366626
```

Suponha que quiséssemos encontrar os valores que excedessem 3 em valor absoluto em uma das colunas:

```
In [94]: col = data[2]

In [95]: col[np.abs(col) > 3]
Out[95]:
```

```
41 -3.399312
136 -3.745356
Name: 2, dtype: float64
```

Para selecionar todas as linhas que tenham um valor que exceda 3 ou –3, podemos utilizar o método any em um DataFrame booleano:

```
In [96]: data[(np.abs(data) > 3).any(1)]
Out[96]:
          0 1 2 3
41 0.457246 -0.025907 -3.399312 -0.974657
60 1.951312 3.260383 0.963301 1.201206
136 0.508391 -0.196713 -3.745356 -1.520113
235 -0.242459 -3.056990 1.918403 -0.578828
258 0.682841 0.326045 0.425384 -3.428254
322 1.179227 -3.184377 1.369891 -1.074833
544 -3.548824 1.553205 -2.186301 1.277104
635 -0.578093 0.193299 1.397822 3.366626
782 -0.207434 3.525865 0.283070 0.544635
803 -3.645860 0.255475 -0.549574 -1.907459
```

Valores podem ser definidos com base nesses critérios. Eis um código para eliminar os valores que estejam fora do intervalo de –3 a 3:

```
In [97]: data[np.abs(data) > 3] = np.sign(data) * 3

In [98]: data.describe()
Out[98]:
              0 1 2 3
count 1000.000000 1000.000000 1000.000000 1000.000000
mean 0.050286 0.025567 -0.001399 -0.051765
std 0.992920 1.004214 0.991414 0.995761
min -3.000000 -3.000000 -3.000000 -3.000000
25% -0.599807 -0.612162 -0.687373 -0.747478
50% 0.047101 -0.013609 -0.022158 -0.088274
75% 0.756646 0.695298 0.699046 0.623331
max 2.653656 3.000000 2.735527 3.000000
```

A instrução np.sign(data) gera valores 1 e –1 com base no fato de os valores em data serem positivos ou negativos:

```
In [99]: np.sign(data).head()
Out[99]:
```

```
    0 1 2 3
0 -1.0 1.0 -1.0 1.0
1 1.0 -1.0 1.0 -1.0
2 1.0 1.0 1.0 -1.0
3 -1.0 -1.0 1.0 -1.0
4 -1.0 1.0 -1.0 -1.0
```

Permutação e amostragem aleatória

Permutar (reordenar aleatoriamente) uma Series ou as linhas de um DataFrame é fácil utilizando a função numpy.random.permutation. Chamar permutation com o tamanho do eixo que você quer permutar gera um array de inteiros informando a nova ordem:

```
In [100]: df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))

In [101]: sampler = np.random.permutation(5)

In [102]: sampler
Out[102]: array([3, 1, 4, 2, 0])
```

Esse array pode então ser usado na indexação baseada em iloc ou na função take equivalente:

```
In [103]: df
Out[103]:
    0 1 2 3
0 0 1 2 3
1 4 5 6 7
2 8 9 10 11
3 12 13 14 15
4 16 17 18 19

In [104]: df.take(sampler)
Out[104]:
    0 1 2 3
3 12 13 14 15
1 4 5 6 7
4 16 17 18 19
2 8 9 10 11
0 0 1 2 3
```

Para selecionar um subconjunto aleatório sem substituição, o método sample pode ser usado em Series e em DataFrame:

```
In [105]: df.sample(n=3)
Out[105]:
   0 1 2 3
3 12 13 14 15
4 16 17 18 19
2 8 9 10 11
```

Para gerar uma amostra *com* substituição (a fim de permitir opções repetidas), passe replace=True para sample:

```
In [106]: choices = pd.Series([5, 7, -1, 6, 4])

In [107]: draws = choices.sample(n=10, replace=True)

In [108]: draws
Out[108]:
4 4
1 7
4 4
2 -1
0 5
3 6
1 7
4 4
0 5
4 4
dtype: int64
```

Calculando variáveis indicadoras/dummy

Outro tipo de transformação para modelagem estatística ou aplicações de aprendizado de máquina (machine learning) consiste em converter uma variável de categorias em uma matriz “dummy” ou “indicadora”. Se uma coluna em um DataFrame tiver k valores distintos, poderíamos derivar uma matriz ou um DataFrame com k colunas contendo somente 1s e 0s. O pandas tem uma função get_dummies para isso, embora criar uma função por conta própria não seria difícil. Vamos retomar um exemplo anterior com

DataFrame:

```
In [109]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   .....: 'data1': range(6)})

In [110]: pd.get_dummies(df['key'])
Out[110]:
   a b c
0 0 1 0
1 0 1 0
2 1 0 0
3 0 0 1
4 1 0 0
5 0 1 0
```

Em alguns casos, talvez você queira adicionar um prefixo às colunas no DataFrame indicador, que poderá então ser mesclado com os outros dados. get_dummies tem um argumento de prefixo para isso:

```
In [111]: dummies = pd.get_dummies(df['key'], prefix='key')

In [112]: df_with_dummy = df[['data1']].join(dummies)

In [113]: df_with_dummy
Out[113]:
   data1 key_a key_b key_c
0 0 0 1 0
1 1 0 1 0
2 2 1 0 0
3 3 0 0 1
4 4 1 0 0
5 5 0 1 0
```

Se uma linha de um DataFrame pertencer a várias categorias, a situação se torna um pouco mais complicada. Vamos observar o conjunto de dados de MovieLens 1M, que será investigado com mais detalhes no Capítulo 14:

```
In [114]: mnames = ['movie_id', 'title', 'genres']

In [115]: movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
```

```
   .....: header=None, names=mnames)

In [116]: movies[:10]
Out[116]:
   movie_id title genres
0 1 Toy Story (1995) Animation|Children's|Comedy
1 2 Jumanji (1995) Adventure|Children's|Fantasy
2 3 Grumpier Old Men (1995) Comedy|Romance
3 4 Waiting to Exhale (1995) Comedy|Drama
4 5 Father of the Bride Part II (1995) Comedy
5 6 Heat (1995) Action|Crime|Thriller
6 7 Sabrina (1995) Comedy|Romance
7 8 Tom and Huck (1995) Adventure|Children's
8 9 Sudden Death (1995) Action
9 10 GoldenEye (1995) Action|Adventure|Thriller
```

Adicionar variáveis indicadoras para cada gênero exige um pouco de manipulação nos dados. Inicialmente extraímos a lista de gêneros únicos do conjunto de dados:

```
In [117]: all_genres = []

In [118]: for x in movies.genres:
   .....: all_genres.extend(x.split('|'))

In [119]: genres = pd.unique(all_genres)
```

Agora temos:

```
In [120]: genres
Out[120]:
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
       'Western'], dtype=object)
```

Uma maneira de construir o DataFrame indicador é começar com um DataFrame contendo apenas zeros:

```
In [121]: zero_matrix = np.zeros((len(movies), len(genres)))

In [122]: dummies = pd.DataFrame(zero_matrix, columns=genres)
```

Agora itere pelos filmes e defina entradas para cada linha de

dummies com 1. Para isso, utilize dummies.columns a fim de calcular os índices das colunas para cada gênero:

```
In [123]: gen = movies.genres[0]

In [124]: gen.split('|')
Out[124]: ['Animation', "Children's", 'Comedy']

In [125]: dummies.columns.get_indexer(gen.split('|'))
Out[125]: array([0, 1, 2])
```

Então podemos usar .iloc para definir valores com base nesses índices:

```
In [126]: for i, gen in enumerate(movies.genres):
   .....: indices = dummies.columns.get_indexer(gen.split('|'))
   .....: dummies.iloc[i, indices] = 1
   .....:
```

Em seguida, como fizemos antes, podemos combinar esses dados com movies:

```
In [127]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [128]: movies_windic.iloc[0]
Out[128]:
movie_id 1
title Toy Story (1995)
genres Animation|Children's|Comedy
Genre_Animation 1
Genre_Children's 1
Genre_Comedy 1
Genre_Adventure 0
Genre_Fantasy 0
Genre_Romance 0
Genre_Drama 0
                    ...
Genre_Crime 0
Genre_Thriller 0
Genre_Horror 0
Genre_Sci-Fi 0
Genre_Documentary 0
Genre_War 0
```

```
Genre_Musical 0
Genre_Mystery 0
Genre_Film-Noir 0
Genre_Western 0
Name: 0, Length: 21, dtype: object
```

Para dados bem maiores, esse método de construir variáveis indicadoras com vários membros não é particularmente ágil. Seria melhor escrever uma função de nível mais baixo que escrevesse diretamente em um array NumPy e então encapsular o resultado em um DataFrame.

Uma receita útil para aplicações estatísticas é combinar get_dummies com uma função de discretização como cut:

```
In [129]: np.random.seed(12345)

In [130]: values = np.random.rand(10)

In [131]: values
Out[131]:
array([ 0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645,
        0.6532, 0.7489, 0.6536])

In [132]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [133]: pd.get_dummies(pd.cut(values, bins))
Out[133]:
   (0.0, 0.2] (0.2, 0.4] (0.4, 0.6] (0.6, 0.8] (0.8, 1.0]
0 0 0 0 0 1
1 0 1 0 0 0
2 1 0 0 0 0
3 0 1 0 0 0
4 0 0 1 0 0
5 0 0 1 0 0
6 0 0 0 0 1
7 0 0 0 1 0
8 0 0 0 1 0
9 0 0 0 1 0
```

Definimos a semente (seed) de aleatoriedade com numpy.random.seed para deixar o exemplo determinístico. Veremos pandas.get_dummies

novamente, mais adiante no livro.

7.3 Manipulação de strings

Python tem sido uma linguagem popular para manipulação de dados brutos há muito tempo, em parte por causa de sua facilidade de uso para processamento de strings e de texto. A maior parte das operações em texto são simplificadas com os métodos embutidos do objeto string. Para uma correspondência de padrões e manipulações de texto mais complexas, o uso de expressões regulares talvez seja necessário. O pandas complementa a mistura permitindo aplicar strings e expressões regulares de forma concisa em arrays inteiros de dados, além de lidar com o transtorno dos dados ausentes.

Métodos de objetos string

Para muitas aplicações de manipulação de strings e de scripting, os métodos embutidos de string são suficientes. Como exemplo, uma string separada por vírgulas pode ser dividida em partes usando split:

```
In [134]: val = 'a,b, guido'

In [135]: val.split(',')
Out[135]: ['a', 'b', ' guido']
```

Com frequência, split é usado em conjunto com strip para remover espaços em branco (incluindo quebras de linha):

```
In [136]: pieces = [x.strip() for x in val.split(',')]

In [137]: pieces
Out[137]: ['a', 'b', 'guido']
```

Essas substrings poderiam ser concatenadas com um delimitador de dois-pontos duplos usando a adição:

```
In [138]: first, second, third = pieces

In [139]: first + '::' + second + '::' + third
```

```
Out[139]: 'a::b::guido'
```

No entanto, esse não é um método genérico prático. Uma forma mais rápida e pythônica consiste em passar uma lista ou uma tupla para o método join na string '::':

```
In [140]: '::'.join(pieces)
Out[140]: 'a::b::guido'
```

Outros métodos dizem respeito à localização de substrings. Usar a palavra reservada in de Python é a melhor maneira de detectar uma substring, embora index e find também possam ser utilizados:

```
In [141]: 'guido' in val
Out[141]: True

In [142]: val.index(',')
Out[142]: 1

In [143]: val.find(':')
Out[143]: -1
```

Observe que a diferença entre find e index é que index lança uma exceção caso a string não seja encontrada (*versus* devolver –1):

```
In [144]: val.index(':')
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)
<ipython-input-144-2c016e7367ac> in <module>()
----> 1 val.index(':')
ValueError: substring not found
```

Relacionado a esse caso, temos count, que devolve o número de ocorrências de uma substring em particular:

```
In [145]: val.count(',')
Out[145]: 2
```

replace substituirá as ocorrências de um padrão por outro. É comumente utilizado também para apagar padrões, passando uma string vazia:

```
In [146]: val.replace(',', '::')
Out[146]: 'a::b:: guido'
```

```
In [147]: val.replace(',', '')
Out[147]: 'ab guido'
```

Veja a Tabela 7.3 que contém uma lista de alguns métodos de string de Python.

Expressões regulares também podem ser usadas com muitas dessas operações, conforme veremos.

Tabela 7.3 – Métodos embutidos de string em Python

Argumento	Descrição
count	Devolve o número de ocorrências de uma substring na string, sem sobreposição.
endswith	Devolve True se a string terminar com o sufixo.
startswith	Devolve True se a string começar com o prefixo.
join	Utiliza a string como delimitadora para concatenar uma sequência de outras strings.
index	Devolve a posição do primeiro caractere de uma substring, se ela for encontrada em uma string: gera ValueError se não encontrar.
find	Devolve a posição do primeiro caractere da *primeira* ocorrência da substring na string; é como index, porém devolve –1 se não encontrar.
rfind	Devolve a posição do primeiro caractere da *última* ocorrência da substring na string; devolve –1 se não encontrar.
replace	Substitui ocorrências de uma string por outra string.
strip, rstrip, lstrip	Remove espaços em branco, incluindo quebras de linha; é equivalente a x.strip() (e a rstrip e lstrip, respectivamente) para cada elemento.
split	Separa a string em uma lista de substrings usando o delimitador especificado.
lower	Converte os caracteres alfabéticos para letras minúsculas.
upper	Converte os caracteres alfabéticos para letras maiúsculas.
casefold	Converte os caracteres para letras minúsculas e converte quaisquer combinações variáveis de caracteres específicos de região para um formato comum comparável.
ljust, rjust	Justifica à esquerda ou à direita, respectivamente; preenche o lado oposto da string com espaços (ou com outro caractere de preenchimento) para devolver uma string com um tamanho mínimo.

Expressões regulares

As *expressões regulares* oferecem uma maneira flexível para fazer pesquisas ou correspondências (em geral, mais complexas) de padrões de string em um texto. Uma única expressão, em geral chamada de *regex*, é uma string composta de acordo com a linguagem da expressão regular. O módulo embutido re de Python é responsável pela aplicação de expressões regulares em strings; apresentarei uma série de exemplos de seu uso nesta seção.

 A arte de escrever expressões regulares poderia ter um capítulo próprio e, desse modo, está fora do escopo do livro. Há muitos tutoriais e referências excelentes disponíveis na internet e em outros livros.

As funções do módulo re se enquadram em três categorias: correspondência, substituição e separação de padrões. Naturalmente, elas estão relacionadas; uma regex descreve um padrão a ser localizado no texto, que pode então ser usado para vários propósitos. Vamos analisar um exemplo simples: suponha que quiséssemos separar uma string com um número variável de caracteres de espaços em branco (tabulações, espaços e quebras de linha). A regex que descreve um ou mais caracteres para espaços em branco é \s+:

```
In [148]: import re

In [149]: text = "foo bar\t baz \tqux"

In [150]: re.split('\s+', text)
Out[150]: ['foo', 'bar', 'baz', 'qux']
```

Quando chamamos re.split('\s+', text), a expressão regular inicialmente é *compilada*; então seu método split é chamado no texto que lhe é passado. Podemos compilar a regex por conta própria com re.compile, criando um objeto regex reutilizável:

```
In [151]: regex = re.compile('\s+')

In [152]: regex.split(text)
```

```
Out[152]: ['foo', 'bar', 'baz', 'qux']
```

Se, em vez disso, quiséssemos obter uma lista de todos os padrões que correspondam à regex, o método findall poderia ser utilizado:

```
In [153]: regex.findall(text)
Out[153]: [' ', '\t ', ' \t']
```

 Para evitar um escaping indesejado com \ em uma expressão regular, utilize literais de string *puros* como r'C:\x' no lugar do 'C:\\x' equivalente.

Criar um objeto regex com re.compile é altamente recomendado caso você pretenda aplicar a mesma expressão a várias strings; fazer isso lhe economizará ciclos de CPU.

match e search estão intimamente relacionados a findall. Enquanto findall devolve todas as correspondências em uma string, search devolve apenas a primeira. De modo mais rigoroso, match faz a correspondência *somente* no início da string. Como um exemplo menos trivial, vamos considerar um bloco de texto e uma expressão regular capaz de identificar a maioria dos endereços de email:

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

# re.IGNORECASE faz com que a regex não diferencie letras minúsculas de
maiúsculas
regex = re.compile(pattern, flags=re.IGNORECASE)
```

Usar findall no texto gera uma lista de endereços de email:

```
In [155]: regex.findall(text)
Out[155]:
['dave@google.com',
 'steve@gmail.com',
 'rob@gmail.com',
 'ryan@yahoo.com']
```

search devolve um objeto especial de correspondência para o

primeiro endereço de email no texto. Para a regex anterior, o objeto de correspondência pode nos dizer apenas a posição de início e de fim do padrão na string:

```
In [156]: m = regex.search(text)

In [157]: m
Out[157]: <_sre.SRE_Match object; span=(5, 20), match='dave@google.com'>

In [158]: text[m.start():m.end()]
Out[158]: 'dave@google.com'
```

regex.match devolve None, pois fará a correspondência somente se o padrão ocorrer no início da string:

```
In [159]: print(regex.match(text))
None
```

Relacionado a esse caso, temos sub, que devolverá uma nova string com as ocorrências do padrão substituídas por uma nova string:

```
In [160]: print(regex.sub('REDACTED', text))
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

Suponha que quiséssemos encontrar os endereços de email e, simultaneamente, segmentar cada endereço em seus três componentes: nome do usuário, nome do domínio e sufixo do domínio. Para isso, coloque parênteses em torno das partes do padrão a fim de segmentá-lo:

```
In [161]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [162]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

Um objeto de correspondência gerado por essa regex modificada devolve uma tupla dos componentes do padrão com seu método groups:

```
In [163]: m = regex.match('wesm@bright.net')

In [164]: m.groups()
```

```
Out[164]: ('wesm', 'bright', 'net')
```

findall devolve uma lista de tuplas se o padrão tiver grupos:

```
In [165]: regex.findall(text)
Out[165]:
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

sub também tem acesso aos grupos em cada correspondência, usando símbolos especiais como \1 e \2. O símbolo \1 corresponde ao primeiro grupo identificado, \2 corresponde ao segundo grupo, e assim sucessivamente:

```
In [166]: print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

Há muito mais informações sobre expressões regulares em Python, a maioria das quais está fora do escopo deste livro. A Tabela 7.4 apresenta um resumo rápido.

Tabela 7.4 – Métodos de expressões regulares

Argumento	Descrição
findall	Devolve todos os padrões correspondentes em uma string, sem sobreposição, na forma de uma lista
finditer	É como findall, porém devolve um iterador
match	Corresponde padrões no início da string e, opcionalmente, segmenta componentes do padrão em grupos; se houver uma correspondência com o padrão, devolve um objeto de correspondência; caso contrário, devolve None
search	Pesquisa a string para verificar se há uma correspondência com o padrão; em caso afirmativo, devolve um objeto de correspondência. De modo diferente de match, a correspondência pode se dar em qualquer ponto da string, em oposição a ocorrer somente no início
split	Separa a string em partes a cada ocorrência do padrão

Argumento	Descrição
sub, subn	Substitui todas (sub) ou as n primeiras (subn) ocorrências do padrão em uma string por uma expressão substituta; utiliza os símbolos \1, \2, ... para referenciar os elementos de grupo da correspondência na string de substituição

Funções de string vetorizadas no pandas

Limpar um conjunto de dados desorganizado para análise em geral exige muita manipulação e regularização de strings. Para complicar mais ainda a situação, uma coluna contendo strings ocasionalmente terá dados ausentes:

```
In [167]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
   .....: 'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [168]: data = pd.Series(data)

In [169]: data
Out[169]:
Dave dave@google.com
Rob rob@gmail.com
Steve steve@gmail.com
Wes NaN
dtype: object

In [170]: data.isnull()
Out[170]:
Dave False
Rob False
Steve False
Wes True
dtype: bool
```

Podemos aplicar métodos de string e de expressões regulares (passando uma lambda ou outra função) para cada valor usando data.map, mas haverá uma falha em valores NA (nulos). Para lidar com isso, Series tem métodos orientados a arrays para operações em string, que ignoram valores NA. Eles são acessados por meio do atributo str de Series; por exemplo, poderíamos verificar se cada

endereço de email contém 'gmail' usando str.contains:

```
In [171]: data.str.contains('gmail')
Out[171]:
Dave False
Rob True
Steve True
Wes NaN
dtype: object
```

Expressões regulares também podem ser usadas, junto com qualquer opção de re, como IGNORECASE:

```
In [172]: pattern
Out[172]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [173]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[173]:
Dave [(dave, google, com)]
Rob [(rob, gmail, com)]
Steve [(steve, gmail, com)]
Wes NaN
dtype: object
```

Há duas maneiras para a obtenção de elementos vetorizados. Você pode usar str.get ou indexar no atributo str:

```
In [174]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [175]: matches
Out[175]:
Dave True
Rob True
Steve True
Wes NaN
dtype: object
```

Para acessar os elementos nas listas embutidas, podemos passar um índice para qualquer uma dessas funções:

```
In [176]: matches.str.get(1)
Out[176]:
Dave NaN
Rob NaN
```

```
Steve NaN
Wes NaN
dtype: float64

In [177]: matches.str[0]
Out[177]:
Dave NaN
Rob NaN
Steve NaN
Wes NaN
dtype: float64
```

De modo semelhante, podemos fatiar as strings usando a sintaxe a seguir:

```
In [178]: data.str[:5]
Out[178]:
Dave dave@
Rob rob@g
Steve steve
Wes NaN
dtype: object
```

Veja a Tabela 7.5 que apresenta outros métodos de string do pandas.

Tabela 7.5 – Listagem parcial dos métodos de string vetorizados

Método	Descrição
cat	Concatena strings em todos os elementos, com um delimitador opcional
contains	Devolve um array booleano se cada string contiver um padrão/uma regex
count	Conta as ocorrências do padrão
extract	Utiliza uma expressão regular com grupos para extrair uma ou mais strings de uma Series de strings; o resultado será um DataFrame com uma coluna por grupo
endswith	Equivalente a x.endswith(pattern) para cada elemento
startswith	Equivalente a x.startswith(pattern) para cada elemento
findall	Calcula uma lista com todas as ocorrências de um padrão/uma regex para cada string

Método	Descrição
get	Indexa cada elemento (obtém o *i*-ésimo elemento)
isalnum	Equivalente ao str.alnum embutido
isalpha	Equivalente ao str.isalpha embutido
isdecimal	Equivalente ao str.isdecimal embutido
isdigit	Equivalente ao str.isdigit embutido
islower	Equivalente ao str.islower embutido
isnumeric	Equivalente ao str.isnumeric embutido
isupper	Equivalente ao str.isupper embutido
join	Junta strings em cada elemento da Series utilizando o separador especificado
len	Calcula o tamanho de cada string
lower, upper	Converte para letras minúsculas ou maiúsculas; equivalente a x.lower() ou a x.upper() para cada elemento
match	Usa re.match com a expressão regular especificada em cada elemento, devolvendo os grupos com os quais houve uma correspondência, na forma de uma lista
pad	Adiciona espaços em branco à esquerda, à direita ou nos dois lados das strings
center	Equivalente a pad(side='both')
repeat	Duplica valores (por exemplo, s.str.repeat(3) é equivalente a x * 3 para cada string)
replace	Substitui ocorrências do padrão/da regex por outra string
slice	Fatia cada string da Series
split	Separa as string no delimitador ou na expressão regular
strip	Remove espaços em branco de ambos os lados, incluindo quebras de linha
rstrip	Remove espaços em branco do lado direito
lstrip	Remove espaços em branco do lado esquerdo

7.4 Conclusão

Uma preparação de dados eficiente pode melhorar de modo significativo a produtividade, permitindo que você invista mais tempo analisando dados e menos tempo preparando-os para a análise.

Exploramos uma série de ferramentas neste capítulo, mas a abrangência, de forma alguma, foi completa. No próximo capítulo, exploraremos as funcionalidades de junção e de agrupamento do pandas.

CAPÍTULO 8

Tratamento de dados: junção, combinação e reformatação

Em muitas aplicações, os dados podem estar espalhados em vários arquivos ou bancos de dados, ou podem estar organizados em um formato que não seja fácil de analisar. Este capítulo tem como foco as ferramentas que ajudam a combinar, juntar e reorganizar dados.

Inicialmente, introduzirei o conceito de *indexação hierárquica* no pandas, a qual é intensamente utilizada em algumas dessas operações. Em seguida, explorarei as manipulações de dados em particular. Você verá diversos usos práticos dessas ferramentas no Capítulo 14.

8.1 Indexação hierárquica

A *indexação hierárquica* é um recurso importante do pandas; ela permite ter vários níveis de índices (dois ou mais) em um eixo. De forma, até certo ponto, abstrata, ela oferece uma maneira de trabalhar com dados de dimensões mais altas em um formato de dimensões menores. Vamos começar com um exemplo simples; crie uma Series com uma lista de listas (ou de arrays) como índice:

```
In [9]: data = pd.Series(np.random.randn(9),
   ...: index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
   ...: [1, 2, 3, 1, 3, 1, 2, 2, 3]])

In [10]: data
Out[10]:
a 1 -0.204708
  2 0.478943
```

```
   3 -0.519439
b 1 -0.555730
   3 1.965781
c 1 1.393406
   2 0.092908
d 2 0.281746
   3 0.769023
dtype: float64
```

O que você está vendo é uma visão elegante de uma Series com um MultiIndex como índice. As “lacunas” na exibição do índice significam “utilize o rótulo imediatamente anterior”:

```
In [11]: data.index
Out[11]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1, 1, 2]])
```

Com um objeto hierarquicamente indexado, a chamada indexação *parcial* é possível, permitindo selecionar subconjuntos dos dados de forma concisa:

```
In [12]: data['b']
Out[12]:
1 -0.555730
3 1.965781
dtype: float64

In [13]: data['b':'c']
Out[13]:
b 1 -0.555730
   3 1.965781
c 1 1.393406
   2 0.092908
dtype: float64

In [14]: data.loc[['b', 'd']]
Out[14]:
b 1 -0.555730
   3 1.965781
d 2 0.281746
   3 0.769023
```

```
dtype: float64
```

A seleção é até mesmo possível a partir de um nível “mais interno”:

```
In [15]: data.loc[:, 2]
Out[15]:
a 0.478943
c 0.092908
d 0.281746
dtype: float64
```

A indexação hierárquica desempenha um papel importante na reformatação dos dados e nas operações baseadas em grupos, como compor uma tabela pivô. Por exemplo, poderíamos reorganizar os dados em um DataFrame usando o seu método unstack:

```
In [16]: data.unstack()
Out[16]:
         1 2 3
a -0.204708 0.478943 -0.519439
b -0.555730 NaN 1.965781
c 1.393406 0.092908 NaN
d NaN 0.281746 0.769023
```

A operação inversa de unstack é stack:

```
In [17]: data.unstack().stack()
Out[17]:
a 1 -0.204708
  2 0.478943
  3 -0.519439
b 1 -0.555730
  3 1.965781
c 1 1.393406
  2 0.092908
d 2 0.281746
  3 0.769023
dtype: float64
```

stack e unstack serão explorados detalhadamente, mais adiante neste capítulo.

Em um DataFrame, qualquer eixo pode ter um índice hierárquico:

```
In [18]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
   ....: index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
   ....: columns=[['Ohio', 'Ohio', 'Colorado'],
   ....: ['Green', 'Red', 'Green']])

In [19]: frame
Out[19]:
     Ohio Colorado
    Green Red Green
a 1 0 1 2
  2 3 4 5
b 1 6 7 8
  2 9 10 11
```

Os níveis hierárquicos podem ter nomes (como strings ou quaisquer objetos Python). Nesse caso, eles serão exibidos na saída do console:

```
In [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
state Ohio Colorado
color Green Red Green
key1 key2
a 1 0 1 2
     2 3 4 5
b 1 6 7 8
     2 9 10 11
```

 Tome cuidado para distinguir os nomes dos índices 'state' e 'color' dos rótulos das linhas.

Com uma indexação parcial de colunas, você poderá, de modo semelhante, selecionar grupos de colunas:

```
In [23]: frame['Ohio']
Out[23]:
color Green Red
```

```
key1 key2
a 1 0 1
    2 3 4
b 1 6 7
    2 9 10
```

Um MultiIndex pode ser criado de modo independente e então ser reutilizado; as colunas no DataFrame anterior com nomes para os níveis poderiam ter sido criadas assim:

```
MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                       names=['state', 'color'])
```

Reorganizando e ordenando níveis

Ocasionalmente precisaremos reorganizar a ordem dos níveis em um eixo ou ordenar os dados de acordo com os valores em um nível específico. swaplevel aceita dois números ou nomes de níveis e devolve um novo objeto com os níveis trocados (os dados, porém, permanecem inalterados):

```
In [24]: frame.swaplevel('key1', 'key2')
Out[24]:
state Ohio Colorado
color Green Red Green
key2 key1
1 a 0 1 2
2 a 3 4 5
1 b 6 7 8
2 b 9 10 11
```

sort_index, por outro lado, ordena os dados usando os valores de um só nível. Ao trocar níveis, não é incomum usar sort_index também para que o resultado seja ordenado em ordem lexicográfica de acordo com o nível indicado:

```
In [25]: frame.sort_index(level=1)
Out[25]:
state Ohio Colorado
color Green Red Green
key1 key2
a 1 0 1 2
```

```
b 1 6 7 8
a 2 3 4 5
b 2 9 10 11

In [26]: frame.swaplevel(0, 1).sort_index(level=0)
Out[26]:
state Ohio Colorado
color Green Red Green
key2 key1
1 a 0 1 2
    b 6 7 8
2 a 3 4 5
    b 9 10 11
```

O desempenho da seleção de dados será muito melhor em objetos indexados hierarquicamente se o índice estiver ordenado em ordem lexicográfica, começando pelo nível mais externo – isto é, o resultado da chamada a sort_index(level=0) ou sort_index().

Estatísticas de resumo por nível

Muitas estatísticas descritivas ou de resumo em DataFrame e em Series têm uma opção level; com ela, podemos especificar o nível de acordo com o qual queremos fazer uma agregação, em um eixo em particular. Considere o DataFrame anterior; podemos fazer uma agregação por nível, seja nas linhas ou nas colunas, da seguinte maneira:

```
In [27]: frame.sum(level='key2')
Out[27]:
state Ohio Colorado
color Green Red Green
key2
1 6 8 10
2 12 14 16

In [28]: frame.sum(level='color', axis=1)
Out[28]:
color Green Red
```

```
key1 key2
a 1 2 1
    2 8 4
b 1 14 7
    2 20 10
```

Internamente, o recurso groupby do pandas foi utilizado nesse caso; ele será discutido detalhadamente mais adiante no livro.

Indexando com as colunas de um DataFrame

Não é incomum querer usar uma ou mais colunas de um DataFrame como índice de linha; de modo alternativo, talvez você queira mover o índice das linhas para as colunas do DataFrame. Eis um DataFrame de exemplo:

```
In [29]: frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),
   ....: 'c': ['one', 'one', 'one', 'two', 'two',
   ....: 'two', 'two'],
   ....: 'd': [0, 1, 2, 0, 1, 2, 3]})

In [30]: frame
Out[30]:
   a b c d
0 0 7 one 0
1 1 6 one 1
2 2 5 one 2
3 3 4 two 0
4 4 3 two 1
5 5 2 two 2
6 6 1 two 3
```

A função set_index de DataFrame criará um novo DataFrame usando uma ou mais de suas colunas como índice:

```
In [31]: frame2 = frame.set_index(['c', 'd'])

In [32]: frame2
Out[32]:
       a b
c d
one 0 0 7
```

```
    1 1 6
    2 2 5
  two 0 3 4
    1 4 3
    2 5 2
    3 6 1
```

Por padrão, as colunas são removidas do DataFrame, embora possam ser mantidas:

```
In [33]: frame.set_index(['c', 'd'], drop=False)
Out[33]:
       a b c d
c d
one 0 0 7 one 0
    1 1 6 one 1
    2 2 5 one 2
two 0 3 4 two 0
    1 4 3 two 1
    2 5 2 two 2
    3 6 1 two 3
```

reset_index, por outro lado, faz o inverso de set_index; os níveis dos índices hierárquicos são passados para as colunas:

```
In [34]: frame2.reset_index()
Out[34]:
     c d a b
0 one 0 0 7
1 one 1 1 6
2 one 2 2 5
3 two 0 3 4
4 two 1 4 3
5 two 2 5 2
6 two 3 6 1
```

8.2 Combinando e mesclando conjuntos de dados

Os dados contidos em objetos do pandas podem ser combinados de várias maneiras:

- pandas.merge conecta linhas em DataFrames com base em uma ou mais chaves. Essa operação será conhecida dos usuários de SQL ou de outros bancos de dados relacionais, pois ela implementa as operações de *junção* (join) dos bancos de dados.
- pandas.concat concatena ou "empilha" objetos ao longo de um eixo.
- O método de instância combine_first permite combinar dados que se sobrepõem a fim de preencher valores ausentes em um objeto com valores de outro objeto.

Discutirei cada uma dessas operações e apresentarei alguns exemplos. Essas operações serão utilizadas em exemplos no restante do livro.

Junções no DataFrame no estilo de bancos de dados

Operações de *merge* (mescla) ou de *junção* (join) combinam conjuntos de dados associando linhas por meio de uma ou mais *chaves*. Essas operações são essenciais em bancos de dados relacionais (por exemplo, naqueles baseados em SQL). A função merge do pandas é o ponto de entrada principal para usar esses algoritmos em seus dados.

Vamos começar com um exemplo simples:

```
In [35]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....: 'data1': range(7)})

In [36]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
   ....: 'data2': range(3)})

In [37]: df1
Out[37]:
   data1 key
0 0 b
1 1 b
2 2 a
```

```
3 3 c
4 4 a
5 5 a
6 6 b


In [38]: df2
Out[38]:
   data2 key
0 0 a
1 1 b
2 2 d
```

O exemplo a seguir é de uma junção de *muitos-para-um* (many-to-one); os dados em `df1` têm várias linhas de rótulos `a` e `b`, enquanto `df2` tem apenas uma linha para cada valor na coluna `key`. Se `merge` for chamado nesses objetos, teremos o seguinte:

```
In [39]: pd.merge(df1, df2)
Out[39]:
   data1 key data2
0 0 b 1
1 1 b 1
2 6 b 1
3 2 a 0
4 4 a 0
5 5 a 0
```

Observe que eu não especifiquei a coluna para fazer a junção. Se essa informação não for especificada, `merge` utilizará como chaves os nomes das colunas que se sobrepõem. No entanto, especificá-la explicitamente é uma boa prática:

```
In [40]: pd.merge(df1, df2, on='key')
Out[40]:
   data1 key data2
0 0 b 1
1 1 b 1
2 6 b 1
3 2 a 0
4 4 a 0
5 5 a 0
```

Se os nomes das colunas forem diferentes em cada objeto, você

poderá especificá-las separadamente:

```
In [41]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....: 'data1': range(7)})

In [42]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
   ....: 'data2': range(3)})

In [43]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[43]:
   data1 lkey data2 rkey
0 0 b 1 b
1 1 b 1 b
2 6 b 1 b
3 2 a 0 a
4 4 a 0 a
5 5 a 0 a
```

Talvez você tenha percebido que os valores 'c' e 'd' e os dados associados estão ausentes no resultado. Por padrão, merge executa uma junção do tipo 'inner' (interna); as chaves no resultado são a intersecção, ou o conjunto comum que se encontra nas duas tabelas. Outras opções possíveis são: ‘left', 'right' e 'outer'. A junção externa (outer join) efetua a união das chaves, combinando o efeito da aplicação das junções tanto à esquerda quanto à direita:

```
In [44]: pd.merge(df1, df2, how='outer')
Out[44]:
   data1 key data2
0 0.0 b 1.0
1 1.0 b 1.0
2 6.0 b 1.0
3 2.0 a 0.0
4 4.0 a 0.0
5 5.0 a 0.0
6 3.0 c NaN
7 NaN d 2.0
```

Veja a Tabela 8.1 que contém um resumo das opções para how.

Tabela 8.1 – Diferentes tipos de junção com o argumento how

Opção	Comportamento

'inner'	Utiliza somente as combinações de chaves observadas nas duas tabelas
'left'	Utiliza todas as combinações de chaves encontradas na tabela à esquerda
'right'	Utiliza todas as combinações de chaves encontradas na tabela à direita
'outer'	Utiliza todas as combinações de chaves observadas nas duas tabelas em conjunto

Merges de *muitos-para-muitos* (many-to-many) têm um comportamento bem definido, embora não seja necessariamente intuitivo. Eis um exemplo:

```
In [45]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   ....: 'data1': range(6)})

In [46]: df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
   ....: 'data2': range(5)})

In [47]: df1
Out[47]:
   data1 key
0 0 b
1 1 b
2 2 a
3 3 c
4 4 a
5 5 b

In [48]: df2
Out[48]:
   data2 key
0 0 a
1 1 b
2 2 a
3 3 b
4 4 d

In [49]: pd.merge(df1, df2, on='key', how='left')
Out[49]:
    data1 key data2
0 0 b 1.0
1 0 b 3.0
```

```
2 1 b 1.0
3 1 b 3.0
4 2 a 0.0
5 2 a 2.0
6 3 c NaN
7 4 a 0.0
8 4 a 2.0
9 5 b 1.0
10 5 b 3.0
```

Junções de muitos para muitos formam o produto cartesiano das linhas. Como havia três linhas 'b' no DataFrame da esquerda e duas no da direita, há seis linhas 'b' no resultado. O método de junção afeta somente os valores de chaves distintos que aparecem no resultado:

```
In [50]: pd.merge(df1, df2, how='inner')
Out[50]:
   data1 key data2
0 0 b 1
1 0 b 3
2 1 b 1
3 1 b 3
4 5 b 1
5 5 b 3
6 2 a 0
7 2 a 2
8 4 a 0
9 4 a 2
```

Para um merge com várias chaves, passe uma lista de nomes de coluna:

```
In [51]: left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
   ....: 'key2': ['one', 'two', 'one'],
   ....: 'lval': [1, 2, 3]})

In [52]: right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   ....: 'key2': ['one', 'one', 'one', 'two'],
   ....: 'rval': [4, 5, 6, 7]})

In [53]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

```
Out[53]:
  key1 key2 lval rval
0 foo one 1.0 4.0
1 foo one 1.0 5.0
2 foo two 2.0 NaN
3 bar one 3.0 6.0
4 bar two NaN 7.0
```

Para determinar quais combinações de chaves aparecerão no resultado conforme a escolha do método de merge, pense nas várias chaves como compondo um array de tuplas a ser usado como uma única chave de junção (apesar de isso não ser realmente implementado dessa forma).

Ao fazer a junção de colunas-sobre-colunas, os índices dos objetos DataFrame especificados serão descartados.

Uma última questão a ser considerada em operações de merge é o tratamento dos nomes de coluna que se sobrepõem. Embora seja possível tratar a sobreposição manualmente (veja a seção anterior sobre renomear rótulos de eixos), merge tem uma opção suffixes para especificar strings a serem concatenadas nos nomes que se sobrepõem, nos objetos DataFrame à esquerda e à direita:

```
In [54]: pd.merge(left, right, on='key1')
Out[54]:
  key1 key2_x lval key2_y rval
0 foo one 1 one 4
1 foo one 1 one 5
2 foo two 2 one 4
3 foo two 2 one 5
4 bar one 3 one 6
5 bar one 3 two 7

In [55]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
Out[55]:
  key1 key2_left lval key2_right rval
0 foo one 1 one 4
1 foo one 1 one 5
```

2 foo two 2 one 4
3 foo two 2 one 5
4 bar one 3 one 6
5 bar one 3 two 7

Veja a Tabela 8.2 que contém uma referência para os argumentos de merge. Fazer junções usando o índice de linhas do DataFrame será o assunto da próxima seção.

Tabela 8.2 – Argumentos da função merge

Argumento	Descrição
left	Dataframe à esquerda para o merge
right	Dataframe à direita para o merge
how	Uma opção entre 'inner', 'outer', 'left' ou 'right'; o default é 'inner'
on	Nomes de coluna para a junção. Deve ser encontrado nos dois objetos DataFrame. Se não for especificado e nenhuma outra chave de junção for definida, usará a intersecção entre os nomes de colunas em left e em right como as chaves da junção
left_on	Colunas no DataFrame left a serem usadas como chaves de junção
right_on	Análogo ao left_on para o DataFrame left
left_index	Utiliza o índice de linha em left como sua chave de junção (ou chaves, se for um MultiIndex)
right_index	Análogo a left_index
sort	Ordena os dados do merge em ordem lexicográfica, de acordo com as chaves de junção; o default é True (desative-o para ter um desempenho melhor em alguns casos, quando os conjuntos de dados forem grandes)
suffixes	Tupla de valores de strings a serem concatenados aos nomes de colunas em caso de sobreposição; o default é ('_x', '_y') (por exemplo, se houver ‘data’ nos dois objetos DataFrame, esses aparecerão como 'data_x' e 'data_y' no resultado)
copy	Se for False, evita copiar dados para a estrutura de dados resultante em alguns casos excepcionais; por padrão, sempre copia
indicator	Adiciona uma coluna especial _merge que informa a origem de cada linha; os valores serão 'left_only', 'right_only' ou 'both' conforme a origem dos dados da junção em cada linha

Fazendo merge com base no índice

Em alguns casos, a(s) chave(s) do merge em um DataFrame serão encontradas em seu índice. Nessa situação, você poderá passar left_index=True ou right_index=True (ou ambos) para informar que o índice deverá ser usado como a chave do merge:

```
In [56]: left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
   ....: 'value': range(6)})

In [57]: right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])

In [58]: left1
Out[58]:
  key value
0 a 0
1 b 1
2 a 2
3 a 3
4 b 4
5 c 5

In [59]: right1
Out[59]:
   group_val
a 3.5
b 7.0

In [60]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[60]:
  key value group_val
0 a 0 3.5
2 a 2 3.5
3 a 3 3.5
1 b 1 7.0
4 b 4 7.0
```

Como o método default de merge consiste em fazer uma intersecção das chaves de junção, você poderá, de modo alternativo, compor a união delas fazendo uma junção externa:

```
In [61]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
Out[61]:
```

```
  key value group_val
0 a 0 3.5
2 a 2 3.5
3 a 3 3.5
1 b 1 7.0
4 b 4 7.0
5 c 5 NaN
```

Com dados hierarquicamente indexados, a situação é mais complicada, pois fazer uma junção com base no índice é implicitamente um merge de várias chaves:

```
In [62]: lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',
   ....: 'Nevada', 'Nevada'],
   ....: 'key2': [2000, 2001, 2002, 2001, 2002],
   ....: 'data': np.arange(5.)})

In [63]: righth = pd.DataFrame(np.arange(12).reshape((6, 2)),
   ....: index=[['Nevada', 'Nevada', 'Ohio', 'Ohio',
   ....: 'Ohio', 'Ohio'],
   ....: [2001, 2000, 2000, 2000, 2001, 2002]],
   ....: columns=['event1', 'event2'])

In [64]: lefth
Out[64]:
  data key1 key2
0 0.0 Ohio 2000
1 1.0 Ohio 2001
2 2.0 Ohio 2002
3 3.0 Nevada 2001
4 4.0 Nevada 2002

In [65]: righth
Out[65]:
           event1 event2
Nevada 2001 0 1
       2000 2 3
Ohio 2000 4 5
       2000 6 7
       2001 8 9
       2002 10 11
```

Nesse caso, você deverá especificar várias colunas com base nas quais o merge será feito, na forma de uma lista (observe o tratamento de valores de índice duplicados com how='outer'):

```
In [66]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
Out[66]:
   data key1 key2 event1 event2
0 0.0 Ohio 2000 4 5
0 0.0 Ohio 2000 6 7
1 1.0 Ohio 2001 8 9
2 2.0 Ohio 2002 10 11
3 3.0 Nevada 2001 0 1

In [67]: pd.merge(lefth, righth, left_on=['key1', 'key2'],
   ....: right_index=True, how='outer')
Out[67]:
   data key1 key2 event1 event2
0 0.0 Ohio 2000 4.0 5.0
0 0.0 Ohio 2000 6.0 7.0
1 1.0 Ohio 2001 8.0 9.0
2 2.0 Ohio 2002 10.0 11.0
3 3.0 Nevada 2001 0.0 1.0
4 4.0 Nevada 2002 NaN NaN
4 NaN Nevada 2000 2.0 3.0
```

Usar os índices dos dois lados do merge também é possível:

```
In [68]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
   ....: index=['a', 'c', 'e'],
   ....: columns=['Ohio', 'Nevada'])

In [69]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
   ....: index=['b', 'c', 'd', 'e'],
   ....: columns=['Missouri', 'Alabama'])

In [70]: left2
Out[70]:
   Ohio Nevada
a 1.0 2.0
c 3.0 4.0
e 5.0 6.0
```

```
In [71]: right2
Out[71]:
   Missouri Alabama
b 7.0 8.0
c 9.0 10.0
d 11.0 12.0
e 13.0 14.0

In [72]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
Out[72]:
   Ohio Nevada Missouri Alabama
a 1.0 2.0 NaN NaN
b NaN NaN 7.0 8.0
c 3.0 4.0 9.0 10.0
d NaN NaN 11.0 12.0
e 5.0 6.0 13.0 14.0
```

O DataFrame tem uma instância conveniente de join para fazer o merge pelo índice. Ela também pode ser usada para combinar vários objetos DataFrame com índices iguais ou semelhantes, porém com colunas que não se sobrepõem. No exemplo anterior, poderíamos ter escrito:

```
In [73]: left2.join(right2, how='outer')
Out[73]:
   Ohio Nevada Missouri Alabama
a 1.0 2.0 NaN NaN
b NaN NaN 7.0 8.0
c 3.0 4.0 9.0 10.0
d NaN NaN 11.0 12.0
e 5.0 6.0 13.0 14.0
```

Em parte por questões de legado (isto é, versões bem antigas do pandas), o método join de DataFrame realiza uma junção à esquerda nas chaves de junção, preservando exatamente o índice de linhas do frame à esquerda. Ele também aceita a junção do índice do DataFrame recebido em uma das colunas do DataFrame que faz a chamada:

```
In [74]: left1.join(right1, on='key')
Out[74]:
```

```
  key value group_val
0 a 0 3.5
1 b 1 7.0
2 a 2 3.5
3 a 3 3.5
4 b 4 7.0
5 c 5 NaN
```

Por fim, para merges simples de índice-sobre-índice, podemos passar uma lista de DataFrames para join como uma alternativa ao uso da função concat, mais genérica, que será descrita na próxima seção:

```
In [75]: another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
   ....: index=['a', 'c', 'e', 'f'],
   ....: columns=['New York', 'Oregon'])

In [76]: another
Out[76]:
  New York Oregon
a 7.0 8.0
c 9.0 10.0
e 11.0 12.0
f 16.0 17.0

In [77]: left2.join([right2, another])
Out[77]:
  Ohio Nevada Missouri Alabama New York Oregon
a 1.0 2.0 NaN NaN 7.0 8.0
c 3.0 4.0 9.0 10.0 9.0 10.0
e 5.0 6.0 13.0 14.0 11.0 12.0

In [78]: left2.join([right2, another], how='outer')
Out[78]:
  Ohio Nevada Missouri Alabama New York Oregon
a 1.0 2.0 NaN NaN 7.0 8.0
b NaN NaN 7.0 8.0 NaN NaN
c 3.0 4.0 9.0 10.0 9.0 10.0
d NaN NaN 11.0 12.0 NaN NaN
e 5.0 6.0 13.0 14.0 11.0 12.0
f NaN NaN NaN NaN 16.0 17.0
```

Concatenando ao longo de um eixo

Outro tipo de operação de combinação de dados é chamado, de forma indistinta, de concatenação, vinculação (binding) ou empilhamento (stacking). A função concatenate do NumPy é capaz de fazer isso com arrays NumPy:

```
In [79]: arr = np.arange(12).reshape((3, 4))

In [80]: arr
Out[80]:
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])

In [81]: np.concatenate([arr, arr], axis=1)
Out[81]:
array([[ 0, 1, 2, 3, 0, 1, 2, 3],
       [ 4, 5, 6, 7, 4, 5, 6, 7],
       [ 8, 9, 10, 11, 8, 9, 10, 11]])
```

No contexto de objetos do pandas, como Series e DataFrame, ter eixos com rótulos permite generalizar melhor a concatenação de arrays. Em particular, há um número adicional de aspectos nos quais devemos pensar:

- Se os objetos estiverem indexados de modo diferente nos outros eixos, devemos combinar os elementos distintos nesses eixos ou usar somente os valores compartilhados (a intersecção)?
- As porções de dados concatenadas devem ser identificáveis no objeto resultante?
- O “eixo de concatenação” contém dados que devam ser preservados? Em muitos casos, será melhor que os rótulos inteiros default em um DataFrame sejam descartados durante a concatenação.

A função concat do pandas oferece uma forma consistente de tratar cada um desses aspectos. Apresentarei uma série de exemplos para mostrar o seu funcionamento. Suponha que tenhamos três

Series sem sobreposição de índices:

```
In [82]: s1 = pd.Series([0, 1], index=['a', 'b'])

In [83]: s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])

In [84]: s3 = pd.Series([5, 6], index=['f', 'g'])
```

Chamar concat com esses objetos em uma lista reúne os valores e os índices:

```
In [85]: pd.concat([s1, s2, s3])
Out[85]:
a 0
b 1
c 2
d 3
e 4
f 5
g 6
dtype: int64
```

Por padrão, concat atua em axis=0, gerando outra Series. Se você passar axis=1, o resultado será um DataFrame (axis=1 são as colunas):

```
In [86]: pd.concat([s1, s2, s3], axis=1)
Out[86]:
    0 1 2
a 0.0 NaN NaN
b 1.0 NaN NaN
c NaN 2.0 NaN
d NaN 3.0 NaN
e NaN 4.0 NaN
f NaN NaN 5.0
g NaN NaN 6.0
```

Nesse caso, não há sobreposição no outro eixo que, como podemos ver, é a união ordenada (a junção 'outer') dos índices. Por outro lado, podemos fazer a sua intersecção passando join='inner':

```
In [87]: s4 = pd.concat([s1, s3])

In [88]: s4
```

```
Out[88]:
a 0
b 1
f 5
g 6
dtype: int64

In [89]: pd.concat([s1, s4], axis=1)
Out[89]:
    0 1
a 0.0 0
b 1.0 1
f NaN 5
g NaN 6

In [90]: pd.concat([s1, s4], axis=1, join='inner')
Out[90]:
  0 1
a 0 0
b 1 1
```

Nesse último exemplo, os rótulos 'f' e 'g' desapareceram por causa da opção join='inner'.

Podemos até mesmo especificar os eixos a serem usados nos demais eixos com join_axes:

```
In [91]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
Out[91]:
    0 1
a 0.0 0.0
c NaN NaN
b 1.0 1.0
e NaN NaN
```

Um possível problema é o fato de as partes concatenadas não serem identificáveis no resultado. Suponha que quiséssemos criar um índice hierárquico no eixo da concatenação. Para isso, utilize o argumento keys:

```
In [92]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])

In [93]: result
```

```
Out[93]:
one a 0
      b 1
two a 0
      b 1
three f 5
      g 6
dtype: int64

In [94]: result.unstack()
Out[94]:
       a b f g
one 0.0 1.0 NaN NaN
two 0.0 1.0 NaN NaN
three NaN NaN 5.0 6.0
```

No caso da combinação de Series ao longo de axis=1, as keys passam a ser os cabeçalhos das colunas do DataFrame:

```
In [95]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
Out[95]:
  one two three
a 0.0 NaN NaN
b 1.0 NaN NaN
c NaN 2.0 NaN
d NaN 3.0 NaN
e NaN 4.0 NaN
f NaN NaN 5.0
g NaN NaN 6.0
```

A mesma lógica se estende aos objetos DataFrame:

```
In [96]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
   ....: columns=['one', 'two'])

In [97]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
   ....: columns=['three', 'four'])

In [98]: df1
Out[98]:
  one two
a 0 1
b 2 3
```

```
c 4 5

In [99]: df2
Out[99]:
   three four
a 5 6
c 7 8

In [100]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
Out[100]:
  level1 level2
     one two three four
a 0 1 5.0 6.0
b 2 3 NaN NaN
c 4 5 7.0 8.0
```

Se você passar um dicionário de objetos no lugar de uma lista, as chaves do dicionário serão usadas para a opção keys:

```
In [101]: pd.concat({'level1': df1, 'level2': df2}, axis=1)
Out[101]:
  level1 level2
     one two three four
a 0 1 5.0 6.0
b 2 3 NaN NaN
c 4 5 7.0 8.0
```

Há argumentos adicionais que determinam o modo como o índice hierárquico é criado (veja a Tabela 8.3). Por exemplo, podemos nomear os níveis criados no eixo com o argumento names:

```
In [102]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
   .....: names=['upper', 'lower'])
Out[102]:
upper level1 level2
lower one two three four
a 0 1 5.0 6.0
b 2 3 NaN NaN
c 4 5 7.0 8.0
```

Uma última consideração diz respeito aos DataFrames em que o índice das linhas não contém nenhum dado relevante:

```
In [103]: df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
```

```
In [104]: df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])

In [105]: df1
Out[105]:
        a b c d
0 1.246435 1.007189 -1.296221 0.274992
1 0.228913 1.352917 0.886429 -2.001637
2 -0.371843 1.669025 -0.438570 -0.539741

In [106]: df2
Out[106]:
        b d a
0 0.476985 3.248944 -1.021228
1 -0.577087 0.124121 0.302614
```

Nesse caso, podemos passar ignore_index=True:

```
In [107]: pd.concat([df1, df2], ignore_index=True)
Out[107]:
        a b c d
0 1.246435 1.007189 -1.296221 0.274992
1 0.228913 1.352917 0.886429 -2.001637
2 -0.371843 1.669025 -0.438570 -0.539741
3 -1.021228 0.476985 NaN 3.248944
4 0.302614 -0.577087 NaN 0.124121
```

Tabela 8.3 – Argumentos da função concat

Argumento	Descrição
objs	Lista ou dicionário de objetos do pandas a serem concatenados; é o único argumento obrigatório
axis	Eixo no qual será feita a concatenação; o default é 0 (nas linhas)
join	Pode ser 'inner' ou 'outer' (o default é 'outer'); indica se será feita a intersecção (inner) ou a união (outer) dos índices ao longo dos outros eixos
join_axes	Índices específicos a serem usados para os outros *n*–1 eixos, em vez de executar a lógica de união/intersecção

Argumento	Descrição
keys	Valores a serem associados aos objetos sendo concatenados, compondo um índice hierárquico ao longo do eixo de concatenação; pode ser uma lista ou um array de valores arbitrários, um array de tuplas ou uma lista de arrays (se arrays de vários níveis forem especificados em levels)
levels	Índices específicos a serem usados como nível ou níveis de índices hierárquicos se chaves forem especificadas
names	Nomes para os níveis hierárquicos criados se keys e/ou levels forem especificados
verify_integrity	Verifica o novo eixo no objeto concatenado em busca de duplicatas e lança uma exceção em caso afirmativo; por padrão (False), permite duplicatas
ignore_index	Não preserva os índices ao longo do axis de concatenação, gerando um novo índice range(total_length) em seu lugar

Combinando dados com sobreposição

Há outra situação de combinação de dados que não pode ser expressa nem com uma operação de merge nem como de concatenação. Podemos ter dois conjuntos de dados cujos índices se sobreponham de forma total ou parcial. Como um exemplo motivador, considere a função where do NumPy, que executa o equivalente a uma expressão if-else, porém orientado a arrays:

```
In [108]: a = pd.Series([np.nan, 2.5, 0.0, 3.5, 4.5, np.nan],
   .....: index=['f', 'e', 'd', 'c', 'b', 'a'])

In [109]: b = pd.Series([0., np.nan, 2., np.nan, np.nan, 5.],
   .....: index=['a', 'b', 'c', 'd', 'e', 'f'])

In [110]: a
Out[110]:
f NaN
e 2.5
d 0.0
c 3.5
b 4.5
a NaN
dtype: float64
```

```
In [111]: b
Out[111]:
a 0.0
b NaN
c 2.0
d NaN
e NaN
f 5.0
dtype: float64

In [112]: np.where(pd.isnull(a), b, a)
Out[112]: array([ 0. , 2.5, 0. , 3.5, 4.5, 5. ])
```

Uma Series tem um método combine_first, que executa o equivalente a essa operação, junto com a lógica usual de alinhamento de dados do pandas:

```
In [113]: b.combine_first(a)
Out[113]:
a 0.0
b 4.5
c 2.0
d 0.0
e 2.5
f 5.0
dtype: float64
```

Com DataFrames, combine_first faz o mesmo coluna a coluna, portanto podemos pensar nele como se estivesse fazendo um “patching” dos dados ausentes no objeto que faz a chamada, com os dados do objeto que você lhe passar:

```
In [114]: df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan],
   .....: 'b': [np.nan, 2., np.nan, 6.],
   .....: 'c': range(2, 18, 4)})

In [115]: df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.],
   .....: 'b': [np.nan, 3., 4., 6., 8.]})

In [116]: df1
Out[116]:
```

```
    a b c
0 1.0 NaN 2
1 NaN 2.0 6
2 5.0 NaN 10
3 NaN 6.0 14

In [117]: df2
Out[117]:
    a b
0 5.0 NaN
1 4.0 3.0
2 NaN 4.0
3 3.0 6.0
4 7.0 8.0

In [118]: df1.combine_first(df2)
Out[118]:
    a b c
0 1.0 NaN 2.0
1 4.0 2.0 6.0
2 5.0 4.0 10.0
3 3.0 6.0 14.0
4 7.0 8.0 NaN
```

8.3 Reformatação e pivoteamento

Há uma série de operações básicas para reorganização de dados tabulares. De modo alternativo, são chamadas de operações de *reformatação* (reshaping) ou de *pivoteamento* (pivoting).

Reformatação com indexação hierárquica

A indexação hierárquica oferece uma forma consistente de reorganizar dados em um DataFrame. Há duas ações principais:

stack

Faz a “rotação” ou o pivoteamento das colunas dos dados para as linhas.

unstack

Faz o pivoteamento das linhas para as colunas.

Demonstrarei essas operações com uma série de exemplos. Considere um pequeno DataFrame com arrays de strings como índices das linhas e das colunas:

```
In [119]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),
   .....: index=pd.Index(['Ohio', 'Colorado'], name='state'),
   .....: columns=pd.Index(['one', 'two', 'three'],
   .....: name='number'))

In [120]: data
Out[120]:
number one two three
state
Ohio 0 1 2
Colorado 3 4 5
```

Utilizar o método stack nesses dados faz o pivoteamento das colunas para as linhas, gerando uma Series:

```
In [121]: result = data.stack()

In [122]: result
Out[122]:
state number
Ohio one 0
        two 1
        three 2
Colorado one 3
        two 4
        three 5
dtype: int64
```

A partir de uma Series indexada hierarquicamente, podemos reorganizar os dados de volta em um DataFrame usando unstack:

```
In [123]: result.unstack()
Out[123]:
number one two three
state
Ohio 0 1 2
Colorado 3 4 5
```

Por padrão, o nível mais interno será desempilhado (o mesmo vale para stack). Podemos desempilhar um nível diferente passando um número ou o nome de um nível:

```
In [124]: result.unstack(0)
Out[124]:
state Ohio Colorado
number
one 0 3
two 1 4
three 2 5

In [125]: result.unstack('state')
Out[125]:
state Ohio Colorado
number
one 0 3
two 1 4
three 2 5
```

Desempilhar pode introduzir dados ausentes se nem todos os valores do nível forem encontrados em cada um dos subgrupos:

```
In [126]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])

In [127]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])

In [128]: data2 = pd.concat([s1, s2], keys=['one', 'two'])

In [129]: data2
Out[129]:
one a 0
    b 1
    c 2
    d 3
two c 4
    d 5
    e 6
dtype: int64

In [130]: data2.unstack()
Out[130]:
```

```
      a b c d e
one 0.0 1.0 2.0 3.0 NaN
two NaN NaN 4.0 5.0 6.0
```

O empilhamento filtra dados ausentes por padrão, de modo que a operação seja mais facilmente revertida:

```
In [131]: data2.unstack()
Out[131]:
      a b c d e
one 0.0 1.0 2.0 3.0 NaN
two NaN NaN 4.0 5.0 6.0

In [132]: data2.unstack().stack()
Out[132]:
one a 0.0
    b 1.0
    c 2.0
    d 3.0
two c 4.0
    d 5.0
    e 6.0
dtype: float64

In [133]: data2.unstack().stack(dropna=False)
Out[133]:
one a 0.0
    b 1.0
    c 2.0
    d 3.0
    e NaN
two a NaN
    b NaN
    c 4.0
    d 5.0
    e 6.0
dtype: float64
```

Quando desempilhamos dados em um DataFrame, o nível desempilhado passa a ser o nível mais baixo no resultado:

```
In [134]: df = pd.DataFrame({'left': result, 'right': result + 5},
   .....: columns=pd.Index(['left', 'right'], name='side'))
```

```
In [135]: df
Out[135]:
side left right
state number
Ohio one 0 5
        two 1 6
        three 2 7
Colorado one 3 8
        two 4 9
        three 5 10

In [136]: df.unstack('state')
Out[136]:
side left right
state Ohio Colorado Ohio Colorado
number
one 0 3 5 8
two 1 4 6 9
three 2 5 7 10
```

Quando chamamos stack, podemos informar o nome do eixo a ser empilhado:

```
In [137]: df.unstack('state').stack('side')
Out[137]:
state Colorado Ohio
number side
one left 3 0
      right 8 5
two left 4 1
      right 9 6
three left 5 2
      right 10 7
```

Fazendo o pivoteamento de um formato “longo” para um formato “largo”

Uma maneira comum de armazenar várias séries temporais em bancos de dados e em CSV é usar o chamado formato *longo* (long) ou *empilhado* (stacked). Vamos carregar alguns dados de exemplo e

fazer uma pequena manipulação de séries temporais e outras operações de limpeza de dados:

```
In [138]: data = pd.read_csv('examples/macrodata.csv')

In [139]: data.head()
Out[139]:
    year quarter realgdp realcons realinv realgovt realdpi cpi \
0 1959.0 1.0 2710.349 1707.4 286.898 470.045 1886.9 28.98
1 1959.0 2.0 2778.801 1733.7 310.859 481.301 1919.7 29.15
2 1959.0 3.0 2775.488 1751.8 289.226 491.260 1916.4 29.35
3 1959.0 4.0 2785.204 1753.7 299.356 484.052 1931.3 29.37
4 1960.0 1.0 2847.699 1770.5 331.722 462.199 1955.5 29.54
     m1 tbilrate unemp pop infl realint
0 139.7 2.82 5.8 177.146 0.00 0.00
1 141.7 3.08 5.1 177.830 2.34 0.74
2 140.5 3.82 5.3 178.657 2.74 1.09
3 140.0 4.33 5.6 179.386 0.27 4.06
4 139.6 3.50 5.2 180.007 2.31 1.19

In [140]: periods = pd.PeriodIndex(year=data.year, quarter=data.quarter,
   .....: name='date')

In [141]: columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item')

In [142]: data = data.reindex(columns=columns)

In [143]: data.index = periods.to_timestamp('D', 'end')

In [144]: ldata = data.stack().reset_index().rename(columns={0: 'value'})
```

Veremos PeriodIndex com um pouco mais de detalhes no Capítulo 11. Para resumir, ele combina as colunas year e quarter de modo a criar um tipo de intervalo de tempo.

Eis a aparência de ldata:

```
In [145]: ldata[:10]
Out[145]:
       date item value
0 1959-03-31 realgdp 2710.349
1 1959-03-31 infl 0.000
```

```
2 1959-03-31 unemp 5.800
3 1959-06-30 realgdp 2778.801
4 1959-06-30 infl 2.340
5 1959-06-30 unemp 5.100
6 1959-09-30 realgdp 2775.488
7 1959-09-30 infl 2.740
8 1959-09-30 unemp 5.300
9 1959-12-31 realgdp 2785.204
```

Esse é o chamado formato *longo* para várias séries temporais, ou outros dados observados com duas ou mais chaves (nesse caso, nossas chaves são a data e o item). Cada linha da tabela representa uma única observação.

Os dados são frequentemente armazenados dessa maneira em bancos de dados relacionais como o MySQL, pois um esquema fixo (nomes de coluna e tipos de dados) permite que o número de valores distintos na coluna item mude à medida que dados são adicionados à tabela. No exemplo anterior, date e item normalmente seriam as chaves primárias (no jargão de bancos de dados relacionais), oferecendo tanto uma integridade relacional quanto junções mais fáceis. Em alguns casos, talvez seja mais difícil trabalhar com os dados nesse formato; você pode preferir ter um DataFrame contendo uma coluna por valor distinto de item indexado por timestamps na coluna date. O método pivot de DataFrame executa exatamente essa transformação:

```
In [146]: pivoted = ldata.pivot('date', 'item', 'value')

In [147]: pivoted
Out[147]:
item infl realgdp unemp
date
1959-03-31 0.00 2710.349 5.8
1959-06-30 2.34 2778.801 5.1
1959-09-30 2.74 2775.488 5.3
1959-12-31 0.27 2785.204 5.6
1960-03-31 2.31 2847.699 5.2
1960-06-30 0.14 2834.390 5.2
1960-09-30 2.70 2839.022 5.6
```

```
1960-12-31 1.21 2802.616 6.3
1961-03-31 -0.40 2819.264 6.8
1961-06-30 1.47 2872.005 7.0
... ... ... ...
2007-06-30 2.75 13203.977 4.5
2007-09-30 3.45 13321.109 4.7
2007-12-31 6.38 13391.249 4.8
2008-03-31 2.82 13366.865 4.9
2008-06-30 8.53 13415.266 5.4
2008-09-30 -3.16 13324.600 6.0
2008-12-31 -8.79 13141.920 6.9
2009-03-31 0.94 12925.410 8.1
2009-06-30 3.37 12901.504 9.2
2009-09-30 3.56 12990.341 9.6
[203 rows x 3 columns]
```

Os dois primeiros valores especificados são as colunas a serem usadas respectivamente como o índice das linhas e das colunas e, então, por fim, um valor de coluna opcional para preencher o DataFrame. Suponha que tivéssemos dois valores de coluna que quiséssemos reformatar simultaneamente:

```
In [148]: ldata['value2'] = np.random.randn(len(ldata))

In [149]: ldata[:10]
Out[149]:
        date item value value2
0 1959-03-31 realgdp 2710.349 0.523772
1 1959-03-31 infl 0.000 0.000940
2 1959-03-31 unemp 5.800 1.343810
3 1959-06-30 realgdp 2778.801 -0.713544
4 1959-06-30 infl 2.340 -0.831154
5 1959-06-30 unemp 5.100 -2.370232
6 1959-09-30 realgdp 2775.488 -1.860761
7 1959-09-30 infl 2.740 -0.860757
8 1959-09-30 unemp 5.300 0.560145
9 1959-12-31 realgdp 2785.204 -1.265934
```

Ao omitir o último argumento, obteremos um DataFrame com colunas hierárquicas:

```
In [150]: pivoted = ldata.pivot('date', 'item')
```

```
In [151]: pivoted[:5]
Out[151]:
          value value2
item infl realgdp unemp infl realgdp unemp
date
1959-03-31 0.00 2710.349 5.8 0.000940 0.523772 1.343810
1959-06-30 2.34 2778.801 5.1 -0.831154 -0.713544 -2.370232
1959-09-30 2.74 2775.488 5.3 -0.860757 -1.860761 0.560145
1959-12-31 0.27 2785.204 5.6 0.119827 -1.265934 -1.063512
1960-03-31 2.31 2847.699 5.2 -2.359419 0.332883 -0.199543

In [152]: pivoted['value'][:5]
Out[152]:
item infl realgdp unemp
date
1959-03-31 0.00 2710.349 5.8
1959-06-30 2.34 2778.801 5.1
1959-09-30 2.74 2775.488 5.3
1959-12-31 0.27 2785.204 5.6
1960-03-31 2.31 2847.699 5.2
```

Observe que pivot é equivalente a criar um índice hierárquico usando set_index seguido de uma chamada a unstack:

```
In [153]: unstacked = ldata.set_index(['date', 'item']).unstack('item')

In [154]: unstacked[:7]
Out[154]:
          value value2
item infl realgdp unemp infl realgdp unemp
date
1959-03-31 0.00 2710.349 5.8 0.000940 0.523772 1.343810
1959-06-30 2.34 2778.801 5.1 -0.831154 -0.713544 -2.370232
1959-09-30 2.74 2775.488 5.3 -0.860757 -1.860761 0.560145
1959-12-31 0.27 2785.204 5.6 0.119827 -1.265934 -1.063512
1960-03-31 2.31 2847.699 5.2 -2.359419 0.332883 -0.199543
1960-06-30 0.14 2834.390 5.2 -0.970736 -1.541996 -1.307030
1960-09-30 2.70 2839.022 5.6 0.377984 0.286350 -0.753887
```

Pivoteamento do formato “largo” para o formato

“longo”

Uma operação inversa de pivot para DataFrames é pandas.melt. Em vez de transformar uma coluna em várias em um novo DataFrame, ele fará o merge de várias colunas em uma só, gerando um DataFrame maior que a entrada. Vamos ver um exemplo:

```
In [156]: df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],
   .....: 'A': [1, 2, 3],
   .....: 'B': [4, 5, 6],
   .....: 'C': [7, 8, 9]})

In [157]: df
Out[157]:
   A B C key
0 1 4 7 foo
1 2 5 8 bar
2 3 6 9 baz
```

A coluna 'key' pode ser um indicador de grupo, e as outras colunas são valores de dados. Ao usar pandas.melt, devemos informar quais colunas (se houver) são indicadores de grupo. Vamos usar 'key' como o único indicador de grupo nesse caso:

```
In [158]: melted = pd.melt(df, ['key'])

In [159]: melted
Out[159]:
   key variable value
0 foo A 1
1 bar A 2
2 baz A 3
3 foo B 4
4 bar B 5
5 baz B 6
6 foo C 7
7 bar C 8
8 baz C 9
```

Usando pivot, podemos reformatar novamente e obter o layout original:

```
In [160]: reshaped = melted.pivot('key', 'variable', 'value')

In [161]: reshaped
Out[161]:
variable A B C
key
bar 2 5 8
baz 3 6 9
foo 1 4 7
```

Como o resultado de pivot cria um índice a partir da coluna usada como os rótulos das linhas, podemos usar reset_index para passar os dados de volta para uma coluna:

```
In [162]: reshaped.reset_index()
Out[162]:
variable key A B C
0 bar 2 5 8
1 baz 3 6 9
2 foo 1 4 7
```

Também é possível especificar um subconjunto de colunas a serem usadas como colunas de valores:

```
In [163]: pd.melt(df, id_vars=['key'], value_vars=['A', 'B'])
Out[163]:
   key variable value
0 foo A 1
1 bar A 2
2 baz A 3
3 foo B 4
4 bar B 5
5 baz B 6
```

pandas.melt também pode ser usado sem nenhum identificador de grupo:

```
In [164]: pd.melt(df, value_vars=['A', 'B', 'C'])
Out[164]:
  variable value
0 A 1
1 A 2
2 A 3
3 B 4
```

```
4 B 5
5 B 6
6 C 7
7 C 8
8 C 9

In [165]: pd.melt(df, value_vars=['key', 'A', 'B'])
Out[165]:
  variable value
0 key foo
1 key bar
2 key baz
3 A 1
4 A 2
5 A 3
6 B 4
7 B 5
8 B 6
```

8.4 Conclusão

Agora que dispomos de algum conhecimento básico do pandas para importação, limpeza e reorganização de dados, estamos prontos para prosseguir em direção à visualização de dados com a matplotlib. Retornaremos ao pandas mais adiante no livro, momento em que discutiremos uma análise de dados mais sofisticada.

CAPÍTULO 9

Plotagem e visualização

Gerar visualizações informativas (às vezes chamadas de *plotagens*) é uma das tarefas mais importantes em análise de dados. Essa tarefa pode fazer parte do processo exploratório – por exemplo, para ajudar a identificar valores discrepantes (outliers) ou transformações necessárias nos dados, ou como uma forma de gerar ideias para modelos. Para outras pessoas, criar uma visualização interativa para a web talvez seja o objetivo final. Python tem muitas bibliotecas add-on para criar visualizações estáticas ou dinâmicas; contudo, manterei o foco principalmente na matplotlib (*http://matplotlib.sourceforge.net/*) e nas bibliotecas desenvolvidas com base nela.

A matplotlib é um pacote de plotagem para desktop, projetada para criar plotagens com qualidade para publicação (em sua maior parte, bidimensionais). O projeto foi criado por John Hunter em 2002 com o intuito de possibilitar uma interface de plotagem do tipo MATLAB em Python. As comunidades da matplotlib e do IPython têm colaborado para simplificar a plotagem interativa a partir do shell IPython (e, atualmente, com o notebook Jupyter). A biblioteca aceita vários backends de GUI em todos os sistemas operacionais e, além disso, é capaz de exportar visualizações para todos os vetores comuns e formatos de gráficos raster (PDF, SVG, JPG, PNG, BMP, GIF etc.). Com exceção de poucos diagramas, quase todos os gráficos neste livro foram gerados com a matplotlib.

Com o passar do tempo, a matplotlib deu origem a uma série de kits de ferramentas add-on para visualização de dados que a utilizam para a sua plotagem subjacente. Um deles é o seaborn

(*http://seaborn.pydata.org/*), que exploraremos mais adiante neste capítulo.

O modo mais simples de acompanhar os códigos de exemplo no capítulo é usar a plotagem interativa no notebook Jupyter. Para configurá-la, execute a instrução a seguir em um notebook Jupyter:

```
%matplotlib notebook
```

9.1 Introdução rápida à API da matplotlib

Com a matplotlib, usamos a seguinte convenção de importação:

```
In [11]: import matplotlib.pyplot as plt
```

Depois de executar `%matplotlib notebook` no Jupyter (ou simplesmente `%matplotlib` no IPython), podemos tentar a criação de uma plotagem simples. Se tudo estiver configurado corretamente, uma plotagem de linha, como mostra a Figura 9.1, deverá ser exibida:

```
In [12]: import numpy as np

In [13]: data = np.arange(10)

In [14]: data
Out[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [15]: plt.plot(data)
```

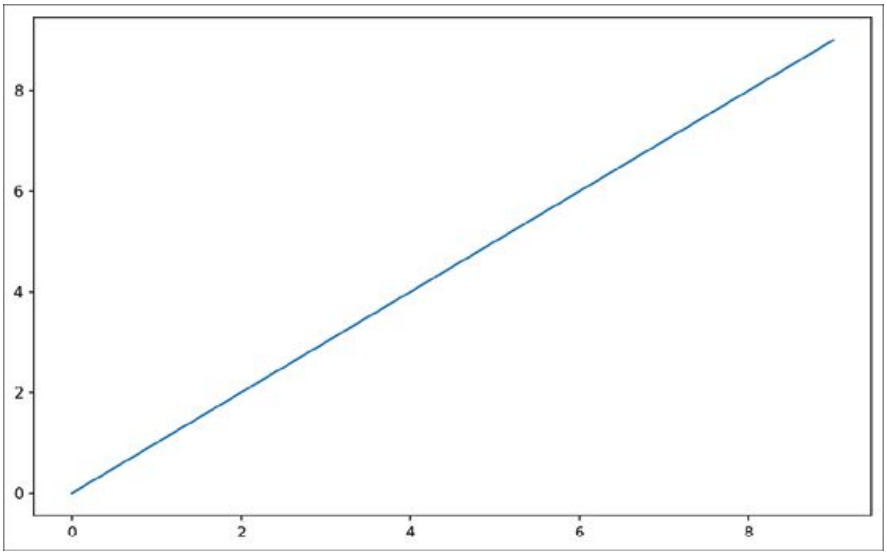


Figura 9.1 – Uma plotagem de linha simples.

Embora bibliotecas como seaborn e as funções embutidas de plotagem do pandas lidem com muitos dos detalhes mundanos da criação de plotagens, caso você precise personalizá-las para além das opções disponibilizadas pelas funções, será necessário conhecer um pouco melhor a API da matplotlib.

Não há espaço suficiente neste livro para um tratamento completo de todos os detalhes das funcionalidades da matplotlib. Ensinar o básico a fim de que você esteja pronto para executá-la deverá ser suficiente. A galeria da matplotlib e a sua documentação são os melhores recursos para conhecer os recursos avançados.

Figuras e subplotagens

As plotagens na matplotlib ficam em um objeto Figure. Podemos criar uma nova figura com plt.figure:

```
In [16]: fig = plt.figure()
```

No IPython, uma janela de plotagem vazia aparecerá, mas, no Jupyter, nada será exibido até que mais alguns comandos sejam usados. plt.figure tem uma série de opções; em especial, figsize garantirá que a figura tenha determinado tamanho e uma razão de aspecto (aspect ratio) se ela for salva em disco.

Não é possível criar uma plotagem com uma figura em branco. Você deve criar uma ou mais subplots usando add_subplot:

```
In [17]: ax1 = fig.add_subplot(2, 2, 1)
```

Essa instrução significa que a figura deve ser de 2 × 2 (portanto até quatro plotagens no total), e estamos selecionando a primeira das quatro subplotagens (numeradas a partir de 1). Se você criar as duas próximas subplotagens, acabará com uma visualização que tem o aspecto mostrado na Figura 9.2:

```
In [18]: ax2 = fig.add_subplot(2, 2, 2)

In [19]: ax3 = fig.add_subplot(2, 2, 3)
```

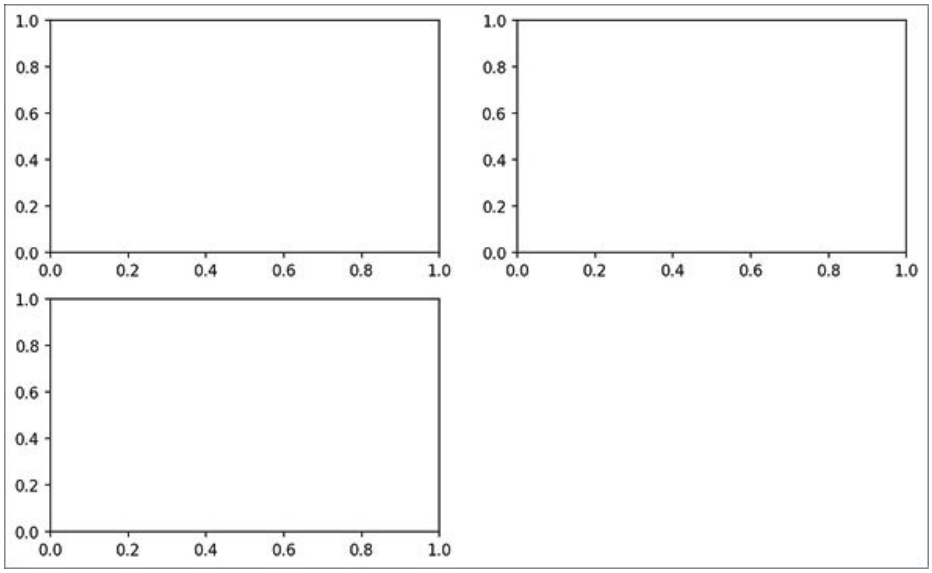


Figura 9.2 – Uma figura vazia com três subplotagens na matplotlib.

Uma das nuances ao usar notebooks Jupyter é que as plotagens são reiniciadas após cada célula ter sido avaliada, portanto, para plotagens mais complexas, devemos colocar todos os comandos de plotagem em uma única célula do notebook.

Nesse exemplo, executaremos todos os comandos a seguir na mesma célula:

```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
```

Quando executamos um comando de plotagem como `plt.plot([1.5, 3.5, -2, 1.6])`, a matplotlib desenhará na última figura e subplotagem usadas (criando uma, se for necessário), ocultando, assim, a criação da figura e da subplotagem. Assim, se adicionarmos o comando a seguir, teremos algo semelhante à Figura 8.3:

```
In [20]: plt.plot(np.random.randn(50).cumsum(), 'k--')
```

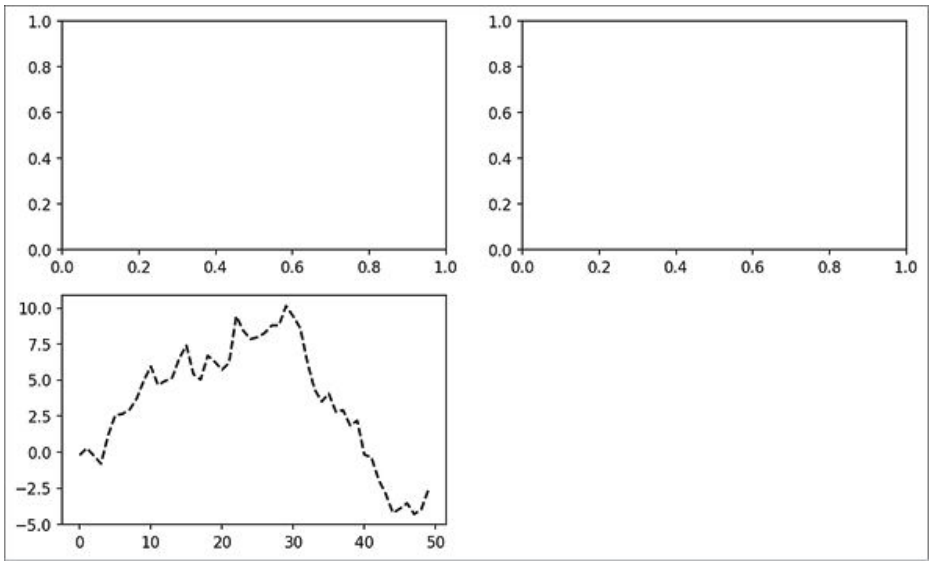


Figura 9.3 – Visualização de dados após uma única plotagem.

O 'k--' é uma opção de *estilo* que instrui a matplotlib a plotar uma linha tracejada preta. Os objetos devolvidos por fig.add_subplot nesse caso são objetos AxesSubplot; você pode plotar diretamente nas outras subplotagens vazias chamando o método de instância de cada uma (veja a Figura 9.4):

```
In [21]: _ = ax1.hist(np.random.randn(100), bins=20, color='k', alpha=0.3)

In [22]: ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.randn(30))
```

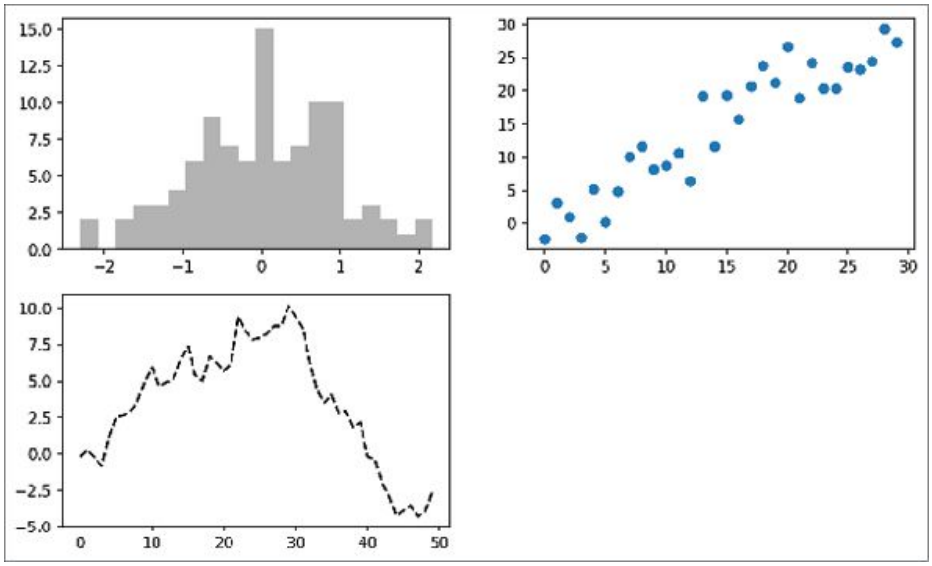


Figura 9.4 – Visualização de dados após plotagens adicionais.

Você poderá encontrar um catálogo completo dos tipos de plotagens

na documentação da matplotlib (*http://matplotlib.sourceforge.net/*).

Criar uma figura com uma grade de subplotagens é uma tarefa bem comum, de modo que a matplotlib inclui um método conveniente, plt.subplots, que cria uma nova figura e devolve um array NumPy contendo os objetos de subplotagem criados:

```
In [24]: fig, axes = plt.subplots(2, 3)

In [25]: axes
Out[25]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f435346c668>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f435338c780>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f43533c37f0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f435337d8d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f4353336908>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f43532ea400>]],
dtype
=object)
```

Isso é muito útil, pois o array axes pode ser facilmente indexado como um array bidimensional; por exemplo, axes[0, 1]. Também podemos informar que as subplotagens devem ter os mesmos eixos x ou y usando sharex e sharey, respectivamente. Isso é particularmente conveniente se estivermos comparando dados na mesma escala; caso contrário, a matplotlib definirá automaticamente uma escala para os limites das plotagens de forma independente. Veja a Tabela 9.1 que contém mais informações sobre esse método.

Tabela 9.1 – Opções de pyplot.subplots

Argumento	Descrição
nrows	Número de linhas das subplotagens
ncols	Número de colunas das subplotagens
sharex	Todas as subplotagens devem usar os mesmos tiques no eixo x (ajustar o xlim afetará todas as subplotagens)
sharey	Todas as subplotagens devem usar os mesmos tiques no eixo y (ajustar o ylim afetará todas as subplotagens)
subplot_kw	Dicionário de argumentos nomeados passados para a chamada a add_subplot para criar cada subplotagem.

Argumento	Descrição
**fig_kw	Argumentos nomeados adicionais para subplots são usadas para criar a figura, por exemplo, plt.subplots(2, 2, figsize=(8, 6))

Ajustando o espaçamento em torno das subplotagens

Por padrão, a matplotlib deixa determinado espaço para preenchimento em torno das subplotagens, além de um espaçamento entre elas. Todos esses espaçamentos são especificados em relação à altura e à largura da plotagem, de modo que, se você redimensionar a plotagem, seja por meio de um programa ou manualmente usando a janela de GUI, ela se ajustará dinamicamente. O espaçamento pode ser alterado usando o método subplots_adjust dos objetos Figure, também disponível como uma função de nível superior:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

wspace e hspace controlam o percentual da largura e da altura da figura, respectivamente, a serem usados como espaçamento entre as subplotagens. Eis um pequeno exemplo em que reduzi o espaçamento totalmente para zero (veja a Figura 9.5):

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i, j].hist(np.random.randn(500), bins=50, color='k', alpha=0.5)
plt.subplots_adjust(wspace=0, hspace=0)
```

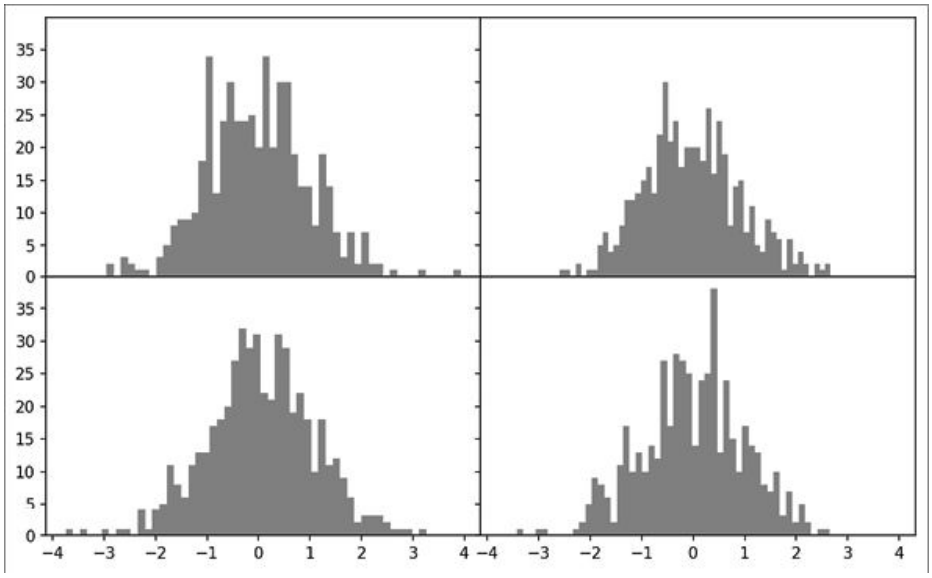


Figura 9.5 – Visualização de dados sem espaçamento entre as subplotagens.

Talvez você tenha notado que há sobreposição entre os rótulos dos eixos. A matplotlib não verifica se os rótulos se sobrepõem, portanto, em um caso como esse, será necessário corrigir os rótulos por conta própria especificando explicitamente os locais dos tiques e seus rótulos (veremos como fazer isso nas próximas seções).

Cores, marcadores e estilos de linha

A função principal plot da matplotlib aceita arrays de coordenadas x e y e, opcionalmente, uma string abreviada informando a cor e o estilo da linha. Por exemplo, para plotar x *versus* y com traços verdes, executaríamos:

```
ax.plot(x, y, 'g--')
```

Essa forma de especificar tanto a cor quanto o estilo da linha em uma string é oferecida como uma conveniência; na prática, se você estiver criando plotagens com um programa, talvez prefira não ter que lidar com strings para criar plotagens com o estilo desejado. A mesma plotagem também poderia ter sido expressa de modo mais explícito, assim:

```
ax.plot(x, y, linestyle='--', color='g')
```

Há uma série de abreviaturas de cores, disponibilizada para cores comumente usadas, mas podemos utilizar qualquer cor do espectro especificando o seu código hexa (por exemplo, '#CECECE'). Veja o conjunto completo de estilos de linha consultando a docstring de plot (utilize plot? no IPython ou no Jupyter).

Plotagens de linha, além do mais, podem ter *marcadores* para destacar os pontos de dados propriamente ditos. Como a matplotlib cria uma plotagem de linha contínua, fazendo uma interpolação entre os pontos, ocasionalmente talvez não esteja claro onde estão os pontos. O marcador pode fazer parte da string de estilo, que deve conter a cor, seguida do tipo de marcador e do estilo da linha (veja a Figura 9.6):

```
In [30]: from numpy.random import randn

In [31]: plt.plot(randn(30).cumsum(), 'ko--')
```

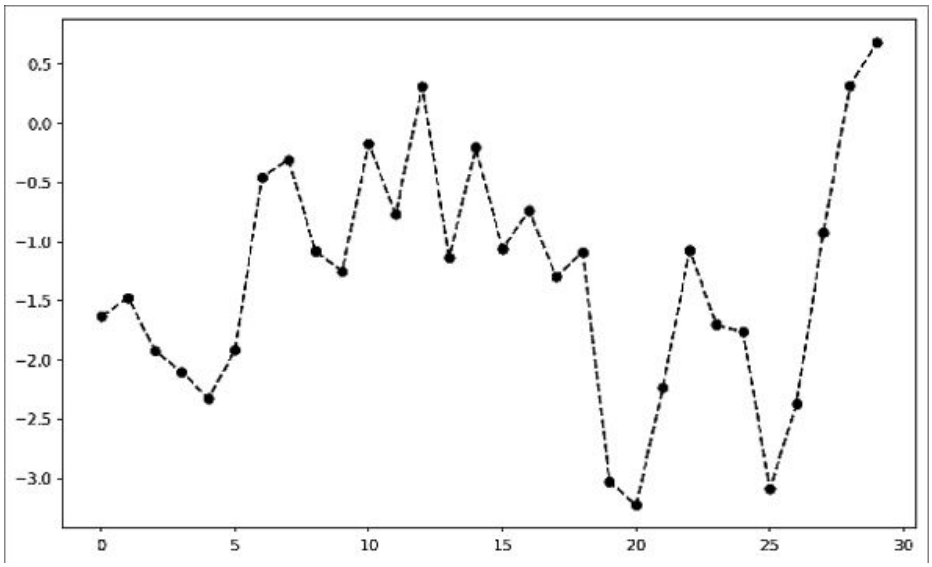


Figura 9.6 – Plotagem de linha com marcadores.

Esse código poderia ter sido escrito mais explicitamente, assim:

```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
```

Para plotagens de linha, você perceberá que pontos subsequentes, por padrão, são interpolados linearmente. Isso pode ser alterado com a opção drawstyle (Figura 9.7):

```
In [33]: data = np.random.randn(30).cumsum()

In [34]: plt.plot(data, 'k--', label='Default')
Out[34]: [<matplotlib.lines.Line2D at 0x7f43502adcf8>]

In [35]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
Out[35]: [<matplotlib.lines.Line2D at 0x7f43502b62e8>]

In [36]: plt.legend(loc='best')
```

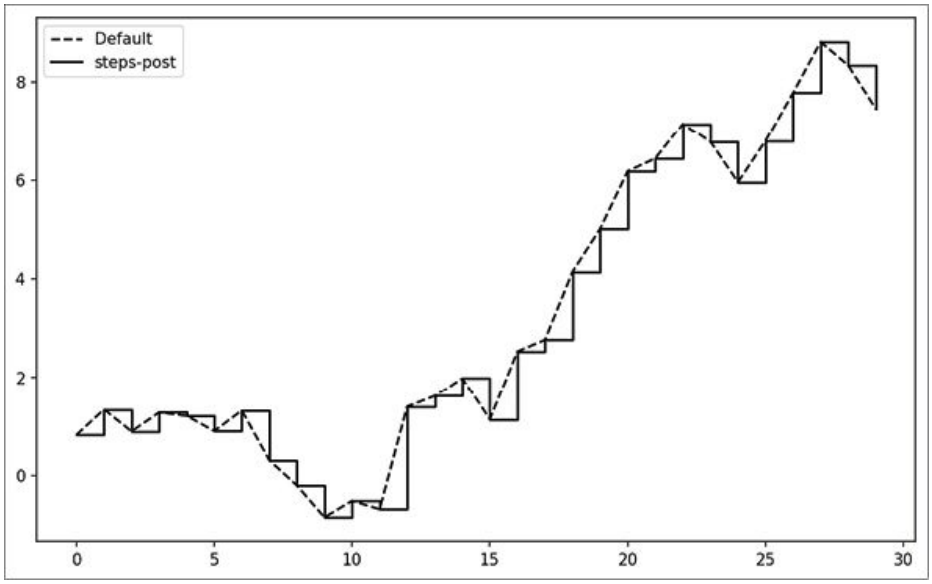


Figura 9.7 – Plotagem de linha com diferentes opções para drawstyle.

Você poderá notar uma saída como <matplotlib.lines.Line2D at ...> quando executar essa instrução. A matplotlib devolve objetos que referenciam o subcomponente da plotagem que acabou de ser adicionado. Muitas vezes você poderá ignorar essa saída, sem problemas. Nesse exemplo, como passamos o argumento label para plot, pudemos criar uma legenda para a plotagem a fim de identificar cada linha usando plt.legend.

Você deve chamar plt.legend (ou ax.legend, se tiver uma referência para os eixos) a fim de criar a legenda, independentemente de ter passado as opções de label ao plotar os dados.

Tiques, rótulos e legendas

Para a maioria dos tipos de decoração das plotagens, há duas maneiras principais de proceder: usar a interface procedural pyplot (isto é, matplotlib.pyplot) ou a API nativa da matplotlib, mais orientada a objetos.

A interface pyplot, projetada para uso interativo, é constituída de métodos como xlim, xticks e xticklabels. Eles controlam o intervalo da plotagem, as localizações dos tiques e seus rótulos, respectivamente. Podem ser usados de duas maneiras.

- Se forem chamados sem argumentos, devolvem o valor atual do parâmetro (por exemplo, plt.xlim() devolve o intervalo atual de plotagem no eixo x).
- Se forem chamados com parâmetros, definem seus valores (por exemplo, plt.xlim([0, 10]) define o intervalo do eixo x como sendo de 0 a 10).

Todos esses métodos atuam no AxesSubplot ativo ou mais recentemente criado. Cada um deles corresponde a dois métodos no objeto de subplotagem; no caso de xlim, são ax.get_xlim e ax.set_xlim. Eu prefiro usar os métodos de instância da subplotagem para ser mais explícito (e, particularmente, quando trabalho com várias subplotagens), mas, certamente, você pode usar a opção que lhe for mais conveniente.

Definindo o título, os rótulos dos eixos, os tiques e os rótulos dos tiques

Para demonstrar a personalização dos eixos, criarei uma figura simples e plotarei um passeio aleatório (random walk – veja a Figura 9.8):

```
In [37]: fig = plt.figure()

In [38]: ax = fig.add_subplot(1, 1, 1)

In [39]: ax.plot(np.random.randn(1000).cumsum())
```

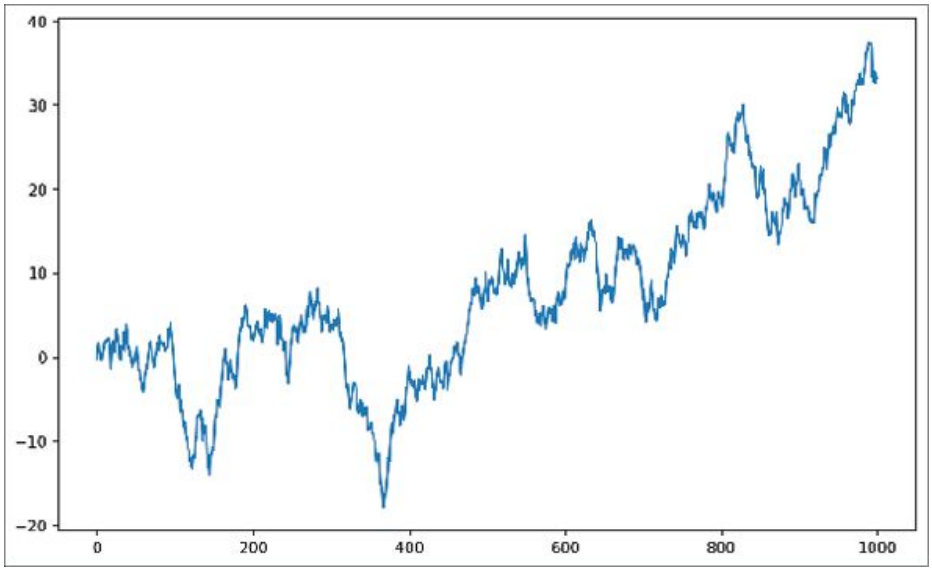


Figura 9.8 – Plotagem simples para demonstração de xticks (com rótulo).

Para alterar os tiques do eixo x, a opção mais simples é usar set_xticks e set_xticklabels. A primeira instrui a matplotlib acerca do local para posicionar os tiques no intervalo de dados; por padrão, esses locais também serão os rótulos. No entanto, podemos definir qualquer outro valor como rótulos usando set_xticklabels:

```
In [40]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])

In [41]: labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'],
   ....: rotation=30, fontsize='small')
```

A opção rotation define os rótulos dos tiques de x com uma rotação de 30°. Por fim, set_xlabel dá um nome ao eixo x, enquanto set_title define o título da subplotagem (veja a Figura 9.9 que exibe a figura resultante):

```
In [42]: ax.set_title('My first matplotlib plot')
Out[42]: Text(0.5,1,'My first matplotlib plot')

In [43]: ax.set_xlabel('Stages')
```

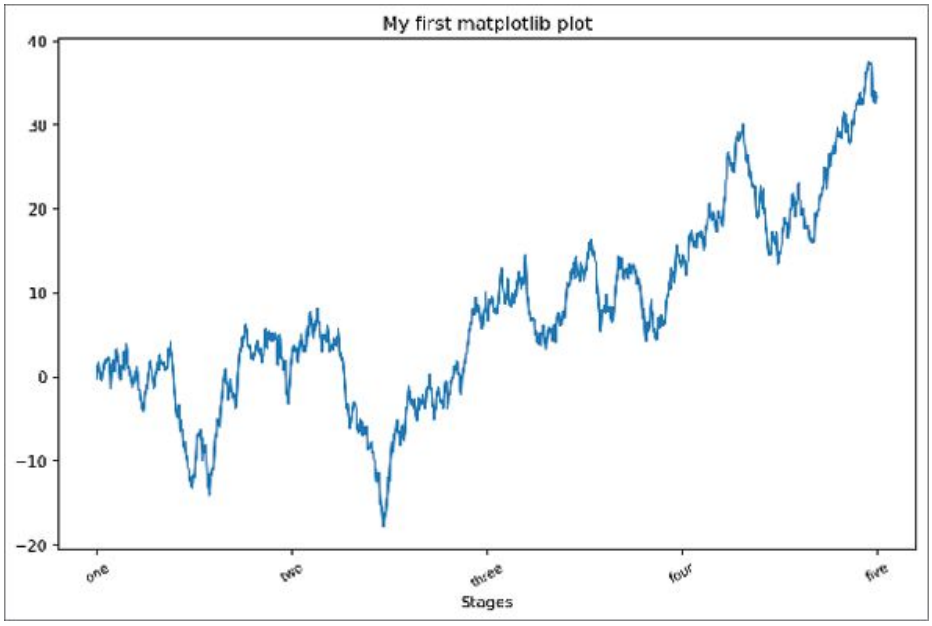


Figura 9.9 – Plotagem simples para demonstração de xticks.

Para modificar o eixo y, o processo é o mesmo, substituindo `x` por `y` no código anterior. A classe dos eixos tem um método `set` que permite a definição das propriedades de plotagem em lote. No exemplo anterior, também poderíamos ter escrito:

```
props = {
    'title': 'My first matplotlib plot',
    'xlabel': 'Stages'
}
ax.set(**props)
```

Acrescentando legendas

As legendas são outro elemento crucial para identificar elementos da plotagem. Há duas maneiras de adicionar uma legenda. O modo mais simples é passar o argumento `label` ao adicionar cada parte da plotagem:

```
In [44]: from numpy.random import randn

In [45]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)

In [46]: ax.plot(randn(1000).cumsum(), 'k', label='one')
Out[46]: [<matplotlib.lines.Line2D at 0x7f4350204da0>]
```

```
In [47]: ax.plot(randn(1000).cumsum(), 'k--', label='two')
Out[47]: [<matplotlib.lines.Line2D at 0x7f435020e390>]

In [48]: ax.plot(randn(1000).cumsum(), 'k.', label='three')
Out[48]: [<matplotlib.lines.Line2D at 0x7f435020e828>]
```

Depois de ter feito isso, podemos chamar ax.legend() ou plt.legend() para criar automaticamente uma legenda. A plotagem resultante pode ser vista na Figura 9.10:

```
In [49]: ax.legend(loc='best')
```

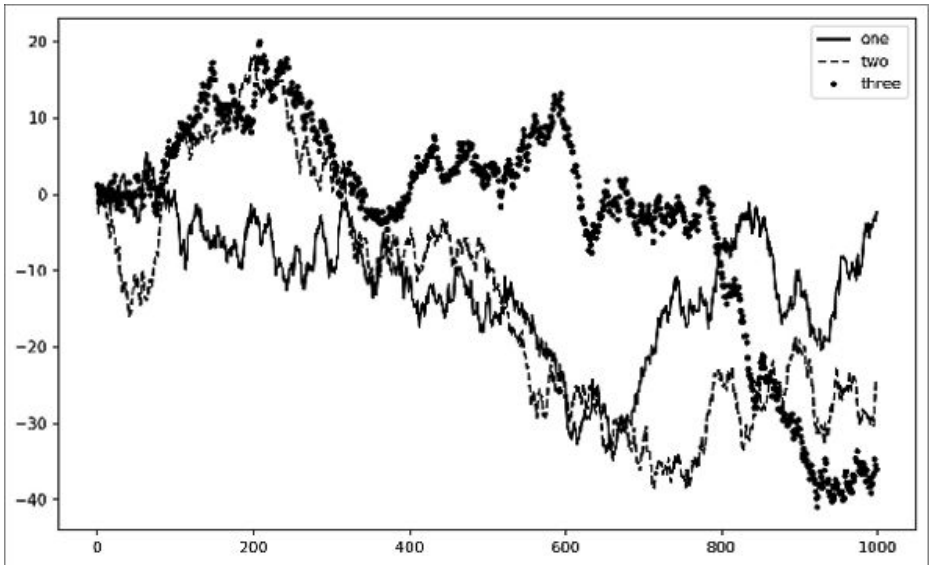


Figura 9.10 – Plotagem simples com três linhas e uma legenda.

O método legend tem várias outras opções para o argumento de localização loc. Consulte a docstring (com ax.legend?) para obter mais informações.

loc diz à matplotlib em que local a plotagem deve ser posicionada. Se você não for excessivamente detalhista, 'best' será uma boa opção, pois ele escolherá uma localização que atrapalhe menos. Para excluir um ou mais elementos da legenda, não passe nenhum rótulo ou passe label='_nolegend_'.

Anotações e desenhos em uma subplotagem

Além dos tipos padrões de plotagem, talvez você queira desenhar suas próprias anotações na plotagem, que poderiam ser compostas

de textos, setas ou outras formas. É possível adicionar anotações e textos usando as funções text, arrow e annotate. A função text desenha um texto nas coordenadas especificadas (x, y) da plotagem, com uma estilização personalizada opcional:

```
ax.text(x, y, 'Hello world!',
        family='monospace', fontsize=10)
```

As anotações podem incluir tanto texto quanto setas, organizados de modo apropriado. Como exemplo, vamos plotar os preços de fechamento do índice S&P 500 a partir de 2007 (obtidos do Yahoo! Finance), e fazer anotações com algumas das datas importantes da crise financeira de 2008 a 2009. Você pode reproduzir facilmente esse código de exemplo, em sua maior parte, em uma única célula de um notebook Jupyter. A Figura 9.11 mostra o resultado:

```
from datetime import datetime

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

data = pd.read_csv('examples/spx.csv', index_col=0, parse_dates=True)
spx = data['SPX']

spx.plot(ax=ax, style='k-')

crisis_data = [
    (datetime(2007, 10, 11), 'Peak of bull market'),
    (datetime(2008, 3, 12), 'Bear Stearns Fails'),
    (datetime(2008, 9, 15), 'Lehman Bankruptcy')
]

for date, label in crisis_data:
    ax.annotate(label, xy=(date, spx.asof(date) + 75),
                xytext=(date, spx.asof(date) + 225),
                arrowprops=dict(facecolor='black', headwidth=4, width=2,
                                headlength=4),
                horizontalalignment='left', verticalalignment='top')

# Faz um zoom no período de 2007 a 2010
ax.set_xlim(['1/1/2007', '1/1/2011'])
```

```
ax.set_ylim([600, 1800])

ax.set_title('Important dates in the 2008-2009 financial crisis')
```

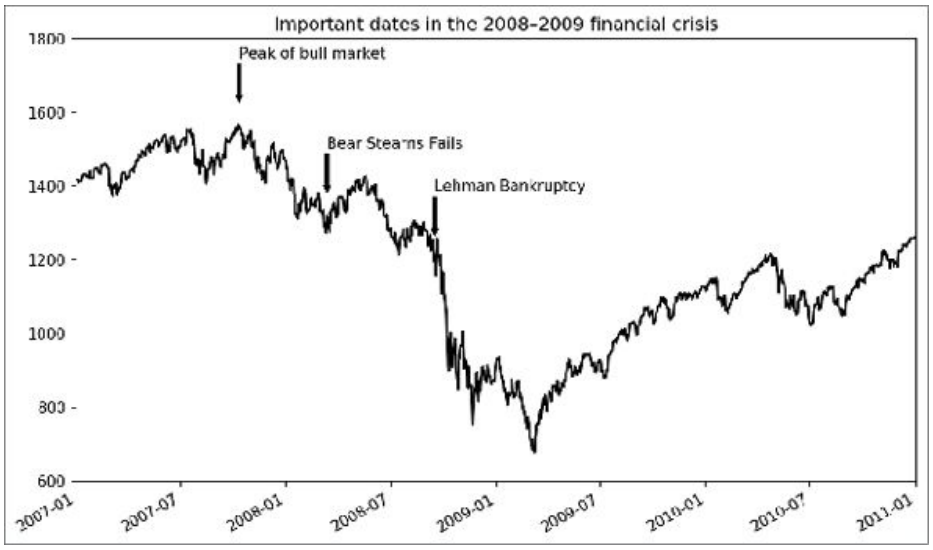


Figura 9.11 – Datas importantes na crise financeira de 2008 a 2009.

Há alguns pontos importantes a serem enfatizados nessa plotagem: o método ax.annotate é capaz de desenhar rótulos nas coordenadas x e y indicadas. Usamos os métodos set_xlim e set_ylim para definir manualmente as fronteiras de início e de fim da plotagem, em vez de usar o default da matplotlib. Por fim, ax.set_title adiciona um título principal à plotagem.

Consulte a galeria online da matplotlib para ver muitos outros exemplos de anotações e aprender com eles.

Desenhar formas exige um pouco mais de cuidado. A matplotlib tem objetos que representam muitas formas comuns, conhecidas como *patches*. Algumas delas, como Rectangle e Circle, se encontram em matplotlib.pyplot, mas o conjunto completo está em matplotlib.patches.

Para adicionar uma forma à plotagem, crie o objeto patch shp e adicione-o a uma subplotagem chamando ax.add_patch(shp) (veja a Figura 9.12):

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
```

```
rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
                   color='g', alpha=0.5)

ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```

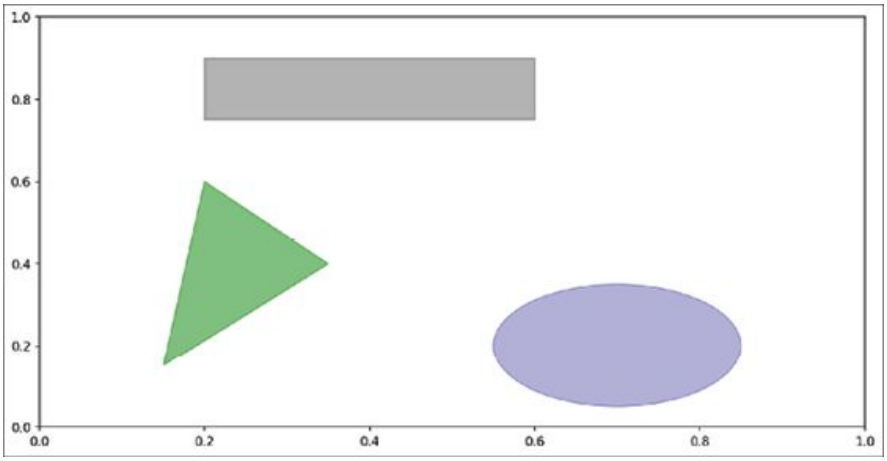


Figura 9.12 – Visualização de dados composta de três patches distintos.

Se observar a implementação de muitos tipos de plotagem familiares, você verá que eles são compostos de patches.

Salvando plotagens em arquivos

Podemos salvar a figura ativa em um arquivo usando plt.savefig. Esse método é equivalente ao método de instância savefig do objeto que representa a figura. Por exemplo, para salvar uma versão SVG de uma figura, basta digitar:

```
plt.savefig('figpath.svg')
```

O tipo do arquivo é inferido a partir de sua extensão. Portanto, se você usou .pdf, terá um PDF. Há duas opções importantes que uso com frequência para gráficos que serão publicados: dpi, que controla a resolução em pontos por polegada, e bbox_inches, que pode remover espaços em branco em torno da figura. Para obter a mesma plotagem como um PNG com um mínimo de espaços em

branco em torno da plotagem e com 400 DPI, poderíamos usar o seguinte:

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

savefig não precisa escrever em disco; ele também pode escrever em qualquer objeto do tipo arquivo, como BytesIO:

```
from io import BytesIO
buffer = BytesIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

Veja a Tabela 9.2 que contém uma lista de outras opções para savefig.

Tabela 9.2 – Opções de figure.savefig

Argumento	Descrição
fname	String contendo um path de arquivo ou um objeto Python do tipo arquivo. O formato da figura é inferido a partir da extensão do arquivo (por exemplo, .pdf para PDF ou .png para PNG)
dpi	É a resolução da figura em pontos por polegada; o default é 100, mas pode ser configurado
facecolor, edgecolor	Cor do plano de fundo da figura, fora das subplotagens; o default é 'w' (branco)
format	Formato de arquivo explícito a ser usado ('png', 'pdf', 'svg', 'ps', 'eps', ...)
bbox_inches	Parte da figura a ser salva; se 'tight' for especificado, tentará remover o espaço vazio em torno da figura

Configuração da matplotlib

A matplotlib vem configurada com esquemas de cores e defaults visando principalmente à preparação das figuras para publicação. Felizmente quase todo o comportamento default pode ser personalizado por meio de um conjunto amplo de parâmetros globais que determinam o tamanho da figura, o espaçamento das subplotagens, as cores, os tamanhos das fontes, os estilos de grade e assim por diante. Uma forma de modificar a configuração por meio de um programa Python é usar o método rc; por exemplo, para

definir o tamanho default da figura globalmente como 10 × 10, poderíamos usar o seguinte:

```
plt.rc('figure', figsize=(10, 10))
```

O primeiro argumento de `rc` é o componente que desejamos personalizar, como `'figure'`, `'axes'`, `'xtick'`, `'ytick'`, `'grid'`, `'legend'` ou vários outros. Depois disso, pode haver uma sequência de argumentos nomeados informando os novos parâmetros. Uma maneira fácil de escrever as opções em seu programa é na forma de um dicionário:

```
font_options = {'family' : 'monospace',
                'weight' : 'bold',
                'size' : 'small'}
plt.rc('font', **font_options)
```

Para uma personalização mais extensa e para ver uma lista de todas as opções, a matplotlib vem com um arquivo de configuração *matplotlibrc* no diretório *matplotlib/mpl-data*. Se esse arquivo for personalizado e colocado em seu diretório home chamado *.matplotlibrc*, ele será carregado sempre que você usar a matplotlib.

Conforme veremos na próxima seção, o pacote seaborn tem vários temas de plotagem embutidos, ou *estilos*, que utilizam o sistema de configuração da matplotlib internamente.

9.2 Plottagem com o pandas e o seaborn

A matplotlib pode ser uma ferramenta razoavelmente de baixo nível. Compomos uma plotagem a partir de seus componentes básicos: modo de exibição dos dados (isto é, o tipo da plotagem: de linha, de barras, de caixa, de dispersão, de contorno etc.), legenda, título, rótulos para os tiques e outras anotações.

No pandas, podemos ter várias colunas de dados, junto com rótulos para linhas e colunas. O próprio pandas tem métodos embutidos que simplificam a criação de visualizações a partir de objetos DataFrame e Series. Outra biblioteca é o seaborn (*https://seaborn.pydata.org/*): uma biblioteca gráfica de estatísticas criada por Michael Waskom. O seaborn simplifica a criação de vários tipos

comuns de visualização.

A importação do seaborn modifica os esquemas de cores e os estilos de plotagem padrões da matplotlib a fim de melhorar a legibilidade e a estética. Mesmo que você não use a API do seaborn, talvez prefira importar essa biblioteca como um modo simples de melhorar a estética visual das plotagens da matplotlib em geral.

Plotagens de linha

Tanto Series quanto DataFrame têm um atributo plot para criar alguns tipos básicos de plotagem. Por padrão, plot() cria plotagens de linha (veja a Figura 9.13):

```
In [60]: s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100,
10))

In [61]: s.plot()
```

O índice do objeto Series é passado para a matplotlib para plotagem no eixo x, embora você possa desativar isso especificando use_index=False. Os tiques e os limites do eixo x podem ser ajustados com as opções xticks e xlim, e os do eixo y respectivamente com yticks e ylim. Veja a Tabela 9.3 que apresenta uma lista completa das opções de plot. Farei outros comentários sobre mais algumas das opções ao longo desta seção e deixarei o restante para você explorar.

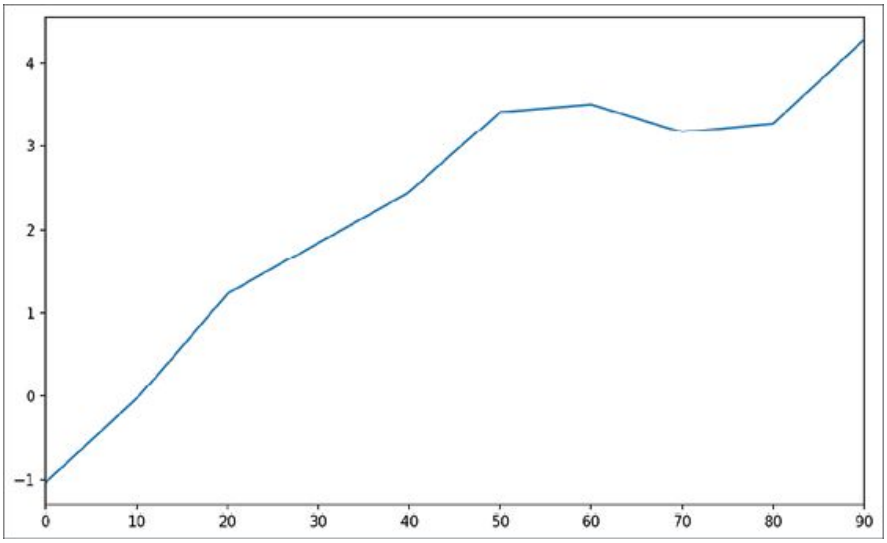


Figura 9-13 – Uma plotagem de uma Series simples.

A maioria dos métodos de plotagem do pandas aceita um parâmetro `ax` opcional, que pode ser um objeto de subplotagem da matplotlib. Isso permite um posicionamento mais flexível das subplotagens em um layout de grade.

O método `plot` de DataFrame plota cada uma de suas colunas como uma linha diferente na mesma subplotagem, criando uma legenda automaticamente (veja a Figura 9.14):

```
In [62]: df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),
   ....: columns=['A', 'B', 'C', 'D'],
   ....: index=np.arange(0, 100, 10))

In [63]: df.plot()
```

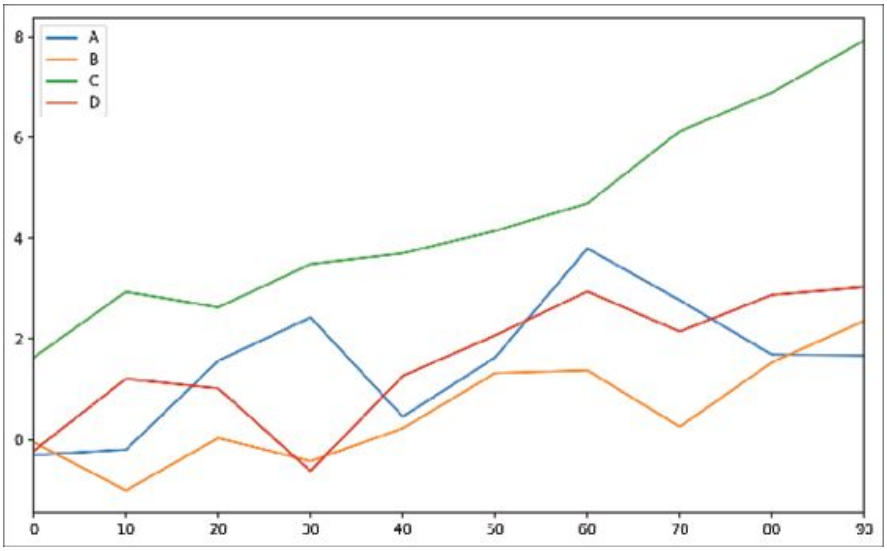


Figura 9.14 – Uma plotagem de um DataFrame simples.

O atributo plot contém uma "família" de métodos para diferentes tipos de plotagem. Por exemplo, df.plot() é equivalente a df.plot.line().Exploraremos alguns desses métodos a seguir.

Argumentos nomeados adicionais para plot são passados para a respectiva função de plotagem da matplotlib, de modo que você poderá personalizar melhor essas plotagens se conhecer melhor a API da matplotlib.

Tabela 9.3 – Argumentos do método Series.plot

Argumento	Descrição
label	Rótulo para a legenda da plotagem
ax	Objeto de plotagem da matplotlib para a plotagem; se nada for especificado, utiliza a subplotagem ativa da matplotlib
style	String de estilo, por exemplo, 'ko--', a ser passada para a matplotlib
alpha	A opacidade de preenchimento (de 0 a 1) da plotagem
kind	Pode ser 'area', 'bar', 'barh', 'density', 'hist', 'kde', 'line', 'pie'
logy	Usa escala logarítmica no eixo y
use_index	Usa o índice do objeto para rótulos dos tiques
rot	Rotação dos rótulos dos tiques (de 0 a 360)
xticks	Valores a serem usados como os tiques no eixo x
yticks	Valores a serem usados como os tiques no eixo y

Argumento	Descrição
xlim	Limites no eixo x (por exemplo, [0, 10])
ylim	Limites no eixo y
grid	Exibe grade do eixo (o default é ativo)

O DataFrame tem uma série de opções que permitem ter certa dose de flexibilidade quanto ao modo como as colunas são tratadas; por exemplo, se elas devem ser todas plotadas na mesma subplotagem ou se subplotagens diferentes devem ser criadas. Veja a Tabela 9.4 que apresenta mais informações sobre elas.

Tabela 9.4 – Argumentos de plotagem específicos para DataFrame

Argumento	Descrição
subplots	Plota cada coluna do DataFrame em uma subplotagem separada
sharex	Se subplots=True, compartilha o mesmo eixo x, ligando os tiques e os limites
sharey	Se subplots=True, compartilha o mesmo eixo y
figsize	Tamanho da figura a ser criada, na forma de uma tupla
title	Título da plotagem como uma string
legend	Adiciona uma legenda para a subplotagem (o default é True)
sort_columns	Plota colunas em ordem alfabética; por padrão, utiliza a ordem presente nas colunas

 Para plotagem de séries temporais, veja o Capítulo 11.

Plotagem de barras

plot.bar() e plot.barh() criam plotagens de barra vertical e horizontal, respectivamente. Nesse caso, o índice de Series ou de DataFrame será usado como os tiques de x (bar) ou de y (barh) – veja a Figura 9.15:

```
In [64]: fig, axes = plt.subplots(2, 1)

In [65]: data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))
```

```
In [66]: data.plot.bar(ax=axes[0], color='k', alpha=0.7)
Out[66]: <matplotlib.axes._subplots.AxesSubplot at 0x7f43500afc18>

In [67]: data.plot.barh(ax=axes[1], color='k', alpha=0.7)
```

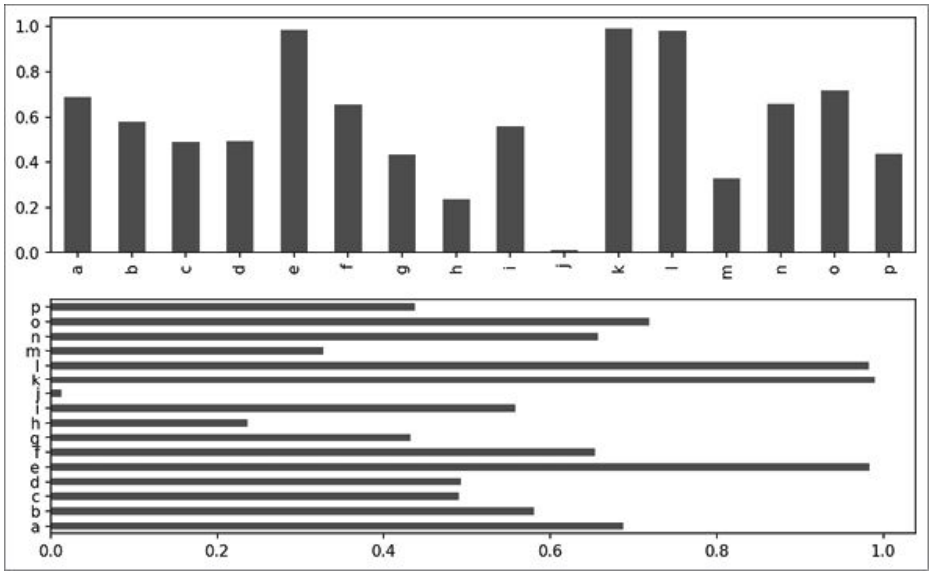


Figura 9.15 – Plotagem de barras horizontal e vertical.

As opções color='k' e alpha=0.7 definem a cor das plotagens como preto e o uso de transparência parcial para o preenchimento.

Com um DataFrame, as plotagens de barra agrupam os valores de cada linha em um grupo de barras, lado a lado, para cada valor. Veja a Figura 9.16:

```
In [69]: df = pd.DataFrame(np.random.rand(6, 4),
   ....: index=['one', 'two', 'three', 'four', 'five', 'six'],
   ....: columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))

In [70]: df
Out[70]:
Genus A B C D
one 0.370670 0.602792 0.229159 0.486744
two 0.420082 0.571653 0.049024 0.880592
three 0.814568 0.277160 0.880316 0.431326
four 0.374020 0.899420 0.460304 0.100843
five 0.433270 0.125107 0.494675 0.961825
six 0.601648 0.478576 0.205690 0.560547
```

In [71]: df.plot.bar()

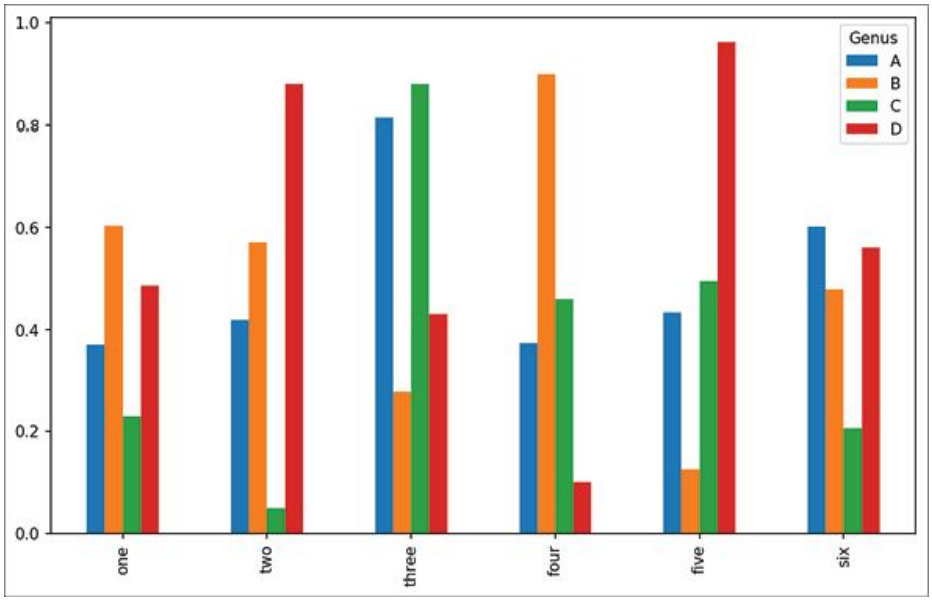


Figura 9.16 – Plotagem de barras para um DataFrame.

Observe que o nome “Genus” das colunas do DataFrame é usado como título da legenda.

Criamos plotagens de barras empilhadas a partir de um DataFrame passando stacked=True; como resultado, o valor de cada linha será empilhado (veja a Figura 9.17):

In [73]: df.plot.barh(stacked=True, alpha=0.5)

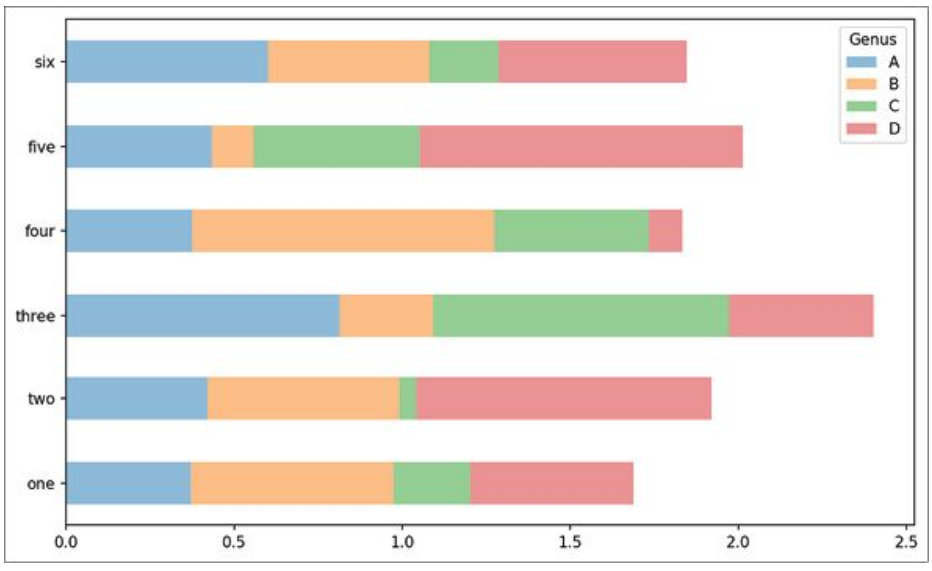


Figura 9.17 – Plotagem de barras empilhadas para um DataFrame.

 Uma receita útil para plotagens de barra consiste em visualizar a frequência de valores de uma Series usando value_counts: s.value_counts().plot.bar().

Voltando ao conjunto de dados de gorjetas usado antes no livro, suponha que quiséssemos gerar uma plotagem de barras empilhadas que exibisse o percentual de pontos de dados para cada tamanho de grupo a cada dia. Carregarei os dados usando read_csv e farei uma tabulação cruzada por dia e por tamanho de grupo:

```
In [75]: tips = pd.read_csv('examples/tips.csv')

In [76]: party_counts = pd.crosstab(tips['day'], tips['size'])

In [77]: party_counts
Out[77]:
size 1 2 3 4 5 6
day
Fri 1 16 1 1 0 0
Sat 2 53 18 13 1 0
Sun 0 39 15 18 3 1
Thur 1 48 4 5 1 3

# Não há muitos grupos de 1 e de 6 pessoas
In [78]: party_counts = party_counts.loc[:, 2:5]
```

Então farei uma normalização de modo que a soma de cada linha seja 1 e farei a plotagem (veja a Figura 9.18):

```
# Normalize para que a soma seja 1
In [79]: party_pcts = party_counts.div(party_counts.sum(1), axis=0)

In [80]: party_pcts
Out[80]:
size 2 3 4 5
day
Fri 0.888889 0.055556 0.055556 0.000000
Sat 0.623529 0.211765 0.152941 0.011765
Sun 0.520000 0.200000 0.240000 0.040000
Thur 0.827586 0.068966 0.086207 0.017241
```

```
In [81]: party_pcts.plot.bar()
```

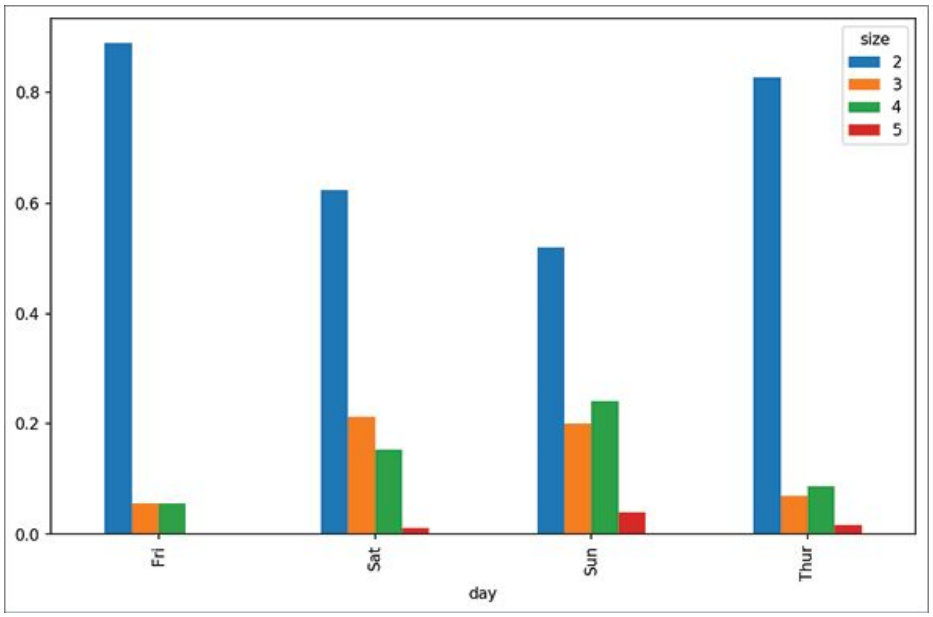


Figura 9.18 – Fração dos grupos por tamanho a cada dia.

Assim, podemos ver que os tamanhos dos grupos parecem aumentar no final de semana nesse conjunto de dados.

Com dados que exijam agregação ou resumo antes da criação de uma plotagem, usar o pacote seaborn pode simplificar bastante. Vamos ver agora o percentual de gorjetas por dia usando o seaborn (veja a Figura 9.19 que exibe a plotagem resultante):

```
In [83]: import seaborn as sns

In [84]: tips['tip_pct'] = tips['tip'] / (tips['total_bill'] - tips['tip'])

In [85]: tips.head()
Out[85]:
   total_bill tip smoker day time size tip_pct
0 16.99 1.01 No Sun Dinner 2 0.063204
1 10.34 1.66 No Sun Dinner 3 0.191244
2 21.01 3.50 No Sun Dinner 3 0.199886
3 23.68 3.31 No Sun Dinner 2 0.162494
4 24.59 3.61 No Sun Dinner 4 0.172069

In [86]: sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
```

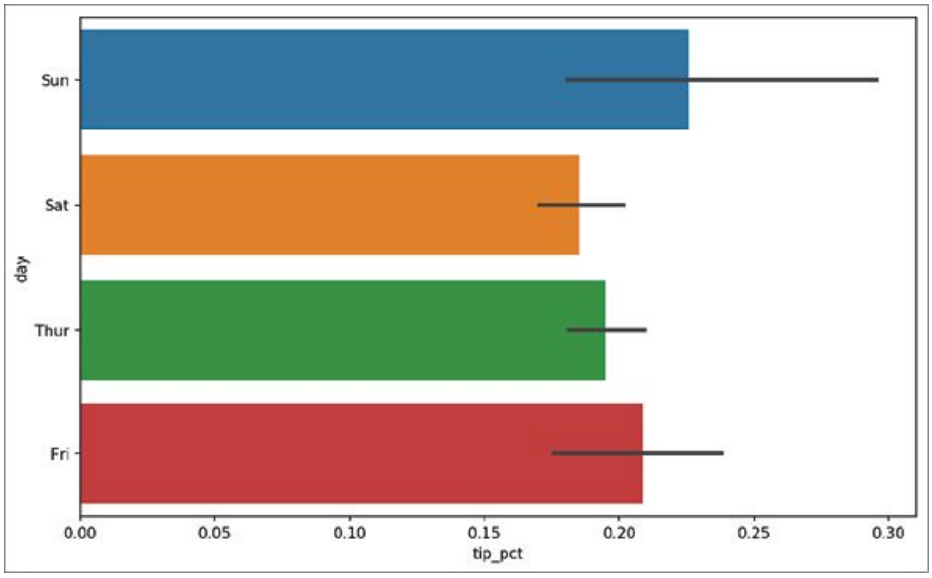


Figura 9.19 – Porcentagem de gorjetas por dia com barras de erro.

As funções de plotagem do seaborn aceitam um argumento data, que pode ser um DataFrame do pandas. Os outros argumentos referem-se aos nomes das colunas. Pelo fato de haver várias observações para cada valor em day, as barras são o valor médio de tip_pct. As linhas pretas desenhadas nas barras representam o intervalo de confiança de 95% (isso pode ser configurado com argumentos opcionais).

seaborn.barplot tem uma opção hue que nos permite fazer uma separação por meio de um valor adicional de categoria (Figura 9.20):

```
In [88]: sns.barplot(x='tip_pct', y='day', hue='time', data=tips, orient='h')
```

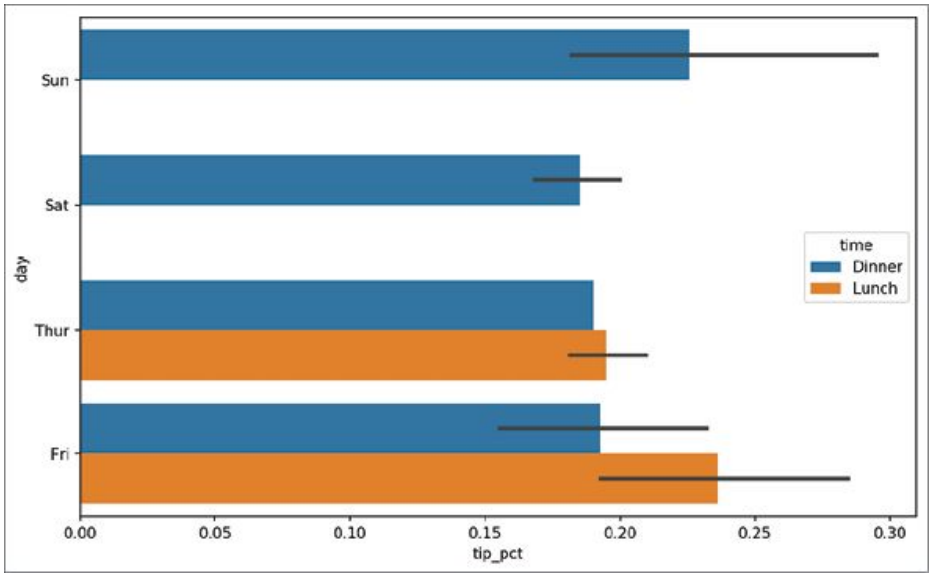


Figura 9.20 – Porcentagem de gorjetas por dia e hora.

Observe que o seaborn alterou automaticamente a estética das plotagens: a paleta de cores default, o plano de fundo da plotagem e as cores das linhas da grade. Podemos alternar entre diferentes aparências para a plotagem usando seaborn.set:

```
In [90]: sns.set(style="whitegrid")
```

Histogramas e plotagens de densidade

Um histograma é um tipo de plotagem de barras que oferece uma exibição discreta das frequências dos valores. Os pontos de dados são separados em compartimentos (bins) discretos, uniformemente espaçados, e o número de pontos de dados em cada compartimento é plotado. Usando os dados de gorjeta anteriores, podemos gerar um histograma das porcentagens de gorjetas sobre o total das contas usando o método plot.hist na Series (veja a Figura 9.21):

```
In [92]: tips['tip_pct'].plot.hist(bins=50)
```

Um tipo relacionado de plotagem é uma *plotagem de densidade*, formada pelo cálculo de uma estimativa de uma distribuição contínua de probabilidades que possa ter gerado os dados observados. O procedimento usual consiste em fazer uma aproximação dessa distribuição como uma mistura de “kernels” –

isto é, de distribuições mais simples, como a distribuição normal. Assim, as plotagens de densidade também são conhecidas como plotagens KDE (Kernel Density Estimate, ou Estimativa de Densidade Kernel).

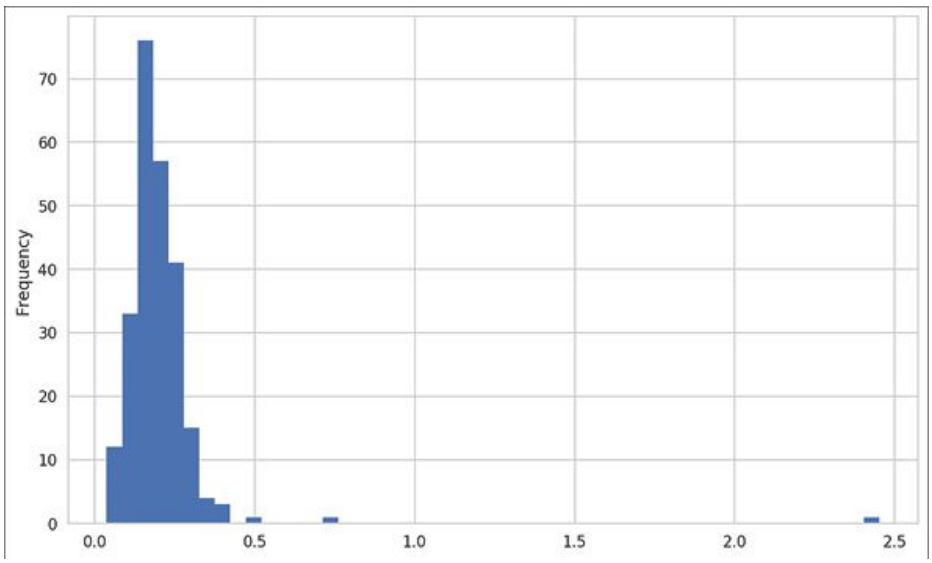


Figura 9.21 – Histograma das porcentagens de gorjeta.

O uso de plot.kde gera uma plotagem de densidade que utiliza a estimativa convencional de combinação-de-normais (veja a Figura 9.22):

```
In [94]: tips['tip_pct'].plot.density()
```

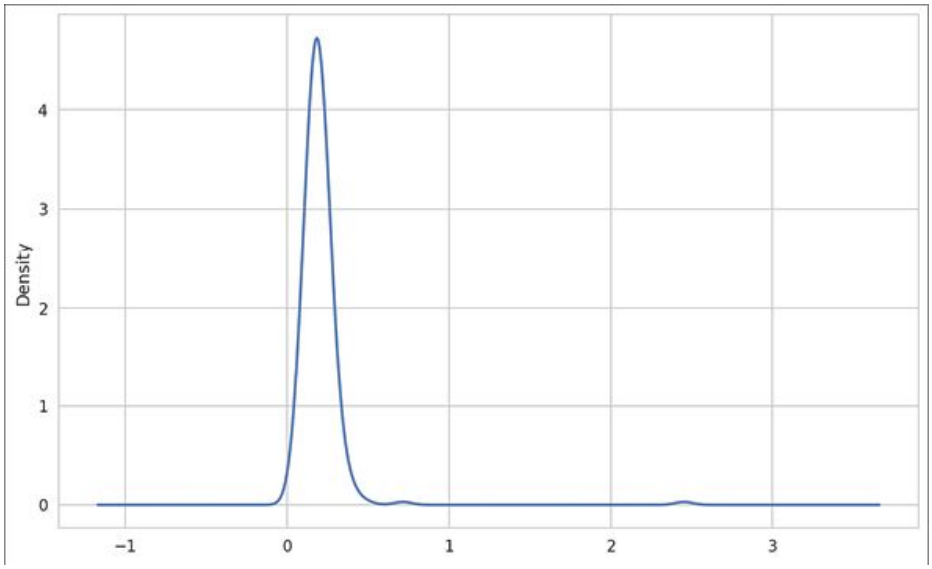


Figura 9.22 – Plotagem de densidade das porcentagens de gorjeta.

O seaborn gera histogramas e plotagens de densidade mais facilmente ainda com o método distplot, capaz de plotar tanto um histograma quanto uma estimativa de densidade contínua simultaneamente. Como exemplo, considere uma distribuição bimodal constituída de amostras de duas distribuições normais padrões distintas (veja a Figura 9.23):

```
In [96]: comp1 = np.random.normal(0, 1, size=200)

In [97]: comp2 = np.random.normal(10, 2, size=200)

In [98]: values = pd.Series(np.concatenate([comp1, comp2]))

In [99]: sns.distplot(values, bins=100, color='k')
```

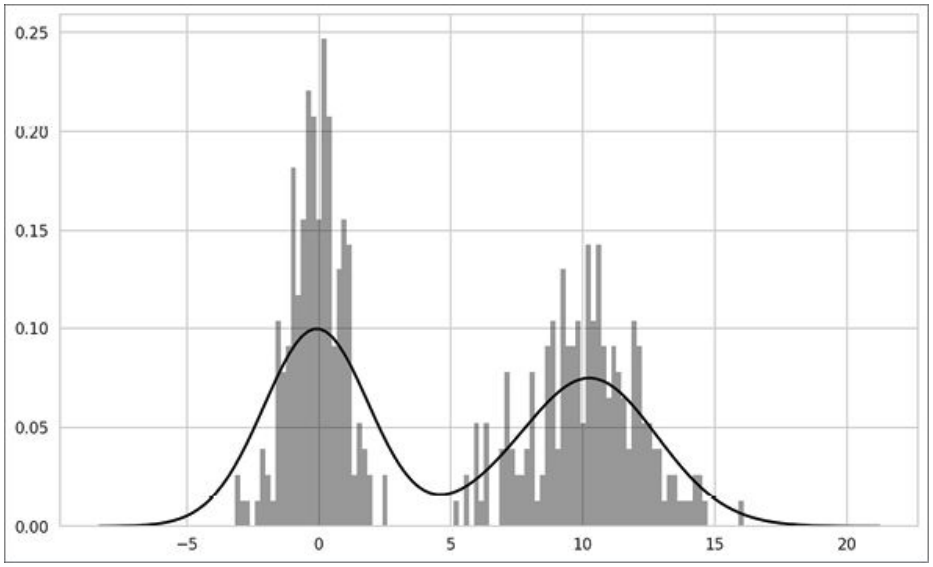


Figura 9.23 – Histograma normalizado de uma combinação de normais com estimativas de densidade.

Plotagens de dispersão ou de pontos

Plotagens de pontos ou de dispersão podem ser uma maneira conveniente de analisar o relacionamento entre duas séries de dados unidimensionais. Por exemplo, carregaremos a seguir o conjunto de dados macrodata do projeto statsmodels, selecionaremos algumas variáveis e então calcularemos as diferenças de log:

```
In [100]: macro = pd.read_csv('examples/macrodata.csv')
```

```
In [101]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]

In [102]: trans_data = np.log(data).diff().dropna()

In [103]: trans_data[-5:]
Out[103]:
          cpi m1 tbilrate unemp
198 -0.007904 0.045361 -0.396881 0.105361
199 -0.021979 0.066753 -2.277267 0.139762
200 0.002340 0.010286 0.606136 0.160343
201 0.008419 0.037461 -0.200671 0.127339
202 0.008894 0.012202 -0.405465 0.042560
```

Podemos então usar o método regplot do seaborn, que gera uma plotagem de dispersão e inclui um traço de regressão linear (veja a Figura 9.24):

```
In [105]: sns.regplot('m1', 'unemp', data=trans_data)
Out[105]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4341097cf8>

In [106]: plt.title('Changes in log %s versus log %s' % ('m1', 'unemp'))
```

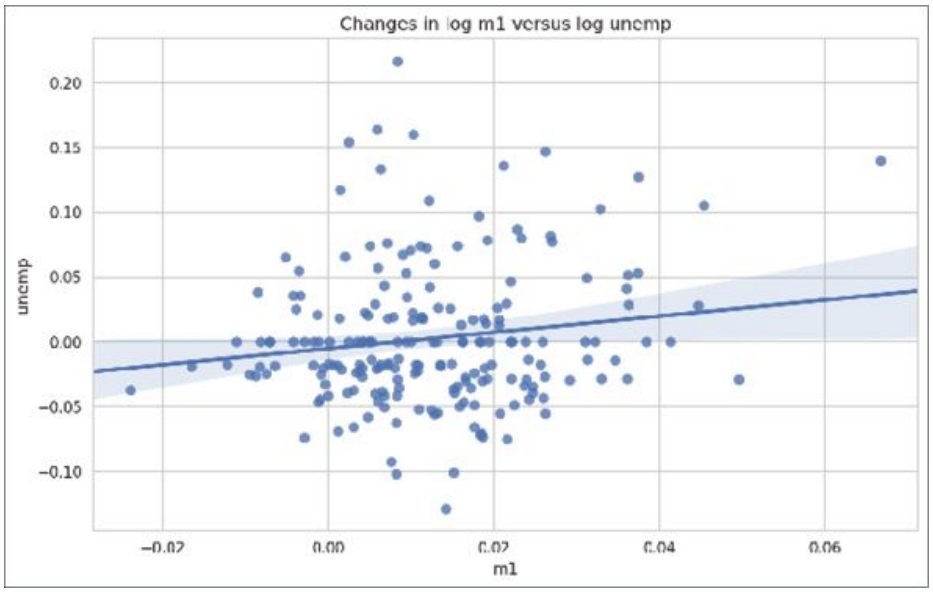


Figura 9.24 – Uma plotagem de regressão/dispersão do seaborn.

Na análise exploratória de dados, é conveniente observar todas as plotagens de dispersão entre um grupo de variáveis; isso é conhecido como *plotagem de pares* ou uma *matriz de plotagem de*

dispersão. Fazer uma plotagem como essa a partir do zero exige um pouco de trabalho; desse modo, o seaborn tem uma função pairplot conveniente, que aceita colocar histogramas ou estimativas de densidade de cada variável ao longo da diagonal (veja a Figura 9.25 que mostra a plotagem resultante):

```
In [107]: sns.pairplot(trans_data, diag_kind='kde', plot_kws={'alpha': 0.2})
```

Talvez você tenha notado o argumento plot_kws. Ele permite passar opções de configuração para as chamadas de plotagem individuais nos elementos fora da diagonal. Consulte a docstring de seaborn.pairplot para ver as opções de configuração mais específicas.

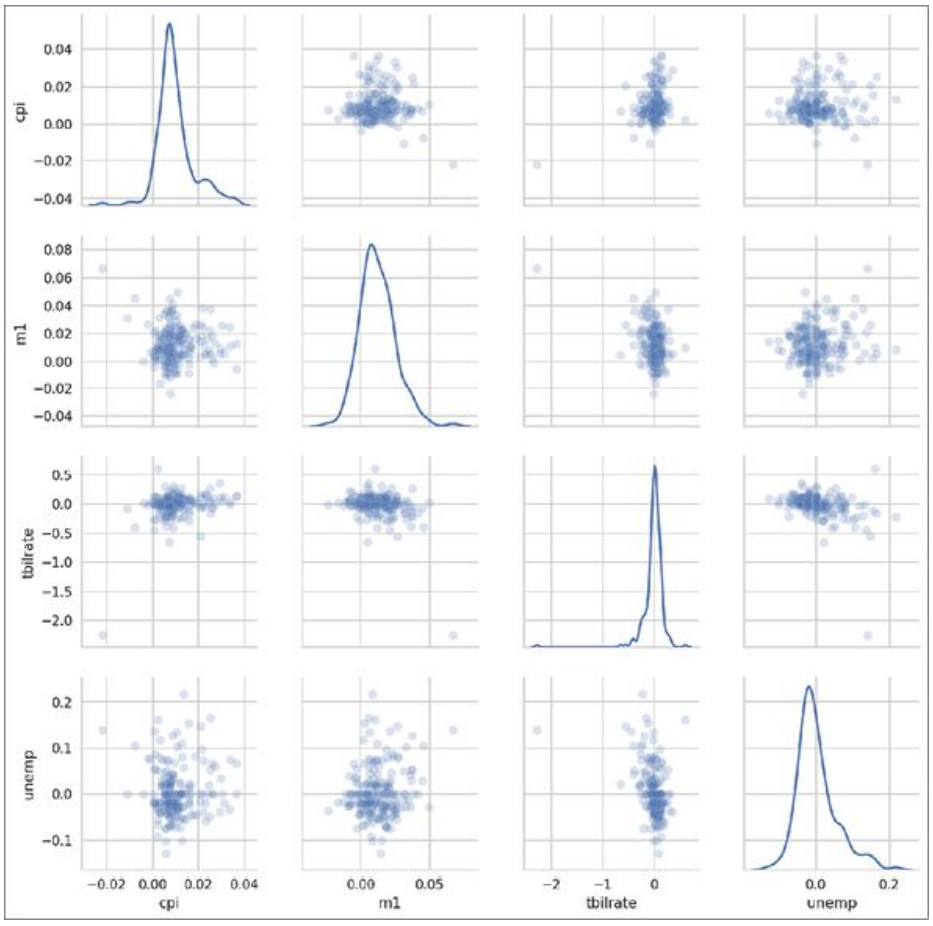


Figura 9.25 – Matriz de plotagens de pares dos dados macro de statsmodels.

Grades de faceta e dados de categoria

E quanto aos conjuntos de dados para os quais temos dimensões de agrupamento adicionais? Uma forma de visualizar dados com muitas variáveis de categoria é por meio de uma *grade de faceta* (facet grid). O seaborn tem uma função embutida factorplot conveniente, que simplifica a criação de vários tipos de plotagem com facetas (veja a Figura 9.26 que apresenta a plotagem resultante):

```
In [108]: sns.factorplot(x='day', y='tip_pct', hue='time', col='smoker',
   .....: kind='bar', data=tips[tips.tip_pct < 1])
```

Em vez de agrupar por 'time' por cores diferentes de barra em uma faceta, podemos também expandir a grade de facetas acrescentando uma linha por valor de time (Figura 9.27):

```
In [109]: sns.factorplot(x='day', y='tip_pct', row='time',
   .....: col='smoker',
   .....: kind='bar', data=tips[tips.tip_pct < 1])
```

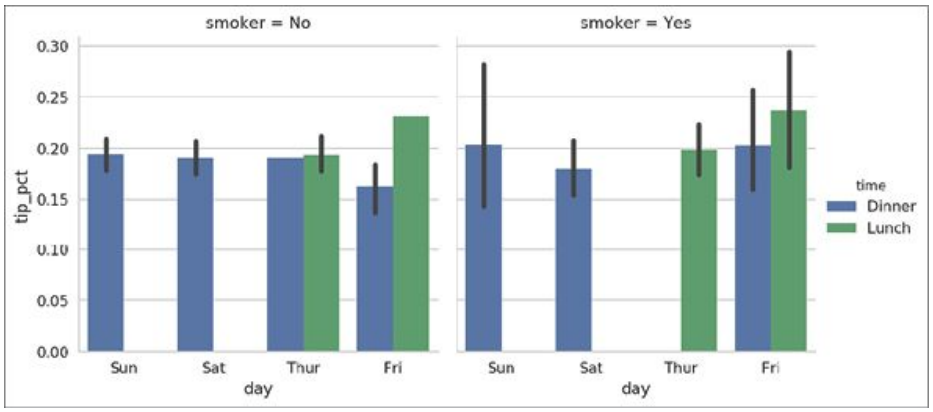


Figura 9.26 – Porcentagem de gorjetas por dia/hora/fumante.

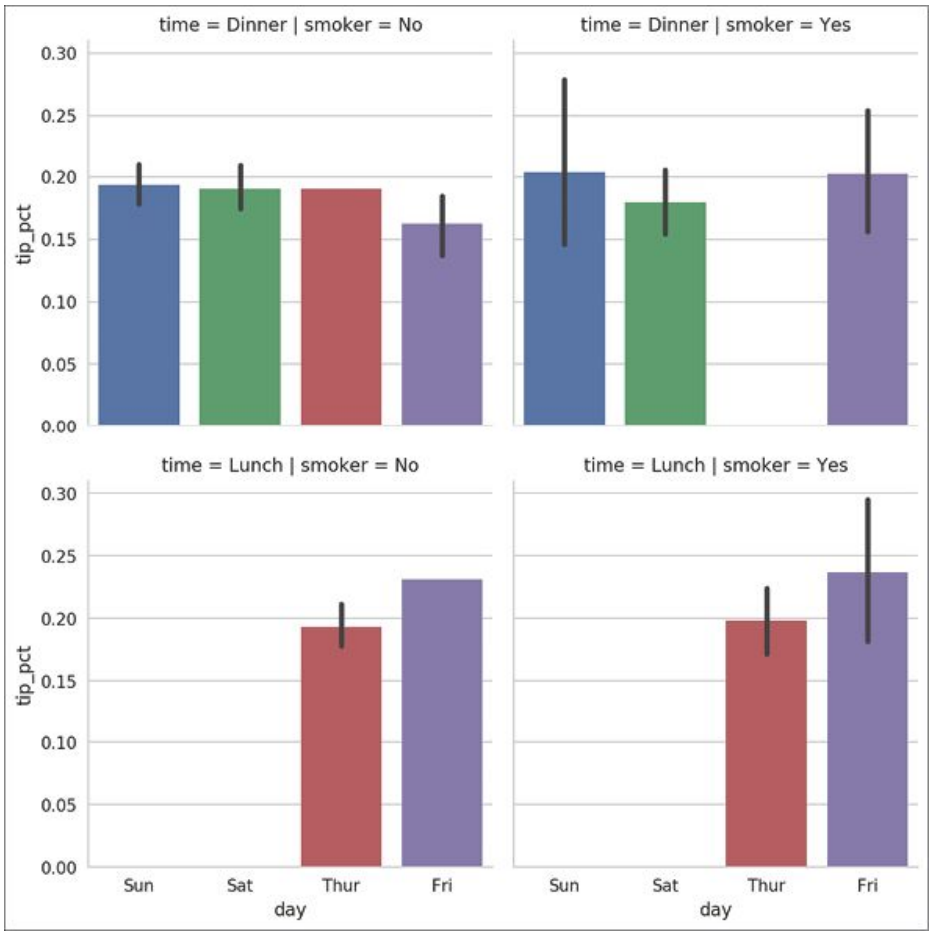


Figura 9.27. tip_pct por dia; faceta por hora/fumante.

factorplot aceita outros tipos de plotagem que podem ser úteis, dependendo do que você estiver tentando exibir. Por exemplo, plotagens de caixa (que mostram a mediana, os quartis e os valores discrepantes) podem ser um tipo de visualização eficaz (Figura 9.28):

```
In [110]: sns.factorplot(x='tip_pct', y='day', kind='box',
   .....: data=tips[tips.tip_pct < 0.5])
```

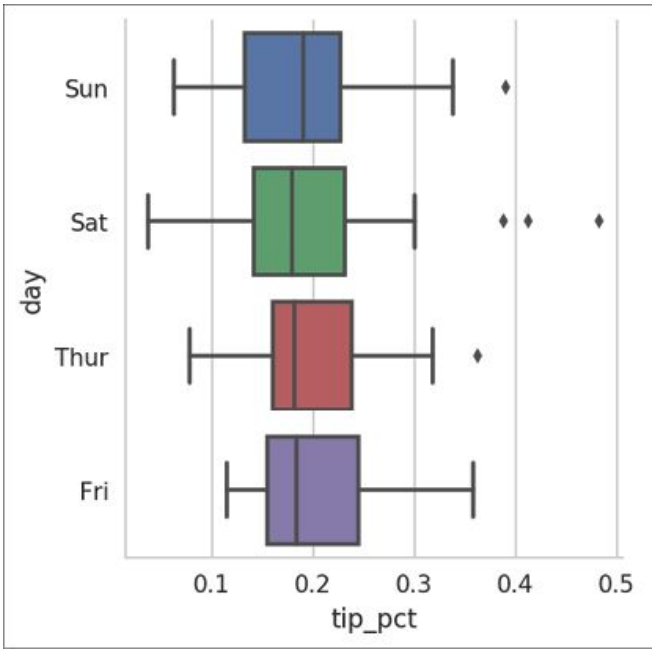


Figura 9.28 – Plotagem de caixa de tip_pct por dia.

Você pode criar suas próprias plotagens de grade de facetas usando a classe mais genérica seaborn.FacetGrid. Consulte a documentação do seaborn (*https://seaborn.pydata.org/*) para obter mais informações.

9.3 Outras ferramentas de visualização de Python

Como é comum no caso de códigos abertos, há uma variedade de opções para criar gráficos em Python (são tantos, a ponto de ser impossível listar todos). Desde 2010, muito esforço de desenvolvimento tem se concentrado na criação de gráficos interativos para publicação na web. Com ferramentas como Bokeh (*http://bokeh.pydata.org/*) e Plotly (*https://github.com/plotly/plotly.py*), atualmente é possível especificar gráficos dinâmicos e interativos em Python, visando a um navegador web.

Para criar gráficos estáticos para impressão ou para web, recomendo usar a matplotlib como default e bibliotecas add-on como o pandas e o seaborn para atender às suas necessidades. Para outros requisitos de visualização de dados, talvez seja conveniente conhecer uma das demais ferramentas disponíveis no mercado. Incentivo você a explorar o ecossistema à medida que ele

continua evoluindo e inovando em direção ao futuro.

9.4 Conclusão

O objetivo deste capítulo foi fazer uma apresentação inicial de algumas visualizações básicas de dados usando o pandas, a matplotlib e o seaborn. Se fazer uma comunicação visual dos resultados da análise de dados for importante em seu trabalho, incentivo você a pesquisar recursos para conhecer melhor o modo de obter uma visualização de dados eficaz. É um campo de pesquisa ativo, e você pode praticar com muitos recursos de aprendizado excelentes disponíveis online e em formato impresso.

No próximo capítulo, voltaremos a nossa atenção para a agregação de dados e as operações de grupo com o pandas.

CAPÍTULO 10

Agregação de dados e operações em grupos

Classificar um conjunto de dados e aplicar uma função a cada grupo, seja uma agregação ou uma transformação, com frequência é um componente essencial em um fluxo de trabalho de análise de dados. Após carregar, mesclar e preparar um conjunto de dados, talvez seja necessário calcular estatísticas de grupos ou, possivelmente, *tabelas pivôs* visando a relatórios e visualizações. O pandas oferece uma interface flexível groupby, que permite manipular e resumir conjuntos de dados de forma natural.

Um dos motivos para a popularidade dos bancos de dados relacionais e do SQL (que quer dizer “Structured Query Language”, ou Linguagem de Consulta Estruturada) é a facilidade com a qual os dados podem ser unidos, filtrados, transformados e agregados. No entanto, as linguagens de consulta como SQL, de certo modo, são limitadas quanto aos tipos de operações em grupo que podem executar. Conforme veremos, com a expressividade de Python e do pandas, podemos executar operações bem complexas em grupos utilizando qualquer função que aceite um objeto do pandas ou um array NumPy. Neste capítulo, você aprenderá a:

- separar um objeto do pandas em partes usando uma ou mais chaves (na forma de funções, arrays ou nomes de colunas de um DataFrame);
- calcular estatísticas de resumo para grupos, como contador, média ou desvio-padrão, ou aplicar uma função definida pelo usuário;

- aplicar transformações em grupos ou fazer outras manipulações como normalização, regressão linear, classificação ou seleção de subconjuntos;
- calcular tabelas pivôs e tabulações cruzadas;
- fazer análise de quantis e outras análises estatísticas de grupos.

A agregação de dados de séries temporais – um caso de uso especial de groupby – é chamada de *reamostragem* (resampling) neste livro, e terá um tratamento à parte no Capítulo 11.

10.1 Funcionamento de GroupBy

Hadley Wickham, autor de vários pacotes bem populares para a linguagem de programação R, cunhou o termo *separar-aplicar-combinar* (split-apply-combine) para descrever operações de grupo. Na primeira etapa do processo, os dados contidos em um objeto do pandas, seja uma Series, um DataFrame ou algo diferente, são *separados* (split) em grupos, com base em uma ou mais *chaves* especificadas por você. A separação é feita em um eixo em particular de um objeto. Por exemplo, um DataFrame pode ser agrupado com base em suas linhas (axis=0) ou suas colunas (axis=1). Depois disso, uma função é *aplicada* (apply) em cada grupo, gerando um novo valor. Por fim, os resultados de todas essas aplicações de função são *combinados* (combine) formando um objeto resultante. O formato desse objeto em geral dependerá do que está sendo feito com os dados. Veja a Figura 10.1 que apresenta um esquema de uma agregação simples em grupos.

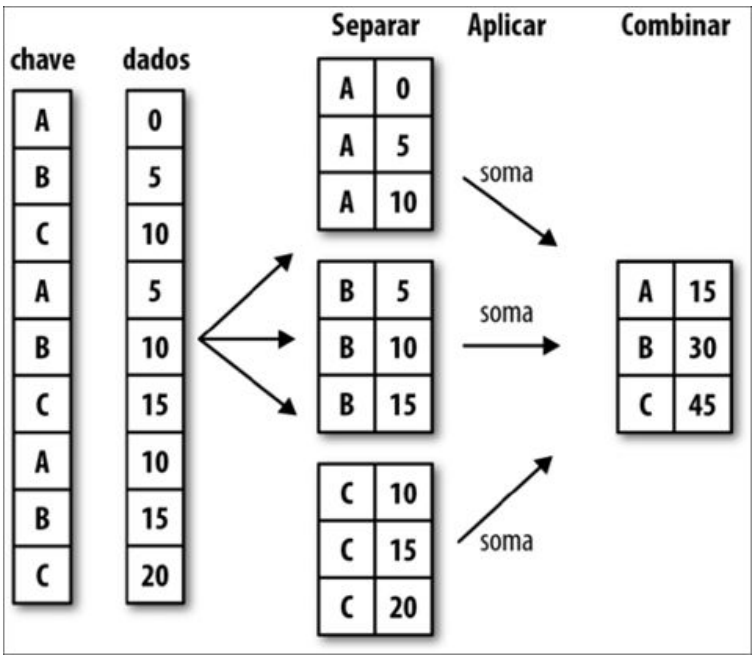


Figura 10.1 – Demonstração de uma agregação em grupos.

Cada chave de grupo pode assumir diversas formas, e as chave não precisam ser todas do mesmo tipo:

- uma lista ou um array de valores de mesmo tamanho que o eixo sendo agrupado;
- um valor indicando um nome de coluna em um DataFrame;
- um dicionário ou uma Series especificando uma correspondência entre os valores do eixo sendo agrupado e os nomes dos grupos;
- uma função a ser chamada no índice do eixo ou os rótulos individuais no índice.

Observe que os últimos três métodos são atalhos para gerar um array de valores a ser usado para separar o objeto. Não se preocupe se tudo isso parecer abstrato demais. No decorrer deste capítulo, apresentarei vários exemplos de todos esses métodos. Para começar, eis um pequeno conjunto de dados tabular na forma de um DataFrame:

```
In [10]: df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
   ....: 'key2' : ['one', 'two', 'one', 'two', 'one'],
   ....: 'data1' : np.random.randn(5),
   ....: 'data2' : np.random.randn(5)})
```

```
In [11]: df
Out[11]:
     data1 data2 key1 key2
0 -0.204708 1.393406 a one
1 0.478943 0.092908 a two
2 -0.519439 0.281746 b one
3 -0.555730 0.769023 b two
4 1.965781 1.246435 a one
```

Suponha que quiséssemos calcular a média da coluna data1 usando os rótulos de key1. Há várias maneiras de fazer isso. Uma delas é acessar data1 e chamar groupby com a coluna (uma Series) em key1:

```
In [12]: grouped = df['data1'].groupby(df['key1'])

In [13]: grouped
Out[13]: <pandas.core.groupby.SeriesGroupBy object at 0x7f85008d0400>
```

Essa variável grouped agora é um objeto *GroupBy*. Na verdade, nada ainda foi calculado, exceto alguns dados intermediários sobre a chave de grupo df['key1']. A ideia é que esse objeto tenha todas as informações necessárias para então aplicar alguma operação em cada um dos grupos. Por exemplo, para calcular as médias dos grupos, podemos chamar o método mean de GroupBy:

```
In [14]: grouped.mean()
Out[14]:
key1
a 0.746672
b -0.537585
Name: data1, dtype: float64
```

Mais adiante, explicarei melhor o que acontece quando chamamos .mean().O aspecto importante, nesse caso, é que os dados (uma Series) foram agregados de acordo com a chave de grupo, gerando uma nova Series, que agora está indexada pelos valores únicos da coluna key1. O índice resultante recebe o nome 'key1' por causa da coluna df['key1'] do DataFrame.

Se, em vez disso, tivéssemos passado vários arrays na forma de uma lista, obteríamos um resultado diferente:

```
In [15]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()

In [16]: means
Out[16]:
key1 key2
a one 0.880536
     two 0.478943
b one -0.519439
     two -0.555730
Name: data1, dtype: float64
```

Nesse caso, agrupamos os dados usando duas chaves, e a Series resultante agora tem um índice hierárquico constituído dos pares de chave únicos observados:

```
In [17]: means.unstack()
Out[17]:
key2 one two
key1
a 0.880536 0.478943
b -0.519439 -0.555730
```

No exemplo a seguir, todas as chaves de grupo são Series, embora pudessem ser qualquer array do tamanho correto:

```
In [18]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])

In [19]: years = np.array([2005, 2005, 2006, 2005, 2006])

In [20]: df['data1'].groupby([states, years]).mean()
Out[20]:
California 2005 0.478943
           2006 -0.519439
Ohio 2005 -0.380219
           2006 1.965781
Name: data1, dtype: float64
```

Com frequência, as informações de agrupamento se encontram no mesmo DataFrame em que estão os dados com os quais você quer trabalhar. Nesse caso, é possível passar os nomes das colunas (sejam elas strings, números ou outros objetos Python) como as chaves de grupo:

```
In [21]: df.groupby('key1').mean()
Out[21]:
         data1 data2
key1
a 0.746672 0.910916
b -0.537585 0.525384

In [22]: df.groupby(['key1', 'key2']).mean()
Out[22]:
              data1 data2
key1 key2
a one 0.880536 1.319920
     two 0.478943 0.092908
b one -0.519439 0.281746
     two -0.555730 0.769023
```

Talvez você tenha percebido que, no primeiro caso com `df.groupby('key1').mean()`, não há nenhuma coluna `key2` no resultado. Como `df['key2']` não é um dado numérico, dizemos que é uma *coluna inconveniente* (nuisance column) e, desse modo, é excluída do resultado. Por padrão, todas as colunas numéricas são agregadas, embora seja possível filtrar e obter um subconjunto, como veremos em breve.

Independentemente do objetivo visado ao usar `groupby`, um método em geral útil de GroupBy é `size`, que devolve uma Series contendo os tamanhos dos grupos:

```
In [23]: df.groupby(['key1', 'key2']).size()
Out[23]:
key1 key2
a one 2
     two 1
b one 1
     two 1
dtype: int64
```

Perceba que qualquer valor ausente em uma chave de grupo será excluído do resultado.

Iterando por grupos

O objeto GroupBy aceita iteração, gerando uma sequência de tuplas de 2 contendo o nome do grupo, junto com a porção de dados. Considere o seguinte:

```
In [24]: for name, group in df.groupby('key1'):
   ....: print(name)
   ....: print(group)
   ....:
a
      data1 data2 key1 key2
0 -0.204708 1.393406 a one
1 0.478943 0.092908 a two
4 1.965781 1.246435 a one
b
      data1 data2 key1 key2
2 -0.519439 0.281746 b one
3 -0.555730 0.769023 b two
```

No caso de várias chaves, o primeiro elemento da tupla será uma tupla de valores de chaves:

```
In [25]: for (k1, k2), group in df.groupby(['key1', 'key2']):
   ....: print((k1, k2))
   ....: print(group)
   ....:
('a', 'one')
      data1 data2 key1 key2
0 -0.204708 1.393406 a one
4 1.965781 1.246435 a one
('a', 'two')
      data1 data2 key1 key2
1 0.478943 0.092908 a two
('b', 'one')
      data1 data2 key1 key2
2 -0.519439 0.281746 b one
('b', 'two')
     data1 data2 key1 key2
3 -0.55573 0.769023 b two
```

É claro que você pode optar por fazer o que quiser com as porções de dados. Uma receita que talvez seja útil é gerar um dicionário de porções de dados usando uma só linha de código:

```
In [26]: pieces = dict(list(df.groupby('key1')))

In [27]: pieces['b']
Out[27]:
      data1 data2 key1 key2
2 -0.519439 0.281746 b one
3 -0.555730 0.769023 b two
```

Por padrão, groupby agrupa em axis=0, mas podemos agrupar em qualquer um dos outros eixos. Por exemplo, é possível agrupar as colunas de nosso exemplo com df de acordo com dtype, assim:

```
In [28]: df.dtypes
Out[28]:
data1 float64
data2 float64
key1 object
key2 object
dtype: object

In [29]: grouped = df.groupby(df.dtypes, axis=1)
```

Podemos exibir os grupos da seguinte maneira:

```
In [30]: for dtype, group in grouped:
   ....: print(dtype)
   ....: print(group)
   ....:
float64
      data1 data2
0 -0.204708 1.393406
1 0.478943 0.092908
2 -0.519439 0.281746
3 -0.555730 0.769023
4 1.965781 1.246435
object
  key1 key2
0 a one
1 a two
2 b one
3 b two
4 a one
```

Selecionando uma coluna ou um subconjunto de colunas

Indexar um objeto GroupBy criado a partir de um DataFrame com um nome de coluna ou um array de nomes de coluna tem o efeito de criar subconjuntos de colunas para agregação. Isso significa que:

```
df.groupby('key1')['data1']
df.groupby('key1')[['data2']]
```

são um açúcar sintático para:

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

Particularmente para conjuntos grandes de dados, fazer agregações somente de algumas colunas pode ser desejável. Por exemplo, no conjunto de dados anterior, para calcular as médias apenas da coluna data2 e obter o resultado na forma de um DataFrame, poderíamos escrever o seguinte:

```
In [31]: df.groupby(['key1', 'key2'])[['data2']].mean()
Out[31]:
              data2
key1 key2
a one 1.319920
     two 0.092908
b one 0.281746
     two 0.769023
```

O objeto devolvido por essa operação de indexação é um DataFrame agrupado se uma lista ou um array for passado ou uma Series agrupada se um único nome de coluna for passado como um escalar:

```
In [32]: s_grouped = df.groupby(['key1', 'key2'])['data2']

In [33]: s_grouped
Out[33]: <pandas.core.groupby.SeriesGroupBy object at 0x7f85008983c8>

In [34]: s_grouped.mean()
Out[34]:
key1 key2
```

```
a one 1.319920
     two 0.092908
b one 0.281746
     two 0.769023
Name: data2, dtype: float64
```

Agrupando com dicionários e Series

Informações de agrupamento podem existir em uma forma que não seja um array. Vamos considerar outro DataFrame como exemplo:

```
In [35]: people = pd.DataFrame(np.random.randn(5, 5),
   ....: columns=['a', 'b', 'c', 'd', 'e'],
   ....: index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])

In [36]: people.iloc[2:3, [1, 2]] = np.nan # Acrescenta alguns valores NA

In [37]: people
Out[37]:
               a b c d e
Joe 1.007189 -1.296221 0.274992 0.228913 1.352917
Steve 0.886429 -2.001637 -0.371843 1.669025 -0.438570
Wes -0.539741 NaN NaN -1.021228 -0.577087
Jim 0.124121 0.302614 0.523772 0.000940 1.343810
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Suponha agora que haja uma correspondência de grupos para as colunas e queremos somá-las por grupo:

```
In [38]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
   ....: 'd': 'blue', 'e': 'red', 'f' : 'orange'}
```

Poderíamos construir um array a partir desse dicionário e passá-lo para groupby, mas, em vez disso, podemos simplesmente passar o dicionário (incluí a chave 'f' para enfatizar que chaves de agrupamento não usadas não são um problema):

```
In [39]: by_column = people.groupby(mapping, axis=1)

In [40]: by_column.sum()
Out[40]:
          blue red
Joe 0.503905 1.063885
```

```
Steve 1.297183 -1.553778
Wes -1.021228 -1.116829
Jim 0.524712 1.770545
Travis -4.230992 -2.405455
```

A mesma funcionalidade vale para Series, que pode ser vista como um mapeamento de tamanho fixo:

```
In [41]: map_series = pd.Series(mapping)

In [42]: map_series
Out[42]:
a red
b red
c blue
d blue
e red
f orange
dtype: object

In [43]: people.groupby(map_series, axis=1).count()
Out[43]:
       blue red
Joe 2 3
Steve 2 3
Wes 1 2
Jim 2 3
Travis 2 3
```

Agrupando com funções

Usar funções Python é uma forma mais genérica de definir um mapeamento de grupos, em comparação com um dicionário ou uma Series. Qualquer função passada como uma chave de grupo será chamada uma vez por valor de índice, com os valores de retorno usados como os nomes dos grupos. De modo mais concreto, considere o DataFrame de exemplo da seção anterior, que tem os primeiros nomes das pessoas como valores de índice. Suponha que quiséssemos agrupar pelo tamanho dos nomes; embora pudéssemos calcular um array de tamanhos de strings, será mais

fácil simplesmente passar a função len:

```
In [44]: people.groupby(len).sum()
Out[44]:
          a b c d e
3 0.591569 -0.993608 0.798764 -0.791374 2.119639
5 0.886429 -2.001637 -0.371843 1.669025 -0.438570
6 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Misturar funções com arrays, dicionários ou Series não é um problema, pois tudo será convertido para arrays internamente:

```
In [45]: key_list = ['one', 'one', 'one', 'two', 'two']

In [46]: people.groupby([len, key_list]).min()
Out[46]:
              a b c d e
3 one -0.539741 -1.296221 0.274992 -1.021228 -0.577087
  two 0.124121 0.302614 0.523772 0.000940 1.343810
5 one 0.886429 -2.001637 -0.371843 1.669025 -0.438570
6 two -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Agrupando por níveis de índice

Um último recurso conveniente para conjuntos de dados hierarquicamente indexados é a capacidade de agregar usando um dos níveis de índice de um eixo. Vamos observar um exemplo:

```
In [47]: columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
   ....: [1, 3, 5, 1, 3]],
   ....: names=['cty', 'tenor'])

In [48]: hier_df = pd.DataFrame(np.random.randn(4, 5), columns=columns)

In [49]: hier_df
Out[49]:
cty US JP
tenor 1 3 5 1 3
0 0.560145 -1.265934 0.119827 -1.063512 0.332883
1 -2.359419 -0.199543 -1.541996 -0.970736 -1.307030
2 0.286350 0.377984 -0.753887 0.331286 1.349742
3 0.069877 0.246674 -0.011862 1.004812 1.327195
```

Para agrupar por nível, passe o número ou o nome do nível usando o argumento nomeado level:

```
In [50]: hier_df.groupby(level='cty', axis=1).count()
Out[50]:
cty JP US
0 2 3
1 2 3
2 2 3
3 2 3
```

10.2 Agregação de dados

As agregações referem-se a qualquer transformação de dados que gere valores escalares a partir de arrays. Os exemplos anteriores usaram várias delas, incluindo mean, count, min e sum. Talvez você esteja se perguntando o que acontece quando chamamos mean() em um objeto GroupBy. Muitas agregações comuns, como aquelas que se encontram na Tabela 10.1, têm implementações otimizadas. Contudo não estamos limitados a apenas esse conjunto de métodos.

Tabela 10.1 – Métodos otimizados de groupby

Nome da função	Descrição
count	Número de valores diferentes de NA no grupo
sum	Soma dos valores diferentes de NA
mean	Média dos valores diferentes de NA
median	Mediana aritmética dos valores diferentes de NA
std, var	Desvio-padrão não tendencioso (denominador n – 1) e variância
min, max	Mínimo e máximo entre os valores diferentes de NA
prod	Produto dos valores diferentes de NA
first, last	Primeiro e último dos valores diferentes de NA

Você pode usar agregações definidas por você mesmo e, além disso, chamar qualquer método que também esteja definido no objeto agrupado. Por exemplo, talvez você se lembre de que quantile

calcula os quantis de amostragem em uma Series ou em colunas de um DataFrame.

Embora quantile não seja explicitamente implementado para GroupBy, é um método de Series e, desse modo, está disponível para ser usado. Internamente, GroupBy fatia a Series de modo eficiente, chama piece.quantile(0.9) para cada parte e então reúne os resultados no objeto resultante:

```
In [51]: df
Out[51]:
      data1 data2 key1 key2
0 -0.204708 1.393406 a one
1 0.478943 0.092908 a two
2 -0.519439 0.281746 b one
3 -0.555730 0.769023 b two
4 1.965781 1.246435 a one

In [52]: grouped = df.groupby('key1')

In [53]: grouped['data1'].quantile(0.9)
Out[53]:
key1
a 1.668413
b -0.523068
Name: data1, dtype: float64
```

Para usar suas próprias funções de agregação, passe qualquer função que agregue um array para o método aggregate ou agg:

```
In [54]: def peak_to_peak(arr):
   ....: return arr.max() - arr.min()

In [55]: grouped.agg(peak_to_peak)
Out[55]:
        data1 data2
key1
a 2.170488 1.300498
b 0.036292 0.487276
```

Você pode notar que alguns métodos como describe também funcionam, embora não sejam de agregação, estritamente falando:

```
In [56]: grouped.describe()
Out[56]:
    data1 \
    count mean std min 25% 50% 75%
key1
a 3.0 0.746672 1.109736 -0.204708 0.137118 0.478943 1.222362
b 2.0 -0.537585 0.025662 -0.555730 -0.546657 -0.537585 -0.528512
            data2 \
          max count mean std min 25% 50%
key1
a 1.965781 3.0 0.910916 0.712217 0.092908 0.669671 1.246435
b -0.519439 2.0 0.525384 0.344556 0.281746 0.403565 0.525384

          75% max
key1
a 1.319920 1.393406
b 0.647203 0.769023
```

Explicarei o que aconteceu nesse caso, de modo mais detalhado, na Seção 10.3: “O método apply: separar-aplicar-combinar genérico”.

Funções personalizadas de agregação em geral são bem mais lentas que as funções otimizadas que se encontram na Tabela 10.1. Isso ocorre porque há certo overhead extra (chamadas de função, reorganização de dados) na construção das porções intermediárias dos dados de grupo.

Aplicação de função nas colunas e aplicação de várias funções

Vamos voltar ao conjunto de dados de gorjetas de exemplos anteriores. Depois de carregá-lo com read_csv, adicionamos uma coluna tip_pct de porcentagem de gorjetas:

```
In [57]: tips = pd.read_csv('examples/tips.csv')

# Acrescenta a porcentagem de gorjeta sobre o total da conta
In [58]: tips['tip_pct'] = tips['tip'] / tips['total_bill']

In [59]: tips[:6]
Out[59]:
```

```
   total_bill tip smoker day time size tip_pct
0 16.99 1.01 No Sun Dinner 2 0.059447
1 10.34 1.66 No Sun Dinner 3 0.160542
2 21.01 3.50 No Sun Dinner 3 0.166587
3 23.68 3.31 No Sun Dinner 2 0.139780
4 24.59 3.61 No Sun Dinner 4 0.146808
5 25.29 4.71 No Sun Dinner 4 0.186240
```

Conforme já vimos, fazer a agregação em uma Series ou em todas as colunas de um DataFrame é uma questão de usar aggregate com a função desejada ou chamar um método como mean ou std. Entretanto, talvez você queira fazer a agregação usando uma função diferente, conforme a coluna, ou utilizando várias funções de uma só vez. Felizmente isso é possível, e farei uma demonstração por meio de uma série de exemplos. Inicialmente agruparei tips de acordo com day e smoker:

```
In [60]: grouped = tips.groupby(['day', 'smoker'])
```

Observe que, para estatísticas descritivas como aquelas da Tabela 10.1, podemos passar o nome da função como uma string:

```
In [61]: grouped_pct = grouped['tip_pct']

In [62]: grouped_pct.agg('mean')
Out[62]:
day smoker
Fri No 0.151650
    Yes 0.174783
Sat No 0.158048
    Yes 0.147906
Sun No 0.160113
    Yes 0.187250
Thur No 0.160298
    Yes 0.163863
Name: tip_pct, dtype: float64
```

Se uma lista de funções ou de nomes de função for especificada, você terá de volta um DataFrame com os nomes das colunas obtidos com base nas funções:

```
In [63]: grouped_pct.agg(['mean', 'std', peak_to_peak])
```

```
Out[63]:
                  mean std peak_to_peak
day smoker
Fri No 0.151650 0.028123 0.067349
    Yes 0.174783 0.051293 0.159925
Sat No 0.158048 0.039767 0.235193
    Yes 0.147906 0.061375 0.290095
Sun No 0.160113 0.042347 0.193226
    Yes 0.187250 0.154134 0.644685
Thur No 0.160298 0.038774 0.193350
    Yes 0.163863 0.039389 0.151240
```

Nesse exemplo, passamos uma lista de funções de agregação para agg a fim de serem avaliadas de modo independente nos grupos de dados.

Você não precisa aceitar os nomes que GroupBy dá às colunas; merece destaque o fato de as funções lambda terem o nome '<lambda>', o que as deixa difíceis de serem identificadas (você pode ver por si mesmo observando o atributo __name__ de uma função). Desse modo, se você passar uma lista de tuplas (name, function), o primeiro elemento de cada tupla será usado como os nomes das colunas do DataFrame (podemos pensar em uma lista de tuplas de 2 elementos como um mapeamento ordenado):

```
In [64]: grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
Out[64]:
                   foo bar
day smoker
Fri No 0.151650 0.028123
    Yes 0.174783 0.051293
Sat No 0.158048 0.039767
    Yes 0.147906 0.061375
Sun No 0.160113 0.042347
    Yes 0.187250 0.154134
Thur No 0.160298 0.038774
    Yes 0.163863 0.039389
```

Com um DataFrame, temos mais opções, pois podemos especificar uma lista de funções a serem aplicadas em todas as colunas, ou diferentes funções por coluna. Para começar, suponha que

quiséssemos calcular as mesmas três estatísticas para as colunas tip_pct e total_bill:

```
In [65]: functions = ['count', 'mean', 'max']

In [66]: result = grouped['tip_pct', 'total_bill'].agg(functions)

In [67]: result
Out[67]:
             tip_pct total_bill
               count mean max count mean max
day smoker
Fri No 4 0.151650 0.187735 4 18.420000 22.75
    Yes 15 0.174783 0.263480 15 16.813333 40.17
Sat No 45 0.158048 0.291990 45 19.661778 48.33
    Yes 42 0.147906 0.325733 42 21.276667 50.81
Sun No 57 0.160113 0.252672 57 20.506667 48.17
    Yes 19 0.187250 0.710345 19 24.120000 45.35
Thur No 45 0.160298 0.266312 45 17.113111 41.19
    Yes 17 0.163863 0.241255 17 19.190588 43.11
```

Como podemos ver, o DataFrame resultante tem colunas hierárquicas – as mesmas que obteríamos se agregássemos cada coluna separadamente e usássemos concat para unir os resultados, utilizando os nomes das colunas como o argumento keys:

```
In [68]: result['tip_pct']
Out[68]:
             count mean max
day smoker
Fri No 4 0.151650 0.187735
    Yes 15 0.174783 0.263480
Sat No 45 0.158048 0.291990
    Yes 42 0.147906 0.325733
Sun No 57 0.160113 0.252672
    Yes 19 0.187250 0.710345
Thur No 45 0.160298 0.266312
    Yes 17 0.163863 0.241255
```

Como antes, uma lista de tuplas com nomes personalizados pode ser passada:

```
In [69]: ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]
```

```
In [70]: grouped['tip_pct', 'total_bill'].agg(ftuples)
Out[70]:
                tip_pct total_bill
           Durchschnitt Abweichung Durchschnitt Abweichung
day smoker
Fri No 0.151650 0.000791 18.420000 25.596333
    Yes 0.174783 0.002631 16.813333 82.562438
Sat No 0.158048 0.001581 19.661778 79.908965
    Yes 0.147906 0.003767 21.276667 101.387535
Sun No 0.160113 0.001793 20.506667 66.099980
    Yes 0.187250 0.023757 24.120000 109.046044
Thur No 0.160298 0.001503 17.113111 59.625081
    Yes 0.163863 0.001551 19.190588 69.808518
```

Suponha agora que você quisesse aplicar funções possivelmente distintas em uma ou mais colunas. Para isso, passe um dicionário para agg, contendo um mapeamento de nomes de colunas para qualquer uma das especificações de função listadas até agora:

```
In [71]: grouped.agg({'tip' : np.max, 'size' : 'sum'})
Out[71]:
              tip size
day smoker
Fri No 3.50 9
    Yes 4.73 31
Sat No 9.00 115
    Yes 10.00 104
Sun No 6.00 167
    Yes 6.50 49
Thur No 6.70 112
    Yes 5.00 40

In [72]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
   ....: 'size' : 'sum'})
Out[72]:
             tip_pct size
                min max mean std sum
day smoker
Fri No 0.120385 0.187735 0.151650 0.028123 9
    Yes 0.103555 0.263480 0.174783 0.051293 31
```

```
Sat No 0.056797 0.291990 0.158048 0.039767 115
    Yes 0.035638 0.325733 0.147906 0.061375 104
Sun No 0.059447 0.252672 0.160113 0.042347 167
    Yes 0.065660 0.710345 0.187250 0.154134 49
Thur No 0.072961 0.266312 0.160298 0.038774 112
    Yes 0.090014 0.241255 0.163863 0.039389 40
```

Um DataFrame terá colunas hierárquicas somente se várias funções forem aplicadas em no mínimo uma coluna.

Devolvendo dados agregados sem índices de linha

Em todos os exemplos até agora, os dados agregados retornavam com um índice, possivelmente hierárquico, composto de combinações únicas de chaves de grupo. Como isso nem sempre é desejável, podemos desativar esse comportamento na maioria dos casos, passando `as_index=False` para `groupby`:

```
In [73]: tips.groupby(['day', 'smoker'], as_index=False).mean()
Out[73]:
   day smoker total_bill tip size tip_pct
0 Fri No 18.420000 2.812500 2.250000 0.151650
1 Fri Yes 16.813333 2.714000 2.066667 0.174783
2 Sat No 19.661778 3.102889 2.555556 0.158048
3 Sat Yes 21.276667 2.875476 2.476190 0.147906
4 Sun No 20.506667 3.167895 2.929825 0.160113
5 Sun Yes 24.120000 3.516842 2.578947 0.187250
6 Thur No 17.113111 2.673778 2.488889 0.160298
7 Thur Yes 19.190588 3.030000 2.352941 0.163863
```

Claro que sempre é possível obter o resultado nesse formato chamando `reset_index` nele. Usar o método `as_index=False` evita alguns processamentos desnecessários.

10.3 Método apply: separar-aplicar-combinar genérico

O método de propósito mais geral de GroupBy é `apply`, que será o assunto do restante desta seção. Conforme mostrado na Figura

10.2, apply separa o objeto sendo manipulado em partes, chama a função recebida em cada parte e, em seguida, tenta concatená-las.

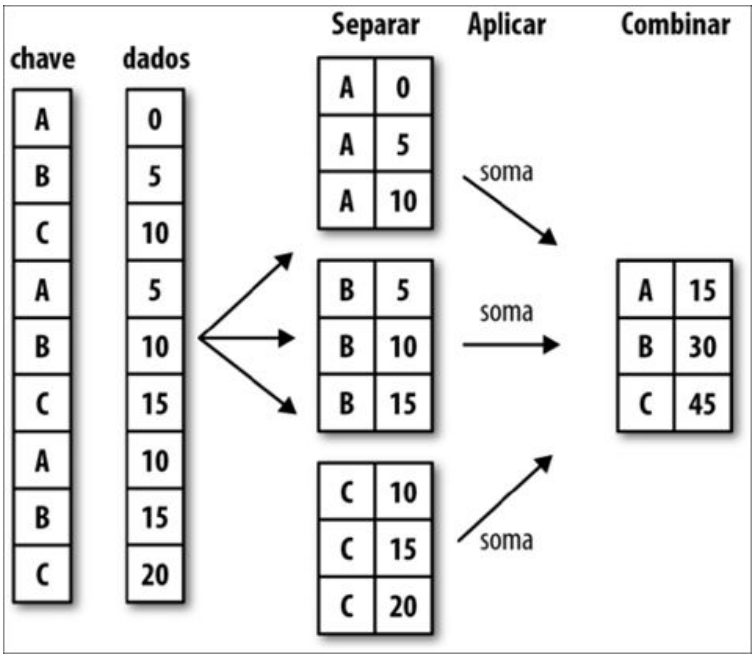


Figura 10.2 – Demonstração de uma agregação em grupos.

Retornando ao conjunto de dados anterior de gorjetas, suponha que quiséssemos selecionar os cinco primeiros valores de tip_pct por grupo. Inicialmente, escreva uma função que selecione as linhas com os maiores valores em uma coluna em particular:

```
In [74]: def top(df, n=5, column='tip_pct'):
   ....: return df.sort_values(by=column)[-n:]

In [75]: top(tips, n=6)
Out[75]:
    total_bill tip smoker day time size tip_pct
109 14.31 4.00 Yes Sat Dinner 2 0.279525
183 23.17 6.50 Yes Sun Dinner 4 0.280535
232 11.61 3.39 No Sat Dinner 2 0.291990
67 3.07 1.00 Yes Sat Dinner 1 0.325733
178 9.60 4.00 Yes Sun Dinner 2 0.416667
172 7.25 5.15 Yes Sun Dinner 2 0.710345
```

Se agruparmos de acordo com smoker, por exemplo, e chamarmos apply com essa função, teremos o seguinte:

```
In [76]: tips.groupby('smoker').apply(top)
Out[76]:
          total_bill tip smoker day time size tip_pct
smoker
No 88 24.71 5.85 No Thur Lunch 2 0.236746
      185 20.69 5.00 No Sun Dinner 5 0.241663
      51 10.29 2.60 No Sun Dinner 2 0.252672
      149 7.51 2.00 No Thur Lunch 2 0.266312
      232 11.61 3.39 No Sat Dinner 2 0.291990
Yes 109 14.31 4.00 Yes Sat Dinner 2 0.279525
      183 23.17 6.50 Yes Sun Dinner 4 0.280535
      67 3.07 1.00 Yes Sat Dinner 1 0.325733
      178 9.60 4.00 Yes Sun Dinner 2 0.416667
      172 7.25 5.15 Yes Sun Dinner 2 0.710345
```

O que aconteceu nesse caso? A função top é chamada em cada grupo de linhas do DataFrame; então os resultados são unidos com pandas.concat, atribuindo os nomes dos grupos como rótulos para cada parte. Assim, o resultado tem um índice hierárquico cujo nível mais interno contém valores de índice do DataFrame original.

Se você passar uma função para apply que aceite outros argumentos ou argumentos nomeados, esses poderão ser passados depois da função:

```
In [77]: tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
Out[77]:
               total_bill tip smoker day time size tip_pct
smoker day
No Fri 94 22.75 3.25 No Fri Dinner 2 0.142857
      Sat 212 48.33 9.00 No Sat Dinner 4 0.186220
      Sun 156 48.17 5.00 No Sun Dinner 6 0.103799
      Thur 142 41.19 5.00 No Thur Lunch 5 0.121389
Yes Fri 95 40.17 4.73 Yes Fri Dinner 4 0.117750
      Sat 170 50.81 10.00 Yes Sat Dinner 3 0.196812
      Sun 182 45.35 3.50 Yes Sun Dinner 3 0.077178
      Thur 197 43.11 5.00 Yes Thur Lunch 4 0.115982
```

 Para além desse uso básico, tirar o máximo de proveito de apply pode exigir um pouco de criatividade. O que ocorre dentro da função passada

para esse método é de sua responsabilidade; ela só precisa devolver um objeto do pandas ou um valor escalar. O restante deste capítulo consistirá principalmente de exemplos que mostram como resolver diversos problemas usando groupby.

Talvez você se recorde de que chamei describe antes em um objeto GroupBy:

```
In [78]: result = tips.groupby('smoker')['tip_pct'].describe()

In [79]: result
Out[79]:
        count mean std min 25% 50% 75% \
smoker
No 151.0 0.159328 0.039910 0.056797 0.136906 0.155625 0.185014
Yes 93.0 0.163196 0.085119 0.035638 0.106771 0.153846 0.195059
             max
smoker
No 0.291990
Yes 0.710345

In [80]: result.unstack('smoker')
Out[80]:
       smoker
count No 151.000000
       Yes 93.000000
mean No 0.159328
       Yes 0.163196
std No 0.039910
       Yes 0.085119
min No 0.056797
       Yes 0.035638
25% No 0.136906
       Yes 0.106771
50% No 0.155625
       Yes 0.153846
75% No 0.185014
       Yes 0.195059
max No 0.291990
       Yes 0.710345
dtype: float64
```

Em GroupBy, quando chamamos um método como describe, esse

será, na verdade, apenas um atalho para:

```
f = lambda x: x.describe()
grouped.apply(f)
```

Suprimindo as chaves de grupo

Nos exemplos anteriores, vemos que o objeto resultante tem um índice hierárquico composto das chaves de grupo, junto com os índices de cada parte do objeto original. Podemos desativar isso passando group_keys=False para groupby:

```
In [81]: tips.groupby('smoker', group_keys=False).apply(top)
Out[81]:
    total_bill tip smoker day time size tip_pct
88 24.71 5.85 No Thur Lunch 2 0.236746
185 20.69 5.00 No Sun Dinner 5 0.241663
51 10.29 2.60 No Sun Dinner 2 0.252672
149 7.51 2.00 No Thur Lunch 2 0.266312
232 11.61 3.39 No Sat Dinner 2 0.291990
109 14.31 4.00 Yes Sat Dinner 2 0.279525
183 23.17 6.50 Yes Sun Dinner 4 0.280535
67 3.07 1.00 Yes Sat Dinner 1 0.325733
178 9.60 4.00 Yes Sun Dinner 2 0.416667
172 7.25 5.15 Yes Sun Dinner 2 0.710345
```

Análise de quantis e de buckets

Como você deve se recordar do Capítulo 8, o pandas tem algumas ferramentas, em particular cut e qcut, para fatiar dados em buckets, com compartimentos (bins) de sua preferência, ou por quantis da amostra. Combinar essas funções com groupby faz com que seja conveniente realizar análises de buckets ou de quantis em um conjunto de dados. Considere um conjunto de dados aleatório simples e uma classificação em buckets de mesmo tamanho usando cut:

```
In [82]: frame = pd.DataFrame({'data1': np.random.randn(1000),
   ....: 'data2': np.random.randn(1000)})

In [83]: quartiles = pd.cut(frame.data1, 4)
```

```
In [84]: quartiles[:10]
Out[84]:
0 (-1.23, 0.489]
1 (-2.956, -1.23]
2 (-1.23, 0.489]
3 (0.489, 2.208]
4 (-1.23, 0.489]
5 (0.489, 2.208]
6 (-1.23, 0.489]
7 (-1.23, 0.489]
8 (0.489, 2.208]
9 (0.489, 2.208]
Name: data1, dtype: category
Categories (4, interval[float64]): [(-2.956, -1.23] < (-1.23, 0.489] < (0.489, 2.
208] < (2.208, 3.928]]
```

O objeto Categorical devolvido por cut pode ser passado diretamente para groupby. Assim, podemos calcular um conjunto de estatísticas para a coluna data2 da seguinte maneira:

```
In [85]: def get_stats(group):
   ....: return {'min': group.min(), 'max': group.max(),
   ....: 'count': group.count(), 'mean': group.mean()}

In [86]: grouped = frame.data2.groupby(quartiles)

In [87]: grouped.apply(get_stats).unstack()
Out[87]:
                count max mean min
data1
(-2.956, -1.23] 95.0 1.670835 -0.039521 -3.399312
(-1.23, 0.489] 598.0 3.260383 -0.002051 -2.989741
(0.489, 2.208] 297.0 2.954439 0.081822 -3.745356
(2.208, 3.928] 10.0 1.765640 0.024750 -1.929776
```

Esses buckets eram de mesmo tamanho; para calcular buckets de mesmo tamanho em quantis da amostra, utilize qcut. Usarei labels=False para obter somente os números dos quantis:

```
# Devolve os números dos quantis
In [88]: grouping = pd.qcut(frame.data1, 10, labels=False)
```

```
In [89]: grouped = frame.data2.groupby(grouping)

In [90]: grouped.apply(get_stats).unstack()
Out[90]:
      count max mean min
data1
0 100.0 1.670835 -0.049902 -3.399312
1 100.0 2.628441 0.030989 -1.950098
2 100.0 2.527939 -0.067179 -2.925113
3 100.0 3.260383 0.065713 -2.315555
4 100.0 2.074345 -0.111653 -2.047939
5 100.0 2.184810 0.052130 -2.989741
6 100.0 2.458842 -0.021489 -2.223506
7 100.0 2.954439 -0.026459 -3.056990
8 100.0 2.735527 0.103406 -3.745356
9 100.0 2.377020 0.220122 -2.064111
```

Veremos o tipo Categorical do pandas com mais detalhes no Capítulo 12.

Exemplo: preenchendo valores ausentes com valores específicos de grupo

Ao limpar dados ausentes, em alguns casos, você substituirá os dados observados usando dropna, porém, em outros, talvez você queira representar valores nulos (NA), isto é, preenchê-los, com um valor fixo ou outro valor derivado dos dados. fillna é a ferramenta correta a ser usada; por exemplo, preencherei a seguir os valores NA com a média:

```
In [91]: s = pd.Series(np.random.randn(6))

In [92]: s[::2] = np.nan

In [93]: s
Out[93]:
0 NaN
1 -0.125921
2 NaN
```

```
3 -0.884475
4 NaN
5 0.227290
dtype: float64

In [94]: s.fillna(s.mean())
Out[94]:
0 -0.261035
1 -0.125921
2 -0.261035
3 -0.884475
4 -0.261035
5 0.227290
dtype: float64
```

Suponha que seja necessário que o valor de preenchimento varie conforme o grupo. Um modo de fazer isso é agrupar os dados e usar `apply` com uma função que chame `fillna` em cada porção de dados. Eis alguns dados de exemplo dos estados norte-americanos, divididos em regiões leste e oeste:

```
In [95]: states = ['Ohio', 'New York', 'Vermont', 'Florida',
   ....: 'Oregon', 'Nevada', 'California', 'Idaho']

In [96]: group_key = ['East'] * 4 + ['West'] * 4

In [97]: data = pd.Series(np.random.randn(8), index=states)

In [98]: data
Out[98]:
Ohio 0.922264
New York -2.153545
Vermont -0.365757
Florida -0.375842
Oregon 0.329939
Nevada 0.981994
California 1.105913
Idaho -1.613716
dtype: float64
```

Observe que a sintaxe `['East'] * 4` gera uma lista contendo quatro cópias dos elementos em `['East']`. Somar listas faz com que elas

sejam concatenadas.

Vamos definir alguns valores como ausentes nos dados:

```
In [99]: data[['Vermont', 'Nevada', 'Idaho']] = np.nan

In [100]: data
Out[100]:
Ohio 0.922264
New York -2.153545
Vermont NaN
Florida -0.375842
Oregon 0.329939
Nevada NaN
California 1.105913
Idaho NaN
dtype: float64

In [101]: data.groupby(group_key).mean()
Out[101]:
East -0.535707
West 0.717926
dtype: float64
```

Podemos preencher os valores NA usando as médias dos grupos, assim:

```
In [102]: fill_mean = lambda g: g.fillna(g.mean())

In [103]: data.groupby(group_key).apply(fill_mean)
Out[103]:
Ohio 0.922264
New York -2.153545
Vermont -0.535707
Florida -0.375842
Oregon 0.329939
Nevada 0.717926
California 1.105913
Idaho 0.717926
dtype: float64
```

Em outro caso, você pode ter valores de preenchimento predefinidos em seu código, que variem conforme o grupo. Como os

grupos têm um atributo name definido internamente, podemos usá-lo:

```
In [104]: fill_values = {'East': 0.5, 'West': -1}

In [105]: fill_func = lambda g: g.fillna(fill_values[g.name])

In [106]: data.groupby(group_key).apply(fill_func)
Out[106]:
Ohio 0.922264
New York -2.153545
Vermont 0.500000
Florida -0.375842
Oregon 0.329939
Nevada -1.000000
California 1.105913
Idaho -1.000000
dtype: float64
```

Exemplo: amostragem aleatória e permutação

Suponha que quiséssemos sortear uma amostra aleatória (com ou sem substituição) a partir de um conjunto de dados grande, visando a uma simulação de Monte Carlo ou outra aplicação. Há algumas maneiras de fazer os “sorteios”; usaremos a seguir o método sample de Series.

Para uma demonstração, eis uma forma de construir um baralho em inglês:

```
# Copas (Hearts), Espada (Spades), Paus (Clubs), Ouro (Diamonds)
suits = ['H', 'S', 'C', 'D']
card_val = (list(range(1, 11)) + [10] * 3) * 4
base_names = ['A'] + list(range(2, 11)) + ['J', 'K', 'Q']
cards = []
for suit in ['H', 'S', 'C', 'D']:
    cards.extend(str(num) + suit for num in base_names)

deck = pd.Series(card_val, index=cards)
```

Agora temos uma Series de tamanho 52 cujo índice contém os nomes das cartas e os valores são aqueles usados no Blackjack (Vinte e Um) e em outros jogos (para manter a simplicidade, deixarei

que o ás 'A' seja somente 1):

```
In [108]: deck[:13]
Out[108]:
AH 1
2H 2
3H 3
4H 4
5H 5
6H 6
7H 7
8H 8
9H 9
10H 10
JH 10
KH 10
QH 10
dtype: int64
```

Com base no que dissemos antes, sortear uma mão de cinco cartas do baralho poderia ser escrito da seguinte maneira:

```
In [109]: def draw(deck, n=5):
   .....: return deck.sample(n)

In [110]: draw(deck)
Out[110]:
AD 1
8C 8
5H 5
KC 10
2C 2
dtype: int64
```

Suponha que quiséssemos duas cartas aleatórias de cada naipe. Como o naipe é o último caractere do nome de cada carta, podemos fazer agrupamentos com base nisso e usar apply:

```
In [111]: get_suit = lambda card: card[-1] # a última letra é o naipe

In [112]: deck.groupby(get_suit).apply(draw, n=2)
Out[112]:
C 2C 2
```

```
   3C 3
D KD 10
   8D 8
H KH 10
   3H 3
S 2S 2
   4S 4
dtype: int64
```

De modo alternativo, poderíamos escrever:

```
In [113]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
Out[113]:
KC 10
JC 10
AD 1
5D 5
5H 5
6H 6
7S 7
KS 10
dtype: int64
```

Exemplo: média ponderada de grupos e correlação

No paradigma separar-aplicar-combinar (split-apply-combine) de groupby, operações entre colunas em um DataFrame ou em duas Series, como uma média ponderada de grupos, são possíveis. Como exemplo, considere o conjunto de dados a seguir contendo chaves de grupo, valores e alguns pesos:

```
In [114]: df = pd.DataFrame({'category': ['a', 'a', 'a', 'a',
   .....: 'b', 'b', 'b', 'b'],
   .....: 'data': np.random.randn(8),
   .....: 'weights': np.random.rand(8)})

In [115]: df
Out[115]:
  category data weights
0 a 1.561587 0.957515
```

```
1 a 1.219984 0.347267
2 a -0.482239 0.581362
3 a 0.315667 0.217091
4 b -0.047852 0.894406
5 b -0.454145 0.918564
6 b -0.556774 0.277825
7 b 0.253321 0.955905
```

A média ponderada dos grupos por category seria então:

```
In [116]: grouped = df.groupby('category')

In [117]: get_wavg = lambda g: np.average(g['data'], weights=g['weights'])

In [118]: grouped.apply(get_wavg)
Out[118]:
category
a 0.811643
b -0.122262
dtype: float64
```

Como outro exemplo, considere um conjunto de dados financeiros, originalmente obtido do Yahoo! Finance, contendo os preços de algumas ações no final do dia e o índice S&P 500 (o símbolo SPX):

```
In [119]: close_px = pd.read_csv('examples/stock_px_2.csv', parse_dates=True,
   .....: index_col=0)

In [120]: close_px.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
AAPL 2214 non-null float64
MSFT 2214 non-null float64
XOM 2214 non-null float64
SPX 2214 non-null float64
dtypes: float64(4)
memory usage: 86.5 KB

In [121]: close_px[-4:]
Out[121]:
            AAPL MSFT XOM SPX
2011-10-11 400.29 27.00 76.27 1195.54
```

```
2011-10-12 402.19 26.96 77.16 1207.25
2011-10-13 408.43 27.18 76.37 1203.66
2011-10-14 422.00 27.27 78.11 1224.58
```

Uma tarefa interessante seria calcular um DataFrame constituído das correlações anuais entre retornos diários (calculados a partir de mudanças percentuais) e SPX. Como forma de fazer isso, inicialmente criaremos uma função que calcule a correlação aos pares entre cada coluna e a coluna 'SPX':

```
In [122]: spx_corr = lambda x: x.corrwith(x['SPX'])
```

Em seguida, calculamos as mudanças percentuais em close_px usando pct_change:

```
In [123]: rets = close_px.pct_change().dropna()
```

Por fim, agrupamos essas mudanças percentuais por ano; esse valor pode ser extraído de cada rótulo de linha com uma função de uma linha que devolve o atributo year de cada rótulo datetime:

```
In [124]: get_year = lambda x: x.year

In [125]: by_year = rets.groupby(get_year)

In [126]: by_year.apply(spx_corr)
Out[126]:
          AAPL MSFT XOM SPX
2003 0.541124 0.745174 0.661265 1.0
2004 0.374283 0.588531 0.557742 1.0
2005 0.467540 0.562374 0.631010 1.0
2006 0.428267 0.406126 0.518514 1.0
2007 0.508118 0.658770 0.786264 1.0
2008 0.681434 0.804626 0.828303 1.0
2009 0.707103 0.654902 0.797921 1.0
2010 0.710105 0.730118 0.839057 1.0
2011 0.691931 0.800996 0.859975 1.0
```

Também poderíamos calcular as correlações entre as colunas. A seguir, calcularemos a correlação anual entre a Apple e a Microsoft:

```
In [127]: by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))
Out[127]:
2003 0.480868
```

```
2004 0.259024
2005 0.300093
2006 0.161735
2007 0.417738
2008 0.611901
2009 0.432738
2010 0.571946
2011 0.581987
dtype: float64
```

Exemplo: regressão linear nos grupos

Seguindo a mesma temática do exemplo anterior, podemos utilizar groupby para realizar análises estatísticas mais complexas nos grupos, desde que a função devolva um objeto do pandas ou um valor escalar. Por exemplo, posso definir a função regress a seguir (usando a biblioteca de econometria statsmodels), que executa uma regressão OLS (Ordinary Least Squares, ou Mínimos Quadrados Ordinários) em cada porção de dados:

```
import statsmodels.api as sm
def regress(data, yvar, xvars):
    Y = data[yvar]
    X = data[xvars]
    X['intercept'] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params
```

Para executar uma regressão linear anual de AAPL nos retornos de SPX, execute:

```
In [129]: by_year.apply(regress, 'AAPL', ['SPX'])
Out[129]:
          SPX intercept
2003 1.195406 0.000710
2004 1.363463 0.004201
2005 1.766415 0.003246
2006 1.645496 0.000080
2007 1.198761 0.003438
2008 0.968016 -0.001110
2009 0.879103 0.002954
```

```
2010 1.052608 0.001261
2011 0.806605 0.001514
```

10.4 Tabelas pivôs e tabulação cruzada

Uma *tabela pivô* é uma ferramenta de sintetização de dados frequentemente encontrada em programas de planilhas e em outros softwares de análise de dados. Ela agrega uma tabela de dados de acordo com uma ou mais chaves, organizando os dados em um retângulo com algumas das chaves de grupo nas linhas e outras nas colunas. As tabelas pivôs em Python com pandas são possíveis por meio do recurso groupby descrito neste capítulo, em conjunto com operações de reformatação que utilizam indexação hierárquica. O DataFrame tem um método pivot_table, e há também uma função pandas.pivot_table de nível superior. Além de oferecer uma interface conveniente para groupby, pivot_table pode somar totais parciais, também conhecidos como *margens* (margins).

Voltando ao conjunto de dados de gorjetas, suponha que quiséssemos calcular uma tabela de médias de grupos (o tipo de agregação default de pivot_table), organizada por day e smoker nas linhas:

```
In [130]: tips.pivot_table(index=['day', 'smoker'])
Out[130]:
                size tip tip_pct total_bill
day smoker
Fri No 2.250000 2.812500 0.151650 18.420000
    Yes 2.066667 2.714000 0.174783 16.813333
Sat No 2.555556 3.102889 0.158048 19.661778
    Yes 2.476190 2.875476 0.147906 21.276667
Sun No 2.929825 3.167895 0.160113 20.506667
    Yes 2.578947 3.516842 0.187250 24.120000
Thur No 2.488889 2.673778 0.160298 17.113111
    Yes 2.352941 3.030000 0.163863 19.190588
```

Esses dados poderiam ter sido gerados diretamente com groupby. Suponha agora que quiséssemos agregar apenas tip_pct e size, e, além disso, agrupar de acordo com time. Colocarei smoker nas

colunas da tabela e day nas linhas:

```
In [131]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],
   .....: columns='smoker')
Out[131]:
                size tip_pct
smoker No Yes No Yes
time day
Dinner Fri 2.000000 2.222222 0.139622 0.165347
       Sat 2.555556 2.476190 0.158048 0.147906
       Sun 2.929825 2.578947 0.160113 0.187250
       Thur 2.000000 NaN 0.159744 NaN
Lunch Fri 3.000000 1.833333 0.187735 0.188937
       Thur 2.500000 2.352941 0.160311 0.163863
```

Poderíamos expandir essa tabela de modo que inclua totais parciais, passando margins=True. Isso tem o efeito de adicionar rótulos All para linhas e colunas, com os valores correspondentes sendo as estatísticas de grupo para todos os dados em uma única camada:

```
In [132]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],
   .....: columns='smoker', margins=True)
Out[132]:
                size tip_pct
smoker No Yes All No Yes All
time day
Dinner Fri 2.000000 2.222222 2.166667 0.139622 0.165347 0.158916
       Sat 2.555556 2.476190 2.517241 0.158048 0.147906 0.153152
       Sun 2.929825 2.578947 2.842105 0.160113 0.187250 0.166897
       Thur 2.000000 NaN 2.000000 0.159744 NaN 0.159744
Lunch Fri 3.000000 1.833333 2.000000 0.187735 0.188937 0.188765
       Thur 2.500000 2.352941 2.459016 0.160311 0.163863 0.161301
All 2.668874 2.408602 2.569672 0.159328 0.163196 0.160803
```

Nesse exemplo, os valores de All são as médias, sem levar em consideração os fumantes *versus* os não fumantes (as colunas de All) nem qualquer um dos dois níveis de agrupamento nas linhas (a linha All).

Para utilizar uma função de agregação diferente, passe-a para aggfunc. Por exemplo, 'count' ou len oferecerão uma tabulação

cruzada (contador ou frequência) dos tamanhos dos grupos:

```
In [133]: tips.pivot_table('tip_pct', index=['time', 'smoker'], columns='day',
   .....: aggfunc=len, margins=True)
Out[133]:
day Fri Sat Sun Thur All
time smoker
Dinner No 3.0 45.0 57.0 1.0 106.0
       Yes 9.0 42.0 19.0 NaN 70.0
Lunch No 1.0 NaN NaN 44.0 45.0
       Yes 6.0 NaN NaN 17.0 23.0
All 19.0 87.0 76.0 62.0 244.0
```

Se algumas combinações forem vazias (ou se forem NA), você poderá passar um fill_value:

```
In [134]: tips.pivot_table('tip_pct', index=['time', 'size', 'smoker'],
   .....: columns='day', aggfunc='mean', fill_value=0)
Out[134]:
day Fri Sat Sun Thur
time size smoker
Dinner 1 No 0.000000 0.137931 0.000000 0.000000
           Yes 0.000000 0.325733 0.000000 0.000000
       2 No 0.139622 0.162705 0.168859 0.159744
           Yes 0.171297 0.148668 0.207893 0.000000
       3 No 0.000000 0.154661 0.152663 0.000000
           Yes 0.000000 0.144995 0.152660 0.000000
       4 No 0.000000 0.150096 0.148143 0.000000
           Yes 0.117750 0.124515 0.193370 0.000000
       5 No 0.000000 0.000000 0.206928 0.000000
           Yes 0.000000 0.106572 0.065660 0.000000
... ... ... ... ...
Lunch 1 No 0.000000 0.000000 0.000000 0.181728
           Yes 0.223776 0.000000 0.000000 0.000000
       2 No 0.000000 0.000000 0.000000 0.166005
           Yes 0.181969 0.000000 0.000000 0.158843
       3 No 0.187735 0.000000 0.000000 0.084246
           Yes 0.000000 0.000000 0.000000 0.204952
       4 No 0.000000 0.000000 0.000000 0.138919
           Yes 0.000000 0.000000 0.000000 0.155410
       5 No 0.000000 0.000000 0.000000 0.121389
       6 No 0.000000 0.000000 0.000000 0.173706
```

[21 rows x 4 columns]

Veja a Tabela 10.2 que contém um resumo dos métodos de pivot_table.

Tabela 10.2 – Opções de pivot_table

Nome da função	Descrição
values	Nome ou nomes das colunas a serem agregadas; por padrão, agrega todas as colunas numéricas
index	Nomes das colunas ou outras chaves de grupo para agrupar nas linhas da tabela pivô resultante
columns	Nomes das colunas ou outras chaves de grupo para agrupar nas colunas da tabela pivô resultante
aggfunc	Função de agregação ou lista de funções (o default é 'mean'); pode ser qualquer função válida no contexto de um groupby
fill_value	Substitui valores ausentes na tabela resultante
dropna	Se for True, não inclui as colunas cujas entradas sejam todas NA
margins	Adiciona subtotais para linhas/colunas e um total geral (o default é False)

Tabulações cruzadas: crosstab

Uma tabulação cruzada (ou *crosstab*, em sua forma abreviada) é um caso especial de uma tabela pivô que calcula frequências de grupos. Eis um exemplo:

```
In [138]: data
Out[138]:
   Sample Nationality Handedness
0 1 USA Right-handed
1 2 Japan Left-handed
2 3 USA Right-handed
3 4 Japan Right-handed
4 5 Japan Left-handed
5 6 Japan Right-handed
6 7 USA Right-handed
7 8 USA Left-handed
8 9 Japan Right-handed
9 10 USA Right-handed
```

Como parte da análise de uma pesquisa, talvez quiséssemos resumir esses dados conforme a nacionalidade e o fato de as pessoas serem destras ou canhotas. Poderíamos usar pivot_table para isso, mas a função pandas.crosstab pode ser mais conveniente:

```
In [139]: pd.crosstab(data.Nationality, data.Handedness, margins=True)
Out[139]:
Handedness Left-handed Right-handed All
Nationality
Japan 2 3 5
USA 1 4 5
All 3 7 10
```

Os dois primeiros argumentos de crosstab podem ser um array ou uma Series ou uma lista de arrays. No caso dos dados de gorjeta, temos:

```
In [140]: pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)
Out[140]:
smoker No Yes All
time day
Dinner Fri 3 9 12
       Sat 45 42 87
       Sun 57 19 76
       Thur 1 0 1
Lunch Fri 1 6 7
       Thur 44 17 61
All 151 93 244
```

10.5 Conclusão

Dominar as ferramentas de agrupamento de dados do pandas pode ajudar tanto no trabalho de limpeza de dados como também na modelagem ou na análise estatística. No Capítulo 14, veremos vários outros exemplos de casos de uso para groupby em dados do mundo real.

No próximo capítulo, voltaremos a nossa atenção para os dados de séries temporais.

CAPÍTULO 11

Séries temporais

Dados de séries temporais são uma forma importante de dados estruturados em muitas áreas diferentes como finanças, economia, ecologia, neurociência e física. Quaisquer dados observados ou medidos em vários pontos no tempo compõem uma série temporal. Muitas séries temporais têm *frequência fixa*, o que equivale a dizer que os pontos de dados ocorrem a intervalos regulares de acordo com alguma regra, por exemplo, a cada 15 segundos, a cada 5 minutos ou uma vez por mês. As séries temporais também podem ser *irregulares*, sem uma unidade de tempo fixa ou um offset (deslocamento) entre as unidades. O modo como marcamos e referenciamos os dados de séries temporais depende da aplicação, e podemos ter um dos seguintes casos:

- *Timestamps*, que são instantes específicos no tempo.
- *Períodos* fixos, por exemplo, o mês de janeiro de 2007 ou o ano todo de 2010.
- *Intervalos* de tempo, representados por um timestamp de início e de fim. Podemos pensar nos períodos como casos especiais de intervalos.
- Tempo do experimento ou tempo decorrido; cada timestamp é uma medida de tempo relativa a um determinado instante de início (por exemplo, o diâmetro de um biscoito que assa a cada segundo desde que foi colocado no forno).

Neste capítulo, estarei interessado principalmente nas séries temporais das três primeiras categorias, embora muitas das técnicas possam ser aplicadas em séries temporais experimentais, em que o

índice pode ser um inteiro ou um número de ponto flutuante indicando o tempo decorrido a partir do início do experimento. O tipo mais simples e mais amplamente utilizado de séries temporais são aquelas indexadas por timestamp.

O pandas oferece suporte também para índices baseados em timedeltas (diferenças de tempo), que podem ser uma forma conveniente de representar a duração do experimento ou o tempo decorrido. Não exploraremos índices timedelta neste livro, mas você pode saber mais sobre esse assunto consultando a documentação do pandas (*http://pandas.pydata.org/*).

O pandas oferece várias ferramentas embutidas e algoritmos de dados para séries temporais. É possível trabalhar com séries temporais bem grandes de modo eficaz e manipular, agregar e fazer uma reamostragem de séries temporais irregulares e de frequência fixa facilmente. Algumas dessas ferramentas são particularmente úteis em aplicações financeiras e econômicas, mas, certamente, você poderia usá-las para analisar dados de log de servidores também.

11.1 Tipos de dados e ferramentas para data e hora

A biblioteca-padrão de Python inclui tipos de dados para datas e horas, assim como funcionalidades relacionadas ao calendário. Os módulos datetime, time e calendar são os principais locais para começar. O tipo datetime.datetime, ou apenas datetime, é amplamente utilizado:

```
In [10]: from datetime import datetime

In [11]: now = datetime.now()

In [12]: now
Out[12]: datetime.datetime(2017, 10, 17, 13, 34, 33, 597499)
```

```
In [13]: now.year, now.month, now.day
Out[13]: (2017, 10, 17)
```

datetime armazena tanto a data quanto a hora, até o nível de microssegundos. timedelta representa a diferença de tempo entre dois objetos datetime:

```
In [14]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

In [15]: delta
Out[15]: datetime.timedelta(926, 56700)

In [16]: delta.days
Out[16]: 926

In [17]: delta.seconds
Out[17]: 56700
```

Podemos somar um timedelta ou um múltiplo dele a um objeto datetime (ou subtrair desse objeto), o que resultará em um novo objeto deslocado:

```
In [18]: from datetime import timedelta

In [19]: start = datetime(2011, 1, 7)

In [20]: start + timedelta(12)
Out[20]: datetime.datetime(2011, 1, 19, 0, 0)

In [21]: start - 2 * timedelta(12)
Out[21]: datetime.datetime(2010, 12, 14, 0, 0)
```

A Tabela 11.1 sintetiza os tipos de dados do módulo datetime. Embora este capítulo esteja interessado principalmente nos tipos de dados do pandas e na manipulação de séries temporais em um nível geral, você poderá deparar com tipos baseados em datetime em vários outros lugares em Python por aí.

Tabela 11.1 – Tipos no módulo datetime

Tipo	Descrição
date	Armazena uma data de calendário (ano, mês, dia) usando o calendário gregoriano

time	Armazena a hora do dia na forma de horas, minutos, segundos e microssegundos
datetime	Armazena tanto a data quanto a hora
timedelta	Representa a diferença entre dois valores de datetime (como dias, segundos e microssegundos)
tzinfo	Tipo-base para armazenar informações de fuso horário

Conversão entre string e datetime

Podemos formatar objetos datetime e objetos Timestamp do pandas, que apresentarei mais adiante, como strings, usando str ou o método strftime, passando uma especificação de formato:

```
In [22]: stamp = datetime(2011, 1, 3)

In [23]: str(stamp)
Out[23]: '2011-01-03 00:00:00'

In [24]: stamp.strftime('%Y-%m-%d')
Out[24]: '2011-01-03'
```

Veja a Tabela 11.2 que contém uma lista completa dos códigos de formatos (reproduzida do Capítulo 2).

Tabela 11-2 – Especificação de formatos de datetime (compatível com ISO C89)

Tipo	Descrição
%Y	Ano com quatro dígitos
%y	Ano com dois dígitos
%m	Mês com dois dígitos [01, 12]
%d	Dia com dois dígitos [01, 31]
%H	Hora (relógio com 24 horas) [00, 23]
%I	Hora (relógio com 12 horas) [01, 12]
%M	Minuto com dois dígitos [00, 59]
%S	Segundos [00, 61] (segundos 60, 61 são usados para segundos intercalares (leap seconds))
%w	Dia da semana como inteiro [0 (Domingo), 6]
%U	Número da semana no ano [00, 53]; domingo é considerado o primeiro dia

	da semana, e os dias antes do primeiro domingo do ano são a “semana 0”
%W	Número da semana no ano [00, 53]; segunda-feira é considerada o primeiro dia da semana, e os dias antes da primeira segunda-feira do ano são a “semana 0”
%z	Offset do fuso horário UTC como +HHMM ou -HHMM; vazio se o fuso horário não for considerado
%F	Atalho para %Y-%m-%d (por exemplo, 2012-4-18)
%D	Atalho para %m/%d/%y (por exemplo, 04/18/12)

Podemos usar esses mesmos códigos de formato para converter strings em datas usando datetime.strptime:

```
In [25]: value = '2011-01-03'

In [26]: datetime.strptime(value, '%Y-%m-%d')
Out[26]: datetime.datetime(2011, 1, 3, 0, 0)

In [27]: datestrs = ['7/6/2011', '8/6/2011']

In [28]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]
Out[28]:
[datetime.datetime(2011, 7, 6, 0, 0),
 datetime.datetime(2011, 8, 6, 0, 0)]
```

datetime.strptime é uma boa maneira de fazer parse de uma data com um formato conhecido. Entretanto pode ser um pouco irritante ter que escrever sempre uma especificação de formato, particularmente para formatos de data comuns. Nesse caso, podemos usar o método parser.parse do pacote de terceiros dateutil (é instalado automaticamente quando você instala o pandas):

```
In [29]: from dateutil.parser import parse

In [30]: parse('2011-01-03')
Out[30]: datetime.datetime(2011, 1, 3, 0, 0)
```

dateutil é capaz de fazer parse da maior parte das representações de data legíveis aos seres humanos:

```
In [31]: parse('Jan 31, 1997 10:45 PM')
Out[31]: datetime.datetime(1997, 1, 31, 22, 45)
```

Fora dos Estados Unidos, o dia estar antes do mês é bem comum,

portanto podemos passar dayfirst=True para sinalizar esse fato:

```
In [32]: parse('6/12/2011', dayfirst=True)
Out[32]: datetime.datetime(2011, 12, 6, 0, 0)
```

O pandas em geral é orientado a trabalhar com arrays de datas, sejam elas usadas como índice de um eixo ou uma coluna em um DataFrame. O método to_datetime faz parse de vários tipos diferentes de representações de data. O parse de formatos-padrões de data, como ISO 8601, pode ser feito rapidamente:

```
In [33]: datestrs = ['2011-07-06 12:00:00', '2011-08-06 00:00:00']

In [34]: pd.to_datetime(datestrs)
Out[34]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'],
dtype='dat
etime64[ns]', freq=None)
```

O parse também lida com valores que devam ser indicativos de dados ausentes (None, string vazia etc.):

```
In [35]: idx = pd.to_datetime(datestrs + [None])

In [36]: idx
Out[36]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00', 'NaT'], dty
pe='datetime64[ns]', freq=None)

In [37]: idx[2]
Out[37]: NaT

In [38]: pd.isnull(idx)
Out[38]: array([False, False, True], dtype=bool)
```

NaT (Not a Time) é o valor nulo do pandas para dados de timestamp.

dateutil.parser é uma ferramenta útil, porém imperfeita. Em especial, ela reconhece algumas strings como datas, as quais talvez você preferisse que não fossem reconhecidas – por exemplo, o parse de '42' será feito como o ano 2042, com a data de calendário de hoje.

Objetos datetime também têm uma série de opções de formatação específicas de localidade para sistemas em outros países ou

idiomas. Por exemplo, os nomes de mês abreviados serão diferentes em sistemas alemães ou franceses se comparados com os sistemas ingleses. Veja a Tabela 11.3 que apresenta uma lista delas.

Tabela 11.3 – Formatação de data específica de localidades

Tipo	Descrição
%a	Nome abreviado para o dia da semana
%A	Nome completo do dia da semana
%b	Nome abreviado para o mês
%B	Nome completo do mês
%c	Data e hora completas (por exemplo, 'Tue 01 May 2012 04:20:57 PM')
%p	Equivalente a AM ou a PM conforme a localidade
%x	Data formatada de modo apropriado à localidade (por exemplo, nos Estados Unidos, May 1, 2012 resultará em '05/01/2012')
%X	Horário apropriado à localidade (por exemplo, '04:24:12 PM')

11.2 Básico sobre séries temporais

Um tipo básico de objeto de série temporal no pandas é uma Series indexada por timestamps que, com frequência, é representada externamente ao pandas na forma de strings Python ou objetos datetime:

```
In [39]: from datetime import datetime

In [40]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),
   ....: datetime(2011, 1, 7), datetime(2011, 1, 8),
   ....: datetime(2011, 1, 10), datetime(2011, 1, 12)]

In [41]: ts = pd.Series(np.random.randn(6), index=dates)

In [42]: ts
Out[42]:
2011-01-02 -0.204708
2011-01-05 0.478943
2011-01-07 -0.519439
2011-01-08 -0.555730
```

```
2011-01-10 1.965781
2011-01-12 1.393406
dtype: float64
```

Internamente, esses objetos datetime foram colocados em um DatetimeIndex:

```
In [43]: ts.index
Out[43]:
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

Como ocorre com outras Series, as operações aritméticas entre séries temporais indexadas de modo diferente fazem automaticamente um alinhamento com base nas datas:

```
In [44]: ts + ts[::2]
Out[44]:
2011-01-02 -0.409415
2011-01-05 NaN
2011-01-07 -1.038877
2011-01-08 NaN
2011-01-10 3.931561
2011-01-12 NaN
dtype: float64
```

Lembre-se de que ts[::2] seleciona um elemento a cada dois em ts.

O pandas armazena timestamps usando o tipo de dado datetime64 do NumPy com resolução de nanossegundos:

```
In [45]: ts.index.dtype
Out[45]: dtype('<M8[ns]')
```

Valores escalares de um DatetimeIndex são objetos Timestamp do pandas:

```
In [46]: stamp = ts.index[0]

In [47]: stamp
Out[47]: Timestamp('2011-01-02 00:00:00')
```

Um Timestamp pode ser substituído em qualquer lugar em que usaríamos um objeto datetime. Além do mais, ele pode armazenar

informações de frequência (se houver) e compreender como fazer conversões de fuso horário e outros tipos de manipulações. Veremos outras informações sobre esse assunto mais adiante.

Indexação, seleção e geração de subconjuntos

As séries temporais se comportam como qualquer outra pandas.Series quando indexamos e selecionamos dados com base no rótulo:

```
In [48]: stamp = ts.index[2]

In [49]: ts[stamp]
Out[49]: -0.51943871505673811
```

Como conveniência, podemos também passar uma string que seja interpretável como uma data:

```
In [50]: ts['1/10/2011']
Out[50]: 1.9657805725027142

In [51]: ts['20110110']
Out[51]: 1.9657805725027142
```

Para séries temporais mais longas, um ano ou somente um ano e um mês podem ser passados para selecionar fatias de dados facilmente:

```
In [52]: longer_ts = pd.Series(np.random.randn(1000),
   ....: index=pd.date_range('1/1/2000', periods=1000))

In [53]: longer_ts
Out[53]:
2000-01-01 0.092908
2000-01-02 0.281746
2000-01-03 0.769023
2000-01-04 1.246435
2000-01-05 1.007189
2000-01-06 -1.296221
2000-01-07 0.274992
2000-01-08 0.228913
2000-01-09 1.352917
2000-01-10 0.886429
```

```
                ...
2002-09-17 -0.139298
2002-09-18 -1.159926
2002-09-19 0.618965
2002-09-20 1.373890
2002-09-21 -0.983505
2002-09-22 0.930944
2002-09-23 -0.811676
2002-09-24 -1.830156
2002-09-25 -0.138730
2002-09-26 0.334088
Freq: D, Length: 1000, dtype: float64

In [54]: longer_ts['2001']
Out[54]:
2001-01-01 1.599534
2001-01-02 0.474071
2001-01-03 0.151326
2001-01-04 -0.542173
2001-01-05 -0.475496
2001-01-06 0.106403
2001-01-07 -1.308228
2001-01-08 2.173185
2001-01-09 0.564561
2001-01-10 -0.190481
                ...
2001-12-22 0.000369
2001-12-23 0.900885
2001-12-24 -0.454869
2001-12-25 -0.864547
2001-12-26 1.129120
2001-12-27 0.057874
2001-12-28 -0.433739
2001-12-29 0.092698
2001-12-30 -1.397820
2001-12-31 1.457823
Freq: D, Length: 365, dtype: float64
```

Nesse caso, a string `'2001'` é interpretada como um ano e seleciona esse período de tempo. Isso também funcionará se o mês for especificado:

```
In [55]: longer_ts['2001-05']
Out[55]:
2001-05-01 -0.622547
2001-05-02 0.936289
2001-05-03 0.750018
2001-05-04 -0.056715
2001-05-05 2.300675
2001-05-06 0.569497
2001-05-07 1.489410
2001-05-08 1.264250
2001-05-09 -0.761837
2001-05-10 -0.331617
                ...
2001-05-22 0.503699
2001-05-23 -1.387874
2001-05-24 0.204851
2001-05-25 0.603705
2001-05-26 0.545680
2001-05-27 0.235477
2001-05-28 0.111835
2001-05-29 -1.251504
2001-05-30 -2.949343
2001-05-31 0.634634
Freq: D, Length: 31, dtype: float64
```

Fatiar com objetos `datetime` também funciona:

```
In [56]: ts[datetime(2011, 1, 7):]
Out[56]:
2011-01-07 -0.519439
2011-01-08 -0.555730
2011-01-10 1.965781
2011-01-12 1.393406
dtype: float64
```

Como a maioria dos dados de séries temporais está ordenada cronologicamente, podemos fatiar com timestamps não contidos em uma série temporal para efetuar uma consulta de intervalo:

```
In [57]: ts
Out[57]:
2011-01-02 -0.204708
2011-01-05 0.478943
```

```
2011-01-07 -0.519439
2011-01-08 -0.555730
2011-01-10 1.965781
2011-01-12 1.393406
dtype: float64

In [58]: ts['1/6/2011':'1/11/2011']
Out[58]:
2011-01-07 -0.519439
2011-01-08 -0.555730
2011-01-10 1.965781
dtype: float64
```

Como vimos antes, podemos passar uma data na forma de string, um datetime ou um timestamp. Lembre-se de que fatiar dessa forma gera visualizações das séries temporais originais, como no caso de fatiamento de arrays NumPy. Isso significa que nenhum dado será copiado, e modificações na fatia se refletirão nos dados originais.

Há um método de instância equivalente, truncate, que fatia uma Series entre duas datas:

```
In [59]: ts.truncate(after='1/9/2011')
Out[59]:
2011-01-02 -0.204708
2011-01-05 0.478943
2011-01-07 -0.519439
2011-01-08 -0.555730
dtype: float64
```

Tudo isso é válido também para um DataFrame, com indexação em suas linhas:

```
In [60]: dates = pd.date_range('1/1/2000', periods=100, freq='W-WED')

In [61]: long_df = pd.DataFrame(np.random.randn(100, 4),
   ....: index=dates,
   ....: columns=['Colorado', 'Texas',
   ....: 'New York', 'Ohio'])

In [62]: long_df.loc['5-2001']
Out[62]:
            Colorado Texas New York Ohio
```

```
2001-05-02 -0.006045 0.490094 -0.277186 -0.707213
2001-05-09 -0.560107 2.735527 0.927335 1.513906
2001-05-16 0.538600 1.273768 0.667876 -0.969206
2001-05-23 1.676091 -0.817649 0.050188 1.951312
2001-05-30 3.260383 0.963301 1.201206 -1.852001
```

Séries temporais com índices duplicados

Em algumas aplicações, pode haver vários dados observados que se enquadram em um timestamp em particular. Eis um exemplo:

```
In [63]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000',
   ....: '1/2/2000', '1/3/2000'])

In [64]: dup_ts = pd.Series(np.arange(5), index=dates)

In [65]: dup_ts
Out[65]:
2000-01-01 0
2000-01-02 1
2000-01-02 2
2000-01-02 3
2000-01-03 4
dtype: int64
```

Podemos dizer se o índice não é único verificando a sua propriedade is_unique:

```
In [66]: dup_ts.index.is_unique
Out[66]: False
```

A indexação nessa série temporal produzirá agora valores escalares ou fatias, conforme um timestamp esteja duplicado:

```
In [67]: dup_ts['1/3/2000'] # não duplicado
Out[67]: 4

In [68]: dup_ts['1/2/2000'] # duplicado
Out[68]:
2000-01-02 1
2000-01-02 2
2000-01-02 3
dtype: int64
```

Suponha que quiséssemos agregar os dados que tenham timestamps não únicos. Uma maneira de fazer isso é usar groupby e passar level=0:

```
In [69]: grouped = dup_ts.groupby(level=0)

In [70]: grouped.mean()
Out[70]:
2000-01-01 0
2000-01-02 2
2000-01-03 4
dtype: int64

In [71]: grouped.count()
Out[71]:
2000-01-01 1
2000-01-02 3
2000-01-03 1
dtype: int64
```

11.3 Intervalos de datas, frequências e deslocamentos

Supõe-se que séries temporais genéricas no pandas sejam irregulares; isso significa que elas não têm uma frequência fixa. Para muitas aplicações, isso é suficiente. No entanto, em geral é desejável trabalhar em relação a uma frequência fixa, como diariamente, mensalmente ou a cada 15 minutos, mesmo que isso signifique introduzir valores ausentes em uma série temporal. Felizmente o pandas tem um pacote completo de frequências padrões para séries temporais, além de ferramentas para fazer reamostragens (resampling), inferir frequências e gerar intervalos de datas com frequências fixas. Por exemplo, podemos converter a série temporal de exemplo para que tenha uma frequência fixa diária chamando resample:

```
In [72]: ts
Out[72]:
2011-01-02 -0.204708
```

```
2011-01-05 0.478943
2011-01-07 -0.519439
2011-01-08 -0.555730
2011-01-10 1.965781
2011-01-12 1.393406
dtype: float64

In [73]: resampler = ts.resample('D')
```

A string 'D' é interpretada como frequência diária.

A conversão entre frequências, ou *reamostragem* (resampling), é um assunto suficientemente amplo a ponto de merecer uma seção própria (Seção 11.6, “Reamostragem e conversão de frequências”). Nessa seção, mostrarei como usar as frequências básicas e seus múltiplos.

Gerando intervalos de datas

Embora eu o tenha usado anteriormente sem explicações, pandas.date_range é responsável por gerar um DatetimeIndex com um tamanho especificado, de acordo com uma frequência em particular:

```
In [74]: index = pd.date_range('2012-04-01', '2012-06-01')

In [75]: index
Out[75]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20',
               '2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24',
               '2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28',
               '2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02',
               '2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06',
               '2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10',
               '2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14',
               '2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18',
               '2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22',
               '2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26',
```

```
               '2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30',
               '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

Por padrão, date_range gera timestamps diariamente. Se você passar apenas uma data de início ou de fim, será necessário especificar o número de períodos a ser gerado:

```
In [76]: pd.date_range(start='2012-04-01', periods=20)
Out[76]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],
              dtype='datetime64[ns]', freq='D')

In [77]: pd.date_range(end='2012-06-01', periods=20)
Out[77]:
DatetimeIndex(['2012-05-13', '2012-05-14', '2012-05-15', '2012-05-16',
               '2012-05-17', '2012-05-18', '2012-05-19', '2012-05-20',
               '2012-05-21', '2012-05-22', '2012-05-23', '2012-05-24',
               '2012-05-25', '2012-05-26', '2012-05-27', '2012-05-28',
               '2012-05-29', '2012-05-30', '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

As datas de início e de fim definem fronteiras rigorosas para o índice de datas gerado. Por exemplo, se você quisesse um índice de datas contendo o último dia útil de cada mês, passaria a frequência 'BM'(final do mês útil; veja uma lista mais completa de frequências na Tabela 11.4), e somente as datas que se enquadrem no intervalo das datas serão incluídas:

```
In [78]: pd.date_range('2000-01-01', '2000-12-01', freq='BM')
Out[78]:
DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31', '2000-04-28',
               '2000-05-31', '2000-06-30', '2000-07-31', '2000-08-31',
               '2000-09-29', '2000-10-31', '2000-11-30'],
              dtype='datetime64[ns]', freq='BM')
```

Tabela 11.4 – Frequências básicas de séries temporais (não está completa)

Alias	Tipo de offset	Descrição
D	Day	Por dia do calendário
B	BusinessDay	Por dia útil
H	Hour	Por hora
T ou min	Minute	Por minuto
S	Second	Por segundo
L ou ms	Milli	Milissegundo (1/1.000 de 1 segundo)
U	Micro	Microssegundo (1/1.000.000 de 1 segundo)
M	MonthEnd	Último dia do mês do calendário
BM	BusinessMonthEnd	Último dia útil (dia da semana) do mês
MS	MonthBegin	Primeiro dia do mês do calendário
BMS	BusinessMonthBegin	Primeiro dia útil do mês
W-MON, W-TUE, ...	Week	Semanalmente em um dado dia da semana (MON, TUE, WED, THU, FRI, SAT ou SUN)
WOM-1MON, WOM-2MON, ...	WeekOfMonth	Gera datas semanalmente na primeira, segunda, terceira ou quarta semana do mês (por exemplo, WOM-3FRI para a terceira sexta-feira de cada mês)
Q-JAN, Q-FEB, ...	QuarterEnd	Datas trimestrais ancoradas no último dia do calendário de cada mês, para o período de 12 meses baseado no mês indicado (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV ou DEC)
BQ-JAN, BQ-FEB, ...	BusinessQuarterEnd	Datas trimestrais ancoradas no último dia útil de cada mês, para o período de 12 meses baseado no mês indicado
QS-JAN, QS-FEB, ...	QuarterBegin	Datas trimestrais ancoradas no primeiro dia do calendário de cada mês, para o período de 12 meses baseado no mês indicado
BQS-JAN, BQS-FEB, ...	BusinessQuarterBegin	Datas trimestrais ancoradas no primeiro dia útil de cada mês, para o período de 12 meses baseado no mês indicado

Alias	Tipo de offset	Descrição
A-JAN, A-FEB, ...	YearEnd	Datas anuais ancoradas no último dia do calendário de um dado mês (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV ou DEC)
BA-JAN, BA-FEB, ...	BusinessYearEnd	Datas anuais ancoradas no último dia útil de um dado mês
AS-JAN, AS-FEB, ...	YearBegin	Datas anuais ancoradas no primeiro dia de um dado mês
BAS-JAN, BAS-FEB, ...	BusinessYearBegin	Datas anuais ancoradas no primeiro dia útil de um dado mês

Por padrão, date_range preserva o horário (se houver) do timestamp de início ou de fim:

```
In [79]: pd.date_range('2012-05-02 12:56:31', periods=5)
Out[79]:
DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:56:31',
               '2012-05-04 12:56:31', '2012-05-05 12:56:31',
               '2012-05-06 12:56:31'],
              dtype='datetime64[ns]', freq='D')
```

Às vezes você terá datas de início ou de fim com informações de hora, mas vai querer gerar um conjunto de timestamps *normalizados* para a meia-noite como convenção. Para isso, temos a opção normalize:

```
In [80]: pd.date_range('2012-05-02 12:56:31', periods=5, normalize=True)
Out[80]:
DatetimeIndex(['2012-05-02', '2012-05-03', '2012-05-04', '2012-05-05',
               '2012-05-06'],
              dtype='datetime64[ns]', freq='D')
```

Frequências e offset de datas

As frequências no pandas são compostas de uma *frequência de base* e um multiplicador. As frequências de base em geral são referenciadas por um alias na forma de string, por exemplo, 'M' para

mensal ou 'H' para uma frequência de hora em hora. Para cada frequência de base, há um objeto definido, referenciado de modo geral como *offset de data.* Por exemplo, uma frequência de hora em hora pode ser representada pela classe Hour:

```
In [81]: from pandas.tseries.offsets import Hour, Minute

In [82]: hour = Hour()

In [83]: hour
Out[83]: <Hour>
```

Podemos definir um múltiplo de um offset especificando um inteiro:

```
In [84]: four_hours = Hour(4)

In [85]: four_hours
Out[85]: <4 * Hours>
```

Na maioria das aplicações, você jamais precisará criar explicitamente um desses objetos, mas usará um alias na forma de string, como 'H' ou '4H'. Inserir um inteiro antes da frequência de base criará um múltiplo dela:

```
In [86]: pd.date_range('2000-01-01', '2000-01-03 23:59', freq='4h')
Out[86]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',
               '2000-01-01 08:00:00', '2000-01-01 12:00:00',
               '2000-01-01 16:00:00', '2000-01-01 20:00:00',
               '2000-01-02 00:00:00', '2000-01-02 04:00:00',
               '2000-01-02 08:00:00', '2000-01-02 12:00:00',
               '2000-01-02 16:00:00', '2000-01-02 20:00:00',
               '2000-01-03 00:00:00', '2000-01-03 04:00:00',
               '2000-01-03 08:00:00', '2000-01-03 12:00:00',
               '2000-01-03 16:00:00', '2000-01-03 20:00:00'],
              dtype='datetime64[ns]', freq='4H')
```

Vários offsets podem ser combinados com uma soma:

```
In [87]: Hour(2) + Minute(30)
Out[87]: <150 * Minutes>
```

De modo semelhante, podemos passar strings de frequência, como '1h30min', cujo parse será feito de modo eficiente, gerando a mesma

expressão:

```
In [88]: pd.date_range('2000-01-01', periods=10, freq='1h30min')
Out[88]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:30:00',
               '2000-01-01 03:00:00', '2000-01-01 04:30:00',
               '2000-01-01 06:00:00', '2000-01-01 07:30:00',
               '2000-01-01 09:00:00', '2000-01-01 10:30:00',
               '2000-01-01 12:00:00', '2000-01-01 13:30:00'],
              dtype='datetime64[ns]', freq='90T')
```

Algumas frequências descrevem pontos no tempo que não são uniformemente espaçados. Por exemplo, 'M' (final do mês no calendário) e 'BM' (último dia útil/da semana do mês) dependem do número de dias em um mês e, no último caso, se o mês termina em um fim de semana ou não. Chamamos a esses de offsets *ancorados* (anchored offsets).

Consulte novamente a Tabela 11.4 que tem uma lista dos códigos de frequência e as classes de offset de datas disponíveis no pandas.

 Os usuários podem definir suas próprias classes de frequência personalizadas a fim de prover uma lógica de datas indisponível no pandas, embora os detalhes completos dessa tarefa estejam fora do escopo deste livro.

Datas com a semana do mês

Uma classe de frequência útil é a “semana do mês”, que começa com WOM. Ela permite obter datas como a terceira sexta-feira de cada mês:

```
In [89]: rng = pd.date_range('2012-01-01', '2012-09-01', freq='WOM-3FRI')

In [90]: list(rng)
Out[90]:
[Timestamp('2012-01-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-02-17 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-03-16 00:00:00', freq='WOM-3FRI'),
```

```
Timestamp('2012-04-20 00:00:00', freq='WOM-3FRI'),
Timestamp('2012-05-18 00:00:00', freq='WOM-3FRI'),
Timestamp('2012-06-15 00:00:00', freq='WOM-3FRI'),
Timestamp('2012-07-20 00:00:00', freq='WOM-3FRI'),
Timestamp('2012-08-17 00:00:00', freq='WOM-3FRI')]
```

Deslocamento de datas (adiantando e atrasando)

"Deslocamento" (shifting) refere-se à movimentação dos dados para trás e para a frente no tempo. Tanto Series quanto DataFrame têm um método shift para deslocamentos triviais para a frente e para trás, deixando o índice inalterado:

```
In [91]: ts = pd.Series(np.random.randn(4),
   ....: index=pd.date_range('1/1/2000', periods=4, freq='M'))

In [92]: ts
Out[92]:
2000-01-31 -0.066748
2000-02-29 0.838639
2000-03-31 -0.117388
2000-04-30 -0.517795
Freq: M, dtype: float64

In [93]: ts.shift(2)
Out[93]:
2000-01-31 NaN
2000-02-29 NaN
2000-03-31 -0.066748
2000-04-30 0.838639
Freq: M, dtype: float64

In [94]: ts.shift(-2)
Out[94]:
2000-01-31 -0.117388
2000-02-29 -0.517795
2000-03-31 NaN
2000-04-30 NaN
Freq: M, dtype: float64
```

Quando fazemos deslocamentos como esses, dados ausentes são

introduzidos no início ou no final da série temporal.

Um uso comum de shift ocorre no cálculo de mudanças de percentual em uma ou em várias séries temporais em colunas de um DataFrame. Isso é expresso como:

```
ts / ts.shift(1) - 1
```

Como deslocamentos triviais deixam o índice inalterado, alguns dados são descartados. Desse modo, se a frequência for conhecida, ela poderá ser passada para shift a fim de avançar os timestamps, em vez de simplesmente fazer isso com os dados:

```
In [95]: ts.shift(2, freq='M')
Out[95]:
2000-03-31 -0.066748
2000-04-30 0.838639
2000-05-31 -0.117388
2000-06-30 -0.517795
Freq: M, dtype: float64
```

Outras frequências também podem ser passadas, possibilitando certa flexibilidade no modo de avançar e retroceder os dados:

```
In [96]: ts.shift(3, freq='D')
Out[96]:
2000-02-03 -0.066748
2000-03-03 0.838639
2000-04-03 -0.117388
2000-05-03 -0.517795
dtype: float64

In [97]: ts.shift(1, freq='90T')
Out[97]:
2000-01-31 01:30:00 -0.066748
2000-02-29 01:30:00 0.838639
2000-03-31 01:30:00 -0.117388
2000-04-30 01:30:00 -0.517795
Freq: M, dtype: float64
```

O T, nesse caso, quer dizer minutos.

Deslocando datas com offsets

Os offsets de data do pandas também podem ser usados com datetime ou com objetos Timestamp:

```
In [98]: from pandas.tseries.offsets import Day, MonthEnd

In [99]: now = datetime(2011, 11, 17)

In [100]: now + 3 * Day()
Out[100]: Timestamp('2011-11-20 00:00:00')
```

Se você adicionar um offset ancorado, como MonthEnd, o primeiro incremento fará uma data “avançar” para a próxima de acordo com a regra de frequência:

```
In [101]: now + MonthEnd()
Out[101]: Timestamp('2011-11-30 00:00:00')

In [102]: now + MonthEnd(2)
Out[102]: Timestamp('2011-12-31 00:00:00')
```

Offsets ancorados podem fazer as datas “avançarem” ou retrocederem explicitamente, bastando usar seus métodos rollforward e rollback, respectivamente:

```
In [103]: offset = MonthEnd()

In [104]: offset.rollforward(now)
Out[104]: Timestamp('2011-11-30 00:00:00')

In [105]: offset.rollback(now)
Out[105]: Timestamp('2011-10-31 00:00:00')
```

Um uso criativo de offsets de datas consiste em utilizar esses métodos com groupby:

```
In [106]: ts = pd.Series(np.random.randn(20),
   .....: index=pd.date_range('1/15/2000', periods=20, freq='4d'))

In [107]: ts
Out[107]:
2000-01-15 -0.116696
2000-01-19 2.389645
2000-01-23 -0.932454
2000-01-27 -0.229331
```

```
2000-01-31 -1.140330
2000-02-04 0.439920
2000-02-08 -0.823758
2000-02-12 -0.520930
2000-02-16 0.350282
2000-02-20 0.204395
2000-02-24 0.133445
2000-02-28 0.327905
2000-03-03 0.072153
2000-03-07 0.131678
2000-03-11 -1.297459
2000-03-15 0.997747
2000-03-19 0.870955
2000-03-23 -0.991253
2000-03-27 0.151699
2000-03-31 1.266151
Freq: 4D, dtype: float64

In [108]: ts.groupby(offset.rollforward).mean()
Out[108]:
2000-01-31 -0.005833
2000-02-29 0.015894
2000-03-31 0.150209
dtype: float64
```

É claro que uma maneira mais fácil e rápida de fazer isso é usar `resample` (discutiremos esse assunto mais detalhadamente na Seção 11.6, “Reamostragem e conversão de frequências”):

```
In [109]: ts.resample('M').mean()
Out[109]:
2000-01-31 -0.005833
2000-02-29 0.015894
2000-03-31 0.150209
Freq: M, dtype: float64
```

11.4 Tratamento de fusos horários

Trabalhar com fusos horários em geral é considerado uma das partes mais desagradáveis na manipulação de séries temporais. Como resultado, muitos usuários de séries temporais preferem

trabalhar com elas em *UTC* (Coodinated Universal Time, ou Tempo Universal Coordenado), que é o sucessor do GMT (Greenwich Mean Time, ou Tempo Médio de Greenwich); o UTC atualmente é o padrão internacional. Os fusos horários são expressos como offsets em relação ao UTC; por exemplo, Nova York está quatro horas atrasada em relação ao UTC durante o horário de verão e cinco horas atrás no restante do ano.

Em Python, as informações de fuso horário são provenientes da biblioteca de terceiros pytz (que pode ser instalada com o pip ou o conda); essa biblioteca expõe o *banco de dados Olson*: uma compilação de informações sobre fusos horários no mundo. Isso é particularmente importante para dados históricos, pois as datas de transição de horário de verão (DST, isto é, Daylight Saving Time) – e até mesmo offsets de UTC – mudaram várias vezes conforme os caprichos dos governos locais. Nos Estados Unidos, os horários de transição do horário de verão mudaram várias vezes desde 1900!

Para obter informações detalhadas sobre a biblioteca pytz, será necessário consultar a documentação da biblioteca. No que concerne a este livro, o pandas encapsula as funcionalidades da pytz, de modo que você pode ignorar sua API, exceto pelos nomes dos fusos horários. Os nomes dos fusos horários podem ser encontrados interativamente e na documentação:

```
In [110]: import pytz

In [111]: pytz.common_timezones[-5:]
Out[111]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']
```

Para obter um objeto de fuso horário da pytz, utilize pytz.timezone:

```
In [112]: tz = pytz.timezone('America/New_York')

In [113]: tz
Out[113]: <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Os métodos do pandas aceitarão tanto os nomes dos fusos horários quanto esses objetos.

Localização e conversão dos fusos horários

Por padrão, as séries temporais no pandas *não consideram fusos horários*. Por exemplo, considere a série temporal a seguir:

```
In [114]: rng = pd.date_range('3/9/2012 9:30', periods=6, freq='D')

In [115]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [116]: ts
Out[116]:
2012-03-09 09:30:00 -0.202469
2012-03-10 09:30:00 0.050718
2012-03-11 09:30:00 0.639869
2012-03-12 09:30:00 0.597594
2012-03-13 09:30:00 -0.797246
2012-03-14 09:30:00 0.472879
Freq: D, dtype: float64
```

O campo tz do índice é None:

```
In [117]: print(ts.index.tz)
None
```

Intervalos de datas podem ser gerados com uma definição de fuso horário:

```
In [118]: pd.date_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')
Out[118]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
               '2012-03-15 09:30:00+00:00', '2012-03-16 09:30:00+00:00',
               '2012-03-17 09:30:00+00:00', '2012-03-18 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='D')
```

A conversão entre uma série que não considera fuso horário em outra série *localizada* é tratada pelo método tz_localize:

```
In [119]: ts
Out[119]:
2012-03-09 09:30:00 -0.202469
2012-03-10 09:30:00 0.050718
2012-03-11 09:30:00 0.639869
```

```
2012-03-12 09:30:00 0.597594
2012-03-13 09:30:00 -0.797246
2012-03-14 09:30:00 0.472879
Freq: D, dtype: float64

In [120]: ts_utc = ts.tz_localize('UTC')

In [121]: ts_utc
Out[121]:
2012-03-09 09:30:00+00:00 -0.202469
2012-03-10 09:30:00+00:00 0.050718
2012-03-11 09:30:00+00:00 0.639869
2012-03-12 09:30:00+00:00 0.597594
2012-03-13 09:30:00+00:00 -0.797246
2012-03-14 09:30:00+00:00 0.472879
Freq: D, dtype: float64

In [122]: ts_utc.index
Out[122]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='D')
```

Depois que uma série temporal for localizada em um fuso horário específico, ela poderá ser convertida para outro fuso com tz_convert:

```
In [123]: ts_utc.tz_convert('America/New_York')
Out[123]:
2012-03-09 04:30:00-05:00 -0.202469
2012-03-10 04:30:00-05:00 0.050718
2012-03-11 05:30:00-04:00 0.639869
2012-03-12 05:30:00-04:00 0.597594
2012-03-13 05:30:00-04:00 -0.797246
2012-03-14 05:30:00-04:00 0.472879
Freq: D, dtype: float64
```

No caso da série temporal anterior, que engloba uma transição de horário de verão no fuso horário America/New_York, poderíamos fazer a localização para EST e converter, por exemplo, para UTC ou para o horário de Berlim:

```
In [124]: ts_eastern = ts.tz_localize('America/New_York')

In [125]: ts_eastern.tz_convert('UTC')
Out[125]:
2012-03-09 14:30:00+00:00 -0.202469
2012-03-10 14:30:00+00:00 0.050718
2012-03-11 13:30:00+00:00 0.639869
2012-03-12 13:30:00+00:00 0.597594
2012-03-13 13:30:00+00:00 -0.797246
2012-03-14 13:30:00+00:00 0.472879
Freq: D, dtype: float64

In [126]: ts_eastern.tz_convert('Europe/Berlin')
Out[126]:
2012-03-09 15:30:00+01:00 -0.202469
2012-03-10 15:30:00+01:00 0.050718
2012-03-11 14:30:00+01:00 0.639869
2012-03-12 14:30:00+01:00 0.597594
2012-03-13 14:30:00+01:00 -0.797246
2012-03-14 14:30:00+01:00 0.472879
Freq: D, dtype: float64
```

`tz_localize` e `tz_convert` também são métodos de instância de `DatetimeIndex`:

```
In [127]: ts.index.tz_localize('Asia/Shanghai')
Out[127]:
DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-10 09:30:00+08:00',
               '2012-03-11 09:30:00+08:00', '2012-03-12 09:30:00+08:00',
               '2012-03-13 09:30:00+08:00', '2012-03-14 09:30:00+08:00'],
              dtype='datetime64[ns, Asia/Shanghai]', freq='D')
```

A localização de timestamps que não consideram horário de verão também implica a verificação de horários ambíguos ou inexistentes por causa das transições de horário de verão.

Operações com objetos Timestamp que consideram fusos horários

De modo semelhante às séries temporais e aos intervalos de datas,

os objetos Timestamp individuais podem ser localizados, passando para dados que consideram fuso horário, e podem convertidos de um fuso horário para outro:

```
In [128]: stamp = pd.Timestamp('2011-03-12 04:00')

In [129]: stamp_utc = stamp.tz_localize('utc')

In [130]: stamp_utc.tz_convert('America/New_York')
Out[130]: Timestamp('2011-03-11 23:00:00-0500', tz='America/New_York')
```

Também podemos passar um fuso horário quando criamos o Timestamp:

```
In [131]: stamp_moscow = pd.Timestamp('2011-03-12 04:00',
tz='Europe/Moscow')

In [132]: stamp_moscow
Out[132]: Timestamp('2011-03-12 04:00:00+0300', tz='Europe/Moscow')
```

Objetos Timestamp que consideram fuso horário armazenam internamente um valor de timestamp UTC na forma de nanossegundos desde a Era Unix (1 de janeiro de 1970); esse valor de UTC não varia entre conversões de fusos horários:

```
In [133]: stamp_utc.value
Out[133]: 1299902400000000000

In [134]: stamp_utc.tz_convert('America/New_York').value
Out[134]: 1299902400000000000
```

Ao executar operações aritméticas com tempo usando objetos DateOffset do pandas, as transições de horário de verão são respeitadas sempre que possível. No exemplo a seguir, construiremos timestamps que ocorrem imediatamente antes de transições de horário de verão (para a frente e para trás). Inicialmente, trinta minutos antes da transição para o horário de verão:

```
In [135]: from pandas.tseries.offsets import Hour

In [136]: stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern')
```

```
In [137]: stamp
Out[137]: Timestamp('2012-03-12 01:30:00-0400', tz='US/Eastern')

In [138]: stamp + Hour()
Out[138]: Timestamp('2012-03-12 02:30:00-0400', tz='US/Eastern')
```

Em seguida, noventa minutos antes de sair do horário de verão:

```
In [139]: stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')

In [140]: stamp
Out[140]: Timestamp('2012-11-04 00:30:00-0400', tz='US/Eastern')

In [141]: stamp + 2 * Hour()
Out[141]: Timestamp('2012-11-04 01:30:00-0500', tz='US/Eastern')
```

Operações entre fusos horários diferentes

Se duas séries temporais com fusos horários diferentes forem combinadas, o resultado será UTC. Como os timestamps são armazenados internamente em UTC, essa é uma operação simples e não exige nenhuma conversão:

```
In [142]: rng = pd.date_range('3/7/2012 9:30', periods=10, freq='B')

In [143]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [144]: ts
Out[144]:
2012-03-07 09:30:00 0.522356
2012-03-08 09:30:00 -0.546348
2012-03-09 09:30:00 -0.733537
2012-03-12 09:30:00 1.302736
2012-03-13 09:30:00 0.022199
2012-03-14 09:30:00 0.364287
2012-03-15 09:30:00 -0.922839
2012-03-16 09:30:00 0.312656
2012-03-19 09:30:00 -1.128497
2012-03-20 09:30:00 -0.333488
Freq: B, dtype: float64

In [145]: ts1 = ts[:7].tz_localize('Europe/London')
```

```
In [146]: ts2 = ts1[2:].tz_convert('Europe/Moscow')

In [147]: result = ts1 + ts2

In [148]: result.index
Out[148]:
DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-08 09:30:00+00:00',
               '2012-03-09 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
               '2012-03-15 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='B')
```

11.5 Períodos e aritmética com períodos

Períodos representam intervalos de tempo, como dias, meses, trimestres ou anos. A classe Period representa esse tipo de dado, exigindo uma string ou um inteiro e uma frequência da Tabela 11.4:

```
In [149]: p = pd.Period(2007, freq='A-DEC')

In [150]: p
Out[150]: Period('2007', 'A-DEC')
```

Nesse caso, o objeto Period representa o intervalo de tempo completo de 1 de janeiro de 2007 a 31 de dezembro de 2007, inclusive. De modo conveniente, somar e subtrair inteiros de períodos tem o efeito de deslocá-los pelas suas frequências:

```
In [151]: p + 5
Out[151]: Period('2012', 'A-DEC')

In [152]: p - 2
Out[152]: Period('2005', 'A-DEC')
```

Se dois períodos tiverem a mesma frequência, sua diferença será o número de unidades entre eles:

```
In [153]: pd.Period('2014', freq='A-DEC') - p
Out[153]: 7
```

Intervalos de períodos regulares podem ser construídos com a função period_range:

```
In [154]: rng = pd.period_range('2000-01-01', '2000-06-30', freq='M')

In [155]: rng
Out[155]: PeriodIndex(['2000-01', '2000-02', '2000-03', '2000-04', '2000-05', '20
00-06'], dtype='period[M]', freq='M')
```

A classe PeriodIndex armazena uma sequência de períodos, e pode servir como índice de um eixo em qualquer estrutura de dados do pandas:

```
In [156]: pd.Series(np.random.randn(6), index=rng)
Out[156]:
2000-01 -0.514551
2000-02 -0.559782
2000-03 -0.783408
2000-04 -1.797685
2000-05 -0.172670
2000-06 0.680215
Freq: M, dtype: float64
```

Se você tiver um array de strings, a classe PeriodIndex também poderá ser usada:

```
In [157]: values = ['2001Q3', '2002Q2', '2003Q1']

In [158]: index = pd.PeriodIndex(values, freq='Q-DEC')

In [159]: index
Out[159]: PeriodIndex(['2001Q3', '2002Q2', '2003Q1'], dtype='period[Q-DEC]',
freq
='Q-DEC')
```

Conversão de frequência de períodos

Períodos e objetos PeriodIndex podem ser convertidos em outra frequência com seu método asfreq. Como exemplo, suponha que tivéssemos um período anual e quiséssemos convertê-lo em um período mensal, seja no início ou no final do ano. É uma operação razoavelmente simples:

```
In [160]: p = pd.Period('2007', freq='A-DEC')

In [161]: p
```

```
Out[161]: Period('2007', 'A-DEC')

In [162]: p.asfreq('M', how='start')
Out[162]: Period('2007-01', 'M')

In [163]: p.asfreq('M', how='end')
Out[163]: Period('2007-12', 'M')
```

Podemos pensar em Period('2007','A-DEC') como uma espécie de cursor que aponta para um intervalo de tempo, subdividido em períodos mensais. Veja a Figura 11.1 que apresenta uma ilustração desse caso. Para um *ano fiscal* que termine em um mês diferente de dezembro, os subperíodos mensais correspondentes são diferentes:

```
In [164]: p = pd.Period('2007', freq='A-JUN')

In [165]: p
Out[165]: Period('2007', 'A-JUN')

In [166]: p.asfreq('M', 'start')
Out[166]: Period('2006-07', 'M')

In [167]: p.asfreq('M', 'end')
Out[167]: Period('2007-06', 'M')
```

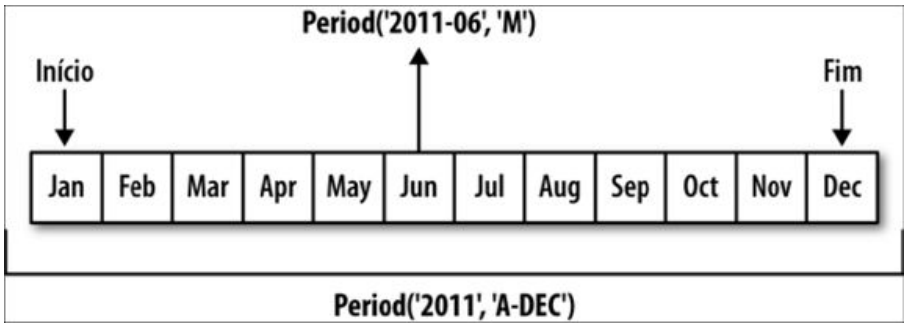


Figura 11.1 – Ilustração da conversão de frequência de período.

Quando você estiver convertendo de uma frequência alta para uma frequência baixa, o pandas determinará o superperíodo dependendo do local a que o subperíodo “pertença”. Por exemplo, em uma frequência A-JUN, o mês Aug-2007 na verdade faz parte do período 2008:

```
In [168]: p = pd.Period('Aug-2007', 'M')
```

```
In [169]: p.asfreq('A-JUN')
Out[169]: Period('2008', 'A-JUN')
```

Objetos PeriodIndex ou séries temporais completas podem ser convertidos, de modo semelhante, com a mesma semântica:

```
In [170]: rng = pd.period_range('2006', '2009', freq='A-DEC')

In [171]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [172]: ts
Out[172]:
2006 1.607578
2007 0.200381
2008 -0.834068
2009 -0.302988
Freq: A-DEC, dtype: float64

In [173]: ts.asfreq('M', how='start')
Out[173]:
2006-01 1.607578
2007-01 0.200381
2008-01 -0.834068
2009-01 -0.302988
Freq: M, dtype: float64
```

Nesse caso, os períodos anuais são substituídos por períodos mensais correspondentes ao primeiro mês que se enquadrar em cada período anual. Se, por outro lado, quiséssemos o último dia útil de cada ano, poderíamos utilizar a frequência 'B' e informar que queremos o final do período:

```
In [174]: ts.asfreq('B', how='end')
Out[174]:
2006-12-29 1.607578
2007-12-31 0.200381
2008-12-31 -0.834068
2009-12-31 -0.302988
Freq: B, dtype: float64
```

Frequências de período trimestrais

Dados trimestrais são padrões em contabilidade, finanças e em outras áreas. Boa parte dos dados trimestrais são informados em relação ao *final de um ano fiscal*, geralmente o último dia do calendário ou o último dia útil de um dos doze meses do ano. Assim, o período 2012Q4 tem um significado diferente, dependendo do final do ano fiscal. O pandas aceita todas as 12 possíveis frequências trimestrais, de Q-JAN a Q-DEC:

```
In [175]: p = pd.Period('2012Q4', freq='Q-JAN')

In [176]: p
Out[176]: Period('2012Q4', 'Q-JAN')
```

No caso do ano fiscal que termine em janeiro, 2012Q4 se estende de novembro a janeiro, o que pode ser verificado fazendo a sua conversão para uma frequência diária. Veja a Figura 11.2 que apresenta uma ilustração.

Figura 11.2 – Conversões de diferentes frequências trimestrais.

```
In [177]: p.asfreq('D', 'start')
Out[177]: Period('2011-11-01', 'D')

In [178]: p.asfreq('D', 'end')
Out[178]: Period('2012-01-31', 'D')
```

Assim, é possível fazer operações aritméticas com períodos facilmente; por exemplo, para obter um timestamp às 16h horas do penúltimo dia útil do trimestre, você poderia fazer o seguinte:

```
In [179]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60

In [180]: p4pm
Out[180]: Period('2012-01-30 16:00', 'T')
```

```
In [181]: p4pm.to_timestamp()
Out[181]: Timestamp('2012-01-30 16:00:00')
```

Podemos gerar intervalos trimestrais usando period_range. As operações aritméticas são também idênticas:

```
In [182]: rng = pd.period_range('2011Q3', '2012Q4', freq='Q-JAN')

In [183]: ts = pd.Series(np.arange(len(rng)), index=rng)

In [184]: ts
Out[184]:
2011Q3 0
2011Q4 1
2012Q1 2
2012Q2 3
2012Q3 4
2012Q4 5
Freq: Q-JAN, dtype: int64

In [185]: new_rng = (rng.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60

In [186]: ts.index = new_rng.to_timestamp()

In [187]: ts
Out[187]:
2010-10-28 16:00:00 0
2011-01-28 16:00:00 1
2011-04-28 16:00:00 2
2011-07-28 16:00:00 3
2011-10-28 16:00:00 4
2012-01-30 16:00:00 5
dtype: int64
```

Convertendo timestamps para períodos (e vice-versa)

Objetos Series e DataFrame indexados por timestamps podem ser convertidos para períodos usando o método to_period:

```
In [188]: rng = pd.date_range('2000-01-01', periods=3, freq='M')
```

```
In [189]: ts = pd.Series(np.random.randn(3), index=rng)

In [190]: ts
Out[190]:
2000-01-31 1.663261
2000-02-29 -0.996206
2000-03-31 1.521760
Freq: M, dtype: float64

In [191]: pts = ts.to_period()

In [192]: pts
Out[192]:
2000-01 1.663261
2000-02 -0.996206
2000-03 1.521760
Freq: M, dtype: float64
```

Como os períodos se referem a intervalos de tempo que não se sobrepõem, um timestamp só pode pertencer a um único período para uma dada frequência. Embora, por padrão, a frequência do novo PeriodIndex seja inferida a partir dos timestamps, você poderá especificar qualquer frequência desejada. Também não há problemas em ter períodos duplicados no resultado:

```
In [193]: rng = pd.date_range('1/29/2000', periods=6, freq='D')

In [194]: ts2 = pd.Series(np.random.randn(6), index=rng)

In [195]: ts2
Out[195]:
2000-01-29 0.244175
2000-01-30 0.423331
2000-01-31 -0.654040
2000-02-01 2.089154
2000-02-02 -0.060220
2000-02-03 -0.167933
Freq: D, dtype: float64

In [196]: ts2.to_period('M')
```

```
Out[196]:
2000-01 0.244175
2000-01 0.423331
2000-01 -0.654040
2000-02 2.089154
2000-02 -0.060220
2000-02 -0.167933
Freq: M, dtype: float64
```

Para fazer a conversão de volta para timestamps, utilize to_timestamp:

```
In [197]: pts = ts2.to_period()

In [198]: pts
Out[198]:
2000-01-29 0.244175
2000-01-30 0.423331
2000-01-31 -0.654040
2000-02-01 2.089154
2000-02-02 -0.060220
2000-02-03 -0.167933
Freq: D, dtype: float64

In [199]: pts.to_timestamp(how='end')
Out[199]:
2000-01-29 0.244175
2000-01-30 0.423331
2000-01-31 -0.654040
2000-02-01 2.089154
2000-02-02 -0.060220
2000-02-03 -0.167933
Freq: D, dtype: float64
```

Criando um PeriodIndex a partir de arrays

Conjuntos de dados com frequência fixa às vezes são armazenados com informações de intervalos de tempo espalhadas em várias colunas. Por exemplo, no conjunto de dados de macroeconomia a seguir, o ano e o trimestre estão em colunas diferentes:

```
In [200]: data = pd.read_csv('examples/macrodata.csv')
```

```
In [201]: data.head(5)
Out[201]:
     year quarter realgdp realcons realinv realgovt realdpi cpi \
0 1959.0 1.0 2710.349 1707.4 286.898 470.045 1886.9 28.98
1 1959.0 2.0 2778.801 1733.7 310.859 481.301 1919.7 29.15
2 1959.0 3.0 2775.488 1751.8 289.226 491.260 1916.4 29.35
3 1959.0 4.0 2785.204 1753.7 299.356 484.052 1931.3 29.37
4 1960.0 1.0 2847.699 1770.5 331.722 462.199 1955.5 29.54
      m1 tbilrate unemp pop infl realint
0 139.7 2.82 5.8 177.146 0.00 0.00
1 141.7 3.08 5.1 177.830 2.34 0.74
2 140.5 3.82 5.3 178.657 2.74 1.09
3 140.0 4.33 5.6 179.386 0.27 4.06
4 139.6 3.50 5.2 180.007 2.31 1.19

In [202]: data.year
Out[202]:
0 1959.0
1 1959.0
2 1959.0
3 1959.0
4 1960.0
5 1960.0
6 1960.0
7 1960.0
8 1961.0
9 1961.0
        ...
193 2007.0
194 2007.0
195 2007.0
196 2008.0
197 2008.0
198 2008.0
199 2008.0
200 2009.0
201 2009.0
202 2009.0
Name: year, Length: 203, dtype: float64

In [203]: data.quarter
```

```
Out[203]:
0 1.0
1 2.0
2 3.0
3 4.0
4 1.0
5 2.0
6 3.0
7 4.0
8 1.0
9 2.0
    ...
193 2.0
194 3.0
195 4.0
196 1.0
197 2.0
198 3.0
199 4.0
200 1.0
201 2.0
202 3.0
Name: quarter, Length: 203, dtype: float64
```

Ao passar esses arrays para PeriodIndex com uma frequência, podemos combiná-los a fim de compor um índice para o DataFrame:

```
In [204]: index = pd.PeriodIndex(year=data.year, quarter=data.quarter,
   .....: freq='Q-DEC')

In [205]: index
Out[205]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', length=203, freq='Q-DEC')

In [206]: data.index = index

In [207]: data.infl
```

```
Out[207]:
1959Q1 0.00
1959Q2 2.34
1959Q3 2.74
1959Q4 0.27
1960Q1 2.31
1960Q2 0.14
1960Q3 2.70
1960Q4 1.21
1961Q1 -0.40
1961Q2 1.47
       ...
2007Q2 2.75
2007Q3 3.45
2007Q4 6.38
2008Q1 2.82
2008Q2 8.53
2008Q3 -3.16
2008Q4 -8.79
2009Q1 0.94
2009Q2 3.37
2009Q3 3.56
Freq: Q-DEC, Name: infl, Length: 203, dtype: float64
```

11.6 Reamostragem e conversão de frequências

A *reamostragem* (resampling) refere-se ao processo de converter uma série temporal de uma frequência para outra. Agregar dados de frequências mais altas para frequências mais baixas é um processo chamado *downsampling* (subamostragem), enquanto converter de uma frequência mais baixa para uma frequência mais alta é chamado de *upsampling* (sobreamostragem). Nem toda reamostragem se enquadra em uma dessas categorias; por exemplo, converter `W-WED` (semanalmente às quartas-feiras) para `W-FRI` não é um upsampling nem um downsampling.

Os objetos do pandas oferecem um método `resample` – a função que representa a força de trabalho para todas as conversões de frequência. `resample` tem uma API semelhante a `groupby`; chame

resample para agrupar os dados e, em seguida, chame uma função de agregação:

```
In [208]: rng = pd.date_range('2000-01-01', periods=100, freq='D')

In [209]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [210]: ts
Out[210]:
2000-01-01 0.631634
2000-01-02 -1.594313
2000-01-03 -1.519937
2000-01-04 1.108752
2000-01-05 1.255853
2000-01-06 -0.024330
2000-01-07 -2.047939
2000-01-08 -0.272657
2000-01-09 -1.692615
2000-01-10 1.423830
               ...
2000-03-31 -0.007852
2000-04-01 -1.638806
2000-04-02 1.401227
2000-04-03 1.758539
2000-04-04 0.628932
2000-04-05 -0.423776
2000-04-06 0.789740
2000-04-07 0.937568
2000-04-08 -2.253294
2000-04-09 -1.772919
Freq: D, Length: 100, dtype: float64

In [211]: ts.resample('M').mean()
Out[211]:
2000-01-31 -0.165893
2000-02-29 0.078606
2000-03-31 0.223811
2000-04-30 -0.063643
Freq: M, dtype: float64

In [212]: ts.resample('M', kind='period').mean()
```

```
Out[212]:
2000-01 -0.165893
2000-02 0.078606
2000-03 0.223811
2000-04 -0.063643
Freq: M, dtype: float64
```

resample é um método flexível e de alto desempenho, e pode ser utilizado para processar séries temporais bem grandes. Os exemplos nas próximas seções mostram sua semântica e seu uso. A Tabela 11.5 resume algumas de suas opções.

Tabela 11.5 – Argumentos do método resample

Argumento	Descrição
freq	String ou DateOffset indicando a frequência de reamostragem desejada (por exemplo, 'M', '5min' ou Second(15))
axis	Eixo para a reamostragem; o default é axis=0
fill_method	Como interpolar em um upsampling, como em 'ffill' ou 'bfill'; por padrão, não faz nenhuma interpolação
closed	Em um downsampling, a extremidade de cada intervalo que está fechada (inclusive), 'right' ou 'left'
label	No downsampling, como rotular o resultado da agregação, com a fronteira de compartimento (bin) 'right' ou 'left' (por exemplo, o intervalo de cinco minutos de 9:30 a 9:35 poderia ser chamado de 9:30 ou 9:35)
loffset	Ajuste de tempo para os rótulos de compartimentos, por exemplo, '-1s' / Second(-1) para deslocar os rótulos das agregações para um segundo mais cedo
limit	Em preenchimentos para a frente (forward) ou para trás (backward), é o número máximo de períodos a serem preenchidos
kind	Agrega para períodos ('period') ou timestamps ('timestamp'); o default é o tipo de índice da série temporal
convention	Ao reamostrar períodos, é a convenção ('start' ou 'end') para converter o período de baixa frequência para alta frequência; o default é 'end'

Downsampling

Fazer uma agregação de dados para uma frequência regular e mais

baixa é uma tarefa bem comum em séries temporais. Os dados que você estiver agregando não precisam ter frequências fixas; a frequência desejada define as *fronteiras dos compartimentos* (bin edges) usadas para fatiar as séries temporais em partes para a agregação. Por exemplo, para uma conversão em uma frequência mensal, 'M' ou 'BM', você deve dividir os dados em intervalos de um mês. Dizemos que cada intervalo é *meio aberto* (half-open); um ponto de dado só pode pertencer a um intervalo, e a união dos intervalos compõe o período de tempo total. Há dois aspectos em que devemos pensar quando usamos resample para fazer um downsampling dos dados:

- qual é o lado de cada intervalo que está *fechado*;
- como atribuir rótulos a cada compartimento de agregação, seja com o início do intervalo ou com o fim.

Para ilustrar, vamos observar alguns dados de um minuto:

```
In [213]: rng = pd.date_range('2000-01-01', periods=12, freq='T')

In [214]: ts = pd.Series(np.arange(12), index=rng)

In [215]: ts
Out[215]:
2000-01-01 00:00:00 0
2000-01-01 00:01:00 1
2000-01-01 00:02:00 2
2000-01-01 00:03:00 3
2000-01-01 00:04:00 4
2000-01-01 00:05:00 5
2000-01-01 00:06:00 6
2000-01-01 00:07:00 7
2000-01-01 00:08:00 8
2000-01-01 00:09:00 9
2000-01-01 00:10:00 10
2000-01-01 00:11:00 11
Freq: T, dtype: int64
```

Suponha que você quisesse agregar esses dados em porções de cinco minutos ou em *barras*, tomando a soma de cada grupo:

```
In [216]: ts.resample('5min', closed='right').sum()
Out[216]:
1999-12-31 23:55:00 0
2000-01-01 00:00:00 15
2000-01-01 00:05:00 40
2000-01-01 00:10:00 11
Freq: 5T, dtype: int64
```

A frequência que você especificar define as fronteiras dos compartimentos em incrementos de cinco minutos. Por padrão, a fronteira *esquerda* do compartimento é inclusiva, portanto o valor 00:00 está incluído no intervalo de 00:00 a 00:05[1]. Passar closed='right' altera o intervalo para que seja fechado à direita:

```
In [217]: ts.resample('5min', closed='right').sum()
Out[217]:
1999-12-31 23:55:00 0
2000-01-01 00:00:00 15
2000-01-01 00:05:00 40
2000-01-01 00:10:00 11
Freq: 5T, dtype: int64
```

A série temporal resultante recebe rótulos com base nos timestamps do lado esquerdo de cada compartimento. Ao passar label='right', podemos atribuir rótulos a eles com a fronteira direita do compartimento:

```
In [218]: ts.resample('5min', closed='right', label='right').sum()
Out[218]:
2000-01-01 00:00:00 0
2000-01-01 00:05:00 15
2000-01-01 00:10:00 40
2000-01-01 00:15:00 11
Freq: 5T, dtype: int64
```

Veja a Figura 11.3 que apresenta uma ilustração dos dados com frequência de minuto, sujeitos a uma reamostragem para uma frequência de cinco minutos.

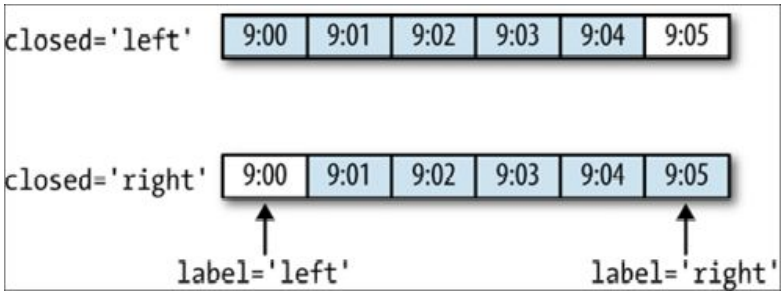


Figura 11.3 – Ilustração de uma reamostragem de cinco minutos, com convenções para closed e label.

Por fim, talvez você queira deslocar o índice dos resultados de certo valor, por exemplo, subtraindo um segundo da fronteira esquerda a fim de deixar mais claro a qual intervalo o timestamp se refere. Para isso, passe um offset na forma de string ou de data para loffset:

```
In [219]: ts.resample('5min', closed='right',
   .....: label='right', loffset='-1s').sum()
Out[219]:
1999-12-31 23:59:59 0
2000-01-01 00:04:59 15
2000-01-01 00:09:59 40
2000-01-01 00:14:59 11
Freq: 5T, dtype: int64
```

Você também poderia ter obtido o efeito de loffset se chamasse o método shift no resultado, sem o loffset.

Reamostragem OHLC (Open-High-Low-Close)

Na área financeira, uma forma conhecida de fazer uma agregação em uma série temporal é calcular quatro valores para cada bucket: o primeiro (open, isto é, de abertura), o último (close, isto é, de fechamento), o máximo (high) e o mínimo (low). Ao usar a função de agregação ohlc, você obterá um DataFrame com colunas contendo essas quatro agregações, que são calculadas eficazmente em uma única varredura dos dados:

```
In [220]: ts.resample('5min').ohlc()
Out[220]:
                     open high low close
2000-01-01 00:00:00 0 4 0 4
```

```
2000-01-01 00:05:00 5 9 5 9
2000-01-01 00:10:00 10 11 10 11
```

Upsampling e interpolação

Ao fazer a conversão de uma frequência baixa para uma frequência mais alta, nenhuma agregação será necessária. Vamos considerar um DataFrame com alguns dados semanais:

```
In [221]: frame = pd.DataFrame(np.random.randn(2, 4),
   .....: index=pd.date_range('1/1/2000', periods=2,
   .....: freq='W-WED'),
   .....: columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [222]: frame
Out[222]:
                Colorado Texas New York Ohio
2000-01-05 -0.896431 0.677263 0.036503 0.087102
2000-01-12 -0.046662 0.927238 0.482284 -0.867130
```

Ao usar uma função de agregação com esses dados, há apenas um valor por grupo, e valores ausentes resultam em lacunas. Usamos o método `asfreq` para converter para a frequência mais alta, sem qualquer agregação:

```
In [223]: df_daily = frame.resample('D').asfreq()

In [224]: df_daily
Out[224]:
                Colorado Texas New York Ohio
2000-01-05 -0.896431 0.677263 0.036503 0.087102
2000-01-06 NaN NaN NaN NaN
2000-01-07 NaN NaN NaN NaN
2000-01-08 NaN NaN NaN NaN
2000-01-09 NaN NaN NaN NaN
2000-01-10 NaN NaN NaN NaN
2000-01-11 NaN NaN NaN NaN
2000-01-12 -0.046662 0.927238 0.482284 -0.867130
```

Suponha que você quisesse fazer um preenchimento para a frente de cada valor semanal nos dias diferentes de quarta-feira. Os mesmos métodos de preenchimento ou de interpolação disponíveis

nos métodos fillna e reindex estão disponíveis para a reamostragem:

```
In [225]: frame.resample('D').ffill()
Out[225]:
            Colorado Texas New York Ohio
2000-01-05 -0.896431 0.677263 0.036503 0.087102
2000-01-06 -0.896431 0.677263 0.036503 0.087102
2000-01-07 -0.896431 0.677263 0.036503 0.087102
2000-01-08 -0.896431 0.677263 0.036503 0.087102
2000-01-09 -0.896431 0.677263 0.036503 0.087102
2000-01-10 -0.896431 0.677263 0.036503 0.087102
2000-01-11 -0.896431 0.677263 0.036503 0.087102
2000-01-12 -0.046662 0.927238 0.482284 -0.867130
```

De modo semelhante, você pode optar por preencher somente certo número de períodos para a frente a fim de limitar até que ponto deve continuar usando o valor observado:

```
In [226]: frame.resample('D').ffill(limit=2)
Out[226]:
            Colorado Texas New York Ohio
2000-01-05 -0.896431 0.677263 0.036503 0.087102
2000-01-06 -0.896431 0.677263 0.036503 0.087102
2000-01-07 -0.896431 0.677263 0.036503 0.087102
2000-01-08 NaN NaN NaN NaN
2000-01-09 NaN NaN NaN NaN
2000-01-10 NaN NaN NaN NaN
2000-01-11 NaN NaN NaN NaN
2000-01-12 -0.046662 0.927238 0.482284 -0.867130
```

Observe que o novo índice de datas não se sobrepõe de modo algum com o antigo:

```
In [227]: frame.resample('W-THU').ffill()
Out[227]:
            Colorado Texas New York Ohio
2000-01-06 -0.896431 0.677263 0.036503 0.087102
2000-01-13 -0.046662 0.927238 0.482284 -0.867130
```

Reamostragem com períodos

Fazer uma reamostragem de dados indexados por períodos é semelhante a timestamps:

```
In [228]: frame = pd.DataFrame(np.random.randn(24, 4),
   .....: index=pd.period_range('1-2000', '12-2001',
   .....: freq='M'),
   .....: columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [229]: frame[:5]
Out[229]:
            Colorado Texas New York Ohio
2000-01 0.493841 -0.155434 1.397286 1.507055
2000-02 -1.179442 0.443171 1.395676 -0.529658
2000-03 0.787358 0.248845 0.743239 1.267746
2000-04 1.302395 -0.272154 -0.051532 -0.467740
2000-05 -1.040816 0.426419 0.312945 -1.115689

In [230]: annual_frame = frame.resample('A-DEC').mean()

In [231]: annual_frame
Out[231]:
          Colorado Texas New York Ohio
2000 0.556703 0.016631 0.111873 -0.027445
2001 0.046303 0.163344 0.251503 -0.157276
```

Fazer um upsampling é uma tarefa com mais nuances, pois você deve tomar uma decisão sobre em qual extremidade do intervalo de tempo na nova frequência devem ser colocados os valores antes da reamostragem, como no método asfreq. O default do argumento convention é 'start', mas também pode ser 'end':

```
# Q-DEC: trimestral, com o ano terminando em dezembro
In [232]: annual_frame.resample('Q-DEC').ffill()
Out[232]:
            Colorado Texas New York Ohio
2000Q1 0.556703 0.016631 0.111873 -0.027445
2000Q2 0.556703 0.016631 0.111873 -0.027445
2000Q3 0.556703 0.016631 0.111873 -0.027445
2000Q4 0.556703 0.016631 0.111873 -0.027445
2001Q1 0.046303 0.163344 0.251503 -0.157276
2001Q2 0.046303 0.163344 0.251503 -0.157276
2001Q3 0.046303 0.163344 0.251503 -0.157276
2001Q4 0.046303 0.163344 0.251503 -0.157276
```

```
In [233]: annual_frame.resample('Q-DEC', convention='end').ffill()
Out[233]:
         Colorado Texas New York Ohio
2000Q4 0.556703 0.016631 0.111873 -0.027445
2001Q1 0.556703 0.016631 0.111873 -0.027445
2001Q2 0.556703 0.016631 0.111873 -0.027445
2001Q3 0.556703 0.016631 0.111873 -0.027445
2001Q4 0.046303 0.163344 0.251503 -0.157276
```

Como os períodos se referem a intervalos de tempo, as regras sobre upsampling e downsampling são mais rígidas:

- No downsampling, a frequência-alvo deve ser um *subperíodo* da frequência original.
- No upsampling, a frequência-alvo deve ser um *superperíodo* da frequência original.

Se essas regras não forem satisfeitas, uma exceção será lançada. Isso afeta principalmente as frequências trimestrais, anuais e semanais; por exemplo, os intervalos de tempo definidos por Q-MAR só se alinharão com A-MAR, A-JUN, A-SEP e A-DEC:

```
In [234]: annual_frame.resample('Q-MAR').ffill()
Out[234]:
         Colorado Texas New York Ohio
2000Q4 0.556703 0.016631 0.111873 -0.027445
2001Q1 0.556703 0.016631 0.111873 -0.027445
2001Q2 0.556703 0.016631 0.111873 -0.027445
2001Q3 0.556703 0.016631 0.111873 -0.027445
2001Q4 0.046303 0.163344 0.251503 -0.157276
2002Q1 0.046303 0.163344 0.251503 -0.157276
2002Q2 0.046303 0.163344 0.251503 -0.157276
2002Q3 0.046303 0.163344 0.251503 -0.157276
```

11.7 Funções de janela móvel

Uma classe importante de transformações de array usada em operações de séries temporais são as estatísticas e outras funções avaliadas em uma janela deslizante ou com pesos exponencialmente decrescentes. Isso pode ser útil para suavizar

ruídos ou dados com lacunas. Chamo-as de *funções de janela móvel* (moving window functions), apesar de incluírem funções sem uma janela de tamanho fixo, como a média móvel exponencialmente ponderada. Assim como outras funções estatísticas, essas também excluem automaticamente os dados ausentes.

Antes de explorar esse assunto com mais detalhes, podemos carregar alguns dados de séries temporais e fazer uma reamostragem deles para a frequência de dias úteis:

```
In [235]: close_px_all = pd.read_csv('examples/stock_px_2.csv',
   .....: parse_dates=True, index_col=0)

In [236]: close_px = close_px_all[['AAPL', 'MSFT', 'XOM']]

In [237]: close_px = close_px.resample('B').ffill()
```

Apresentarei agora o operador rolling, que se comporta de modo semelhante a resample e a groupby. Ele pode ser chamado em uma Series ou um DataFrame, junto com uma window (expressa como um número de períodos; veja a Figura 11.4 que apresenta a plotagem gerada):

```
In [238]: close_px.AAPL.plot()
Out[238]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa59eaf0b00>

In [239]: close_px.AAPL.rolling(250).mean().plot()
```

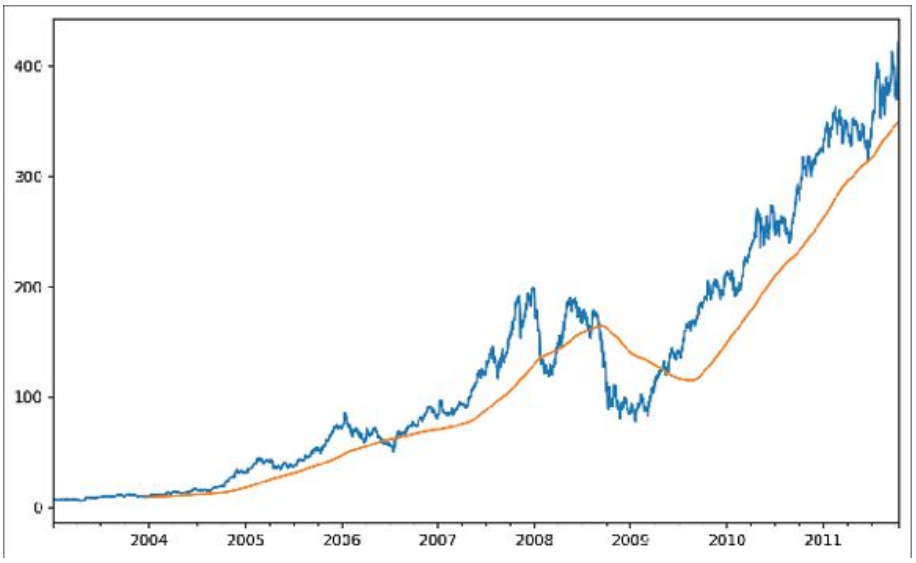


Figure 11-4 – Preço da Apple com média móvel de 250 dias.

A expressão rolling(250) é semelhante em comportamento a groupby, mas, em vez de agrupar, ela cria um objeto que permite agrupamentos em uma janela deslizante de 250 dias. Assim, nesse exemplo, temos a média da janela móvel de 250 dias do preço das ações da Apple.

Por padrão, as funções de rolagem exigem que todos os valores da janela sejam diferentes de NA. Esse comportamento pode ser alterado para levar em consideração dados ausentes e, em particular, o fato de que você terá menos que window períodos de dados no início da série temporal (veja a Figura 11.5):

```
In [241]: appl_std250 = close_px.AAPL.rolling(250, min_periods=10).std()

In [242]: appl_std250[5:12]
Out[242]:
2003-01-09 NaN
2003-01-10 NaN
2003-01-13 NaN
2003-01-14 NaN
2003-01-15 0.077496
2003-01-16 0.074760
2003-01-17 0.112368
Freq: B, Name: AAPL, dtype: float64

In [243]: appl_std250.plot()
```

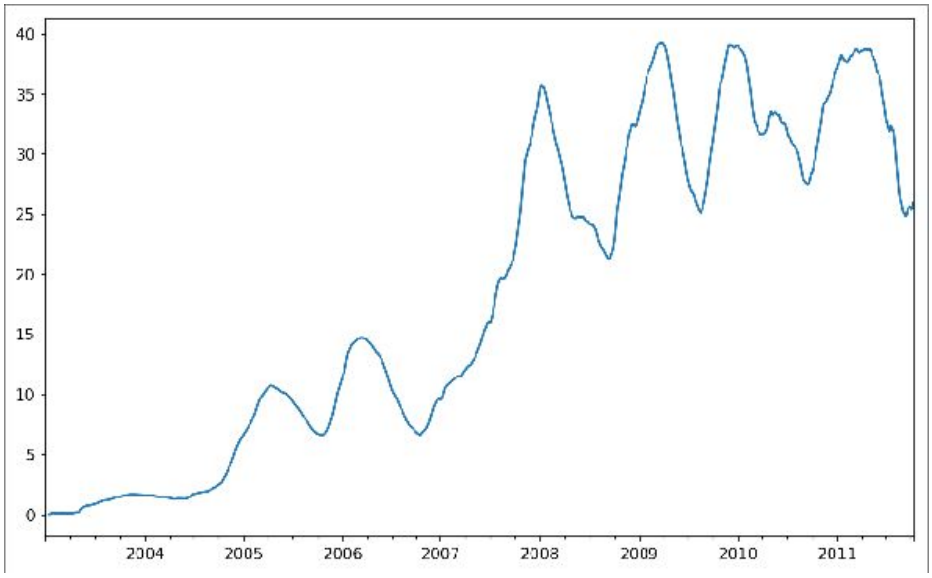


Figura 11.5 – Desvio-padrão do retorno diário de 250 dias da Apple.

Para calcular uma *média de janela em expansão* (expanding window mean), utilize o operador expanding no lugar de rolling. A média de expansão inicia a janela de tempo no começo da série temporal e aumenta o tamanho da janela até que ela englobe toda a série. Uma média de janela em expansão na série temporal apple_std250 tem o seguinte aspecto:

```
In [244]: expanding_mean = appl_std250.expanding().mean()
```

Chamar uma função de janela móvel em um DataFrame aplica a transformação em cada coluna (veja a Figura 11.6):

```
In [246]: close_px.rolling(60).mean().plot(logy=True)
```

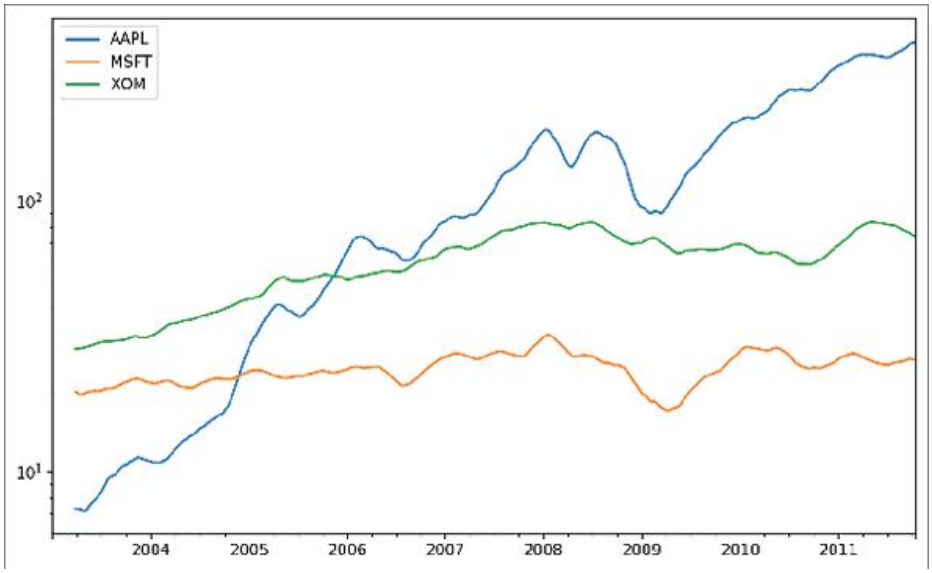


Figura 11.6 – Preços das ações em média móvel de 60 dias (log eixo y).

A função rolling também aceita uma string informando um offset de tempo de tamanho fixo em vez de um número definido de períodos. Usar essa notação pode ser conveniente em séries temporais irregulares. Essas são as mesmas strings que podemos passar para resample. Por exemplo, poderíamos calcular uma média móvel de 20 dias assim:

```
In [247]: close_px.rolling('20D').mean()
Out[247]:
                AAPL MSFT XOM
2003-01-02 7.400000 21.110000 29.220000
2003-01-03 7.425000 21.125000 29.230000
2003-01-06 7.433333 21.256667 29.473333
2003-01-07 7.432500 21.425000 29.342500
2003-01-08 7.402000 21.402000 29.240000
2003-01-09 7.391667 21.490000 29.273333
2003-01-10 7.387143 21.558571 29.238571
2003-01-13 7.378750 21.633750 29.197500
2003-01-14 7.370000 21.717778 29.194444
2003-01-15 7.355000 21.757000 29.152000
... ... ... ...
2011-10-03 398.002143 25.890714 72.413571
2011-10-04 396.802143 25.807857 72.427143
2011-10-05 395.751429 25.729286 72.422857
```

```
2011-10-06 394.099286 25.673571 72.375714
2011-10-07 392.479333 25.712000 72.454667
2011-10-10 389.351429 25.602143 72.527857
2011-10-11 388.505000 25.674286 72.835000
2011-10-12 388.531429 25.810000 73.400714
2011-10-13 388.826429 25.961429 73.905000
2011-10-14 391.038000 26.048667 74.185333
[2292 rows x 3 columns]
```

Funções exponencialmente ponderadas

Uma alternativa ao uso de um tamanho estático de janela com observações igualmente ponderadas consiste em especificar um *fator de decaimento* (decay factor) constante para atribuir mais peso às observações mais recentes. Há algumas maneiras de especificar o fator de decaimento. Uma opção popular é usar um *span*, que deixa o resultado comparável a uma função simples de janela móvel com o tamanho da janela igual ao span.

Como uma estatística exponencialmente ponderada atribui mais peso a observações mais recentes, ela se "adapta" mais rapidamente às mudanças, se comparada à versão igualmente ponderada.

O pandas tem um operador `ewm` para ser usado em conjunto com `rolling` e `expanding`. Eis um exemplo que compara uma média móvel de 60 dias do preço da ação da Apple com uma média móvel exponencialmente ponderada usando `span=60` (veja a Figura 11.7):

```
In [249]: aapl_px = close_px.AAPL['2006':'2007']

In [250]: ma60 = aapl_px.rolling(30, min_periods=20).mean()

In [251]: ewma60 = aapl_px.ewm(span=30).mean()

In [252]: ma60.plot(style='k--', label='Simple MA')
Out[252]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa59e5df908>

In [253]: ewma60.plot(style='k-', label='EW MA')
Out[253]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa59e5df908>
```

In [254]: plt.legend()

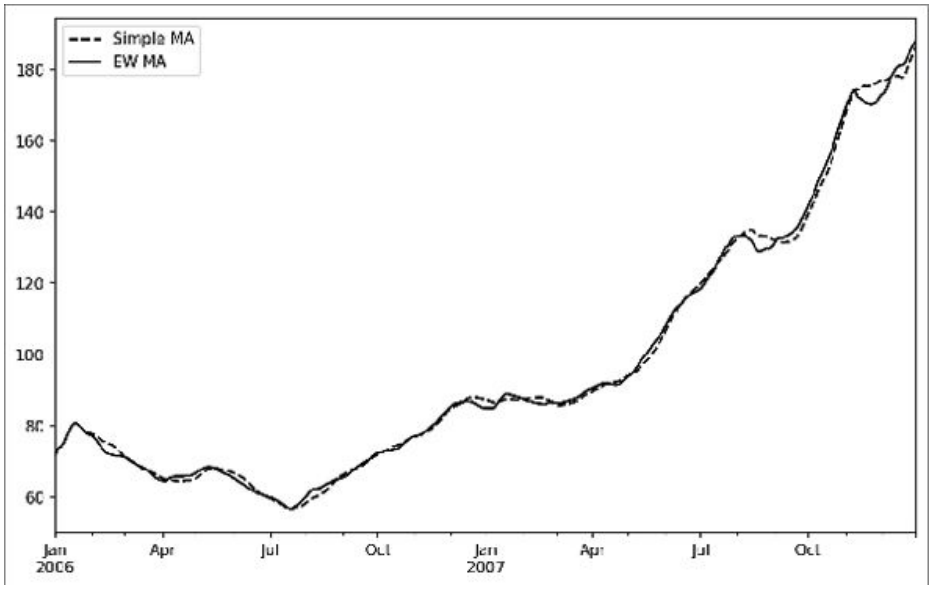


Figura 11.7 – Média móvel simples versus exponencialmente ponderada.

Funções de janela móvel binárias

Alguns operadores estatísticos, como correlação e covariância, precisam atuar em duas séries temporais. Como exemplo, os analistas financeiros muitas vezes estão interessados na correlação entre uma ação e um índice de benchmark, como o S&P 500. Para observar esse cenário, inicialmente calcularemos a mudança percentual para todas as séries temporais de nosso interesse:

In [256]: spx_px = close_px_all['SPX']

In [257]: spx_rets = spx_px.pct_change()

In [258]: returns = close_px.pct_change()

A função de agregação corr, depois que chamamos rolling, pode então calcular a correlação móvel com spx_rets (veja a Figura 11.8 que apresenta a plotagem resultante):

In [259]: corr = returns.AAPL.rolling(125, min_periods=100).corr(spx_rets)

In [260]: corr.plot()

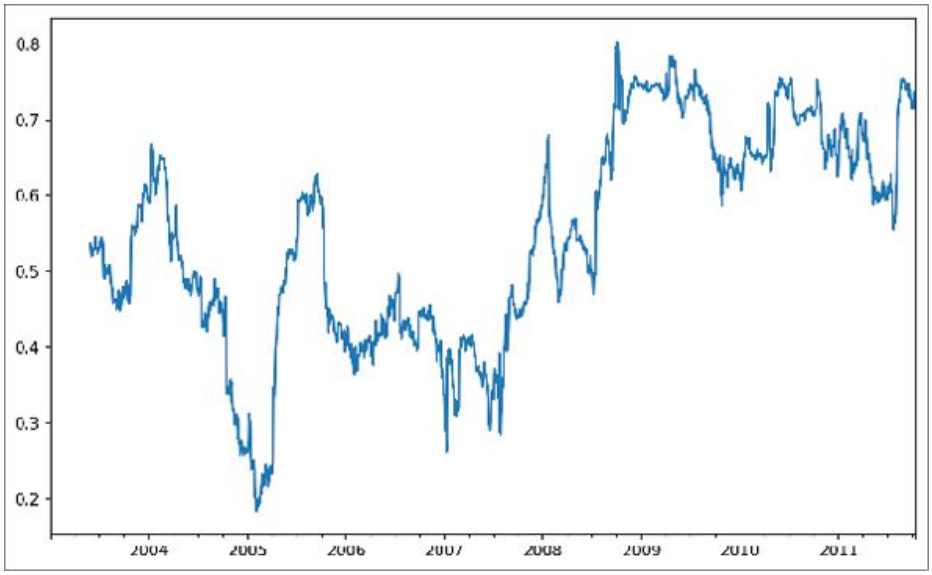


Figura 11.8 – Correlação entre o retorno de AAPL e o S&P 500 em seis meses.

Suponha que quiséssemos calcular a correlação entre o índice S&P 500 com várias ações de uma só vez. Escrever um laço e criar um novo DataFrame seria fácil, mas poderia se tornar repetitivo; desse modo, se você passar uma Series e um DataFrame, uma função como rolling_corr calculará a correlação entre a Series (spx_rets, nesse caso) e cada coluna do DataFrame (veja a Figura 11.9 que apresenta uma plotagem do resultado):

In [262]: corr = returns.rolling(125, min_periods=100).corr(spx_rets)

In [263]: corr.plot()

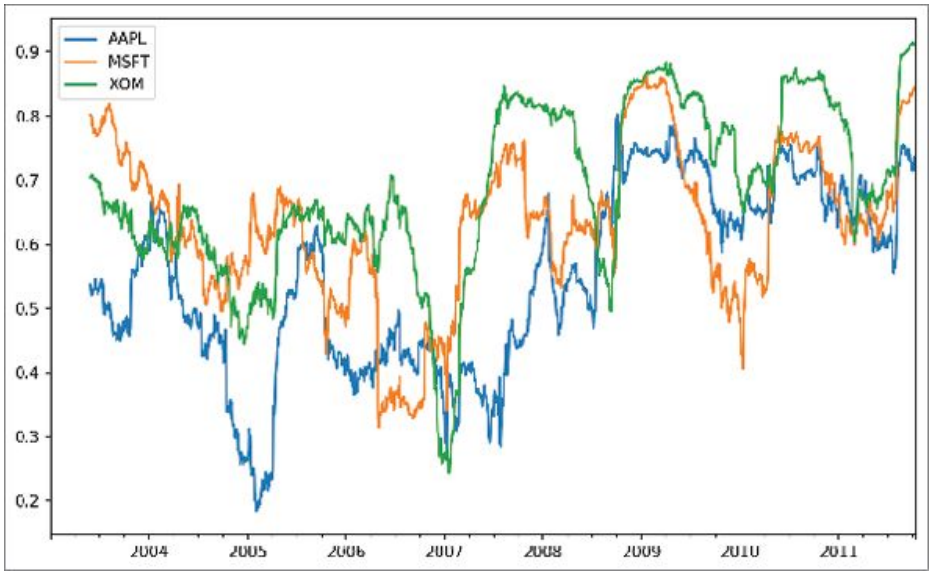


Figura 11.9 – Correlações de retorno com S&P 500 em seis meses.

Funções de janela móvel definidas pelo usuário

O método apply em rolling e métodos relacionados oferecem um meio de aplicar uma função de array criada por você mesmo em uma janela móvel. O único requisito é que a função gere um único valor (uma redução) a partir de cada parte do array. Por exemplo, embora seja possível calcular quantis da amostra usando rolling(...).quantile(q), podemos estar interessados na classificação de percentil de um determinado valor na amostra. A função scipy.stats.percentileofscore faz exatamente isso (veja a Figura 11.10 que apresenta a plotagem resultante):

```
In [265]: from scipy.stats import percentileofscore

In [266]: score_at_2percent = lambda x: percentileofscore(x, 0.02)

In [267]: result = returns.AAPL.rolling(250).apply(score_at_2percent)

In [268]: result.plot()
```

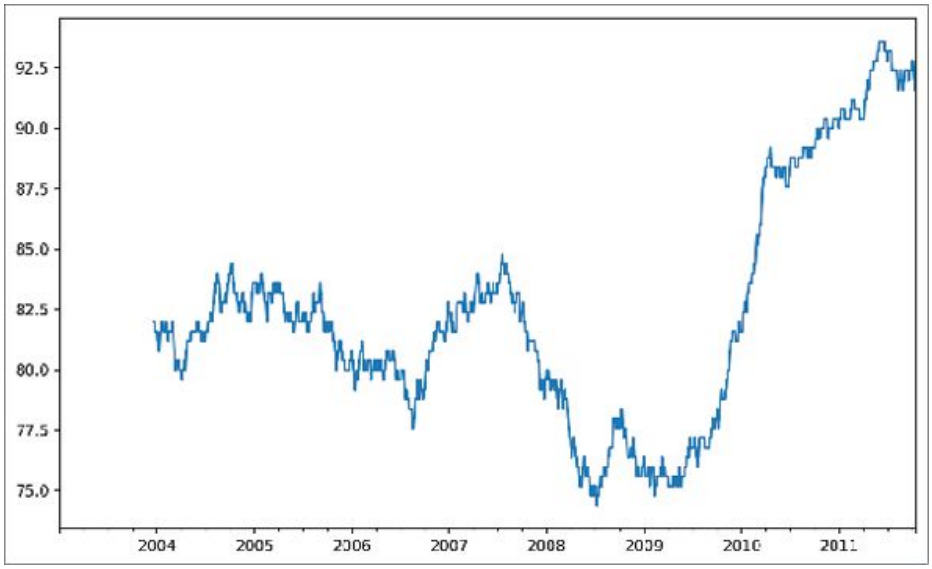


Figura 11.10 – Classificação de percentil de 2% de retorno da AAPL durante uma janela de um ano.

Caso não tenha o SciPy, poderá instalá-lo com o conda ou o pip.

11.8 Conclusão

Dados de séries temporais exigem tipos diferentes de ferramentas de análise e de transformação de dados em comparação com outros tipos de dados que exploramos nos capítulos anteriores.

Nos próximos capítulos, prosseguiremos conhecendo os métodos mais avançados do pandas e mostraremos como começar a usar bibliotecas de modelagem como o statsmodels e o scikit-learn.

---

1 A escolha dos valores default para closed e label pode parecer um pouco inusitada para alguns usuários. Na prática, a escolha é, de certo modo, arbitrária; para algumas frequências visadas, closed='left' é preferível, embora para outras, closed='right' fará mais sentido. O importante é que você tenha em mente exatamente como está segmentando os dados.

CAPÍTULO 12

Pandas avançado

Os capítulos anteriores tiveram como foco a apresentação de diferentes tipos de fluxos de trabalho para tratamento de dados e recursos do NumPy, do pandas e de outras bibliotecas. Com o tempo, o pandas desenvolveu uma variedade de recursos para usuários sofisticados. Este capítulo explora outras áreas de recursos mais avançados que ajudarão a aprofundar o seu expertise como usuário do pandas.

12.1 Dados categorizados

Esta seção apresenta o tipo Categorical do pandas. Mostrarei como é possível alcançar melhor desempenho e uso de memória em algumas operações do pandas utilizando esse tipo. Apresentarei também algumas ferramentas para utilizar dados categorizados em aplicações estatísticas e de aprendizado de máquina (machine learning).

Informações básicas e motivação

Com frequência, uma coluna em uma tabela pode conter instâncias repetidas de um conjunto menor de valores distintos. Já vimos funções como unique e value_counts, que nos permitem extrair os valores distintos de um array e calcular suas frequências, respectivamente:

```
In [10]: import numpy as np; import pandas as pd

In [11]: values = pd.Series(['apple', 'orange', 'apple',
   ....: 'apple'] * 2)
```

```
In [12]: values
Out[12]:
0 apple
1 orange
2 apple
3 apple
4 apple
5 orange
6 apple
7 apple
dtype: object

In [13]: pd.unique(values)
Out[13]: array(['apple', 'orange'], dtype=object)

In [14]: pd.value_counts(values)
Out[14]:
apple 6
orange 2
dtype: int64
```

Muitos sistemas de dados – para armazém de dados (data warehousing), cálculos estatísticos ou outros usos – desenvolveram abordagens especializadas para representar dados com valores repetidos visando a uma armazenagem e a um processamento mais eficazes. Em armazéns de dados, uma boa prática consiste em utilizar as chamadas *tabelas de dimensão* (dimension tables), que contêm os valores distintos, e armazenar os principais dados observados como chaves inteiras que referenciam a tabela de dimensão*:*

```
In [15]: values = pd.Series([0, 1, 0, 0] * 2)

In [16]: dim = pd.Series(['apple', 'orange'])

In [17]: values
Out[17]:
0 0
1 1
```

```
2 0
3 0
4 0
5 1
6 0
7 0
dtype: int64

In [18]: dim
Out[18]:
0 apple
1 orange
dtype: object
```

Podemos usar o método take para restaurar a Series original de strings:

```
In [19]: dim.take(values)
Out[19]:
0 apple
1 orange
0 apple
0 apple
0 apple
1 orange
0 apple
0 apple
dtype: object
```

Essa representação na forma de inteiros chama-se representação *categorizada* (categorical) ou *codificada em dicionários* (dictionary-encoded). O array de valores distintos pode ser chamado de *categorias*, *dicionário* ou *níveis dos dados*. Neste livro, usaremos os termos *categorizado* e *categorias*. Os valores inteiros que referenciam as categorias são chamados de *códigos de categoria* ou simplesmente *códigos*.

A representação em categorias pode resultar em melhorias significativas de desempenho quando analisamos dados. Também podemos realizar transformações nas categorias, ao mesmo tempo que deixamos os códigos inalterados. Alguns exemplos de

transformações que podem ser feitas com um custo relativamente baixo são:

- renomear as categorias;
- concatenar uma nova categoria sem alterar a ordem ou a posição das categorias existentes.

Tipo Categorical do pandas

O pandas tem um tipo especial Categorical para armazenar dados que utilizam a representação ou a *codificação* em categorias baseadas em inteiros. Vamos considerar a Series usada no exemplo anterior:

```
In [20]: fruits = ['apple', 'orange', 'apple', 'apple'] * 2

In [21]: N = len(fruits)

In [22]: df = pd.DataFrame({'fruit': fruits,
   ....: 'basket_id': np.arange(N),
   ....: 'count': np.random.randint(3, 15, size=N),
   ....: 'weight': np.random.uniform(0, 4, size=N)},
   ....: columns=['basket_id', 'fruit', 'count', 'weight'])

In [23]: df
Out[23]:
   basket_id fruit count weight
0 0 apple 5 3.858058
1 1 orange 8 2.612708
2 2 apple 4 2.995627
3 3 apple 7 2.614279
4 4 apple 12 2.990859
5 5 orange 8 3.845227
6 6 apple 5 0.033553
7 7 apple 4 0.425778
```

Nesse caso, df['fruit'] é um array de objetos string de Python. Podemos convertê-lo em categorias chamando:

```
In [24]: fruit_cat = df['fruit'].astype('category')
```

```
In [25]: fruit_cat
Out[25]:
0 apple
1 orange
2 apple
3 apple
4 apple
5 orange
6 apple
7 apple
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

Os valores de fruit_cat não são um array NumPy, mas uma instância de pandas.Categorical:

```
In [26]: c = fruit_cat.values

In [27]: type(c)
Out[27]: pandas.core.categorical.Categorical
```

O objeto Categorical tem os atributos categories e codes:

```
In [28]: c.categories
Out[28]: Index(['apple', 'orange'], dtype='object')

In [29]: c.codes
Out[29]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

Podemos converter a coluna de um DataFrame em dados categorizados atribuindo-lhe o resultado da conversão:

```
In [30]: df['fruit'] = df['fruit'].astype('category')

In [31]: df.fruit
Out[31]:
0 apple
1 orange
2 apple
3 apple
4 apple
5 orange
6 apple
7 apple
```

```
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

Também é possível criar um pandas.Categorical diretamente a partir de outros tipos de sequências Python:

```
In [32]: my_categories = pd.Categorical(['foo', 'bar', 'baz', 'foo', 'bar'])

In [33]: my_categories
Out[33]:
[foo, bar, baz, foo, bar]
Categories (3, object): [bar, baz, foo]
```

Se você tiver obtido dados categorizados codificados de outra fonte, poderá usar o construtor alternativo from_codes:

```
In [34]: categories = ['foo', 'bar', 'baz']

In [35]: codes = [0, 1, 2, 0, 0, 1]

In [36]: my_cats_2 = pd.Categorical.from_codes(codes, categories)

In [37]: my_cats_2
Out[37]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo, bar, baz]
```

A menos que estejam explicitamente especificadas, as conversões de categoria não pressupõem nenhuma ordem específica para as categorias. Assim, o array categories pode estar em uma ordem diferente conforme a ordem dos dados de entrada. Ao usar from_codes ou qualquer um dos demais construtores, podemos informar se as categorias têm uma ordem significativa:

```
In [38]: ordered_cat = pd.Categorical.from_codes(codes, categories,
   ....: ordered=True)

In [39]: ordered_cat
Out[39]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

A saída [foo < bar < baz] indica que 'foo' antecede 'bar' na sequência, e

assim por diante. Uma instância não ordenada de categorias pode ser ordenada com as_ordered:

```
In [40]: my_cats_2.as_ordered()
Out[40]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

Como última observação, os dados categorizados não precisam ser strings, apesar de eu ter mostrado somente exemplos com elas. Um array categorizado pode ser constituído de qualquer tipo de valor imutável.

Processamentos com Categoricals

Usar Categorical no pandas, em comparação com a versão não codificada (como um array de strings), em geral apresenta o mesmo comportamento. Algumas partes do pandas, como a função groupby, têm desempenho melhor quando trabalham com dados categorizados. Há também algumas funções que podem utilizar a flag ordered.

Vamos considerar alguns dados numéricos aleatórios e utilizar a função pandas.qcut para uma separação em compartimentos (binning). Essa função devolve um pandas.Categorical; já havíamos usado pandas.cut antes no livro, mas tínhamos deixado de lado os detalhes de como os dados categorizados funcionam:

```
In [41]: np.random.seed(12345)

In [42]: draws = np.random.randn(1000)

In [43]: draws[:5]
Out[43]: array([-0.2047, 0.4789, -0.5194, -0.5557, 1.9658])
```

Vamos compartimentar esses dados em quartis e extrair algumas estatísticas:

```
In [44]: bins = pd.qcut(draws, 4)

In [45]: bins
```

```
Out[45]:
[(-0.684, -0.0101], (-0.0101, 0.63], (-0.684, -0.0101], (-0.684, -0.0101], (0.63,
 3.928], ..., (-0.0101, 0.63], (-0.684, -0.0101], (-2.95, -0.684], (-0.0101, 0.63
], (0.63, 3.928]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.684] < (-0.684, -0.0101] < (-0.010
1, 0.63] <
                                (0.63, 3.928]]
```

Embora sejam convenientes, os quartis exatos da amostra talvez sejam menos úteis do que seus nomes para gerar um relatório. Podemos fazer isso com o argumento labels em qcut:

```
In [46]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])

In [47]: bins
Out[47]:
[Q2, Q3, Q2, Q2, Q4, ..., Q3, Q2, Q1, Q3, Q4]
Length: 1000
Categories (4, object): [Q1 < Q2 < Q3 < Q4]

In [48]: bins.codes[:10]
Out[48]: array([1, 2, 1, 1, 3, 3, 2, 2, 3, 3], dtype=int8)
```

Os dados categorizados com rótulos em bins não contêm informações sobre as fronteiras dos compartimentos em seus dados; assim, podemos usar groupby para extrair algumas estatísticas de resumo:

```
In [49]: bins = pd.Series(bins, name='quartile')

In [50]: results = (pd.Series(draws)
   ....: .groupby(bins)
   ....: .agg(['count', 'min', 'max'])
   ....: .reset_index())

In [51]: results
Out[51]:
  quartile count min max
0 Q1 250 -2.949343 -0.685484
1 Q2 250 -0.683066 -0.010115
2 Q3 250 -0.010032 0.628894
```

```
3 Q4 250 0.634238 3.927528
```

A coluna 'quartile' no resultado preserva as informações originais de categorias, incluindo a ordem, presentes em bins:

```
In [52]: results['quartile']
Out[52]:
0 Q1
1 Q2
2 Q3
3 Q4
Name: quartile, dtype: category
Categories (4, object): [Q1 < Q2 < Q3 < Q4]
```

Melhor desempenho com dados categorizados

Se você faz muitas análises em um conjunto de dados particular, fazer a conversão para dados categorizados pode resultar, de modo geral, em ganhos substanciais de desempenho. Além do mais, uma versão de uma coluna de DataFrame com dados categorizados com frequência utiliza significativamente menos memória. Vamos considerar uma Series com 10 milhões de elementos e um número reduzido de categorias distintas:

```
In [53]: N = 10000000

In [54]: draws = pd.Series(np.random.randn(N))

In [55]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] * (N // 4))
```

Vamos agora converter labels para dados de categoria:

```
In [56]: categories = labels.astype('category')
```

Percebemos agora que labels utiliza significativamente mais memória que categories:

```
In [57]: labels.memory_usage()
Out[57]: 80000080

In [58]: categories.memory_usage()
Out[58]: 10000272
```

A conversão para categorias obviamente não é gratuita, porém é um

custo ao qual incorremos apenas uma vez:

```
In [59]: %time _ = labels.astype('category')
CPU times: user 360 ms, sys: 140 ms, total: 500 ms
Wall time: 502 ms
```

As operações de groupby podem ser significativamente mais rápidas com dados categorizados, pois os algoritmos subjacentes utilizam o array de códigos baseado em inteiros em vez de usar um array de strings.

Métodos para dados categorizados

As Series contendo dados categorizados têm vários métodos especiais semelhantes aos métodos especializados de string Series.str. Esses métodos também oferecem acesso conveniente às categorias e aos códigos. Considere a Series a seguir:

```
In [60]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)

In [61]: cat_s = s.astype('category')

In [62]: cat_s
Out[62]:
0 a
1 b
2 c
3 d
4 a
5 b
6 c
7 d
dtype: category
Categories (4, object): [a, b, c, d]
```

O atributo especial cat oferece acesso aos métodos de categoria:

```
In [63]: cat_s.cat.codes
Out[63]:
0 0
1 1
2 2
```

```
3 3
4 0
5 1
6 2
7 3
dtype: int8

In [64]: cat_s.cat.categories
Out[64]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

Suponha que soubéssemos que o verdadeiro conjunto de categorias para esses dados se estendesse para além dos quatro valores observados nos dados. Podemos utilizar o método set_categories para alterá-lo:

```
In [65]: actual_categories = ['a', 'b', 'c', 'd', 'e']

In [66]: cat_s2 = cat_s.cat.set_categories(actual_categories)

In [67]: cat_s2
Out[67]:
0 a
1 b
2 c
3 d
4 a
5 b
6 c
7 d
dtype: category
Categories (5, object): [a, b, c, d, e]
```

Embora pareça que os dados não tenham mudado, as novas categorias se refletirão nas operações que as usam. Por exemplo, value_counts respeita as categorias, se elas estiverem presentes:

```
In [68]: cat_s.value_counts()
Out[68]:
d 2
c 2
b 2
a 2
```

```
dtype: int64

In [69]: cat_s2.value_counts()
Out[69]:
d 2
c 2
b 2
a 2
e 0
dtype: int64
```

Em conjuntos de dados grandes, dados categorizados em geral são usados como uma ferramenta conveniente para economizar memória e ter melhor desempenho. Depois de filtrar um DataFrame ou uma Series grande, muitas das categorias talvez não apareçam nos dados. Para ajudar nesse caso, podemos utilizar o método remove_unused_categories e remover as categorias que não sejam observadas:

```
In [70]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]

In [71]: cat_s3
Out[71]:
0 a
1 b
4 a
5 b
dtype: category
Categories (4, object): [a, b, c, d]

In [72]: cat_s3.cat.remove_unused_categories()
Out[72]:
0 a
1 b
4 a
5 b
dtype: category
Categories (2, object): [a, b]
```

Veja a Tabela 12.1 que contém uma lista dos métodos disponíveis para categorias.

Tabela 12.1 – Métodos de categoria para Series no pandas

Método	Descrição
add_categories	Concatena novas categorias (não usadas) no final das categorias existentes
as_ordered	Deixa as categorias ordenadas
as_unordered	Deixa as categorias não ordenadas
remove_categories	Remove categorias, definindo qualquer valor removido com nulo
remove_unused_categories	Remove qualquer valor de categoria que não apareça nos dados
rename_categories	Substitui as categorias pelo conjunto indicado de novos nomes de categorias; não pode alterar o número de categorias
reorder_categories	Comporta-se como rename_categories, mas também pode alterar o resultado para ter categorias ordenadas
set_categories	Substitui as categorias pelo conjunto indicado de novas categorias; pode adicionar ou remover categorias

Criando variáveis dummy para modelagem

Quando usamos ferramentas estatísticas ou de aprendizado de máquina, com frequência transformaremos dados categorizados em *variáveis dummy*, também conhecidas como codificação *one-hot*. Essa operação envolve a criação de um DataFrame com uma coluna para cada categoria distinta; essas colunas contêm 1 para as ocorrências de uma dada categoria e 0 caso contrário.

Considere o exemplo anterior:

```
In [73]: cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2, dtype='category')
```

Conforme mencionamos antes no Capítulo 7, a função pandas.get_dummies converte esses dados categorizados unidimensionais em um DataFrame contendo a variável dummy:

```
In [74]: pd.get_dummies(cat_s)
Out[74]:
   a b c d
```

```
0 1 0 0 0
1 0 1 0 0
2 0 0 1 0
3 0 0 0 1
4 1 0 0 0
5 0 1 0 0
6 0 0 1 0
7 0 0 0 1
```

12.2 Uso avançado de GroupBy

Embora já tenhamos discutido o uso do método groupby em Series e DataFrame de modo detalhado no Capítulo 10, há algumas técnicas adicionais que talvez você ache úteis.

Transformações de grupos e GroupBys "não encapsulados"

No Capítulo 10, vimos o método apply em operações de grupo, usado para realizar transformações. Há outro método embutido chamado transform, que é semelhante a apply, porém impõe mais restrições quanto ao tipo de função que você pode usar:

- pode gerar um valor escalar com o qual um broadcast será feito para a dimensão do grupo;
- pode gerar um objeto com o mesmo formato que o grupo de entrada;
- não deve modificar a sua entrada.

Vamos considerar um exemplo simples para demonstração:

```
In [75]: df = pd.DataFrame({'key': ['a', 'b', 'c'] * 4,
   ....: 'value': np.arange(12.)})

In [76]: df
Out[76]:
   key value
0 a 0.0
1 b 1.0
```

2 c 2.0
3 a 3.0
4 b 4.0
5 c 5.0
6 a 6.0
7 b 7.0
8 c 8.0
9 a 9.0
10 b 10.0
11 c 11.0

Eis as médias de grupo por chave:

```
In [77]: g = df.groupby('key').value

In [78]: g.mean()
Out[78]:
key
a 4.5
b 5.5
c 6.5
Name: value, dtype: float64
```

Suponha que, em vez disso, quiséssemos gerar uma Series de mesmo formato que `df['value']`, mas com valores substituídos pela média agrupada por `'key'`. Podemos passar a função `lambda x: x.mean()` para `transform`:

```
In [79]: g.transform(lambda x: x.mean())
Out[79]:
0 4.5
1 5.5
2 6.5
3 4.5
4 5.5
5 6.5
6 4.5
7 5.5
8 6.5
9 4.5
10 5.5
11 6.5
Name: value, dtype: float64
```

Para as funções de agregação embutidas, podemos passar um alias na forma de string, como no método agg de GroupBy:

```
In [80]: g.transform('mean')
Out[80]:
0 4.5
1 5.5
2 6.5
3 4.5
4 5.5
5 6.5
6 4.5
7 5.5
8 6.5
9 4.5
10 5.5
11 6.5
Name: value, dtype: float64
```

Assim como apply, transform trabalha com funções que devolvem Series, mas o resultado deve ter o mesmo tamanho que a entrada. Por exemplo, podemos multiplicar cada grupo por 2 usando uma função lambda:

```
In [81]: g.transform(lambda x: x * 2)
Out[81]:
0 0.0
1 2.0
2 4.0
3 6.0
4 8.0
5 10.0
6 12.0
7 14.0
8 16.0
9 18.0
10 20.0
11 22.0
Name: value, dtype: float64
```

Como um exemplo mais complicado, podemos calcular as classificações em ordem decrescente para cada grupo:

```
In [82]: g.transform(lambda x: x.rank(ascending=False))
Out[82]:
0 4.0
1 4.0
2 4.0
3 3.0
4 3.0
5 3.0
6 2.0
7 2.0
8 2.0
9 1.0
10 1.0
11 1.0
Name: value, dtype: float64
```

Considere uma função de transformação de grupo composta de agregações simples:

```
def normalize(x):
    return (x - x.mean()) / x.std()
```

Podemos obter resultados equivalentes nesse caso, seja usando transform, seja com apply:

```
In [84]: g.transform(normalize)
Out[84]:
0 -1.161895
1 -1.161895
2 -1.161895
3 -0.387298
4 -0.387298
5 -0.387298
6 0.387298
7 0.387298
8 0.387298
9 1.161895
10 1.161895
11 1.161895
Name: value, dtype: float64

In [85]: g.apply(normalize)
```

```
Out[85]:
0 -1.161895
1 -1.161895
2 -1.161895
3 -0.387298
4 -0.387298
5 -0.387298
6 0.387298
7 0.387298
8 0.387298
9 1.161895
10 1.161895
11 1.161895
Name: value, dtype: float64
```

Funções de agregação embutidas como 'mean' ou 'sum' em geral são mais rápidas que uma função apply genérica. Elas também têm um "passo rápido" quando usadas com transform, o que nos permite executar a chamada operação de grupo *não encapsulada*:

```
In [86]: g.transform('mean')
Out[86]:
0 4.5
1 5.5
2 6.5
3 4.5
4 5.5
5 6.5
6 4.5
7 5.5
8 6.5
9 4.5
10 5.5
11 6.5
Name: value, dtype: float64

In [87]: normalized = (df['value'] - g.transform('mean')) / g.transform('std')

In [88]: normalized
Out[88]:
0 -1.161895
```

```
1 -1.161895
2 -1.161895
3 -0.387298
4 -0.387298
5 -0.387298
6 0.387298
7 0.387298
8 0.387298
9 1.161895
10 1.161895
11 1.161895
Name: value, dtype: float64
```

Embora uma operação de grupo não encapsulada possa envolver várias agregações de grupo, a vantagem das operações vetorizadas em geral muitas vezes compensará.

Reamostragem de tempo em grupos

Para dados de séries temporais, o método resample é semanticamente uma operação de grupo baseada em intervalos de tempo. Eis uma pequena tabela como exemplo:

```
In [89]: N = 15

In [90]: times = pd.date_range('2017-05-20 00:00', freq='1min', periods=N)

In [91]: df = pd.DataFrame({'time': times,
   ....: 'value': np.arange(N)})

In [92]: df
Out[92]:
                time value
0 2017-05-20 00:00:00 0
1 2017-05-20 00:01:00 1
2 2017-05-20 00:02:00 2
3 2017-05-20 00:03:00 3
4 2017-05-20 00:04:00 4
5 2017-05-20 00:05:00 5
6 2017-05-20 00:06:00 6
7 2017-05-20 00:07:00 7
```

```
8 2017-05-20 00:08:00 8
9 2017-05-20 00:09:00 9
10 2017-05-20 00:10:00 10
11 2017-05-20 00:11:00 11
12 2017-05-20 00:12:00 12
13 2017-05-20 00:13:00 13
14 2017-05-20 00:14:00 14
```

Nesse caso, podemos indexar por 'time' e então fazer uma reamostragem:

```
In [93]: df.set_index('time').resample('5min').count()
Out[93]:
                     value
time
2017-05-20 00:00:00 5
2017-05-20 00:05:00 5
2017-05-20 00:10:00 5
```

Suponha que um DataFrame contenha várias séries temporais, marcadas por uma coluna adicional de chave de grupo:

```
In [94]: df2 = pd.DataFrame({'time': times.repeat(3),
   ....: 'key': np.tile(['a', 'b', 'c'], N),
   ....: 'value': np.arange(N * 3.)})

In [95]: df2[:7]
Out[95]:
  key time value
0 a 2017-05-20 00:00:00 0.0
1 b 2017-05-20 00:00:00 1.0
2 c 2017-05-20 00:00:00 2.0
3 a 2017-05-20 00:01:00 3.0
4 b 2017-05-20 00:01:00 4.0
5 c 2017-05-20 00:01:00 5.0
6 a 2017-05-20 00:02:00 6.0
```

Para fazer a mesma reamostragem para cada valor de 'key', apresentaremos o objeto pandas.TimeGrouper:

```
In [96]: time_key = pd.TimeGrouper('5min')
```

Podemos então definir o índice de tempo, agrupar por 'key' e time_key e agregar:

```
In [97]: resampled = (df2.set_index('time')
   ....: .groupby(['key', time_key])
   ....: .sum())

In [98]: resampled
Out[98]:
                     value
key time
a 2017-05-20 00:00:00 30.0
    2017-05-20 00:05:00 105.0
    2017-05-20 00:10:00 180.0
b 2017-05-20 00:00:00 35.0
    2017-05-20 00:05:00 110.0
    2017-05-20 00:10:00 185.0
c 2017-05-20 00:00:00 40.0
    2017-05-20 00:05:00 115.0
    2017-05-20 00:10:00 190.0

In [99]: resampled.reset_index()
Out[99]:
  key time value
0 a 2017-05-20 00:00:00 30.0
1 a 2017-05-20 00:05:00 105.0
2 a 2017-05-20 00:10:00 180.0
3 b 2017-05-20 00:00:00 35.0
4 b 2017-05-20 00:05:00 110.0
5 b 2017-05-20 00:10:00 185.0
6 c 2017-05-20 00:00:00 40.0
7 c 2017-05-20 00:05:00 115.0
8 c 2017-05-20 00:10:00 190.0
```

Uma limitação ao usar TimeGrouper está no fato de o tempo ter que ser o índice da Series ou do DataFrame.

12.3 Técnicas para encadeamento de métodos

Ao aplicar uma sequência de transformações em um conjunto de dados, é possível que você se veja criando diversas variáveis temporárias que jamais serão usadas em sua análise. Considere o exemplo a seguir:

```
df = load_data()
df2 = df[df['col2'] < 0]
df2['col1_demeaned'] = df2['col1'] - df2['col1'].mean()
result = df2.groupby('key').col1_demeaned.std()
```

Embora não estejamos usando nenhum dado real nesse caso, o exemplo destaca alguns métodos novos. Inicialmente, o método DataFrame.assign é uma alternativa *funcional* para atribuições de colunas no formato df[k] = v. Em vez de modificar o objeto in-place, ele devolve um novo DataFrame com as modificações indicadas. Assim, as instruções a seguir são equivalentes:

```
# Modo usual, não funcional
df2 = df.copy()
df2['k'] = v

# Modo de atribuição funcional
df2 = df.assign(k=v)
```

Fazer uma atribuição in-place pode executar mais rapidamente que usar assign, porém assign permite encadear métodos com mais facilidade:

```
result = (df2.assign(col1_demeaned=df2.col1 - df2.col2.mean())
          .groupby('key')
          .col1_demeaned.std())
```

Utilizei os parênteses externos para que a adição de quebras de linha fosse mais conveniente.

Um aspecto para se ter em mente ao fazer encadeamento de métodos é que talvez você precise referenciar objetos temporários. No exemplo anterior, não podemos referenciar o resultado de load_data até que ele tenha sido atribuído à variável temporária df. Para ajudar nesse caso, assign e muitas outras funções do pandas aceitam argumentos do tipo função, também conhecidos como *callables*.

Para mostrar as callables em ação, considere um fragmento do exemplo anterior:

```
df = load_data()
df2 = df[df['col2'] < 0]
```

Esse código pode ser reescrito assim:

```
df = (load_data()
    [lambda x: x['col2'] < 0])
```

Nesse caso, o resultado de load_data não é atribuído a uma variável, de modo que a função passada em [] é então *vinculada* ao objeto nessa etapa do encadeamento de métodos.

Podemos prosseguir, então, e escrever toda a sequência como uma única expressão encadeada:

```
result = (load_data()
        [lambda x: x.col2 < 0]
        .assign(col1_demeaned=lambda x: x.col1 - x.col1.mean())
        .groupby('key')
        .col1_demeaned.std())
```

O fato de você preferir escrever seu código nesse estilo é uma questão de gosto pessoal, e separar a expressão em vários passos pode deixar seu código mais legível.

Método pipe

Podemos executar muitas tarefas com as funções embutidas do pandas e as abordagens para encadeamento de métodos com callables que acabamos de ver. Às vezes, porém, você terá que usar suas próprias funções ou funções de bibliotecas de terceiros. É em situações como essa que o método pipe entra em cena.

Considere uma sequência de chamadas de função:

```
a = f(df, arg1=v1)
b = g(a, v2, arg3=v3)
c = h(b, arg4=v4)
```

Ao usar funções que aceitem e devolvam objetos Series ou DataFrame, podemos reescrever esse código usando chamadas a pipe:

```
result = (df.pipe(f, arg1=v1)
        .pipe(g, v2, arg3=v3)
        .pipe(h, arg4=v4))
```

As instruções f(df) e df.pipe(f) são equivalentes, porém pipe facilita a chamada de métodos encadeados.

Um padrão possivelmente útil para pipe está em generalizar sequências de operações em funções reutilizáveis. Como exemplo, vamos considerar a subtração das médias dos grupos de uma coluna:

```
g = df.groupby(['key1', 'key2'])
df['col1'] = df['col1'] - g.transform('mean')
```

Suponha que quiséssemos subtrair a média de mais de uma coluna e alterar facilmente as chaves de grupo. Além do mais, talvez você quisesse fazer essa transformação em uma cadeia de métodos. Eis um exemplo de implementação:

```
def group_demean(df, by, cols):
    result = df.copy()
    g = df.groupby(by)
    for c in cols:
        result[c] = df[c] - g[c].transform('mean')
    return result
```

Então é possível escrever o seguinte:

```
result = (df[df.col1 < 0]
          .pipe(group_demean, ['key1', 'key2'], ['col1']))
```

12.4 Conclusão

O pandas, assim como muitos projetos de software de código aberto, continua mudando e adquirindo funcionalidades novas e melhores. Como em todos os outros pontos neste livro, o foco deste capítulo está nas funcionalidades mais estáveis, cuja probabilidade de mudar nos próximos anos é mais baixa.

Para aprofundar seu expertise como usuário do pandas, incentivo você a explorar a documentação (*http://pandas.pydata.org/*) e ler as notas de lançamento de versão (release notes) à medida que a equipe de desenvolvimento lançar novas versões do código aberto. Também convidamos você a se juntar ao desenvolvimento do

pandas: corrigindo bugs, desenvolvendo novos recursos e melhorando a documentação.

CAPÍTULO 13

Introdução às bibliotecas de modelagem em Python

Neste livro, meu foco é oferecer uma base de programação para fazer análises de dados em Python. Como os analistas e os cientistas de dados com frequência relatam gastar um período de tempo desproporcional com o tratamento e a preparação de dados, a estrutura do livro reflete a importância de dominar essas técnicas.

A biblioteca que você usará para desenvolver modelos dependerá da aplicação. Muitos problemas estatísticos podem ser solucionados por meio de técnicas mais simples, como regressão de mínimos quadrados ordinários, enquanto outros talvez exijam métodos mais sofisticados de aprendizado de máquina. Felizmente, Python se tornou uma das linguagens preferidas para implementar métodos de análises, portanto há muitas ferramentas que você pode explorar depois que terminar a leitura deste livro.

Neste capítulo, analisarei alguns recursos do pandas que poderão ser úteis quando você estiver indo e voltando com o tratamento de dados (data wrangling) no pandas, com a adequação a modelos e a pontuação. Então farei uma introdução rápida de dois kits de ferramentas populares para modelagem: o statsmodels (*http://statsmodels.org/*) e o scikit-learn (*http://scikit-learn.org*). Considerando que cada um desses projetos é suficientemente grande para merecer o seu próprio livro, não dedicarei esforços para ser abrangente; em vez disso, orientarei você a consultar a documentação online dos dois projetos, junto com outros livros sobre ciência de dados, estatísticas e aprendizado de máquina,

baseados em Python.

13.1 Interface entre o pandas e o código dos modelos

Um fluxo de trabalho comum para desenvolvimento de modelos é usar o pandas para carga e limpeza de dados antes de passar para uma biblioteca de modelagem e construir o modelo propriamente dito. Uma parte importante do processo de desenvolvimento de modelos é chamada de *engenharia de características* (feature engineering) em aprendizado de máquina. Ela pode descrever qualquer transformação ou análise de dados que extraia informações de um conjunto de dados brutos, as quais sejam úteis em um contexto de modelagem. As ferramentas de agregação de dados e o GroupBy que exploramos neste livro são usados com frequência em um contexto de engenharia de características.

Embora detalhes sobre uma “boa” engenharia de características estejam fora do escopo deste livro, apresentarei alguns métodos para fazer com que a passagem da manipulação de dados com o pandas para a modelagem seja a mais tranquila possível.

O ponto de contato entre o pandas e outras bibliotecas de análise em geral são os arrays NumPy. Para transformar um DataFrame em um array NumPy, utilize a propriedade .values:

```
In [10]: import pandas as pd

In [11]: import numpy as np

In [12]: data = pd.DataFrame({
   ....: 'x0': [1, 2, 3, 4, 5],
   ....: 'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....: 'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [13]: data
Out[13]:
   x0 x1 y
0 1 0.01 -1.5
```

```
1 2 -0.01 0.0
2 3 0.25 3.6
3 4 -4.10 1.3
4 5 0.00 -2.0

In [14]: data.columns
Out[14]: Index(['x0', 'x1', 'y'], dtype='object')

In [15]: data.values
Out[15]:
array([[ 1. , 0.01, -1.5 ],
       [ 2. , -0.01, 0. ],
       [ 3. , 0.25, 3.6 ],
       [ 4. , -4.1 , 1.3 ],
       [ 5. , 0. , -2. ]])
```

Para converter de volta para um DataFrame, conforme você deve recordar de capítulos anteriores, podemos passar um ndarray bidimensional com nomes opcionais para as colunas:

```
In [16]: df2 = pd.DataFrame(data.values, columns=['one', 'two', 'three'])

In [17]: df2
Out[17]:
   one two three
0 1.0 0.01 -1.5
1 2.0 -0.01 0.0
2 3.0 0.25 3.6
3 4.0 -4.10 1.3
4 5.0 0.00 -2.0
```

 O atributo .values tem como finalidade ser usado quando seus dados são homogêneos – por exemplo, todos são do tipo numérico. Se você tiver dados heterogêneos, o resultado será um ndarray de objetos Python:

```
In [18]: df3 = data.copy()

In [19]: df3['strings'] = ['a', 'b', 'c', 'd', 'e']

In [20]: df3
Out[20]:
```

```
   x0 x1 y strings
0 1 0.01 -1.5 a
1 2 -0.01 0.0 b
2 3 0.25 3.6 c
3 4 -4.10 1.3 d
4 5 0.00 -2.0 e
```

```
In [21]: df3.values
Out[21]:
array([[1, 0.01, -1.5, 'a'],
       [2, -0.01, 0.0, 'b'],
       [3, 0.25, 3.6, 'c'],
       [4, -4.1, 1.3, 'd'],
       [5, 0.0, -2.0, 'e']], dtype=object)
```

Para alguns modelos, talvez você queira usar apenas um subconjunto das colunas. Recomendo utilizar a indexação `loc` com `values`:

```
In [22]: model_cols = ['x0', 'x1']

In [23]: data.loc[:, model_cols].values
Out[23]:
array([[ 1. , 0.01],
       [ 2. , -0.01],
       [ 3. , 0.25],
       [ 4. , -4.1 ],
       [ 5. , 0. ]])
```

Algumas bibliotecas têm suporte nativo para o pandas e fazem parte dessa tarefa para você automaticamente: realizando a conversão para NumPy a partir de um DataFrame e associando nomes de parâmetros de modelos às colunas das tabelas ou Series de saída. Em outros casos, você terá que executar esse "gerenciamento de metadados" manualmente.

No Capítulo 12, vimos o tipo `Categorical` do pandas e a função `pandas.get_dummies`. Suponha que tivéssemos uma coluna não numérica em nosso conjunto de dados de exemplo:

```
In [24]: data['category'] = pd.Categorical(['a', 'b', 'a', 'a', 'b'],
   ....: categories=['a', 'b'])
```

```
In [25]: data
Out[25]:
   x0 x1 y category
0 1 0.01 -1.5 a
1 2 -0.01 0.0 b
2 3 0.25 3.6 a
3 4 -4.10 1.3 a
4 5 0.00 -2.0 b
```

Se quiséssemos substituir a coluna 'category' por variáveis dummy, criaríamos essas variáveis, descartaríamos a coluna 'category' e então faríamos uma junção (join) do resultado:

```
In [26]: dummies = pd.get_dummies(data.category, prefix='category')

In [27]: data_with_dummies = data.drop('category', axis=1).join(dummies)

In [28]: data_with_dummies
Out[28]:
   x0 x1 y category_a category_b
0 1 0.01 -1.5 1 0
1 2 -0.01 0.0 0 1
2 3 0.25 3.6 1 0
3 4 -4.10 1.3 1 0
4 5 0.00 -2.0 0 1
```

Há algumas nuances para a adequação de certos modelos estatísticos com variáveis dummy. Talvez seja mais simples e menos suscetível a erros usar o Patsy (assunto da próxima seção) quando você tiver mais que simples colunas numéricas.

13.2 Criando descrições de modelos com o Patsy

O Patsy (*https://patsy.readthedocs.io*) é uma biblioteca Python para descrever modelos estatísticos (especialmente modelos lineares) com uma “sintaxe de fórmula” baseada em pequenas strings, inspirada (mas não exatamente igual) na sintaxe de fórmulas usada pelas linguagens de programação estatística R e S.

O Patsy tem bom suporte para especificar modelos lineares no

statsmodels, portanto manterei o foco em algumas das principais funcionalidades a fim de ajudar você a estar pronto para usá-lo. As *fórmulas* do Patsy têm uma sintaxe especial de string com o seguinte aspecto:

```
y ~ x0 + x1
```

A sintaxe `a + b` não significa somar `a` com `b`, mas sim que são *termos* da *matriz de design* criada para o modelo. A função `patsy.dmatrices` aceita uma string com uma fórmula, junto com um conjunto de dados (que pode ser um DataFrame ou um dicionário de arrays), e gera matrizes de design para um modelo linear:

```
In [29]: data = pd.DataFrame({
   ....: 'x0': [1, 2, 3, 4, 5],
   ....: 'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....: 'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [30]: data
Out[30]:
   x0 x1 y
0 1 0.01 -1.5
1 2 -0.01 0.0
2 3 0.25 3.6
3 4 -4.10 1.3
4 5 0.00 -2.0

In [31]: import patsy

In [32]: y, X = patsy.dmatrices('y ~ x0 + x1', data)
```

Agora temos:

```
In [33]: y
Out[33]:
DesignMatrix with shape (5, 1)
     y
  -1.5
   0.0
   3.6
   1.3
  -2.0
```

```
  Terms:
    'y' (column 0)

In [34]: X
Out[34]:
DesignMatrix with shape (5, 3)
  Intercept x0 x1
          1 1 0.01
          1 2 -0.01
          1 3 0.25
          1 4 -4.10
          1 5 0.00
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'x1' (column 2)
```

Essas instâncias de DesignMatrix do Patsy são ndarrays NumPy com metadados adicionais:

```
In [35]: np.asarray(y)
Out[35]:
array([[-1.5],
       [ 0. ],
       [ 3.6],
       [ 1.3],
       [-2. ]])

In [36]: np.asarray(X)
Out[36]:
array([[ 1. , 1. , 0.01],
       [ 1. , 2. , -0.01],
       [ 1. , 3. , 0.25],
       [ 1. , 4. , -4.1 ],
       [ 1. , 5. , 0. ]])
```

Você deve estar se perguntando de onde veio o termo Intercept. Essa é uma convenção para modelos lineares, como o de regressão de OLS (Ordinary Least Squares, Mínimos Quadrados Ordinários). Podemos suprimir a interceptação adicionando o termo `+ 0` ao modelo:

```
In [37]: patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]
Out[37]:
DesignMatrix with shape (5, 2)
  x0 x1
   1 0.01
   2 -0.01
   3 0.25
   4 -4.10
   5 0.00
  Terms:
    'x0' (column 0)
    'x1' (column 1)
```

Os objetos do Patsy podem ser passados diretamente para algoritmos como numpy.linalg.lstsq, que executa uma regressão de mínimos quadrados ordinários:

```
In [38]: coef, resid, _, _ = np.linalg.lstsq(X, y)
```

Os metadados do modelo são mantidos no atributo design_info, portanto você pode associar novamente os nomes das colunas do modelo aos coeficientes apropriados a fim de obter uma Series, por exemplo:

```
In [39]: coef
Out[39]:
array([[ 0.3129],
       [-0.0791],
       [-0.2655]])

In [40]: coef = pd.Series(coef.squeeze(), index=X.design_info.column_names)

In [41]: coef
Out[41]:
Intercept 0.312910
x0 -0.079106
x1 -0.265464
dtype: float64
```

Transformações de dados em fórmulas do Patsy

Podemos misturar código Python nas fórmulas do Patsy; ao avaliar

a fórmula, a biblioteca tentará encontrar as funções usadas no escopo que as incluem:

```
In [42]: y, X = patsy.dmatrices('y ~ x0 + np.log(np.abs(x1) + 1)', data)

In [43]: X
Out[43]:
DesignMatrix with shape (5, 3)
  Intercept x0 np.log(np.abs(x1) + 1)
          1 1 0.00995
          1 2 0.00995
          1 3 0.22314
          1 4 1.62924
          1 5 0.00000
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'np.log(np.abs(x1) + 1)' (column 2)
```

Algumas transformações de variáveis comumente utilizadas incluem padronização (para média 0 e variância 1) e centralização (subtração da média). O Patsy tem funções embutidas para isso:

```
In [44]: y, X = patsy.dmatrices('y ~ standardize(x0) + center(x1)', data)

In [45]: X
Out[45]:
DesignMatrix with shape (5, 3)
  Intercept standardize(x0) center(x1)
          1 -1.41421 0.78
          1 -0.70711 0.76
          1 0.00000 1.02
          1 0.70711 -3.33
          1 1.41421 0.77
  Terms:
    'Intercept' (column 0)
    'standardize(x0)' (column 1)
    'center(x1)' (column 2)
```

Como parte de um processo de modelagem, você pode fazer a adequação de um modelo a um conjunto de dados e então avaliar o modelo em outro conjunto. Esse pode ser uma porção *reservada* dos

dados ou novos dados observados depois. Ao aplicar transformações como centralização e padronização, tome cuidado ao usar o modelo para fazer previsões baseadas em novos dados. Essas transformações são chamadas de *stateful* (com estado) porque você deve usar estatísticas como média ou desvio-padrão do conjunto de dados original quando fizer transformações em um novo conjunto de dados.

A função patsy.build_design_matrices é capaz de aplicar transformações em novos dados *fora da amostra* (out-of-sample) usando as informações salvas do conjunto de dados original *da amostra* (in-sample):

```
In [46]: new_data = pd.DataFrame({
   ....: 'x0': [6, 7, 8, 9],
   ....: 'x1': [3.1, -0.5, 0, 2.3],
   ....: 'y': [1, 2, 3, 4]})

In [47]: new_X = patsy.build_design_matrices([X.design_info], new_data)

In [48]: new_X
Out[48]:
[DesignMatrix with shape (4, 3)
  Intercept standardize(x0) center(x1)
          1 2.12132 3.87
          1 2.82843 0.27
          1 3.53553 0.77
          1 4.24264 3.07
  Terms:
    'Intercept' (column 0)
    'standardize(x0)' (column 1)
    'center(x1)' (column 2)]
```

Como o símbolo de adição (+) no contexto das fórmulas do Patsy não significa soma, se quisermos somar colunas de um conjunto de dados pelo nome, devemos encapsulá-las na função especial *I*:

```
In [49]: y, X = patsy.dmatrices('y ~ I(x0 + x1)', data)

In [50]: X
Out[50]:
```

```
DesignMatrix with shape (5, 2)
  Intercept I(x0 + x1)
          1 1.01
          1 1.99
          1 3.25
          1 -0.10
          1 5.00
  Terms:
    'Intercept' (column 0)
    'I(x0 + x1)' (column 1)
```

O Patsy tem várias outras transformações embutidas no módulo patsy.builtins. Consulte a documentação online para obter mais informações.

Os dados categorizados apresentam uma classe especial de transformações, que explicarei a seguir.

Dados categorizados e o Patsy

Dados não numéricos podem ser transformados em uma matriz de design de modelo de várias maneiras diferentes. Uma abordagem completa sobre esse assunto está fora do escopo deste livro e seria mais apropriada para um curso de estatística.

Quando usamos termos não numéricos em uma fórmula do Patsy, eles são, por padrão, convertidos para variáveis dummy. Se houver uma interceptação, um dos níveis será deixado de fora a fim de evitar colinearidade:

```
In [51]: data = pd.DataFrame({
   ....: 'key1': ['a', 'a', 'b', 'b', 'a', 'b', 'a', 'b'],
   ....: 'key2': [0, 1, 0, 1, 0, 1, 0, 0],
   ....: 'v1': [1, 2, 3, 4, 5, 6, 7, 8],
   ....: 'v2': [-1, 0, 2.5, -0.5, 4.0, -1.2, 0.2, -1.7]
   ....: })

In [52]: y, X = patsy.dmatrices('v2 ~ key1', data)

In [53]: X
Out[53]:
```

```
DesignMatrix with shape (8, 2)
  Intercept key1[T.b]
          1 0
          1 0
          1 1
          1 1
          1 0
          1 1
          1 0
          1 1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
```

Se você omitir a interceptação do modelo, colunas para cada valor de categoria serão incluídas na matriz de design do modelo:

```
In [54]: y, X = patsy.dmatrices('v2 ~ key1 + 0', data)

In [55]: X
Out[55]:
DesignMatrix with shape (8, 2)
  key1[a] key1[b]
        1 0
        1 0
        0 1
        0 1
        1 0
        0 1
        1 0
        0 1
  Terms:
    'key1' (columns 0:2)
```

Colunas numéricas podem ser interpretadas como categorizadas com a função `C`:

```
In [56]: y, X = patsy.dmatrices('v2 ~ C(key2)', data)

In [57]: X
Out[57]:
DesignMatrix with shape (8, 2)
  Intercept C(key2)[T.1]
```

```
        1 0
        1 1
        1 0
        1 1
        1 0
        1 1
        1 0
        1 0
  Terms:
    'Intercept' (column 0)
    'C(key2)' (column 1)
```

Se vários termos categorizados forem usados em um modelo, a situação talvez se torne mais complicada, pois você poderá incluir termos de interação no formato key1:key2, os quais poderão ser utilizados, por exemplo, em modelos de ANOVA (Analysis of Variance, ou Análise de Variância):

```
In [58]: data['key2'] = data['key2'].map({0: 'zero', 1: 'one'})

In [59]: data
Out[59]:
  key1 key2 v1 v2
0 a zero 1 -1.0
1 a one 2 0.0
2 b zero 3 2.5
3 b one 4 -0.5
4 a zero 5 4.0
5 b one 6 -1.2
6 a zero 7 0.2
7 b zero 8 -1.7

In [60]: y, X = patsy.dmatrices('v2 ~ key1 + key2', data)

In [61]: X
Out[61]:
DesignMatrix with shape (8, 3)
  Intercept key1[T.b] key2[T.zero]
          1 0 1
          1 0 0
          1 1 1
```

```
        1 1 0
        1 0 1
        1 1 0
        1 0 1
        1 1 1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
    'key2' (column 2)

In [62]: y, X = patsy.dmatrices('v2 ~ key1 + key2 + key1:key2', data)

In [63]: X
Out[63]:
DesignMatrix with shape (8, 4)
  Intercept key1[T.b] key2[T.zero] key1[T.b]:key2[T.zero]
        1 0 1 0
        1 0 0 0
        1 1 1 1
        1 1 0 0
        1 0 1 0
        1 1 0 0
        1 0 1 0
        1 1 1 1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
    'key2' (column 2)
    'key1:key2' (column 3)
```

O Patsy oferece outras maneiras de transformar dados categorizados, incluindo transformações para termos com uma ordem em particular. Consulte a documentação online para obter mais informações.

13.3 Introdução ao statsmodels

O statsmodels (*http://www.statsmodels.org*) é uma biblioteca Python para adequação de vários tipos de modelos estatísticos; ele realiza testes estatísticos, além de possibilitar a exploração e a visualização de

dados. O statsmodels contém métodos estatísticos frequentistas mais “clássicos”, enquanto os métodos bayesianos e de aprendizado de máquina são encontrados em outras bibliotecas.

Alguns tipos de modelos que se encontram no statsmodels incluem:

- modelos lineares, modelos lineares generalizados e modelos lineares robustos;
- modelos lineares de efeitos mistos;
- métodos de ANOVA (Analysis of Variance, ou Análise de Variância);
- processos para séries temporais e modelos de estado-espaço;
- métodos de momentos generalizados.

Nas próximas páginas, usaremos algumas ferramentas básicas do statsmodels e veremos como utilizar as interfaces de modelagem com fórmulas do Patsy e objetos DataFrame do pandas.

Estimando modelos lineares

Há vários tipos de modelos de regressão linear no statsmodels, dos mais básicos (por exemplo, mínimos quadrados ordinários) aos mais complexos (por exemplo, mínimos quadrados ponderados iterativamente).

Os modelos lineares no statsmodels têm duas interfaces principais diferentes: baseadas em array e baseadas em fórmula. Elas são acessadas por meio das seguintes importações de módulos de API:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

Para mostrar como usá-las, geraremos um modelo linear a partir de alguns dados aleatórios:

```
def dnorm(mean, variance, size=1):
    if isinstance(size, int):
        size = size,
    return mean + np.sqrt(variance) * np.random.randn(*size)
```

```
# Para possibilitar uma reprodução
np.random.seed(12345)

N = 100
X = np.c_[dnorm(0, 0.4, size=N),
          dnorm(0, 0.6, size=N),
          dnorm(0, 0.2, size=N)]
eps = dnorm(0, 0.1, size=N)
beta = [0.1, 0.3, 0.5]

y = np.dot(X, beta) + eps
```

Nesse caso, escrevi o modelo “verdadeiro” com parâmetros beta conhecidos. No exemplo, dnorm é uma função auxiliar para gerar dados normalmente distribuídos, com uma média e uma variância específicas. Então agora temos:

```
In [66]: X[:5]
Out[66]:
array([[-0.1295, -1.2128, 0.5042],
       [ 0.3029, -0.4357, -0.2542],
       [-0.3285, -0.0253, 0.1384],
       [-0.3515, -0.7196, -0.2582],
       [ 1.2433, -0.3738, -0.5226]])

In [67]: y[:5]
Out[67]: array([ 0.4279, -0.6735, -0.0909, -0.4895, -0.1289])
```

Uma adequação a um modelo linear em geral é feita com um termo de interceptação, conforme vimos antes com o Patsy. A função sm.add_constant pode adicionar uma coluna de interceptação em uma matriz existente:

```
In [68]: X_model = sm.add_constant(X)

In [69]: X_model[:5]
Out[69]:
array([[ 1. , -0.1295, -1.2128, 0.5042],
       [ 1. , 0.3029, -0.4357, -0.2542],
       [ 1. , -0.3285, -0.0253, 0.1384],
       [ 1. , -0.3515, -0.7196, -0.2582],
       [ 1. , 1.2433, -0.3738, -0.5226]])
```

A classe sm.OLS pode fazer a adequação a uma regressão linear de mínimos quadrados ordinários:

```
In [70]: model = sm.OLS(y, X)
```

O método fit do modelo devolve um objeto de resultado da regressão contendo os parâmetros estimados do modelo e outros diagnósticos:

```
In [71]: results = model.fit()

In [72]: results.params
Out[72]: array([ 0.1783, 0.223 , 0.501 ])
```

O método summary em results é capaz de exibir uma saída de diagnóstico do detalhamento do modelo:

```
In [73]: print(results.summary())
OLS Regression Results
==============================================================================
Dep. Variable: y R-squared: 0.430
Model: OLS Adj. R-squared: 0.413
Method: Least Squares F-statistic: 24.42
Date: Tue, 17 Oct 2017 Prob (F-statistic): 7.44e-12
Time: 13:34:52 Log-Likelihood: -34.305
No. Observations: 100 AIC: 74.61
Df Residuals: 97 BIC: 82.42
Df Model: 3
Covariance Type: nonrobust
==============================================================================
                 coef std err t P>|t| [0.025 0.975]
------------------------------------------------------------------------------
x1 0.1783 0.053 3.364 0.001 0.073 0.283
x2 0.2230 0.046 4.818 0.000 0.131 0.315
x3 0.5010 0.080 6.237 0.000 0.342 0.660
==============================================================================
Omnibus: 4.662 Durbin-Watson: 2.201
Prob(Omnibus): 0.097 Jarque-Bera (JB): 4.098
Skew: 0.481 Prob(JB): 0.129
Kurtosis: 3.243 Cond. No. 1.74
```

```
==============================================================================
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

Os nomes dos parâmetros nesse caso receberam os nomes genéricos x1, x2 e assim por diante. Suponha, por outro lado, que todos os parâmetros do modelo estejam em um DataFrame:

```
In [74]: data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])

In [75]: data['y'] = y

In [76]: data[:5]
Out[76]:
      col0 col1 col2 y
0 -0.129468 -1.212753 0.504225 0.427863
1 0.302910 -0.435742 -0.254180 -0.673480
2 -0.328522 -0.025302 0.138351 -0.090878
3 -0.351475 -0.719605 -0.258215 -0.489494
4 1.243269 -0.373799 -0.522629 -0.128941
```

Agora podemos usar a API de fórmulas do statsmodels e as strings de fórmula do Patsy:

```
In [77]: results = smf.ols('y ~ col0 + col1 + col2', data=data).fit()

In [78]: results.params
Out[78]:
Intercept 0.033559
col0 0.176149
col1 0.224826
col2 0.514808
dtype: float64

In [79]: results.tvalues
Out[79]:
Intercept 0.952188
col0 3.319754
col1 4.850730
col2 6.303971
dtype: float64
```

Observe como o statsmodels devolveu o resultado na forma de Series com os nomes das colunas do DataFrame associados. Também não é necessário usar add_constant quando utilizamos fórmulas e objetos do pandas.

Se novos dados fora da amostra forem especificados, poderemos calcular os valores previstos, considerando os parâmetros estimados pelo modelo:

```
In [80]: results.predict(data[:5])
Out[80]:
0 -0.002327
1 -0.141904
2 0.041226
3 -0.323070
4 -0.100535
dtype: float64
```

Há várias ferramentas adicionais para análise, diagnósticos e visualização de resultados de modelos lineares no statsmodels, as quais você poderá explorar. Há também outros tipos de modelos lineares além dos mínimos quadrados ordinários.

Estimando processos de séries temporais

Outra classe de modelos do statsmodels é a de análise de séries temporais. Entre eles estão os processos autorregressivos, a filtragem de Kalman e outros modelos estado-espaço, além de modelos autorregressivos multivariados.

Vamos simular alguns dados de séries temporais com uma estrutura autorregressiva e ruído:

```
init_x = 4

import random
values = [init_x, init_x]
N = 1000

b0 = 0.8
b1 = -0.4
```

```
noise = dnorm(0, 0.1, N)
for i in range(N):
    new_x = values[-1] * b0 + values[-2] * b1 + noise[i]
    values.append(new_x)
```

Esses dados têm uma estrutura AR(2) (duas *defasagens*, isto é, lags) com parâmetros 0.8 e –0.4. Quando fazemos a adequação a um modelo AR, talvez o número de termos defasados a ser incluído não seja conhecido, portanto podemos fazer a adequação do modelo com um número maior de defasagens:

```
In [82]: MAXLAGS = 5

In [83]: model = sm.tsa.AR(values)

In [84]: results = model.fit(MAXLAGS)
```

Os parâmetros estimados no resultado têm a interceptação antes e as estimativas para as duas primeiras defasagens a seguir:

```
In [85]: results.params
Out[85]: array([-0.0062, 0.7845, -0.4085, -0.0136, 0.015 , 0.0143])
```

Mais detalhes sobre esses modelos e como interpretar seus resultados estão além do que posso abordar neste livro, mas há muito mais informações a serem desvendadas na documentação do statsmodels.

13.4 Introdução ao scikit-learn

O scikit-learn (*http://scikit-learn.org/)* é um dos kits de ferramentas Python de propósito geral para aprendizado de máquina mais amplamente utilizado e confiável. Ele contém uma variada seleção de métodos padrões supervisionados e não supervisionados de aprendizado de máquina, com ferramentas para seleção e avaliação de modelos, transformação e carga de dados e persistência de modelos. Esses modelos podem ser usados para classificação, clustering, previsão e outras tarefas comuns.

Há excelentes recursos online e impressos, como o scikit-learn e o TensorFlow, para estudar o aprendizado de máquina e aplicar

bibliotecas a fim de resolver problemas do mundo real. Nesta seção, apresentarei uma rápida amostra do estilo de API do scikit-learn.

Atualmente (quando escrevi este livro), o scikit-learn não tem uma integração profunda com o pandas, embora haja alguns pacotes add-on de terceiros em desenvolvimento. O pandas, porém, pode ser muito útil para tratamento de conjuntos de dados antes da adequação a um modelo.

Como exemplo, usarei um conjunto de dados atualmente clássico de uma competição do Kaggle (*https://www.kaggle.com/c/titanic*) sobre as taxas de sobrevivência de passageiros do *Titanic*, navio que afundou em 1912. Carregaremos o conjunto de dados de teste e de treinamento com o pandas:

```
In [86]: train = pd.read_csv('datasets/titanic/train.csv')

In [87]: test = pd.read_csv('datasets/titanic/test.csv')

In [88]: train[:4]
Out[88]:
   PassengerId Survived Pclass \
0 1 0 3
1 2 1 1
2 3 1 3
3 4 1 1
                                                Name Sex Age SibSp \
0 Braund, Mr. Owen Harris male 22.0 1
1 Cumings, Mrs. John Bradley (Florence Briggs Th... female 38.0 1
2 Heikkinen, Miss. Laina female 26.0 0
3 Futrelle, Mrs. Jacques Heath (Lily May Peel) female 35.0 1
   Parch Ticket Fare Cabin Embarked
0 0 A/5 21171 7.2500 NaN S
1 0 PC 17599 71.2833 C85 C
2 0 STON/O2. 3101282 7.9250 NaN S
3 0 113803 53.1000 C123 S
```

Bibliotecas como o statsmodels e o scikit-learn em geral não podem receber dados ausentes, portanto analisaremos as colunas para ver se há alguma que contenha dados ausentes:

```
In [89]: train.isnull().sum()
Out[89]:
PassengerId 0
Survived 0
Pclass 0
Name 0
Sex 0
Age 177
SibSp 0
Parch 0
Ticket 0
Fare 0
Cabin 687
Embarked 2
dtype: int64

In [90]: test.isnull().sum()
Out[90]:
PassengerId 0
Pclass 0
Name 0
Sex 0
Age 86
SibSp 0
Parch 0
Ticket 0
Fare 1
Cabin 327
Embarked 0
dtype: int64
```

Nos exemplos de estatística e de aprendizado de máquina como esse, uma tarefa típica é prever se um passageiro sobreviveria, com base nas características dos dados. Fazemos a adequação de um modelo a um conjunto de dados de *treinamento* e, então, uma avaliação é realizada em um conjunto de dados de testes que não estão na amostra.

Gostaria de usar `Age` como um dado de previsão, porém ele tem dados ausentes. Há algumas maneiras de imputar um valor aos

dados ausentes, mas utilizarei uma alternativa simples: a mediana do conjunto de dados de treinamento será usada para preencher os nulos nas duas tabelas:

```
In [91]: impute_value = train['Age'].median()

In [92]: train['Age'] = train['Age'].fillna(impute_value)

In [93]: test['Age'] = test['Age'].fillna(impute_value)
```

Agora, temos que especificar os nossos modelos. Adicionarei uma coluna `IsFemale` como uma versão codificada da coluna `'Sex'`:

```
In [94]: train['IsFemale'] = (train['Sex'] == 'female').astype(int)

In [95]: test['IsFemale'] = (test['Sex'] == 'female').astype(int)
```

Então decidiremos sobre algumas variáveis do modelo e criaremos arrays NumPy:

```
In [96]: predictors = ['Pclass', 'IsFemale', 'Age']

In [97]: X_train = train[predictors].values

In [98]: X_test = test[predictors].values

In [99]: y_train = train['Survived'].values

In [100]: X_train[:5]
Out[100]:
array([[ 3., 0., 22.],
       [ 1., 1., 38.],
       [ 3., 1., 26.],
       [ 1., 1., 35.],
       [ 3., 0., 35.]])

In [101]: y_train[:5]
Out[101]: array([0, 1, 1, 1, 0])
```

Não farei nenhuma argumentação de que esse seja um bom modelo nem de que essas características tenham sido definidas de forma apropriada. Usaremos o modelo `LogisticRegression` do scikit-learn e criaremos uma instância do modelo:

```
In [102]: from sklearn.linear_model import LogisticRegression

In [103]: model = LogisticRegression()
```

De modo semelhante ao statsmodels, podemos fazer a adequação desse modelo aos dados de treinamento utilizando o método fit do modelo:

```
In [104]: model.fit(X_train, y_train)
Out[104]:
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

Podemos agora fazer previsões para o conjunto de dados de teste usando model.predict:

```
In [105]: y_predict = model.predict(X_test)

In [106]: y_predict[:10]
Out[106]: array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0])
```

Se você tivesse os valores verdadeiros para o conjunto de dados de teste, seria possível calcular um percentual de exatidão ou outra métrica de erro:

```
(y_true == y_predict).mean()
```

Na prática, em geral, há muitas camadas adicionais de complexidade no treinamento do modelo. Muitos modelos têm parâmetros que podem ser ajustados, e há técnicas como a *validação cruzada*, que podem ser usadas para ajuste de parâmetros a fim de evitar uma adequação exagerada nos dados de treinamento. Com frequência isso pode resultar em um desempenho melhor para as previsões ou em mais robustez para os dados novos.

A validação cruzada funciona por meio da separação dos dados de treinamento a fim de simular uma previsão fora da amostra. Com base na pontuação da exatidão do modelo, como o erro quadrático médio, podemos fazer uma busca de grade (grid search) nos parâmetros do modelo. Alguns modelos, como a regressão logística,

têm classes de estimativa com validação cruzada embutida. Por exemplo, a classe LogisticRegressionCV pode ser usada com um parâmetro que indica o nível de granularidade que uma busca de grade deve ter no parâmetro C de regularização do modelo:

```
In [107]: from sklearn.linear_model import LogisticRegressionCV

In [108]: model_cv = LogisticRegressionCV(10)

In [109]: model_cv.fit(X_train, y_train)
Out[109]:
LogisticRegressionCV(Cs=10, class_weight=None, cv=None, dual=False,
        fit_intercept=True, intercept_scaling=1.0, max_iter=100,
        multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
        refit=True, scoring=None, solver='lbfgs', tol=0.0001, verbose=0)
```

Para fazer uma validação cruzada manualmente, podemos usar a função auxiliar cross_val_score, que cuida do processo de separação dos dados. Por exemplo, para fazer a validação cruzada de nosso modelo com quatro conjuntos separados dos dados de treinamento sem intersecção, podemos executar o seguinte:

```
In [110]: from sklearn.model_selection import cross_val_score

In [111]: model = LogisticRegression(C=10)

In [112]: scores = cross_val_score(model, X_train, y_train, cv=4)

In [113]: scores
Out[113]: array([ 0.7723, 0.8027, 0.7703, 0.7883])
```

A métrica de pontuação padrão é dependente do modelo, mas é possível escolher uma função explícita de pontuação. Modelos de validação cruzada demoram mais para efetuar o treinamento; no entanto, muitas vezes podem levar a um desempenho melhor do modelo.

13.5 Dando prosseguimento à sua educação

Embora eu tenha somente tocado a superfícies de algumas

bibliotecas de modelagem Python, há cada vez mais frameworks para vários tipos de estatísticas e para aprendizado de máquina, sejam implementados em Python ou com uma interface de usuário Python.

O foco deste livro centra-se particularmente no tratamento de dados (data wrangling), mas há muitos outros livros dedicados a ferramentas de modelagem e ciência de dados. Algumas obras excelentes incluem:

- *Introduction to Machine Learning with Python*, de Andreas Mueller e Sarah Guido (O'Reilly);
- *Python Data Science Handbook*, de Jake VanderPlas (O'Reilly);
- *Data science do zero: primeiras regras com o Python*, de Joel Grus (Altabooks);
- *Python Machine Learning*, de Sebastian Raschka (Packt Publishing);
- *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, de Aurélien Géron (O'Reilly).

Embora constituam recursos importantes para o aprendizado, às vezes os livros podem ficar desatualizados quando os softwares de código aberto subjacentes mudam. Ter familiaridade com a documentação dos vários frameworks para estatística e aprendizado de máquina de modo a permanecer atualizado quanto aos recursos mais recentes e a API é uma boa ideia.

CAPÍTULO 14

Exemplos de análises de dados

Agora que alcançamos o final dos principais capítulos deste livro, veremos alguns conjuntos de dados do mundo real. Para cada conjunto de dados, usaremos as técnicas apresentadas neste livro para extrair informações significativas dos dados brutos. As técnicas demonstradas podem ser aplicadas a vários outros tipos de conjuntos de dados, incluindo os seus próprios conjuntos. Este capítulo contém uma coleção de exemplos de conjuntos de dados variados que você pode usar para exercitar as ferramentas vistas neste livro.

Os conjuntos de dados de exemplo se encontram no repositório do GitHub (*https://github.com/wesm/pydata-book*) que acompanha este livro.

14.1 Dados de 1.USA.gov do Bitly

Em 2011, o serviço de abreviatura de URL Bitly (*https://bitly.com/*) fez uma parceria com o site do governo norte-americano USA.gov (*https://www.usa.gov/*) para oferecer um feed de dados anônimos coletados dos usuários que abreviam links terminados com *.gov* ou *.mil*. Em 2011, um feed ativo, assim como imagens instantâneas (snapshots) de hora em hora estavam disponíveis na forma de arquivos-texto que podiam ser baixados. Atualmente (em 2017, quando escrevi o livro), esse serviço está desativado, mas preservamos um dos arquivos de dados para os exemplos da obra.

No caso das imagens instantâneas de hora em hora, cada linha nos arquivos contém um formato comum de dados web conhecido como JSON (JavaScript Object Notation, ou Notação de Objetos

JavaScript). Por exemplo, se lermos somente a primeira linha de um arquivo, poderemos ver algo como:

```
In [5]: path = 'datasets/bitly_usagov/example.txt'

In [6]: open(path).readline()
Out[6]: '{ "a": "Mozilla\\/5.0 (Windows NT 6.1; WOW64) AppleWebKit\\/535.11
(KHTML, like Gecko) Chrome\\/17.0.963.78 Safari\\/535.11", "c": "US", "nk": 1,
"tz": "America\\/New_York", "gr": "MA", "g": "A6qOVH", "h": "wfLQtf", "l":
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r":
"http:\\/\\/www.facebook.com\\/l\\/7AQEFzjSi\\/1.usa.gov\\/wfLQtf", "u":
"http:\\/\\/www.ncbi.nlm.nih.gov\\/pubmed\\/22415991", "t": 1331923247, "hc":
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }\n'
```

Python tem bibliotecas tanto embutidas quanto de terceiros para converter uma string JSON em um objeto de dicionário Python. Nesse caso, usaremos o módulo json e sua função loads chamada a cada linha do arquivo de exemplo que baixamos:

```
import json
path = 'datasets/bitly_usagov/example.txt'
records = [json.loads(line) for line in open(path)]
```

O objeto resultante records agora é uma lista de dicionários Python:

```
In [18]: records[0]
Out[18]:
{'a': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like
Gecko)
Chrome/17.0.963.78 Safari/535.11',
 'al': 'en-US,en;q=0.8',
 'c': 'US',
 'cy': 'Danvers',
 'g': 'A6qOVH',
 'gr': 'MA',
 'h': 'wfLQtf',
 'hc': 1331822918,
 'hh': '1.usa.gov',
 'l': 'orofrog',
 'll': [42.576698, -70.954903],
 'nk': 1,
 'r': 'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',
 't': 1331923247,
```

```
 'tz': 'America/New_York',
 'u': 'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

Contando fusos horários em Python puro

Suponha que estivéssemos interessados em descobrir os fusos horários que ocorrem com mais frequência no conjunto de dados (o campo `tz`). Há muitas maneiras pelas quais poderíamos fazer isso. Em primeiro lugar, vamos extrair uma lista de fusos horários, novamente, usando uma list comprehension (abrangência de lista):

```
In [12]: time_zones = [rec['tz'] for rec in records]
---------------------------------------------------------------------------
KeyError Traceback (most recent call last)
<ipython-input-12-f3fbbc37f129> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]
<ipython-input-12-f3fbbc37f129> in <listcomp>(.0)
----> 1 time_zones = [rec['tz'] for rec in records]
KeyError: 'tz'
```

Opa! O fato é que nem todos os registros têm um campo de fuso horário. É fácil cuidar disso, pois podemos acrescentar a verificação `if 'tz' in rec` no final da list comprehension:

```
In [13]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]

In [14]: time_zones[:10]
Out[14]:
['America/New_York',
 'America/Denver',
 'America/New_York',
 'America/Sao_Paulo',
 'America/New_York',
 'America/New_York',
 'Europe/Warsaw',
 '',
 '',
 '']
```

Basta observar os primeiros dez fusos horários e veremos que alguns deles são desconhecidos (string vazia). Podemos filtrá-los também, mas vamos deixá-los aí por enquanto. Para gerar

contadores por fuso horário, apresentarei duas abordagens: o modo mais difícil (usando somente a biblioteca-padrão de Python) e o modo mais fácil (usando o pandas). Uma forma de fazer a contagem é usar um dicionário para armazenar os contadores enquanto iteramos pelos fusos horários:

```
def get_counts(sequence):
    counts = {}
    for x in sequence:
        if x in counts:
            counts[x] += 1
        else:
            counts[x] = 1
    return counts
```

Usando ferramentas mais sofisticadas da biblioteca-padrão de Python, podemos escrever o mesmo código de modo mais conciso:

```
from collections import defaultdict

def get_counts2(sequence):
    counts = defaultdict(int) # valores serão inicializados com 0
    for x in sequence:
        counts[x] += 1
    return counts
```

Coloquei essa lógica em uma função apenas para deixá-la mais reutilizável. Para usá-la nos fusos horários, basta passar a lista time_zones:

```
In [17]: counts = get_counts(time_zones)

In [18]: counts['America/New_York']
Out[18]: 1251

In [19]: len(time_zones)
Out[19]: 3440
```

Se quisermos os 10 primeiros fusos horários e seus contadores, poderemos fazer um pouco de acrobacias com o dicionário:

```
def top_counts(count_dict, n=10):
    value_key_pairs = [(count, tz) for tz, count in count_dict.items()]
```

```
    value_key_pairs.sort()
    return value_key_pairs[-n:]
```

Temos então:

```
In [21]: top_counts(counts)
Out[21]:
[(33, 'America/Sao_Paulo'),
 (35, 'Europe/Madrid'),
 (36, 'Pacific/Honolulu'),
 (37, 'Asia/Tokyo'),
 (74, 'Europe/London'),
 (191, 'America/Denver'),
 (382, 'America/Los_Angeles'),
 (400, 'America/Chicago'),
 (521, ''),
 (1251, 'America/New_York')]
```

Se fizer pesquisas na biblioteca-padrão de Python, encontrará a classe collections.Counter, que facilitará bastante essa tarefa:

```
In [22]: from collections import Counter

In [23]: counts = Counter(time_zones)

In [24]: counts.most_common(10)
Out[24]:
[('America/New_York', 1251),
 ('', 521),
 ('America/Chicago', 400),
 ('America/Los_Angeles', 382),
 ('America/Denver', 191),
 ('Europe/London', 74),
 ('Asia/Tokyo', 37),
 ('Pacific/Honolulu', 36),
 ('Europe/Madrid', 35),
 ('America/Sao_Paulo', 33)]
```

Contando fusos horários com o pandas

Criar um DataFrame a partir do conjunto original de registros é fácil e basta passar a lista deles para pandas.DataFrame:

```
In [25]: import pandas as pd

In [26]: frame = pd.DataFrame(records)

In [27]: frame.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3560 entries, 0 to 3559
Data columns (total 18 columns):
_heartbeat_ 120 non-null float64
a 3440 non-null object
al 3094 non-null object
c 2919 non-null object
cy 2919 non-null object
g 3440 non-null object
gr 2919 non-null object
h 3440 non-null object
hc 3440 non-null float64
hh 3440 non-null object
kw 93 non-null object
l 3440 non-null object
ll 2919 non-null object
nk 3440 non-null float64
r 3440 non-null object
t 3440 non-null float64
tz 3440 non-null object
u 3440 non-null object
dtypes: float64(4), object(14)
memory usage: 500.7+ KB

In [28]: frame['tz'][:10]
Out[28]:
0 America/New_York
1 America/Denver
2 America/New_York
3 America/Sao_Paulo
4 America/New_York
5 America/New_York
6 Europe/Warsaw
7
8
9
```

```
Name: tz, dtype: object
```

A saída mostrada para frame é a *visão resumida* (summary view), exibida para objetos DataFrame grandes. Podemos então utilizar o método value_counts de Series:

```
In [29]: tz_counts = frame['tz'].value_counts()

In [30]: tz_counts[:10]
Out[30]:
America/New_York 1251
                 521
America/Chicago 400
America/Los_Angeles 382
America/Denver 191
Europe/London 74
Asia/Tokyo 37
Pacific/Honolulu 36
Europe/Madrid 35
America/Sao_Paulo 33
Name: tz, dtype: int64
```

Esses dados podem ser visualizados com a matplotlib. Podemos fazer algumas manipulações para preencher dados de fusos horários desconhecidos e ausentes com um valor substituto nos registros. Substituiremos os valores ausentes com o método fillna e usaremos uma indexação booleana de array para as strings vazias:

```
In [31]: clean_tz = frame['tz'].fillna('Missing')

In [32]: clean_tz[clean_tz == ''] = 'Unknown'

In [33]: tz_counts = clean_tz.value_counts()

In [34]: tz_counts[:10]
Out[34]:
America/New_York 1251
Unknown 521
America/Chicago 400
America/Los_Angeles 382
America/Denver 191
Missing 120
```

```
Europe/London 74
Asia/Tokyo 37
Pacific/Honolulu 36
Europe/Madrid 35
Name: tz, dtype: int64
```

Nesse ponto, podemos usar o pacote seaborn (*http://seaborn.pydata.org/*) para gerar uma plotagem de barras horizontais (veja a Figura 14.1 que mostra a visualização resultante):

```
In [36]: import seaborn as sns

In [37]: subset = tz_counts[:10]

In [38]: sns.barplot(y=subset.index, x=subset.values)
```

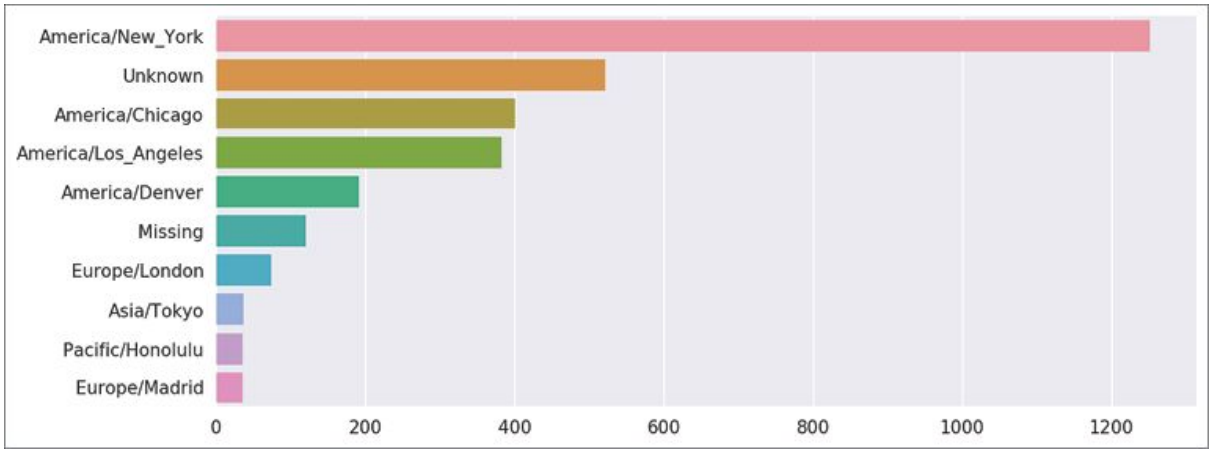


Figura 14.1 – Principais fusos horários nos dados de amostra de 1.usa.gov.

O campo a contém informações sobre o navegador, o dispositivo ou a aplicação utilizados para efetuar a abreviatura do URL:

```
In [39]: frame['a'][1]
Out[39]: 'GoogleMaps/RochesterNY'

In [40]: frame['a'][50]
Out[40]: 'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101
Firefox/10.0.2'

In [41]: frame['a'][51][:50] # linha longa
Out[41]: 'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P9'
```

Fazer parse de todas as informações interessantes nessas strings de “agente” pode parecer uma tarefa desanimadora. Uma possível estratégia consiste em dividir o primeiro token da string (correspondente, grosso modo, aos recursos do navegador) e gerar outro resumo do comportamento dos usuários:

```
In [42]: results = pd.Series([x.split()[0] for x in frame.a.dropna()])

In [43]: results[:5]
Out[43]:
0 Mozilla/5.0
1 GoogleMaps/RochesterNY
2 Mozilla/4.0
3 Mozilla/5.0
4 Mozilla/5.0
dtype: object

In [44]: results.value_counts()[:8]
Out[44]:
Mozilla/5.0 2594
Mozilla/4.0 601
GoogleMaps/RochesterNY 121
Opera/9.80 34
TEST_INTERNET_AGENT 24
GoogleProducer 21
Mozilla/6.0 5
BlackBerry8520/5.0.0.681 4
dtype: int64
```

Suponha agora que quiséssemos decompor os principais fusos horários em usuários de Windows e usuários que não utilizam Windows. Como simplificação, vamos supor que um usuário use Windows se a string 'Windows' estiver na string de agente. Como alguns dos agentes não estão presentes, eles serão excluídos dos dados:

```
In [45]: cframe = frame[frame.a.notnull()]
```

Queremos então calcular um valor que informe se cada linha é Windows ou não:

```
In [47]: cframe['os'] = np.where(cframe['a'].str.contains('Windows'),
```

```
....: 'Windows', 'Not Windows')

In [48]: cframe['os'][:5]
Out[48]:
0 Windows
1 Not Windows
2 Windows
3 Not Windows
4 Windows
Name: os, dtype: object
```

Em seguida, podemos agrupar os dados de acordo com a coluna de fusos horários e essa nova lista de sistemas operacionais:

```
In [49]: by_tz_os = cframe.groupby(['tz', 'os'])
```

Os contadores de grupo, de modo análogo à função value_counts, podem ser calculados com size. Esse resultado é então redimensionado para uma tabela usando unstack:

```
In [50]: agg_counts = by_tz_os.size().unstack().fillna(0)

In [51]: agg_counts[:10]
Out[51]:
os Not Windows Windows
tz
                                245.0 276.0
Africa/Cairo 0.0 3.0
Africa/Casablanca 0.0 1.0
Africa/Ceuta 0.0 2.0
Africa/Johannesburg 0.0 1.0
Africa/Lusaka 0.0 1.0
America/Anchorage 4.0 1.0
America/Argentina/Buenos_Aires 1.0 0.0
America/Argentina/Cordoba 0.0 1.0
America/Argentina/Mendoza 0.0 1.0
```

Por fim, vamos selecionar os principais fusos horários em geral. Para isso, construirei um array de índices indiretos a partir dos contadores de linha em agg_counts:

```
# Use para deixar em ordem crescente
In [52]: indexer = agg_counts.sum(1).argsort()
```

```
In [53]: indexer[:10]
Out[53]:
tz
                                24
Africa/Cairo 20
Africa/Casablanca 21
Africa/Ceuta 92
Africa/Johannesburg 87
Africa/Lusaka 53
America/Anchorage 54
America/Argentina/Buenos_Aires 57
America/Argentina/Cordoba 26
America/Argentina/Mendoza 55
dtype: int64
```

Usarei take para selecionar as linhas nessa ordem e então fatiarei as dez últimas linhas (os maiores valores):

```
In [54]: count_subset = agg_counts.take(indexer[-10:])

In [55]: count_subset
Out[55]:
os Not Windows Windows
tz
America/Sao_Paulo 13.0 20.0
Europe/Madrid 16.0 19.0
Pacific/Honolulu 0.0 36.0
Asia/Tokyo 2.0 35.0
Europe/London 43.0 31.0
America/Denver 132.0 59.0
America/Los_Angeles 130.0 252.0
America/Chicago 115.0 285.0
                        245.0 276.0
America/New_York 339.0 912.0
```

O pandas tem um método conveniente chamado nlargest que faz o mesmo:

```
In [56]: agg_counts.sum(1).nlargest(10)
Out[56]:
tz
America/New_York 1251.0
```

```
                    521.0
America/Chicago 400.0
America/Los_Angeles 382.0
America/Denver 191.0
Europe/London 74.0
Asia/Tokyo 37.0
Pacific/Honolulu 36.0
Europe/Madrid 35.0
America/Sao_Paulo 33.0
dtype: float64
```

Então, conforme mostrado no bloco de código anterior, podemos fazer uma plotagem de barras desses dados; criarei uma plotagem de barras empilhadas (stacked) passando um argumento adicional para a função barplot do seaborn (veja a Figura 14.2):

```
# Reorganiza os dados para a plotagem
In [58]: count_subset = count_subset.stack()

In [59]: count_subset.name = 'total'

In [60]: count_subset = count_subset.reset_index()

In [61]: count_subset[:10]
Out[61]:
                tz os total
0 America/Sao_Paulo Not Windows 13.0
1 America/Sao_Paulo Windows 20.0
2 Europe/Madrid Not Windows 16.0
3 Europe/Madrid Windows 19.0
4 Pacific/Honolulu Not Windows 0.0
5 Pacific/Honolulu Windows 36.0
6 Asia/Tokyo Not Windows 2.0
7 Asia/Tokyo Windows 35.0
8 Europe/London Not Windows 43.0
9 Europe/London Windows 31.0

In [62]: sns.barplot(x='total', y='tz', hue='os', data=count_subset)
```

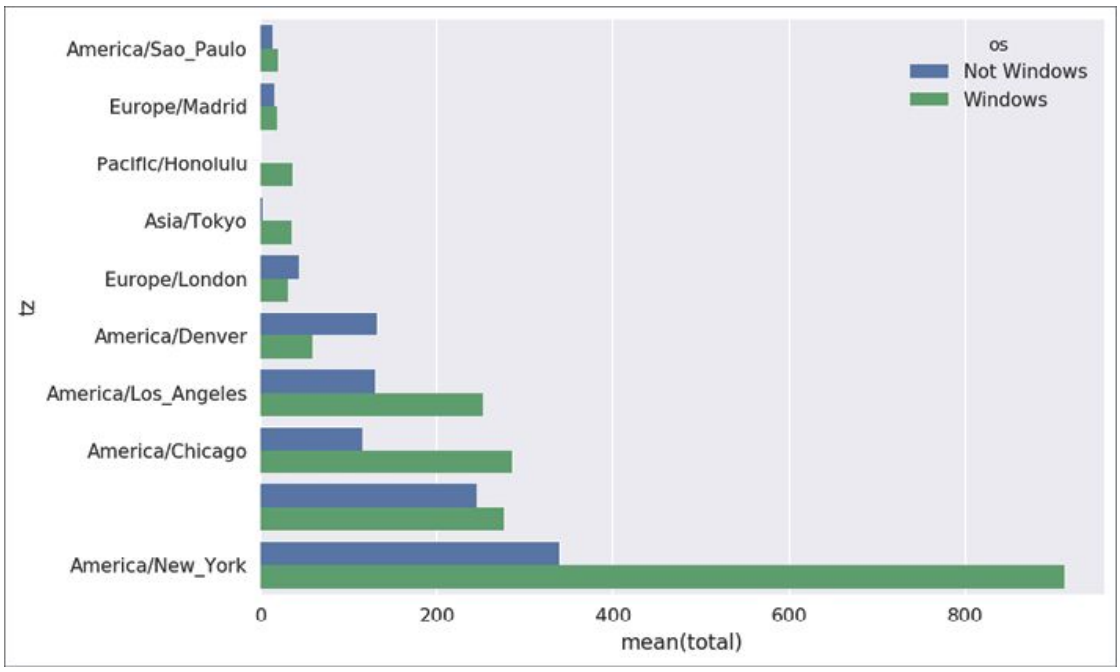


Figura 14.2 – Principais fusos horários separados por usuários de Windows e usuários que não usam Windows.

A plotagem não facilita ver o percentual relativo de usuários do Windows nos grupos menores, portanto vamos normalizar os percentuais dos grupos para que somem 1:

```
def norm_total(group):
    group['normed_total'] = group.total / group.total.sum()
    return group

results = count_subset.groupby('tz').apply(norm_total)
```

Então, geramos a plotagem exibida na Figura 14.3:

```
In [65]: sns.barplot(x='normed_total', y='tz', hue='os', data=results)
```

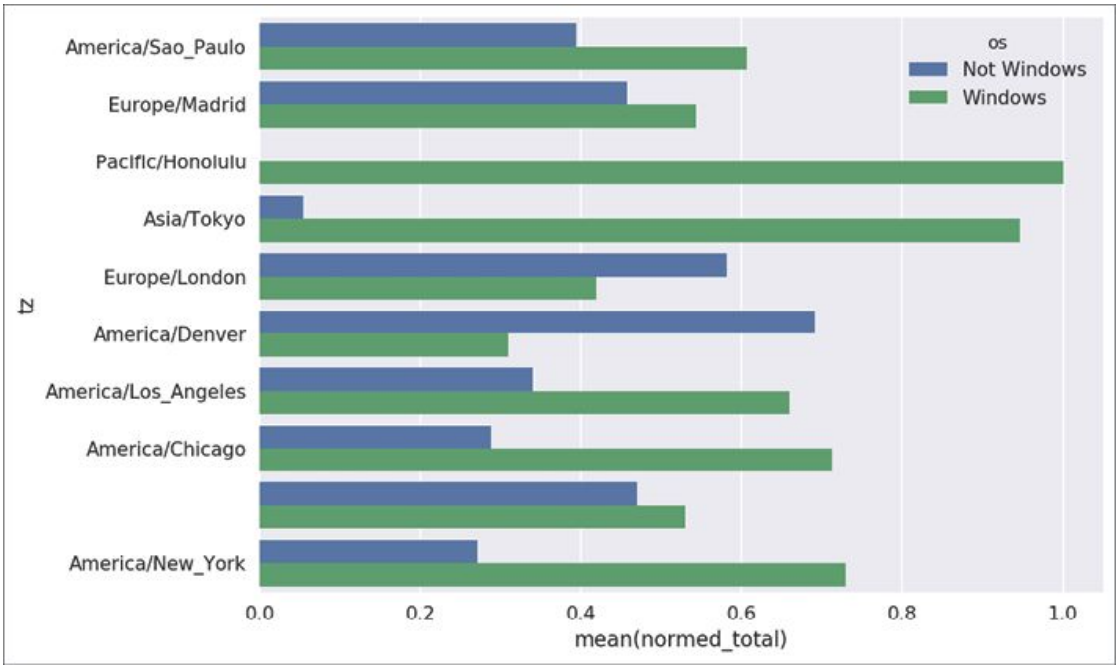


Figura 14.3 – Percentuais de usuários de Windows e usuários que não usam Windows nos fusos horários mais frequentes.

Poderíamos ter calculado a soma normalizada de modo mais eficaz usando o método transform com groupby:

```
In [66]: g = count_subset.groupby('tz')

In [67]: results2 = count_subset.total / g.total.transform('sum')
```

14.2 Conjunto de dados do MovieLens 1M

O GroupLens Research (*http://www.grouplens.org/node/73*) disponibiliza uma série de coleções de dados sobre avaliações de filmes, coletadas de usuários do MovieLens no final dos anos 90 e início dos anos 2000. Os dados contêm avaliações sobre os filmes, metadados (gêneros e ano), além de dados demográficos sobre os usuários (idade, CEP, identificação de gênero e profissão). Dados como esses muitas vezes são de interesse no desenvolvimento de sistemas de recomendação baseados em algoritmos de aprendizado de máquina. Embora não exploraremos técnicas de aprendizado de máquina em detalhes neste livro, mostrarei como manipular conjuntos de dados como esses para que tenham o

formato exato necessário a você.

O conjunto de dados MovieLens 1M contém um milhão de avaliações coletadas de 6 mil usuários sobre 4 mil filmes. Os dados estão espalhados em três tabelas: avaliações, informações de usuários e informações sobre filmes. Depois de extrair os dados do arquivo ZIP, podemos carregar cada tabela em um objeto Dataframe do pandas usando pandas.read_table:

```
import pandas as pd

# Reduz a exibição
pd.options.display.max_rows = 10

unames = ['user_id', 'gender', 'age', 'occupation', 'zip']
users = pd.read_table('datasets/movielens/users.dat', sep='::',
                      header=None, names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('datasets/movielens/ratings.dat', sep='::',
                        header=None, names=rnames)

mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
                       header=None, names=mnames)
```

É possível conferir se tudo ocorreu de forma bem-sucedida observando as primeiras linhas de cada DataFrame com a sintaxe de fatias de Python:

```
In [69]: users[:5]
Out[69]:
   user_id gender age occupation zip
0 1 F 1 10 48067
1 2 M 56 16 70072
2 3 M 25 15 55117
3 4 M 45 7 02460
4 5 M 25 20 55455

In [70]: ratings[:5]
Out[70]:
   user_id movie_id rating timestamp
```

```
0 1 1193 5 978300760
1 1 661 3 978302109
2 1 914 3 978301968
3 1 3408 4 978300275
4 1 2355 5 978824291

In [71]: movies[:5]
Out[71]:
   movie_id title genres
0 1 Toy Story (1995) Animation|Children's|Comedy
1 2 Jumanji (1995) Adventure|Children's|Fantasy
2 3 Grumpier Old Men (1995) Comedy|Romance
3 4 Waiting to Exhale (1995) Comedy|Drama
4 5 Father of the Bride Part II (1995) Comedy

In [72]: ratings
Out[72]:
        user_id movie_id rating timestamp
0 1 1193 5 978300760
1 1 661 3 978302109
2 1 914 3 978301968
3 1 3408 4 978300275
4 1 2355 5 978824291
... ... ... ... ...
1000204 6040 1091 1 956716541
1000205 6040 1094 5 956704887
1000206 6040 562 5 956704746
1000207 6040 1096 4 956715648
1000208 6040 1097 4 956715569
[1000209 rows x 4 columns]
```

Observe que as idades e as profissões estão codificadas como inteiros referenciando grupos descritos no arquivo *README* do conjunto de dados. Analisar os dados espalhados em três tabelas não é uma tarefa simples; por exemplo, suponha que você quisesse calcular as avaliações médias para um filme em particular, de acordo com o sexo e a idade. Como veremos, isso será muito mais simples de fazer se todos os dados estiverem combinados em uma só tabela. Usando a função merge do pandas, inicialmente faremos o merge de ratings e users, e então faremos o merge desse resultado

com os dados de movies. O pandas infere quais colunas deve usar como as chaves do merge (ou da *junção*) com base nos nomes que se sobrepõem:

```
In [73]: data = pd.merge(pd.merge(ratings, users), movies)

In [74]: data
Out[74]:
        user_id movie_id rating timestamp gender age occupation zip \
0 1 1193 5 978300760 F 1 10 48067
1 2 1193 5 978298413 M 56 16 70072
2 12 1193 4 978220179 M 25 12 32793
3 15 1193 4 978199279 M 25 7 22903
4 17 1193 5 978158471 M 50 1 95350
... ... ... ... ... ... ... ... ...
1000204 5949 2198 5 958846401 M 18 17 47901
1000205 5675 2703 3 976029116 M 35 14 30030
1000206 5780 2845 1 958153068 M 18 17 92886
1000207 5851 3607 5 957756608 F 18 20 55410
1000208 5938 2909 4 957273353 M 25 1 35401
                                        title genres
0 One Flew Over the Cuckoo's Nest (1975) Drama
1 One Flew Over the Cuckoo's Nest (1975) Drama
2 One Flew Over the Cuckoo's Nest (1975) Drama
3 One Flew Over the Cuckoo's Nest (1975) Drama
4 One Flew Over the Cuckoo's Nest (1975) Drama
... ... ...
1000204 Modulations (1998) Documentary
1000205 Broken Vessels (1998) Drama
1000206 White Boys (1999) Drama
1000207 One Little Indian (1973) Comedy|Drama|Western
1000208 Five Wives, Three Secretaries and Me (1998) Documentary
[1000209 rows x 10 columns]

In [75]: data.iloc[0]
Out[75]:
user_id 1
movie_id 1193
rating 5
timestamp 978300760
gender F
```

```
age 1
occupation 10
zip 48067
title One Flew Over the Cuckoo's Nest (1975)
genres Drama
Name: 0, dtype: object
```

Para obter as avaliações médias de cada filme agrupadas pela identificação de gênero da pessoa, o método pivot_table pode ser usado:

```
In [76]: mean_ratings = data.pivot_table('rating', index='title',
   ....: columns='gender', aggfunc='mean')

In [77]: mean_ratings[:5]
Out[77]:
gender F M
title
$1,000,000 Duck (1971) 3.375000 2.761905
'Night Mother (1986) 3.388889 3.352941
'Til There Was You (1997) 2.675676 2.733333
'burbs, The (1989) 2.793478 2.962085
...And Justice for All (1979) 3.828571 3.689024
```

Com isso, geramos outro DataFrame contendo as avaliações médias com os títulos dos filmes como rótulos das linhas (o “índice”) e o gênero como rótulos das colunas. Inicialmente, filtrarei os filmes que receberam pelo menos 250 avaliações (um número totalmente arbitrário); para isso, agruparei os dados por título e usarei size() para obter uma Series com os tamanhos dos grupos para cada título:

```
In [78]: ratings_by_title = data.groupby('title').size()

In [79]: ratings_by_title[:10]
Out[79]:
title
$1,000,000 Duck (1971) 37
'Night Mother (1986) 70
'Til There Was You (1997) 52
'burbs, The (1989) 303
...And Justice for All (1979) 199
```

```
1-900 (1994) 2
10 Things I Hate About You (1999) 700
101 Dalmatians (1961) 565
101 Dalmatians (1996) 364
12 Angry Men (1957) 616
dtype: int64

In [80]: active_titles = ratings_by_title.index[ratings_by_title >= 250]

In [81]: active_titles
Out[81]:
Index(["burbs, The (1989)', '10 Things I Hate About You (1999)',
       '101 Dalmatians (1961)', '101 Dalmatians (1996)', '12 Angry Men (1957)',
       '13th Warrior, The (1999)', '2 Days in the Valley (1996)',
       '20,000 Leagues Under the Sea (1954)', '2001: A Space Odyssey (1968)',
       '2010 (1984)',
       ...
       'X-Men (2000)', 'Year of Living Dangerously (1982)',
       'Yellow Submarine (1968)', 'You've Got Mail (1998)',
       'Young Frankenstein (1974)', 'Young Guns (1988)',
       'Young Guns II (1990)', 'Young Sherlock Holmes (1985)',
       'Zero Effect (1998)', 'eXistenZ (1999)'],
      dtype='object', name='title', length=1216)
```

O índice dos títulos que receberam no mínimo 250 avaliações pode então ser usado para selecionar as linhas de mean_ratings:

```
# Seleciona linhas no índice
In [82]: mean_ratings = mean_ratings.loc[active_titles]

In [83]: mean_ratings
Out[83]:
gender F M
title
'burbs, The (1989) 2.793478 2.962085
10 Things I Hate About You (1999) 3.646552 3.311966
101 Dalmatians (1961) 3.791444 3.500000
101 Dalmatians (1996) 3.240000 2.911215
12 Angry Men (1957) 4.184397 4.328421
... ... ...
Young Guns (1988) 3.371795 3.425620
Young Guns II (1990) 2.934783 2.904025
```

```
Young Sherlock Holmes (1985) 3.514706 3.363344
Zero Effect (1998) 3.864407 3.723140
eXistenZ (1999) 3.098592 3.289086
[1216 rows x 2 columns]
```

Para ver os principais filmes entre as telespectadoras femininas, podemos ordenar de acordo com a coluna `F` em ordem decrescente:

```
In [85]: top_female_ratings = mean_ratings.sort_values(by='F',
ascending=False)

In [86]: top_female_ratings[:10]
Out[86]:
gender F M
title
Close Shave, A (1995) 4.644444 4.473795
Wrong Trousers, The (1993) 4.588235 4.478261
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950) 4.572650 4.464589
Wallace & Gromit: The Best of Aardman Animation... 4.563107 4.385075
Schindler's List (1993) 4.562602 4.491415
Shawshank Redemption, The (1994) 4.539075 4.560625
Grand Day Out, A (1992) 4.537879 4.293255
To Kill a Mockingbird (1962) 4.536667 4.372611
Creature Comforts (1990) 4.513889 4.272277
Usual Suspects, The (1995) 4.513317 4.518248
```

Avaliando a discrepância nas avaliações

Suponha que você quisesse saber quais são os filmes que mais dividem as opiniões entre os telespectadores masculinos e femininos. Uma forma de fazer isso é adicionar uma coluna em `mean_ratings` contendo a diferença nas médias e, então, ordenar de acordo com esse dado:

```
In [87]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F']
```

Ordenar por `'diff'` resulta nos filmes com as maiores diferenças de avaliação, de modo que podemos ver quais são os preferidos pelas mulheres:

```
In [88]: sorted_by_diff = mean_ratings.sort_values(by='diff')

In [89]: sorted_by_diff[:10]
```

```
Out[89]:
gender F M diff
title
Dirty Dancing (1987) 3.790378 2.959596 -0.830782
Jumpin' Jack Flash (1986) 3.254717 2.578358 -0.676359
Grease (1978) 3.975265 3.367041 -0.608224
Little Women (1994) 3.870588 3.321739 -0.548849
Steel Magnolias (1989) 3.901734 3.365957 -0.535777
Anastasia (1997) 3.800000 3.281609 -0.518391
Rocky Horror Picture Show, The (1975) 3.673016 3.160131 -0.512885
Color Purple, The (1985) 4.158192 3.659341 -0.498851
Age of Innocence, The (1993) 3.827068 3.339506 -0.487561
Free Willy (1993) 2.921348 2.438776 -0.482573
```

Se invertermos a ordem das linhas e fatiarmos novamente separando as dez primeiras, teremos os filmes preferidos pelos homens, e que não tiveram avaliações tão altas por parte das mulheres:

```
# Inverta a ordem das linhas e obtenha as 10 primeiras
In [90]: sorted_by_diff[::-1][:10]
Out[90]:
gender F M diff
title
Good, The Bad and The Ugly, The (1966) 3.494949 4.221300 0.726351
Kentucky Fried Movie, The (1977) 2.878788 3.555147 0.676359
Dumb & Dumber (1994) 2.697987 3.336595 0.638608
Longest Day, The (1962) 3.411765 4.031447 0.619682
Cable Guy, The (1996) 2.250000 2.863787 0.613787
Evil Dead II (Dead By Dawn) (1987) 3.297297 3.909283 0.611985
Hidden, The (1987) 3.137931 3.745098 0.607167
Rocky III (1982) 2.361702 2.943503 0.581801
Caddyshack (1980) 3.396135 3.969737 0.573602
For a Few Dollars More (1965) 3.409091 3.953795 0.544704
```

Por outro lado, suponha que quiséssemos os filmes que tiveram mais discrepância entre os telespectadores, independentemente da identificação de gênero. É possível avaliar as discrepâncias pela variância ou pelo desvio-padrão das avaliações:

```
# Desvio-padrão das avaliações com dados agrupados por título
In [91]: rating_std_by_title = data.groupby('title')['rating'].std()
```

```
# Filtra para active_titles
In [92]: rating_std_by_title = rating_std_by_title.loc[active_titles]

# Ordena a Series por valor em ordem decrescente
In [93]: rating_std_by_title.sort_values(ascending=False)[:10]
Out[93]:
title
Dumb & Dumber (1994) 1.321333
Blair Witch Project, The (1999) 1.316368
Natural Born Killers (1994) 1.307198
Tank Girl (1995) 1.277695
Rocky Horror Picture Show, The (1975) 1.260177
Eyes Wide Shut (1999) 1.259624
Evita (1996) 1.253631
Billy Madison (1995) 1.249970
Fear and Loathing in Las Vegas (1998) 1.246408
Bicentennial Man (1999) 1.245533
Name: rating, dtype: float64
```

Talvez você tenha notado que os gêneros dos filmes estão especificados em uma string separada com pipe (|). Se você quisesse fazer alguma análise por gênero do filme, seria necessário um trabalho adicional para transformar as informações de gênero em um formato mais apropriado.

14.3 Nomes e bebês americanos de 1880 a 2010

O SSA (Social Security Administration, ou Administração de Seguridade Social) dos Estados Unidos disponibilizou dados sobre a frequência de nomes de bebês de 1880 até o presente. Hadley Wickham, autor de vários pacotes populares de R, tem frequentemente usado esse conjunto de dados para demonstração de manipulação de dados em R.

Precisamos fazer um tratamento para carregar esse conjunto de dados, mas, feito isso, teremos um DataFrame com a seguinte aparência:

```
In [4]: names.head(10)
```

```
Out[4]:
      name sex births year
0 Mary F 7065 1880
1 Anna F 2604 1880
2 Emma F 2003 1880
3 Elizabeth F 1939 1880
4 Minnie F 1746 1880
5 Margaret F 1578 1880
6 Ida F 1472 1880
7 Alice F 1414 1880
8 Bertha F 1320 1880
9 Sarah F 1288 1880
```

Há muitas tarefas que você pode querer fazer com o conjunto de dados:

- visualizar a proporção de bebês, dado um nome em particular (o seu próprio nome ou outro), no tempo;
- determinar a posição de classificação relativa de um nome;
- determinar os nomes mais populares a cada ano ou os nomes cuja popularidade aumentou ou diminuiu mais;
- analisar tendências nos nomes: vogais, consoantes, tamanhos, diversidade em geral, mudanças na grafia, primeira e última letras;
- analisar origens externas para as tendências: nomes bíblicos, celebridades, mudanças demográficas.

Com as ferramentas deste livro, muitos desses tipos de análise estão à disposição, portanto descreverei alguns deles.

Atualmente (quando escrevi este livro), o SSA dos Estados Unidos disponibiliza arquivos de dados, um por ano, contendo o número total de nascimentos para cada combinação de sexo/nome. O arquivo bruto com esses dados pode ser obtido em *http://www.ssa.gov/oact/babynames/limits.html*.

Caso essa página tenha sido movida para outro lugar quando você estiver lendo este livro, é bem provável que ela possa ser localizada novamente com uma pesquisa na internet. Depois de fazer

download do arquivo de “dados nacionais” *names.zip* e descompactá-lo, você terá um diretório contendo uma série de arquivos como *yob1880.txt*. Usarei o comando head do Unix para observar as dez primeiras linhas de um dos arquivos (no Windows, você pode usar o comando more ou abri-lo em um editor de texto):

```
In [94]: !head -n 10 datasets/babynames/yob1880.txt
Mary,F,7065
Anna,F,2604
Emma,F,2003
Elizabeth,F,1939
Minnie,F,1746
Margaret,F,1578
Ida,F,1472
Alice,F,1414
Bertha,F,1320
Sarah,F,1288
```

Como já estão em um formato organizado, separados por vírgula, esses dados poderão ser carregados em um DataFrame com pandas.read_csv:

```
In [95]: import pandas as pd

In [96]: names1880 = pd.read_csv('datasets/babynames/yob1880.txt',
   ....: names=['name', 'sex', 'births'])

In [97]: names1880
Out[97]:
            name sex births
0 Mary F 7065
1 Anna F 2604
2 Emma F 2003
3 Elizabeth F 1939
4 Minnie F 1746
... ... .. ...
1995 Woodie M 5
1996 Worthy M 5
1997 Wright M 5
1998 York M 5
1999 Zachariah M 5
```

```
[2000 rows x 3 columns]
```

Esses arquivos contêm apenas os nomes com no mínimo cinco ocorrências em cada ano, portanto, por questões de simplicidade, podemos utilizar a soma da coluna de nascimentos por sexo como o número total de nascimentos nesse ano:

```
In [98]: names1880.groupby('sex').births.sum()
Out[98]:
sex
F 90993
M 110493
Name: births, dtype: int64
```

Como o conjunto de dados está separado em arquivos por ano, uma das primeiras tarefas é reunir todos os dados em um único DataFrame e então adicionar um campo year. Isso pode ser feito usando pandas.concat:

```
years = range(1880, 2011)

pieces = []
columns = ['name', 'sex', 'births']

for year in years:
    path = 'datasets/babynames/yob%d.txt' % year
    frame = pd.read_csv(path, names=columns)

    frame['year'] = year
    pieces.append(frame)

# Concatena tudo em um único DataFrame
names = pd.concat(pieces, ignore_index=True)
```

Há alguns pontos a serem observados aqui. Em primeiro lugar, lembre-se de que concat por padrão une objetos DataFrame por linhas. Em segundo lugar, você deve passar ignore_index=True porque não estamos interessados em preservar os números das linhas originais devolvidos por read_csv. Desse modo, temos agora um DataFrame bem grande contendo todos os dados sobre os nomes:

```
In [100]: names
```

```
Out[100]:
            name sex births year
0 Mary F 7065 1880
1 Anna F 2604 1880
2 Emma F 2003 1880
3 Elizabeth F 1939 1880
4 Minnie F 1746 1880
... ... .. ... ...
1690779 Zymaire M 5 2010
1690780 Zyonne M 5 2010
1690781 Zyquarius M 5 2010
1690782 Zyran M 5 2010
1690783 Zzyzx M 5 2010

[1690784 rows x 4 columns]
```

Com esses dados em mãos, já podemos começar a agregá-los em níveis de ano e sexo usando groupby ou pivot_table (veja a Figura 14.4):

```
In [101]: total_births = names.pivot_table('births', index='year',
   .....: columns='sex', aggfunc=sum)

In [102]: total_births.tail()
Out[102]:
sex F M
year
2006 1896468 2050234
2007 1916888 2069242
2008 1883645 2032310
2009 1827643 1973359
2010 1759010 1898382

In [103]: total_births.plot(title='Total births by sex and year')
```

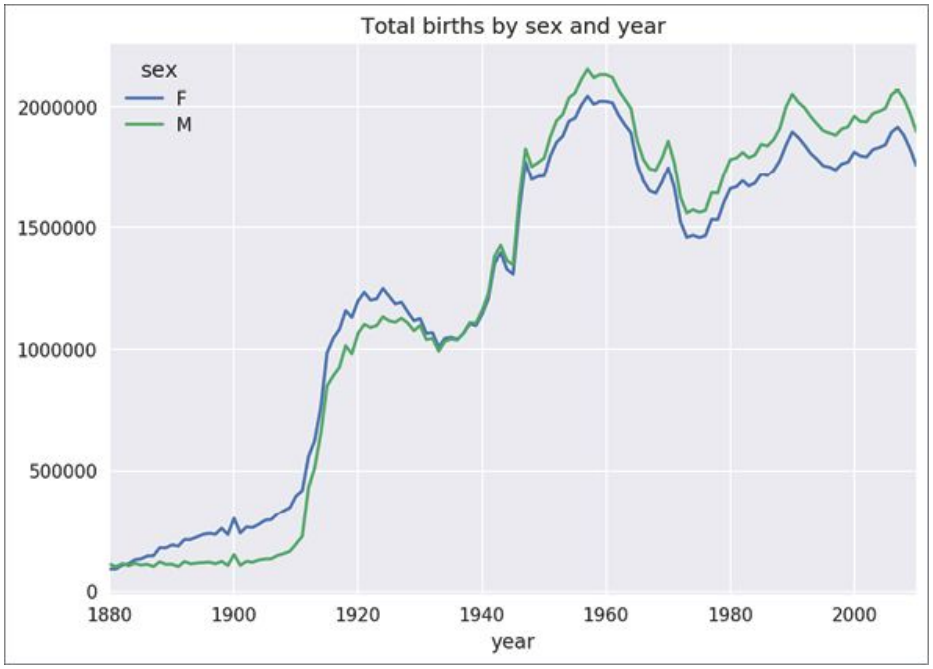


Figura 14.4 – Total de nascimentos por sexo e por ano.

A seguir, vamos inserir uma coluna prop com a fração dos bebês que receberam cada nome, em relação ao número total de nascimentos. Um valor de prop igual a .02 indicaria que 2 de cada 100 bebês receberam um nome em particular. Assim, agruparemos os dados por ano e por sexo e então adicionaremos a nova coluna em cada grupo:

```
def add_prop(group):
    group['prop'] = group.births / group.births.sum()
    return group
names = names.groupby(['year', 'sex']).apply(add_prop)
```

O conjunto de dados completo resultante agora tem as colunas a seguir:

```
In [105]: names
Out[105]:
            name sex births year prop
0 Mary F 7065 1880 0.077643
1 Anna F 2604 1880 0.028618
2 Emma F 2003 1880 0.022013
3 Elizabeth F 1939 1880 0.021309
4 Minnie F 1746 1880 0.019188
... ... .. ... ... ...
```

```
1690779 Zymaire M 5 2010 0.000003
1690780 Zyonne M 5 2010 0.000003
1690781 Zyquarius M 5 2010 0.000003
1690782 Zyran M 5 2010 0.000003
1690783 Zzyzx M 5 2010 0.000003
[1690784 rows x 5 columns]
```

Ao executar uma operação de grupo como essa, muitas vezes é importante fazer uma verificação de sanidade, por exemplo, verificando se a coluna prop tem soma igual a 1 em todos os grupos:

```
In [106]: names.groupby(['year', 'sex']).prop.sum()
Out[106]:
year sex
1880 F 1.0
     M 1.0
1881 F 1.0
     M 1.0
1882 F 1.0
          ...
2008 M 1.0
2009 F 1.0
     M 1.0
2010 F 1.0
     M 1.0
Name: prop, Length: 262, dtype: float64
```

Agora que fizemos isso, extrairei um subconjunto dos dados para facilitar outra análise: os primeiros 1.000 nomes para cada combinação de sexo/ano. Esta é outra operação de grupo:

```
def get_top1000(group):
    return group.sort_values(by='births', ascending=False)[:1000]
grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
# Descarta o índice de grupo, que não será necessário
top1000.reset_index(inplace=True, drop=True)
```

Se preferir uma abordagem do tipo “faça você mesmo”, experimente executar o seguinte:

```
pieces = []
for year, group in names.groupby(['year', 'sex']):
```

```
    pieces.append(group.sort_values(by='births', ascending=False)[:1000])
top1000 = pd.concat(pieces, ignore_index=True)
```

O conjunto de dados resultante agora é bem menor:

```
In [108]: top1000
Out[108]:
            name sex births year prop
0 Mary F 7065 1880 0.077643
1 Anna F 2604 1880 0.028618
2 Emma F 2003 1880 0.022013
3 Elizabeth F 1939 1880 0.021309
4 Minnie F 1746 1880 0.019188
... ... .. ... ... ...
261872 Camilo M 194 2010 0.000102
261873 Destin M 194 2010 0.000102
261874 Jaquan M 194 2010 0.000102
261875 Jaydan M 194 2010 0.000102
261876 Maxton M 193 2010 0.000102
[261877 rows x 5 columns]
```

Usaremos esse conjunto de dados com os 1.000 primeiros nomes nas próximas investigações a serem feitas nos dados.

Analisando tendências para os nomes

Com o conjunto de dados completo e aquele com os 1.000 primeiros nomes em mãos, podemos começar a analisar diversas tendências que nos interessem para os nomes. Separar os 1.000 primeiros nomes em partes referentes a nomes de menino e nomes de menina é fácil de fazer inicialmente:

```
In [109]: boys = top1000[top1000.sex == 'M']


In [110]: girls = top1000[top1000.sex == 'F']
```

Séries temporais simples, como o número de Johns ou de Marys em cada ano, podem ser plotadas, porém exigem um pouco de tratamento dos dados para que sejam mais úteis. Vamos compor uma tabela pivô com o número total de nascimentos por ano e por nome:

```
In [111]: total_births = top1000.pivot_table('births', index='year',
```

```
.....: columns='name',
.....: aggfunc=sum)
```

Agora esses dados podem ser plotados para uma porção de nomes com o método plot de DataFrame (a Figura 14.5 mostra o resultado):

```
In [112]: total_births.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
Columns: 6868 entries, Aaden to Zuri
dtypes: float64(6868)
memory usage: 6.9 MB

In [113]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]

In [114]: subset.plot(subplots=True, figsize=(12, 10), grid=False,
.....: title="Number of births per year")
```

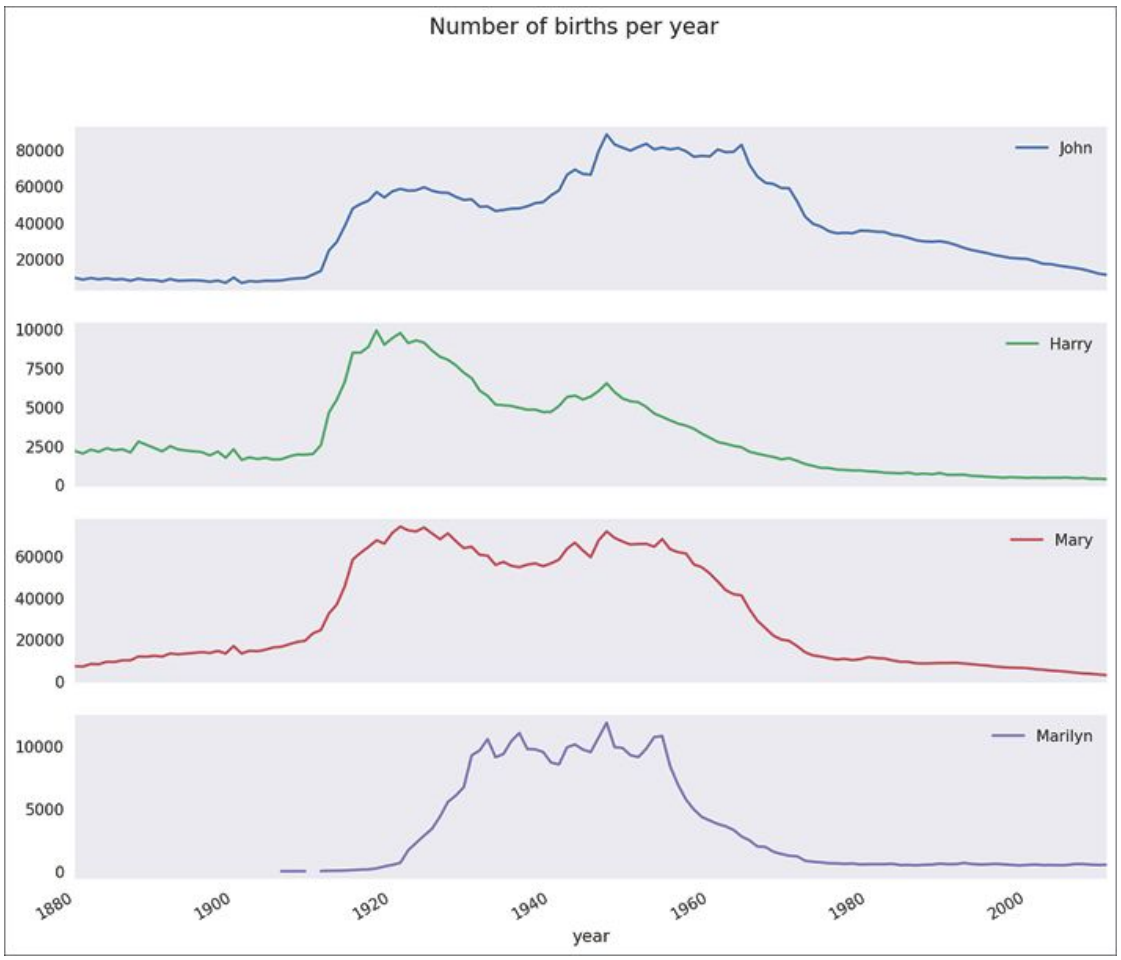


Figura 14.5 – Alguns nomes de meninos e de meninas no tempo.

Ao observar o resultado, poderíamos concluir que esses nomes deixaram de ser os preferidos da população norte-americana. A história, porém, na verdade é mais complicada, conforme exploraremos na próxima seção.

Avaliando o aumento na diversidade dos nomes

Uma explicação para o decréscimo nas plotagens é que menos pais estão escolhendo nomes comuns para seus filhos. Essa hipótese pode ser explorada e confirmada observando os dados. Uma medida é a proporção de nascimentos representada pelos 1.000 nomes mais populares, para os quais fiz uma agregação e plotei por ano e por sexo (a Figura 14.6 mostra a plotagem resultante):

```
In [116]: table = top1000.pivot_table('prop', index='year',
   .....: columns='sex', aggfunc=sum)

In [117]: table.plot(title='Sum of table1000.prop by year and sex',
   .....: yticks=np.linspace(0, 1.2, 13), xticks=range(1880, 2020, 10)
)
```

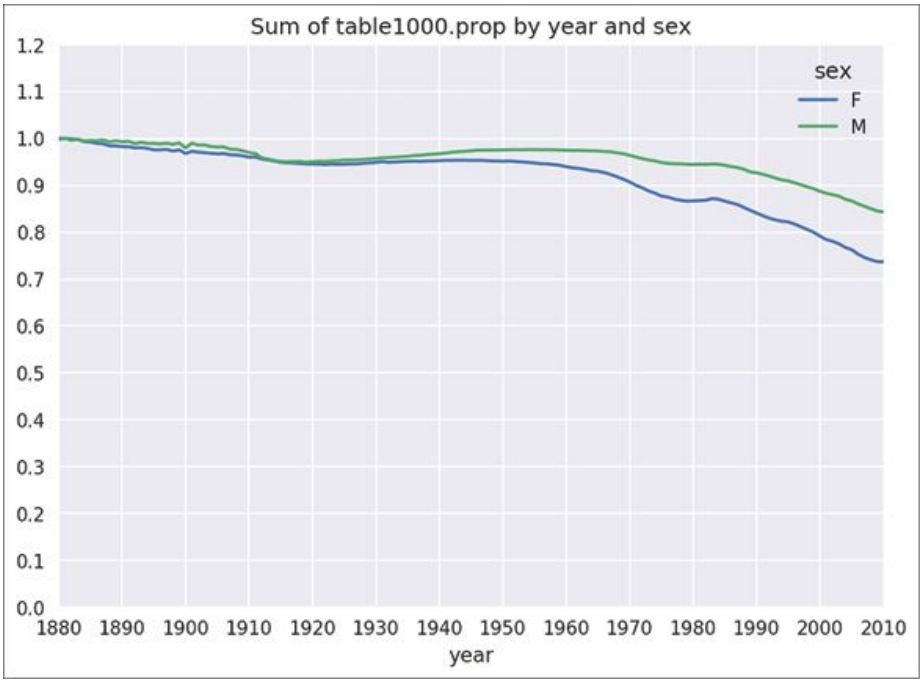


Figura 14.6 – Proporção de nascimentos representada nos 1.000 primeiros nomes, por sexo.

Podemos ver que, realmente, parece haver um aumento na diversidade dos nomes (redução da proporção total nos 1.000 primeiros). Outra métrica interessante é a quantidade de nomes distintos, tomados em ordem de popularidade, do mais popular para o menos, nos primeiros 50% dos nascimentos. Esse número é um pouco mais complicado de calcular. Vamos considerar somente os nomes de meninos em 2010:

```
In [118]: df = boys[boys.year == 2010]

In [119]: df
Out[119]:
        name sex births year prop
260877 Jacob M 21875 2010 0.011523
260878 Ethan M 17866 2010 0.009411
260879 Michael M 17133 2010 0.009025
260880 Jayden M 17030 2010 0.008971
260881 William M 16870 2010 0.008887
... ... .. ... ... ...
261872 Camilo M 194 2010 0.000102
261873 Destin M 194 2010 0.000102
261874 Jaquan M 194 2010 0.000102
261875 Jaydan M 194 2010 0.000102
261876 Maxton M 193 2010 0.000102
[1000 rows x 5 columns]
```

Depois de ordenar prop em ordem decrescente, queremos saber quantos dos nomes mais populares são necessários para alcançar 50%. Você poderia implementar um laço for para isso, porém utilizar uma solução vetorizada com o NumPy seria um pouco mais inteligente. Obter a soma cumulativa cumsum de prop e então chamar o método searchsorted devolve a posição na soma cumulativa em que 0.5 deveria ser inserido para manter a ordenação:

```
In [120]: prop_cumsum = df.sort_values(by='prop',
ascending=False).prop.cumsum()

In [121]: prop_cumsum[:10]
Out[121]:
260877 0.011523
```

```
260878 0.020934
260879 0.029959
260880 0.038930
260881 0.047817
260882 0.056579
260883 0.065155
260884 0.073414
260885 0.081528
260886 0.089621
Name: prop, dtype: float64

In [122]: prop_cumsum.values.searchsorted(0.5)
Out[122]: 116
```

Como os arrays são indexados de zero, ao somar 1, o resultado é igual a 117. Em comparação, em 1900, esse número era bem menor:

```
In [123]: df = boys[boys.year == 1900]

In [124]: in1900 = df.sort_values(by='prop', ascending=False).prop.cumsum()

In [125]: in1900.values.searchsorted(0.5) + 1
Out[125]: 25
```

Podemos agora aplicar essa operação em cada combinação de ano/sexo, usar `groupby` nesses campos e aplicar uma função com `apply`, devolvendo o contador para cada grupo:

```
def get_quantile_count(group, q=0.5):
    group = group.sort_values(by='prop', ascending=False)
    return group.prop.cumsum().values.searchsorted(q) + 1

diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_count)
diversity = diversity.unstack('sex')
```

Esse DataFrame `diversity` resultante agora tem duas séries temporais, uma para cada sexo, indexada por ano. Ele pode ser inspecionado no IPython e plotado como antes (veja a Figura 14.7):

```
In [128]: diversity.head()
Out[128]:
sex F M
```

```
year
1880 38 14
1881 38 14
1882 38 15
1883 39 15
1884 39 16

In [129]: diversity.plot(title="Number of popular names in top 50%")
```

Como podemos ver, os nomes de menina têm sido sempre mais diversificados que os nomes de meninos, e isso apenas se acentuou com o passar do tempo. Outras análises sobre o que, exatamente, determina essa diversidade, como o aumento de grafias alternativas, serão deixadas a cargo do leitor.

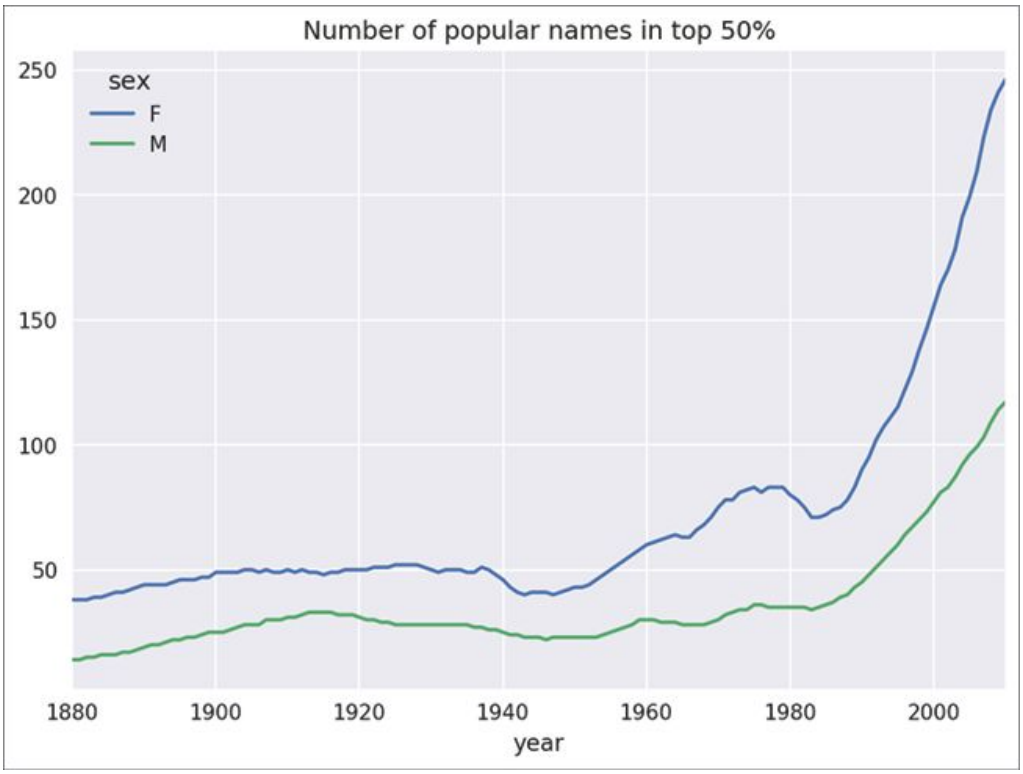


Figura 14.7 – Plotagem da métrica de diversidade por ano.

A revolução da “última letra”

Em 2007, a pesquisadora de nomes de bebês Laura Wattenberg destacou em seu site (*http://www.babynamewizard.com/*) que a distribuição de nomes de meninos de acordo com a última letra

havia mudado significativamente nos últimos 100 anos. Para conferir isso, inicialmente fizemos a agregação de todos os nascimentos do conjunto de dados completo por ano, sexo e a última letra:

```
# extrai a última letra da coluna de nomes
get_last_letter = lambda x: x[-1]
last_letters = names.name.map(get_last_letter)
last_letters.name = 'last_letter'

table = names.pivot_table('births', index=last_letters,
                          columns=['sex', 'year'], aggfunc=sum)
```

Então selecionamos três anos representativos dessa história e exibimos as primeiras linhas:

```
In [131]: subtable = table.reindex(columns=[1910, 1960, 2010], level='year')

In [132]: subtable.head()
Out[132]:
sex F M
year 1910 1960 2010 1910 1960 2010
last_letter
a 108376.0 691247.0 670605.0 977.0 5204.0 28438.0
b NaN 694.0 450.0 411.0 3912.0 38859.0
c 5.0 49.0 946.0 482.0 15476.0 23125.0
d 6750.0 3729.0 2607.0 22111.0 262112.0 44398.0
e 133569.0 435013.0 313833.0 28655.0 178823.0 129012.0
```

Em seguida, normalizamos a tabela pelo total de nascimentos a fim de gerar uma nova tabela contendo a proporção do total de nascimentos para cada sexo, cujo nome termine com cada letra:

```
In [133]: subtable.sum()
Out[133]:
sex year
F 1910 396416.0
    1960 2022062.0
    2010 1759010.0
M 1910 194198.0
    1960 2132588.0
    2010 1898382.0
dtype: float64
```

```
In [134]: letter_prop = subtable / subtable.sum()

In [135]: letter_prop
Out[135]:
sex F M
year 1910 1960 2010 1910 1960 2010
last_letter
a 0.273390 0.341853 0.381240 0.005031 0.002440 0.014980
b NaN 0.000343 0.000256 0.002116 0.001834 0.020470
c 0.000013 0.000024 0.000538 0.002482 0.007257 0.012181
d 0.017028 0.001844 0.001482 0.113858 0.122908 0.023387
e 0.336941 0.215133 0.178415 0.147556 0.083853 0.067959
... ... ... ... ... ... ...
v NaN 0.000060 0.000117 0.000113 0.000037 0.001434
w 0.000020 0.000031 0.001182 0.006329 0.007711 0.016148
x 0.000015 0.000037 0.000727 0.003965 0.001851 0.008614
y 0.110972 0.152569 0.116828 0.077349 0.160987 0.058168
z 0.002439 0.000659 0.000704 0.000170 0.000184 0.001831
[26 rows x 6 columns]
```

Com as proporções das letras agora em mãos, podemos gerar plotagens de barras para cada sexo, separadas por ano (veja a Figura 14.8):

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
letter_prop['M'].plot(kind='bar', rot=0, ax=axes[0], title='Male')
letter_prop['F'].plot(kind='bar', rot=0, ax=axes[1], title='Female', legend=False)
```

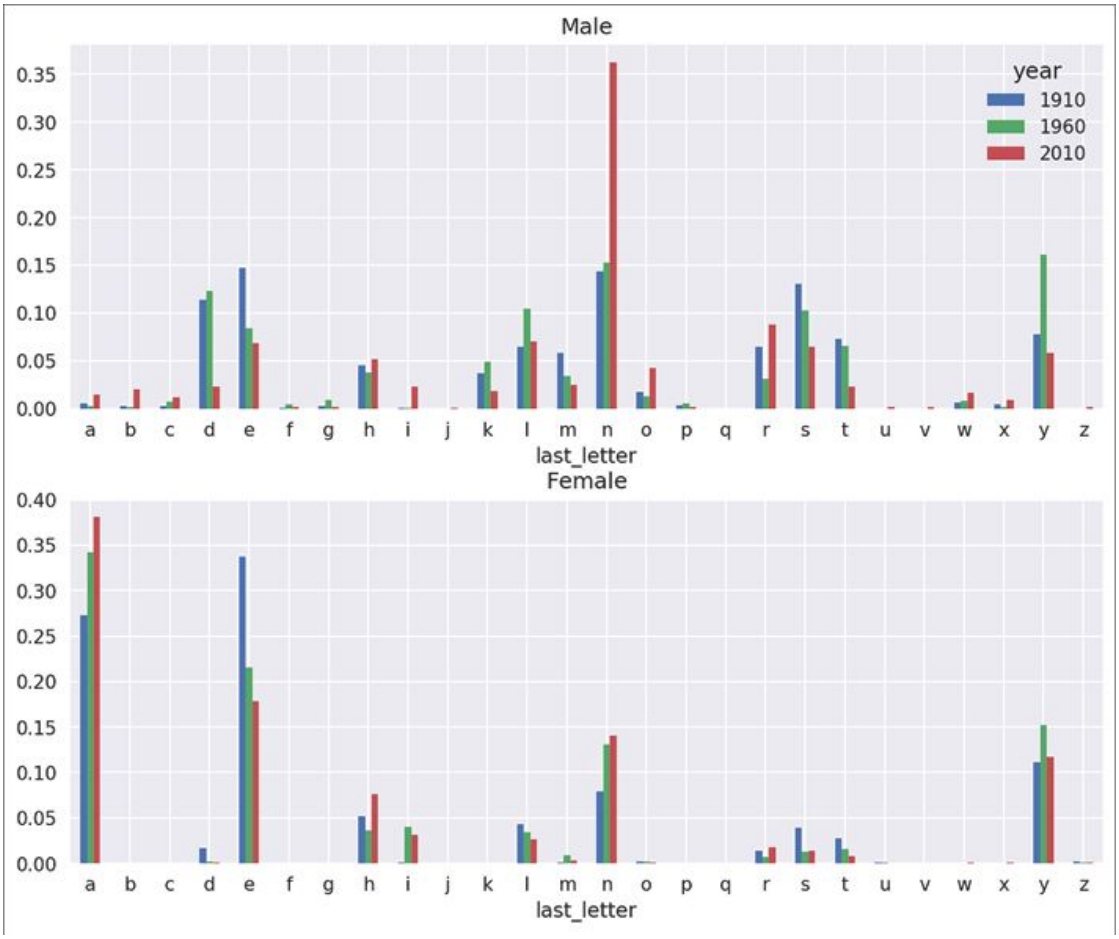


Figura 14.8 – Proporção de nomes de meninos e de meninas que terminam com cada letra.

Como podemos ver, os nomes de meninos que terminam com *n* passaram por aumentos significativos desde os anos 1960. Voltando à tabela completa criada antes, normalizarei novamente por ano e por sexo e selecionarei um subconjunto de letras para nomes de meninos; por fim, farei a transposição para que cada coluna seja uma série temporal:

```
In [138]: letter_prop = table / table.sum()

In [139]: dny_ts = letter_prop.loc[['d', 'n', 'y'], 'M'].T

In [140]: dny_ts.head()
Out[140]:
last_letter d n y
year
```

```
1880 0.083055 0.153213 0.075760
1881 0.083247 0.153214 0.077451
1882 0.085340 0.149560 0.077537
1883 0.084066 0.151646 0.079144
1884 0.086120 0.149915 0.080405
```

Com esse DataFrame de séries temporais em mãos, podemos gerar uma plotagem das tendências no tempo, novamente usando o seu método plot (veja a Figura 14.9):

```
In [143]: dny_ts.plot()
```

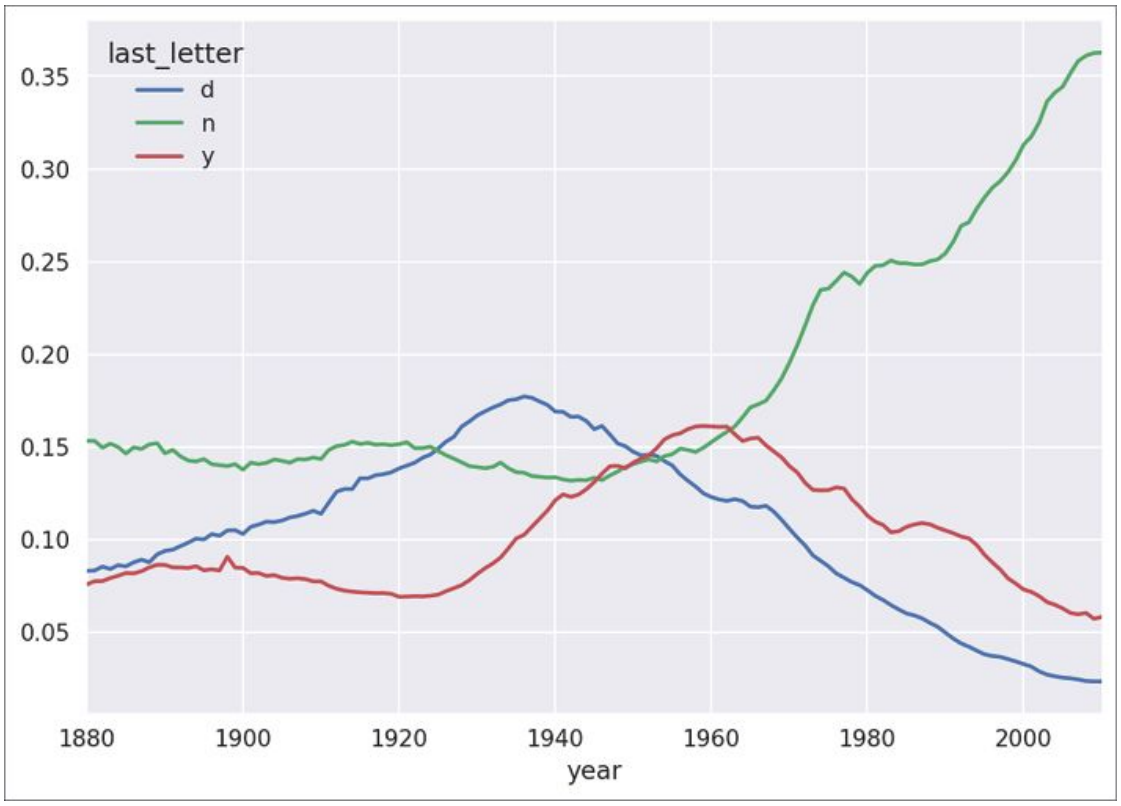


Figura 14.9 – Proporção de meninos nascidos cujos nomes terminam com d/n/y no tempo.

Nomes de meninos que passaram a ser nomes de meninas (e vice-versa)

Outra tendência interessante é observar os nomes de meninos que eram populares para um sexo antes na amostra, mas “mudaram de sexo” atualmente. Um exemplo é o nome Lesley ou Leslie. Voltando ao DataFrame top1000, extrairei uma lista de nomes presentes no

conjunto de dados que comecem com “lesl”:

```
In [144]: all_names = pd.Series(top1000.name.unique())

In [145]: lesley_like = all_names[all_names.str.lower().str.contains('lesl')]

In [146]: lesley_like
Out[146]:
632 Leslie
2294 Lesley
4262 Leslee
4728 Lesli
6103 Lesly
dtype: object
```

A partir desses dados, podemos filtrar para ter somente esses nomes e somar os nascimentos agrupados por nome a fim de ver as frequências relativas:

```
In [147]: filtered = top1000[top1000.name.isin(lesley_like)]

In [148]: filtered.groupby('name').births.sum()
Out[148]:
name
Leslee 1082
Lesley 35022
Lesli 929
Leslie 370429
Lesly 10067
Name: births, dtype: int64
```

Em seguida, vamos agregar por sexo e por ano, e normalizar no ano:

```
In [149]: table = filtered.pivot_table('births', index='year',
   .....: columns='sex', aggfunc='sum')

In [150]: table = table.div(table.sum(1), axis=0)

In [151]: table.tail()
Out[151]:
sex F M
year
```

```
2006 1.0 NaN
2007 1.0 NaN
2008 1.0 NaN
2009 1.0 NaN
2010 1.0 NaN
```

Por fim, agora é possível fazer uma plotagem separada por sexo no tempo (Figura 14.10):

```
In [153]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

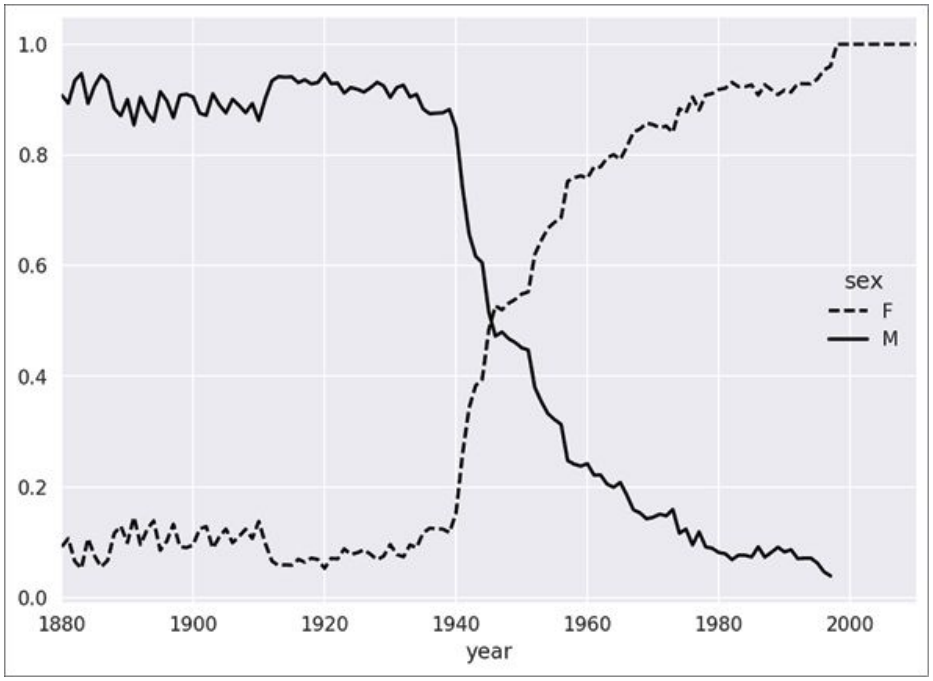


Figura 14.10 – Proporção de nomes masculinos/femininos semelhantes a Lesley no tempo.

14.4 Banco de dados de alimentos do USDA

O US Department of Agriculture (Departamento de Agricultura dos Estados Unidos) disponibiliza um banco de dados com informações sobre nutrientes nos alimentos. O programador Ashley Williams disponibilizou uma versão desse banco de dados em formato JSON. Os registros têm a seguinte aparência:

```
{
  "id": 21441,
  "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY,
```

```
  Wing, meat and skin with breading",
    "tags": ["KFC"],
    "manufacturer": "Kentucky Fried Chicken",
    "group": "Fast Foods",
    "portions": [
      {
        "amount": 1,
        "unit": "wing, with skin",
        "grams": 68.0
      },

      ...
    ],
    "nutrients": [
      {
        "value": 20.8,
        "units": "g",
        "description": "Protein",
        "group": "Composition"
      },

      ...
    ]
  }
```

Cada alimento tem uma série de atributos de identificação, junto com duas listas com nutrientes e tamanhos das porções. Os dados nesse formato não favorecem particularmente uma análise, portanto será necessário um pouco de trabalho para manipular os dados deixando-os em um formato mais apropriado.

Depois de fazer download e de extrair os dados do link, você poderá carregá-los em Python usando qualquer biblioteca JSON de sua preferência. Utilizarei o módulo embutido `json` de Python:

```
In [154]: import json

In [155]: db = json.load(open('datasets/usda_food/database.json'))

In [156]: len(db)
Out[156]: 6636
```

Cada entrada em db é um dicionário contendo todos os dados para um único alimento. O campo 'nutrients' é uma lista de dicionários, um para cada nutriente:

```
In [157]: db[0].keys()
Out[157]: dict_keys(['id', 'description', 'tags', 'manufacturer', 'group', 'porti
ons', 'nutrients'])

In [158]: db[0]['nutrients'][0]
Out[158]:
{'description': 'Protein',
 'group': 'Composition',
 'units': 'g',
 'value': 25.18}

In [159]: nutrients = pd.DataFrame(db[0]['nutrients'])

In [160]: nutrients[:7]
Out[160]:
                description group units value
0 Protein Composition g 25.18
1 Total lipid (fat) Composition g 29.20
2 Carbohydrate, by difference Composition g 3.06
3 Ash Other g 3.28
4 Energy Energy kcal 376.00
5 Water Composition g 39.28
6 Energy Energy kJ 1573.00
```

Ao converter uma lista de dicionários em um DataFrame, podemos especificar uma lista de campos a serem extraídos. Usaremos os nomes dos alimentos, o grupo, o ID e o fabricante:

```
In [161]: info_keys = ['description', 'group', 'id', 'manufacturer']

In [162]: info = pd.DataFrame(db, columns=info_keys)

In [163]: info[:5]
Out[163]:
                        description group id \
0 Cheese, caraway Dairy and Egg Products 1008
1 Cheese, cheddar Dairy and Egg Products 1009
2 Cheese, edam Dairy and Egg Products 1018
```

```
3 Cheese, feta Dairy and Egg Products 1019
4 Cheese, mozzarella, part skim milk Dairy and Egg Products 1028
  manufacturer
0
1
2
3
4

In [164]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
description 6636 non-null object
group 6636 non-null object
id 6636 non-null int64
manufacturer 5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB
```

Podemos ver a distribuição dos grupos de alimentos com value_counts:

```
In [165]: pd.value_counts(info.group)[:10]
Out[165]:
Vegetables and Vegetable Products 812
Beef Products 618
Baked Products 496
Breakfast Cereals 403
Legumes and Legume Products 365
Fast Foods 365
Lamb, Veal, and Game Products 345
Sweets 341
Pork Products 328
Fruits and Fruit Juices 328
Name: group, dtype: int64
```

Para fazer uma análise em todos os dados de nutrientes, será mais fácil reunir os nutrientes de cada alimento em uma única tabela grande. Para isso, precisaremos executar vários passos. Inicialmente converterei cada lista de nutrientes dos alimentos em um DataFrame, adicionarei uma coluna para o id do alimento e

concatenarei o DataFrame em uma lista. Em seguida, esses dados poderão ser concatenados com concat.

Se tudo correr bem, nutrients deverá ter o seguinte aspecto:

```
In [167]: nutrients
Out[167]:
                                    description group units value id
0 Protein Composition g 25.180 1008
1 Total lipid (fat) Composition g 29.200 1008
2 Carbohydrate, by difference Composition g 3.060 1008
3 Ash Other g 3.280 1008
4 Energy Energy kcal 376.000 1008
... ... ... ... ... ...
389350 Vitamin B-12, added Vitamins mcg 0.000 43546
389351 Cholesterol Other mg 0.000 43546
389352 Fatty acids, total saturated Other g 0.072 43546
389353 Fatty acids, total monounsaturated Other g 0.028 43546
389354 Fatty acids, total polyunsaturated Other g 0.041 43546
[389355 rows x 5 columns]
```

Percebi que há duplicatas nesse DataFrame, portanto será mais fácil se nós as descartarmos:

```
In [168]: nutrients.duplicated().sum() # número de duplicatas
Out[168]: 14179

In [169]: nutrients = nutrients.drop_duplicates()
```

Como 'group' e 'description' estão nos dois objetos DataFrame, podemos renomeá-los por questões de clareza:

```
In [170]: col_mapping = {'description' : 'food',
   .....: 'group' : 'fgroup'}

In [171]: info = info.rename(columns=col_mapping, copy=False)

In [172]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
food 6636 non-null object
fgroup 6636 non-null object
```

```
id 6636 non-null int64
manufacturer 5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB

In [173]: col_mapping = {'description' : 'nutrient',
   .....: 'group' : 'nutgroup'}

In [174]: nutrients = nutrients.rename(columns=col_mapping, copy=False)

In [175]: nutrients
Out[175]:
                                nutrient nutgroup units value id
0 Protein Composition g 25.180 1008
1 Total lipid (fat) Composition g 29.200 1008
2 Carbohydrate, by difference Composition g 3.060 1008
3 Ash Other g 3.280 1008
4 Energy Energy kcal 376.000 1008
... ... ... ... ... ...
389350 Vitamin B-12, added Vitamins mcg 0.000 43546
389351 Cholesterol Other mg 0.000 43546
389352 Fatty acids, total saturated Other g 0.072 43546
389353 Fatty acids, total monounsaturated Other g 0.028 43546
389354 Fatty acids, total polyunsaturated Other g 0.041 43546
[375176 rows x 5 columns]
```

Depois de fazer tudo isso, estamos prontos para um merge de info com nutrients:

```
In [176]: ndata = pd.merge(nutrients, info, on='id', how='outer')

In [177]: ndata.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
Data columns (total 8 columns):
nutrient 375176 non-null object
nutgroup 375176 non-null object
units 375176 non-null object
value 375176 non-null float64
id 375176 non-null int64
food 375176 non-null object
fgroup 375176 non-null object
```

```
manufacturer 293054 non-null object
dtypes: float64(1), int64(1), object(6)
memory usage: 25.8+ MB

In [178]: ndata.iloc[30000]
Out[178]:
nutrient Glycine
nutgroup Amino Acids
units g
value 0.04
id 6158
food Soup, tomato bisque, canned, condensed
fgroup Soups, Sauces, and Gravies
manufacturer
Name: 30000, dtype: object
```

Podemos agora fazer uma plotagem dos valores das medianas por grupo de alimentos e tipo de nutriente (veja a Figura 14.11):

```
In [180]: result = ndata.groupby(['nutrient', 'fgroup'])['value'].quantile(0.5)

In [181]: result['Zinc, Zn'].sort_values().plot(kind='barh')
```

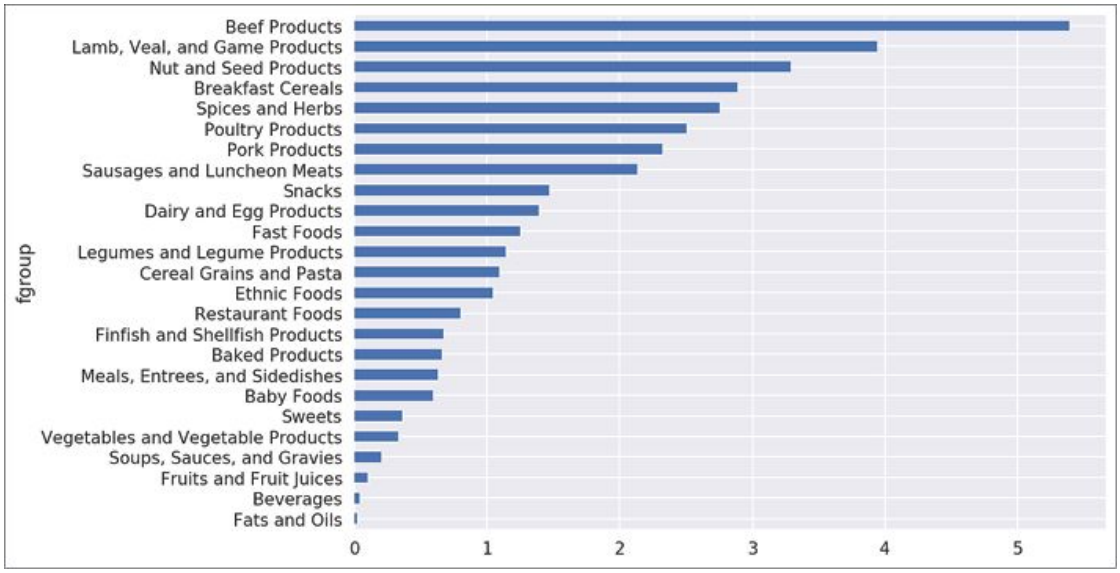


Figura 14.11 – Valores da mediana de zinco por grupo de nutrientes.

Com um pouco de esperteza, podemos descobrir quais alimentos são mais ricos em cada nutriente:

```
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])
```

```
get_maximum = lambda x: x.loc[x.value.idxmax()]
get_minimum = lambda x: x.loc[x.value.idxmin()]

max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]

# deixa o alimento um pouco menor
max_foods.food = max_foods.food.str[:50]
```

O DataFrame resultante é um pouco grande demais para ser exibido no livro; eis apenas o grupo do nutriente 'Amino Acids':

```
In [183]: max_foods.loc['Amino Acids']['food']
Out[183]:
nutrient
Alanine Gelatins, dry powder, unsweetened
Arginine Seeds, sesame flour, low-fat
Aspartic acid Soy protein isolate
Cystine Seeds, cottonseed flour, low fat (glandless)
Glutamic acid Soy protein isolate
                                ...
Serine Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Threonine Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Tryptophan Sea lion, Steller, meat with fat (Alaska Native)
Tyrosine Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Valine Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Name: food, Length: 19, dtype: object
```

14.5 Banco de dados da Federal Election Commission em 2012

A Federal Election Commission (Comissão Eleitoral Federal) dos Estados Unidos publica dados sobre contribuições para campanhas políticas. Esses dados incluem os nomes dos colaboradores, a profissão e o empregador, o endereço e o valor da contribuição. Um conjunto de dados interessante é aquele da eleição presidencial norte-americana de 2012. Uma versão do conjunto de dados que baixei em junho de 2012 contém um arquivo CSV *P00000001-ALL.csv* de 150 megabytes (veja o repositório de dados do livro), que pode ser carregado com pandas.read_csv:

```
In [184]: fec = pd.read_csv('datasets/fec/P00000001-ALL.csv')

In [185]: fec.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001731 entries, 0 to 1001730
Data columns (total 16 columns):
cmte_id 1001731 non-null object
cand_id 1001731 non-null object
cand_nm 1001731 non-null object
contbr_nm 1001731 non-null object
contbr_city 1001712 non-null object
contbr_st 1001727 non-null object
contbr_zip 1001620 non-null object
contbr_employer 988002 non-null object
contbr_occupation 993301 non-null object
contb_receipt_amt 1001731 non-null float64
contb_receipt_dt 1001731 non-null object
receipt_desc 14166 non-null object
memo_cd 92482 non-null object
memo_text 97770 non-null object
form_tp 1001731 non-null object
file_num 1001731 non-null int64
dtypes: float64(1), int64(1), object(14)
memory usage: 122.3+ MB
```

Um exemplo de registro do DataFrame tem o seguinte aspecto:

```
In [186]: fec.iloc[123456]
Out[186]:
cmte_id C00431445
cand_id P80003338
cand_nm Obama, Barack
contbr_nm ELLMAN, IRA
contbr_city TEMPE
                  ...
receipt_desc NaN
memo_cd NaN
memo_text NaN
form_tp SA17A
file_num 772372
Name: 123456, Length: 16, dtype: object
```

Você pode pensar em algumas maneiras de começar a manipular esses dados a fim de extrair estatísticas informativas sobre doadores e padrões nas contribuições de campanha. Mostrarei a você uma série de análises diferentes que aplicam técnicas apresentadas neste livro.

Podemos ver que não há filiações político-partidárias nos dados, portanto seria conveniente acrescentar essa informação. Uma lista com todos os candidatos políticos únicos pode ser obtida com unique:

```
In [187]: unique_cands = fec.cand_nm.unique()

In [188]: unique_cands
Out[188]:
array(['Bachmann, Michelle', 'Romney, Mitt', 'Obama, Barack',
       "Roemer, Charles E. 'Buddy' III", 'Pawlenty, Timothy',
       'Johnson, Gary Earl', 'Paul, Ron', 'Santorum, Rick', 'Cain, Herman',
       'Gingrich, Newt', 'McCotter, Thaddeus G', 'Huntsman, Jon',
       'Perry, Rick'], dtype=object)

In [189]: unique_cands[2]
Out[189]: 'Obama, Barack'
```

Uma forma de indicar a filiação partidária é usar um dicionário:[1]

```
parties = {'Bachmann, Michelle': 'Republican',
           'Cain, Herman': 'Republican',
           'Gingrich, Newt': 'Republican',
           'Huntsman, Jon': 'Republican',
           'Johnson, Gary Earl': 'Republican',
           'McCotter, Thaddeus G': 'Republican',
           'Obama, Barack': 'Democrat',
           'Paul, Ron': 'Republican',
           'Pawlenty, Timothy': 'Republican',
           'Perry, Rick': 'Republican',
           "Roemer, Charles E. 'Buddy' III": 'Republican',
           'Romney, Mitt': 'Republican',
           'Santorum, Rick': 'Republican'}
```

Usando agora esse mapeamento e o método map de objetos Series, podemos gerar um array de partidos políticos a partir dos nomes

dos candidatos:

```
In [191]: fec.cand_nm[123456:123461]
Out[191]:
123456 Obama, Barack
123457 Obama, Barack
123458 Obama, Barack
123459 Obama, Barack
123460 Obama, Barack
Name: cand_nm, dtype: object

In [192]: fec.cand_nm[123456:123461].map(parties)
Out[192]:
123456 Democrat
123457 Democrat
123458 Democrat
123459 Democrat
123460 Democrat
Name: cand_nm, dtype: object

# Adiciona-o como uma coluna
In [193]: fec['party'] = fec.cand_nm.map(parties)

In [194]: fec['party'].value_counts()
Out[194]:
Democrat 593746
Republican 407985
Name: party, dtype: int64
```

Eis alguns aspectos quanto à preparação dos dados. Inicialmente, esses dados incluem tanto as contribuições quanto as restituições (valores negativos de contribuição):

```
In [195]: (fec.contb_receipt_amt > 0).value_counts()
Out[195]:
True 991475
False 10256
Name: contb_receipt_amt, dtype: int64
```

Para simplificar a análise, restringirei o conjunto de dados às contribuições positivas:

```
In [196]: fec = fec[fec.contb_receipt_amt > 0]
```

Como Barack Obama e Mitt Romney eram os dois principais candidatos, também prepararei um subconjunto com as contribuições somente para as suas campanhas:

```
In [197]: fec_mrbo = fec[fec.cand_nm.isin(['Obama, Barack', 'Romney, Mitt'])]
```

Estatísticas sobre as doações de acordo com a profissão e o empregador

As doações por profissão são outra estatística estudada com frequência. Por exemplo, advogados tendem a doar mais dinheiro aos democratas, enquanto executivos de negócios tendem a doar mais aos republicanos. Você não tem nenhum motivo para acreditar em mim; você mesmo pode ver isso analisando os dados. Em primeiro lugar, é fácil obter o número total de doações por profissão:

```
In [198]: fec.contbr_occupation.value_counts()[:10]
Out[198]:
RETIRED 233990
INFORMATION REQUESTED 35107
ATTORNEY 34286
HOMEMAKER 29931
PHYSICIAN 23432
INFORMATION REQUESTED PER BEST EFFORTS 21138
ENGINEER 14334
TEACHER 13990
CONSULTANT 13273
PROFESSOR 12555
Name: contbr_occupation, dtype: int64
```

Observando as profissões, você perceberá que muitas se referem ao mesmo tipo básico de trabalho, ou que há diversas variantes da mesma profissão. O trecho de código a seguir mostra uma técnica para limpar algumas delas fazendo um mapeamento de uma profissão para outra; observe o “truque” de usar dict.get, permitindo que as profissões sem mapeamento “passem direto”:

```
occ_mapping = {
   'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
   'INFORMATION REQUESTED' : 'NOT PROVIDED',
```

```
  'INFORMATION REQUESTED (BEST EFFORTS)' : 'NOT PROVIDED',
  'C.E.O.': 'CEO'
}

# Se não houver nenhum mapeamento disponível, devolve x
f = lambda x: occ_mapping.get(x, x)
fec.contbr_occupation = fec.contbr_occupation.map(f)
```

Farei o mesmo também para os empregadores:

```
emp_mapping = {
  'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
  'INFORMATION REQUESTED' : 'NOT PROVIDED',
  'SELF' : 'SELF-EMPLOYED',
  'SELF EMPLOYED' : 'SELF-EMPLOYED',
}

# Se não houver nenhum mapeamento disponível, devolve x
f = lambda x: emp_mapping.get(x, x)
fec.contbr_employer = fec.contbr_employer.map(f)
```

Agora podemos usar pivot_table para agregar os dados por partido e por profissão e então filtrar a fim de obter o subconjunto daqueles que doaram no mínimo 2 milhões de dólares de modo geral:

```
In [201]: by_occupation = fec.pivot_table('contb_receipt_amt',
   .....: index='contbr_occupation',
   .....: columns='party', aggfunc='sum')

In [202]: over_2mm = by_occupation[by_occupation.sum(1) > 2000000]

In [203]: over_2mm
Out[203]:
party Democrat Republican
contbr_occupation
ATTORNEY 11141982.97 7.477194e+06
CEO 2074974.79 4.211041e+06
CONSULTANT 2459912.71 2.544725e+06
ENGINEER 951525.55 1.818374e+06
EXECUTIVE 1355161.05 4.138850e+06
... ... ...
PRESIDENT 1878509.95 4.720924e+06
PROFESSOR 2165071.08 2.967027e+05
```

```
REAL ESTATE 528902.09 1.625902e+06
RETIRED 25305116.38 2.356124e+07
SELF-EMPLOYED 672393.40 1.640253e+06
[17 rows x 2 columns]
```

Talvez seja mais fácil observar esses dados graficamente, na forma de uma plotagem em barras ('barh' quer dizer plotagem em barras horizontais; veja a Figura 14.12):

```
In [205]: over_2mm.plot(kind='barh')
```

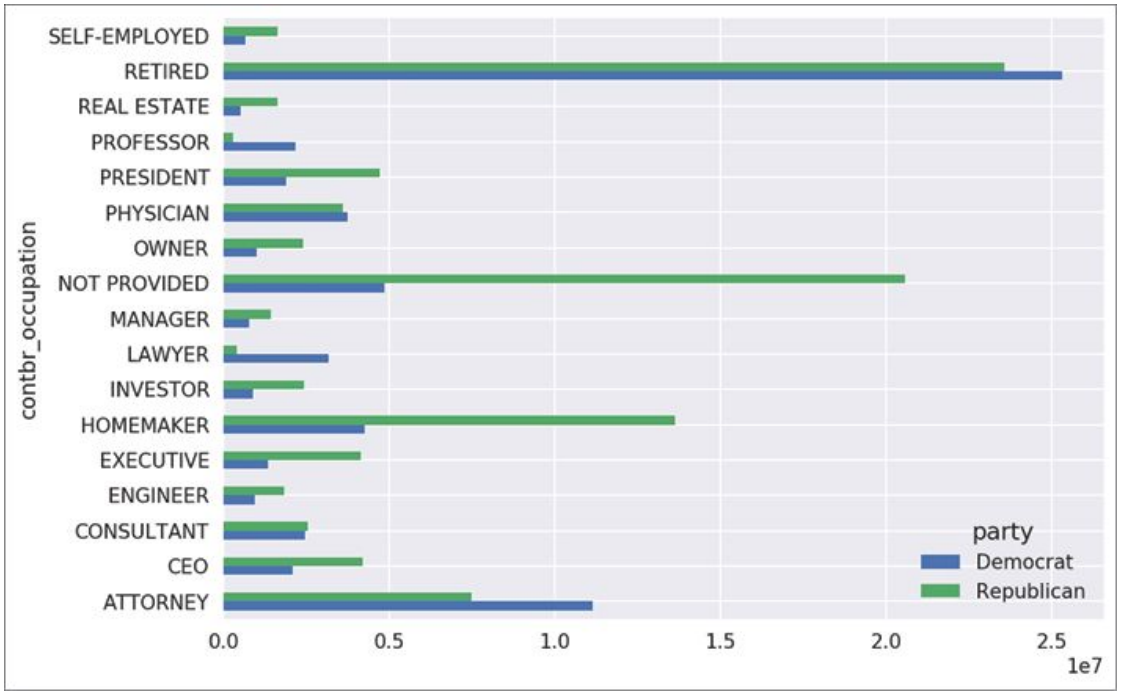


Figura 14.12 – Total de doações por partido para as principais profissões.

Você poderia estar interessado nas profissões dos principais doadores ou nas principais empresas que fizeram doações para Obama e para Romney. Para isso, poderá agrupar os dados por nome de candidato e usar uma variante do método top que vimos antes no capítulo:

```
def get_top_amounts(group, key, n=5):
    totals = group.groupby(key)['contb_receipt_amt'].sum()
    return totals.nlargest(n)
```

Em seguida, faça uma agregação por profissão e por empregador:

```
In [207]: grouped = fec_mrbo.groupby('cand_nm')
```

```
In [208]: grouped.apply(get_top_amounts, 'contbr_occupation', n=7)
Out[208]:
cand_nm contbr_occupation
Obama, Barack RETIRED 25305116.38
                ATTORNEY 11141982.97
                INFORMATION REQUESTED 4866973.96
                HOMEMAKER 4248875.80
                PHYSICIAN 3735124.94
                                    ...
Romney, Mitt HOMEMAKER 8147446.22
                ATTORNEY 5364718.82
                PRESIDENT 2491244.89
                EXECUTIVE 2300947.03
                C.E.O. 1968386.11
Name: contb_receipt_amt, Length: 14, dtype: float64

In [209]: grouped.apply(get_top_amounts, 'contbr_employer', n=10)
Out[209]:
cand_nm contbr_employer
Obama, Barack RETIRED 22694358.85
                SELF-EMPLOYED 17080985.96
                NOT EMPLOYED 8586308.70
                INFORMATION REQUESTED 5053480.37
                HOMEMAKER 2605408.54
                                    ...
Romney, Mitt CREDIT SUISSE 281150.00
                MORGAN STANLEY 267266.00
                GOLDMAN SACH & CO. 238250.00
                BARCLAYS CAPITAL 162750.00
                H.I.G. CAPITAL 139500.00
Name: contb_receipt_amt, Length: 20, dtype: float64
```

Separando os valores das doações em buckets

Um modo conveniente de analisar esses dados é usar a função cut a fim de tornar discretos os valores das contribuições, separando-os em buckets conforme o montante:

```
In [210]: bins = np.array([0, 1, 10, 100, 1000, 10000,
   .....: 100000, 1000000, 10000000])
```

```
In [211]: labels = pd.cut(fec_mrbo.contb_receipt_amt, bins)

In [212]: labels
Out[212]:
411 (10, 100]
412 (100, 1000]
413 (100, 1000]
414 (10, 100]
415 (10, 100]
           ...
701381 (10, 100]
701382 (100, 1000]
701383 (1, 10]
701384 (10, 100]
701385 (100, 1000]
Name: contb_receipt_amt, Length: 694282, dtype: category
Categories (8, interval[int64]): [(0, 1] < (1, 10] < (10, 100] < (100, 1000] < (1
000, 10000] <
                              (10000, 100000] < (100000, 1000000] < (1000000,
 10000000]]
```

Podemos então agrupar os dados para Obama e Romney por nome e rótulo do compartimento (bin) a fim de obter um histograma de acordo com o montante da doação:

```
In [213]: grouped = fec_mrbo.groupby(['cand_nm', labels])

In [214]: grouped.size().unstack(0)
Out[214]:
cand_nm Obama, Barack Romney, Mitt
contb_receipt_amt
(0, 1] 493.0 77.0
(1, 10] 40070.0 3681.0
(10, 100] 372280.0 31853.0
(100, 1000] 153991.0 43357.0
(1000, 10000] 22284.0 26186.0
(10000, 100000] 2.0 1.0
(100000, 1000000] 3.0 NaN
(1000000, 10000000] 4.0 NaN
```

Esses dados mostram que Obama recebeu um número

significativamente maior de doações menores do que Romney. Também podemos somar os valores das contribuições e normalizar nos buckets a fim de visualizar o percentual sobre o total de doações de cada montante, por candidato (a Figura 14.13 mostra a plotagem resultante):

```
In [216]: bucket_sums = grouped.contb_receipt_amt.sum().unstack(0)

In [217]: normed_sums = bucket_sums.div(bucket_sums.sum(axis=1), axis=0)

In [218]: normed_sums
Out[218]:
cand_nm Obama, Barack Romney, Mitt
contb_receipt_amt
(0, 1] 0.805182 0.194818
(1, 10] 0.918767 0.081233
(10, 100] 0.910769 0.089231
(100, 1000] 0.710176 0.289824
(1000, 10000] 0.447326 0.552674
(10000, 100000] 0.823120 0.176880
(100000, 1000000] 1.000000 NaN
(1000000, 10000000] 1.000000 NaN

In [219]: normed_sums[:-2].plot(kind='barh')
```

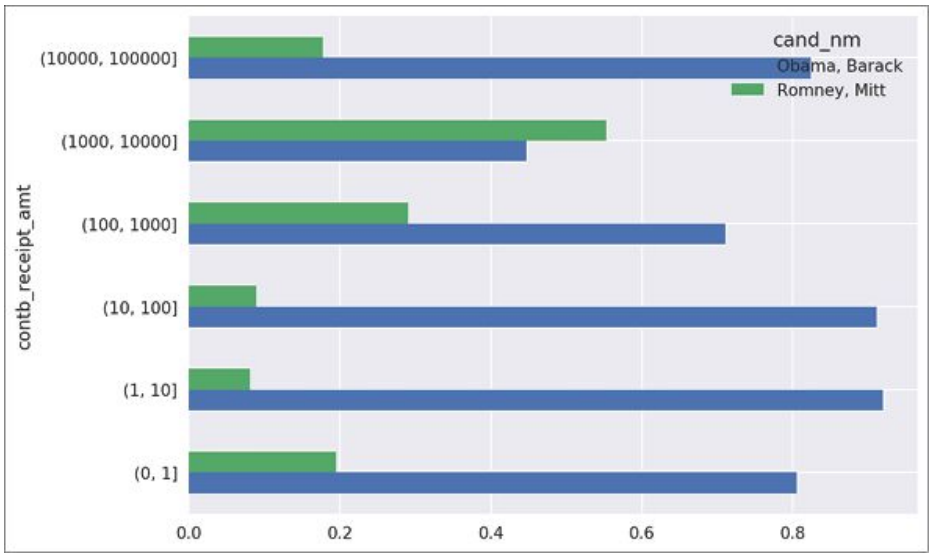


Figura 14.13 – Porcentagem do total de doações recebidas pelos candidatos para cada montante de doação.

Excluí os dois maiores compartimentos, pois não são doações feitas por indivíduos.

Essa análise pode ser refinada e aprimorada de várias maneiras. Por exemplo, poderíamos agregar as doações de acordo com o nome dos doadores e o CEP e ajustar para doadores que contribuíram com várias quantias pequenas *versus* uma ou mais doações maiores. Incentivo você a fazer download e a explorar o conjunto de dados por conta própria.

Estatísticas sobre as doações conforme o estado

Agregar os dados por candidato e por estado é uma tarefa rotineira:

```
In [220]: grouped = fec_mrbo.groupby(['cand_nm', 'contbr_st'])

In [221]: totals = grouped.contb_receipt_amt.sum().unstack(0).fillna(0)

In [222]: totals = totals[totals.sum(1) > 100000]

In [223]: totals[:10]
Out[223]:
cand_nm Obama, Barack Romney, Mitt
contbr_st
AK 281840.15 86204.24
AL 543123.48 527303.51
AR 359247.28 105556.00
AZ 1506476.98 1888436.23
CA 23824984.24 11237636.60
CO 2132429.49 1506714.12
CT 2068291.26 3499475.45
DC 4373538.80 1025137.50
DE 336669.14 82712.00
FL 7318178.58 8338458.81
```

Se você dividir cada linha pelo valor total da contribuição, o percentual relativo do total das doações por estado para cada candidato será obtido:

```
In [224]: percent = totals.div(totals.sum(1), axis=0)
```

```
In [225]: percent[:10]
Out[225]:
cand_nm Obama, Barack Romney, Mitt
contbr_st
AK 0.765778 0.234222
AL 0.507390 0.492610
AR 0.772902 0.227098
AZ 0.443745 0.556255
CA 0.679498 0.320502
CO 0.585970 0.414030
CT 0.371476 0.628524
DC 0.810113 0.189887
DE 0.802776 0.197224
FL 0.467417 0.532583
```

14.6 Conclusão

Chegamos ao final dos principais capítulos do livro. Incluí um conteúdo adicional nos apêndices, que talvez você ache útil.

Nos cinco anos desde que a primeira edição deste livro foi publicada, Python se transformou em uma linguagem popular e difundida para análise de dados. As habilidades de programação que você desenvolveu aqui permanecerão relevantes por muito tempo no futuro. Espero que as ferramentas de programação e as bibliotecas que exploramos sejam bastante úteis a você em seu trabalho.

---

1 Estamos fazendo uma suposição simplificadora segundo a qual Gary Johnson é republicano, embora, posteriormente, ele tenha se tornado candidato pelo Partido Libertário.

APÊNDICE A

NumPy avançado

Neste apêndice, exploraremos melhor a biblioteca NumPy para processamento de arrays. Serão incluídos mais detalhes internos sobre o tipo ndarray, além de manipulações e algoritmos mais sofisticados para arrays.

Este apêndice contém assuntos variados e não precisa ser lido necessariamente de forma linear.

A.1 Organização interna do objeto ndarray

O ndarray do NumPy possibilita uma forma de interpretar um bloco de dados homogêneo (seja contíguo ou em passos) como um objeto array multidimensional. O tipo do dado, ou *dtype*, determina como os dados são interpretados: como números de ponto flutuante, inteiros, booleanos ou qualquer outro tipo que vimos antes.

Parte do que torna o ndarray flexível é o fato de todo objeto array ser uma visão *em passos* (strided) de um bloco de dados. Talvez você esteja se perguntando, por exemplo, como a visão do array `arr[::2, ::-1]` não implica nenhuma cópia de dados. O motivo está no fato de o ndarray ser mais do que simplesmente uma porção de memória e um dtype; ele também contém informações sobre "passos" (striding), que permite que o array percorra a memória com tamanhos variados de passos. De modo mais exato, o ndarray consiste internamente das seguintes informações:

- um *ponteiro para os dados* – isto é, um bloco de dados em RAM ou em um arquivo mapeado em memória;
- o *tipo do dado*, ou dtype, que descreve as células de tamanho fixo

para os valores do array;

- uma tupla que informa o *formato* (shape) do array;
- uma tupla de passos (*strides*), isto é, inteiros que informam o número de bytes para "pular" a fim de avançar um elemento em uma dimensão.

Veja a Figura A.1 que exibe uma representação simples da organização interna de um ndarray.

Por exemplo, um array de 10 × 5 teria um formato (shape) igual a (10, 5):

```
In [10]: np.ones((10, 5)).shape
Out[10]: (10, 5)
```

Um array de 3 × 4 × 5 típico (ordem C) de valores float64 (8 bytes) tem passos de (160, 40, 8) (conhecer os passos pode ser útil, pois, em geral, quanto maiores os passos em determinado eixo, mais custoso será executar processamentos nesse eixo):

```
In [11]: np.ones((3, 4, 5), dtype=np.float64).strides
Out[11]: (160, 40, 8)
```

Embora seja raro um usuário típico do NumPy se interessar pelos passos do array, eles são o ingrediente essencial na construção de visualizações de arrays "sem cópias". Os passos podem ser até mesmo negativos, o que permite que o array ande "para trás" na memória (esse seria, por exemplo, o caso de uma fatia como obj[::-1] ou obj[:, ::-1]).

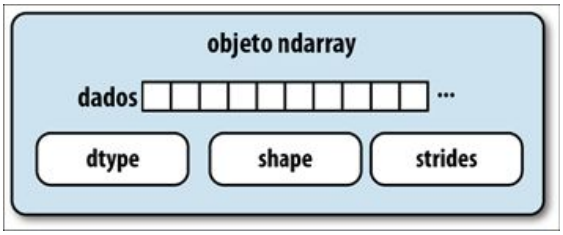


Figura A.1 – O objeto ndarray do NumPy.

A hierarquia de dtypes do NumPy

Ocasionalmente, é possível que você tenha um código que

necessite verificar se um array contém inteiros, números de ponto flutuante, strings ou objetos Python. Considerando que há vários tipos de números de ponto flutuante (de float16 a float128), verificar se o dtype está entre uma lista de tipos seria uma operação extensa. Felizmente os dtypes têm superclasses como np.integer e np.floating, que podem ser usadas em conjunto com a função np.issubdtype:

```
In [12]: ints = np.ones(10, dtype=np.uint16)

In [13]: floats = np.ones(10, dtype=np.float32)

In [14]: np.issubdtype(ints.dtype, np.integer)
Out[14]: True

In [15]: np.issubdtype(floats.dtype, np.floating)
Out[15]: True
```

Todas as classes-pais de um dtype específico podem ser vistas chamando o método mro do tipo:

```
In [16]: np.float64.mro()
Out[16]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```

Desse modo, temos também:

```
In [17]: np.issubdtype(ints.dtype, np.number)
Out[17]: True
```

A maioria dos usuários do NumPy jamais terá que conhecer essas informações; contudo, ocasionalmente, elas poderão ser úteis. Veja a Figura A.2 que mostra um diagrama da hierarquia de dtypes e os relacionamentos entre as classes-pais e as subclasses.[1]

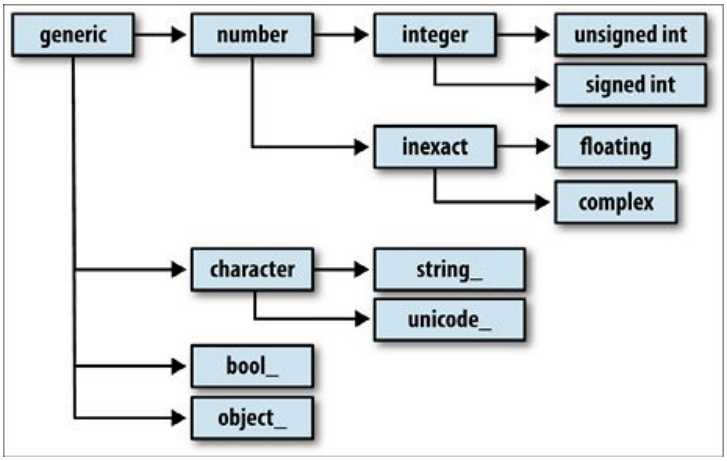


Figura A.2 – A hierarquia de classes dos dtypes do NumPy.

A.2 Manipulação avançada de arrays

Há muitas maneiras de trabalhar com arrays além da indexação sofisticada (fancy indexing), o fatiamento e a geração de subconjuntos booleanos. Embora boa parte do trabalho pesado nas aplicações de análise de dados seja tratada por funções de nível mais alto no pandas, em algum momento, é possível que você precise escrever um algoritmo de dados que não se encontre em nenhuma das bibliotecas existentes.

Redefinindo o formato de arrays

Em muitos casos, podemos converter um array de um formato para outro sem copiar qualquer dado. Para isso, passe uma tupla informando o novo formato para o método de instância reshape do array. Por exemplo, suponha que tivéssemos um array unidimensional de valores e quiséssemos reorganizá-lo na forma de uma matriz (a Figura A.3 mostra o resultado):

```
In [18]: arr = np.arange(8)

In [19]: arr
Out[19]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [20]: arr.reshape((4, 2))
Out[20]:
```

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

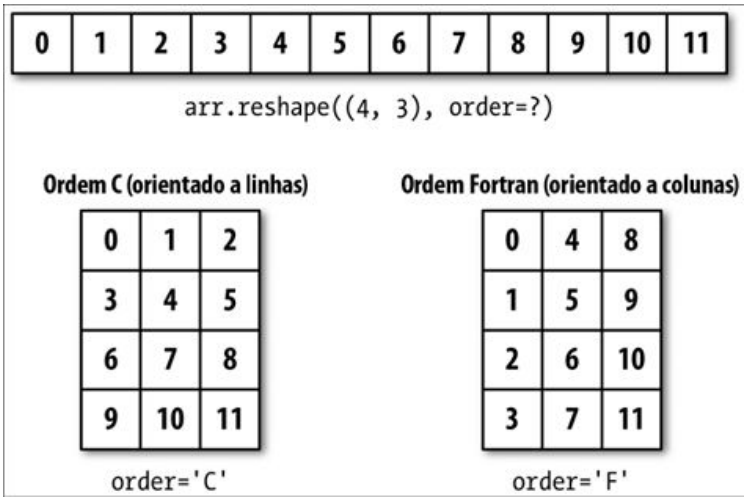


Figura A.3 – Redefinindo o formato na ordem C (orientado a linhas) ou na ordem Fortran (orientado a colunas).

Um array multidimensional também pode ter o formato redefinido:

```
In [21]: arr.reshape((4, 2)).reshape((2, 4))
Out[21]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

Uma das dimensões do formato especificado pode ser –1, caso em que o valor utilizado para essa dimensão será inferido a partir dos dados:

```
In [22]: arr = np.arange(15)

In [23]: arr.reshape((5, -1))
Out[23]:
array([[ 0, 1, 2],
       [ 3, 4, 5],
       [ 6, 7, 8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Como o atributo `shape` de um array é uma tupla, esse pode ser passado para `reshape` também:

```
In [24]: other_arr = np.ones((3, 5))

In [25]: other_arr.shape
Out[25]: (3, 5)

In [26]: arr.reshape(other_arr.shape)
Out[26]:
array([[ 0, 1, 2, 3, 4],
       [ 5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
```

A operação inversa de reshape de um formato unidimensional para um formato de dimensões maiores em geral é conhecida como *linearização* (flattening) ou *raveling*:

```
In [27]: arr = np.arange(15).reshape((5, 3))

In [28]: arr
Out[28]:
array([[ 0, 1, 2],
       [ 3, 4, 5],
       [ 6, 7, 8],
       [ 9, 10, 11],
       [12, 13, 14]])

In [29]: arr.ravel()
Out[29]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

ravel não gerará uma cópia dos valores subjacentes se os valores no resultado forem contíguos no array original. O método flatten comporta-se como ravel, exceto pelo fato de que ele sempre devolverá uma cópia dos dados:

```
In [30]: arr.flatten()
Out[30]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

Os dados podem ter o formato redefinido ou podem ser linearizados em ordens diferentes. Esse é um assunto que apresenta leves nuances para os novos usuários do NumPy e, desse modo, será o nosso próximo tópico.

Ordem C versus ordem Fortran

O NumPy permite que você tenha controle e flexibilidade sobre o layout de seus dados na memória. Por padrão, os arrays NumPy são criados em uma ordem *orientada a linhas* (row major). Do ponto de vista espacial, isso significa que, se você tiver um array de dados bidimensional, os itens em cada linha do array serão armazenados em posições de memória adjacentes. A alternativa à ordem orientada a linhas é a ordem *orientada a colunas* (column major), que significa que os valores em cada coluna de dados são armazenados em posições de memória adjacentes.

Por motivos históricos, as ordens orientadas a linhas e orientadas a colunas também são conhecidas como ordem C e ordem Fortran, respectivamente. Na linguagem FORTRAN 77, todas as matrizes são orientadas a colunas.

Funções como `reshape` e `ravel` aceitam um argumento `order` que especifica a ordem em que os dados do array serão usados. Geralmente ele é definido com `'C'` ou com `'F'` na maioria dos casos (há também as opções `'A'` e `'K`, menos usuais; consulte a documentação do NumPy e observe novamente a Figura A.3 que ilustra essas opções):

```
In [31]: arr = np.arange(12).reshape((3, 4))

In [32]: arr
Out[32]:
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])

In [33]: arr.ravel()
Out[33]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

In [34]: arr.ravel('F')
Out[34]: array([ 0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11])
```

Redefinir o formato de arrays com mais de duas dimensões pode ser um pouco difícil de entender (veja a Figura A.3). A principal diferença entre a ordem C e a ordem Fortran está no modo como as

dimensões são percorridas:

Ordem C/orientada a linhas (row major)
: As dimensões mais altas são percorridas *antes* (por exemplo, o eixo 1 antes de passar para o eixo 0).

Ordem C/orientada a colunas (column major)
: As dimensões mais altas são percorridas *depois* (por exemplo, o eixo 0 antes de passar para o eixo 1).

Concatenando e separando arrays

numpy.concatenate recebe uma sequência (tupla, lista etc.) de arrays e faz a sua junção na ordem, ao longo do eixo de entrada:

```
In [35]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])

In [36]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])

In [37]: np.concatenate([arr1, arr2], axis=0)
Out[37]:
array([[ 1, 2, 3],
       [ 4, 5, 6],
       [ 7, 8, 9],
       [10, 11, 12]])

In [38]: np.concatenate([arr1, arr2], axis=1)
Out[38]:
array([[ 1, 2, 3, 7, 8, 9],
       [ 4, 5, 6, 10, 11, 12]])
```

Há algumas funções convenientes, como vstack e hstack, para os tipos comuns de concatenação. As operações anteriores poderiam ter sido expressas assim:

```
In [39]: np.vstack((arr1, arr2))
Out[39]:
array([[ 1, 2, 3],
       [ 4, 5, 6],
       [ 7, 8, 9],
       [10, 11, 12]])
```

```
In [40]: np.hstack((arr1, arr2))
Out[40]:
array([[ 1, 2, 3, 7, 8, 9],
       [ 4, 5, 6, 10, 11, 12]])
```

split, por outro lado, fatia um array em vários ao longo de um eixo:

```
In [41]: arr = np.random.randn(5, 2)

In [42]: arr
Out[42]:
array([[-0.2047, 0.4789],
       [-0.5194, -0.5557],
       [ 1.9658, 1.3934],
       [ 0.0929, 0.2817],
       [ 0.769 , 1.2464]])

In [43]: first, second, third = np.split(arr, [1, 3])

In [44]: first
Out[44]: array([[-0.2047, 0.4789]])

In [45]: second
Out[45]:
array([[-0.5194, -0.5557],
       [ 1.9658, 1.3934]])

In [46]: third
Out[46]:
array([[ 0.0929, 0.2817],
       [ 0.769 , 1.2464]])
```

O valor [1, 3] passado para np.split informa os índices nos quais o array será separado em partes.

Veja a Tabela A.1 que contém uma lista de todas as funções de concatenação e de separação relevantes, algumas das quais são oferecidas somente como alternativas convenientes à função de propósito geral concatenate.

Tabela A.1 – Funções para concatenação de arrays

Função	Descrição
concatenate	Função de caráter geral, concatena coleções de arrays ao longo de um eixo
vstack, row_stack	Empilha arrays ao longo das linhas (eixo 0)
hstack	Empilha arrays ao longo das colunas (eixo 1)
column_stack	Como hstack, porém converte arrays 1D para vetores de colunas 2D antes
dstack	Empilha array em “profundidade” (ao longo do eixo 2)
split	Separa o array nos locais especificados ao longo de um eixo em particular
hsplit/vsplit	Funções convenientes para separação no eixo 0 e no eixo 1, respectivamente

Auxiliares para empilhamento: r_ e c_

Há dois objetos especiais no namespace do NumPy, `r_` e `c_`, que deixam o empilhamento de arrays mais conciso:

```
In [47]: arr = np.arange(6)

In [48]: arr1 = arr.reshape((3, 2))

In [49]: arr2 = np.random.randn(3, 2)

In [50]: np.r_[arr1, arr2]
Out[50]:
array([[ 0. , 1. ],
       [ 2. , 3. ],
       [ 4. , 5. ],
       [ 1.0072, -1.2962],
       [ 0.275 , 0.2289],
       [ 1.3529, 0.8864]])

In [51]: np.c_[np.r_[arr1, arr2], arr]
Out[51]:
array([[ 0. , 1. , 0. ],
       [ 2. , 3. , 1. ],
       [ 4. , 5. , 2. ],
       [ 1.0072, -1.2962, 3. ],
```

```
       [ 0.275 , 0.2289, 4. ],
       [ 1.3529, 0.8864, 5. ]])
```

Além do mais, eles podem traduzir fatias em arrays:

```
In [52]: np.c_[1:6, -10:-5]
Out[52]:
array([[ 1, -10],
       [ 2, -9],
       [ 3, -8],
       [ 4, -7],
       [ 5, -6]])
```

Consulte a docstring para saber mais sobre o que pode ser feito com `r_` e `c_`.

Repetindo elementos: tile e repeat

Duas ferramentas úteis para repetir ou replicar arrays a fim de gerar arrays maiores são as funções `repeat` e `tile`. A função `repeat` replica cada elemento de um array determinado número de vezes, gerando um array maior:

```
In [53]: arr = np.arange(3)

In [54]: arr
Out[54]: array([0, 1, 2])

In [55]: arr.repeat(3)
Out[55]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

A necessidade de replicar ou repetir arrays talvez seja menos comum com o NumPy do que com outros frameworks de programação para arrays, como o MATLAB. Um dos motivos para isso é que o *broadcasting* em geral atende melhor a essa necessidade; esse será o assunto da próxima seção.

Por padrão, se você passar um inteiro, cada elemento será repetido esse número de vezes. Se você passar um array de inteiros, cada elemento será repetido um número diferente de vezes:

```
In [56]: arr.repeat([2, 3, 4])
Out[56]: array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

Arrays multidimensionais podem ter seus elementos repetidos ao longo de um eixo em particular.

```
In [57]: arr = np.random.randn(2, 2)

In [58]: arr
Out[58]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [59]: arr.repeat(2, axis=0)
Out[59]:
array([[-2.0016, -0.3718],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386]])
```

Observe que, se nenhum eixo for especificado, o array será linearizado antes, o que, provavelmente, não é o que você quer. De modo semelhante, podemos passar um array de inteiros ao repetir um array multidimensional, de modo a repetir uma dada fatia um número diferente de vezes:

```
In [60]: arr.repeat([2, 3], axis=0)
Out[60]:
array([[-2.0016, -0.3718],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386]])

In [61]: arr.repeat([2, 3], axis=1)
Out[61]:
array([[-2.0016, -2.0016, -0.3718, -0.3718, -0.3718],
       [ 1.669 , 1.669 , -0.4386, -0.4386, -0.4386]])
```

tile, por outro lado, é um atalho para empilhar cópias de um array ao longo de um eixo. Visualmente podemos pensar nele como semelhante a “assentar azulejos”:

```
In [62]: arr
Out[62]:
```

```
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [63]: np.tile(arr, 2)
Out[63]:
array([[-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386, 1.669 , -0.4386]])
```

O segundo argumento é o número de repetições; com um escalar, a repetição é feita linha a linha em vez de ser efetuada coluna a coluna. O segundo argumento de tile pode ser uma tupla que informa o layout da "repetição":

```
In [64]: arr
Out[64]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [65]: np.tile(arr, (2, 1))
Out[65]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [66]: np.tile(arr, (3, 2))
Out[66]:
array([[-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386, 1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386, 1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386, 1.669 , -0.4386]])
```

Equivalentes à indexação sofisticada: take e put

Conforme você deve se lembrar do que vimos no Capítulo 4, uma maneira de obter e de definir subconjuntos de arrays é usar a *indexação sofisticada* (fancy indexing) com arrays de inteiros:

```
In [67]: arr = np.arange(10) * 100
```

```
In [68]: inds = [7, 1, 2, 6]

In [69]: arr[inds]
Out[69]: array([700, 100, 200, 600])
```

Há métodos alternativos de ndarray que são úteis no caso especial de fazer uma seleção somente em um único eixo:

```
In [70]: arr.take(inds)
Out[70]: array([700, 100, 200, 600])

In [71]: arr.put(inds, 42)

In [72]: arr
Out[72]: array([ 0, 42, 42, 300, 400, 500, 42, 42, 800, 900])

In [73]: arr.put(inds, [40, 41, 42, 43])

In [74]: arr
Out[74]: array([ 0, 41, 42, 300, 400, 500, 43, 40, 800, 900])
```

Para usar `take` em outros eixos, podemos passar o argumento nomeado `axis`:

```
In [75]: inds = [2, 0, 2, 1]

In [76]: arr = np.random.randn(2, 4)

In [77]: arr
Out[77]:
array([[-0.5397, 0.477 , 3.2489, -1.0212],
       [-0.5771, 0.1241, 0.3026, 0.5238]])

In [78]: arr.take(inds, axis=1)
Out[78]:
array([[ 3.2489, -0.5397, 3.2489, 0.477 ],
       [ 0.3026, -0.5771, 0.3026, 0.1241]])
```

`put` não aceita um argumento `axis`, mas usa índices da versão linearizada (unidimensional, em ordem C) do array. Assim, se houver necessidade de definir elementos usando um array de índices em outros eixos, em geral será mais fácil utilizar a indexação sofisticada.

A.3 Broadcasting

O *broadcasting* descreve o funcionamento da aritmética entre arrays de formatos distintos. Pode ser um recurso eficaz, mas talvez cause confusão, mesmo para os usuários experientes. O exemplo mais simples de broadcasting ocorre quando combinamos um valor escalar com um array:

```
In [79]: arr = np.arange(5)

In [80]: arr
Out[80]: array([0, 1, 2, 3, 4])

In [81]: arr * 4
Out[81]: array([ 0, 4, 8, 12, 16])
```

Nesse caso, dizemos que fizemos o *broadcast* do valor escalar 4 para todos os outros elementos na operação de multiplicação.

Por exemplo, podemos fazer a subtração da média de cada coluna de um array. Nesse caso, é uma operação bem simples:

```
In [82]: arr = np.random.randn(4, 3)

In [83]: arr.mean(0)
Out[83]: array([-0.3928, -0.3824, -0.8768])

In [84]: demeaned = arr - arr.mean(0)

In [85]: demeaned
Out[85]:
array([[ 0.3937, 1.7263, 0.1633],
       [-0.4384, -1.9878, -0.9839],
       [-0.468 , 0.9426, -0.3891],
       [ 0.5126, -0.6811, 1.2097]])

In [86]: demeaned.mean(0)
Out[86]: array([-0., 0., -0.])
```

Veja a Figura A.4 que ilustra essa operação. Subtrair a média das linhas como uma operação de broadcast exige um pouco mais de cuidado. Felizmente fazer o broadcasting de valores de dimensões

possivelmente menores em qualquer dimensão de um array (por exemplo, subtrair as médias das linhas de cada coluna de um array bidimensional) é possível, desde que você siga as regras.

Isso nos leva ao seguinte:

A regra de broadcasting

Dois arrays serão compatíveis para broadcasting se, para cada *dimensão final* (isto é, começando pelo final), os tamanhos dos eixos coincidirem ou se um dos tamanhos for 1. O broadcasting será então efetuado nas dimensões ausentes ou de tamanho 1.

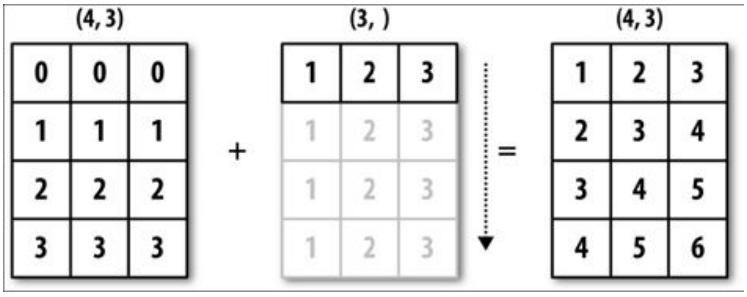


Figura A.4 – Broadcasting no eixo 0 com um array 1D.

Mesmo como um usuário experiente do NumPy, com frequência me vejo tendo que fazer uma pausa e desenhando um diagrama enquanto penso na regra de broadcasting. Considere o último exemplo e suponha que quiséssemos, por outro lado, subtrair o valor da média de cada linha. Como `arr.mean(0)` tem tamanho 3, ele é compatível para broadcasting no eixo 0, pois a dimensão final em `arr` é 3, e, desse modo, há uma coincidência. De acordo com as regras, para fazer uma subtração no eixo 1 (isto é, subtrair a média da linha de cada linha), o array menor deve ter a dimensão `(4, 1)`:

```
In [87]: arr
Out[87]:
array([[ 0.0009, 1.3438, -0.7135],
       [-0.8312, -2.3702, -1.8608],
       [-0.8608, 0.5601, -1.2659],
       [ 0.1198, -1.0635, 0.3329]])
```

```
In [88]: row_means = arr.mean(1)

In [89]: row_means.shape
Out[89]: (4,)

In [90]: row_means.reshape((4, 1))
Out[90]:
array([[ 0.2104],
       [-1.6874],
       [-0.5222],
       [-0.2036]])

In [91]: demeaned = arr - row_means.reshape((4, 1))

In [92]: demeaned.mean(1)
Out[92]: array([ 0., -0., 0., 0.])
```

Veja a Figura A.5 que ilustra essa operação.

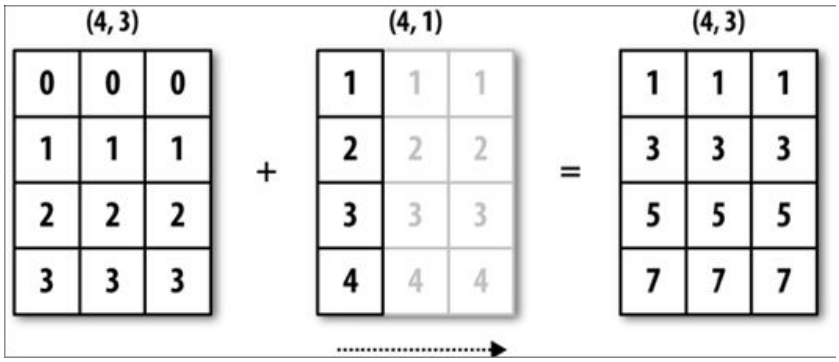


Figura A.5 – Broadcasting no eixo 1 de um array 2D.

Veja a Figura A.6 que apresenta outra ilustração, dessa vez somando um array bidimensional com um array tridimensional no eixo 0.

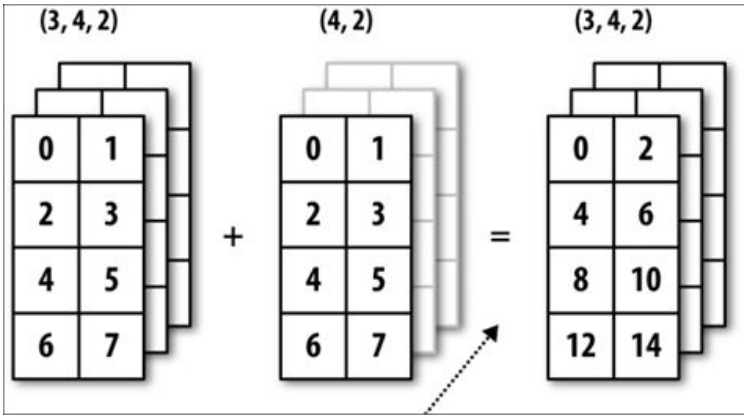


Figura A.6 – Broadcasting no eixo 0 de um array 3D.

Broadcasting em outros eixos

O broadcasting com arrays de dimensões maiores pode parecer mais complicado ainda, mas, na verdade, é uma questão de seguir as regras. Se não as seguir, você obterá um erro como este:

```
In [93]: arr - arr.mean(1)
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)
<ipython-input-93-8b8ada26fac0> in <module>()
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```

É muito comum querer fazer uma operação aritmética com um array de dimensões menores em eixos diferentes do eixo 0. De acordo com a regra de broadcasting, as “dimensões do broadcast” devem ser 1 no array menor. No exemplo da subtração da média mostrado nesta seção, isso significou redimensionar as médias das linhas para o formato `(4, 1)` em vez de `(4,)`:

```
In [94]: arr - arr.mean(1).reshape((4, 1))
Out[94]:
array([[-0.2095, 1.1334, -0.9239],
       [ 0.8562, -0.6828, -0.1734],
       [-0.3386, 1.0823, -0.7438],
       [ 0.3234, -0.8599, 0.5365]])
```

No caso tridimensional, o broadcasting em qualquer uma das três dimensões é apenas uma questão de redefinir o formato dos dados

para que haja compatibilidade. A Figura A.7 apresenta uma boa visualização dos formatos necessários para fazer um broadcast em cada eixo de um array tridimensional.

Assim, um problema comum é a necessidade de adicionar um novo eixo de tamanho 1 especificamente com o intuito de fazer um broadcasting. Usar reshape é uma opção, mas inserir um eixo exige a construção de uma tupla indicando o novo formato. Com frequência, esse pode ser um exercício maçante. Desse modo, os arrays NumPy oferecem uma sintaxe especial para inserir novos eixos por indexação. Usamos o atributo especial np.newaxis, junto com fatias "completas" para inserir o novo eixo:

```
In [95]: arr = np.zeros((4, 4))

In [96]: arr_3d = arr[:, np.newaxis, :]

In [97]: arr_3d.shape
Out[97]: (4, 1, 4)

In [98]: arr_1d = np.random.normal(size=3)

In [99]: arr_1d[:, np.newaxis]
Out[99]:
array([[-2.3594],
       [-0.1995],
       [-1.542 ]])

In [100]: arr_1d[np.newaxis, :]
Out[100]: array([[-2.3594, -0.1995, -1.542 ]])
```

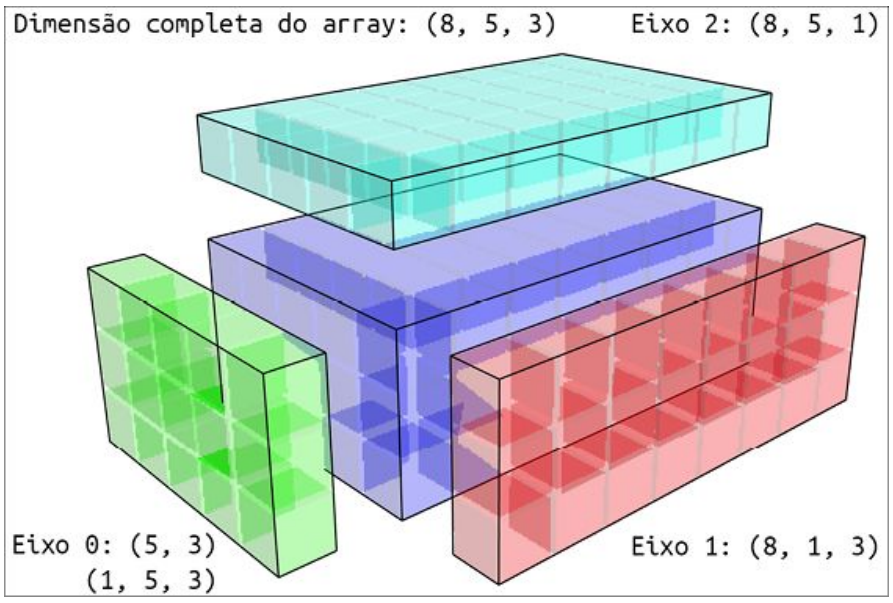


Figura A.7 – Formatos de arrays 2D compatíveis para broadcasting em um array 3D.

Assim, se tivéssemos um array tridimensional e quiséssemos subtrair a média do eixo 2, por exemplo, teríamos que escrever o código a seguir:

```
In [101]: arr = np.random.randn(3, 4, 5)

In [102]: depth_means = arr.mean(2)

In [103]: depth_means
Out[103]:
array([[-0.4735, 0.3971, -0.0228, 0.2001],
       [-0.3521, -0.281 , -0.071 , -0.1586],
       [ 0.6245, 0.6047, 0.4396, -0.2846]])

In [104]: depth_means.shape
Out[104]: (3, 4)

In [105]: demeaned = arr - depth_means[:, :, np.newaxis]

In [106]: demeaned.mean(2)
Out[106]:
array([[ 0., 0., -0., -0.],
       [ 0., 0., -0., 0.],
       [ 0., 0., -0., -0.]])
```

Você deve estar se perguntando se não há uma maneira de generalizar a subtração da média de um eixo sem sacrificar o desempenho. Há uma opção, mas ela exige um pouco de ginástica na indexação:

```
def demean_axis(arr, axis=0):
    means = arr.mean(axis)

    # Generaliza algo como [:, :, np.newaxis] para N dimensões
    indexer = [slice(None)] * arr.ndim
    indexer[axis] = np.newaxis
    return arr - means[indexer]
```

Definindo valores de array para broadcasting

A mesma regra de broadcasting que orienta as operações aritméticas também se aplica à definição de valores com indexação de arrays. Em um caso simples, podemos fazer algo como:

```
In [107]: arr = np.zeros((4, 3))

In [108]: arr[:] = 5

In [109]: arr
Out[109]:
array([[ 5., 5., 5.],
       [ 5., 5., 5.],
       [ 5., 5., 5.],
       [ 5., 5., 5.]])
```

No entanto, se tivermos um array unidimensional de valores que quiséssemos definir nas colunas do array, podemos fazer isso, desde que o formato seja compatível:

```
In [110]: col = np.array([1.28, -0.42, 0.44, 1.6])

In [111]: arr[:] = col[:, np.newaxis]

In [112]: arr
Out[112]:
array([[ 1.28, 1.28, 1.28],
       [-0.42, -0.42, -0.42],
```

```
       [ 0.44, 0.44, 0.44],
       [ 1.6 , 1.6 , 1.6 ]])

In [113]: arr[:2] = [[-1.37], [0.509]]

In [114]: arr
Out[114]:
array([[-1.37 , -1.37 , -1.37 ],
       [ 0.509, 0.509, 0.509],
       [ 0.44 , 0.44 , 0.44 ],
       [ 1.6 , 1.6 , 1.6 ]])
```

A.4 Usos avançados de ufuncs

Embora muitos usuários do NumPy apenas façam uso das operações rápidas pelos elementos, oferecidas pelas funções universais, há uma série de recursos adicionais que ocasionalmente poderão ajudar você a escrever um código mais conciso, sem laços.

Métodos de instância de ufuncs

Cada uma das ufuncs binárias do NumPy tem métodos especiais para executar determinados tipos de operações vetorizadas especiais. Elas estão sintetizadas na Tabela A.2, mas apresentarei alguns exemplos concretos para mostrar o seu funcionamento.

`reduce` aceita um único array e agrega seus valores, opcionalmente ao longo de um eixo, realizando uma sequência de operações binárias. Por exemplo, um modo alternativo de somar elementos em um array é usar `np.add.reduce`:

```
In [115]: arr = np.arange(10)

In [116]: np.add.reduce(arr)
Out[116]: 45

In [117]: arr.sum()
Out[117]: 45
```

O valor inicial (0 para `add`) depende da ufunc. Se um eixo for

passado, a redução será feita ao longo desse eixo. Isso permite que você responda a determinados tipos de perguntas de maneira concisa. Como um exemplo menos trivial, podemos utilizar np.logical_and para verificar se os valores em cada linha de um array estão ordenados:

```
In [118]: np.random.seed(12346) # para possibilitar uma reprodução

In [119]: arr = np.random.randn(5, 5)

In [120]: arr[::2].sort(1) # ordena algumas linhas

In [121]: arr[:, :-1] < arr[:, 1:]
Out[121]:
array([[ True, True, True, True],
       [False, True, False, False],
       [ True, True, True, True],
       [ True, False, True, True],
       [ True, True, True, True]], dtype=bool)

In [122]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1)
Out[122]: array([ True, False, True, False, True], dtype=bool)
```

Observe que logical_and.reduce é equivalente ao método all.

accumulate está relacionado a reduce, assim como cumsum está relacionado a sum. Ele gera um array de mesmo tamanho, com os valores “acumulados” intermediários:

```
In [123]: arr = np.arange(15).reshape((3, 5))

In [124]: np.add.accumulate(arr, axis=1)
Out[124]:
array([[ 0, 1, 3, 6, 10],
       [ 5, 11, 18, 26, 35],
       [10, 21, 33, 46, 60]])
```

outer efetua um produto cruzado aos pares entre dois arrays:

```
In [125]: arr = np.arange(3).repeat([1, 2, 2])

In [126]: arr
Out[126]: array([0, 1, 1, 2, 2])
```

```
In [127]: np.multiply.outer(arr, np.arange(5))
Out[127]:
array([[0, 0, 0, 0, 0],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8],
       [0, 2, 4, 6, 8]])
```

A saída de outer terá uma dimensão que será a soma das dimensões das entradas:

```
In [128]: x, y = np.random.randn(3, 4), np.random.randn(5)

In [129]: result = np.subtract.outer(x, y)

In [130]: result.shape
Out[130]: (3, 4, 5)
```

O último método, reduceat, faz uma “redução local”; essencialmente, é uma operação groupby de array na qual fatias do array são agregadas. Ele aceita uma sequência de “fronteiras de compartimentos” (bin edges) que informam como separar e agregar os valores:

```
In [131]: arr = np.arange(10)

In [132]: np.add.reduceat(arr, [0, 5, 8])
Out[132]: array([10, 18, 17])
```

Os resultados são as reduções (nesse caso, as somas) efetuadas sobre arr[0:5], arr[5:8] e arr[8:]. Como no caso dos demais métodos, é possível passar um argumento com o eixo:

```
In [133]: arr = np.multiply.outer(np.arange(4), np.arange(5))

In [134]: arr
Out[134]:
array([[ 0, 0, 0, 0, 0],
       [ 0, 1, 2, 3, 4],
       [ 0, 2, 4, 6, 8],
       [ 0, 3, 6, 9, 12]])
```

```
In [135]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[135]:
array([[ 0, 0, 0],
       [ 1, 5, 4],
       [ 2, 10, 8],
       [ 3, 15, 12]])
```

Veja a Tabela A.2 que apresenta uma lista parcial dos métodos de ufunc.

Tabela A.2 – Métodos de ufunc

Método	Descrição
reduce(x)	Agrega valores por meio de sucessivas aplicações da operação
accumulate(x)	Agrega valores preservando todas as agregações parciais
reduceat(x, bins)	Redução “local” ou “agrupar por”; reduz fatias contíguas de dados para gerar um array com agregação
outer(x, y)	Aplica a operação a todos os pares de elementos em x e y; o array resultante tem o formato x.shape + y.shape

Escrevendo novas ufuncs em Python

Há uma série de recursos para criar as próprias ufuncs NumPy. A mais genérica consiste em usar a API C do NumPy, mas isso está além do escopo deste livro. Nesta seção, veremos as ufuncs em Python puro.

numpy.frompyfunc aceita uma função Python, junto com uma especificação do número de entradas e de saídas. Por exemplo, uma função simples que faça uma soma dos elementos poderia ser especificada assim:

```
In [136]: def add_elements(x, y):
   .....: return x + y

In [137]: add_them = np.frompyfunc(add_elements, 2, 1)

In [138]: add_them(np.arange(8), np.arange(8))
Out[138]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

Funções criadas com frompyfunc sempre devolvem arrays de objetos

Python, o que pode ser inconveniente. Felizmente há uma função alternativa (porém com um pouco menos de recursos), numpy.vectorize, que permite especificar o tipo da saída:

```
In [139]: add_them = np.vectorize(add_elements, otypes=[np.float64])

In [140]: add_them(np.arange(8), np.arange(8))
Out[140]: array([ 0., 2., 4., 6., 8., 10., 12., 14.])
```

Essas funções oferecem uma maneira de criar funções do tipo ufunc, mas são muito lentas, pois exigem uma chamada de função Python para calcular cada elemento, o que é muito mais lento que os laços das ufuncs do NumPy baseadas em C:

```
In [141]: arr = np.random.randn(10000)

In [142]: %timeit add_them(arr, arr)
3.54 ms +- 178 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

In [143]: %timeit np.add(arr, arr)
6.76 us +- 370 ns per loop (mean +- std. dev. of 7 runs, 100000 loops each)
```

Mais adiante neste capítulo, mostraremos como criar ufuncs rápidas em Python usando o projeto Numba (*http://numba.pydata.org/*).

A.5 Arrays estruturados e de registros

Talvez você tenha percebido a essa altura que o ndarray é um contêiner de dados *homogêneos*, isto é, ele representa um bloco de memória em que cada elemento ocupa o mesmo número de bytes, determinado pelo dtype. À primeira vista, poderia parecer que não seria possível representar dados heterogêneos nem do tipo tabular. Um array *estruturado* é um ndarray no qual cada elemento pode ser pensado como uma representação de uma *struct* em C (daí o nome “estruturado”) ou uma linha de uma tabela SQL com vários campos nomeados:

```
In [144]: dtype = [('x', np.float64), ('y', np.int32)]

In [145]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)
```

```
In [146]: sarr
Out[146]:
array([( 1.5 , 6), ( 3.1416, -2)],
      dtype=[('x', '<f8'), ('y', '<i4')])
```

Há várias maneiras de especificar um dtype estruturado (consulte a documentação online do NumPy). Um modo típico é na forma de uma lista de tuplas com (field_name, field_data_type). Agora os elementos do array serão objetos do tipo tupla cujos elementos podem ser acessados como um dicionário:

```
In [147]: sarr[0]
Out[147]: ( 1.5, 6)

In [148]: sarr[0]['y']
Out[148]: 6
```

Os nomes dos campos são armazenados no atributo dtype.names. Ao acessar um campo do array estruturado, uma visão dos dados em passos (strided view) é devolvida, sem que haja nenhuma cópia:

```
In [149]: sarr['x']
Out[149]: array([ 1.5 , 3.1416])
```

dtypes aninhados e campos multidimensionais

Ao especificar um dtype estruturado, podemos passar adicionalmente um formato (como um int ou uma tupla):

```
In [150]: dtype = [('x', np.int64, 3), ('y', np.int32)]

In [151]: arr = np.zeros(4, dtype=dtype)

In [152]: arr
Out[152]:
array([([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0)],
      dtype=[('x', '<i8', (3,)), ('y', '<i4')])
```

Nesse caso, o campo x agora se refere a um array de tamanho 3 para cada registro:

```
In [153]: arr[0]['x']
Out[153]: array([0, 0, 0])
```

De modo conveniente, acessar arr['x'] devolve um array bidimensional, em vez de unidimensional como nos exemplos anteriores:

```
In [154]: arr['x']
Out[154]:
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Isso permite que você expresse estruturas aninhadas mais complexas como um único bloco de memória em um array. Você também pode aninhar dtypes a fim de criar estruturas mais complexas. Eis um exemplo:

```
In [155]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.int32)]

In [156]: data = np.array([((1, 2), 5), ((3, 4), 6)], dtype=dtype)

In [157]: data['x']
Out[157]:
array([( 1., 2.), ( 3., 4.)],
      dtype=[('a', '<f8'), ('b', '<f4')])

In [158]: data['y']
Out[158]: array([5, 6], dtype=int32)

In [159]: data['x']['a']
Out[159]: array([ 1., 3.])
```

O DataFrame do pandas não tem suporte direto para esse recurso, embora ele seja semelhante à indexação hierárquica.

Por que usar arrays estruturados?

Em comparação com, digamos, um DataFrame do pandas, os arrays estruturados do NumPy são uma ferramenta relativamente de baixo nível. Eles oferecem uma forma de interpretar um bloco de memória como uma estrutura tabular, com colunas aninhadas, arbitrariamente complexas. Como cada elemento do array é

representado na memória com um número fixo de bytes, os array estruturados oferecem um modo muito rápido e eficiente de escrever e ler dados de disco (incluindo mapeamentos em memória), transportá-los pela rede e outros fins como esses.

Para ilustrar outro uso comum dos array estruturados, temos a gravação de arquivos de dados como streams de bytes para registros de tamanho fixo, que é um modo usual de serializar dados em código C e C++, comumente encontrados em sistemas legados no mercado. Desde que o formato do arquivo seja conhecido (o tamanho de cada registro e a ordem, o tamanho em bytes e o tipo de dado de cada elemento), os dados poderão ser lidos em memória com np.fromfile. Usos especializados como esse estão além do escopo deste livro, mas vale a pena saber que operações desse tipo são possíveis.

A.6 Mais sobre ordenação

Assim como a lista embutida de Python, o método de instância sort de ndarray faz uma ordenação *in-place*, o que significa que o conteúdo do array é reorganizado sem que um novo array seja gerado:

```
In [160]: arr = np.random.randn(6)

In [161]: arr.sort()

In [162]: arr
Out[162]: array([-1.082 , 0.3759, 0.8014, 1.1397, 1.2888, 1.8413])
```

Quando ordenamos arrays in-place, lembre-se de que, se o array é uma visualização de um ndarray diferente, o array original será modificado:

```
In [163]: arr = np.random.randn(3, 5)

In [164]: arr
Out[164]:
array([[-0.3318, -1.4711, 0.8705, -0.0847, -1.1329],
```

```
       [-1.0111, -0.3436, 2.1714, 0.1234, -0.0189],
       [ 0.1773, 0.7424, 0.8548, 1.038 , -0.329 ]])

In [165]: arr[:, 0].sort() # Ordena os valores da primeira coluna in-place

In [166]: arr
Out[166]:
array([[-1.0111, -1.4711, 0.8705, -0.0847, -1.1329],
       [-0.3318, -0.3436, 2.1714, 0.1234, -0.0189],
       [ 0.1773, 0.7424, 0.8548, 1.038 , -0.329 ]])
```

Por outro lado, numpy.sort cria uma nova cópia ordenada de um array. Exceto por isso, ele aceita os mesmos argumentos (por exemplo, kind) que ndarray.sort:

```
In [167]: arr = np.random.randn(5)

In [168]: arr
Out[168]: array([-1.1181, -0.2415, -2.0051, 0.7379, -1.0614])

In [169]: np.sort(arr)
Out[169]: array([-2.0051, -1.1181, -1.0614, -0.2415, 0.7379])

In [170]: arr
Out[170]: array([-1.1181, -0.2415, -2.0051, 0.7379, -1.0614])
```

Todos esses métodos de ordenação aceitam o eixo como argumento para ordenar as seções de dados ao longo do eixo especificado de modo independente:

```
In [171]: arr = np.random.randn(3, 5)

In [172]: arr
Out[172]:
array([[ 0.5955, -0.2682, 1.3389, -0.1872, 0.9111],
       [-0.3215, 1.0054, -0.5168, 1.1925, -0.1989],
       [ 0.3969, -1.7638, 0.6071, -0.2222, -0.2171]])

In [173]: arr.sort(axis=1)

In [174]: arr
Out[174]:
```

```
array([[-0.2682, -0.1872, 0.5955, 0.9111, 1.3389],
       [-0.5168, -0.3215, -0.1989, 1.0054, 1.1925],
       [-1.7638, -0.2222, -0.2171, 0.3969, 0.6071]])
```

Talvez você tenha notado que nenhum dos métodos de ordenação tem uma opção para gerar uma ordem decrescente. Na prática, isso é um problema, pois um fatiamento de array gera visualizações, e, desse modo, não produz uma cópia nem exige qualquer trabalho de processamento. Muitos usuários de Python têm familiaridade com o "truque" segundo o qual, para uma lista values, values[::-1] devolve uma lista na ordem inversa. O mesmo vale para ndarrays:

```
In [175]: arr[:, ::-1]
Out[175]:
array([[ 1.3389, 0.9111, 0.5955, -0.1872, -0.2682],
       [ 1.1925, 1.0054, -0.1989, -0.3215, -0.5168],
       [ 0.6071, 0.3969, -0.2171, -0.2222, -1.7638]])
```

Ordenações indiretas: argsort e lexsort

Em análise de dados, talvez você precise reordenar conjuntos de dados de acordo com uma ou mais chaves. Por exemplo, é possível que seja necessário ordenar uma tabela de dados de alguns alunos de acordo com o sobrenome e então pelo primeiro nome. Esse é um exemplo de ordenação *indireta*, e, se você já leu os capítulos relacionados ao pandas, terá visto muitos exemplos gerais. Dada uma chave ou chaves (um array de valores ou vários arrays de valores), é possível obter um array de *índices* inteiros (refiro-me a eles de modo coloquial como *indexadores*) que informe como reorganizar os dados para que estejam ordenados. Dois métodos para fazer isso são argsort e numpy.lexsort. Veja um exemplo:

```
In [176]: values = np.array([5, 0, 1, 3, 2])

In [177]: indexer = values.argsort()

In [178]: indexer
Out[178]: array([1, 2, 4, 3, 0])

In [179]: values[indexer]
```

```
Out[179]: array([0, 1, 2, 3, 5])
```

Como um exemplo mais complexo, o código a seguir reordena um array bidimensional de acordo com a sua primeira linha:

```
In [180]: arr = np.random.randn(3, 5)

In [181]: arr[0] = values

In [182]: arr
Out[182]:
array([[ 5. , 0. , 1. , 3. , 2. ],
       [-0.3636, -0.1378, 2.1777, -0.4728, 0.8356],
       [-0.2089, 0.2316, 0.728 , -1.3918, 1.9956]])

In [183]: arr[:, arr[0].argsort()]
Out[183]:
array([[ 0. , 1. , 2. , 3. , 5. ],
       [-0.1378, 2.1777, 0.8356, -0.4728, -0.3636],
       [ 0.2316, 0.728 , 1.9956, -1.3918, -0.2089]])
```

lexsort é semelhante a argsort, mas faz uma ordenação *lexicográfica* indireta em vários arrays de chave. Suponha que quiséssemos ordenar alguns dados identificados pelo primeiro nome e pelo sobrenome:

```
In [184]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'])

In [185]: last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones', 'Walters'])

In [186]: sorter = np.lexsort((first_name, last_name))

In [187]: sorter
Out[187]: array([1, 2, 3, 0, 4])

In [188]: zip(last_name[sorter], first_name[sorter])
Out[188]: <zip at 0x7efeec8e38c8>
```

lexsort pode ser um pouco confuso na primeira vez que você o utiliza porque a ordem na qual as chaves são usadas para ordenar os dados começa pelo *último* array especificado. Nesse exemplo, last_name foi usado antes de first_name.

 Os métodos do pandas, por exemplo, sort_values de Series e de DataFrame, são implementados com variantes dessas funções (que também devem levar em consideração os valores ausentes).

Algoritmos de ordenação alternativos

Um algoritmo de ordenação *estável* preserva a posição relativa de elementos iguais. Isso pode ser particularmente importante em ordenações indiretas, em que a ordenação relativa é significativa:

```
In [189]: values = np.array(['2:first', '2:second', '1:first', '1:second',
   .....: '1:third'])

In [190]: key = np.array([2, 2, 1, 1, 1])

In [191]: indexer = key.argsort(kind='mergesort')

In [192]: indexer
Out[192]: array([2, 3, 4, 0, 1])

In [193]: values.take(indexer)
Out[193]:
array(['1:first', '1:second', '1:third', '2:first', '2:second'], dtype='<U8')
```

A única ordenação estável disponível é *mergesort*, que tem um desempenho garantido de O(n log n) (para os entusiastas por complexidade), mas seu desempenho, em média, é pior do que o método quicksort padrão. Veja a Tabela A.3 que apresenta um resumo dos métodos disponíveis e seus desempenhos relativos (e as garantias de desempenho). Essas não são informações em que a maioria dos usuários terá que pensar, mas é conveniente saber que existem.

Tabela A.3 – Métodos para ordenação de arrays

Tipo	Velocidade	Estável	Espaço de trabalho	Pior caso
'quicksort'	1	Não	0	O(n^2)
'mergesort'	2	Sim	n / 2	O(n log n)
'heapsort'	3	Não	0	O(n log n)

Ordenando arrays parcialmente

Um dos objetivos da ordenação pode ser determinar o maior ou o menor elemento de um array. O NumPy tem métodos otimizados, numpy.partition e np.argpartition, para particionar um array em torno do k-ésimo menor elemento:

```
In [194]: np.random.seed(12345)

In [195]: arr = np.random.randn(20)

In [196]: arr
Out[196]:
array([-0.2047, 0.4789, -0.5194, -0.5557, 1.9658, 1.3934, 0.0929,
       0.2817, 0.769 , 1.2464, 1.0072, -1.2962, 0.275 , 0.2289,
       1.3529, 0.8864, -2.0016, -0.3718, 1.669 , -0.4386])

In [197]: np.partition(arr, 3)
Out[197]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -0.2047,
       0.2817, 0.769 , 0.4789, 1.0072, 0.0929, 0.275 , 0.2289,
       1.3529, 0.8864, 1.3934, 1.9658, 1.669 , 1.2464])
```

Depois de chamar partition(arr, 3), os três primeiros elementos do resultado serão os três menores valores, em nenhuma ordem específica. numpy.argpartition, de modo semelhante a numpy.argsort, devolve os índices que reorganizam os dados na ordem equivalente:

```
In [198]: indices = np.argpartition(arr, 3)

In [199]: indices
Out[199]:
array([16, 11, 3, 2, 17, 19, 0, 7, 8, 1, 10, 6, 12, 13, 14, 15, 5, 4, 18, 9])

In [200]: arr.take(indices)
Out[200]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -0.2047,
       0.2817, 0.769 , 0.4789, 1.0072, 0.0929, 0.275 , 0.2289,
       1.3529, 0.8864, 1.3934, 1.9658, 1.669 , 1.2464])
```

numpy.searchsorted: encontrando elementos em

um array ordenado

searchsorted é um método de array que faz uma busca binária em um array ordenado, devolvendo a posição no array em que o valor deveria ser inserido para mantê-lo ordenado:

```
In [201]: arr = np.array([0, 1, 7, 12, 15])

In [202]: arr.searchsorted(9)
Out[202]: 3
```

Podemos também passar um array de valores a fim de obter de volta um array de índices:

```
In [203]: arr.searchsorted([0, 8, 11, 16])
Out[203]: array([0, 3, 3, 5])
```

Talvez você tenha percebido que searchsorted devolveu 0 para o elemento 0. Isso ocorre porque o comportamento default consiste em devolver o índice do lado esquerdo de um grupo de valores iguais:

```
In [204]: arr = np.array([0, 0, 0, 1, 1, 1, 1])

In [205]: arr.searchsorted([0, 1])
Out[205]: array([0, 3])

In [206]: arr.searchsorted([0, 1], side='right')
Out[206]: array([3, 7])
```

Como outra aplicação de searchsorted, suponha que tivéssemos um array de valores entre 0 e 10.000, e um array separado de “fronteiras de buckets” que quiséssemos usar para separar os dados em compartimentos:

```
In [207]: data = np.floor(np.random.uniform(0, 10000, size=50))

In [208]: bins = np.array([0, 100, 1000, 5000, 10000])

In [209]: data
Out[209]:
array([ 9940., 6768., 7908., 1709., 268., 8003., 9037., 246.,
        4917., 5262., 5963., 519., 8950., 7282., 8183., 5002.,
```

```
       8101., 959., 2189., 2587., 4681., 4593., 7095., 1780.,
       5314., 1677., 7688., 9281., 6094., 1501., 4896., 3773.,
       8486., 9110., 3838., 3154., 5683., 1878., 1258., 6875.,
       7996., 5735., 9732., 6340., 8884., 4954., 3516., 7142.,
       5039., 2256.])
```

Para obter então rótulos informando a qual intervalo cada ponto de dado pertence (em que 1 significaria o bucket [0, 100)), basta usar searchsorted:

```
In [210]: labels = bins.searchsorted(data)

In [211]: labels
Out[211]:
array([4, 4, 4, 3, 2, 4, 4, 2, 3, 4, 4, 2, 4, 4, 4, 4, 4, 2, 3, 3, 3, 3, 4,
       3, 4, 3, 4, 4, 4, 3, 3, 3, 4, 4, 3, 3, 4, 3, 3, 4, 4, 4, 4, 4, 4, 3,
       3, 4, 4, 3])
```

Essa operação, em conjunto com o groupby do pandas, pode ser usada para compartimentar os dados:

```
In [212]: pd.Series(data).groupby(labels).mean()
Out[212]:
2 498.000000
3 3064.277778
4 7389.035714
dtype: float64
```

A.7 Escrevendo funções NumPy rápidas com o Numba

O Numba (*http://numba.pydata.org/*) é um projeto de código aberto para criação de funções rápidas para dados do tipo NumPy usando CPUs, GPUs ou outros hardwares. Ele utiliza o Projeto LLVM (*http://llvm.org/*) para traduzir código Python em código de máquina compilado.

Para apresentar o Numba, vamos considerar uma função em Python puro que calcule a expressão (x - y).mean() usando um laço for:

```
import numpy as np
```

```
def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

Essa função é bem lenta:

```
In [209]: x = np.random.randn(10000000)

In [210]: y = np.random.randn(10000000)

In [211]: %timeit mean_distance(x, y)
1 loop, best of 3: 2 s per loop

In [212]: %timeit (x - y).mean()
100 loops, best of 3: 14.7 ms per loop
```

A versão do NumPy é mais de 100 vezes mais rápida. Podemos transformar essa função em uma função Numba compilada usando a função numba.jit:

```
In [213]: import numba as nb

In [214]: numba_mean_distance = nb.jit(mean_distance)
```

Também poderíamos ter escrito esse código usando um decorador:

```
@nb.jit
def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

A função resultante, na verdade, é mais rápida que a versão vetorizada do NumPy:

```
In [215]: %timeit numba_mean_distance(x, y)
```

```
100 loops, best of 3: 10.3 ms per loop
```

O Numba não é capaz de compilar um código Python arbitrário, mas aceita um subconjunto significativo de Python puro que é mais útil para escrever algoritmos numéricos.

O Numba é uma biblioteca detalhada, com suporte para diferentes tipos de hardware, modos de compilação e extensões de usuários. Também é capaz de compilar um subconjunto substancial da API Python do NumPy sem laços `for` explícitos. O Numba consegue reconhecer construções que podem ser compiladas para código de máquina, ao mesmo tempo em que substitui chamadas à API CPython para funções às quais ele não sabe como compilar. A função `jit` do Numba tem uma opção `nopython=True` que restringe o código permitido a um código Python que possa ser compilado para LLVM sem qualquer chamada da API C de Python. `jit(nopython=True)` tem um alias mais conciso, `numba.njit`.

No exemplo anterior, poderíamos ter escrito o seguinte:

```
from numba import float64, njit

@njit(float64(float64[:], float64[:]))
def mean_distance(x, y):
    return (x - y).mean()
```

Incentivo você a aprender mais lendo a documentação online do Numba (*http://numba.pydata.org/*). A próxima seção mostra um exemplo de criação de objetos ufunc personalizados do NumPy.

Criando objetos numpy.ufunc personalizados com o Numba

A função `numba.vectorize` cria ufuncs NumPy personalizadas, que se comportam como ufuncs embutidas. Vamos considerar uma implementação Python de `numpy.add`:

```
from numba import vectorize

@vectorize
def nb_add(x, y):
```

```
    return x + y
```

Agora temos:

```
In [13]: x = np.arange(10)

In [14]: nb_add(x, x)
Out[14]: array([ 0., 2., 4., 6., 8., 10., 12., 14., 16., 18.])

In [15]: nb_add.accumulate(x, 0)
Out[15]: array([ 0., 1., 3., 6., 10., 15., 21., 28., 36., 45.])
```

A.8 Operações avançadas de entrada e saída com arrays

No Capítulo 4, fomos apresentados a np.save e a np.load para armazenar arrays em formato binário em disco. Há uma série de opções adicionais a serem consideradas para usos mais sofisticados. Em particular, o mapeamento em memória tem a vantagem extra de permitir que você trabalhe com conjuntos de dados que não caibam em RAM.

Arquivos mapeados em memória

Um arquivo *mapeado em memória* é um método para interagir com dados binários em disco como se eles estivessem armazenados em um array na memória. O NumPy implementa um objeto memmap, que é do tipo ndarray, permitindo que pequenos segmentos de um arquivo grande sejam lidos e escritos sem ler o array todo na memória. Além disso, um memmap tem os mesmos métodos que um array em memória e, desse modo, pode ser usado como substituto em muitos algoritmos em que um ndarray seria esperado.

Para criar um novo mapeamento em memória, utilize a função np.memmap e passe um path de arquivo, o dtype, o formato e o modo do arquivo:

```
In [214]: mmap = np.memmap('mymmap', dtype='float64', mode='w+',
   .....: shape=(10000, 10000))
```

```
In [215]: mmap
Out[215]:
memmap([[ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.],
        ...,
        [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.]])
```

Fatiar um memmap devolve visualizações dos dados em disco:

```
In [216]: section = mmap[:5]
```

Se você atribuir dados a esse mapeamento, eles serão armazenados em um buffer na memória (como ocorre com um objeto arquivo de Python), mas você poderá gravá-los em disco chamando flush:

```
In [217]: section[:] = np.random.randn(5, 10000)

In [218]: mmap.flush()

In [219]: mmap
Out[219]:
memmap([[ 0.7584, -0.6605, 0.8626, ..., 0.6046, -0.6212, 2.0542],
        [-1.2113, -1.0375, 0.7093, ..., -1.4117, -0.1719, -0.8957],
        [-0.1419, -0.3375, 0.4329, ..., 1.2914, -0.752 , -0.44 ],
        ...,
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ],
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ],
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ]])

In [220]: del mmap
```

Sempre que um mapeamento em memória sair do escopo e for submetido à coleta de lixo (garbage collection), qualquer alteração será descarregada em disco também. Ao *abrir um mapeamento em memória* já existente, você ainda terá que especificar o dtype e o formato, pois o arquivo é apenas um bloco de dados binários, sem metadados em disco:

```
In [221]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000,
```

```
10000))

In [222]: mmap
Out[222]:
memmap([[ 0.7584, -0.6605, 0.8626, ..., 0.6046, -0.6212, 2.0542],
        [-1.2113, -1.0375, 0.7093, ..., -1.4117, -0.1719, -0.8957],
        [-0.1419, -0.3375, 0.4329, ..., 1.2914, -0.752 , -0.44 ],
        ...,
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ],
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ],
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ]])
```

Os mapeamentos em memória também funcionam com dtypes estruturados ou aninhados, conforme descritos em uma seção anterior.

HDF5 e outras opções para armazenagem de arrays

PyTables e h5py são dois projetos Python que oferecem interfaces apropriadas ao NumPy para armazenar dados de array no formato HDF5, que é eficaz e pode ser compactado (HDF quer dizer *Hierarchical Data Format*, ou Formato de Dados Hierárquicos). Você pode armazenar centenas de gigabytes ou até mesmo terabytes de dados no formato HDF5, com segurança. Para saber mais sobre o uso de HDF5 com Python, recomendo ler a documentação online do pandas (*http://pandas.pydata.org/*).

A.9 Dicas para o desempenho

Ter um bom desempenho de um código que utilize o NumPy em geral é simples, pois as operações com arrays usualmente substituem os laços comparativamente muito mais lentos em Python puro. A lista a seguir resume alguns pontos para se ter em mente:

- converter laços Python e lógicas condicionais em operações de array e em operações de arrays booleanos;
- usar broadcasting sempre que for possível;

- usar visualizações de arrays (fatiamento) para evitar cópias de dados;
- utilizar ufuncs e métodos de ufuncs.

Se você não puder alcançar o desempenho necessário depois de esgotar os recursos oferecidos exclusivamente pelo NumPy, considere escrever o código em C, em Fortran ou em Cython. Utilizo Cython (*http://cython.org/*) com frequência em meu próprio trabalho como uma maneira fácil de ter um desempenho semelhante ao de C com um mínimo de desenvolvimento.

A importância da memória contígua

Embora a extensão completa desse assunto esteja um pouco fora do escopo deste livro, em algumas aplicações, o layout de memória de um array pode afetar significativamente a velocidade dos processamentos. Em parte, esse fato baseia-se nas diferenças de desempenho associadas à hierarquia de caches da CPU; operações que acessem blocos contíguos de memória (por exemplo, somar as linhas de um array que utilize a ordem C) em geral serão mais rápidas porque o subsistema de memória inserirá os blocos de memória apropriados em um buffer na ultrarrápida cache L1 ou L2 da CPU. Além disso, determinados caminhos de código na base de código C do NumPy foram otimizados para o caso contíguo, em que é possível evitar um acesso de memória genérico em passos.

Dizer que o layout de memória de um array é *contíguo* significa que os elementos são armazenados na memória na ordem em que aparecem no array no que diz respeito à ordenação Fortran (orientado a colunas, isto é, column major) ou C (orientado a linhas, isto é, row major). Por padrão, os arrays NumPy são criados como *contíguos em ordem C*, ou apenas contíguos. Dizemos que um array orientado a colunas, como a transposta de um array C contíguo, é contíguo em ordem Fortran. Essas propriedades podem ser explicitamente verificadas por meio do atributo `flags` do ndarray:

```
In [225]: arr_c = np.ones((1000, 1000), order='C')
```

```
In [226]: arr_f = np.ones((1000, 1000), order='F')

In [227]: arr_c.flags
Out[227]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [228]: arr_f.flags
Out[228]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [229]: arr_f.flags.f_contiguous
Out[229]: True
```

Nesse exemplo, somar as linhas desses arrays, teoricamente, deve ser mais rápido para `arr_c` do que para `arr_f`, pois as linhas estão contíguas na memória. A seguir, farei uma verificação definitiva usando `%timeit` no IPython:

```
In [230]: %timeit arr_c.sum(1)
842 us +- 40.3 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)

In [231]: %timeit arr_f.sum(1)
879 us +- 23.6 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
```

Se estivermos tentando extrair o máximo de desempenho do NumPy, com frequência esse será um local para investir esforços. Se você tiver um array que não tenha a ordem desejada na memória, utilize `copy` e passe `'C'` ou `'F'`:

```
In [232]: arr_f.copy('C').flags
Out[232]:
```

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

Ao construir uma visualização de um array, tenha em mente que não há garantias de que o resultado será contíguo:

```
In [233]: arr_c[:50].flags.contiguous
Out[233]: True

In [234]: arr_c[:, :50].flags
Out[234]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

---

1 Alguns dos dtypes têm underscores no final dos nomes. Eles estão presentes para evitar conflitos de nomes de variáveis entre os tipos específicos do NumPy e os tipos embutidos de Python.

APÊNDICE B

Mais sobre o sistema IPython

No Capítulo 2, vimos o básico sobre o uso do shell IPython e do Notebook Jupyter. Neste capítulo, exploraremos algumas funcionalidades mais sofisticadas do sistema IPython, as quais podem ser usadas a partir do console ou no Jupyter.

B.1 Usando o histórico de comandos

O IPython mantém um pequeno banco de dados em disco contendo o texto de cada comando que você executar. Esse recurso tem diversos propósitos, possibilitando:

- procurar, completar e executar comandos dados anteriormente com o mínimo de digitação;
- fazer a persistência do histórico de comandos entre as sessões;
- fazer logging do histórico de entrada/saída em um arquivo.

Esses recursos são mais úteis no shell do que no notebook, pois o notebook, por design, mantém um log da entrada e da saída em cada célula de código.

Pesquisando e reutilizando o histórico de comandos

O shell IPython permite pesquisar e executar códigos anteriores ou outros comandos. Isso é conveniente, pois, com frequência, você poderá se ver repetindo os mesmos comandos, por exemplo, um comando `%run` ou outra porção de código. Suponha que você tivesse executado o seguinte comando:

```
In[7]: %run first/second/third/data_script.py
```

e então tivesse explorado o resultado do script (considerando que ele tivesse executado com sucesso), somente para descobrir que fez um cálculo incorreto. Depois de ter descoberto o problema e modificado *data_script.py*, poderá começar a digitar algumas letras do comando %run e então pressionar a combinação de teclas Ctrl-P ou a seta para cima. Isso fará com que o histórico de comandos seja pesquisado em busca do primeiro comando anterior cujas letras coincidam com aquelas que você digitou. Pressionar Ctrl-P ou a seta para cima várias vezes continuará a pesquisa no histórico. Se você passar pelo comando que deseja executar, não se preocupe. É possível ir *para frente* no histórico de comandos pressionando Ctrl-N ou a tecla de seta para baixo. Depois de fazer isso algumas vezes, passará a pressionar essas teclas sem pensar!

Usar Ctrl-R possibilita ter o mesmo recurso de pesquisa parcial incremental disponibilizado pelo readline usado em shells de estilo Unix, por exemplo, o shell bash. No Windows, a funcionalidade de readline é emulada pelo IPython. Para usá-la, pressione Ctrl-R e então digite alguns caracteres contidos na linha de entrada que deseja procurar:

```
In [1]: a_command = foo(x, y, z)

(reverse-i-search)`com': a_command = foo(x, y, z)
```

Pressionar Ctrl-R percorrerá o histórico ciclicamente em busca de cada linha cujos caracteres coincidam com aqueles que você digitou.

Variáveis de entrada e de saída

Esquecer-se de atribuir o resultado de uma chamada de função a uma variável pode ser muito irritante. Uma sessão do IPython armazena referências *tanto* para os comandos de entrada *quanto* para os objetos Python de saída em variáveis especiais. As duas saídas anteriores são armazenadas nas variáveis _ (um underscore) e __

(dois underscores), respectivamente:

```
In [24]: 2 ** 27
Out[24]: 134217728

In [25]: _
Out[25]: 134217728
```

As variáveis de entrada são armazenadas em variáveis de nomes como _iX, em que X é o número da linha de entrada. Para cada variável de entrada, há uma variável de saída _X correspondente. Assim, depois da linha de entrada 27, por exemplo, haverá duas novas variáveis _27 (para a saída) e _i27 (para a entrada):

```
In [26]: foo = 'bar'

In [27]: foo
Out[27]: 'bar'

In [28]: _i27
Out[28]: u'foo'

In [29]: _27
Out[29]: 'bar'
```

Como as variáveis de entrada são strings, elas podem ser executadas novamente com a palavra reservada exec de Python:

```
In [30]: exec(_i27)
```

Nesse caso, _i27 refere-se ao código de entrada em In [27].

Várias funções mágicas nos permitem trabalhar com o histórico de entrada e de saída. %hist é capaz de exibir todo o histórico de entrada, ou parte dele, com ou sem os números das linhas. %reset serve para limpar o namespace interativo e, opcionalmente, os caches de entrada e de saída. A função mágica %xdel tem como propósito remover todas as referências a um objeto *em particular* do sistema IPython. Consulte a documentação sobre essas duas funções mágicas para obter mais detalhes.

Ao trabalhar com conjuntos de dados bem grandes, tenha em mente que o histórico de entrada e de saída do IPython faz com que nenhum objeto referenciado aí esteja sujeito à coleta de lixo (garbage collection, que libera a memória), mesmo que você apague as variáveis do namespace interativo usando a palavra reservada del. Em casos como esse, um uso cuidadoso de %xdel e de %reset pode ajudá-lo a não ter problemas de memória.

B.2 Interagindo com o sistema operacional

Outro recurso do IPython é que ele permite que você acesse naturalmente o sistema de arquivos e o shell do sistema operacional. Entre outras coisas, isso significa que você pode executar a maioria das ações padrões da linha de comando como se estivesse no shell do Windows ou do Unix (Linux, macOS), sem ter que sair do IPython. Esses incluem comandos de shell, mudança de diretórios e armazenagem dos resultados de um comando em um objeto Python (lista ou string). Há também recursos para atribuição simples de aliases a comandos e criação de marcadores (bookmarking) para diretórios.

Veja a Tabela B.1 que apresenta um resumo das funções mágicas e a sintaxe para chamar comandos do shell. Apresentarei rapidamente esses recursos nas próximas seções.

Tabela B.1 – Comandos do IPython relacionados ao sistema

Comando	Descrição
!cmd	Executa cmd no shell do sistema
output = !cmd args	Executa cmd e armazena o stdout em output
%alias *nome_do_alias cmd*	Define um alias para um comando do sistema (shell)
%bookmark	Utiliza o sistema de marcadores (bookmarks) de diretório do IPython
%cd *diretório*	Muda o diretório de trabalho do sistema para o diretório especificado
%pwd	Devolve o diretório de trabalho atual do sistema
%pushd *diretório*	Coloca o diretório atual na pilha e muda para o diretório

	alvo
%popd	Muda para o diretório retirado do topo da pilha
%dirs	Devolve uma lista contendo a pilha de diretórios atuais
%dhist	Exibe o histórico dos diretórios acessados
%env	Devolve as variáveis de ambiente do sistema na forma de um dicionário
%matplotlib	Configura as opções de integração com a matplotlib

Comandos do shell e aliases

Iniciar uma linha no IPython com um ponto de exclamação `!`, ou bang, informa ao IPython que ele deve executar tudo que estiver depois do bang no shell do sistema. Isso significa que você pode apagar arquivos (usando `rm` ou `del`, conforme o seu sistema operacional), mudar de diretório ou executar qualquer outro processo.

É possível armazenar a saída do console de um comando de shell em uma variável atribuindo a expressão escapada com `!` a uma variável. Por exemplo, em meu computador baseado em Linux conectado à internet via ethernet, posso obter meu endereço IP como uma variável Python:

```
In [1]: ip_info = !ifconfig wlan0 | grep "inet "

In [2]: ip_info[0].strip()
Out[2]: 'inet addr:10.0.0.11 Bcast:10.0.0.255 Mask:255.255.255.0'
```

O objeto Python devolvido `ip_info`, na verdade, é um tipo lista personalizado contendo diversas versões da saída do console.

O IPython também pode fazer substituições por valores Python definidos no ambiente atual quando`!` é usado. Para isso, coloque o sinal de cifrão `$` antes do nome da variável:

```
In [3]: foo = 'test*'

In [4]: !ls $foo
test4.py test.py test.xml
```

A função mágica `%alias` pode definir atalhos personalizados para

comandos de shell. Eis um exemplo simples:

```
In [1]: %alias ll ls -l

In [2]: ll /usr
total 332
drwxr-xr-x 2 root root 69632 2012-01-29 20:36 bin/
drwxr-xr-x 2 root root 4096 2010-08-23 12:05 games/
drwxr-xr-x 123 root root 20480 2011-12-26 18:08 include/
drwxr-xr-x 265 root root 126976 2012-01-29 20:36 lib/
drwxr-xr-x 44 root root 69632 2011-12-26 18:08 lib32/
lrwxrwxrwx 1 root root 3 2010-08-23 16:02 lib64 -> lib/
drwxr-xr-x 15 root root 4096 2011-10-13 19:03 local/
drwxr-xr-x 2 root root 12288 2012-01-12 09:32 sbin/
drwxr-xr-x 387 root root 12288 2011-11-04 22:53 share/
drwxrwsr-x 24 root src 4096 2011-07-17 18:38 src/
```

Podemos executar vários comandos exatamente como na linha de comando, separando-os com ponto e vírgula:

```
In [558]: %alias test_alias (cd examples; ls; cd ..)

In [559]: test_alias
macrodata.csv spx.csv tips.csv
```

Você perceberá que o IPython “esquecerá” qualquer alias que você definir interativamente assim que a sessão for encerrada. Para criar aliases permanentes, utilize o sistema de configuração.

Sistema de marcadores de diretórios

O IPython tem um sistema simples de marcação (bookmarking) de diretórios que permite salvar aliases para diretórios comuns, para que você os acesse facilmente. Por exemplo, suponha que você quisesse criar um marcador que apontasse para os materiais suplementares deste livro:

```
In [6]: %bookmark py4da /home/wesm/code/pydata-book
```

Depois de fazer isso, se usarmos a função mágica `%cd`, poderemos utilizar qualquer marcador que tivermos definido:

```
In [7]: cd py4da
```

```
(bookmark:py4da) -> /home/wesm/code/pydata-book
/home/wesm/code/pydata-book
```

Se houver um conflito entre o nome do marcador e um nome de diretório em seu diretório de trabalho atual, você poderá utilizar a flag `-b` para sobrescrever e usar o local do marcador. Utilizar a opção `-l` com `%bookmark` fará todos os seus marcadores serem listados:

```
In [8]: %bookmark -l
Current bookmarks:
py4da -> /home/wesm/code/pydata-book-source
```

Os marcadores, de modo diferente dos aliases, têm persistência automática entre as sessões do IPython.

B.3 Ferramentas para desenvolvimento de software

Além de ser um ambiente confortável para processamento interativo e exploração de dados, o IPython também pode ser um companheiro conveniente para um desenvolvimento de software genérico em Python. Em aplicações de análise de dados, é importante ter, antes de tudo, um código *correto.* Felizmente o IPython tem uma boa integração com o depurador embutido `pdb` de Python, aperfeiçoando-o. Em segundo lugar, você vai querer que o seu código seja *rápido.* Para isso, o IPython tem ferramentas fáceis de usar para medir o tempo de execução de códigos e gerar seus perfis. Apresentarei uma visão geral dessas ferramentas de forma detalhada nesta seção.

Depurador interativo

O depurador do IPython aperfeiçoou o `pdb`, oferecendo preenchimento automático com tabulação, destaque de sintaxe e contexto para cada linha nos tracebacks de exceções. Um dos melhores momentos para depurar o código é logo depois que um erro ocorrer. O comando `%debug`, quando executado imediatamente após uma exceção, chama o depurador “post-mortem” e coloca

você dentro do stack frame em que a exceção foi lançada:

```
In [2]: run examples/ipython_bug.py
---------------------------------------------------------------------------
AssertionError Traceback (most recent call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
    13 throws_an_exception()
    14
---> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_things()
    11 def calling_things():
    12 works_fine()
---> 13 throws_an_exception()
    14
    15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in
throws_an_exception()
     7 a = 5
     8 b = 6
----> 9 assert(a + b == 10)
    10
    11 def calling_things():

AssertionError:

In [3]: %debug
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(9)throws_an_exception()
     8 b = 6
----> 9 assert(a + b == 10)
    10

ipdb>
```

Depois que estiver no depurador, você poderá executar um código Python arbitrário e explorar todos os objetos e dados (que foram “mantidos vivos” pelo interpretador) em cada stack frame. Por padrão, você começará no nível mais baixo, no lugar em que o erro ocorreu. Ao pressionar `u` (para cima) e `d` (para baixo), será possível

alternar entre os níveis do stack trace:

```
ipdb> u
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_things()
    12 works_fine()
---> 13 throws_an_exception()
    14
```

Executar o comando %pdb faz com que o IPython chame automaticamente o depurador depois de qualquer exceção – um modo que muitos usuários acharão particularmente útil.

Também é fácil utilizar o depurador para ajudar a desenvolver o código, especialmente quando você deseja definir breakpoints ou entrar na execução de uma função ou de um script a fim de analisar o estado em cada etapa. Há várias maneiras de fazer isso. A primeira é usar %run com a flag -d, que chama o depurador antes de executar qualquer código no script especificado. Pressione s (step) imediatamente para entrar no script:

```
In [5]: run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(1)<module>()
1---> 1 def works_fine():
     2 a = 5
     3 b = 6
```

Depois desse ponto, caberá a você decidir de que modo o arquivo será percorrido. Por exemplo, na exceção anterior, poderíamos definir um breakpoint imediatamente antes de chamar o método works_fine e executar o script até que ele seja alcançado, pressionando c (continuar).

```
ipdb> b 12
ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py(12)calling_things()
    11 def calling_things():
```

```
2--> 12 works_fine()
     13 throws_an_exception()
```

Nesse momento, você pode executar step em works_fine() ou executar works_fine() pressionado n (next) a fim de avançar para a próxima linha:

```
ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_things()
2 12 works_fine()
---> 13 throws_an_exception()
     14
```

Então poderíamos entrar em throws_an_exception, avançar até a linha em que o erro ocorre e observar as variáveis no escopo. Observe que os comandos do depurador têm precedência sobre os nomes das variáveis; em casos como esse, coloque ! antes das variáveis para analisar o seu conteúdo:

```
ipdb> s
--Call--
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(6)throws_an_exception()
      5
----> 6 def throws_an_exception():
      7 a = 5

ipdb> n
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(7)throws_an_exception()
      6 def throws_an_exception():
----> 7 a = 5
      8 b = 6

ipdb> n
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(8)throws_an_exception()
      7 a = 5
----> 8 b = 6
      9 assert(a + b == 10)

ipdb> n
```

```
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(9)throws_an_exception()
      8 b = 6
----> 9 assert(a + b == 10)
     10

ipdb> !a
5
ipdb> !b
6
```

Ter proficiência com o depurador interativo, em boa medida, é uma questão de prática e de experiência. Veja a Tabela B.2 que tem um catálogo completo dos comandos do depurador. Se você está acostumado a usar um IDE, talvez ache que o depurador orientado a terminal deixa um pouco a desejar à primeira vista, mas isso melhorará com o tempo. Alguns dos IDEs Python têm excelentes depuradores com GUIs, portanto a maioria dos usuários poderá encontrar algo que lhes seja apropriado.

Tabela B.2 – Comandos do depurador do (I)Python

Comando	Ação
h(elp)	Exibe a lista de comandos
help *comando*	Mostra a documentação para o *comando*
c(ontinue)	Retoma a execução do programa
q(uit)	Sai do depurador sem executar nenhum outro código adicional
b(reak) *número*	Define um breakpoint em *número* no arquivo atual
b *path/para/arquivo.py:número*	Define um breakpoint na linha *número* do arquivo especificado
s(tep)	*Entra* na chamada da função
n(ext)	Executa a linha atual e avança para a próxima linha no nível atual
u(p)/d(own)	Move para cima/para baixo na pilha de chamadas de função
a(rgs)	Mostra os argumentos da função atual
debug *instrução*	Chama a *instrução* em um novo depurador (recursivo)

Comando	Ação
l(ist) *instrução*	Mostra a posição atual e o contexto no nível atual da pilha
w(here)	Exibe o stack trace completo com o contexto na posição atual

Outras formas de fazer uso do depurador

Há duas outras maneiras convenientes de chamar o depurador. A primeira é usar uma função especial set_trace (que recebeu esse nome por causa de pdb.set_trace) que, basicamente, é um “breakpoint de pobre”. Eis duas pequenas receitas que talvez você queira guardar em algum lugar para seu uso geral (possivelmente as adicionando em seu perfil do IPython, como eu faço):

```
from IPython.core.debugger import Pdb

def set_trace():
    Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    pdb = Pdb(color_scheme='Linux')
    return pdb.runcall(f, *args, **kwargs)
```

A primeira função, set_trace, é bem simples. Você pode usar um set_trace em qualquer parte de seu código que queira interromper temporariamente a fim de analisá-lo com mais cuidado (por exemplo, imediatamente antes de uma exceção ocorrer):

```
In [7]: run examples/ipython_bug.py
> /home/wesm/code/pydata-book/examples/ipython_bug.py(16)calling_things()
     15 set_trace()
---> 16 throws_an_exception()
     17
```

Pressionar c (continuar) fará o código retomar a execução normalmente, sem qualquer prejuízo.

A função debug que acabamos de ver permite que você chame o depurador interativo facilmente em uma chamada de função qualquer. Suponha que tivéssemos escrito uma função como a que

vemos a seguir e quiséssemos entrar em sua lógica:

```
def f(x, y, z=1):
    tmp = x + y
    return tmp / z
```

Em geral, o uso de `f` teria um aspecto como `f(1, 2, z=3)`. Para entrar na execução de `f`, passe-o como o primeiro argumento de `debug`, seguido dos argumentos posicionais e nomeados a serem passados para `f`:

```
In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()
      1 def f(x, y, z):
----> 2 tmp = x + y
      3 return tmp / z

ipdb>
```

Acho que essas duas receitas simples me ajudam a economizar bastante tempo no dia a dia.

Por fim, o depurador pode ser usado em conjunto com `%run`. Ao executar um script com `%run -d`, você será levado diretamente para dentro do depurador, e estará pronto para definir qualquer breakpoint e iniciar o script:

```
In [1]: %run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb>
```

Adicionar `-b` com um número de linha inicia o depurador com um breakpoint já definido:

```
In [2]: %run -d -b2 examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:2
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py(2)works_fine()
```

```
      1 def works_fine():
1---> 2 a = 5
      3 b = 6

ipdb>
```

Medindo o tempo de execução de um código: %time e %timeit

Para aplicações de análise de dados de larga escala ou de execução prolongada, talvez você queira medir o tempo de execução dos vários componentes, de instruções individuais ou de chamadas de função. É possível gerar um relatório informando quais funções estão consumindo a maior parte do tempo em um processo complexo. Felizmente o IPython permite que essas informações sejam obtidas com facilidade enquanto você estiver desenvolvendo e testando o seu código.

Medir o tempo consumido por um código manualmente usando o módulo embutido time e suas funções time.clock e time.time com frequência é tedioso e repetitivo, pois é necessário escrever o mesmo código boilerplate maçante:

```
import time
start = time.time()
for i in range(iterations):
    # algum código a ser executado aqui
elapsed_per = (time.time() - start) / iterations
```

Como essa é uma operação muito comum, o IPython tem duas funções mágicas, %time e %timeit, para automatizar esse processo para você.

%time executa uma instrução uma vez, informando o tempo total de execução. Suponha que tivéssemos uma lista grande de strings e quiséssemos comparar diferentes métodos para selecionar todas as strings que comecem com um prefixo em particular. Eis uma lista simples com 600 mil strings e dois métodos idênticos para selecionar apenas aquelas que comecem com 'foo':

```
# uma lista bem grande de strings
strings = ['foo', 'foobar', 'baz', 'qux',
        'python', 'Guido Van Rossum'] * 100000

method1 = [x for x in strings if x.startswith('foo')]

method2 = [x for x in strings if x[:3] == 'foo']
```

Parece que esses métodos deveriam ter aproximadamente o mesmo desempenho, não é mesmo? Podemos verificar ao certo usando %time:

```
In [561]: %time method1 = [x for x in strings if x.startswith('foo')]
CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 s
Wall time: 0.19 s

In [562]: %time method2 = [x for x in strings if x[:3] == 'foo']
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
Wall time: 0.09 s
```

O Wall time (abreviatura de “wall-clock time”, isto é, o tempo decorrido) é o principal número de interesse. Desse modo, parece que o primeiro método demora mais do que o dobro do tempo, mas essa não é uma medida muito exata. Se você experimentar usar %time nessas instruções várias vezes, verá que os resultados, de certo modo, variam. Para obter uma medida mais exata, utilize a função mágica %timeit. Dada uma instrução arbitrária, essa função tem um método heurístico para executar uma instrução várias vezes e gerar um tempo de execução médio mais exato:

```
In [563]: %timeit [x for x in strings if x.startswith('foo')]
10 loops, best of 3: 159 ms per loop

In [564]: %timeit [x for x in strings if x[:3] == 'foo']
10 loops, best of 3: 59.3 ms per loop
```

Esse exemplo aparentemente inócuo mostra que vale a pena entender as características de desempenho da biblioteca-padrão de Python, do NumPy, do pandas e de outras bibliotecas usadas neste livro. Em aplicações de análise de dados de larga escala, esses milissegundos começarão a se somar!

%timeit é particularmente conveniente para analisar instruções e funções com tempos de execução muito rápidos, mesmo no nível de microssegundos (milionésimos de um segundo) ou de nanossegundos (bilionésimos de um segundo). Podem parecer intervalos de tempo insignificantes, mas é claro que uma função que demore 20 microssegundos, chamada 1 milhão de vezes, demorará 15 segundos a mais do que uma função que demore 5 microssegundos. No exemplo anterior, poderíamos comparar de modo extremamente direto as duas operações de string a fim de compreender as suas características de desempenho:

```
In [565]: x = 'foobar'

In [566]: y = 'foo'

In [567]: %timeit x.startswith(y)
1000000 loops, best of 3: 267 ns per loop

In [568]: %timeit x[:3] == y
10000000 loops, best of 3: 147 ns per loop
```

Geração básica de perfis: %prun e %run -p

Gerar o perfil de códigos (profiling) está intimamente relacionado a medir o seu tempo de execução, porém a primeira está preocupada em determinar os *lugares* em que o tempo está sendo consumido. A principal ferramenta Python para geração de perfis é o módulo cProfile, que não é específico do IPython. Esse módulo executa um programa ou qualquer bloco de código arbitrário, ao mesmo tempo em que monitora quanto tempo é gasto em cada função.

Um modo comum de usar o cProfile é na linha de comando, executando um programa completo e gerando como saída o tempo agregado por função. Suponha que tivéssemos um script simples que executasse algumas operações de álgebra linear em um laço – calculando os autovalores (eigenvalues) máximos absolutos de uma série de matrizes 100 × 100:

```
import numpy as np
```

```
from numpy.linalg import eigvals

def run_experiment(niter=100):
    K = 100
    results = []
    for _ in xrange(niter):
        mat = np.random.randn(K, K)
        max_eigenvalue = np.abs(eigvals(mat)).max()
        results.append(max_eigenvalue)
    return results
some_results = run_experiment()
print 'Largest one we saw: %s' % np.max(some_results)
```

Podemos executar esse script com cProfile usando o seguinte na linha de comando:

```
python -m cProfile cprof_example.py
```

Se tentar executar esse comando, você verá que a saída estará ordenada por nome de função. Isso faz com que seja um pouco difícil ter uma ideia dos pontos em que a maior parte do tempo está sendo consumida, portanto é muito comum especificar uma *ordem* usando a flag -s:

```
$ python -m cProfile -s cumulative cprof_example.py
Largest one we saw: 11.923204422
    15116 function calls (14927 primitive calls) in 0.720 seconds

Ordered by: cumulative time

ncalls tottime percall cumtime percall filename:lineno(function)
     1 0.001 0.001 0.721 0.721 cprof_example.py:1(<module>)
   100 0.003 0.000 0.586 0.006 linalg.py:702(eigvals)
   200 0.572 0.003 0.572 0.003 {numpy.linalg.lapack_lite.dgeev}
     1 0.002 0.002 0.075 0.075 __init__.py:106(<module>)
   100 0.059 0.001 0.059 0.001 {method 'randn')
     1 0.000 0.000 0.044 0.044 add_newdocs.py:9(<module>)
     2 0.001 0.001 0.037 0.019 __init__.py:1(<module>)
     2 0.003 0.002 0.030 0.015 __init__.py:2(<module>)
     1 0.000 0.000 0.030 0.030 type_check.py:3(<module>)
     1 0.001 0.001 0.021 0.021 __init__.py:15(<module>)
     1 0.013 0.013 0.013 0.013 numeric.py:1(<module>)
```

```
  1 0.000 0.000 0.009 0.009 __init__.py:6(<module>)
  1 0.001 0.001 0.008 0.008 __init__.py:45(<module>)
262 0.005 0.000 0.007 0.000 function_base.py:3178(add_newdoc)
100 0.003 0.000 0.005 0.000 linalg.py:162(_assertFinite)
...
```

Somente as primeiras 15 linhas da saída foram mostradas. É mais fácil ler observando a coluna cumtime para ver quanto tempo total foi gasto *dentro* de cada função. Observe que, se uma função chamar outra função, *o relógio não será interrompido*. cProfile registra o instante inicial e final de cada chamada de função e utiliza esses dados para calcular o tempo.

Além do uso na linha de comando, cProfile também pode ser utilizado em um programa para gerar o perfil de blocos de código arbitrários sem ser necessário executar um novo processo. O IPython tem uma interface conveniente para esse recurso utilizando o comando %prun e a opção -p em %run. %prun aceita as mesmas "opções de linha de comando" de cProfile, mas gerará o perfil de uma instrução Python arbitrária, em vez de fazê-lo para um arquivo *.py* completo:

```
In [4]: %prun -l 7 -s cumulative run_experiment()
         4203 function calls in 0.643 seconds

   Ordered by: cumulative time
   List reduced from 32 to 7 due to restriction <7>

   ncalls tottime percall cumtime percall filename:lineno(function)
        1 0.000 0.000 0.643 0.643 <string>:1(<module>)
        1 0.001 0.001 0.643 0.643 cprof_example.py:4(run_experiment)
      100 0.003 0.000 0.583 0.006 linalg.py:702(eigvals)
      200 0.569 0.003 0.569 0.003 {numpy.linalg.lapack_lite.dgeev}
      100 0.058 0.001 0.058 0.001 {method 'randn'}
      100 0.003 0.000 0.005 0.000 linalg.py:162(_assertFinite)
      200 0.002 0.000 0.002 0.000 {method 'all' of 'numpy.ndarray'}
```

De modo semelhante, chamar %run -p -s cumulative cprof_example.py tem o mesmo efeito que a abordagem da linha de comando, exceto que você jamais terá que sair do IPython.

No notebook Jupyter, a função mágica %%prun (dois sinais de %)

poderá ser usada para gerar o perfil de um bloco de código inteiro. Uma janela separada contendo o perfil como saída será apresentada. Isso pode ser útil para obter respostas possivelmente rápidas a perguntas como: “Por que aquele bloco de código demora tanto para executar?”.

Há outras ferramentas disponíveis que ajudam a deixar os perfis mais fáceis de entender se você estiver usando o IPython ou o Jupyter. Uma delas é o SnakeViz (*https://github.com/jiffyclub/snakeviz/*), que gera uma visualização interativa do resultado do perfil usando d3.js.

Gerando o perfil de uma função linha a linha

Em alguns casos, as informações obtidas com %prun (ou com outro método de geração de perfis baseado no cProfile) podem não nos contar toda a história sobre o tempo de execução de uma função, ou podem ser tão complexas a ponto de os resultados, agregados por nome de função, serem difíceis de interpretar. Para esse caso, há uma pequena biblioteca chamada line_profiler (pode ser obtida por meio do PyPI ou com uma das ferramentas de gerenciamento de pacotes). Ela contém uma extensão para o IPython, que permite executar uma nova função mágica %lprun; essa função gera um perfil linha a linha de uma ou mais funções. Essa extensão pode ser ativada modificando a sua configuração do IPython (consulte a documentação do IPython ou a seção sobre configuração, mais adiante neste capítulo) de modo que a seguinte linha seja incluída:

```
# Uma lista de nomes de módulos com pontos usados como extensões do
# IPython, a ser carregada.
c.TerminalIPythonApp.extensions = ['line_profiler']
```

Você também pode executar este comando:

```
%load_ext line_profiler
```

line_profiler pode ser usado em um programa (veja a documentação completa), mas talvez seja mais eficaz quando utilizada interativamente no IPython. Suponha que tivéssemos um módulo

prof_mod com o código a seguir executando algumas operações de array NumPy:

```
from numpy.random import randn

def add_and_sum(x, y):
    added = x + y
    summed = added.sum(axis=1)
    return summed

def call_function():
    x = randn(1000, 1000)
    y = randn(1000, 1000)
    return add_and_sum(x, y)
```

Se quisermos compreender o desempenho da função add_and_sum , %prun nos dará o seguinte:

```
In [569]: %run prof_mod

In [570]: x = randn(3000, 3000)

In [571]: y = randn(3000, 3000)

In [572]: %prun add_and_sum(x, y)
         4 function calls in 0.049 seconds
   Ordered by: internal time
   ncalls tottime percall cumtime percall filename:lineno(function)
        1 0.036 0.036 0.046 0.046 prof_mod.py:3(add_and_sum)
        1 0.009 0.009 0.009 0.009 {method 'sum' of 'numpy.ndarray'}
        1 0.003 0.003 0.049 0.049 <string>:1(<module>)
```

Esses dados não são particularmente esclarecedores. Com a extensão line_profiler do IPython ativada, um novo comando %lprun estará disponível. A única diferença no uso é que devemos instruir o %lprun acerca da função ou das funções para as quais queremos gerar os perfis. Eis a sintaxe geral:

```
%lprun -f func1 -f func2 instrução_para_gerar_o_perfil
```

Nesse caso, queremos gerar o perfil de add_and_sum, portanto executamos:

```
In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s
Line # Hits Time Per Hit % Time Line Contents
==============================================================
    3 def add_and_sum(x, y):
    4 1 36510 36510.0 79.5 added = x + y
    5 1 9425 9425.0 20.5 summed = added.sum(axis=1)
    6 1 1 1.0 0.0 return summed
```

Esses dados podem ser muito mais fáceis de serem interpretados. Nesse caso, geramos o perfil da mesma função que utilizamos na instrução. Observando o código do módulo anterior, poderíamos chamar `call_function` e gerar o seu perfil, assim como o perfil de `add_and_sum`, obtendo, assim, um quadro geral do desempenho do código:

```
In [574]: %lprun -f add_and_sum -f call_function call_function()
Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.005526 s
Line # Hits Time Per Hit % Time Line Contents
==============================================================
    3 def add_and_sum(x, y):
    4 1 4375 4375.0 79.2 added = x + y
    5 1 1149 1149.0 20.8 summed = added.sum(axis=1)
    6 1 2 2.0 0.0 return summed
File: prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s
Line # Hits Time Per Hit % Time Line Contents
==============================================================
    8 def call_function():
    9 1 57169 57169.0 47.2 x = randn(1000, 1000)
   10 1 58304 58304.0 48.2 y = randn(1000, 1000)
```

11 1 5543 5543.0 4.6 return add_and_sum(x, y)

Como regra geral, tenho a tendência a dar preferência para %prun (cProfile) para gerar um perfil “macro” e usar %lprun (line_profiler) para um perfil “micro”. Vale a pena ter uma boa compreensão das duas ferramentas.

 O motivo pelo qual você deve especificar explicitamente os nomes das funções para as quais você quer gerar o perfil com %lprun é que o overhead de “monitorar” o tempo de execução de cada linha é substancial. Monitorar funções que não sejam de seu interesse tem o potencial de alterar significativamente o resultado do perfil.

B.4 Dicas para um desenvolvimento de código produtivo usando o IPython

Escrever código de uma maneira que facilite o desenvolvimento, a depuração e, em última análise, o *uso* interativo pode ser uma mudança de paradigma para muitos usuários. Há detalhes de procedimentos, como recarregar o código, que talvez exijam alguns ajustes, assim como preocupações com o estilo da programação.

Desse modo, implementar a maior parte das estratégias descritas nesta seção é mais uma arte do que uma ciência, e exigirá alguns experimentos de sua parte para determinar qual é o modo mais eficaz de escrever um código Python para você. Em última instância, você deve estruturar o seu código de modo que facilite usá-lo iterativamente e deve ser capaz de explorar os resultados da execução de um programa ou de uma função com o mínimo de esforço possível. Tenho percebido que é mais fácil trabalhar com um software projetado com o IPython em mente, em comparação com um código criado com o intuito de ser executado apenas como uma aplicação independente de linha de comando. Isso se torna particularmente importante quando algo dá errado e é necessário diagnosticar um erro em um código que você ou outra pessoa possa ter escrito há meses ou há anos.

Recarregando dependências de módulos

Em Python, quando digitamos import some_lib, o código em some_lib é executado, e todas as variáveis, funções e importações definidas aí são armazenadas no namespace recém-criado para o módulo some_lib. Na próxima vez que digitar import some_lib, você obterá uma referência para o namespace existente do módulo. A dificuldade em potencial em um desenvolvimento de código interativo com o IPython surge quando, por exemplo, executamos um script com %run, e esse script depende de algum outro módulo no qual você talvez tenha feito mudanças. Suponha que tivéssemos o código a seguir em *test_script.py*:

```
import some_lib

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

Se fôssemos executar %run test_script.py e então modificássemos *some_lib.py*, na próxima vez que executássemos %run test_script.py, ainda obteríamos a *versão antiga* de some_lib.py por causa do sistema de Python de "carregar módulos somente uma vez". Esse comportamento difere de outros ambientes de análise de dados, como o MATLAB, em que as mudanças de código são automaticamente propagadas.[1] Para lidar com essa situação, temos duas opções. A primeira é usar a função reload do módulo importlib da biblioteca-padrão:

```
import some_lib
import importlib

importlib.reload(some_lib)
```

Isso garante que você obterá uma nova cópia de *some_lib.py* sempre que executar *test_script.py*. Obviamente, se as dependências forem mais profundas, talvez seja um pouco mais complicado inserir usos de reload em todos os lugares. Para esse problema, o IPython tem uma função especial dreload (*não* é uma função mágica) para uma

recarga “profunda” (recursiva) dos módulos. Se eu fosse executar *some_lib.py* e então digitasse dreload(some_lib), ele tentaria recarregar some_lib, assim como todas as suas dependências. Isso não funcionará em todos os casos, infelizmente, mas, se funcionar, será melhor do que ter que reiniciar o IPython.

Dicas para design de código

Não há nenhuma receita simples para isso, mas apresentarei a seguir alguns princípios gerais que acho eficazes em meu próprio trabalho.

Mantenha os objetos e dados relevantes ativos

Não é incomum ver um programa escrito para a linha de comando com uma estrutura, de certo modo, semelhante ao exemplo trivial a seguir:

```
from my_functions import g

def f(x, y):
    return g(x + y)

def main():
    x = 6
    y = 7.5
    result = x + y

if __name__ == '__main__':
    main()
```

Você percebe o que poderia dar errado se executássemos esse programa no IPython? Depois de executado, nenhum dos resultados ou objetos definidos na função main serão acessíveis no shell IPython. Uma melhor maneira é fazer com que qualquer código que esteja em main execute diretamente no namespace global do módulo (ou no bloco if __name__ == '__main__':, se você quiser que o módulo também seja importado). Desse modo, quando executar o código com %run, será possível ver todas as variáveis definidas em main.

Isso é equivalente a definir variáveis de nível superior em células no notebook Jupyter.

Plano é melhor que aninhado

Códigos profundamente aninhados me fazem pensar nas várias camadas de uma cebola. Ao testar ou depurar uma função, quantas camadas da cebola devemos descascar para alcançar o código de nosso interesse? A ideia de que “plano é melhor que aninhado” faz parte do Zen de Python e aplica-se, de modo geral, ao desenvolvimento de código para uso interativo também. Deixar as funções e as classes o mais desacoplado e modular possível faz com que seja mais fácil testá-las (se você estiver escrevendo testes de unidade), depurá-las e usá-las interativamente.

Supere o medo de ter arquivos maiores

Se você vem de uma experiência anterior com Java (ou com outra linguagem desse tipo), talvez tenham lhe dito para manter seus arquivos pequenos. Em muitas linguagens, esse é um conselho enfático; um tamanho grande geralmente é um “code smell” ruim, sinalizando a necessidade de uma refatoração ou de uma reorganização. No entanto, ao desenvolver código usando o IPython, trabalhar com dez arquivos pequenos, porém interconectados (por exemplo, com menos de 100 linhas cada), provavelmente vai lhe causar mais dores de cabeça, em geral, do que trabalhar com dois ou três arquivos maiores. Menos arquivos significa menos módulos para recarregar e também menos saltos entre os arquivos enquanto você estiver editando-os. Acho que manter módulos maiores, cada um com uma coesão *interna* alta, é muito mais conveniente e pythônico. Depois de iterar em direção a uma solução, às vezes fará sentido refatorar arquivos maiores em arquivos menores.

Obviamente, não sou a favor de levar esse argumento ao extremo, que seria colocar todo o seu código em um único arquivo monstruoso. Determinar uma estrutura de módulos e pacotes

sensata e intuitiva para uma base de código grande em geral exige um pouco de trabalho, mas é particularmente importante para trabalhar de forma apropriada em equipes. Cada módulo deve ser internamente coeso, e o local em que as funções e as classes responsáveis por cada uma das funcionalidades se encontram deve ser o mais óbvio possível.

B.5 Recursos avançados do IPython

Fazer uso completo do sistema IPython pode levar você a escrever o seu código de modo um pouco diferente, ou a explorar a configuração.

Deixando suas próprias classes mais apropriadas ao IPython

O IPython faz todo tipo de esforço possível para exibir uma representação em string mais apropriada ao console, para qualquer objeto que você inspecionar. Para muitos objetos, como dicionários, listas e tuplas, o módulo embutido `pprint` será usado para uma formatação elegante. Em classes definidas pelo usuário, porém, você terá que gerar a saída de string desejada por conta própria. Suponha que tivéssemos a classe simples a seguir:

```
class Message:
    def __init__(self, msg):
        self.msg = msg
```

Se você escreveu esse código, ficaria desapontado ao descobrir que a saída default para a sua classe não é muito elegante:

```
In [576]: x = Message('I have a secret')

In [577]: x
Out[577]: <__main__.Message instance at 0x60ebbd8>
```

O IPython toma a string devolvida pelo método mágico `__repr__` (executando `output = repr(obj)`) e exibe-a no console. Assim, podemos adicionar um método `__repr__` simples à classe anterior para obter

uma saída mais conveniente:

```
class Message:
    def __init__(self, msg):
        self.msg = msg

    def __repr__(self):
        return 'Message: %s' % self.msg
In [579]: x = Message('I have a secret')

In [580]: x
Out[580]: Message: I have a secret
```

Perfis e configuração

A maioria dos aspectos quanto à aparência (cores, prompt, espaçamento entre as linhas etc.) e o comportamento dos ambientes do IPython e do Jupyter são configuráveis por meio de um sistema abrangente de configuração. Eis algumas ações que podem ser executadas por meio de configuração:

- alterar o esquema de cores;
- alterar a aparência dos prompts de entrada e de saída ou remover a linha em branco após `Out` e antes do próximo prompt `In`;
- executar uma lista arbitrária de instruções Python (por exemplo, importações que você usa o tempo todo ou qualquer outra ação que deseja que ocorra sempre que o IPython for iniciado);
- deixar extensões IPython sempre ativas, por exemplo, a função mágica `%lprun` em `line_profiler`;
- ativar extensões do Jupyter;
- definir suas próprias funções mágicas ou aliases do sistema.

Configurações para o shell IPython são especificadas em arquivos *ipython_config.py* especiais, que em geral se encontram no diretório *.ipython/*, no diretório home de seu usuário. A configuração é feita com base em um *perfil* particular. Ao iniciar o IPython usualmente,

por padrão, o perfil default, armazenado em Ao iniciar o IPython usualmente, por padrão, o, armazenado no diretório *profile_default*, será carregado., será carregado. Assim, em meu sistema operacional Linux, o path completo para o meu arquivo de configuração default do IPython é:

```
/home/wesm/.ipython/profile_default/ipython_config.py
```

Para inicializar esse arquivo em seu sistema, execute o seguinte no terminal:

```
ipython profile create
```

Pouparei você dos detalhes atrozes acerca do conteúdo desse arquivo. Felizmente há comentários que descrevem para que serve cada opção de configuração, portanto deixarei que o leitor faça ajustes e personalize esse arquivo. Um recurso adicional útil é que é possível ter Um recurso adicional conveniente é o fato de ser possível ter *vários perfis*.. Suponha que você quisesse ter uma configuração alternativa para o IPython, personalizada para uma aplicação ou um projeto específico. Criar um novo perfil é simples e basta digitar algo como:

```
ipython profile create secret_project
```

Depois de fazer isso, edite os arquivos de configuração no recém-criado diretório *profile_secret_project* e então inicie o IPython assim:

```
$ ipython --profile=secret_project
Python 3.5.1 | packaged by conda-forge | (default, May 20 2016, 05:22:56)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

IPython profile: secret_project
```

Como sempre, a documentação online do IPython é um recurso excelente para saber mais sobre perfis e configurações.

A configuração para o Jupyter funciona de modo um pouco diferente porque você pode usar seus notebooks com linguagens diferentes de Python. Para criar um arquivo de configuração análogo para o Jupyter, execute:

```
jupyter notebook --generate-config
```

Esse comando grava um arquivo de configuração default no arquivo *.jupyter/jupyter_notebook_config.py* em seu diretório home. Depois de editar esse arquivo de modo que ele atenda às suas necessidades, é possível renomeá-lo com um nome de arquivo diferente, assim:

```
$ mv ~/.jupyter/jupyter_notebook_config.py ~/.jupyter/my_custom_config.py
```

Ao iniciar o Jupyter, você poderá então acrescentar o argumento --config:

```
jupyter notebook --config=~/.jupyter/my_custom_config.py
```

B.6 Conclusão

À medida que trabalhar com os códigos de exemplo neste livro e desenvolver suas habilidades como programador Python, incentivo-o a continuar conhecendo os ecossistemas do IPython e do Jupyter. Como esses projetos foram concebidos para auxiliar no aumento da produtividade do usuário, você poderá descobrir ferramentas que possibilitarão executar o seu trabalho mais facilmente, se comparado ao cenário em que você usasse somente a linguagem Python e suas bibliotecas de processamento.

Há também uma variedade de notebooks Jupyter interessantes no site nbviewer (*https://nbviewer.jupyter.org/*).

---

1 Como um módulo ou um pacote pode ser importado em vários lugares diferentes em um programa em particular, o Python faz cache do código de um módulo na primeira vez que ele é importado, em vez de sempre o executar. Caso contrário, a modularidade e uma boa organização do código poderiam causar possíveis ineficiências em uma aplicação.

Sobre o autor

Wes McKinney é desenvolvedor de software e empreendedor, e vive em Nova York. Depois de terminar sua graduação em matemática no MIT em 2007, foi trabalhar na área financeira quantitativa na AQR Capital Management em Greenwich, no estado norte-americano de Connecticut. Frustrado com as ferramentas de análise de dados inadequadas, aprendeu Python e começou a desenvolver o que mais tarde se tornaria o projeto pandas. Atualmente é membro ativo da comunidade de dados de Python, além de defensor do uso dessa linguagem em aplicações de análise de dados, financeiras e de processamento estatístico.

Posteriormente, Wes foi cofundador e CEO da DataPad, cujo patrimônio tecnológico e a equipe foram adquiridos pela Cloudera em 2014. Desde então, passou a se envolver com tecnologia de big data, associando-se aos Project Management Committees (Comitês para Gerenciamento de Projetos) para os projetos Apache Arrow e Apache Parquet na Apache Software Foundation. Em 2016, juntou-se à Two Sigma Investments na cidade de Nova York; nessa empresa, continua trabalhando para tornar a análise de dados mais simples e rápida com o uso de software de código aberto.

Colofão

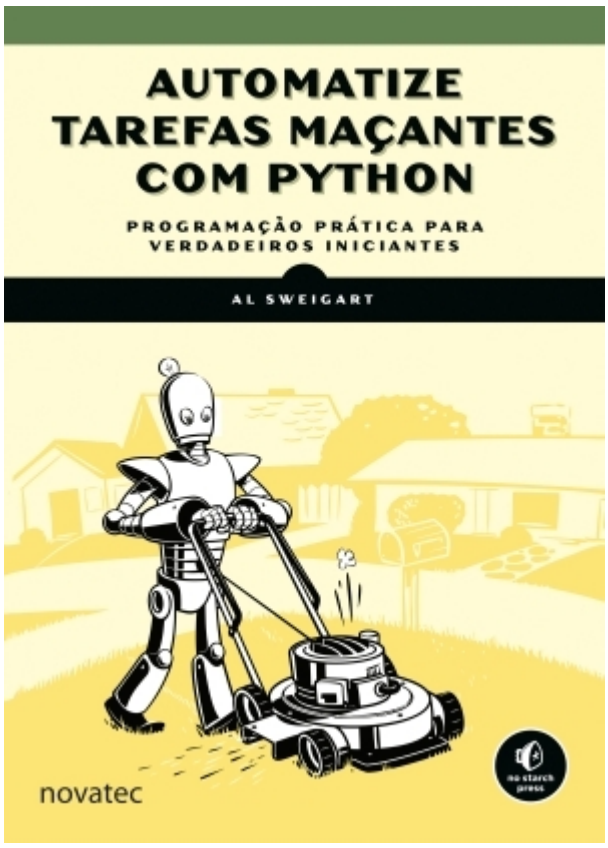
O animal na capa de *Python para Análise de Dados* é um musaranho-arborícola (*Ptilocercus lowii*), com cauda dourada em forma de pena. O musaranho-arborícola é o único de sua espécie do gênero *Ptilocercus* e da família *Ptilocercidae*; todos os demais musaranhos são da família *Tupaiidae*. Os musaranhos-arborícolas são identificados por caudas longas e pelos castanho-avermelhados. Conforme descrito, o musaranho-arborícola tem uma cauda que lembra a pena usada nas antigas canetas. Esses animais são onívoros, alimentando-se principalmente de insetos, frutas, sementes e pequenos vertebrados.

Encontrados predominantemente na Indonésia, na Malásia e na Tailândia, esses mamíferos selvagens são conhecidos pelo consumo crônico de álcool. Descobriu-se que os musaranhos-arborícolas da Malásia passam várias horas consumindo o néctar naturalmente fermentado da palmeira *Eugeissona tristis*, o que equivale a aproximadamente de 10 a 12 copos de vinho com teor alcoólico de 3,8%. Apesar disso, o musaranho-arborícola jamais fica bêbado, principalmente graças à sua impressionante capacidade de quebrar o etanol, o que inclui metabolizar o álcool de um modo não utilizado pelos seres humanos. Esse animal também tem uma característica mais impressionante do que qualquer uma de suas contrapartidas mamíferas, incluindo os seres humanos: a razão entre a massa do cérebro e do corpo.

Apesar do nome desses mamíferos, o musaranho-arborícola não é um verdadeiro musaranho, estando mais intimamente relacionado com os primatas. Por causa desse relacionamento próximo, os musaranhos-arborícolas têm se tornado uma alternativa aos primatas em experimentos médicos para miopia, estresse

psicossocial e hepatite.

A imagem da capa foi retirada do livro *Cassell's Natural History*.


AUTOMATIZE
TAREFAS MAÇANTES
COM PYTHON
PROGRAMAÇÃO PRÁTICA PARA
VERDADEIROS INICIANTES
AL SWEIGART
novatec
no starch press

Automatize tarefas maçantes com Python

Sweigart, Al

9788575226087

568 páginas

APRENDA PYTHON. FAÇA O QUE TEM DE SER FEITO. Se você já passou horas renomeando arquivos ou atualizando centenas de células de planilhas, sabe quão maçantes podem ser esses tipos de tarefa. Que tal se você pudesse fazer o seu computador executá-las para você? Com o livro Automatize tarefas maçantes com Python, você aprenderá a usar o Python para criar programas que farão em minutos o que exigiria horas para ser feito manualmente – sem que seja necessário ter qualquer experiência anterior com programação. Após ter dominado o básico sobre programação, você criará programas Python que realizarão proezas úteis e impressionantes de automação sem nenhum esforço: Pesquisar texto em um arquivo ou em vários arquivos. Criar, atualizar, mover e renomear arquivos e pastas. Pesquisar na Web e fazer download de conteúdos online. Atualizar e formatar dados em planilhas Excel de qualquer tamanho. Separar, combinar, adicionar marcas-d'água e criptografar PDFs. Enviar emails para lembretes e notificações textuais. Preencher formulários online. Instruções passo a passo descreverão cada programa e projetos práticos no final de cada capítulo desafiarão

você a aperfeiçoar esses programas e a usar suas habilidades recém-adquiridas para automatizar tarefas semelhantes. Não gaste seu tempo executando tarefas que um macaquinho bem treinado poderia fazer. Mesmo que não tenha jamais escrito uma linha de código, você poderá fazer o seu computador realizar o trabalho pesado. Saiba como em Automatize tarefas maçantes com Python.

O'REILLY

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações nativas de nuvem

Bilgin Ibryam
Roland Huß

novatec

Padrões para Kubernetes

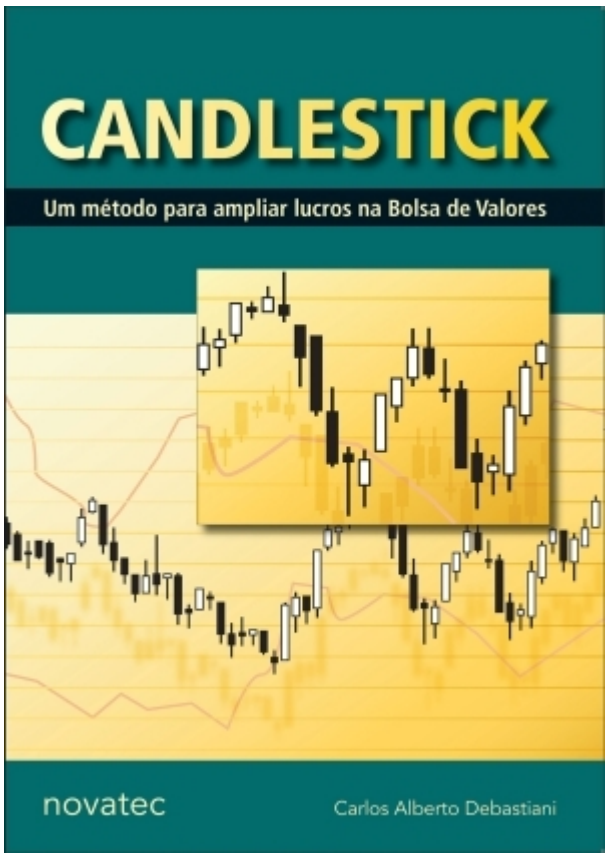
Ibryam, Bilgin

9788575228159

272 páginas

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões: • Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres. • Padrões comportamentais, que exploram conceitos mais

específicos para administrar contêineres e interações com a plataforma. • Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. • Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. • Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).


CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

Compre agora e leia


AVALIANDO
EMPRESAS
INVESTINDO
EM AÇÕES
A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS
novatec
CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

Compre agora e leia


Marcos Abe
MANUAL DE
ANÁLISE
TÉCNICA
ESSÊNCIA E ESTRATÉGIAS AVANÇADAS
TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE
novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.