

O'REILLY®

2ª EDIÇÃO



# Web Scraping com Python

COLETANDO MAIS DADOS NA WEB MODERNA

O'REILLY®  
novatec

Ryan Mitchell

# Web Scraping com Python

**2ª Edição**

**Ryan Mitchell**

**O'REILLY\***  
Novatec

Authorized Portuguese translation of the English edition of titled Web Scraping with Python, 2E, ISBN 9781491985571 © 2018 Ryan Mitchell. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Tradução em português autorizada da edição em inglês da obra Web Scraping with Python, 2E, ISBN 9781491985571 © 2018 Ryan Mitchell. 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.

© Novatec Editora Ltda. [2019].

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Tradução: Lúcia A. Kinoshita

Revisão gramatical: Tássia Carvalho

Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-734-3

Histórico de edições impressas:

Março/2019 Segunda edição

Agosto/2016 Primeira reimpressão

Agosto/2015 Primeira edição (ISBN: 978-85-7522-447-2)

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](mailto:novatec@novatec.com.br)

Site: [www.novatec.com.br](http://www.novatec.com.br)

Twitter: [twitter.com/novateceditora](https://twitter.com/novateceditora)

Facebook: [facebook.com/novatec](https://facebook.com/novatec)

LinkedIn: [linkedin.com/in/novatec](https://linkedin.com/in/novatec)

# Sumário

## [Prefácio](#)

## **[Parte I ■ Construindo scrapers](#)**

### **[Capítulo 1 ■ Seu primeiro web scraper](#)**

[Conectando](#)

[Introdução ao BeautifulSoup](#)

[Instalando o BeautifulSoup](#)

[Executando o BeautifulSoup](#)

[Conectando-se de forma confiável e tratando exceções](#)

### **[Capítulo 2 ■ Parsing de HTML avançado](#)**

[Nem sempre um martelo é necessário](#)

[Outras utilidades do BeautifulSoup](#)

[find\(\) e find\\_all\(\) com o BeautifulSoup](#)

[Outros objetos do BeautifulSoup](#)

[Navegando em árvores](#)

[Expressões regulares](#)

[Expressões regulares e o BeautifulSoup](#)

[Acessando atributos](#)

[Expressões lambda](#)

### **[Capítulo 3 ■ Escrevendo web crawlers](#)**

[Percorrendo um único domínio](#)

[Rastreamento de um site completo](#)

[Coletando dados de um site completo](#)

[Rastreamento pela internet](#)

### **[Capítulo 4 ■ Modelos de web crawling](#)**

[Planejando e definindo objetos](#)

[Lidando com diferentes layouts de sites](#)

[Estruturando os crawlers](#)

[Rastreamento de sites por meio de pesquisa](#)

[Rastreamento de sites por meio de links](#)

[Rastreamento de vários tipos de páginas](#)

[Pensando nos modelos de web crawlers](#)

### **[Capítulo 5 ■ Scrapy](#)**

[Instalando o Scrapy](#)  
[Escrevendo um scraper simples](#)  
[Spidering com regras](#)  
[Criando itens](#)  
[Apresentando itens](#)  
[Pipeline de itens](#)  
[Fazendo log com o Scrapy](#)  
[Outros recursos](#)

## **Capítulo 6 ■ Armazenando dados**

[Arquivos de mídia](#)  
[Armazenando dados no formato CSV](#)  
[MySQL](#)  
[Instalando o MySQL](#)  
[Alguns comandos básicos](#)  
[Integração com Python](#)  
[Técnicas de banco de dados e boas práticas](#)  
[“Six Degrees” no MySQL](#)  
[Email](#)

## **Parte II ■ Coleta de dados avançada**

### **Capítulo 7 ■ Lendo documentos**

[Codificação de documentos](#)  
[Texto](#)  
[Codificação de texto e a internet global](#)  
[CSV](#)  
[Lendo arquivos CSV](#)  
[PDF](#)  
[Microsoft Word e .docx](#)

### **Capítulo 8 ■ Limpando dados sujos**

[Código para limpeza de dados](#)  
[Normalização de dados](#)  
[Limpeza dos dados após a coleta](#)  
[OpenRefine](#)

### **Capítulo 9 ■ Lendo e escrevendo em idiomas naturais**

[Resumindo dados](#)  
[Modelos de Markov](#)  
[Six Degrees of Wikipedia: conclusão](#)  
[Natural Language Toolkit](#)  
[Instalação e configuração](#)  
[Análise estatística com o NLTK](#)  
[Análise lexicográfica com o NLTK](#)

[Recursos adicionais](#)

## **[Capítulo 10](#) ■ [Rastreamento de formulários e logins](#)**

[Biblioteca Python Requests](#)

[Submetendo um formulário básico](#)

[Botões de rádio, caixas de seleção e outras entradas](#)

[Submetendo arquivos e imagens](#)

[Lidando com logins e cookies](#)

[Autenticação de acesso básica do HTTP](#)

[Outros problemas de formulário](#)

## **[Capítulo 11](#) ■ [Scraping de JavaScript](#)**

[Introdução rápida ao JavaScript](#)

[Bibliotecas JavaScript comuns](#)

[Ajax e HTML dinâmico](#)

[Executando JavaScript em Python com o Selenium](#)

[Webdrivers adicionais do Selenium](#)

[Lidando com redirecionamentos](#)

[Última observação sobre o JavaScript](#)

## **[Capítulo 12](#) ■ [Rastreamento por meio de APIs](#)**

[Introdução rápida às APIs](#)

[Métodos HTTP e APIs](#)

[Mais sobre respostas de APIs<sup>o</sup>](#)

[Parsing de JSON](#)

[APIs não documentadas](#)

[Encontrando APIs não documentadas](#)

[Documentando APIs não documentadas](#)

[Encontrando e documentando APIs de modo automático](#)

[Combinando APIs com outras fontes de dados](#)

[Mais sobre APIs](#)

## **[Capítulo 13](#) ■ [Processamento de imagens e reconhecimento de texto](#)**

[Visão geral das bibliotecas](#)

[Pillow](#)

[Tesseract](#)

[NumPy](#)

[Processando textos bem formatados](#)

[Ajustes automáticos nas imagens](#)

[Coletando texto de imagens em sites](#)

[Lendo CAPTCHAs e treinando o Tesseract](#)

[Treinando o Tesseract](#)

[Lendo CAPTCHAs e enviando soluções](#)

## **[Capítulo 14](#) ■ [Evitando armadilhas no scraping](#)**

[Uma observação sobre ética](#)  
[Parecendo um ser humano](#)  
[Ajuste seus cabeçalhos](#)  
[Lidando com cookies em JavaScript](#)  
[Tempo é tudo](#)  
[Recursos de segurança comuns em formulários](#)  
[Valores de campos de entrada ocultos](#)  
[Evitando honeypots](#)  
[Lista de verificação para parecer um ser humano](#)

## **[Capítulo 15](#) ■ [Testando seu site com scrapers](#)**

[Introdução aos testes](#)  
[O que são testes de unidade?](#)  
[Módulo unittest de Python](#)  
[Testando a Wikipédia](#)  
[Testando com o Selenium](#)  
[Interagindo com o site  
unittest ou Selenium?](#)

## **[Capítulo 16](#) ■ [Web Crawling em paralelo](#)**

[Processos versus threads](#)  
[Crawling com várias threads](#)  
[Condições de concorrência e filas](#)  
[Módulo threading](#)  
[Rastreamento com multiprocesso](#)  
[Rastreamento da Wikipédia com multiprocesso](#)  
[Comunicação entre processos](#)  
[Rastreamento com multiprocesso – outra abordagem](#)

## **[Capítulo 17](#) ■ [Fazendo scraping remotamente](#)**

[Por que usar servidores remotos?](#)  
[Evitando o bloqueio de endereços IP](#)  
[Portabilidade e extensibilidade](#)  
[Tor](#)  
[PySocks](#)  
[Hospedagem remota](#)  
[Executando de uma conta que hospeda sites](#)  
[Executando a partir da nuvem](#)  
[Recursos adicionais](#)

## **[Capítulo 18](#) ■ [Aspectos legais e éticos do web scraping](#)**

[Marcas registradas, direitos autorais, patentes, oh, céus!](#)  
[Lei de direitos autorais](#)  
[Invasão de bens móveis](#)  
[Lei de Fraude e Abuso de Computadores](#)  
[robots.txt e Termos de Serviço](#)

[Três web scrapers](#)

[eBay versus Bidder's Edge e transgressão a bens móveis](#)

[Estados Unidos versus Auernheimer e a Lei de Fraude e Abuso de Computadores](#)

[Field versus Google: direitos autorais e robots.txt](#)

[Seguindo em frente](#)

# Prefácio

Para aqueles que ainda não desenvolveram a habilidade para a programação de computadores, esta pode parecer um tipo de magia. Se a programação é mágica, o *web scraping* é a prática dessa magia: é a aplicação da mágica para fazer proezas particularmente impressionantes e úteis – embora, de forma surpreendente, sem exigir muito esforço.

Nos anos que venho trabalhando como engenheiro de software, percebo que há poucas práticas de programação que deixam tanto programadores quanto os leigos tão empolgados como o web scraping. A capacidade de escrever um bot simples, que colete dados e faça seu streaming para um terminal ou os armazene em um banco de dados, embora não seja uma tarefa difícil, jamais deixa de proporcionar certa emoção e um senso de possibilidades, não importa quantas vezes você já tenha feito isso antes.

Infelizmente, quando falo de web scraping a outros programadores, há muito equívoco e confusão a respeito dessa prática. Algumas pessoas não sabem ao certo se ela é uma prática legal (sim, é), ou como cuidar de problemas como páginas que façam uso intenso de JavaScript ou que exijam logins. Muitos se sentem confusos para iniciar um projeto grande de web scraping, ou até mesmo para saber onde encontrar os dados que estão procurando. Este livro procura colocar um ponto final em muitas dessas dúvidas comuns e nos conceitos equivocados sobre web scraping, ao mesmo tempo que serve um guia abrangente para as tarefas mais comuns envolvendo essa atividade.

O web scraping é um campo diversificado, que muda com rapidez, e procuro apresentar tanto os conceitos genéricos quanto exemplos concretos que englobem praticamente qualquer tipo de projeto de coleta de dados com os quais provavelmente você vai deparar. Ao longo do livro, exemplos de códigos serão apresentados para demonstrar esses conceitos e permitir a você testá-los. Os próprios exemplos podem ser usados e modificados com ou sem citação (embora um reconhecimento seja sempre apreciado). Todos os códigos de exemplo estão disponíveis no [GitHub](http://www.pythonscraping.com/code/) (<http://www.pythonscraping.com/code/>) para visualização e download.

## O que é web scraping?

A coleta automatizada de dados da internet é quase tão antiga quanto a própria internet. Embora *web scraping* não seja um termo novo, no passado, a prática era mais conhecida como *screen scraping*, *data mining*, *web harvesting* ou variações similares. Atualmente, o consenso geral parece favorecer o termo *web scraping*, portanto esse é o termo que uso neste livro, embora também possa me referir aos programas que especificamente percorram várias páginas como *web crawlers*, ou aos próprios programas de web scraping como *bots*.

Teoricamente, web scraping é a prática de coletar dados por qualquer meio que não seja um programa interagindo com uma API (ou, obviamente, por um ser humano usando um navegador web). Isso é comumente feito escrevendo um programa automatizado que consulta um servidor web, requisita dados (em geral, na forma de HTML e de outros arquivos que compõem as páginas web) e então faz parse desses dados para extrair as informações necessárias.

Na prática, o web scraping engloba uma grande variedade de técnicas de programação e de tecnologias, por exemplo, análise de dados, parsing de idiomas naturais e segurança de informação. Como o escopo do campo é muito amplo, este livro descreve o básico sobre web scraping e crawling na *Parte I* e explora tópicos avançados com mais detalhes na *Parte II*. Sugiro que todos os leitores leiam com atenção a primeira parte e explorem as especificidades da segunda parte conforme a necessidade.

## Por que usar web scraping?

Se sua única maneira de acessar a internet for usando um navegador, você está ignorando uma variedade enorme de possibilidades. Apesar de os navegadores serem convenientes para executar JavaScript, exibir imagens e organizar objetos em um formato mais legível aos seres humanos (entre outras tarefas), os web scrapers são ótimos para coletar e processar grandes volumes de dados rapidamente. Em vez de visualizar uma página de cada vez em uma janela pequena de um monitor, você poderá visualizar bancos de dados com informações que se estendem por milhares – ou até mesmo milhões – de páginas, de uma só vez.

Além disso, os web scrapers podem acessar lugares que as ferramentas de

pesquisa tradicionais não conseguem. Uma pesquisa no Google por “voos mais baratos para Boston” resultará em uma grande quantidade de anúncios publicitários e sites populares para busca de voos. O Google sabe apenas o que esses sites dizem em suas páginas de conteúdo, mas não os resultados exatos de várias consultas fornecidas a uma aplicação de busca de voos. No entanto, um web scraper bem desenvolvido pode colocar em um gráfico o custo de um voo para Boston ao longo do tempo para uma variedade de sites e informar qual é o melhor momento para comprar uma passagem.

Você pode estar se perguntando: “Mas as APIs não servem para coletar dados?”. (Se você não tem familiaridade com as APIs, consulte o *Capítulo 12*.) Bem, as APIs podem ser incríveis se você encontrar uma que atenda aos seus propósitos. Elas são projetadas para fornecer um stream conveniente de dados bem formatados, de um programa de computador para outro. É possível encontrar uma API para vários tipos de dados que você queira usar, por exemplo, postagens do Twitter ou páginas da Wikipédia. Em geral, é preferível usar uma API (caso haja uma), em vez de implementar um bot para obter os mesmos dados. Contudo, talvez não haja uma API ou, quem sabe, ela não seja conveniente para seus propósitos, por diversos motivos:

- Você está coletando conjuntos relativamente pequenos e finitos de dados de um conjunto grande de sites e não há uma API coesa.
- O conjunto de dados que você quer é razoavelmente pequeno ou os dados são incomuns, e o criador não considerou que merecessem uma API.
- A fonte de dados não tem a infraestrutura nem a habilidade técnica para criar uma API.
- Os dados são valiosos e/ou protegidos, e não há intenção de que sejam amplamente disseminados.

Mesmo que *haja* uma API, os limites para volume e taxa de requisição dos dados, ou os tipos e formatos disponibilizados, podem ser insuficientes para seus propósitos.

É aí que o web scraping entra em cena. Com poucas exceções, se você puder visualizar os dados em seu navegador, será possível acessá-los por meio de um script Python. Se for possível acessá-los com um script, você poderá armazená-los em um banco de dados. E, se puder armazená-los em

um banco de dados, poderá fazer praticamente de tudo com esses dados. Sem dúvida, há muitas aplicações extremamente práticas decorrentes de ter acesso a um volume praticamente ilimitado de dados: previsões de mercado, tradução de idiomas por computador e até mesmo diagnósticos médicos têm se beneficiado muito com a capacidade de obter e analisar dados de sites de notícias, textos traduzidos e fóruns da área de saúde, respectivamente.

Até mesmo no mundo da arte, o web scraping tem aberto novas fronteiras para a criação. O projeto *“We Feel Fine”* (<http://wefeelfine.org/>) de 2006, de Jonathan Harris e Sep Kamvar, coletou dados de vários sites de blogs em inglês em busca de expressões como “I feel” (Sinto) ou “I am feeling” (Estou me sentindo). Isso resultou em uma visualização de dados populares, que descrevia como o mundo estava se sentindo dia a dia e minuto a minuto.

Independentemente de sua área de atuação, o web scraping quase sempre oferece uma forma de orientar práticas de negócios com mais eficiência, melhorar a produtividade ou até mesmo dar origem a uma área totalmente nova.

## **Sobre este livro**

Este livro foi concebido para servir não só como uma introdução ao web scraping, mas também como um guia abrangente para coletar, transformar e usar dados de fontes independentes. Embora o livro use a linguagem de programação Python e descreva muitos aspectos básicos dessa linguagem, não deve ser utilizado como uma introdução à linguagem.

Se você não conhece absolutamente nada de Python, este livro talvez seja um pouco desafiador. Por favor, não o use como um texto introdutório para Python. Apesar disso, tentei manter todos os conceitos e os exemplos de código em um nível de programação Python de iniciante para intermediário a fim de que o conteúdo fosse acessível a uma grande gama de leitores. Para isso, há explicações ocasionais sobre tópicos mais avançados de programação Python e de ciência da computação em geral, quando forem apropriadas. Se você é um leitor de nível mais avançado, sinta-se à vontade para passar rapidamente por essas partes!

Se estiver procurando um recurso mais completo para conhecer Python, o livro *Introducing Python* de Bill Lubanovic (O’Reilly) é um bom guia,

apesar de longo. Para as pessoas que conseguem manter a atenção apenas por períodos mais curtos, a série de vídeos *Introduction to Python* de Jessica McKellar (O'Reilly) é um ótimo recurso. Também gostei do livro *Pense em Python* de um ex-professor meu, Allen Downey (Novatec). Esse último, em particular, é ideal para iniciantes em programação e ensina conceitos de ciência da computação e de engenharia de software, junto com a linguagem Python.

Livros técnicos em geral são capazes de manter o foco em uma única linguagem ou tecnologia, mas o web scraping é um assunto relativamente diferente, com práticas que exigem o uso de bancos de dados, servidores web, HTTP, HTML, segurança na internet, processamento de imagens, ciência de dados e outras ferramentas. Este livro procura abordar todos esses assuntos, além de outros, do ponto de vista da “coleta de dados”. A obra não deve ser usada como uma descrição completa para nenhum desses assuntos, mas acredito que eles sejam abordados com detalhes suficientes para você começar a escrever web scrapers!

A *Parte I* aborda o web scraping e o web crawling em detalhes, com bastante foco em um pequeno conjunto de bibliotecas usadas no livro. Essa parte pode ser facilmente utilizada como uma referência abrangente para essas bibliotecas e técnicas (com algumas exceções, quando referências adicionais serão fornecidas). As habilidades ensinadas na primeira parte provavelmente serão úteis a qualquer pessoa que escreva um web scraper, independentemente de seus objetivos específicos ou da aplicação.

A *Parte II* discute assuntos adicionais que o leitor poderá achar úteis ao escrever web scrapers, mas que talvez não sejam úteis para todos os scrapers o tempo todo. Esses assuntos, infelizmente, são amplos demais para serem incluídos de modo apropriado em um único capítulo. Por causa disso, referências frequentes serão feitas a outros recursos para informações adicionais.

A estrutura deste livro permite pular facilmente de um capítulo a outro a fim de encontrar apenas a técnica de web scraping ou a informação que você estiver procurando. Quando um conceito ou uma porção de código estiver baseado em outro mencionado em um capítulo anterior, referenciarei explicitamente a seção na qual ele foi apresentado.

## Convenções usadas no livro

As convenções tipográficas a seguir são usadas neste livro:

### *Itálico*

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

### Largura fixa

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

### **Largura fixa em negrito**

Mostra comandos ou outros textos que devam ser digitados pelo usuário.

### *Largura fixa em itálico*

Mostra texto que tenha de 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

Materiais suplementares (exemplos de código, exercícios etc.) estão disponíveis para download em <https://github.com/REMitchell/python-scraping>.

Este livro está aqui para ajudá-lo a fazer seu trabalho. Se o código de exemplo for útil, você poderá usá-lo 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. No entanto, vender ou distribuir um CD-ROM de exemplos de livros da O'Reilly requer. 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.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: “*Web Scraping with Python*, 2ª edição, de Ryan Mitchell (O’Reilly). Copyright 2018 Ryan Mitchell, 978-1-491-998557-1”.

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

Infelizmente é difícil manter os livros impressos atualizados. Com o web scraping, isso representa um desafio a mais, pois muitas bibliotecas e sites referenciados no livro e dos quais o código muitas vezes depende podem ter sido ocasionalmente modificados, e os exemplos de código poderão falhar ou gerar resultados inesperados. Se você optar por executar esses exemplos, por favor, execute-os a partir do repositório do GitHub em vez de copiá-los diretamente do livro. Eu e os leitores deste livro que decidiram colaborar (incluindo você, talvez) nos esforçaremos ao máximo para manter o repositório atualizado com as modificações e as observações necessárias.

Além dos exemplos de código, comandos de terminal muitas vezes são apresentados para ilustrar a instalação e a execução de um software. Em geral, esses comandos estão voltados para sistemas operacionais baseados em Linux, mas, em geral, são aplicáveis para usuários de Windows com um ambiente Python devidamente configurado e a instalação do pip. Quando não for o caso, apresentarei instruções para todos os principais sistemas operacionais ou fornecerei referências externas para que usuários do Windows executem a tarefa.

## Como entrar em contato

Envie seus comentários e suas dúvidas sobre este livro à editora escrevendo para: [novatec@novatec.com.br](mailto:novatec@novatec.com.br).

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

- Página da edição em português

<https://novatec.com.br/livros/web-scraping-com-python-2ed>

- Página da edição original em inglês

<http://shop.oreilly.com/product/0636920078067.do>

- Página com material suplementar (exemplos de códigos, exercícios etc.).

<https://github.com/REMitchell/python-scraping>

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

## Agradecimentos

Assim como alguns dos melhores produtos surgem de um grande volume de feedback de usuários, este livro jamais poderia ter existido em nenhum formato útil se não fosse a ajuda de muitos colaboradores, torcedores e editores. Agradeço à equipe da O'Reilly e ao seu apoio incrível a esse assunto, de certo modo, nada convencional, aos meus amigos e familiares que me ofereceram conselhos e aceitaram fazer leituras extemporâneas e aos meus colegas de trabalho da HedgeServ, a quem agora, provavelmente, devo muitas horas de trabalho.

Agradeço, em particular, a Allyson MacDonald, Brian Anderson, Miguel Grinberg e Eric VanWyk o feedback, a orientação e, ocasionalmente, o tratamento duro, mas por amor. Muitas seções e exemplos de código foram escritos como resultado direto de suas sugestões inspiradoras.

Agradeço a Yale Specht a paciência ilimitada nos últimos quatro anos e duas edições, dando-me a coragem inicial para levar adiante este projeto, além de oferecer feedback quanto ao estilo durante o processo de escrita. Sem ele, este livro teria sido escrito em metade do tempo, mas não estaria nem perto de ser tão útil.

Por fim, agradeço a Jim Waldo, que realmente deu início a tudo isso muitos anos atrás, quando enviou uma máquina Linux e o livro *The Art and Science of C* a uma adolescente jovem e sugestionável.

# Construindo scrapers

A primeira parte deste livro tem como foco o funcionamento básico do web scraping: como usar Python para requisitar informações de um servidor web, como fazer o tratamento básico da resposta do servidor e como começar a interagir com um site de modo automatizado. No final, você estará navegando pela internet com facilidade, construindo scrapers capazes de passar de um domínio para outro, coletar informações e armazená-las para uso futuro.

Falando francamente, o web scraping é uma área incrível para embarcar, caso você queira uma recompensa enorme por um investimento prévio relativamente baixo. É bem provável que 90% dos projetos de web scraping que você verá sejam baseados nas técnicas usadas nos próximos seis capítulos. Esta seção aborda o que o público em geral (não obstante, com conhecimento técnico) tende a pensar quando pensa em “web scrapers”:

- obter dados HTML de um nome de domínio;
- fazer parse desses dados em busca das informações desejadas;
- armazenar as informações desejadas;
- opcionalmente, passar para outra página e repetir o processo.

Com isso, você terá uma base sólida antes de avançar para projetos mais complexos na *Parte II*. Não se deixe enganar achando que esta primeira seção não seja tão importante quanto alguns dos projetos mais avançados da segunda metade. Você usará quase todas as informações da primeira metade deste livro diariamente quando estiver escrevendo web scrapers!

# Seu primeiro web scraper

Depois de começar a fazer web scraping, você passará a apreciar todas as pequenas tarefas que os navegadores fazem para você. A web, sem uma camada de formatação de HTML, estilização com CSS, execução de JavaScript e renderização de imagens, pode parecer um pouco assustadora à primeira vista; porém, neste capítulo, assim como no próximo, veremos como formatar e interpretar dados sem a ajuda de um navegador.

Este capítulo começa com o básico sobre como enviar uma requisição GET (uma requisição para buscar, ou “obter” (get) o conteúdo de uma página web) a um servidor web a fim de obter uma página específica, ler a saída HTML dessa página e fazer uma extração simples de dados para isolar o conteúdo que você estiver procurando.

## Conectando

Se você ainda não dedicou muito tempo a redes ou segurança de redes, o modo de funcionamento da internet pode parecer um pouco misterioso. Você não vai querer pensar no que a rede faz, exatamente, sempre que abrir um navegador e acessar <http://google.com>, e, hoje em dia, isso não é necessário. Na verdade, eu diria que o fato de as interfaces de computador terem avançado tanto, a ponto de a maioria das pessoas que usam a internet não ter a mínima ideia de como ela funciona, é incrível.

No entanto, o web scraping exige abrir mão de parte dessa camada protetora da interface – não só no nível do navegador (o modo como ele interpreta todos esses códigos HTML, CSS e JavaScript), mas também, ocasionalmente, no nível da conexão de rede.

Para ter uma ideia da infraestrutura necessária para que as informações cheguem até o seu navegador, vamos usar o exemplo a seguir. Alice tem um servidor web. Bob usa um computador desktop, que está tentando se conectar com o servidor de Alice. Quando uma máquina quer conversar com outra, algo como a troca a seguir ocorre:

1. O computador de Bob envia um stream de bits 1s e 0s, representados como tensões altas e baixas em um fio. Esses bits compõem uma informação, com um cabeçalho (header) e um corpo (body). O cabeçalho contém um destino imediato, que é o endereço MAC do roteador local de Bob, e um destino final, que é o endereço IP de Alice. O corpo contém a requisição para a aplicação de servidor de Alice.
2. O roteador local de Bob recebe todos esses 1s e 0s e os interpreta como um pacote, do endereço MAC de Bob, destinado ao endereço IP de Alice. O roteador carimba o seu próprio endereço IP no pacote como o endereço IP de “origem” (from) e o envia pela internet.
3. O pacote de Bob passa por vários servidores intermediários, que o direcionam pelo caminho físico/conectado correto até o servidor de Alice.
4. O servidor de Alice recebe o pacote em seu endereço IP.
5. O servidor de Alice lê a porta de destino do pacote no cabeçalho, passando-o para a aplicação apropriada – a aplicação do servidor web. (A porta de destino do pacote é quase sempre a porta 80 para aplicações web; podemos pensar nela como o número de um apartamento para dados de pacote, enquanto o endereço IP seria como o endereço da rua.)
6. A aplicação de servidor web recebe um stream de dados do processador do servidor. Esses dados dizem algo como:
  - Esta é uma requisição GET.
  - O arquivo a seguir está sendo requisitado: *index.html*.
7. O servidor web localiza o arquivo HTML correto, insere esse arquivo em um novo pacote a ser enviado para Bob e o envia por meio de seu roteador local, para que seja transportado de volta para o computador de Bob, pelo mesmo processo.

E voilà! Temos a Internet.

Então, nessa troca, em que ponto o navegador web entra em cena? Absolutamente, em nenhum lugar. Na verdade, os navegadores são uma invenção relativamente recente na história da internet, considerando que o Nexus foi lançado em 1990.

Sim, o navegador web é uma aplicação útil para criar esses pacotes de informações, dizendo ao seu sistema operacional que os envie e interpretando os dados recebidos na forma de imagens bonitas, áudios, vídeos e texto. Entretanto, um navegador web é somente um código, e um

código pode ser dividido, separado em seus componentes básicos, reescrito, reutilizado, e você pode fazer com que ele aja como você quiser. Um navegador web pode dizer ao processador que envie dados para a aplicação que trata a sua interface sem fio (ou com fio), mas você pode fazer o mesmo em Python usando apenas três linhas de código:

```
from urllib.request import urlopen

html = urlopen('http://pythonscraping.com/pages/page1.html')
print(html.read())
```

Para executar esse código, use o [notebook iPython do Capítulo 1](https://github.com/REMitchell/python-scraping/blob/master/Chapter01_BeginningToScrape.ipynb) ([https://github.com/REMitchell/python-scraping/blob/master/Chapter01\\_BeginningToScrape.ipynb](https://github.com/REMitchell/python-scraping/blob/master/Chapter01_BeginningToScrape.ipynb)) que está no repositório do GitHub, ou salve-o localmente como *scrapetest.py* e execute-o em seu terminal com o comando a seguir:

```
$ python scrapetest.py
```

Observe que, se você também tiver Python 2.x instalado em seu computador e estiver executando as duas versões de Python lado a lado, talvez seja necessário chamar explicitamente o Python 3.x executando o comando da seguinte maneira:

```
$ python3 scrapetest.py
```

Esse comando exibe o código HTML completo de *page1*, que está no URL <http://pythonscraping.com/pages/page1.html>. Para ser mais exato, ele exibe o arquivo HTML *page1.html*, que se encontra no diretório `<web root>/pages`, no servidor localizado no domínio <http://pythonscraping.com>.

Por que é importante começar a pensar nesses endereços como “arquivos” em vez de “páginas”? A maioria das páginas web modernas tem muitos arquivos de recursos associados a elas. Podem ser arquivos de imagens, arquivos JavaScript, arquivos CSS ou qualquer outro conteúdo associado à página sendo requisitada. Quando um navegador web encontra uma tag como ``, ele sabe que deve fazer outra requisição ao servidor a fim de obter os dados do arquivo *cuteKitten.jpg* e renderizar totalmente a página para o usuário.

É claro que seu script Python não tem a lógica para voltar e requisitar vários arquivos (ainda); ele é capaz de ler apenas o único arquivo HTML que você requisitou diretamente.

```
from urllib.request import urlopen
```

significa o que parece que significa: a instrução olha para o módulo `request`

de Python (que se encontra na biblioteca *urllib*) e importa somente a função `urlopen`.

*urllib* é uma biblioteca-padrão de Python (ou seja, não é necessário instalar nada extra para executar esse exemplo); ela contém funções para requisitar dados da web, tratando cookies e até mesmo modificando metadados como cabeçalhos e o agente de usuário (user agent). Usaremos bastante a *urllib* neste livro, portanto recomendo que você leia a [documentação de Python dessa biblioteca](https://docs.python.org/3/library/urllib.html) (<https://docs.python.org/3/library/urllib.html>).

`urlopen` é usada para abrir um objeto remoto em uma rede e lê-lo. Por ser uma função razoavelmente genérica (é capaz de ler facilmente arquivos HTML, arquivos de imagens ou qualquer outro stream de arquivo), ela será usada com muita frequência neste livro.

## Introdução ao BeautifulSoup

Bela Sopa, tão rica e verde,

À espera em uma terrina quente!

Quem não se derreteria por uma iguaria como essa?

Sopa do jantar, bela Sopa!<sup>1</sup>

A biblioteca *BeautifulSoup*<sup>2</sup> tem o mesmo nome de um poema de Lewis Carroll que aparece no livro *Alice no País das Maravilhas*. Na história, esse poema é declamado por uma personagem chamada Falsa Tartaruga (Mock Turtle) – por si só, é um trocadilho com o nome do popular prato vitoriano chamado Sopa Falsa de Tartaruga (Mock Turtle Soup), que não é feito de tartaruga, mas de vaca).

Assim como o seu homônimo no País das Maravilhas, o *BeautifulSoup* tenta dar sentido ao que não faz sentido: ajuda a formatar e a organizar a web confusa, fazendo correções em um código HTML mal formatado e apresentando objetos Python que podem ser facilmente percorridos, os quais representam estruturas XML.

## Instalando o BeautifulSoup

Como a biblioteca *BeautifulSoup* não é uma biblioteca Python default, ela deve ser instalada. Se você já tem experiência em instalar bibliotecas Python, use seu instalador favorito e vá direto para a próxima seção, “*Executando o BeautifulSoup*”.

Para aqueles que ainda não instalaram bibliotecas Python (ou que precisam de um lembrete), o método genérico a seguir será usado para instalar várias bibliotecas ao longo do livro, portanto você poderá consultar esta seção no futuro.

Usaremos a biblioteca BeautifulSoup 4 (também conhecida como BS4) neste livro. As instruções completas para instalar o BeautifulSoup 4 podem ser encontradas em [Crummy.com](https://www.crummy.com/software/BeautifulSoup/bs4/doc/) (<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>); no entanto, o método básico para Linux é exibido a seguir:

```
$ sudo apt-get install python-bs4
```

Para Macs, execute:

```
$ sudo easy_install pip
```

Esse comando instala o gerenciador de pacotes Python *pip*. Então execute o comando a seguir para instalar a biblioteca:

```
$ pip install beautifulsoup4
```

Novamente, observe que, se você tiver tanto Python 2.x quanto Python 3.x instalados em seu computador, talvez seja necessário chamar `python3` explicitamente:

```
$ python3 myScript.py
```

Certifique-se de usar também esse comando ao instalar pacotes; caso contrário, os pacotes poderão ser instalados para Python 2.x, mas não para Python 3.x:

```
$ sudo python3 setup.py install
```

Se estiver usando `pip`, você também poderá chamar `pip3` para instalar as versões dos pacotes para Python 3.x:

```
$ pip3 install beautifulsoup4
```

Instalar pacotes no Windows é um processo quase idêntico àquele usado em Mac e Linux. Faça download da versão mais recente do BeautifulSoup 4 a partir da *página de download* (<http://www.crummy.com/software/BeautifulSoup/#Download>), vá para o diretório no qual você o descompactou e execute:

```
> python setup.py install
```

É isso! O BeautifulSoup agora será reconhecido como uma biblioteca Python em seu computador. Você pode fazer um teste abrindo um terminal Python e importando a biblioteca:

```
$ python  
> from bs4 import BeautifulSoup
```

A importação deverá ser concluída sem erros.

Além disso, há um [instalador .exe para pip no Windows](https://pypi.org/project/setuptools/) (<https://pypi.org/project/setuptools/>), para que você possa instalar e gerenciar pacotes facilmente:

```
> pip install beautifulsoup4
```

## Mantendo as bibliotecas organizadas em ambientes virtuais

Se você pretende trabalhar com vários projetos Python, ou precisa de uma maneira fácil de empacotar projetos com todas as bibliotecas associadas, ou está preocupado com possíveis conflitos entre bibliotecas instaladas, é possível instalar um ambiente virtual Python a fim de manter tudo separado e simples de administrar.

Ao instalar uma biblioteca Python sem um ambiente virtual, ela será instalada *globalmente*. Em geral, isso exige que você seja um administrador ou execute como root, e que a biblioteca Python esteja disponível para todos os usuários de todos os projetos no computador. Felizmente, criar um ambiente virtual é fácil:

```
$ virtualenv scrapingEnv
```

Esse comando cria um novo ambiente chamado *scrapingEnv*, que deve ser ativado para ser usado:

```
$ cd scrapingEnv/
```

```
$ source bin/activate
```

Depois de ativado, o nome do ambiente será exibido no prompt de comandos para que você lembre que está trabalhando nele no momento. Qualquer biblioteca que você instalar ou qualquer script que executar estarão somente nesse ambiente virtual.

Trabalhando no ambiente *scrapingEnv* recém-criado, você poderá instalar e usar o BeautifulSoup; por exemplo:

```
(scrapingEnv)ryan$ pip install beautifulsoup4
```

```
(scrapingEnv)ryan$ python
```

```
> from bs4 import BeautifulSoup
```

```
>
```

O comando *deactivate* pode ser usado para sair do ambiente; depois disso, não será mais possível acessar nenhuma biblioteca que tenha sido instalada no ambiente virtual:

```
(scrapingEnv)ryan$ deactivate
```

```
ryan$ python
```

```
> from bs4 import BeautifulSoup
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ImportError: No module named 'bs4'
```

Manter todas as bibliotecas separadas por projeto também facilita compactar a pasta completa do ambiente e enviá-la para outra pessoa. Desde que essas pessoas tenham a mesma versão de Python instalada em seus computadores, o código funcionará no ambiente virtual, sem exigir a instalação de nenhuma biblioteca.

Embora eu não dê explicitamente instruções para que você use um ambiente virtual nos exemplos deste livro, lembre-se de que pode utilizar um ambiente desse tipo a qualquer momento, apenas o ativando previamente.

## Executando o BeautifulSoup

O objeto mais comum usado na biblioteca BeautifulSoup, como não

podria deixar de ser, é o objeto `BeautifulSoup`. Vamos observá-lo em ação, modificando o exemplo apresentado no início deste capítulo:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page1.html')
bs = BeautifulSoup(html.read(), 'html.parser')
print(bs.h1)
```

Eis a saída:

```
<h1>An Interesting Title</h1>
```

Note que somente a primeira instância da tag `h1` encontrada na página é devolvida. Por convenção, apenas uma tag `h1` deve ser usada em uma página; contudo, as convenções muitas vezes são desrespeitadas na web, portanto saiba que esse código devolverá somente a primeira instância da tag, e não necessariamente aquela que você está procurando.

Como nos exemplos anteriores de web scraping, a função `urlopen` está sendo importada e a função `html.read()` é chamada para obter o conteúdo HTML da página. Além da string de texto, o `BeautifulSoup` também pode usar diretamente o objeto de arquivo devolvido por `urlopen`, sem precisar chamar `.read()` antes:

```
bs = BeautifulSoup(html, 'html.parser')
```

O conteúdo HTML é então transformado em um objeto `BeautifulSoup` com a seguinte estrutura:

- **html** → `<html><head>...</head><body>...</body></html>`
  - **head** → `<head><title>A Useful Page<title></head>`
  - **title** → `<title>A Useful Page</title>`
  - **body** → `<body><h1>An Int...</h1><div>Lorem ip...</div></body>`
  - **h1** → `<h1>An Interesting Title</h1>`
  - **div** → `<div>Lorem Ipsum dolor...</div>`

Observe que a tag `h1` que você extraiu da página está aninhada a dois níveis de profundidade na estrutura do objeto `BeautifulSoup` (`html` → `body` → `h1`). No entanto, ao buscá-la no objeto, é possível acessar a tag `h1` diretamente:

```
bs.h1
```

Com efeito, qualquer uma das chamadas de função a seguir produziria o mesmo resultado:

```
bs.html.body.h1
```

```
bs.body.h1
bs.html.h1
```

Ao criar um objeto `BeautifulSoup`, dois argumentos são passados:

```
bs = BeautifulSoup(html.read(), 'html.parser')
```

O primeiro é o texto HTML no qual o objeto se baseia, e o segundo especifica o parser que queremos que o `BeautifulSoup` use para criar esse objeto. Na maioria dos casos, o parser escolhido não fará diferença.

`html.parser` é um parser incluído no Python 3, e não exige nenhuma instalação extra para ser usado. Exceto quando outro parser for necessário, usaremos esse parser no livro.

Outro parser conhecido é o `lxml` (<http://lxml.de/parsing.html>). Esse parser pode ser instalado com o pip:

```
$ pip3 install lxml
```

O `lxml` pode ser usado com o `BeautifulSoup` se a string de parser especificada for alterada:

```
bs = BeautifulSoup(html.read(), 'lxml')
```

O `lxml` tem algumas vantagens em relação ao `html.parser`: em geral, ele é melhor para fazer parse de código html “confuso” ou mal formatado. É um parser mais tolerante e corrige problemas como tags sem fechamento, tags indevidamente aninhadas e tags de cabeçalho e de corpo ausentes. De certo modo, é também um parser mais rápido que o `html.parser`, embora a velocidade não seja necessariamente uma vantagem no web scraping, considerando que a própria velocidade da rede quase sempre será o principal gargalo.

Uma das desvantagens do `lxml` é que ele deve ser instalado separadamente e depende de bibliotecas C de terceiros para funcionar. Isso pode causar problemas de portabilidade e de facilidade de uso, em comparação com o `html.parser`.

Outro parser HTML conhecido é o `html5lib`. Assim como o `lxml`, o `html5lib` é um parser extremamente tolerante, com mais iniciativa ainda para corrigir um código HTML com falhas. Também tem dependência externa e é mais lento quando comparado tanto com o `lxml` quanto com o `html.parser`. Apesar disso, essa pode ser uma boa opção caso você esteja trabalhando com sites HTML confusos ou escritos manualmente.

Para usá-lo, faça a sua instalação e passe a string `html5lib` para o objeto `BeautifulSoup`:

```
bs = BeautifulSoup(html.read(), 'html5lib')
```

Espero que essa pequena amostra do *BeautifulSoup* tenha dado a você uma ideia da eficácia e da simplicidade dessa biblioteca. Qualquer informação pode ser virtualmente extraída de qualquer arquivo HTML (ou XML), desde que ela tenha uma tag de identificação ao redor ou próxima a ela. O *Capítulo 2* detalha melhor as chamadas de função mais complexas do BeautifulSoup, além de apresentar as expressões regulares e mostrar como elas podem ser usadas com o BeautifulSoup para extrair informações dos sites.

## Conectando-se de forma confiável e tratando exceções

A web é confusa. Os dados são mal formatados, os sites ficam inativos e tags de fechamento podem estar ausentes. Uma das experiências mais frustrantes em web scraping é ir dormir com um scraper executando, sonhar com todos os dados que estarão em seu banco de dados no dia seguinte – somente para descobrir que o scraper deparou com um erro em algum formato de dados inesperado e interrompeu a execução logo depois que você parou de olhar para a tela. Em situações como essas, pode ser tentador amaldiçoar o nome do desenvolvedor que criou o site (e os dados peculiarmente formatados), mas a pessoa que você de fato deveria acusar é a si mesmo, por não ter previsto a exceção, antes de tudo!

Vamos analisar a primeira linha de nosso scraper, depois das instruções de importação, e descobrir como lidar com qualquer exceção que seja lançada:

```
html = urlopen('http://www.pythonscraping.com/pages/page1.html')
```

Dois erros podem ocorrer nessa linha:

- A página não é encontrada no servidor (ou houve um erro ao obtê-la).
- O servidor não foi encontrado.

Na primeira situação, um erro de HTTP será devolvido. Esse erro pode ser “404 Page Not Found” (página não encontrada), “500 Internal Server Error” (erro interno do servidor), e assim por diante. Em todos esses casos, a função `urlopen` lançará a exceção genérica `HTTPError`. Essa exceção pode ser tratada da seguinte maneira:

```
from urllib.request import urlopen
from urllib.error import HTTPError

try:
    html = urlopen('http://www.pythonscraping.com/pages/page1.html')
except HTTPError as e:
```

```

    print(e)
    # devolve null, executa um break ou algum outro "Plano B"
else:
    # o programa continua. Nota: se você retornar ou executar um break no
    # catch da exceção, não será necessário usar a instrução "else"

```

Se um código de erro HTTP for devolvido, o programa agora exibirá o erro e não executará o resto do programa que está na instrução `else`.

Se o servidor não for encontrado (se, por exemplo, <http://www.pythonscraping.com> estiver fora do ar, ou o URL estiver grafado incorretamente), `urlopen` lançará um `URLError`. Isso quer dizer que não foi possível acessar nenhum servidor e, como o servidor remoto é responsável por devolver códigos de status HTTP, um `HTTPError` não pôde ser lançado, e o erro `URLError`, mais grave, deve ser capturado. Podemos acrescentar uma verificação para saber se é isso que está acontecendo:

```

from urllib.request import urlopen
from urllib.error import HTTPError
from urllib.error import URLError

try:
    html = urlopen('https://pythonscrapingthisurldoesnotexist.com')
except HTTPError as e:
    print(e)
except URLError as e:
    print('The server could not be found!')
else:
    print('It Worked!')

```

É claro que, se a página for obtida do servidor com sucesso, há ainda o problema de o conteúdo da página não ser exatamente o que você esperava. Sempre que acessar uma tag em um objeto `BeautifulSoup`, acrescentar uma verificação para garantir que ela realmente exista é uma atitude inteligente. Se você tentar acessar uma tag que não existe, o `BeautifulSoup` devolverá um objeto `None`. O problema é que tentar acessar uma tag em um objeto `None` resultará no lançamento de um `AttributeError`.

A linha a seguir (em que `nonExistentTag` é uma tag inventada, e não o nome de uma verdadeira função do `BeautifulSoup`):

```
print(bs.nonExistentTag)
```

devolve um objeto `None`. É perfeitamente razoável conferir e tratar esse objeto. O problema ocorrerá se você não o conferir, mas prosseguir tentando chamar outra função no objeto `None`, conforme mostra o código a seguir:

```
print(bs.nonExistentTag.someTag)
```

Uma exceção será devolvida:

```
AttributeError: 'NoneType' object has no attribute 'someTag'
```

Então, como é possível se proteger nessas duas situações? O modo mais fácil é verificar as duas situações explicitamente:

```
try:
    badContent = bs.nonExistingTag.anotherTag
except AttributeError as e:
    print('Tag was not found')
else:
    if badContent == None:
        print ('Tag was not found')
    else:
        print(badContent)
```

Essa verificação e o tratamento de cada erro parecem muito trabalhosos à primeira vista, mas é fácil fazer uma pequena reorganização nesse código, deixando-o menos difícil de ser escrito (e, acima de tudo, muito menos difícil de ler). O código a seguir, por exemplo, é o nosso mesmo scraper escrito de modo um pouco diferente:

```
from urllib.request import urlopen
from urllib.error import HTTPError
from bs4 import BeautifulSoup

def getTitle(url):
    try:
        html = urlopen(url)
    except HTTPError as e:
        return None
    try:
        bs = BeautifulSoup(html.read(), 'html.parser')
        title = bs.body.h1
    except AttributeError as e:
        return None
    return title

title = getTitle('http://www.pythonscraping.com/pages/page1.html')
if title == None:
    print('Title could not be found')
else:
    print(title)
```

Nesse exemplo, criamos uma função `getTitle` que devolve o título da página, ou um objeto `None` caso tenha havido algum problema para obtê-lo. Em `getTitle`, verificamos se houve um `HTTPError`, como no exemplo anterior, e encapsulamos duas das linhas do `BeautifulSoup` em uma instrução `try`. Um `AttributeError` pode ser lançado por qualquer uma dessas linhas (se o

servidor não existir, `html` seria um objeto `None` e `html.read()` lançaria um `AttributeError`). Na verdade, você poderia incluir quantas linhas quiser em uma instrução `try`, ou chamar outra função totalmente diferente, que poderia lançar um `AttributeError` em qualquer ponto.

Ao escrever scrapers, é importante pensar no padrão geral de seu código a fim de lidar com as exceções e, ao mesmo tempo, deixá-lo legível. É provável que você queira também fazer uma intensa reutilização de código. Ter funções genéricas como `getSiteHTML` e `getTitle` (completas, com todo o tratamento para exceções) facilita fazer uma coleta de dados da web de forma rápida – e confiável.

---

<sup>1</sup> N.T.: Tradução livre, com base no original em inglês.

<sup>2</sup> N.T.: *Beautiful Soup* pode ser traduzido como “Bela Sopa”.

# Parsing de HTML avançado

Quando perguntaram a Michelangelo como ele havia conseguido esculpir uma obra de arte tão majestosa quanto o seu *Davi*, dizem que ele teria dado a seguinte resposta famosa: “É fácil. Basta remover as partes da pedra que não se pareçam com Davi”.

Embora o web scraping em nada se pareça com escultura em mármore na maioria dos outros aspectos, podemos assumir uma postura semelhante quando se trata de extrair as informações desejadas de páginas web complicadas. Várias técnicas podem ser usadas para remover o conteúdo que não se pareça com o que estamos procurando, até chegarmos nas informações desejadas. Neste capítulo, veremos como fazer parse de páginas HTML complicadas a fim de extrair somente as informações desejadas.

## Nem sempre um martelo é necessário

Se estivermos diante de um emaranhado de tags, pode ser tentador usar de imediato instruções com várias linhas para extrair as informações desejadas. No entanto, lembre-se de que dispor das técnicas usadas nesta seção sem planejamento pode resultar em um código difícil de depurar, frágil – ou ambos. Antes de começar, vamos analisar algumas formas de evitar totalmente a necessidade de fazer um parsing avançado de HTML!

Suponha que você esteja visando a certo conteúdo. Talvez seja um nome, uma estatística ou um bloco de texto. A informação pode estar escondida a 20 tags de profundidade em um HTML confuso, sem tags ou atributos HTML convenientes à vista. Suponha que você decida deixar totalmente de lado a cautela e escreva um código como a linha a seguir, na tentativa de extrair os dados:

```
bs.find_all('table')[4].find_all('tr')[2].find('td').find_all('div')[1].find('a')
```

Esse código não parece muito bom. Além do aspecto estético da linha, até mesmo a menor das modificações feita por um administrador no site

poderá causar falhas em seu web scraper. O que aconteceria se o desenvolvedor do site decidisse acrescentar outra tabela ou outra coluna de dados? E se o desenvolvedor adicionasse outro componente (com algumas tags `div`) no início da página? A linha anterior é frágil e depende de a estrutura do site jamais mudar.

Então quais são as suas opções?

- Procure um link para “imprimir a página” ou, quem sabe, uma versão móvel do site com um HTML mais bem formatado (outras informações sobre como se apresentar como um dispositivo móvel – e receber versões móveis dos sites – no *Capítulo 14*).
- Procure as informações ocultas em um arquivo JavaScript. Lembre-se de que talvez seja necessário analisar os arquivos JavaScript importados para fazer isso. Por exemplo, uma vez, coletei endereços de ruas (junto com as latitudes e as longitudes) de um site em um array muito bem formatado analisando o JavaScript em busca do Google Map incluído, o qual exibiu um marcador para cada endereço.
- É mais comum para títulos de página, mas a informação pode estar disponível no próprio URL da página.
- Se, por algum motivo, a informação procurada for exclusiva do site em questão, você estará sem sorte. Se não for, procure pensar em outras fontes a partir das quais seria possível obter essas informações. Há outro site com os mesmos dados? Esse site exibe dados coletados ou consolidados a partir de outro site?

Em especial, se estiver diante de dados escondidos ou mal formatados, é importante não começar simplesmente a escrever um código que faça você cavar um buraco do qual talvez não saia. Respire fundo e pense nas alternativas.

Se tiver certeza de que não há alternativas, no resto do capítulo, descreveremos soluções padrões e criativas para selecionar tags de acordo com suas posições, o contexto, os atributos e o conteúdo. As técnicas apresentadas, quando usadas corretamente, serão muito convenientes para escrever web crawlers mais estáveis e confiáveis.

## **Outras utilidades do BeautifulSoup**

No *Capítulo 1*, vimos rapidamente como instalar e executar o

BeautifulSoup, assim como selecionar objetos, um de cada vez. Nesta seção, discutiremos como procurar tags por atributos, trabalhar com listas de tags e navegar por árvores de parse.

Praticamente todo site que você encontrar conterá folhas de estilo. Embora você ache que uma camada de estilização em sites, projetada especificamente para ser interpretada por navegadores e por seres humanos seja inconveniente, o surgimento do CSS foi extremamente vantajoso para os web scrapers. O CSS se baseia na diferenciação de elementos HTML – que, de outro modo, poderiam ter exatamente a mesma marcação – para estilizá-los de modo distinto. Algumas tags podem ter o seguinte aspecto:

```
<span class="green"></span>
```

Outras podem ser assim:

```
<span class="red"></span>
```

Os web scrapers podem separar facilmente essas tags de acordo com a classe; por exemplo, o BeautifulSoup pode ser usado para obter todos os textos em vermelho (red), mas não os textos em verde (green). Como o CSS depende desses atributos de identificação para estilizar adequadamente os sites, é quase certo que esses atributos de classe e ID sejam abundantes na maioria dos sites modernos.

Vamos criar um web scraper de exemplo que colete dados da página em <http://www.pythonscraping.com/pages/warandpeace.html>.

Nessa página, as linhas com frases ditas pelas personagens da história estão em vermelho, enquanto os nomes das personagens estão em verde. Podemos ver as tags `span`, que fazem referência às classes CSS apropriadas, na seguinte amostra do código-fonte da página:

```
<span class="red">Heavens! what a virulent attack!</span> replied  
<span class="green">the prince</span>, not in the least disconcerted  
by this reception.
```

Considerando a página inteira, podemos criar um objeto BeautifulSoup com ela usando um programa semelhante àquele utilizado no *Capítulo 1*:

```
from urllib.request import urlopen  
from bs4 import BeautifulSoup  
  
html = urlopen('http://www.pythonscraping.com/pages/page1.html')  
bs = BeautifulSoup(html.read(), 'html.parser')
```

Com esse objeto BeautifulSoup, a função `find_all` pode ser usada para extrair uma lista Python com nomes próprios, encontrados ao selecionar somente o texto entre as tags `<span class="green"></span>` (`find_all` é uma função

extremamente flexível que usaremos bastante mais adiante neste livro):

```
nameList = bs.findAll('span', {'class':'green'})
for name in nameList:
    print(name.get_text())
```

Quando executado, esse código deve listar todos os nomes próprios do texto, na ordem em que aparecem em *War and Peace* (Guerra e Paz). Então, o que está acontecendo nesse caso? Havíamos chamado `bs.tagName` antes para obter a primeira ocorrência dessa tag na página. Agora chamamos `bs.find_all(tagName, tagAttributes)` para obter uma lista de todas as tags da página, em vez de obter somente a primeira.

Depois de obter uma lista de nomes, o programa itera por todos os nomes da lista e exibe `name.get_text()` para separar o conteúdo das tags.

Quando usar `get_text()` e quando preservar as tags

`.get_text()` remove todas as tags do documento com o qual você está trabalhando e devolve uma string Unicode contendo somente o texto. Por exemplo, se você estiver trabalhando com um bloco de texto grande, contendo muitos hiperlinks, parágrafos e outras tags, tudo isso será removido, e restará um bloco de texto sem tags.

Lembre-se de que é muito mais fácil encontrar o que você procura em um objeto BeautifulSoup do que em um bloco de texto. Chamar `.get_text()` deve ser sempre a sua última tarefa, imediatamente antes de exibir, armazenar ou manipular os dados finais. Em geral, você deve se esforçar ao máximo para tentar preservar a estrutura de tags de um documento.

## find() e find\_all() com o BeautifulSoup

`find()` e `find_all()` são as duas funções do BeautifulSoup que provavelmente serão mais usadas. Com elas, é possível filtrar facilmente as páginas HTML e encontrar as listas de tags desejadas – ou uma só tag – de acordo com seus vários atributos.

As duas funções são extremamente parecidas, como mostram suas definições na documentação do BeautifulSoup:

```
find_all(tag, attributes, recursive, text, limit, keywords)
find(tag, attributes, recursive, text, keywords)
```

É bem provável que, em 95% do tempo, somente os dois primeiros argumentos serão necessários: `tag` e `attributes`. Não obstante, vamos analisar todos os argumentos com mais detalhes.

Já vimos antes o argumento `tag`; podemos passar o nome de uma tag como string ou até mesmo uma lista Python de nomes de tags definidos como strings. Por exemplo, o código a seguir devolve uma lista com todas as tags de cabeçalho em um documento:<sup>1</sup>

```
.find_all(['h1', 'h2', 'h3', 'h4', 'h5', 'h6'])
```

O argumento `attributes` aceita um dicionário Python de atributos e faz a correspondência com tags que contenham qualquer um desses atributos. Por exemplo, a função a seguir devolve as tags `span` do documento HTML, *tanto* para `green` (verde) *quanto* para `red` (vermelho):

```
.find_all('span', {'class':{'green', 'red'}})
```

O argumento `recursive` é um booleano. Qual é o nível de aprofundamento que você gostaria de alcançar no documento? Se `recursive` for `True`, a função `find_all` analisará os filhos, e os filhos dos filhos, em busca de tags que correspondam aos parâmetros. Se for `False`, somente as tags de nível mais alto no documento serão verificadas. Por padrão, `find_all` funciona de modo recursivo (`recursive` é definido com `True`); em geral, é uma boa ideia deixá-lo assim, a menos que você saiba exatamente o que deve ser feito e o desempenho seja um problema.

O argumento `text` é peculiar por fazer a correspondência com base no conteúdo textual das tags, em vez de usar suas propriedades. Por exemplo, se quiser encontrar o número de vezes que “the prince” está cercado por tags na página de exemplo, a função `.find_all()` poderia ser substituída no exemplo anterior pelas linhas a seguir:

```
nameList = bs.find_all(text='the prince')
print(len(nameList))
```

O resultado desse código é 7.

O argumento `limit` obviamente é usado somente no método `find_all`; `find` é equivalente à chamada de `find_all` com um limite igual a 1. Esse parâmetro pode ser definido se você estiver interessado apenas em obter os primeiros *x* itens da página. Saiba, porém, que serão obtidos os primeiros itens da página na ordem em que ocorrerem, e não necessariamente os primeiros que você quer.

O argumento `keyword` permite selecionar tags que contenham um atributo ou um conjunto de atributos específicos. Por exemplo:

```
title = bs.find_all(id='title', class_='text')
```

Esse comando devolve a primeira tag com a palavra “text” no atributo `class_` e “title” no atributo `id`. Observe que, por convenção, cada valor de um `id` deve ser usado somente uma vez na página. Assim, na prática, uma linha como essa talvez não seja particularmente útil, e deveria ser equivalente a:

```
title = bs.find(id='title')
```

## Argumentos nomeados e “class”

O argumento `keyword` pode ser útil em algumas situações. Entretanto, ele é tecnicamente redundante como um recurso do BeautifulSoup. Tenha em mente que tudo que puder ser feito com `keyword` também poderá ser feito com técnicas discutidas mais adiante neste capítulo (consulte as seções “*Expressões regulares*” e “*Expressões lambda*”).

Por exemplo, as duas linhas a seguir são idênticas:

```
bs.find_all(id='text')
bs.find_all('', {'id':'text'})
```

Além disso, você poderá ocasionalmente deparar com problemas ao usar `keyword`, sobretudo se estiver procurando elementos de acordo com seu atributo `class`, pois `class` é uma palavra reservada protegida em Python, ou seja, ela não pode ser usada como uma variável ou como nome de argumento (não há relação com o argumento `keyword` de `BeautifulSoup.find_all()`,

discutido antes) <sup>2</sup>. Por exemplo, se tentarmos fazer a chamada a seguir, veremos um erro de sintaxe em consequência do uso não padrão de `class`:

```
bs.find_all(class='green')
```

Em vez disso, podemos usar a solução, de certa forma deselegante, do BeautifulSoup, que envolve o acréscimo de um underscore:

```
bs.find_all(class_='green')
```

Como alternativa, `class` pode ser usada entre aspas:

```
bs.find_all('', {'class':'green'})
```

A essa altura, você poderia estar se perguntando o seguinte: “Espere um momento, eu já não sei como obter uma tag com uma lista de atributos passando-os para a função em uma lista de dicionário?”.

Lembre-se de que passar uma lista de tags para `.find_all()` por meio da lista de atributos atua como um filtro “ou” (seleciona uma lista com todas as tags que tenham `tag1`, `tag2` ou `tag3...`). Se tivermos uma lista de tags longa, poderemos acabar com muita informação indesejada. O argumento `keyword` permite acrescentar um filtro “e”.

## Outros objetos do BeautifulSoup

Até agora no livro, vimos dois tipos de objetos na biblioteca BeautifulSoup:

### *Objetos BeautifulSoup*

Instâncias vistas nos exemplos anteriores de código como a variável `bs`.

### *Objetos Tag*

Obtidos na forma de listas – ou individualmente – por meio da chamada a `find` e `find_all` em um objeto BeautifulSoup, ou descendo níveis, assim:

```
bs.div.h1
```

No entanto, há outros dois objetos na biblioteca que, apesar de serem usados com menos frequência, são importantes:

### Objetos `NavigableString`

Usados para representar texto em tags, em vez das próprias tags (algumas funções atuam em `NavigableStrings` e o geram, em vez de usar objetos tag).

### Objeto `Comment`

Usado para encontrar comentários HTML em tags de comentário, `<!--como este-->`.

Esses quatro objetos são os únicos que você verá na biblioteca BeautifulSoup (atualmente, quando este livro foi escrito).

## Navegando em árvores

A função `find_all` é responsável por encontrar tags com base em seus nomes e atributos. Como seria se tivéssemos que achar uma tag com base em sua localização em um documento? É nesse cenário que uma navegação em árvore se torna conveniente. No *Capítulo 1*, vimos como navegar em uma árvore do BeautifulSoup em uma única direção:

```
bs.tag.subTag.outraSubTag
```

Vamos agora ver como navegar para cima, para os lados e na diagonal em árvores HTML. Usaremos nosso site de compras extremamente questionável em <http://www.pythonscraping.com/pages/page3.html> como uma página de exemplo para coleta de dados, conforme vemos na Figura 2.1.

## **Totally Normal Gifts**

Here is a collection of totally normal, totally reasonable gifts that your friends are sure to love! Our collection is hand-curated

We haven't figured out how to make online shopping carts yet, but you can send us a check to:

123 Main St.

Abuja, Nigeria

We will then send your totally amazing gift, pronto! Please include an extra \$5.00 for gift wrapping.

Item Title	Description	Cost	Image
Vegetable Basket	This vegetable basket is the perfect gift for your health conscious (or overweight) friends! <b><i>Now with super-colorful bell peppers!</i></b>	\$15.00	
Russian Nesting Dolls	Hand-painted by trained monkeys, these exquisite dolls are priceless! And by "priceless," we mean "extremely expensive"! <b><i>8 entire dolls per set! Octuple the presents!</i></b>	\$10,000.52	
Fish Painting	If something seems fishy about this painting, it's because it's a fish! <b><i>Also hand-painted by trained monkeys!</i></b>	\$10,005.00	
Dead Parrot	This is an ex-parrot! <b><i>Or maybe he's only resting?</i></b>	\$0.50	

Figura 2.1 – Imagem da tela em <http://www.pythonscraping.com/pages/page3.html>.

O HTML dessa página, mapeado na forma de árvore (com algumas tags omitidas para sermos mais sucintos), tem o seguinte aspecto:

- HTML
  - body
    - div.wrapper
      - h1
      - div.content
        - table#giftList
          - tr
            - th
            - th
            - th
            - th
          - tr.gift#gift1

- td
- td
- span.excitingNote
- td
- td
- img
- ... outras linhas da tabela...
- div.footer

Essa mesma estrutura HTML será usada como exemplo nas próximas seções.

### Lidando com filhos e outros descendentes

Em ciência da computação e em alguns ramos da matemática, muitas vezes ouvimos falar de ações horríveis cometidas com os filhos: movê-los, armazená-los, removê-los e até mesmo matá-los. Felizmente, esta seção tem como foco apenas a sua seleção!

Na biblioteca BeautifulSoup, assim como em muitas outras bibliotecas, há uma distinção entre *filhos* e *descendentes*: de modo muito parecido com uma árvore genealógica humana, os filhos estão sempre exatamente uma tag abaixo de um pai, enquanto os descendentes podem estar em qualquer nível da árvore abaixo de um pai. Por exemplo, as tags `tr` são filhas da tag `table`, enquanto `tr`, `th`, `td`, `img` e `span` são todas descendentes da tag `table` (pelo menos em nossa página de exemplo). Todos os filhos são descendentes, mas nem todos os descendentes são filhos.

Em geral, as funções do BeautifulSoup sempre lidam com os descendentes da tag selecionada no momento. Por exemplo, `bs.body.h1` seleciona a primeira tag `h1` que é descendente da tag `body`. As tags localizadas fora do corpo não serão encontradas.

De modo semelhante, `bs.div.find_all('img')` encontrará a primeira tag `div` no documento, e então obterá uma lista de todas as tags `img` que são descendentes dessa tag `div`.

Se quiser encontrar somente os descendentes que sejam filhos, a tag `.children` pode ser usada:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
```

```
bs = BeautifulSoup(html, 'html.parser')

for child in bs.find('table',{'id':'giftList'}).children:
    print(child)
```

Esse código exibe a lista das linhas de produto da tabela `giftList`, incluindo a linha inicial com os rótulos das colunas. Se fôssemos escrevê-lo com a função `descendants()` no lugar de `children()`, aproximadamente duas dúzias de tags seriam encontradas na tabela e seriam exibidas, incluindo as tags `img`, `span` e as tags `td` individuais. Sem dúvida, é importante fazer uma diferenciação entre filhos e descendentes!

### Lidando com irmãos

Com a função `next_siblings()` do BeautifulSoup, é trivial coletar dados de tabelas, particularmente daquelas com linhas de título.

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')

for sibling in bs.find('table', {'id':'giftList'}).tr.next_siblings:
    print(sibling)
```

Esse código tem como saída todas as linhas de produtos da tabela, exceto a primeira linha com os títulos. Por que a linha com títulos é ignorada? Objetos não podem ser irmãos deles mesmos. Sempre que os irmãos de um objeto forem obtidos, o próprio objeto não estará incluído na lista. Como implica o nome da função, ela chama somente os *próximos* (next) irmãos. Se uma linha no meio da lista, por exemplo, fosse selecionada, e `next_siblings` fosse chamada, somente os irmãos subsequentes seriam devolvidos. Assim, ao selecionar a linha de títulos e chamar `next_siblings`, podemos selecionar todas as linhas da tabela, sem selecionar a própria linha de títulos.

### Deixe as seleções mais específicas

○ código anterior funcionará igualmente de forma apropriada se você selecionar `bs.table.tr` ou até mesmo somente `bs.tr` para selecionar a primeira linha da tabela. Entretanto, no código, eu me dei ao trabalho de escrever tudo em um formato mais longo:

```
bs.find('table',{'id':'giftList'}).tr
```

Mesmo que pareça haver somente uma tabela (ou outra tag desejada) na página, é fácil cometer erros. Além do mais, os layouts das páginas mudam o tempo todo. O que poderia ter sido a primeira tag da página um dia pode vir a ser a segunda ou a terceira tag desse tipo encontrada na página. Para deixar seus scrapers mais robustos, é melhor ser o mais específico possível nas seleções de tags. Tire proveito dos atributos de tags se estiverem disponíveis.

Como complemento para `next_siblings`, a função `previous_siblings` muitas vezes pode ser útil se houver uma tag facilmente selecionável no final de uma lista de tags irmãs que você queira obter.

Além disso, é claro, temos as funções `next_sibling` e `previous_sibling`, que têm quase a mesma função de `next_siblings` e `previous_siblings`, mas devolvem uma única tag, em vez de devolver uma lista delas.

## Lidando com pais

Ao coletar dados de páginas, é provável que você perceba que precisará encontrar pais de tags com menos frequência do que terá de encontrar seus filhos ou seus irmãos. Em geral, ao analisar páginas HTML com o objetivo de rastreá-las, começamos observando as tags de nível mais alto e, em seguida, descobrimos como descer para outros níveis até alcançar a porção exata de dados que queremos. Ocasionalmente, porém, poderemos nos ver em situações peculiares que exigirão as funções do BeautifulSoup para encontrar os pais: `.parent` e `.parents`. Por exemplo:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')
print(bs.find('img',
              {'src': '../img/gifts/img1.jpg'})
      .parent.previous_sibling.get_text())
```

Esse código exibirá o preço do objeto representado pela imagem no endereço `../img/gifts/img1.jpg` (nesse caso, o preço é de 15 dólares).

Como isso funciona? O diagrama a seguir representa a estrutura de árvore da parte da página HTML com a qual estamos trabalhando, com os passos numerados:

- `<tr>`
  - `td`
  - `td`
  - `td` ③
    - `"$15.00"` ④
  - `td` ②

– `` ❶

- ❶ A tag de imagem com `src="../../img/gifts/img1.jpg"` é inicialmente selecionada.
- ❷ Selecionamos o pai dessa tag (nesse caso, é a tag `td`).
- ❸ Selecionamos a `previous_sibling` da tag `td` (nesse caso, é a tag `td` que contém o valor do produto em dólar).
- ❹ Selecionamos o texto nessa tag: “\$15.00”.

## Expressões regulares

Como diz a velha piada em ciência da computação: “Suponha que você tenha um problema e decida resolvê-lo usando expressões regulares. Bem, agora você tem dois problemas”.

Infelizmente, as expressões regulares (em geral, abreviadas como *regex*) muitas vezes são ensinadas com tabelas grandes de símbolos aleatórios, encadeados de modo a parecer um conjunto sem sentido. Isso tende a afastar as pessoas; mais tarde, elas saem para o mercado de trabalho e escrevem funções complicadas e desnecessárias para pesquisar e filtrar, quando apenas precisavam de uma expressão regular com uma só linha!

Felizmente para você, não é tão difícil começar a trabalhar com expressões regulares, e é fácil aprender a usá-las analisando e fazendo experimentos com alguns exemplos simples.

As *expressões regulares* recebem esse nome porque são usadas para identificar strings regulares; elas podem seguramente afirmar o seguinte: “Sim, esta string que você me deu segue as regras e será devolvida” ou “Esta string não segue as regras e será descartada”. Esse recurso pode ser excepcionalmente conveniente para analisar documentos grandes em busca de strings que sejam números de telefone ou endereços de email, de forma rápida.

Observe que usei a expressão *strings regulares*. O que é uma string regular? É qualquer string que seja gerada por uma série de regras lineares<sup>3</sup>, como estas:

1. Escreva a letra *a* no mínimo uma vez.
2. Concatene aí a letra *b* exatamente cinco vezes.
3. Concatene aí a letra *c* qualquer número par de vezes.
4. Escreva a letra *d* ou a letra *e* no final.

Strings que seguem essas regras são *aaaabbbbbcccd*, *aabbbbbcce* e assim por diante (há um número infinito de variações).

As expressões regulares são simplesmente uma forma concisa de expressar esses conjuntos de regras. Por exemplo, eis a expressão regular para a série de passos que acabamos de descrever:

`aa*bbbb(cc)*(d|e)`

Essa string pode parecer um pouco assustadora à primeira vista, mas se tornará mais clara se for separada em seus componentes:

`aa*`

A letra *a* é escrita, seguida de *a\** (leia como *a* *asterisco*), que significa “qualquer quantidade de *as*, incluindo nenhum”. Dessa forma, podemos garantir que a letra *a* seja escrita pelo menos uma vez.

`bbbb`

Não há efeitos especiais nesse caso – somente cinco *bs* em sequência.

`(cc)*`

Qualquer número par de itens pode ser agrupado em pares, portanto, para impor essa regra sobre itens pares, podemos escrever dois *cs*, colocá-los entre parênteses e escrever um asterisco depois, o que significa que podemos ter qualquer número de *pares* de *cs* (observe que isso pode significar nenhum par também).

`(d|e)`

Acrescentar uma barra no meio de duas expressões significa que ela pode ser “isto *ou* aquilo”. Nesse caso, estamos dizendo para “adicionar um *d* ou um *e*”. Assim, garantimos que haja exatamente um desses dois caracteres.

### Fazendo experimentos com RegEx

Quando aprendemos a escrever expressões regulares, é essencial fazer experimentos para ter uma noção de como elas funcionam. Se você não estiver disposto a abrir um editor de código, escrever algumas linhas e executar seu programa para ver se uma expressão regular funciona conforme esperado, é possível acessar um site como o *Regex Pal* (<http://regexpal.com/>) e testar suas expressões regulares enquanto são criadas.

A Tabela 2.1 lista os símbolos comuns usados em expressões regulares,

com explicações breves e exemplos. Essa lista não é, de forma alguma, completa; conforme mencionamos antes, pequenas variações podem ser encontradas de linguagem para linguagem. No entanto, esses 12 símbolos são as expressões regulares mais comuns usadas em Python e podem ser utilizadas para encontrar e coletar praticamente qualquer tipo de string.

*Tabela 2.2 – Símbolos comuns usados em expressões regulares*

Símbolo(s)	Significado	Exemplo	Exemplo de correspondência
*	Faz a correspondência com o caractere, a subexpressão ou o caractere entre colchetes anterior, 0 ou mais vezes.	a*b*	aaaaaaaa, aaabbbbb, bbbbbb
+	Faz a correspondência com o caractere, a subexpressão ou o caractere entre colchetes anterior, 1 ou mais vezes.	a+b+	aaaaaaaaab, aaabbbbb, abbbbb
[ ]	Faz a correspondência com qualquer caractere entre os colchetes (isto é, “Escolha qualquer um destes itens”).	[A-Z]*	APPLE, CAPITALS, QWERTY
( )	Uma subexpressão agrupada (são avaliadas antes, na “ordem de operações” das expressões regulares).	(a*b)*	aaabaab, abaaab, ababaaaaab
{m, n}	Faz a correspondência com o caractere, a subexpressão ou o caractere entre colchetes anterior, entre <i>m</i> e <i>n</i> vezes (inclusive).	a{2,3}b{2,3}	aabbb, aaabbb, aabb
[^ ]	Faz a correspondência com qualquer caractere único que <i>não</i> esteja entre colchetes.	[^A-Z]*	apple, lowercase, qwerty
	Faz a correspondência com qualquer caractere, string de caracteres ou subexpressão, separado pela   (observe que essa é uma barra vertical, ou <i>pipe</i> , e não a letra i maiúscula).	b(a i e)d	bad, bid, bed
.	Faz a correspondência com qualquer caractere único (incluindo símbolos, números, um espaço etc.).	b.d	bad, bzd, b\$d, b d
^	Indica que um caractere ou subexpressão ocorre no início de uma string.	^a	apple, asdf, a
\	Um caractere de escape (permite usar caracteres especiais com seus significados literais).	\. \  \\	.   \
\$	Usado com frequência no final de uma expressão regular, significa “faz a correspondência disto até o final da string”. Sem ele, toda expressão regular tem um “.” de fato no final dela, aceitando strings nas quais somente a primeira parte	[A-Z]*[a-z]*\$	ABCabc, zzzyx, Bob

	dela seja correspondente. Podemos pensar nisso como análogo ao símbolo ^.		
?!	“Não contém.” Esse par peculiar de símbolos, imediatamente antes de um caractere (ou de uma expressão regular), indica que esse caractere não deve ser encontrado nesse lugar específico na string maior. Isso pode ser complicado de usar; afinal de contas, o caractere pode ser encontrado em uma parte diferente da string. Se estiver tentando eliminar totalmente um caractere, use-o em conjunto com um ^ e um \$ nas extremidades.	^((?![A-Z]).)*\$	no-caps- here, \$ymb0ls a4e f!ne

Um exemplo clássico de expressões regulares pode ser visto na prática de identificação de endereços de email. Embora as regras exatas que determinam os endereços de email variem um pouco de servidor de emails para servidor de emails, é possível definir algumas regras gerais. A segunda coluna mostra a expressão regular correspondente para cada uma dessas regras:

<b>Regra 1</b> A primeira parte de um endereço de email contém no mínimo um dos seguintes dados: letras maiúsculas, letras minúsculas, os números de 0–9, pontos (.), sinais de adição (+) ou underscores ( _ ).	[A-Za-z0-9\._+]+ A expressão regular concisa é bem inteligente. Por exemplo, ela sabe que “A-Z” significa “qualquer letra maiúscula, de A a Z”. Ao colocar todas essas sequências e símbolos possíveis entre colchetes (em oposição a parênteses), estamos dizendo que “este símbolo pode ser qualquer um destes dados que listamos entre os colchetes”. Observe também que o sinal de + significa que “estes caracteres podem ocorrer quantas vezes quiserem, mas devem ocorrer pelo menos uma vez”.
<b>Regra 2</b> Depois disso, o endereço de email contém o símbolo @.	@ Isto é bem simples: o símbolo @ deve ocorrer no meio, e deve aparecer exatamente uma vez.
<b>Regra 3</b> O endereço de email então deve conter no mínimo uma letra maiúscula ou minúscula.	[A-Za-z]+ Você pode usar somente letras na primeira parte do nome do domínio, depois do símbolo @. Além disso, deve haver no mínimo um caractere.
<b>Regra 4</b> Um ponto deve estar presente a seguir ( . ).	\. Um ponto ( . ) deve estar incluído antes do nome do domínio. A barra invertida é usada nesse caso como um caractere de escape.
<b>Regra 5</b> Por fim, o endereço de email termina com <i>com</i> , <i>org</i> , <i>edu</i> ou <i>net</i> (na verdade, há muitos domínios de nível mais alto	(com org edu net) Lista as possíveis sequências de letras que podem ocorrer depois do ponto na segunda parte de um endereço de email.

possíveis, mas esses quatro devem ser suficientes para o exemplo).

Ao concatenar todas as regras, temos a seguinte expressão regular:

```
[A-Za-z0-9\._+]+@[A-Za-z]+\.(com|org|edu|net)
```

Quando tentar escrever qualquer expressão regular a partir do zero, é melhor criar antes uma lista de passos que descreva concretamente a aparência da string desejada. Preste atenção nos casos extremos. Por exemplo, se estiver identificando números de telefone, você está considerando códigos de país e extensões?

Expressões regulares: nem sempre são regulares!

A versão padrão das expressões regulares (a versão incluída neste livro e usada por Python e pelo BeautifulSoup) é baseada na sintaxe usada em Perl. A maioria das linguagens de programação modernas usa essa versão ou uma versão similar. Contudo, saiba que, se você estiver usando expressões regulares em outra linguagem, é possível que haja problemas. Mesmo algumas linguagens modernas, por exemplo, Java, apresentam pequenas diferenças no modo como as expressões regulares são tratadas. Na dúvida, leia a documentação!

## Expressões regulares e o BeautifulSoup

Se a seção anterior sobre expressões regulares pareceu um pouco desvinculada do objetivo deste livro, eis o que amarrará tudo. O BeautifulSoup e as expressões regulares andam de mãos dadas quando se trata de coletar dados na web. Com efeito, a maioria das funções que aceita um argumento do tipo string (por exemplo, `find(id="umaTagIdAqui")`) também aceitará uma expressão regular.

Vamos observar alguns exemplos, coletando dados da página que se encontra em <http://www.pythonscraping.com/pages/page3.html>.

Observe que o site tem várias imagens de produtos no seguinte formato:

```

```

Se quiséssemos obter os URLs de todas as imagens dos produtos, à primeira vista pareceria muito simples: bastaria obter todas as tags de imagens usando `.find_all("img")`, certo? Contudo, há um problema. Além das evidentes imagens “extras” (por exemplo, logos), os sites modernos muitas vezes têm imagens ocultas, imagens em branco usadas para espaçamento e alinhamento de elementos, além de outras tags de imagens aleatórias que talvez não sejam de seu conhecimento. Certamente não é possível contar com o fato de que as únicas imagens da página sejam de produtos.

Vamos supor também que o layout da página possa mudar, ou que, por qualquer outro motivo, não queremos depender da *posição* da imagem na página para encontrar a tag correta. Isso pode acontecer se estivermos tentando obter elementos ou dados específicos espalhados aleatoriamente em um site. Por exemplo, a imagem de um produto em destaque pode aparecer em um layout especial no início de algumas páginas, mas não em outras.

A solução é procurar algo que sirva como identificador da própria tag. Nesse caso, podemos analisar o path de arquivo das imagens dos produtos:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')
images = bs.find_all('img',
    {'src':re.compile('\.\./img/gifts/img.*\.jpg')})
for image in images:
    print(image['src'])
```

Esse código exibe somente os paths relativos das imagens que comecem com `../img/gifts/img` e terminem com `.jpg`, e o resultado é este:

```
../img/gifts/img1.jpg
../img/gifts/img2.jpg
../img/gifts/img3.jpg
../img/gifts/img4.jpg
../img/gifts/img6.jpg
```

Uma expressão regular pode ser inserida como qualquer argumento em uma expressão do BeautifulSoup, o que permite ter bastante flexibilidade para encontrar os elementos desejados.

## Acessando atributos

Até agora, vimos como acessar e filtrar tags e acessar conteúdos dentro delas. No entanto, com frequência no web scraping, não estaremos procurando o conteúdo de uma tag, mas os seus atributos. Isso será particularmente útil com tags como `a`, em que o URL para o qual elas apontam está contido no atributo `href`, ou para a tag `img`, em que a imagem desejada está no atributo `src`.

Com objetos de tag, uma lista Python com os atributos pode ser automaticamente acessada por meio de uma chamada como esta:

```
myTag.attrs
```

Tenha em mente que esse código devolve literalmente um objeto de dicionário Python, fazendo com que seja trivial acessar e manipular esses atributos. O local em que está a fonte de uma imagem, por exemplo, pode ser encontrado com a linha a seguir:

```
myImgTag.attrs['src']
```

## Expressões lambda

Se você teve uma educação formal em ciência da computação, provavelmente deve ter conhecido as expressões lambda enquanto estudava e jamais voltou a usá-las. Se não teve, talvez não as conheça (ou conheça somente como “aquele assunto que pretendia estudar algum dia”). Esta seção não entra em detalhes sobre esses tipos de funções, mas mostra como elas podem ser úteis no web scraping.

Essencialmente, uma *expressão lambda* é uma função que é passada para outra função como uma variável; em vez de definir uma função como  $f(x, y)$ , é possível defini-la como  $f(g(x), y)$  ou até mesmo como  $f(g(x), h(x))$ .

O BeautifulSoup permite passar determinados tipos de funções como parâmetros da função `find_all`.

A única restrição é que essas funções devem aceitar um objeto tag como argumento e devolver um booleano. Todo objeto tag encontrado pelo BeautifulSoup é avaliado por essa função, e as tags avaliadas como `True` são devolvidas, enquanto as demais são descartadas.

Por exemplo, o código a seguir obtém todas as tags que tenham exatamente dois atributos:

```
bs.find_all(lambda tag: len(tag.attrs) == 2)
```

Nesse caso, a função passada como argumento é `len(tag.attrs) == 2`. Quando ela for `True`, a função `find_all` devolverá a tag. Isso significa que ela encontrará as tags com dois atributos, por exemplo:

```
<div class="body" id="content"></div>
<span style="color:red" class="title"></span>
```

As funções lambda são tão úteis que você pode usá-las até mesmo para substituir funções do BeautifulSoup:

```
bs.find_all(lambda tag: tag.get_text() ==
             'Or maybe he\'s only resting?')
```

Isso pode ser feito sem uma função lambda:

```
bs.find_all('', text='Or maybe he\'s only resting?')
```

Todavia, se você se lembrar da sintaxe da função lambda e de como acessar propriedades de tags, talvez não seja necessário se lembrar de mais nenhuma outra sintaxe do BeautifulSoup novamente!

Como a função lambda fornecida pode ser qualquer função que devolva um valor `True` ou `False`, podemos até mesmo combiná-la com expressões regulares para encontrar tags com um atributo que corresponda a determinado padrão de string.

- 
- [1](#) Se você estiver tentando obter uma lista com todas as tags `h<algum_nível>` do documento, há maneiras mais sucintas de escrever esse código e fazer o mesmo. Veremos outras formas de abordar esses tipos de problemas na seção “*Expressões regulares e o BeautifulSoup*”.
  - [2](#) O Guia de Referência da Linguagem Python (Python Language Reference) contém uma lista completa das palavras reservadas protegidas ([https://docs.python.org/3/reference/lexical\\_analysis.html#keywords](https://docs.python.org/3/reference/lexical_analysis.html#keywords)).
  - [3](#) Você pode estar se perguntando: “Há expressões ‘irregulares’?”. Expressões não regulares estão além do escopo deste livro, mas incluem strings como “escreva um número primo de *as*, seguido exatamente do dobro desse número de *bs*” ou “escreva um palíndromo”. É impossível identificar strings desse tipo com uma expressão regular. Felizmente, jamais me vi em uma situação em que meu web scraper tivesse de identificar esses tipos de strings.

## Escrevendo web crawlers

Até agora, vimos páginas estáticas únicas, com exemplos, de certo modo, artificiais. Neste capítulo, começaremos a analisar problemas do mundo real, com scrapers percorrendo várias páginas – e até mesmo vários sites.

Os *web crawlers* (rastreadores web) recebem esse nome porque rastreiam (crawl) a web. Em seu núcleo, encontra-se um elemento de recursão. Eles devem obter o conteúdo da página de um URL, analisar essa página em busca de outro URL e obter *essa* página, *ad infinitum*.

Saiba, porém, que não é só porque você pode rastrear a web que deve sempre fazê-lo. Os scrapers usados nos exemplos anteriores funcionam muito bem nas situações em que todos os dados necessários estão em uma só página. Com os web crawlers, você deve ser extremamente zeloso quanto ao volume de banda que usar, fazendo o máximo de esforço para determinar se há alguma maneira de aliviar a carga do servidor consultado.

### Percorrendo um único domínio

Mesmo que você não tenha ouvido falar do Six Degrees of Wikipedia (Seis graus de separação na Wikipédia), é quase certo que já tenha ouvido falar de seu homônimo, o Six Degrees of Kevin Bacon (Seis graus de separação de Kevin Bacon). Nos dois jogos, o objetivo é associar dois assuntos improváveis (no primeiro caso, artigos da Wikipédia ligados uns aos outros, e, no segundo, atores que aparecem no mesmo filme) por uma cadeia contendo não mais do que seis elementos no total (incluindo os dois itens originais).

Por exemplo, Eric Idle atuou no filme *Polícia desmontada* (Dudley Do-Right) com Brendan Fraser, que atuou em *Ligados pelo crime* (The Air I Breathe) com Kevin Bacon<sup>1</sup>. Nesse caso, a cadeia de Eric Idle até Kevin Bacon tem apenas três elementos.

Nesta seção, iniciaremos um projeto que encontrará uma solução para o

Six Degrees of Wikipedia. Você poderá começar na [página de Eric Idle](https://en.wikipedia.org/wiki/Eric_Idle) ([https://en.wikipedia.org/wiki/Eric\\_Idle](https://en.wikipedia.org/wiki/Eric_Idle)) e encontrar o menor número de cliques em links que levarão até a [página de Kevin Bacon](https://en.wikipedia.org/wiki/Kevin_Bacon) ([https://en.wikipedia.org/wiki/Kevin\\_Bacon](https://en.wikipedia.org/wiki/Kevin_Bacon)).

## Mas e a carga do servidor da Wikipédia?

De acordo com a Wikimedia Foundation (a organização de nível mais alto, responsável pela Wikipédia), as propriedades web do site recebem aproximadamente 2.500 hits por *segundo*, com mais de 99% deles para o domínio da Wikipédia (veja a seção “Traffic Volume” [Volume de tráfego] na página “Wikimedia in Figures” [Wikimedia em números]) ([https://meta.wikimedia.org/wiki/Wikimedia\\_in\\_figures\\_-\\_Wikipedia#Traffic\\_volume](https://meta.wikimedia.org/wiki/Wikimedia_in_figures_-_Wikipedia#Traffic_volume)). Por causa do imenso volume de tráfego, é improvável que seus web scrapers causem qualquer impacto perceptível na carga do servidor da Wikipédia. No entanto, se você vai executar os códigos de exemplo deste livro intensamente, ou vai criar os próprios projetos que coletarão dados da Wikipédia, incentive você a fazer [uma doação dedutível no imposto de renda para a Wikimedia Foundation](https://wikimediafoundation.org/wiki/Ways_to_Give) ([https://wikimediafoundation.org/wiki/Ways\\_to\\_Give](https://wikimediafoundation.org/wiki/Ways_to_Give)) – não só para compensar pela carga do servidor, mas também para ajudar a deixar recursos educacionais disponíveis a outras pessoas.

Não se esqueça também de que, se o desenvolvimento de um projeto de grande porte envolvendo dados da Wikipédia estiver em seus planos, você deve verificar se esses dados já não estão disponíveis por meio da [API da Wikipédia](https://www.mediawiki.org/wiki/API:Main_page) ([https://www.mediawiki.org/wiki/API:Main\\_page](https://www.mediawiki.org/wiki/API:Main_page)). Com frequência, o site da Wikipédia é usado para demonstração de scrapers e crawlers por ter uma estrutura de HTML simples e ser relativamente estável. Contudo, suas APIs muitas vezes deixam esses mesmos dados acessíveis de modo mais eficaz.

Você já deve saber como escrever um script Python que obtenha uma página arbitrária da Wikipédia e gere uma lista de links dessa página:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://en.wikipedia.org/wiki/Kevin_Bacon')
bs = BeautifulSoup(html, 'html.parser')
for link in bs.find_all('a'):
    if 'href' in link.attrs:
        print(link.attrs['href'])
```

Se observarmos a lista de links gerada, percebemos que todos os artigos esperados estão presentes: “Apollo 13”, “Philadelphia”, “Primetime Emmy Award”, e assim por diante. No entanto, há alguns itens indesejados também:

```
//wikimediafoundation.org/wiki/Privacy_policy
//en.wikipedia.org/wiki/Wikipedia:Contact_us
```

De fato, a Wikipédia está repleta de links para caixas de texto, rodapés e cabeçalhos, presentes em todas as páginas, além de links para as páginas de categoria, páginas de discussão e outras páginas que não contêm artigos diferentes:

```
/wiki/Category:Articles_with_unsourced_statements_from_April_2014  
/wiki/Talk:Kevin_Bacon
```

Recentemente, um amigo meu, enquanto trabalhava em um projeto semelhante de coleta de dados da Wikipédia, mencionou que havia escrito uma função longa de filtragem, com mais de cem linhas de código, a fim de determinar se um link interno da Wikipédia era de uma página de artigo. Infelizmente, ele não havia investido muito tempo antes tentando encontrar padrões entre “links para artigos” e “outros links”; do contrário, ele teria descoberto o truque. Se analisarmos os links que apontam para páginas de artigos (em oposição a outras páginas internas), veremos que todos eles têm três características em comum:

- Estão na `div` com o `id` definido com `bodyContent`.
- Os URLs não contêm dois-pontos.
- Os URLs começam com `/wiki/`.

Essas regras podem ser usadas para uma pequena revisão no código a fim de obter somente os links desejados para artigos, usando a expressão regular `^(/wiki/)((?!:).)*$`:

```
from urllib.request import urlopen  
from bs4 import BeautifulSoup  
import re  
  
html = urlopen('http://en.wikipedia.org/wiki/Kevin_Bacon')  
bs = BeautifulSoup(html, 'html.parser')  
for link in bs.find('div', {'id': 'bodyContent'}).find_all(  
    'a', href=re.compile('^(/wiki/)((?!:).)*$')):  
    if 'href' in link.attrs:  
        print(link.attrs['href'])
```

Se esse código for executado, veremos uma lista de todos os URLs de artigos para os quais o artigo da Wikipédia sobre Kevin Bacon aponta.

É claro que ter um script que encontre todos os links de artigos em um único artigo da Wikipédia previamente definido, apesar de ser interessante, é um tanto quanto inútil na prática. É necessário transformar esse código em algo mais parecido com o seguinte:

- Uma única função, `getLinks`, que receba um URL de um artigo da Wikipédia no formato `/wiki/<Nome_do_Artigo>` e devolva uma lista com os URLs de todos os artigos associados, no mesmo formato.
- Uma função principal que chame `getLinks` com um artigo inicial, escolha um link de artigo aleatório na lista devolvida e chame `getLinks` novamente, até que você interrompa o programa ou nenhum link de

artigo seja encontrado na nova página.

Eis o código completo para isso:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import datetime
import random
import re

random.seed(datetime.datetime.now())
def getLinks(articleUrl):
    html = urlopen('http://en.wikipedia.org{}'.format(articleUrl))
    bs = BeautifulSoup(html, 'html.parser')
    return bs.find('div', {'id':'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))

links = getLinks('/wiki/Kevin_Bacon')
while len(links) > 0:
    newArticle = links[random.randint(0, len(links)-1)].attrs['href']
    print(newArticle)
    links = getLinks(newArticle)
```

A primeira tarefa do programa, depois de importar as bibliotecas necessárias, é definir a semente (seed) para o gerador de números aleatórios com o horário atual do sistema. Isso praticamente garante que haja um caminho aleatório novo e interessante pelos artigos da Wikipédia sempre que o programa executar.

## Números pseudoaleatórios e sementes aleatórias

No exemplo anterior, o gerador de números aleatórios de Python foi usado para selecionar um artigo aleatório em cada página a fim de continuar percorrendo a Wikipédia aleatoriamente. No entanto, os números aleatórios devem ser utilizados com cautela.

Embora sejam ótimos para calcular respostas corretas, os computadores são péssimos para inventar algo. Por esse motivo, os números aleatórios podem ser um desafio. A maioria dos algoritmos para números aleatórios se esforça em gerar uma sequência de números uniformemente distribuídos e difíceis de prever, mas um número para “semente” (seed) deve ser fornecido a esses algoritmos para que tenham um valor com o qual poderão trabalhar inicialmente. A mesma semente sempre produzirá exatamente a mesma sequência de números “aleatórios”; por esse motivo, usei o relógio do sistema para iniciar novas sequências de números aleatórios e, desse modo, novas sequências de artigos aleatórios. Isso faz com que executar o programa seja um pouco mais emocionante.

Para os curiosos, o gerador de números pseudoaleatórios de Python usa o *algoritmo Mersenne Twister*. Embora gere números aleatórios difíceis de prever e uniformemente distribuídos, ele exige um pouco do processador. Números aleatórios bons assim não são baratos!

Em seguida, o programa define a função `getLinks`, que aceita o URL de um artigo no formato `/wiki/...`, insere o nome de domínio da Wikipédia, `http://en.wikipedia.org`, como prefixo e obtém o objeto `BeautifulSoup` para o HTML que está nesse domínio. Então, uma lista de tags com links para

artigos é gerada com base nos parâmetros discutidos antes, e essa lista é devolvida.

O corpo principal do programa começa definindo uma lista de tags de links para artigos (a variável `links`) com a lista de links da página inicial: [https://en.wikipedia.org/wiki/Kevin\\_Bacon](https://en.wikipedia.org/wiki/Kevin_Bacon). Em seguida, o código executa um laço, encontrando uma tag de link aleatória para um artigo na página, extraindo o atributo `href` dela, exibindo a página e obtendo uma nova lista de links do URL extraído.

É claro que um pouco mais de trabalho é necessário para resolver o problema do Six Degrees of Wikipedia, além de construir um scraper que ande de página em página. Também é necessário armazenar e analisar os dados resultantes. Para ver uma continuação da solução desse problema, leia o *Capítulo 6*.

Trate suas exceções!

Embora esses códigos de exemplo omitam a maior parte do tratamento de exceções para que sejam mais concisos, esteja ciente de que podem surgir muitos problemas possíveis. O que aconteceria se a Wikipédia mudasse o nome da tag `bodyContent`, por exemplo? Quando o programa tentasse extrair o texto da tag, um `AttributeError` seria gerado.

Embora não haja problemas em executar esses scripts como exemplos a serem observados de perto, um código autônomo em um ambiente de produção exigirá muito mais tratamento de exceções do que seria apropriado inserir neste livro. Volte ao *Capítulo 1* para obter mais informações a esse respeito.

## Rastreando um site completo

Na seção anterior, percorremos um site aleatoriamente, indo de um link a outro. O que aconteceria se precisássemos catalogar sistematicamente ou pesquisar todas as páginas de um site? Rastrear um site completo, particularmente um site grande, é um processo que exige bastante memória, e é mais apropriado para aplicações em que um banco de dados para armazenar os resultados do rastreamento esteja prontamente disponível. No entanto, podemos explorar o comportamento desses tipos de aplicações sem executá-los de forma completa. Para saber mais sobre a execução dessas aplicações usando um banco de dados, consulte o *Capítulo 6*.

### Dark web e deep web

É provável que você já tenha ouvido os termos *deep web*, *dark web* ou *hidden web* (internet oculta) por aí, sobretudo na mídia ultimamente. O que esses termos significam?

A *deep web* é qualquer parte da web que não faz parte da *surface web*<sup>2</sup>. A superfície (surface) é a

parte da internet indexada pelas ferramentas de pesquisa. As estimativas variam bastante, mas é quase certo que a deep web constitua aproximadamente 90% da internet. Como o Google não é capaz de fazer tarefas como submeter formulários, encontrar páginas para as quais não haja links a partir de um domínio de nível mais alto ou investigar sites proibidos por um *robots.txt*, a surface web permanece relativamente pequena.

A *dark web*, também conhecida como *darknet*, é uma criatura totalmente diferente<sup>3</sup>. Ela executa na infraestrutura de hardware da rede existente, porém utiliza o Tor, ou outro cliente, com um protocolo de aplicação sobre o HTTP, oferecendo um canal seguro para troca de informações. Embora seja possível coletar dados da dark web, como faríamos com qualquer outro site, fazer isso está além do escopo deste livro.

De modo diferente da dark web, é relativamente fácil coletar dados da deep web. Muitas ferramentas neste livro ensinarão você a rastrear e a coletar informações de vários lugares que os bots do Google não conseguem alcançar.

Quando rastrear um site completo poderia ser conveniente, e quando poderia ser prejudicial? Web scrapers que percorrem um site completo são apropriados para muitas tarefas, incluindo:

### *Gerar um mapa do site*

Há alguns anos, eu me vi diante de um problema: um cliente importante queria uma estimativa para refazer o design de um site, mas não queria dar acesso interno ao seu sistema atual de gerenciamento de conteúdo à minha empresa, e não possuía um mapa do site disponível publicamente. Pude usar um crawler para percorrer o site todo, coletar todos os links internos e organizar as páginas na estrutura de pastas usada no site. Isso me permitiu encontrar rapidamente seções do site que eu nem mesmo sabia que existiam e contar exatamente quantos designs de páginas seriam necessários e o volume de conteúdo que teria de ser migrado.

### *Coletar dados*

Outro cliente meu queria coletar artigos (histórias, postagens de blog, artigos de notícia etc.) para criar um protótipo funcional de uma plataforma de pesquisa especializada. Embora esse rastreamento de sites não tivesse de ser completo, era necessário que fosse razoavelmente abrangente (estávamos interessados em obter dados apenas de alguns sites). Pude criar crawlers que percorriam recursivamente cada site e coletavam somente os dados encontrados nas páginas de artigos.

A abordagem geral para um rastreamento completo de sites consiste em começar com uma página de nível mais alto (por exemplo, a página inicial) e procurar uma lista de todos os links internos nessa página. Cada um desses links é então rastreado, e listas adicionais de links são encontradas em cada um deles, disparando outra rodada de rastreamento.

Claramente, essa é uma situação que poderia escalar rapidamente. Se cada página tiver 10 links internos, e um site tiver 5 páginas de profundidade (uma profundidade razoavelmente típica para um site de tamanho médio), o número de páginas que deverão ser rastreadas seria igual a  $10^5$ , ou seja, 100.000 páginas, para ter certeza de que o site foi exaustivamente percorrido. O estranho, porém, é que, embora “5 páginas de profundidade e 10 links internos por página” sejam dimensões razoavelmente típicas para um site, bem poucos têm 100.000 páginas ou mais. O motivo, evidentemente, é que a grande maioria dos links internos são duplicações.

Para evitar que a mesma página seja rastreada duas vezes, é extremamente importante que todos os links internos descobertos estejam formatados de modo consistente e sejam mantidos em um conjunto dinâmico para serem facilmente consultados enquanto o programa estiver executando. Um *conjunto* é semelhante a uma lista, mas os elementos não têm uma ordem específica, e somente elementos únicos são armazenados, o que é ideal para as nossas necessidades. Apenas links que sejam “novos” devem ser rastreados e pesquisados para saber se há links adicionais:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

pages = set()
def getLinks(pageUrl):
    global pages
    html = urlopen('http://en.wikipedia.org{}'.format(pageUrl))
    bs = BeautifulSoup(html, 'html.parser')
    for link in bs.find_all('a', href=re.compile('^(/wiki/)')):
        if 'href' in link.attrs:
            if link.attrs['href'] not in pages:
                #Encontramos uma página nova
                newPage = link.attrs['href']
                print(newPage)
                pages.add(newPage)
                getLinks(newPage)
getLinks('')
```

Para mostrar o efeito completo desse rastreamento de site, flexibilizei os padrões acerca do que constitui um link interno (conforme os exemplos anteriores). Em vez de limitar o scraper a páginas de artigos, ele procura todos os links que comecem com */wiki/*, independentemente de onde estiverem na página ou se contêm dois-pontos. Lembre-se de que páginas de artigos não contêm dois-pontos, mas páginas de upload de arquivos,

páginas de discussão e páginas semelhantes sim.

Inicialmente, `getLinks` é chamada com um URL vazio. Isso será traduzido para “a página frontal da Wikipédia” assim que o URL vazio receber `http://en.wikipedia.org` como prefixo na função. Então percorremos cada link da primeira página e verificamos se está no *conjunto global* de páginas (um conjunto das páginas que o script já encontrou). Se não estiver, o link é adicionado na lista, exibido na tela, e a função `getLinks` é chamada recursivamente nesse link.

### Um aviso quanto à recursão

Este é um aviso raramente visto em livros de software, mas achei que você deveria saber: se deixar o programa anterior executando por tempo suficiente, é quase certo que ele apresentará uma falha.

Python tem um limite default para recursão (o número de vezes que um programa pode chamar a si mesmo recursivamente) igual a 1.000. Como a rede de links da Wikipédia é extremamente grande, em algum momento esse programa atingirá esse limite de recursão e será interrompido, a menos que você coloque um contador de recursão ou algo para impedir que isso aconteça.

Para sites “planos”, com menos de 1.000 links de profundidade, esse método em geral funciona bem, com algumas exceções incomuns. Por exemplo, uma vez, encontrei um bug em um URL gerado dinamicamente, que dependia do endereço da página atual para escrever o link nessa página. Isso resultava em paths se repetindo infinitamente, por exemplo, `/blogs/blogs.../blogs/blog-post.php`.

Na maioria das ocasiões, porém, essa técnica recursiva não deverá apresentar problemas para qualquer site típico que você possa encontrar.

## Coletando dados de um site completo

Os web crawlers seriam razoavelmente enfadonhos se tudo que fizessem fosse pular de uma página para outra. Para torná-los úteis, é preciso ser capaz de fazer algo na página enquanto estivermos nela. Vamos ver como construir um scraper que colete o título, o primeiro parágrafo do conteúdo e o link para editar a página (se estiver disponível).

Como sempre, o primeiro passo para determinar a melhor maneira de fazer isso é observar algumas páginas do site e estabelecer um padrão. Ao observar algumas páginas da Wikipédia (páginas tanto de artigos quanto outras, como a página de política de privacidade), os aspectos a seguir deverão se tornar evidentes:

- Todos os títulos (em todas as páginas, independentemente de seu status como sendo uma página de artigo, uma página de histórico de modificações ou qualquer outra página) estão em tags `h1` → `span`, e essas são as únicas tags `h1` na página.

- Conforme mencionamos antes, todo o texto do corpo encontra-se na tag `div#bodyContent`. Entretanto, se você quiser ser mais específico e acessar somente o primeiro parágrafo do texto, talvez seja melhor usar `div#mw-content-text` → `p` (selecionando apenas a tag do primeiro parágrafo). Isso vale para todas as páginas de conteúdo, exceto as páginas de arquivos (por exemplo, [https://en.wikipedia.org/wiki/File:Orbit\\_of\\_274301\\_Wikipedia.svg](https://en.wikipedia.org/wiki/File:Orbit_of_274301_Wikipedia.svg)), que não têm seções textuais de conteúdo.
- Links para edição estão presentes somente nas páginas de artigos. Se estiverem presentes, serão encontrados na tag `li#ca-edit`, em `li#ca-edit` → `span` → `a`.

Ao modificar nosso código básico de rastreamento, podemos criar um programa que combine rastreamento/coleta de dados (ou, pelo menos, exibição de dados):

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

pages = set()
def getLinks(pageUrl):
    global pages
    html = urlopen('http://en.wikipedia.org{}'.format(pageUrl))
    bs = BeautifulSoup(html, 'html.parser')
    try:
        print(bs.h1.get_text())
        print(bs.find(id='mw-content-text').find_all('p')[0])
        print(bs.find(id='ca-edit').find('span')
              .find('a').attrs['href'])
    except AttributeError:
        print('This page is missing something! Continuing.')

    for link in bs.find_all('a', href=re.compile('^(/wiki/)')):
        if 'href' in link.attrs:
            if link.attrs['href'] not in pages:
                #Encontramos uma página nova
                newPage = link.attrs['href']
                print('- '*20)
                print(newPage)
                pages.add(newPage)
                getLinks(newPage)

getLinks('')
```

O laço `for` nesse programa é essencialmente o mesmo do programa original de rastreamento (com o acréscimo de traços para clareza, separando o

conteúdo exibido).

Como não é possível ter certeza absoluta de que todos os dados estejam em cada página, cada instrução `print` é organizada na ordem que é mais provável de aparecer no site. Isto é, a tag de título `h1` aparece em todas as páginas (tanto quanto posso afirmar, pelo menos), então tentamos obter esses dados antes. O conteúdo de texto aparece na maioria das páginas (exceto nas páginas de arquivos), portanto esse é o segundo dado obtido. O botão Edit (Editar) aparece somente nas páginas em que tanto o título quanto o conteúdo já existem, mas não está presente em todas as páginas.

### Padrões diferentes para necessidades diferentes

Obviamente, há alguns riscos envolvidos ao englobar várias linhas em um handler de exceção. Não é possível dizer qual das linhas lançou a exceção, para começar. Além do mais, se, por algum motivo, uma página contiver um botão Edit, mas não tiver um título, o botão não será registrado. No entanto, é suficiente para muitos casos em que haja uma ordem provável de itens presentes no site, e perder alguns pontos de dados inadvertidamente ou deixar de manter logs detalhados não seja um problema.

Você deve ter percebido que, nesse exemplo e nos exemplos anteriores, não “coletamos” dados, apenas os “exibimos”. Obviamente, é difícil manipular dados em seu terminal. Veremos mais detalhes sobre como armazenar informações e criar bancos de dados no *Capítulo 5*.

## Lidando com redirecionamentos

Os redirecionamentos permitem que um servidor web aponte um nome de domínio ou um URL para um conteúdo em um local diferente. Há dois tipos de redirecionamentos:

- Redirecionamentos do lado do servidor, em que o URL é alterado antes de a página ser carregada.
- Redirecionamentos do lado cliente, às vezes visto com uma mensagem do tipo “Você será redirecionado em 10 segundos”, em que a página é carregada antes do redirecionamento para a nova página.

Com redirecionamentos do lado do servidor, em geral não é preciso se preocupar. Se você estiver usando a biblioteca `urllib` com Python 3.x, ela tratará os redirecionamentos automaticamente! Se a biblioteca `requests` estiver sendo usada, não se esqueça de definir a flag para permitir redirecionamentos com `True`:

```
r = requests.get('http://github.com', allow_redirects=True)
```

Saiba somente que, ocasionalmente, o URL da página que você estiver rastreando pode não ser exatamente o URL com o qual você entrou nela.

Para mais informações sobre redirecionamentos do lado cliente, executados usando JavaScript ou HTML, consulte o *Capítulo 12*.

## Rastreando pela internet

Toda vez que faço uma apresentação sobre web scraping, alguém inevitavelmente pergunta: “Como você constrói um Google?”. Minha

resposta sempre tem duas partes: “Em primeiro lugar, obtenha muitos bilhões de dólares para comprar os maiores armazéns de dados (data warehouses) do mundo e coloque-os em lugares secretos por todo o planeta. Em segundo lugar, construa um web crawler”.

Quando o Google começou em 1996, havia somente dois alunos formados em Stanford com um velho servidor e um web crawler Python. Agora que você sabe como coletar dados da web, oficialmente já tem as ferramentas necessárias para se tornar o próximo multibilionário da tecnologia!

Falando seriamente, os web crawlers estão no coração de muitas das tecnologias web modernas, e não precisamos necessariamente de um grande armazém de dados para usá-los. Para fazer qualquer análise de dados envolvendo vários domínios, precisamos construir crawlers capazes de interpretar e de armazenar dados de uma série de páginas da internet.

Assim como no exemplo anterior, os web crawlers que você desenvolverá seguirão links de uma página a outra, construindo um mapa da web. Dessa vez, porém, os links externos não serão ignorados: eles serão seguidos.

#### Águas desconhecidas à frente

Tenha em mente que o código da próxima seção pode acessar *qualquer lugar* na internet. Se aprendemos algo com o Six Degrees of Wikipedia, é que é totalmente possível ir de um site como <http://www.sesamestreet.org/> para algo menos respeitável com apenas alguns saltos.

Crianças, peçam permissão aos pais antes de executar esse código. Para aqueles que tenham constituições mais sensíveis ou que possuam restrições religiosas que possam proibir ver textos de um site lascivo, prossigam lendo os códigos de exemplo, mas tomem cuidado ao executá-los.

Antes de começar a escrever um crawler que siga todos os links de saída cegamente, você deve se fazer algumas perguntas:

- Quais dados estou tentando obter? Isso pode ser feito coletando dados de apenas alguns sites predefinidos (quase sempre, é a opção mais simples), ou meu crawler deve ser capaz de descobrir novos sites que eu talvez desconheça?
- Quando meu crawler alcançar um site em particular, ele seguirá imediatamente o próximo link de saída para um novo site ou permanecerá por um tempo no site para explorar os níveis mais profundos?
- Há alguma condição para eu não querer coletar dados de um site em particular? Estou interessado em um conteúdo em outro idioma?
- Como estou me protegendo de uma ação legal caso meu web crawler

chame a atenção de um webmaster de um dos sites que ele acessar? (Veja o *Capítulo 18* para obter mais informações sobre esse assunto.)

Um conjunto flexível de funções Python que podem ser combinadas para executar vários tipos de web scraping pode ser facilmente escrito com menos de 60 linhas de código:

```
from urllib.request import urlopen
from urllib.parse import urlparse
from bs4 import BeautifulSoup
import re
import datetime
import random

pages = set()
random.seed(datetime.datetime.now())

#Obtém uma lista de todos os links internos encontrados em uma página
def getInternalLinks(bs, includeUrl):
    includeUrl = '{}://{}'.format(urlparse(includeUrl).scheme,
        urlparse(includeUrl).netloc)
    internalLinks = []
    #Encontra todos os links que começam com "/"
    for link in bs.find_all('a',
        href=re.compile('^(/|.*/'+includeUrl+'')')):
        if link.attrs['href'] is not None:
            if link.attrs['href'] not in internalLinks:
                if(link.attrs['href'].startswith('/')):
                    internalLinks.append(
                        includeUrl+link.attrs['href'])
                else:
                    internalLinks.append(link.attrs['href'])
    return internalLinks

#Obtém uma lista de todos os links externos encontrados em uma página
def getExternalLinks(bs, excludeUrl):
    externalLinks = []
    #Encontra todos os links que começam com "http" e que
    #não contenham o URL atual
    for link in bs.find_all('a',
        href=re.compile('^((http|www)(?!'+excludeUrl+'.)*$'))):
        if link.attrs['href'] is not None:
            if link.attrs['href'] not in externalLinks:
                externalLinks.append(link.attrs['href'])
    return externalLinks

def getRandomExternalLink(startingPage):
    html = urlopen(startingPage)
    bs = BeautifulSoup(html, 'html.parser')
    externalLinks = getExternalLinks(bs,
```

```

urlparse(startingPage).netloc)
if len(externalLinks) == 0:
    print('No external links, looking around the site for one')
    domain = '{}://{}'.format(urlparse(startingPage).scheme,
        urlparse(startingPage).netloc)
    internalLinks = getInternalLinks(bs, domain)
    return getRandomExternalLink(internalLinks[random.randint(0,
        len(internalLinks)-1)])
else:
    return externalLinks[random.randint(0, len(externalLinks)-1)]

def followExternalOnly(startingSite):
    externalLink = getRandomExternalLink(startingSite)
    print('Random external link is: {}'.format(externalLink))
    followExternalOnly(externalLink)

followExternalOnly('http://oreilly.com')

```

O programa anterior começa em <http://oreilly.com> e pula aleatoriamente de um link externo para outro link externo. Eis um exemplo da saída que ele gera:

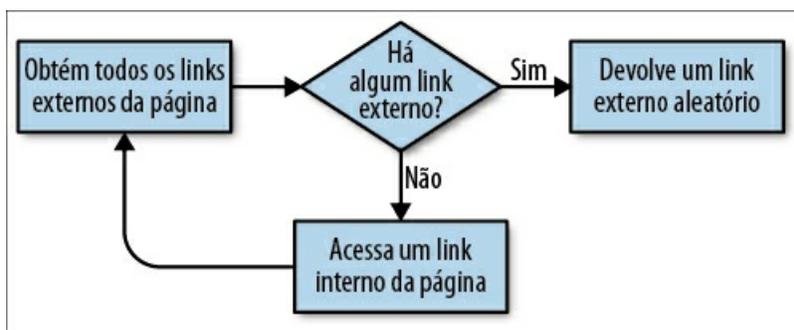
```

http://igniteshow.com/
http://feeds.feedburner.com/oreilly/news
http://hire.jobvite.com/CompanyJobs/Careers.aspx?c=q319
http://makerfaire.com/

```

Nem sempre é possível garantir que links externos serão encontrados na primeira página de um site. Para encontrar links externos nesse caso, um método semelhante àquele usado no exemplo anterior de rastreamento é empregado para explorar recursivamente os níveis mais profundos de um site até que um link externo seja encontrado.

A Figura 3.1 mostra a operação na forma de um fluxograma.



*Figura 3.1 – Fluxograma de nosso script que rastreia sites na internet.*

Não coloque os programas de exemplo no ambiente de produção

Fico repetindo isso, mas, em razão do espaço e da legibilidade, os programas de exemplo deste livro nem sempre contêm as verificações e os tratamentos de exceção necessários a um

código apropriado para um ambiente de produção. Por exemplo, se um link externo não estiver presente em nenhum lugar em um site encontrado por esse crawler, (é improvável, mas pode acontecer em algum momento se você o executar por tempo suficiente), esse programa continuará executando até atingir o limite de recursão de Python.

Uma maneira simples de melhorar a robustez desse crawler seria combiná-lo com o código de tratamento de exceção para a conexão, que vimos no *Capítulo 1*. Isso permitiria que o código escolhesse um URL diferente para acessar caso um erro de HTTP ou uma exceção de servidor ocorresse ao acessar a página.

Antes de executar esse código com qualquer finalidade séria, certifique-se de estar colocando verificações para lidar com possíveis problemas.

O aspecto interessante de separar tarefas em funções simples como “encontrar todos os links externos dessa página” é que o código pode ser facilmente refatorado no futuro para executar uma tarefa diferente de rastreamento. Por exemplo, se seu objetivo é rastrear um site todo em busca de links externos e tomar nota de cada um deles, a seguinte função pode ser acrescentada:

```
# Coleta uma lista de todos os URLs externos encontrados no site
allExtLinks = set()
allIntLinks = set()

def getAllExternalLinks(siteUrl):
    html = urlopen(siteUrl)
    domain = '{}://{}'.format(urlparse(siteUrl).scheme,
                              urlparse(siteUrl).netloc)
    bs = BeautifulSoup(html, 'html.parser')
    internalLinks = getInternalLinks(bs, domain)
    externalLinks = getExternalLinks(bs, domain)

    for link in externalLinks:
        if link not in allExtLinks:
            allExtLinks.add(link)
            print(link)
    for link in internalLinks:
        if link not in allIntLinks:
            allIntLinks.add(link)
            getAllExternalLinks(link)

allIntLinks.add('http://oreilly.com')
getAllExternalLinks('http://oreilly.com')
```

Podemos pensar nesse código como tendo dois laços – um que coleta links internos, e outro que coleta links externos – funcionando em conjunto. O fluxograma tem o aspecto mostrado na Figura 3.2.

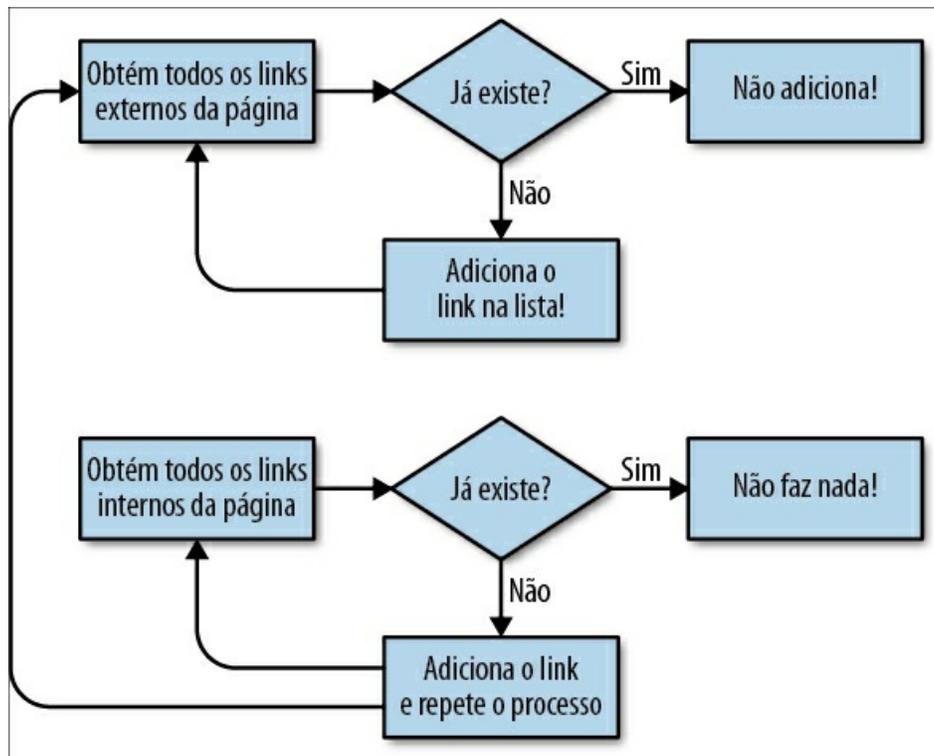


Figura 3.2 – Fluxograma do web crawler que coleta todos os links externos.

Esboçar ou criar diagramas sobre o que o código deverá fazer antes de escrevê-lo é um excelente hábito a ser adquirido, e que pode fazer você economizar bastante tempo e evitar frustrações à medida que seus crawlers se tornarem mais complicados.

<sup>1</sup> Obrigado ao [The Oracle of Bacon](http://oracleofbacon.org) (<http://oracleofbacon.org>) por satisfazer minha curiosidade sobre essa cadeia em particular.

<sup>2</sup> Veja o artigo [“Exploring a ‘Deep Web’ that Google Can’t Grasp”](http://nyti.ms/2pohZmu) (<http://nyti.ms/2pohZmu>, Explorando a ‘deep web’ que o Google não alcança) de Alex Wright.

<sup>3</sup> Veja o artigo [“Hacker Lexicon: What is the Dark Web?”](http://bit.ly/2psIw2M) (<http://bit.ly/2psIw2M>, Hacker Lexicon: o que é a dark web?) de Andy Greenberg.

# Modelos de web crawling

Escrever um código organizado e escalável já é difícil quando temos controle sobre os dados e as entradas. Escrever código para web crawlers, que podem coletar e armazenar uma variedade de dados de conjuntos diversos de sites sobre os quais o programador não tem controle algum, muitas vezes representa desafios únicos quanto à organização.

Talvez lhe peçam que colete artigos de notícias ou postagens de blog de diversos sites, cada um com templates e layouts distintos. A tag `h1` de um site contém o título do artigo, outra tag `h1` contém o título do próprio site e o título do artigo está em `<span id="title">`.

Talvez seja necessário ter um controle flexível sobre os sites dos quais os dados são coletados e como foram coletados, e uma forma de adicionar novos sites ou modificar os sites existentes rapidamente, sem escrever várias linhas de código.

Pode ser que lhe peçam que colete preços de produtos de sites diferentes, com o objetivo de comparar os preços de produtos iguais. Quem sabe esses preços estejam em moedas distintas, e talvez seja necessário também combinar isso com dados externos de outras fontes que não estejam na internet.

Embora as aplicações dos web crawlers sejam quase ilimitadas, crawlers grandes e escaláveis tendem a se enquadrar em um de diversos padrões. Ao conhecer esses padrões e reconhecer as situações em que eles se aplicam, podemos melhorar bastante a manutenibilidade e a robustez dos web crawlers.

Este capítulo tem como foco principalmente os web crawlers que coletam um número limitado de “tipos” de dados (por exemplo, avaliações de restaurantes, artigos de notícias, perfis de empresas) de uma variedade de sites, e armazenam esses tipos de dados na forma de objetos Python, lidos e escritos em um banco de dados.

## Planejando e definindo objetos

Uma armadilha comum no web scraping é definir os dados que desejamos coletar com base exclusivamente no que está disponível diante de nossos olhos. Por exemplo, se quiséssemos coletar dados de produtos, poderíamos observar inicialmente uma loja de vestuário e decidir que os dados coletados de cada produto deverão ter os campos a seguir:

- nome do produto;
- preço;
- descrição;
- tamanhos;
- cores;
- tipo do tecido;
- avaliação dos clientes.

Observando outro site, percebemos que há SKUs (Stock Keeping Units, ou Unidades de Controle de Estoque, usados para controle e pedido de itens) listados na página. Certamente vamos querer coletar esses dados também, apesar de não estarem presentes no primeiro site. Acrescentamos este campo:

- SKU do item;

Embora vestuários sejam um ótimo ponto de partida, também queremos garantir que seja possível estender esse crawler para outros tipos de produtos. Começamos a analisar as seções de produtos de outros sites e decidimos que também será necessário coletar as seguintes informações:

- capa dura/brochura;
- impressão fosca/brilhante;
- número de avaliações de clientes;
- link para o fabricante.

Essa é uma abordagem claramente insustentável. Simplesmente adicionar atributos ao tipo do produto sempre que você ver novas informações em um site resultará em uma quantidade excessiva de campos para controlar. Além disso, sempre que coletar dados de um novo site, você será forçado a executar uma análise detalhada dos campos que o site tiver e dos campos acumulados até agora e, possivelmente, acrescentar novos campos (modificando seu tipo de objeto Python e a estrutura de seu banco de

dados). Você terá um conjunto de dados confuso e difícil de ler, o qual poderá resultar em problemas para usá-lo.

Uma das melhores atitudes que podem ser tomadas ao decidir quais dados devem ser coletados muitas vezes é ignorar totalmente os sites. Não comece um projeto previsto para ser grande e escalável observando um único site e dizendo “O que é que existe?”, mas pergunte “O que é que eu preciso?”, e então encontre maneiras de buscar as informações necessárias a partir daí.

Talvez o que você realmente queira fazer seja comparar preços de produtos de várias lojas e monitorar esses preços com o passar do tempo. Nesse caso, é necessário ter informações suficientes para identificar unicamente o produto, isto é:

- nome do produto;
- fabricante;
- número de ID do produto (se estiver disponível ou for relevante).

É importante observar que nenhuma dessas informações é específica de uma loja em particular. Por exemplo, avaliações do produto, classificações, preço e até mesmo as descrições são específicas da instância desse produto em uma loja em particular. Essas informações podem ser armazenadas separadamente.

Outras informações (cores disponíveis para o produto, do que ele é feito) são específicas do produto, mas podem ser esparsas – não são aplicáveis a todos os produtos. É importante dar um passo para trás e criar uma lista de verificações para cada item a ser considerado e fazer as seguintes perguntas a si mesmo:

- Essa informação contribuirá para alcançar os objetivos do projeto? Haverá algum impedimento caso eu não a tenha, ou é apenas “bom tê-la”, mas, em última análise, não causará nenhum impacto?
- Se *puder* ajudar no futuro, mas não estou certo disso, qual é a dificuldade de voltar e coletar os dados mais tarde?
- Esses dados são redundantes considerando os dados que já coletei?
- Faz sentido, do ponto de vista lógico, armazenar os dados nesse objeto em particular? (Como mencionamos antes, armazenar uma descrição em um produto não faz sentido se essa descrição mudar de site para site para o mesmo produto.)

Se você realmente decidir que deve coletar os dados, é importante fazer mais algumas perguntas para, então, decidir como armazená-los e tratá-los no código:

- Esses dados são esparsos ou densos? Eles serão relevantes e estarão presentes em todas as listas, ou estarão em apenas alguns do conjunto?
- Quão grandes são os dados?
- Sobretudo no caso de dados grandes, será necessário obtê-los regularmente sempre que eu executar minha análise, ou apenas ocasionalmente?
- Quão variável é esse tipo de dado? Terei de acrescentar regularmente novos atributos, modificar tipos (por exemplo, padrões de tecido, que podem ser acrescentados com frequência), ou eles são imutáveis (tamanho dos calçados)?

Suponha que você está planejando fazer algumas análises metarrelacionadas a atributos e preços de produtos: por exemplo, o número de páginas de um livro ou o tipo de tecido com que uma peça de vestuário é confeccionada e, possivelmente, outros atributos no futuro, correlacionados a preços. Você analisa as perguntas e percebe que esses dados são esparsos (relativamente poucos produtos têm um desses atributos), e que existe a possibilidade de acrescentar ou remover atributos com frequência. Nesse caso, faz sentido criar um tipo de produto como este:

- nome do produto;
- fabricante;
- número de ID do produto (se estiver disponível ou for relevante).
- atributos (lista ou dicionário opcional).

E um tipo de atributo como este:

- nome do atributo;
- valor do atributo.

Com isso, é possível acrescentar novos atributos de produtos de forma flexível com o passar do tempo, sem que seja necessário refazer o design de seu esquema de dados nem reescrever o código. Ao decidir como armazenar esses atributos no banco de dados, podemos usar JSON no campo `attribute`, ou armazenar cada atributo em uma tabela separada com um ID de produto. Veja o *Capítulo 6*, que tem mais informações sobre

como implementar esses tipos de modelos de banco de dados.

As perguntas anteriores podem ser aplicadas a outras informações que devam ser armazenadas também. Para manter o controle dos preços encontrados para cada produto, é provável que as seguintes informações sejam necessárias:

- ID do produto;
- ID da loja;
- preço;
- data/timestamp de quando o preço foi encontrado.

Mas e se houver uma situação em que os atributos do produto na verdade modifiquem seu preço? Por exemplo, as lojas podem cobrar mais por uma camisa grande do que por uma camisa pequena, porque a camisa grande exige mais trabalho ou mais material. Nesse caso, podemos considerar separar o produto único em itens separados para cada tamanho de camisa nas listas (de modo que cada camisa receba preços de forma independente) ou criar um novo tipo de item para armazenar informações sobre as instâncias de um produto, contendo os campos a seguir:

- ID do produto;
- tipo da instância (o tamanho da camisa, nesse caso).

E cada preço então teria os seguintes dados:

- ID da instância do produto;
- ID da loja;
- preço;
- data/timestamp de quando o preço foi encontrado.

Embora o assunto “produtos e preços” pareça específico demais, as perguntas básicas que você deve fazer a si mesmo, e a lógica usada ao fazer o design de seus objetos Python, se aplicam a quase todas as situações.

Se estiver coletando dados de artigos de notícias, talvez você queira informações básicas como estas:

- título;
- autor;
- data;
- conteúdo.

Entretanto, suponha que alguns artigos contenham uma “data de revisão”

ou “artigos relacionados” ou o “número de compartilhamentos em redes sociais”. Precisamos dessas informações? Elas são relevantes para o projeto? Como podemos armazenar o número de compartilhamentos em redes sociais de forma eficiente e flexível quando nem todos os sites de notícias usam todas as formas de redes sociais, e os sites de redes sociais podem ter mais ou menos popularidade com o tempo?

Diante de um novo projeto, pode ser tentador mergulhar de cabeça e começar a escrever código Python para coletar dados dos sites imediatamente. O modelo de dados, deixado para ser pensado depois, muitas vezes passa a ser fortemente influenciado pela disponibilidade e pelo formato dos dados coletados no primeiro site.

No entanto, o modelo de dados é a base para todo o código que o utilize. Uma decisão ruim quanto ao seu modelo pode facilmente levar a problemas para escrever e manter o código no futuro, ou a dificuldades para extrair e usar os dados resultantes de modo eficaz. Ao lidar particularmente com vários sites – tanto conhecidos quanto desconhecidos –, é essencial pensar seriamente e planejar quais são, exatamente, os dados que devemos coletar e como devemos armazená-los.

## **Lidando com diferentes layouts de sites**

Uma das proezas mais impressionantes de uma ferramenta de pesquisa como o Google é o fato de conseguir extrair dados relevantes e úteis de uma variedade de sites sem ter nenhum conhecimento prévio da estrutura do site em questão. Embora nós, seres humanos, sejamos capazes de identificar prontamente o título e o conteúdo principal de uma página (exceto nos casos de um design extremamente ruim do site), é muito mais difícil fazer com que um bot faça o mesmo.

Felizmente, na maioria dos casos de web crawling, a intenção não é coletar dados de sites que nunca vimos antes, mas de alguns sites – ou de algumas dúzias deles – previamente selecionados por um ser humano. Isso significa que não é necessário usar algoritmos complicados nem aprendizado de máquina (machine learning) para detectar qual texto na página “se parece mais com um título” ou qual texto, provavelmente, corresponde ao “conteúdo principal”. Podemos determinar quais são esses elementos manualmente.

A abordagem mais óbvia é escrever um web crawler ou um parser de

página separado para cada site. Cada um deles pode aceitar um URL, uma string ou um objeto BeautifulSoup, e devolver um objeto Python com os dados coletados.

A seguir, apresentamos um exemplo de uma classe Content (que representa o conteúdo de um site, por exemplo, um artigo de notícia) e duas funções de coleta de dados que aceitam um objeto BeautifulSoup e devolvem uma instância de Content:

```
import requests

class Content:
    def __init__(self, url, title, body):
        self.url = url
        self.title = title
        self.body = body

def getPage(url):
    req = requests.get(url)
    return BeautifulSoup(req.text, 'html.parser')

def scrapeNYTimes(url):
    bs = getPage(url)
    title = bs.find("h1").text
    lines = bs.find_all("p", {"class":"story-content"})
    body = '\n'.join([line.text for line in lines])
    return Content(url, title, body)

def scrapeBrookings(url):
    bs = getPage(url)
    title = bs.find("h1").text
    body = bs.find("div",{"class","post-body"}).text
    return Content(url, title, body)

url = 'https://www.brookings.edu/blog/future-development/'
    '/2018/01/26/delivering-inclusive-urban-access-3-unc/'
    'omfortable-truths/'
content = scrapeBrookings(url)
print('Title: {}'.format(content.title))
print('URL: {}\n'.format(content.url))
print(content.body)

url = 'https://www.nytimes.com/2018/01/25/opinion/sunday/'
    'silicon-valley-immortality.html'
content = scrapeNYTimes(url)
print('Title: {}'.format(content.title))
print('URL: {}\n'.format(content.url))
print(content.body)
```

À medida que começamos a acrescentar funções de coleta de dados para

novos sites de notícias, poderemos notar um padrão se formando. Toda função de parsing de sites faz essencialmente o mesmo:

- seleciona o elemento de título e extrai o texto do título;
- seleciona o conteúdo principal do artigo;
- seleciona outros itens de conteúdo conforme for necessário;
- devolve um objeto `Content` instanciado com as strings encontradas antes.

As únicas variáveis que realmente dependem do site, nesse caso, são os seletores CSS usados para obter cada informação. As funções `find` e `find_all` do BeautifulSoup aceitam dois argumentos: uma tag na forma de string e um dicionário de atributos chave/valor. Assim, podemos passar esses argumentos como parâmetros que definem a estrutura do próprio site e a localização dos dados desejados.

Para que tudo fique mais conveniente ainda, em vez de lidar com todos esses argumentos de tag e pares chave/valor, a função `select` do BeautifulSoup pode ser usada com uma única string de seletor CSS para cada informação que queremos coletar, e podemos colocar todos esses seletores em um objeto de dicionário:

```
class Content:
    """
    Classe-base comum para todos os artigos/páginas
    """

    def __init__(self, url, title, body):
        self.url = url
        self.title = title
        self.body = body

    def print(self):
        """
        Uma função flexível de exibição controla a saída
        """
        print("URL: {}".format(self.url))
        print("TITLE: {}".format(self.title))
        print("BODY:\n{}".format(self.body))

class Website:
    """
    Contém informações sobre a estrutura do site
    """

    def __init__(self, name, url, titleTag, bodyTag):
        self.name = name
        self.url = url
```

```
self.titleTag = titleTag
self.bodyTag = bodyTag
```

Observe que a classe `Website` não armazena informações coletadas das páginas individuais, mas instruções sobre *como* coletar esses dados. Ela não armazena o título “Título da minha página”. Ela simplesmente armazena a string de tag `h1` que indica o lugar em que os títulos podem ser encontrados. É por isso que a classe se chama `Website` (as informações nessa classe são pertinentes a todo site), e não `Content` (que contém informações de apenas uma única página).

Ao usar as classes `Content` e `Website`, podemos então escrever um `Crawler` para coletar o título e o conteúdo de qualquer URL fornecido para uma dada página web de um dado site:

```
import requests
from bs4 import BeautifulSoup

class Crawler:

    def getPage(self, url):
        try:
            req = requests.get(url)
        except requests.exceptions.RequestException:
            return None
        return BeautifulSoup(req.text, 'html.parser')

    def safeGet(self, pageObj, selector):
        """
        Função utilitária usada para obter uma string de conteúdo de um
        objeto BeautifulSoup e um seletor. Devolve uma string
        vazia caso nenhum objeto seja encontrado para o dado seletor
        """
        selectedElems = pageObj.select(selector)
        if selectedElems is not None and len(selectedElems) > 0:
            return '\n'.join(
                [elem.get_text() for elem in selectedElems])
        return ''

    def parse(self, site, url):
        """
        Extraí conteúdo de um dado URL de página
        """
        bs = self.getPage(url)
        if bs is not None:
            title = self.safeGet(bs, site.titleTag)
            body = self.safeGet(bs, site.bodyTag)
            if title != '' and body != '':
                content = Content(url, title, body)
```

```
content.print()
```

Eis o código que define os objetos do site e dá início ao processo:

```
crawler = Crawler()

siteData = [
    ['0\Reilly Media', 'http://oreilly.com',
     'h1', 'section#product-description'],
    ['Reuters', 'http://reuters.com', 'h1',
     'div.StandardArticleBody_body_1gnLA'],
    ['Brookings', 'http://www.brookings.edu',
     'h1', 'div.post-body'],
    ['New York Times', 'http://nytimes.com',
     'h1', 'p.story-content']
]
websites = []
for row in siteData:
    websites.append(Website(row[0], row[1], row[2], row[3]))

crawler.parse(websites[0], 'http://shop.oreilly.com/product/'\
               '0636920028154.do')
crawler.parse(websites[1], 'http://www.reuters.com/article/'\
               'us-usa-epa-pruitt-idUSKBN19W2D0')
crawler.parse(websites[2], 'https://www.brookings.edu/blog/'\
               'techtank/2016/03/01/idea-to-retire-old-methods-of-policy-education/')
crawler.parse(websites[3], 'https://www.nytimes.com/2018/01/'\
               '28/business/energy-environment/oil-boom.html')
```

Embora, à primeira vista, esse novo método não pareça excepcionalmente mais simples do que escrever uma nova função Python para cada novo site, pense no que acontecerá se você passar de um sistema com 4 sites para um sistema com 20 ou 200.

Cada lista de strings é relativamente fácil de escrever. Ela não ocupa muito espaço e pode ser carregada a partir de um banco de dados ou de um arquivo CSV. Pode ser importada de uma fonte remota, ou podemos entregá-la para uma pessoa que não seja um programador, mas tenha alguma experiência em frontend, a fim de que preencha e acrescente novos sites, sem que ela jamais tenha de olhar para uma única linha de código.

É claro que a desvantagem é que teremos de abrir mão de certa dose de flexibilidade. No primeiro exemplo, cada site recebe a própria função em formato livre, na qual selecionamos e fazemos parse de HTML conforme necessário a fim de obter o resultado. No segundo exemplo, cada site deve ter uma determinada estrutura em que se garante que os campos existam, os dados devem ser limpos quando extraídos do campo e cada campo desejado deve ter um seletor CSS único e confiável.

No entanto, acredito que a eficácia e a relativa flexibilidade dessa abordagem mais que compensam suas desvantagens reais ou percebidas. Na próxima seção, descreveremos aplicações e expansões específicas desse template básico para que você possa, por exemplo, lidar com campos ausentes, coletar diferentes tipos de dados, rastrear somente partes específicas de um site e armazenar informações mais complexas sobre as páginas.

## **Estruturando os crawlers**

Criar tipos de layout flexíveis e modificáveis para os sites não é muito vantajoso se ainda tivermos de localizar manualmente cada link do qual queremos coletar dados. O capítulo anterior mostrou vários métodos de rastreamento de sites e como encontrar novas páginas de forma automatizada.

Esta seção descreve como incorporar esses métodos em um web crawler bem estruturado e expansível, capaz de coletar links e descobrir dados de modo automatizado. Apresentarei apenas três estruturas básicas de web crawlers, embora acredite que eles se aplicam à maioria das situações com as quais provavelmente você vai deparar quando rastrear sites por aí, talvez com algumas modificações aqui e ali. Caso se veja em uma situação incomum, com os próprios problemas de rastreamento, também espero que seja possível usar essas estruturas como inspiração para criar um design elegante e robusto para o crawler.

## **Rastreamento de sites por meio de pesquisa**

Um dos modos mais fáceis de rastrear um site é usar o mesmo método utilizado pelos seres humanos: a caixa de pesquisa. Embora o processo de pesquisar um site em busca de uma palavra-chave ou um tópico e obter uma lista dos resultados da pesquisa pareça uma tarefa muito variável de site para site, vários pontos essenciais fazem com que isso seja surpreendentemente trivial:

- Na maioria dos sites, uma lista dos resultados de busca para um tópico em particular é obtida passando esse tópico como uma string por meio de um parâmetro no URL. Por exemplo: `http://example.com?search=meuTopico`. A primeira parte desse URL pode ser salva como uma propriedade do objeto `Website`, e o tópico pode ser simplesmente

concatenado aí.

- Após a pesquisa, a maioria dos sites apresenta as páginas resultantes como uma lista facilmente identificável de links, em geral com uma tag conveniente ao redor, como `<span class="result">`, cujo formato exato também pode ser armazenado como uma propriedade do objeto `Website`.
- Cada *link de resultado* é um URL relativo (por exemplo, `/articles/page.html`) ou um URL absoluto (por exemplo, `http://example.com/articles/page.html`). O fato de esperarmos um URL absoluto ou relativo pode ser armazenado como uma propriedade do objeto `Website`.
- Depois de ter localizado e normalizado os URLs na página de pesquisa, reduzimos o problema ao exemplo da seção anterior, com sucesso – extrair dados de uma página, dado um formato de site.

Vamos observar uma implementação desse algoritmo no código. A classe `Content` é praticamente a mesma dos exemplos anteriores. Acrescentamos a propriedade `URL` para manter o controle dos lugares em que o conteúdo foi encontrado:

```
class Content:
    """Classe-base comum para todos os artigos/páginas"""

    def __init__(self, topic, url, title, body):
        self.topic = topic
        self.title = title
        self.body = body
        self.url = url

    def print(self):
        """
        Uma função flexível de exibição controla a saída
        """
        print("New article found for topic: {}".format(self.topic))
        print("TITLE: {}".format(self.title))
        print("BODY:\n{}".format(self.body))
        print("URL: {}".format(self.url))
```

Algumas propriedades novas foram acrescentadas na classe `Website`. `searchUrl` define o local a ser acessado para obter os resultados de pesquisa se o tópico que você estiver procurando for concatenado. `resultListing` define a “caixa” que contém as informações de cada resultado, e `resultUrl` define a tag dentro dessa caixa, que fornecerá o URL exato do resultado. A propriedade `absoluteUrl` é um booleano que informa se esses resultados de busca são URLs absolutos ou relativos.

```

class Website:
    """Contém informações sobre a estrutura do site"""

    def __init__(self, name, url, searchUrl, resultListing,
                 resultUrl, absoluteUrl, titleTag, bodyTag):
        self.name = name
        self.url = url
        self.searchUrl = searchUrl
        self.resultListing = resultListing
        self.resultUrl = resultUrl
        self.absoluteUrl=absoluteUrl
        self.titleTag = titleTag
        self.bodyTag = bodyTag

```

*crawler.py* foi um pouco expandido e contém os dados de nosso `Website`, uma lista de tópicos a serem pesquisados e dois laços que iteram por todos os tópicos e em todos os sites. Também contém uma função `search` para navegar até a página de pesquisa de um site com um tópico em particular e extrai todos os URLs resultantes, listados nessa página.

```

import requests
from bs4 import BeautifulSoup

class Crawler:

    def getPage(self, url):
        try:
            req = requests.get(url)
        except requests.exceptions.RequestException:
            return None
        return BeautifulSoup(req.text, 'html.parser')

    def safeGet(self, pageObj, selector):
        childObj = pageObj.select(selector)
        if childObj is not None and len(childObj) > 0:
            return childObj[0].get_text()
        return ""

    def search(self, topic, site):
        """
        Pesquisa um dado site em busca de um dado tópico e registra
        todas as páginas encontradas
        """
        bs = self.getPage(site.searchUrl + topic)
        searchResults = bs.select(site.resultListing)
        for result in searchResults:
            url = result.select(site.resultUrl)[0].attrs["href"]
            # Verifica se é um URL relativo ou absoluto
            if(site.absoluteUrl):
                bs = self.getPage(url)

```

```

else:
    bs = self.getPage(site.url + url)
if bs is None:
    print("Something was wrong with that page or URL. Skipping!")
    return
title = self.safeGet(bs, site.titleTag)
body = self.safeGet(bs, site.bodyTag)
if title != '' and body != '':
    content = Content(topic, title, body, url)
    content.print()

crawler = Crawler()

siteData = [
    ['O'Reilly Media', 'http://oreilly.com',
     'https://ssearch.oreilly.com/?q=', 'article.product-result',
     'p.title a', True, 'h1', 'section#product-description'],
    ['Reuters', 'http://reuters.com',
     'http://www.reuters.com/search/news?blob=',
     'div.search-result-content', 'h3.search-result-title a',
     False, 'h1', 'div.StandardArticleBody_body_1gnLA'],
    ['Brookings', 'http://www.brookings.edu',
     'https://www.brookings.edu/search/?s=',
     'div.list-content article', 'h4.title a', True, 'h1',
     'div.post-body']
]
sites = []
for row in siteData:
    sites.append(Website(row[0], row[1], row[2],
                        row[3], row[4], row[5], row[6], row[7]))

topics = ['python', 'data science']
for topic in topics:
    print("GETTING INFO ABOUT: " + topic)
    for targetSite in sites:
        crawler.search(topic, targetSite)

```

Esse script percorre todos os tópicos da lista `topics` em um laço e apresenta uma informação antes de começar a coletar dados sobre um tópico:

```
GETTING INFO ABOUT python
```

Em seguida, o script percorre todos os sites da lista `sites` em um laço e rastreia cada site em particular em busca de um tópico específico. Sempre que informações sobre uma página forem coletadas com sucesso, elas são exibidas no console:

```

New article found for topic: python
URL: http://example.com/examplepage.html
TITLE: Page Title Here
BODY: Body content is here

```

Observe que o laço percorre todos os tópicos e depois percorre todos os sites no laço interno. Por que não fazer o inverso, coletando todos os tópicos de um site e depois todos os tópicos do próximo site? Percorrer todos os tópicos antes é uma maneira de distribuir a carga imposta a qualquer servidor web único de modo mais uniforme. Será particularmente importante se você tiver uma lista com centenas de tópicos e dezenas de sites. Você não fará dezenas de milhares de requisições a um site de uma só vez; fará 10 requisições, esperará alguns minutos, fará outras 10 requisições, esperará mais alguns minutos, e assim por diante.

Embora o número de requisições, em última análise, seja o mesmo em qualquer caso, em geral é melhor distribuir essas requisições no tempo do modo mais sensato possível. Prestar atenção em como seus laços são estruturados é uma maneira simples de fazer isso.

## Rastreando sites por meio de links

No capítulo anterior, vimos algumas maneiras de identificar links internos e externos em páginas web e então usar esses links para rastrear o site. Nesta seção, combinaremos esses mesmos métodos básicos em um web crawler mais flexível, capaz de seguir qualquer link que corresponda a um padrão específico de URL.

Esse tipo de crawler funciona bem para projetos em que queremos coletar todos os dados de um site – e não apenas os dados de um resultado específico de busca ou de uma lista de páginas. Também funciona bem quando as páginas do site estejam desorganizadas ou amplamente dispersas.

Esses tipos de crawlers não exigem um método estruturado para localizar links, como na seção anterior sobre rastreamento por meio de páginas de pesquisa, portanto os atributos que descrevem a página de pesquisa não são necessários no objeto `Website`. Todavia, como o crawler não recebe instruções específicas para as localizações/posições dos links procurados, é necessário ter algumas regras que definam quais tipos de página devem ser selecionados. Forneça um `targetPattern` (uma expressão regular para os URLs visados) e deixe a variável booleana `absoluteUrl` para fazer isso:

```
class Website:
    def __init__(self, name, url, targetPattern, absoluteUrl,
                 titleTag, bodyTag):
        self.name = name
        self.url = url
```

```

self.targetPattern = targetPattern
self.absoluteUrl=absoluteUrl
self.titleTag = titleTag
self.bodyTag = bodyTag

```

```

class Content:
    def __init__(self, url, title, body):
        self.url = url
        self.title = title
        self.body = body

    def print(self):
        print("URL: {}".format(self.url))
        print("TITLE: {}".format(self.title))
        print("BODY:\n{}".format(self.body))

```

A classe content é a mesma usada no exemplo do primeiro crawler.

A classe crawler foi escrita para começar na página inicial do site, localizar os links internos e fazer parse do conteúdo de cada link interno encontrado:

```

import re

class Crawler:
    def __init__(self, site):
        self.site = site
        self.visited = []

    def getPage(self, url):
        try:
            req = requests.get(url)
        except requests.exceptions.RequestException:
            return None
        return BeautifulSoup(req.text, 'html.parser')

    def safeGet(self, pageObj, selector):
        selectedElems = pageObj.select(selector)
        if selectedElems is not None and len(selectedElems) > 0:
            return '\n'.join([elem.get_text() for
                              elem in selectedElems])
        return ''

    def parse(self, url):
        bs = self.getPage(url)
        if bs is not None:
            title = self.safeGet(bs, self.site.titleTag)
            body = self.safeGet(bs, self.site.bodyTag)
            if title != '' and body != '':
                content = Content(url, title, body)
                content.print()

```

```

def crawl(self):
    """
    Obtém páginas da página inicial do site
    """
    bs = self.getPage(self.site.url)
    targetPages = bs.findAll('a',
                             href=re.compile(self.site.targetPattern))
    for targetPage in targetPages:
        targetPage = targetPage.attrs['href']
        if targetPage not in self.visited:
            self.visited.append(targetPage)
            if not self.site.absoluteUrl:
                targetPage = '{}{}'.format(self.site.url, targetPage)
            self.parse(targetPage)

    reuters = Website('Reuters', 'https://www.reuters.com', '^(/article/)', False,
                      'h1', 'div.StandardArticleBody_body_1gnLA')
    crawler = Crawler(reuters)
    crawler.crawl()

```

Eis outra mudança que não foi feita nos exemplos anteriores: o objeto `Website` (nesse caso, a variável `reuters`) é uma propriedade do próprio objeto `Crawler`. Isso funciona bem para armazenar as páginas visitadas (`visited`) no `Crawler`, mas significa que um novo `Crawler` deve ser instanciado para cada site, em vez de reutilizar o mesmo para rastrear uma lista de sites.

O fato de você optar por deixar o `Crawler` independente do site ou fazer do site um atributo do `Crawler` é uma decisão de design que deve ser avaliada no contexto de suas próprias necessidades específicas. Em geral, qualquer uma das abordagens será apropriada.

Outro detalhe a ser observado é que esse `Crawler` obterá as páginas da página inicial, mas não continuará rastreando depois que todas essas páginas tiverem sido registradas. Talvez você queira escrever um `Crawler` que incorpore um dos padrões do *Capítulo 3* e o faça procurar mais alvos em cada página visitada. Você pode até mesmo seguir todos os URLs de cada página (e não somente aqueles que correspondam ao padrão desejado) para procurar URLs contendo o padrão visado.

## Rastreamento vários tipos de página

De modo diferente de rastrear um conjunto predeterminado de páginas, rastrear todos os links internos em um site pode representar um desafio, pois nunca sabemos ao certo o que será obtido. Felizmente, há quatro maneiras básicas de identificar o tipo da página:

## *Pelo URL*

Todas as postagens de blog em um site podem conter um URL (*http://example.com/blog/title-of-post*, por exemplo).

## *Pela presença ou ausência de determinados campos em um site*

Se uma página tiver uma data, mas não tiver nenhum nome de autor, ela poderá ser classificada como um comunicado de imprensa. Se tiver um título, uma imagem principal, preço, mas nenhum conteúdo principal, pode ser uma página de produto.

## *Pela presença de determinadas tags na página que a identifiquem*

Podemos tirar proveito das tags, mesmo que os dados que estejam aí não sejam coletados. Seu crawler poderia procurar um elemento como `<div id="produtos-relacionados">` para identificar a página como uma página de produto, mesmo que o crawler não esteja interessado no conteúdo dos produtos relacionados.

Para manter o controle dos vários tipos de página, é necessário ter vários tipos de objetos para os tipos de páginas em Python. Podemos fazer isso de duas maneiras.

Se todas as páginas forem semelhantes (todas elas têm basicamente o mesmo tipo de conteúdo), um atributo `pageType` pode ser adicionado ao seu objeto de página web:

```
class Website:
    """Classe-base comum para todos os artigos/páginas"""

    def __init__(self, type, name, url, searchUrl, resultListing,
                 resultUrl, absoluteUrl, titleTag, bodyTag):
        self.name = name
        self.url = url
        self.titleTag = titleTag
        self.bodyTag = bodyTag
        self.pageType = pageType
```

Se você estiver armazenando essas páginas em um banco de dados do tipo SQL, esse tipo de padrão sinaliza que todas essas páginas provavelmente seriam armazenadas na mesma tabela e que uma coluna `pageType` extra seria acrescentada.

Se as páginas/conteúdo dos quais os dados estão sendo coletados forem suficientemente diferentes uns dos outros (contêm tipos diferentes de campos), pode-se justificar a criação de novos objetos para cada tipo de

página. É claro que algumas informações serão comuns a todas as páginas web – todas terão um URL e, provavelmente, também terão um nome ou um título. Essa é uma situação ideal para usar subclasses:

```
class Webpage:
    """Classe-base comum para todos os artigos/páginas"""

    def __init__(self, name, url, titleTag):
        self.name = name
        self.url = url
        self.titleTag = titleTag
```

Esse não é um objeto que será usado diretamente pelo seu crawler, mas referenciado pelos tipos de página:

```
class Product(Website):
    """Contém informações para coletar dados de uma página de produto"""
    def __init__(self, name, url, titleTag, productNumber, price):
        Website.__init__(self, name, url, TitleTag)
        self.productNumberTag = productNumberTag
        self.priceTag = priceTag

class Article(Website):
    """Contém informações para coletar dados de uma página de artigo"""
    def __init__(self, name, url, titleTag, bodyTag, dateTag):
        Website.__init__(self, name, url, titleTag)
        self.bodyTag = bodyTag
        self.dateTag = dateTag
```

Essa página `Product` estende a classe-base `Website` e acrescenta os atributos `productNumber` e `price`, que se aplicam somente aos produtos, enquanto a classe `Article` adiciona os atributos `body` e `date`, que não se aplicam aos produtos.

Essas duas classes podem ser usadas para coletar dados, por exemplo, do site de uma loja que contenha postagens de blog ou comunicados de imprensa, além dos produtos.

## Pensando nos modelos de web crawlers

Coletar informações da internet pode ser como beber água de uma mangueira para incêndio. Há muitas informações por aí, e nem sempre está claro quais são as informações necessárias e como são necessárias. O primeiro passo em qualquer projeto de web scraping de grande porte (e até mesmo em alguns projetos pequenos) deve ser responder a essas perguntas. Ao coletar dados similares em vários domínios ou de várias fontes, seu objetivo quase sempre deve ser tentar normalizá-los. Lidar com dados com

campos idênticos e comparáveis é muito mais fácil do que lidar com dados que sejam totalmente dependentes do formato da fonte original.

Em muitos casos, devemos construir scrapers partindo do pressuposto de que outras fontes de dados serão acrescentadas no futuro, e com o objetivo de minimizar o overhead de programação exigido para adicionar essas novas fontes. Mesmo que um site pareça não se enquadrar ao seu modelo à primeira vista, pode haver maneiras mais sutis com as quais ele poderia se adequar. Ser capaz de perceber esses padrões subjacentes pode fazer você economizar tempo, dinheiro e evitar muitas dores de cabeça no longo prazo.

As conexões entre os dados também não devem ser ignoradas. Você está procurando informações que tenham propriedades como “tipo”, “tamanho” ou “tópico”, espalhadas em várias fontes de dados? Como esses atributos serão armazenados, obtidos e conceituados?

Arquitetura de software é um assunto amplo e importante, que pode exigir toda uma carreira para ser dominado. Felizmente, a arquitetura de software para web scraping exige um conjunto de habilidades muito mais restrito e administrável, que pode ser facilmente adquirido. À medida que continuar a coletar dados, é provável que você veja os mesmos padrões se repetirem continuamente. Criar um web scraper bem estruturado não exige muito conhecimento misterioso, porém exige que você dê um passo para trás e pense sobre o seu projeto.

# Scrapy

O capítulo anterior apresentou algumas técnicas e padrões para criar web crawlers grandes, escaláveis e (acima de tudo!) possíveis de manter. Embora seja fácil criá-los manualmente, muitas bibliotecas, frameworks e até mesmo ferramentas GUI farão isso para você, ou pelo menos tentarão facilitar um pouco sua vida.

Este capítulo apresenta um dos melhores frameworks para desenvolvimento de crawlers: o Scrapy. Quando escrevi a primeira edição de *Web Scraping com Python*, o Scrapy ainda não havia sido lançado para Python 3.x, e sua inclusão no texto ficou limitada a uma única seção. Desde então, a biblioteca foi atualizada para que aceitasse Python 3.3+, recursos adicionais foram acrescentados e estou empolgado em expandir aquela seção em um capítulo próprio.

Um dos desafios de escrever web crawlers é que, com frequência, as mesmas tarefas são executadas de modo repetido: encontrar todos os links em uma página, avaliar a diferença entre links internos e externos, acessar novas páginas. É conveniente conhecer esses padrões básicos e ser capaz de escrevê-los do zero, mas a biblioteca Scrapy cuida de muitos desses detalhes para você.

É claro que o Scrapy não lê sua mente. Continua necessário definir templates de página, especificar os locais para começar a coletar dados e definir os padrões de URL das páginas que você estiver procurando. Para esses casos, porém, o Scrapy é um bom framework para manter seu código organizado.

## Instalando o Scrapy

O Scrapy disponibiliza o [download](http://scrapy.org/download/) (<http://scrapy.org/download/>) da ferramenta em seu site, assim como instruções para instalá-lo com gerenciadores de terceiros, como o pip.

Por causa do tamanho relativamente grande e da complexidade, o Scrapy

em geral não é um framework possível de instalar do modo tradicional com:

```
$ pip install Scrapy
```

Note que eu disse “em geral” porque, embora seja teoricamente possível, com frequência deparo com um ou mais problemas complicados de dependência, incompatibilidade de versões e bugs não resolvidos.

Se você estiver determinado a instalar o Scrapy com o pip, usar um ambiente virtual (veja a seção “*Mantendo as bibliotecas organizadas em ambientes virtuais*” para obter mais informações sobre ambientes virtuais) é extremamente recomendável.

Meu método preferido de instalação é usar o [gerenciador de pacotes Anaconda](https://docs.continuum.io/anaconda/) (<https://docs.continuum.io/anaconda/>). O Anaconda é um produto da empresa Continuum, projetado para reduzir o atrito quando se trata de encontrar e instalar pacotes Python conhecidos para ciência de dados. Muitos dos pacotes que ele administra, por exemplo, o NumPy e o NLTK, serão usados nos próximos capítulos também.

Depois que o Anaconda estiver disponível, o Scrapy poderá ser instalado com o comando a seguir:

```
conda install -c conda-forge scrapy
```

Se você deparar com problemas ou precisar de informações atualizadas, consulte o [guia de instalação do Scrapy](https://doc.scrapy.org/en/latest/intro/install.html) (<https://doc.scrapy.org/en/latest/intro/install.html>) para obter mais informações.

### Iniciando um novo spider

Depois que o framework Scrapy estiver instalado, é necessária uma pequena configuração para cada spider. Um *spider*<sup>1</sup> constitui um projeto Scrapy que, como seu aracnídeo homônimo, é projetado para rastrear webs. Neste capítulo, uso o termo “spider” para descrever especificamente um projeto Scrapy, e “crawler” para “qualquer programa genérico que rastreie a web usando ou não o Scrapy”.

Para criar um novo spider no diretório atual, execute o seguinte na linha de comando:

```
$ scrapy startproject wikiSpider
```

Esse comando cria um novo subdiretório chamado *wikiSpider* no diretório em que o projeto foi criado. Nesse diretório, temos a seguinte estrutura de arquivos:

- *scrapy.cfg*
- *wikiSpider*
  - *spiders*
    - *\_\_init.py\_\_*
  - *items.py*
  - *middlewares.py*
  - *pipelines.py*
  - *settings.py*
  - *\_\_init.py\_\_*

Esses arquivos Python são inicializados com código stub, oferecendo um meio rápido para criar um novo projeto spider. Todas as seções deste capítulo trabalham com esse projeto *wikiSpider*.

## Escrevendo um scraper simples

Para criar um crawler, um novo arquivo deve ser acrescentado no diretório *spiders*, em *wikiSpider/wikiSpider/spiders/article.py*. Em seu arquivo *article.py* recém-criado, escreva o seguinte:

```
import scrapy

class ArticleSpider(scrapy.Spider):
    name='article'

    def start_requests(self):
        urls = [
            'http://en.wikipedia.org/wiki/Python_'
            '%28programming_language%29',
            'https://en.wikipedia.org/wiki/Functional_programming',
            'https://en.wikipedia.org/wiki/Monty_Python']
        return [scrapy.Request(url=url, callback=self.parse)
                for url in urls]

    def parse(self, response):
        url = response.url
        title = response.css('h1::text').extract_first()
        print('URL is: {}'.format(url))
        print('Title is: {}'.format(title))
```

O nome dessa classe (*ArticleSpider*) é diferente do nome do diretório (*wikiSpider*), informando que essa classe em particular é responsável pela coleta de dados somente de páginas de artigos, na categoria mais ampla do *wikiSpider*; mais tarde, você poderá usá-lo para procurar outros tipos de

páginas.

Para sites grandes, com vários tipos de conteúdo, podemos ter itens Scrapy separados para cada tipo (postagens de blog, comunicados de imprensa, artigos etc.), cada um com campos distintos, mas todos executando no mesmo projeto Scrapy. O nome de cada spider deve ser único no projeto.

Os outros detalhes importantes para observar nesse spider são as duas funções `start_requests` e `parse`.

`start_requests` é o ponto de entrada para o programa, definido pelo Scrapy e usado para gerar objetos `Request` que o Scrapy utiliza para rastrear o site.

`parse` é uma função de callback definida pelo usuário e passada para o objeto `Request` com `callback=self.parse`. Mais tarde, veremos tarefas mais sofisticadas que podem ser feitas com a função `parse`, mas, por ora, ela exibe o título da página.

Esse spider `article` pode ser executado acessando o diretório `wikiSpider/wikiSpider` e usando o comando:

```
$ scrapy runspider article.py
```

A saída default do Scrapy é bastante extensa. Junto com informações de depuração, linhas como estas são exibidas:

```
2018-01-21 23:28:57 [scrapy.core.engine] DEBUG: Crawled (200)
<GET https://en.wikipedia.org/robots.txt> (referer: None)
2018-01-21 23:28:57 [scrapy.downloadermiddlewares.redirect]
DEBUG: Redirecting (301) to <GET https://en.wikipedia.org/wiki/
Python_%28programming_language%29> from <GET http://en.wikipedia.org/
wiki/Python_%28programming_language%29>
2018-01-21 23:28:57 [scrapy.core.engine] DEBUG: Crawled (200)
<GET https://en.wikipedia.org/wiki/Functional_programming>
(referer: None)
URL is: https://en.wikipedia.org/wiki/Functional_programming
Title is: Functional programming
2018-01-21 23:28:57 [scrapy.core.engine] DEBUG: Crawled (200)
<GET https://en.wikipedia.org/wiki/Monty_Python> (referer: None)
URL is: https://en.wikipedia.org/wiki/Monty_Python
Title is: Monty Python
```

O scraper acessa as três páginas listadas em `urls`, coleta informações e então termina.

## Spidering com regras

O spider da seção anterior não é exatamente um crawler, pois está restrito a coletar dados apenas da lista de URLs especificada. Ele não é capaz de

procurar novas páginas por conta própria. Para transformá-lo em um crawler completo, é necessário usar a classe `CrawlSpider` disponibilizada pelo Scrapy.

### Organização do código no repositório do GitHub

Infelizmente, o framework Scrapy não pode ser facilmente executado a partir de um notebook Jupyter, dificultando apresentar uma progressão linear do código. Com o objetivo de apresentar todos os exemplos de código no texto, o scraper da seção anterior está no arquivo `article.py`, enquanto o exemplo a seguir, que cria um spider Scrapy para percorrer várias páginas, está armazenado em `articles.py` (observe o uso do plural).

Exemplos futuros também serão armazenados em arquivos separados, com novos nomes de arquivo atribuídos em cada seção. Certifique-se de estar usando o nome do arquivo correto quando executar esses exemplos.

A classe a seguir pode ser encontrada em `articles.py` no repositório do Github:

```
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule

class ArticleSpider(CrawlSpider):
    name = 'articles'
    allowed_domains = ['wikipedia.org']
    start_urls = ['https://en.wikipedia.org/wiki/'
                 'Benevolent_dictator_for_life']
    rules = [Rule(LinkExtractor(allow=r'.*'), callback='parse_items',
                 follow=True)]

    def parse_items(self, response):
        url = response.url
        title = response.css('h1::text').extract_first()
        text = response.xpath('//div[@id="mw-content-text"]//text()')
            .extract()
        lastUpdated = response.css('li#footer-info-lastmod::text')
            .extract_first()
        lastUpdated = lastUpdated.replace(
            'This page was last edited on ', '')
        print('URL is: {}'.format(url))
        print('title is: {}'.format(title))
        print('text is: {}'.format(text))
        print('Last updated: {}'.format(lastUpdated))
```

Esse novo `ArticleSpider` estende a classe `CrawlSpider`. Em vez de disponibilizar uma função `start_requests`, ela define uma lista de `start_urls` e `allowed_domains`. Isso informa ao spider por onde ele deve começar a rastrear, e se deve seguir ou ignorar um link com base no domínio.

Uma lista `rules` também é fornecida. Ela contém outras instruções sobre quais links devem ser seguidos ou ignorados (nesse caso, estamos

permitindo todos os URLs com a expressão regular .\*).

Além de extrair o título e o URL de cada página, dois novos itens foram adicionados. O conteúdo textual de cada página é extraído usando um seletor XPath. O XPath é geralmente usado para obter um conteúdo textual, incluindo textos em tags filhas (por exemplo, uma tag <a> em um bloco de texto). Se o seletor CSS for usado para isso, todo o texto das tags filhas será ignorado.

Um parse também é feito na string com a data da última atualização no rodapé da página, a qual é armazenada na variável `lastUpdated`.

Execute esse exemplo indo para o diretório `wikiSpider/wikiSpider` e usando o comando a seguir:

```
$ scrapy runspider articles.py
```

Aviso: esse código executará indefinidamente

Esse spider executará na linha de comando do mesmo modo que o spider anterior, mas não terminará (pelo menos, não por muito, muito tempo), até que você interrompa a execução com Ctrl-C ou feche o terminal. Por favor, seja gentil com a carga do servidor da Wikipédia e não execute esse spider por muito tempo.

Ao ser executado, o spider percorre `wikipedia.org`, seguindo todos os links no domínio `wikipedia.org`, exibindo os títulos das páginas e ignorando todos os links externos (fora do site):

```
2018-01-21 01:30:36 [scrapy.spidermiddlewares.offsite]
DEBUG: Filtered offsite request to 'www.chicagomag.com':
<GET http://www.chicagomag.com/Chicago-Magazine/June-2009/Street-Wise/>
2018-01-21 01:30:36 [scrapy.downloadermiddlewares.robotstxt]
DEBUG: Forbidden by robots.txt: <GET https://en.wikipedia.org/w/
index.php?title=Adrian_Holovaty&action=edit&section=3>
title is: Ruby on Rails
URL is: https://en.wikipedia.org/wiki/Ruby_on_Rails
text is: ['Not to be confused with ', 'Ruby (programming language)',
'.', '\n', '\n', 'Ruby on Rails', ... ]
Last updated: 9 January 2018, at 10:32.
```

É um ótimo crawler até o momento, mas poderia ter alguns limites. Em vez de visitar apenas páginas de artigos na Wikipédia, ele é livre para visitar também as páginas que não contêm artigos, por exemplo:

```
title is: Wikipedia:General disclaimer
```

Vamos observar melhor a linha que usa `Rule` e `LinkExtractor` do Scrapy:

```
rules = [Rule(LinkExtractor(allow=r'.*'), callback='parse_items', follow=True)]
```

Essa linha especifica uma lista de objetos `Rule` do Scrapy, os quais definem as regras para filtrar todos os links encontrados. Se houver várias regras

definidas, estas serão verificadas para cada link, na sequência. A primeira regra atendida será aquela usada para determinar como o link é tratado. Se não obedecer a nenhuma regra, o link será ignorado.

Um `rule` pode receber seis argumentos:

### `link_extractor`

É o único argumento obrigatório: um objeto `LinkExtractor`.

### `callback`

É a função que deve ser usada para parse do conteúdo da página.

### `cb_kwargs`

É um dicionário com os argumentos a serem passados para a função de callback. Esse dicionário é formatado como `{arg_name1: arg_value1, arg_name2: arg_value2}`, e pode ser uma ferramenta conveniente para reutilizar as mesmas funções de parsing em tarefas um pouco diferentes.

### `follow`

Informa se você quer que os links encontrados nessa página sejam incluídos em um rastreamento futuro. Se nenhuma função de callback for especificada, o default é `True` (afinal de contas, se você não está fazendo nada com a página, faz sentido que queira usá-la no mínimo para continuar rastreando o site). Se uma função de callback for especificada, o default será `False`.

`LinkExtractor` é uma classe simples, concebida apenas para reconhecer e devolver links de uma página de conteúdo HTML com base nas regras fornecidas a ela. Essa classe tem uma série de argumentos que podem ser usados para aceitar ou recusar um link de acordo com seletores CSS e XPath, tags (podemos procurar links em outras tags que não apenas as tags de âncora!), domínios e outro critérios.

A classe `LinkExtractor` pode até mesmo ser estendida, e argumentos personalizados podem ser criados. Consulte a [documentação do Scrapy](https://doc.scrapy.org/en/latest/topics/link-extractors.html) (<https://doc.scrapy.org/en/latest/topics/link-extractors.html>) sobre extratores de links para obter outras informações.

Apesar de todos os recursos flexíveis da classe `LinkExtractor`, os argumentos mais comuns que serão provavelmente mais usados são:

### `allow`

Permite todos os links que correspondam à expressão regular fornecida.

## deny

Recusa todos os links que correspondam à expressão regular fornecida.

Ao usar duas classes `Rule` e `LinkExtractor` separadas com uma única função de parsing, podemos criar um spider que rastreie a Wikipédia, identificando todas as páginas de artigos e sinalizando as páginas que não o são (*articlesMoreRules.py*).

```
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule

class ArticleSpider(CrawlSpider):
    name = 'articles'
    allowed_domains = ['wikipedia.org']
    start_urls = ['https://en.wikipedia.org/wiki/'
                  'Benevolent_dictator_for_life']
    rules = [
        Rule(LinkExtractor(allow='^(/wiki/)((?!:).)*$'),
              callback='parse_items', follow=True,
              cb_kwargs={'is_article': True}),
        Rule(LinkExtractor(allow='.*'), callback='parse_items',
              cb_kwargs={'is_article': False})
    ]

    def parse_items(self, response, is_article):
        print(response.url)
        title = response.css('h1::text').extract_first()
        if is_article:
            url = response.url
            text = response.xpath('//div[@id="mw-content-text"]'
                                  '//text()').extract()
            lastUpdated = response.css('li#footer-info-lastmod'
                                       '::text').extract_first()
            lastUpdated = lastUpdated.replace('This page was '
                                             'last edited on ', '')
            print('Title is: {}'.format(title))
            print('title is: {}'.format(title))
            print('text is: {}'.format(text))
        else:
            print('This is not an article: {}'.format(title))
```

Lembre-se de que as regras são aplicadas em cada link na ordem em que aparecem na lista. Todas as páginas de artigos (páginas que começam com `/wiki/` e não contêm dois-pontos) são passadas para a função `parse_items` antes, com o parâmetro default `is_article=True`. Em seguida, todos os outros links para páginas que não são de artigo são passados para a função `parse_items` com o argumento `is_article=False`.

É claro que, se você quiser coletar apenas as páginas de artigos e ignorar todas as demais, essa abordagem seria impraticável. Seria muito mais fácil ignorar as páginas que não correspondam ao padrão de URL de artigo e deixar a segunda regra (e a variável `is_article`) totalmente de lado. No entanto, esse tipo de abordagem pode ser conveniente em casos peculiares, em que informações do URL, ou informações coletadas durante o rastreamento, causem impacto no parsing da página.

## Criando itens

Até agora, vimos várias maneiras de encontrar, fazer parse e rastrear sites com o Scrapy, mas este também oferece ferramentas úteis para manter os itens coletados organizados e armazenados em objetos personalizados, com campos bem definidos.

Para ajudar a organizar todas as informações coletadas, é necessário criar um objeto `Article`. Defina um novo item chamado `Article` no arquivo `items.py`.

Ao abrir o arquivo `items.py`, ele deverá conter o seguinte:

```
# -*- coding: utf-8 -*-

# Define here the models for your scraped items
#
# See documentation in:
# http://doc.scrapy.org/en/latest/topics/items.html

import scrapy

class WikispiderItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    pass
```

Substitua o stub default desse `Item` por uma nova classe `Article` que estenda `scrapy.Item`:

```
import scrapy

class Article(scrapy.Item):
    url = scrapy.Field()
    title = scrapy.Field()
    text = scrapy.Field()
    lastUpdated = scrapy.Field()
```

Estamos definindo três campos que serão coletados de cada página: um título, o URL e a data em que a página foi atualizada pela última vez.

Se estiver coletando dados de vários tipos de página, defina cada tipo diferente como uma classe própria em *items.py*. Se os itens forem grandes, ou se você começar a mover outras funcionalidades de parsing para os itens, talvez queira também colocar cada item em seus próprios arquivos. Contudo, enquanto os itens forem pequenos, prefiro mantê-los em um único arquivo.

No arquivo *articleSpider.py*, observe as modificações feitas na classe *articleSpider* a fim de criar o novo item *Article*:

```
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule
from wikiSpider.items import Article

class ArticleSpider(CrawlSpider):
    name = 'articleItems'
    allowed_domains = ['wikipedia.org']
    start_urls = ['https://en.wikipedia.org/wiki/Benevolent'
                 '_dictator_for_life']
    rules = [
        Rule(LinkExtractor(allow='(/wiki/)((?!:).)*$'),
            callback='parse_items', follow=True),
    ]

    def parse_items(self, response):
        article = Article()
        article['url'] = response.url
        article['title'] = response.css('h1::text').extract_first()
        article['text'] = response.xpath('//div[@id='
            '"mw-content-text"]//text()').extract()
        lastUpdated = response.css('li#footer-info-lastmod::text')
            .extract_first()
        article['lastUpdated'] = lastUpdated.replace('This page was '
            'last edited on ', '')
        return article
```

Quando esse arquivo é executado com:

```
$ scrapy runspider articleItems.py
```

a saída mostrará os dados usuais de depuração do Scrapy, junto com cada item de artigo na forma de um dicionário Python:

```
2018-01-21 22:52:38 [scrapy.spidermiddlewares.offsite] DEBUG:
Filtered offsite request to 'wikimediafoundation.org':
<GET https://wikimediafoundation.org/wiki/Terms_of_Use>
2018-01-21 22:52:38 [scrapy.core.engine] DEBUG: Crawled (200)
<GET https://en.wikipedia.org/wiki/Benevolent_dictator_for_life
# mw-head> (referer: https://en.wikipedia.org/wiki/Benevolent_dictator_for_life)
2018-01-21 22:52:38 [scrapy.core.scrapers] DEBUG: Scraped from
<200 https://en.wikipedia.org/wiki/Benevolent_dictator_for_life>
```

```
{'lastUpdated': ' 13 December 2017, at 09:26.',  
'text': ['For the political term, see ',  
        'Benevolent dictatorship',  
        '.',  
        ...
```

Usar o recurso `Items` do Scrapy não serve apenas para ter uma boa organização no código ou dispô-lo de maneira mais legível. Os itens oferecem muitas ferramentas para apresentação e processamento dos dados; elas serão discutidas nas próximas seções.

## Apresentando itens

O Scrapy usa os objetos `Item` para determinar quais informações das páginas visitadas devem ser salvas. Essas informações podem ser salvas pelo Scrapy em diversos formatos, por exemplo, arquivos CSV, JSON ou XML, usando os comandos a seguir:

```
$ scrapy runspider articleItems.py -o articles.csv -t csv  
$ scrapy runspider articleItems.py -o articles.json -t json  
$ scrapy runspider articleItems.py -o articles.xml -t xml
```

Cada um desses comandos executa o scraper `articleItems` e escreve a saída no formato e no arquivo especificados. O arquivo será criado caso ainda não exista.

Talvez você tenha notado que, no spider de artigos criado nos exemplos anteriores, a variável de texto é uma lista de strings, e não uma única string. Cada string nessa lista representa o texto em um único elemento HTML, enquanto o conteúdo em `<div id="mw-content-text">`, do qual estamos coletando os dados de texto, é composto de vários elementos filhos.

O Scrapy administra bem esses valores mais complexos. No formato CSV, por exemplo, ele converte listas de strings e escapa todas as vírgulas, de modo que uma lista de texto é exibida em uma única célula do CSV.

Em XML, cada elemento dessa lista é preservado em tags filhas:

```
<items>  
<item>  
  <url>https://en.wikipedia.org/wiki/Benevolent_dictator_for_life</url>  
  <title>Benevolent dictator for life</title>  
  <text>  
    <value>For the political term, see </value>  
    <value>Benevolent dictatorship</value>  
    ...  
  </text>  
</lastUpdated> 13 December 2017, at 09:26.</lastUpdated>
```

```
</item>
```

```
....
```

No formato JSON, as listas são preservadas como listas.

É claro que você pode usar os objetos `Item` por conta própria e escrevê-los em um arquivo ou banco de dados do modo que quiser, simplesmente acrescentando o código apropriado na função de parsing no crawler.

## Pipeline de itens

Apesar de executar em uma só thread, o Scrapy é capaz de fazer e tratar várias requisições de modo assíncrono. Isso faz com que ele seja mais rápido que os scrapers que escrevemos até agora neste livro, embora sempre acreditei firmemente que mais rápido nem sempre é melhor quando se trata de web scraping.

O servidor web do site do qual você está tentando coletar dados deve tratar cada uma dessas requisições, e é importante ser um bom cidadão e avaliar se esse tipo de carga imposta ao servidor é apropriada (ou se, inclusive, é uma atitude inteligente em seu interesse próprio, pois muitos sites têm a capacidade e a disposição para bloquear o que poderiam ver como uma atividade maliciosa de coleta de dados). Para mais informações sobre o código de ética no web scraping, assim como sobre a importância de fazer um throttling apropriado nos scrapers, consulte o *Capítulo 18*.

Apesar do que foi dito, usar o pipeline de itens do Scrapy pode melhorar mais ainda a velocidade de seu web scraper, pois todo o processamento de dados é feito enquanto se espera que as requisições sejam devolvidas, em vez de esperar que os dados sejam processados antes de fazer outra requisição. Esse tipo de otimização às vezes é inclusive necessário se o processamento de dados exigir bastante tempo ou cálculos com intenso uso do processador tiverem de ser feitos.

Para criar um pipeline de itens, reveja o arquivo *settings.py*, criado no início do capítulo. Você verá as seguintes linhas comentadas:

```
# Configure item pipelines
# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html
# ITEM_PIPELINES = {
# 'wikiSpider.pipelines.WikiSpiderPipeline': 300,
# }
```

Remova o caractere de comentário das três últimas linhas, substituindo-as pelo seguinte:

```
ITEM_PIPELINES = {
    'wikiSpider.pipelines.WikispiderPipeline': 300,
}
```

Com isso, temos uma classe Python, `wikiSpider.pipelines.WikispiderPipeline`, que será usada para processar os dados, assim como um inteiro que representa a ordem de execução do pipeline se houver várias classes de processamento. Embora seja possível usar qualquer inteiro nesse caso, os números de 0 a 1000 são tipicamente utilizados, e serão executados na ordem crescente.

Agora temos de acrescentar a classe de pipeline e reescrever o spider original de modo que ele colete dados e o pipeline faça o trabalho pesado de processá-los. Pode ser tentador fazer o método `parse_items` do spider original devolver a resposta e deixar o pipeline criar o objeto `Article`:

```
def parse_items(self, response):
    return response
```

No entanto, o framework Scrapy não permite isso, e um objeto `Item` (por exemplo, um `Article`, que estende `Item`) deve ser devolvido. Assim, o objetivo de `parse_items` agora é extrair os dados brutos, fazendo o mínimo possível de processamento, a fim de que esses dados sejam passados para o pipeline:

```
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule
from wikiSpider.items import Article

class ArticleSpider(CrawlSpider):
    name = 'articlePipelines'
    allowed_domains = ['wikipedia.org']
    start_urls = ['https://en.wikipedia.org/wiki/Benevolent_dictator_for_life']
    rules = [
        Rule(LinkExtractor(allow='(/wiki/)((?!:).)*$'),
            callback='parse_items', follow=True),
    ]

    def parse_items(self, response):
        article = Article()
        article['url'] = response.url
        article['title'] = response.css('h1::text').extract_first()
        article['text'] = response.xpath('//div[@id='
            '"mw-content-text"]//text()').extract()
        article['lastUpdated'] = response.css('li#'
            'footer-info-lastmod::text').extract_first()
        return article
```

Esse arquivo foi salvo no repositório do GitHub como *articlePipelines.py*.

É claro que agora é necessário associar o arquivo *settings.py* e o spider atualizado com o acréscimo do pipeline. Quando o projeto Scrapy foi iniciado, havia um arquivo em *wikiSpider/wikiSpider/settings.py*:

```
# -*- coding: utf-8 -*-

# Define your item pipelines here
#
# Don't forget to add your pipeline to the ITEM_PIPELINES setting
# See: http://doc.scrapy.org/en/latest/topics/item-pipeline.html
class WikispiderPipeline(object):
    def process_item(self, item, spider):
        return item
```

Essa classe stub deve ser substituída pelo novo código de pipeline. Nas seções anteriores, estávamos coletando dois campos em um formato bruto, e eles poderiam se beneficiar com um processamento adicional: `lastUpdated` (um objeto string mal formatado representando uma data) e `text` (um array confuso de fragmentos de string).

Use o código a seguir para substituir o código stub em *wikiSpider/wikiSpider/settings.py*:

```
from datetime import datetime
from wikiSpider.items import Article
from string import whitespace

class WikispiderPipeline(object):
    def process_item(self, article, spider):
        dateStr = article['lastUpdated']
        article['lastUpdated'] = article['lastUpdated']
            .replace('This page was last edited on', '')
        article['lastUpdated'] = article['lastUpdated'].strip()
        article['lastUpdated'] = datetime.strptime(
            article['lastUpdated'], '%d %B %Y, at %H:%M.')
        article['text'] = [line for line in article['text']
            if line not in whitespace]
        article['text'] = ''.join(article['text'])
        return article
```

A classe `WikispiderPipeline` tem um método `process_item` que recebe um objeto `Article`, faz parse da string `lastUpdated` gerando um objeto Python `datetime`, além de limpar e juntar o texto em uma única string a partir de uma lista de strings.

`process_item` é um método obrigatório para toda classe de pipeline. O Scrapy usa esse método para passar `Items` coletados pelo spider de modo assíncrono. O objeto `Article`, cujo parse foi feito e que é devolvido nesse caso, será registrado em log ou exibido pelo Scrapy se, por exemplo,

estivermos gerando itens em formato JSON ou CSV, como fizemos na seção anterior.

Agora temos duas opções quanto se trata de decidir o local em que o processamento de dados será feito: o método `parse_items` no spider, ou o método `process_items` no pipeline.

Vários pipelines com diferentes tarefas podem ser declarados no arquivo *settings.py*. No entanto, o Scrapy passa todos os itens, qualquer que seja o tipo, para cada pipeline, na sequência. Se um parsing for específico de um item, pode ser melhor que seja feito no spider, antes que os dados cheguem até o pipeline. No entanto, se esse parsing demorar muito, você poderia considerar movê-lo para o pipeline (poderá ser processado aí de modo assíncrono) e acrescentar uma verificação para o tipo do item:

```
def process_item(self, item, spider):
    if isinstance(item, Article):
        # Processamento específico de um Article deve ser inserido aqui
```

O tipo de processamento e o local em que deve ser feito são considerações importantes quando se trata de escrever projetos Scrapy, sobretudo se forem projetos grandes.

## Fazendo log com o Scrapy

As informações de depuração geradas pelo Scrapy podem ser úteis, mas, como você provavelmente deve ter notado, em geral elas são muito extensas. É possível ajustar facilmente o nível de logging acrescentando uma linha no arquivo *settings.py* em seu projeto Scrapy:

```
LOG_LEVEL = 'ERROR'
```

O Scrapy usa uma hierarquia padrão de níveis de logging:

- CRITICAL
- ERROR
- WARNING
- DEBUG
- INFO

Se o logging for definido com `ERROR`, somente logs `CRITICAL` e `ERROR` serão exibidos. Se for definido com `INFO`, todos os logs serão exibidos, e assim por diante.

Além de controlar o logging por meio do arquivo *settings.py*, podemos

controlar para onde os logs serão enviados a partir da linha de comando. Para gerar os logs em um arquivo separado, em vez de enviá-los para o terminal, defina um arquivo de log usando a linha de comando:

```
$ scrapy crawl articles -s LOG_FILE=wiki.log
```

Esse comando cria um novo arquivo de log em seu diretório atual caso ainda não exista e envia todos os logs para esse arquivo, deixando o terminal limpo para exibir apenas as instruções print de Python adicionadas manualmente.

## Outros recursos

O Scrapy é uma ferramenta eficaz, que cuida de vários problemas associados ao rastreamento da web. Ele coleta automaticamente todos os URLs e os compara em relação a regras predefinidas, garante que todos os URLs sejam únicos, normaliza URLs relativos quando necessário e oferece recursão para alcançar níveis mais profundos nas páginas.

Embora este capítulo mal tenha tocado a superfície dos recursos do Scrapy, incentivo você a consultar a [documentação do Scrapy](https://doc.scrapy.org/en/latest/news.html) (<https://doc.scrapy.org/en/latest/news.html>), bem como o livro [Learning Scrapy](http://shop.oreilly.com/product/9781784399788.do) de Dimitrios Kouzis-Loukas (O'Reilly, <http://shop.oreilly.com/product/9781784399788.do>), que têm um conteúdo completo sobre o framework.

O Scrapy é uma biblioteca muito grande e abrangente, com vários recursos. Suas funcionalidades estão bem integradas, mas há muitas áreas de sobreposição que permitem aos usuários desenvolver o próprio estilo com facilidade. Se houver algo que você queira fazer com o Scrapy, mas que não tenha sido mencionado neste capítulo, é provável que haja uma maneira (ou várias) de fazê-lo!

---

<sup>1</sup> N.T.: Literalmente, spider significa aranha.

# Armazenando dados

Embora exibir dados no terminal seja muito divertido, não é muito útil quando se trata de agregá-los e analisá-los. Para que a maioria dos web scrapers seja minimamente útil, é necessário poder salvar as informações que eles coletam.

Este capítulo descreve três métodos principais de gerenciamento de dados que são suficientes para praticamente qualquer aplicação imaginável. Você precisa alimentar o backend de um site ou criar a sua própria API? Provavelmente vai querer que seus scrapers escrevam em um banco de dados. Precisa de um modo rápido e fácil de coletar documentos da internet e colocá-los em seu disco rígido? É provável que queira criar um stream de arquivos para isso. Precisa de alertas ocasionais ou de dados agregados uma vez ao dia? Envie um email a si mesmo!

Muito além do web scraping, a capacidade de armazenar e de interagir com grandes volumes de dados é muito importante para praticamente qualquer aplicação de programação moderna. Com efeito, as informações deste capítulo são necessárias para implementar vários dos exemplos das futuras seções do livro. Recomendo que você pelo menos leia rapidamente este capítulo caso não tenha familiaridade com a armazenagem automática de dados.

## Arquivos de mídia

Há duas maneiras principais de armazenar arquivos de mídia: por referência e fazendo download do arquivo propriamente dito. Podemos armazenar um arquivo por referência guardando o URL em que está o arquivo. Essa opção tem as seguintes vantagens:

- Os scrapers executam mais rápido e exigem muito menos banda quando não precisam fazer download dos arquivos.
- Você economiza espaço em suas próprias máquinas ao armazenar somente os URLs.

- É mais fácil escrever um código que armazene somente URLs e não tenha de lidar com downloads adicionais de arquivos.
- Podemos reduzir a carga no servidor host ao evitar downloads de arquivos grandes.

Eis as desvantagens:

- Incluir esses URLs em seu próprio site ou aplicação é conhecido como *hotlinking*, e fazer isso é uma maneira rápida de se colocar em uma situação problemática na internet.
- Você não deve usar os ciclos de servidor de outra pessoa para hospedar mídias das próprias aplicações.
- O arquivo hospedado em qualquer URL em particular está sujeito a mudança. Isso pode ter efeitos constrangedores se, por exemplo, você estiver incluindo um hotlink para uma imagem em um blog público. Se estiver armazenando os URLs com o intuito de obter o arquivo mais tarde para pesquisas no futuro, em algum momento ele poderá deixar de existir ou mudar para algo totalmente irrelevante.
- Navegadores web de verdade não requisitam simplesmente o HTML de uma página e seguem adiante. Eles também fazem download de todos os recursos necessários à página. Fazer download de arquivos pode ajudar a fazer seu scraper parecer um ser humano navegando no site, e isso pode ser uma vantagem.

Se estiver em dúvida quanto a armazenar um arquivo ou um URL em um arquivo, pergunte a si mesmo se é mais provável que você visualize ou leia esse arquivo mais de uma ou duas vezes, ou se esse banco de dados de arquivos ficará parado, acumulando poeira eletrônica na maior parte de sua vida. Se a resposta for a última opção, provavelmente será melhor apenas armazenar o URL. Se for a primeira, continue lendo!

A biblioteca `urllib`, usada para obter o conteúdo de páginas web, também contém funções para obter o conteúdo de arquivos. O programa a seguir usa `urllib.request.urlretrieve` para fazer download de imagens de um URL remoto:

```
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com')
bs = BeautifulSoup(html, 'html.parser')
imageLocation = bs.find('a', {'id': 'logo'}).find('img')['src']
```

```
urlretrieve (imageLocation, 'logo.jpg')
```

Esse código faz o download do logo de <http://pythonscraping.com> e o armazena como *logo.jpg* no mesmo diretório em que o script está executando.

O código funciona bem se for necessário fazer o download de apenas um arquivo e você souber como ele se chama e qual a sua extensão. A maioria dos scrapers, porém, não faz download de um só arquivo e dá o trabalho por encerrado. Os downloads a seguir são de arquivos internos, associados ao atributo `src` de qualquer tag, da página inicial de <http://pythonscraping.com>:

```
import os
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup

downloadDirectory = 'downloaded'
baseUrl = 'http://pythonscraping.com'

def getAbsoluteURL(baseUrl, source):
    if source.startswith('http://www.'):
        url = 'http://{0}'.format(source[11:])
    elif source.startswith('http://'):
        url = source
    elif source.startswith('www.'):
        url = source[4:]
        url = 'http://{0}'.format(source)
    else:
        url = '{0}/{1}'.format(baseUrl, source)
    if baseUrl not in url:
        return None
    return url

def getDownloadPath(baseUrl, absoluteUrl, downloadDirectory):
    path = absoluteUrl.replace('www.', '')
    path = path.replace(baseUrl, '')
    path = downloadDirectory+path
    directory = os.path.dirname(path)

    if not os.path.exists(directory):
        os.makedirs(directory)

    return path

html = urlopen('http://www.pythonscraping.com')
bs = BeautifulSoup(html, 'html.parser')
downloadList = bs.findAll(src=True)
```

```
for download in downloadList:
    fileUrl = getAbsoluteURL(baseUrl, download['src'])
    if fileUrl is not None:
        print(fileUrl)

urlretrieve(fileUrl, getDownloadPath(baseUrl, fileUrl, downloadDirectory))
```

### Execute com cautela

Você já ouviu falar de todas essas advertências sobre fazer download de arquivos desconhecidos da internet? O script anterior faz download de tudo que encontrar e grava no disco rígido de seu computador. Isso inclui scripts bash aleatórios, arquivos *.exe* e outros malwares em potencial.

Você acha que está seguro porque, na verdade, nunca executou nada que tenha sido enviado para a sua pasta de downloads? Se executássemos esse programa como administrador em particular, estaríamos pedindo por problemas. O que aconteceria se encontrássemos um arquivo em um site que enviasse a si mesmo para *../../../../usr/bin/python*? Na próxima vez que executássemos um script Python na linha de comando, poderíamos estar instalando um malware no computador!

Esse programa foi escrito somente com finalidades ilustrativas; ele jamais deverá ser empregado aleatoriamente, sem uma verificação mais rígida dos nomes de arquivos, e deve ser executado apenas em uma conta com permissões limitadas. Como sempre, fazer backup de seus arquivos, não armazenar informações confidenciais em seu disco rígido e usar um pouco de bom senso já ajuda bastante.

Esse script usa uma função lambda (apresentada no *Capítulo 2*) para selecionar todas as tags da página inicial que contenham o atributo `src`, e em seguida limpa e normaliza os URLs para obter um path absoluto para cada download (além de garantir que os links externos sejam descartados). Então, o download de cada arquivo é feito em seu próprio path na pasta local *downloaded* em seu computador.

Observe que o módulo `os` de Python é usado rapidamente para obter o diretório-alvo de cada download e criar os diretórios que ainda não existem no path, se for necessário. O módulo `os` atua como uma interface entre Python e o sistema operacional, permitindo manipular paths de arquivo, criar diretórios, obter informações sobre processos em execução e variáveis de ambiente, além de fazer várias outras tarefas úteis.

## Armazenando dados no formato CSV

O CSV (Comma-Separated Values, ou Valores Separados por Vírgula) é um dos formatos mais conhecidos de arquivos para armazenar dados de planilha. É aceito pelo Microsoft Excel e por muitas outras aplicações por causa de sua simplicidade. A seguir, temos um exemplo de um arquivo CSV perfeitamente válido:

```
fruit, cost
```

```
apple,1.00
banana,0.30
pear,1.25
```

Como em Python, espaços em branco são importantes nesse caso: cada linha é separada por um caractere de quebra de linha, enquanto as colunas na linha são separadas por vírgulas (daí o nome “Separados por Vírgula”). Outros formatos de arquivos CSV (às vezes chamados de arquivos com *valores separados por caractere* (character-separated value) usam tabulações ou outros caracteres para separar as linhas, mas esses formatos são menos comuns e menos aceitos amplamente.

Se você estiver tentando fazer download de arquivos CSV direto da web e armazená-los localmente, sem qualquer parse ou modificação, esta seção não será necessária. Faça download deles como faria com qualquer outro arquivo e salve-os no formato de arquivo CSV usando os métodos descritos na seção anterior.

Modificar um arquivo CSV ou até mesmo criar um totalmente do zero é muito fácil com a biblioteca Python *csv*:

```
import csv

csvFile = open('test.csv', 'w+')
try:
    writer = csv.writer(csvFile)
    writer.writerow(('number', 'number plus 2', 'number times 2'))
    for i in range(10):
        writer.writerow( (i, i+2, i*2))
finally:
    csvFile.close()
```

Um lembrete como precaução: a criação de arquivos em Python é razoavelmente à prova de balas. Se *test.csv* ainda não existir, Python criará o arquivo (mas não o diretório) automaticamente. Se já existir, o arquivo *test.csv* será sobrescrito com os novos dados.

Depois de executar o código, um arquivo CSV será visto:

```
number,number plus 2,number times 2
0,2,0
1,3,2
2,4,4
...
```

Uma tarefa comum em web scraping é obter uma tabela HTML e gravá-la em um arquivo CSV. O artigo [Comparison of Text Editors](#) (Comparação entre editores de texto,

[https://en.wikipedia.org/wiki/Comparison\\_of\\_text\\_editors](https://en.wikipedia.org/wiki/Comparison_of_text_editors)) da Wikipédia apresenta uma tabela HTML um tanto quanto complexa, completa, com código de cores, links, ordenação e outros lixos HTML que devem ser descartados antes que seja escrita como CSV. Usando bastante o BeautifulSoup e a função `get_text()`, podemos fazer isso com menos de 20 linhas:

```
import csv
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://en.wikipedia.org/wiki/'
               'Comparison_of_text_editors')
bs = BeautifulSoup(html, 'html.parser')
# A tabela principal de comparação é atualmente a primeira tabela da página
table = bs.findAll('table',{'class':'wikitable'})[0]
rows = table.findAll('tr')

csvFile = open('editors.csv', 'wt+')
writer = csv.writer(csvFile)
try:
    for row in rows:
        csvRow = []
        for cell in row.findAll(['td', 'th']):
            csvRow.append(cell.get_text())
        writer.writerow(csvRow)
finally:
    csvFile.close()
```

Há um modo mais fácil de buscar uma única tabela

Esse script é ótimo para ser integrado em scrapers caso você encontre muitas tabelas HTML que devam ser convertidas em arquivos CSV, ou muitas tabelas HTML que devam ser coletadas e inseridas em um único arquivo CSV. No entanto, se for necessário fazer isso apenas uma vez, há uma ferramenta mais apropriada: copiar e colar. Selecionar e copiar todo o conteúdo de uma tabela HTML e colá-lo no Excel ou no Google Docs dará a você o arquivo CSV desejado sem executar um script!

O resultado será um arquivo CSV bem formatado, salvo localmente como `../files/editors.csv`.

## MySQL

MySQL (pronunciado oficialmente como “mai esse-que-ele,” embora muitos digam, “mai siquel”) é o sistema de gerenciamento de banco de dados relacional de código aberto mais conhecido hoje em dia. De modo um pouco incomum para um projeto de código aberto com grandes concorrentes, a popularidade do MySQL, historicamente, vem sendo

disputada palmo a palmo com dois outros grandes sistemas de banco de dados de código fechado: o SQL Server da Microsoft e o DBMS da Oracle. Sua popularidade não é desprovida de razão. Para a maioria das aplicações, é difícil cometer um erro caso a escolha seja o MySQL. É um DBMS escalável, robusto e completo, usado por grandes sites: o YouTube<sup>1</sup>, o Twitter<sup>2</sup> e o Facebook<sup>3</sup>, entre vários outros.

Por causa de sua ubiquidade, do preço (“gratuito” é um ótimo preço) e da usabilidade imediata, o MySQL é um banco de dados incrível para projetos de web scraping e será usado no resto deste livro.

### **Banco de dados “relacional”?**

*Dados relacionais* são dados que apresentam relações. Fico feliz por ter esclarecido essa questão! Brincadeira! Quando falam de dados relacionais, os cientistas da computação estão se referindo a dados que não existem de forma isolada – são dados cujas propriedades os relacionam a outras porções de dados. Por exemplo, “O Usuário A estuda na Instituição B”, em que o Usuário A pode ser encontrado na tabela de Usuários do banco de dados e a Instituição B pode ser encontrada na tabela de Instituições.

Mais adiante neste capítulo, veremos como modelar diferentes tipos de relações e armazenar dados no MySQL (ou em qualquer outro banco de dados relacional) de modo eficaz.

### **Instalando o MySQL**

Se o MySQL for uma novidade para você, instalar um banco de dados pode parecer um pouco intimidador (se já tiver experiência nisso, sintase à vontade para ignorar esta seção). Na verdade, é tão simples quanto instalar praticamente qualquer outro tipo de software. Em seu núcleo, o MySQL funciona com um conjunto de arquivos de dados, armazenados em um servidor ou no computador local, contendo todas as informações guardadas no banco de dados. A camada de software do MySQL acima desses dados oferece uma maneira conveniente de interagir com os dados por meio de uma interface de linha de comando. Por exemplo, o comando a seguir percorre os arquivos de dados e devolve uma lista de todos os usuários no banco de dados cujo primeiro nome seja “Ryan”:

```
SELECT * FROM users WHERE firstname = "Ryan"
```

Se você estiver em uma distribuição Linux baseada em Debian (ou em um sistema com `apt-get`), instalar o MySQL é simples e basta executar o seguinte:

```
$ sudo apt-get install mysql-server
```

Preste atenção no processo de instalação, aprove os requisitos de memória e insira uma nova senha para o novo usuário root quando for solicitada.

No macOS e no Windows, a situação é um pouco mais complicada. Caso ainda não tenha uma conta Oracle, crie uma antes de fazer download do pacote.

Se estiver no macOS, é necessário obter antes [o pacote de instalação](http://dev.mysql.com/downloads/mysql/) (<http://dev.mysql.com/downloads/mysql/>).

Selecione o pacote *.dmg* e faça login com sua conta Oracle – ou crie uma – para fazer download do arquivo. Depois de abri-lo, você receberá orientações de um assistente de instalação razoavelmente simples (*Figure 6.1*).

Os passos default da instalação devem ser suficientes; neste livro, partirei do pressuposto de que você tem uma instalação MySQL default.



*Figura 6.1 – Instalador do MySQL no macOS.*

Se o fato de fazer download e executar um instalador parece um pouco enfadonho e você está usando um Mac, é sempre possível instalar o gerenciador de pacotes *Homebrew* (<https://brew.sh/>). Com o Homebrew disponível, o MySQL também pode ser instalado executando o seguinte:

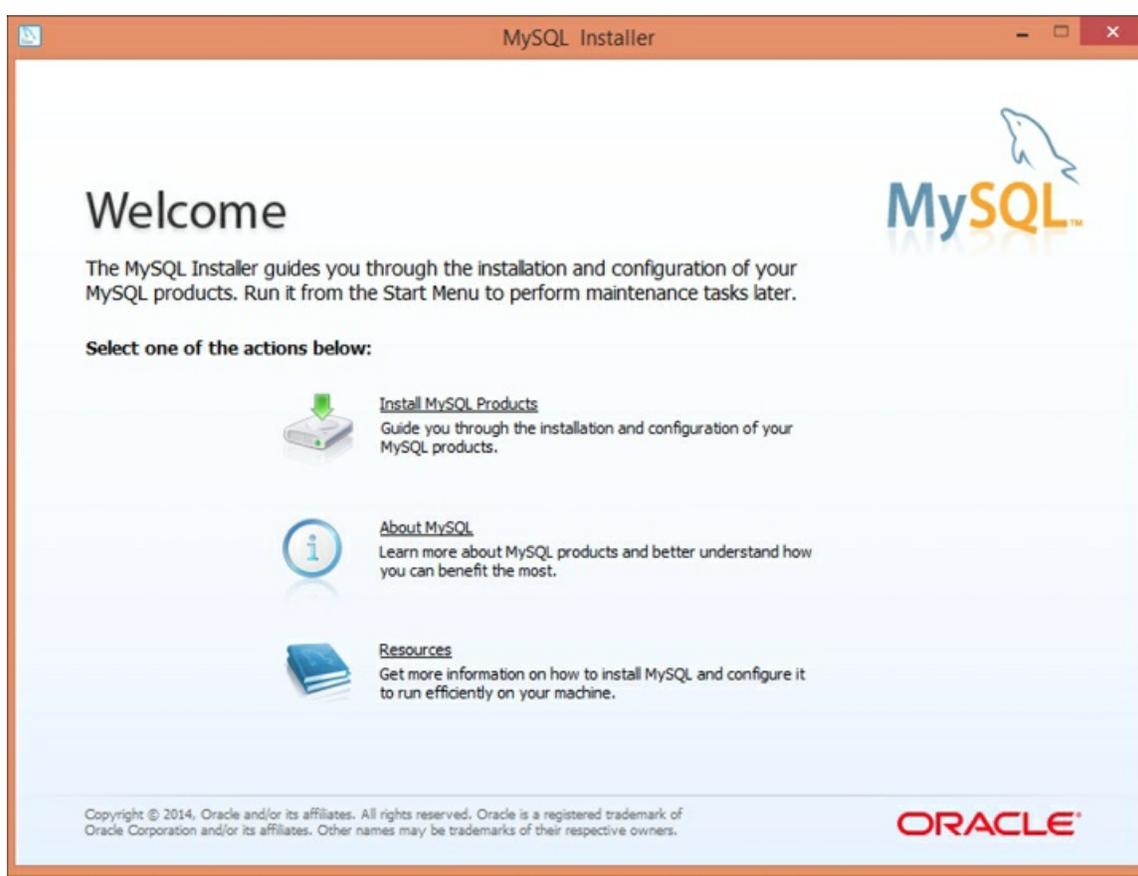
```
$ brew install mysql
```

O Homebrew é um ótimo projeto de código aberto, com uma boa integração com pacotes Python. A maioria dos módulos Python de terceiros usados neste livro pode ser facilmente instalada com o

Homebrew. Caso ainda não o tenha, recomendo que dê uma olhada nele! Depois que o MySQL estiver instalado no macOS, o servidor MySQL pode ser iniciado assim:

```
$ cd /usr/local/mysql
$ sudo ./bin/mysqld_safe
```

No Windows, instalar e executar o MySQL é um pouco mais complicado, mas a boa notícia é que [um instalador conveniente](http://dev.mysql.com/downloads/windows/installer/) (<http://dev.mysql.com/downloads/windows/installer/>) simplifica o processo. Feito o download, ele orientará você pelos passos necessários (veja a Figura 6.2).



*Figura 6.2 – Instalador do MySQL no Windows.*

Você deve ser capaz de instalar o MySQL usando as opções default, com uma exceção: na página Setup Type (Tipo de configuração), recomendo selecionar Server Only (Somente servidor) a fim de evitar a instalação de vários softwares adicionais e bibliotecas da Microsoft. A partir daí, será possível usar as configurações default da instalação e seguir os prompts para iniciar seu servidor MySQL.

## Alguns comandos básicos

Depois que seu servidor MySQL estiver executando, haverá muitas opções para interagir com o banco de dados. Várias ferramentas de software atuam como intermediárias para que você não tenha de lidar com comandos MySQL (ou no mínimo lidar com eles com menos frequência). Ferramentas como phpMyAdmin e MySQL Workbench podem facilitar a visualização, a ordenação e a inserção rápida de dados. Apesar disso, é importante saber trabalhar com a linha de comando.

Exceto pelos nomes de variáveis, o MySQL não diferencia letras maiúsculas de minúsculas; por exemplo, `SELECT` é igual a `select`. No entanto, por convenção, todas as palavras reservadas do MySQL são escritas com letras maiúsculas quando escrevemos uma instrução MySQL. Por outro lado, a maioria dos desenvolvedores prefere nomear suas tabelas e bancos de dados com letras minúsculas, embora esse padrão muitas vezes seja ignorado.

Ao fazer login pela primeira vez no MySQL, não haverá nenhum banco de dados para acrescentar dados, mas podemos criar um:

```
> CREATE DATABASE scraping;
```

Como toda instância do MySQL pode ter vários bancos de dados, antes de começar a interagir com um deles, é necessário informar ao MySQL qual banco de dados queremos usar:

```
> USE scraping;
```

A partir desse ponto (ou pelo menos até a conexão com o MySQL ser encerrada ou passarmos para outro banco de dados), todos os comandos inseridos serão executados no novo banco de dados `scraping`.

Tudo isso parece bem simples. Deve ser igualmente fácil criar uma tabela no banco de dados, certo? Vamos tentar criar uma tabela para armazenar uma coleção de páginas web das quais coletamos dados:

```
> CREATE TABLE pages;
```

Esse comando resulta em um erro:

```
ERROR 1113 (42000): A table must have at least 1 column
```

De modo diferente de um banco de dados, que pode existir sem qualquer tabela, uma tabela no MySQL não pode existir sem que haja colunas. Para definir colunas no MySQL, essas devem ser especificadas como uma lista delimitada por vírgulas, entre parênteses, após a instrução `CREATE TABLE <nomedatabela>`:

```
> CREATE TABLE pages (id BIGINT(7) NOT NULL AUTO_INCREMENT,  
title VARCHAR(200), content VARCHAR(10000),  
created TIMESTAMP DEFAULT CURRENT_TIMESTAMP, PRIMARY KEY(id));
```

A definição de cada coluna tem três partes:

- o nome (`id`, `title`, `created` etc.);
- o tipo da variável (`BIGINT(7)`), `VARCHAR`, `TIMESTAMP`);
- opcionalmente, qualquer atributo adicional (`NOT NULL AUTO_INCREMENT`).

No final da lista de colunas, uma *chave* (key) da tabela deve ser definida. O MySQL usa chaves para organizar o conteúdo da tabela para consultas rápidas. Mais adiante neste capítulo, descreverei como tirar proveito dessas chaves para ter bancos de dados mais rápidos; por enquanto, porém, usar a coluna `id` de uma tabela como chave em geral é a melhor opção.

Depois de executar o comando, podemos ver como é a estrutura da tabela a qualquer momento usando `DESCRIBE`:

```
> DESCRIBE pages;
```

```
+-----+-----+-----+-----+-----+-----+  
| Field | Type          | Null | Key | Default          | Extra          |  
+-----+-----+-----+-----+-----+-----+  
| id    | bigint(7)     | NO   | PRI | NULL             | auto_increment |  
| title | varchar(200)  | YES  |     | NULL             |                |  
| content | varchar(10000) | YES  |     | NULL             |                |  
| created | timestamp     | NO   |     | CURRENT_TIMESTAMP |                |  
+-----+-----+-----+-----+-----+-----+  
4 rows in set (0.01 sec)
```

É claro que essa tabela ainda está vazia. Podemos inserir dados de teste na tabela `pages` com a linha a seguir:

```
> INSERT INTO pages (title, content) VALUES ("Test page title",  
"This is some test page content. It can be up to 10,000 characters long.");
```

Perceba que, apesar de a tabela ter quatro colunas (`id`, `title`, `content`, `created`), é necessário definir apenas duas delas (`title` e `content`) para inserir uma linha. Isso ocorre porque a coluna `id` é incrementada de modo automático (o MySQL soma 1 automaticamente a cada vez que uma nova linha é inserida) e, em geral, é capaz de cuidar de si mesma. Além disso, a coluna `timestamp` foi definida para conter o horário atual como default.

É claro que *podemos* sobrescrever esses defaults:

```
> INSERT INTO pages (id, title, content, created) VALUES (3,  
"Test page title",
```

```
"This is some test page content. It can be up to 10,000 characters  
long.", "2014-09-21 10:25:32");
```

Desde que o inteiro fornecido para a coluna `id` não exista ainda no banco de dados, essa sobrescrita funcionará perfeitamente. No entanto, fazer isso em geral não é uma boa prática; é melhor deixar que o MySQL cuide das colunas `id` e `timestamp`, a menos que haja um motivo convincente para proceder de outra forma.

Agora que temos alguns dados na tabela, podemos usar vários métodos para selecioná-los. Eis alguns exemplos de instruções `SELECT`:

```
> SELECT * FROM pages WHERE id = 2;
```

Essa instrução diz o seguinte ao MySQL: “Selecione tudo de `pages` cujo `id` seja 2”. O asterisco (\*) atua como um caractere-curinga, devolvendo todas as linhas em que a cláusula (`where id equals 2`) é verdadeira. A instrução devolve a segunda linha da tabela, ou um resultado vazio se não houver nenhuma linha cujo `id` seja 2. Por exemplo, a consulta a seguir, que não diferencia letras maiúsculas de minúsculas, devolve todas as linhas cujo campo `title` contenha “test” (o símbolo % atua como um caractere-curinga em strings MySQL):

```
> SELECT * FROM pages WHERE title LIKE "%test%";
```

O que aconteceria se tivéssemos uma tabela com várias colunas e quiséssemos somente que uma porção específica dos dados fosse devolvida? Em vez de selecionar tudo, podemos executar um comando como:

```
> SELECT id, title FROM pages WHERE content LIKE "%page content%";
```

Essa instrução devolve `id` e `title` quando o conteúdo contiver a expressão “page content”.

Instruções `DELETE` têm praticamente a mesma sintaxe das instruções `SELECT`:

```
> DELETE FROM pages WHERE id = 1;
```

Por esse motivo, é uma boa ideia, sobretudo quando trabalhamos com bancos de dados importantes, impossíveis de serem facilmente restaurados, escrever qualquer instrução `DELETE` como uma instrução `SELECT` antes (nesse caso, `SELECT * FROM pages WHERE id = 1`), testar para garantir que apenas as linhas que queremos apagar serão devolvidas, e então substituir `SELECT *` por `DELETE`. Muitos programadores contam histórias terríveis sobre erros na cláusula de uma instrução `DELETE` – ou, pior ainda, sobre como se esqueceram totalmente de defini-la por estarem com pressa – arruinando os dados do cliente. Não deixe que isso aconteça com você!

Precauções semelhantes devem ser tomadas com instruções UPDATE:

```
> UPDATE pages SET title="A new title",  
content="Some new content" WHERE id=2;
```

Neste livro, trabalharemos apenas com instruções MySQL simples, fazendo seleções, inserções e atualizações básicas. Se você estiver interessado em conhecer outros comandos e técnicas relacionadas a essa ferramenta de banco de dados eficaz, recomendo o livro [MySQL Cookbook](http://shop.oreilly.com/product/0636920032274.do) de Paul DuBois (O'Reilly, <http://shop.oreilly.com/product/0636920032274.do>).

## Integração com Python

Infelizmente, não há suporte incluído para MySQL em Python. No entanto, várias bibliotecas de código aberto, tanto para Python 2.x quanto para Python 3.x, permitem interagir com um banco de dados MySQL. Uma das mais conhecidas é o [PyMySQL](https://pypi.org/project/PyMySQL/) (<https://pypi.org/project/PyMySQL/>).

Atualmente (quando este livro foi escrito), a versão do PyMySQL é a 0.6.7, que pode ser instalada com o pip:

```
$ pip install PyMySQL
```

Também podemos fazer download e instalá-lo a partir do código-fonte; isso pode ser conveniente se você quiser usar uma versão específica da biblioteca:

```
$ curl -L https://pypi.python.org/packages/source/P/PyMySQL/PyMySQL-0.6.7.tar.gz\  
| tar xz  
$ cd PyMySQL-PyMySQL-f953785/  
$ python setup.py install
```

Após a instalação, você terá acesso ao pacote PyMySQL de modo automático. Enquanto seu servidor MySQL local estiver executando, o script a seguir poderá ser executado com sucesso (lembre-se de adicionar a senha de root para o seu banco de dados):

```
import pymysql  
conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',  
                        user='root', passwd=None, db='mysql')  
cur = conn.cursor()  
cur.execute('USE scraping')  
cur.execute('SELECT * FROM pages WHERE id=1')  
print(cur.fetchone())  
cur.close()  
conn.close()
```

Dois novos tipos de objetos estão em ação nesse exemplo: o objeto de conexão (`conn`) e o objeto cursor (`cur`).

O modelo conexão/cursor é comum em programação de banco de dados, embora alguns usuários possam achar complicado diferenciá-los no começo. A conexão é responsável por, bem, conectar-se com o banco de dados, é claro, mas também por enviar informações do banco de dados, tratar rollbacks (quando for necessário abortar uma consulta ou um conjunto de consultas e o banco de dados tiver de voltar ao seu estado anterior) e criar novos objetos cursores.

Uma conexão pode ter vários cursores. Um cursor controla certas informações de *estado*, por exemplo, o banco de dados que ele está usando. Se houver vários bancos de dados e for necessário escrever informações em todos, vários cursores poderão ser usados. Um cursor também contém o resultado da última consulta executada. Ao chamar funções no cursor, por exemplo, `cur.fetchone()`, essas informações podem ser acessadas.

É importante que tanto o cursor quanto a conexão sejam fechados depois que forem usados. Não fazer isso pode resultar em *vazamentos de conexão* (connection leaks): um acúmulo de conexões abertas que não estão mais sendo utilizadas, mas que o software não é capaz de fechar por achar que ainda estão em uso. É o tipo de ocorrência que faz os bancos de dados caírem o tempo todo (já escrevi e corriji muitos bugs de vazamento de conexões), portanto lembre-se de fechar suas conexões!

A tarefa mais comum, que provavelmente você vai querer fazer logo no início, é armazenar os resultados de sua coleta de dados em um banco de dados. Vamos ver como isso pode ser feito usando um exemplo anterior: o scraper da Wikipédia.

Lidar com texto Unicode pode ser difícil ao usar web scraping. Por padrão, o MySQL não trata Unicode. Felizmente, podemos ativar esse recurso (basta lembrar que, ao fazer isso, o tamanho de seu banco de dados aumentará). Como você deverá deparar com uma variedade de caracteres diferentes na Wikipédia, agora é uma boa hora de dizer ao banco de dados que espere alguns dados Unicode:

```
ALTER DATABASE scraping CHARACTER SET = utf8mb4 COLLATE = utf8mb4_unicode_ci;
ALTER TABLE pages CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
ALTER TABLE pages CHANGE title title VARCHAR(200) CHARACTER SET utf8mb4 COLLATE
utf8mb4_unicode_ci;
ALTER TABLE pages CHANGE content content VARCHAR(10000) CHARACTER SET utf8mb4 CO
LLATE utf8mb4_unicode_ci;
```

Essas quatro linhas alteram o conjunto de caracteres default do banco de dados, da tabela e das duas colunas – de `utf8mb4` (tecnicamente é Unicode,

mas tem má fama pelo suporte ruim para a maioria dos caracteres Unicode) para `utf8mb4_unicode_ci`.

Você saberá se foi bem-sucedido se tentar inserir alguns trechos ou caracteres em mandarim nos campos `title` ou `content` do banco de dados e não houver erros.

Agora que o banco de dados está preparado para aceitar uma grande variedade de tudo que a Wikipédia pode lançar aí, podemos executar o seguinte:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import datetime
import random
import pymysql
import re

conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
                       user='root', passwd=None, db='mysql', charset='utf8')
cur = conn.cursor()
cur.execute("USE scraping")

random.seed(datetime.datetime.now())

def store(title, content):
    cur.execute('INSERT INTO pages (title, content) VALUES '
               '("%s", "%s")', (title, content))
    cur.connection.commit()

def getLinks(articleUrl):
    html = urlopen('http://en.wikipedia.org'+articleUrl)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    content = bs.find('div', {'id':'mw-content-text'}).find('p')
        .get_text()
    store(title, content)
    return bs.find('div', {'id':'bodyContent'}).findAll('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))

links = getLinks('/wiki/Kevin_Bacon')
try:
    while len(links) > 0:
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        print(newArticle)
        links = getLinks(newArticle)
finally:
    cur.close()
    conn.close()
```

Há alguns detalhes a serem observados nesse caso: em primeiro lugar,

"charset='utf8'" foi adicionado na string de conexão com o banco de dados. Isso diz à conexão que ela deve enviar todas as informações ao banco de dados como UTF-8 (e, é claro, o banco de dados já deve estar configurado para tratá-las).

Em segundo lugar, observe o acréscimo de uma função `store`. Ela aceita duas variáveis do tipo string, `title` e `content`, e as adiciona em uma instrução `INSERT` executada pelo cursor e cujo `commit` é feito pela conexão do cursor. Esse é um excelente exemplo da separação entre o cursor e a conexão; embora o cursor tenha armazenado informações sobre o banco de dados e o seu próprio contexto, ele deve atuar por meio da conexão para enviar informações de volta ao banco de dados e inserir informações.

Por fim, vemos que uma instrução `finally` foi adicionada no laço principal do programa, no final do código. Isso garante que, não importa como o programa seja interrompido ou as exceções sejam lançadas durante a execução (considerando que a web é confusa, sempre suponha que exceções serão lançadas), o cursor e a conexão serão ambos fechados de imediato antes que o programa termine. É uma boa ideia incluir uma instrução `try...finally` como essa sempre que estiver coletando dados da web e houver uma conexão aberta com o banco de dados.

Apesar de o PyMySQL não ser um pacote enorme, há um número razoável de funções úteis que não foram descritas neste livro. Você pode conferir a [documentação](https://pymysql.readthedocs.io/en/latest/) (<https://pymysql.readthedocs.io/en/latest/>) no site do PyMySQL.

## **Técnicas de banco de dados e boas práticas**

Algumas pessoas passam a carreira inteira estudando, fazendo ajustes e inventando bancos de dados. Não sou uma dessas pessoas e este não é o tipo de livro que faça isso. Como em muitas áreas da ciência da computação, porém, há alguns truques que podem ser aprendidos de modo rápido, ao menos para deixar seus bancos de dados aceitáveis – e suficientemente rápidos – para a maioria das aplicações.

Em primeiro lugar, com raras exceções, sempre adicione colunas `id` em suas tabelas. Todas as tabelas no MySQL devem ter pelo menos uma chave primária (a coluna de chave com base na qual o MySQL faz a ordenação), de modo que ele saiba ordená-las, mas muitas vezes pode ser difícil escolher essas chaves de forma inteligente.

A discussão sobre usar uma coluna de `id` criada artificialmente para essa chave ou usar um atributo único, por exemplo, um `username`, tem sido acirrada entre cientistas de dados e engenheiros de software há anos, embora me sinto inclinada para o lado da criação de colunas com `id`. Isso vale *sobretudo* quando lidamos com web scraping e armazenamos dados de outras pessoas. Não temos ideia do que realmente é único ou não, e já me surpreendi antes.

A coluna `id` deve ser incrementada de modo automático e usada como chave primária em todas as suas tabelas.

Em segundo lugar, use uma indexação inteligente. Um dicionário (o livro, não o objeto Python) é uma lista de palavras indexadas em ordem alfabética. Isso permite fazer consultas com rapidez sempre que quisermos uma palavra, desde que saibamos como ela é grafada. Também poderíamos pensar em um dicionário organizado em ordem alfabética de acordo com a definição da palavra. Não seria nem de perto tão útil, a menos que estivéssemos em algum tipo de jogo inusitado de perguntas e respostas, em que uma definição fosse apresentada e você tivesse de dizer a palavra. Contudo, no mundo das consultas em bancos de dados, esses tipos de situação podem ocorrer. Por exemplo, podemos ter um campo no banco de dados que seria frequentemente consultado assim:

```
>SELECT * FROM dictionary WHERE definition="A small furry animal that says meow";
+-----+-----+-----+
| id   | word | definition |
+-----+-----+-----+
| 200 | cat  | A small furry animal that says meow |
+-----+-----+-----+
1 row in set (0.00 sec)
```

Um índice poderia muito bem ser adicionado nessa tabela (além do índice já esperado com base no `id`) na coluna `definition` para que as consultas nessa coluna sejam mais rápidas. Lembre-se, porém, de que adicionar indexação exige mais espaço para o novo índice bem como tempo adicional de processamento para a inserção de novas linhas. Em especial quando lidamos com grandes volumes de dados, devemos considerar com cuidado o custo-benefício entre seus índices e o volume a ser indexado. Para deixar esse índice das “definições” mais leve, podemos dizer ao MySQL que indexe apenas os primeiros caracteres do valor da coluna. O comando a seguir cria um índice com os 16 primeiros caracteres do campo

definition:

```
CREATE INDEX definition ON dictionary (id, definition(16));
```

Esse índice deixará suas consultas mais rápidas quando procurarmos palavras com base em sua definição completa (sobretudo se os 16 primeiros caracteres dos valores da definição tiverem a tendência de serem bem diferentes uns dos outros), sem exigir muito no tocante a espaço extra ou tempo prévio de processamento.

Quando se trata de tempo de consulta *versus* tamanho do banco de dados (uma das tarefas fundamentais relacionadas a equilíbrio na engenharia de banco de dados), um dos erros mais comuns, particularmente com web scraping envolvendo grandes volumes de textos em línguas naturais, é armazenar muitos dados repetidos. Por exemplo, suponha que queremos contabilizar a frequência de determinadas expressões que apareçam em vários sites. Essas expressões podem ser encontradas em uma dada lista ou ser automaticamente geradas por um algoritmo de análise de textos. Poderíamos ficar tentados a armazenar os dados de modo semelhante a este:

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
url	varchar(200)	YES		NULL	
phrase	varchar(200)	YES		NULL	

Adicionamos uma linha no banco de dados sempre que uma expressão em um site for encontrada e registramos o URL no qual ela foi encontrada. Entretanto, ao separar os dados em três tabelas separadas, podemos reduzir bastante o conjunto de dados:

```
>DESCRIBE phrases
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
phrase	varchar(200)	YES		NULL	

```
>DESCRIBE urls
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
url	varchar(200)	YES		NULL	

```
>DESCRIBE foundInstances
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
urlId	int(11)	YES		NULL	
phraseId	int(11)	YES		NULL	
occurrences	int(11)	YES		NULL	

Embora as definições das tabelas sejam maiores, podemos notar que a maioria das colunas são apenas campos `id` inteiros. Elas ocupam muito menos espaço. Além disso, o texto completo de cada URL e de cada expressão é armazenado exatamente uma única vez.

A menos que você instale um pacote de terceiros ou mantenha um registro meticuloso, talvez seja impossível dizer quando um dado foi adicionado, atualizado ou removido do banco de dados. Conforme o espaço disponível para os dados, a frequência das alterações e a importância de determinar quando essas alterações ocorreram, podemos considerar manter vários timestamps: `created`, `updated` e `deleted`.

## “Six Degrees” no MySQL

No *Capítulo 3*, apresentamos o problema do Six Degrees of Wikipedia

(Seis graus de separação na Wikipédia), em que o objetivo era encontrar a conexão entre dois artigos quaisquer da Wikipédia por uma série de links (isto é, encontrar uma maneira de ir de um artigo da Wikipédia para outro somente clicando em links de uma página para a próxima). Para resolver esse problema, é necessário não só construir bots capazes de rastrear o site (já fizemos isso), mas armazenar as informações de modo que tenham uma arquitetura robusta, facilitando uma análise de dados no futuro.

Colunas de `id` incrementados de forma automática, timestamps e várias tabelas: tudo isso entra em cena nesse caso. Para determinar a melhor maneira de armazenar essas informações, é necessário pensar de forma abstrata. Um link nada mais é do que algo que conecta a Página A à Página B. Ele poderia igualmente conectar a Página B à Página A, mas seria um outro link. Podemos identificar unicamente um link dizendo o seguinte: “Há um link na página A que faz a conexão com a página B, isto é, `INSERT INTO links (fromPageId, toPageId) VALUES (A, B);` (em que A e B são os IDs únicos das duas páginas).”

Um sistema de duas tabelas, projetado para armazenar páginas e links, junto com datas de criação e IDs únicos, pode ser implementado assim:

```
CREATE TABLE `wikipedia`.`pages` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `url` VARCHAR(255) NOT NULL,  
  `created` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,  
  PRIMARY KEY (`id`));
```

```
CREATE TABLE `wikipedia`.`links` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `fromPageId` INT NULL,  
  `toPageId` INT NULL,  
  `created` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,  
  PRIMARY KEY (`id`));
```

Observe que, de modo diferente dos crawlers anteriores que exibiam o título da página, não estamos nem mesmo armazenando essa informação na tabela `pages`. Por quê? Bem, registrar o título da página exige que a página seja acessada para obtê-lo. Se quisermos construir um web crawler eficaz para preencher essas tabelas, devemos armazenar a página, assim como os links para ela, mesmo que não tenhamos necessariamente visitado ainda a página.

Embora não seja verdade para todos os sites, o aspecto interessante sobre os links e os títulos das páginas na Wikipédia é que um pode ser

transformado no outro com uma manipulação simples. Por exemplo, [http://en.wikipedia.org/wiki/Monty\\_Python](http://en.wikipedia.org/wiki/Monty_Python) indica que o título da página é “Monty Python”.

O código a seguir armazenará todas as páginas da Wikipédia que tenham um “número de Bacon” (o número de links entre ela e a páginas de Kevin Bacon, inclusive) menor ou igual a 6.

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import pymysql
from random import shuffle

conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
                       user='root', passwd=None, db='mysql', charset='utf8')
cur = conn.cursor()
cur.execute('USE wikipedia')

def insertPageIfNotExists(url):
    cur.execute('SELECT * FROM pages WHERE url = %s', (url))
    if cur.rowcount == 0:
        cur.execute('INSERT INTO pages (url) VALUES (%s)', (url))
        conn.commit()
        return cur.lastrowid
    else:
        return cur.fetchone()[0]

def loadPages():
    cur.execute('SELECT * FROM pages')
    pages = [row[1] for row in cur.fetchall()]
    return pages

def insertLink(fromPageId, toPageId):
    cur.execute('SELECT * FROM links WHERE fromPageId = %s '
               'AND toPageId = %s', (int(fromPageId), int(toPageId)))
    if cur.rowcount == 0:
        cur.execute('INSERT INTO links (fromPageId, toPageId) VALUES (%s, %s)',
                   (int(fromPageId), int(toPageId)))
        conn.commit()

def getLinks(pageUrl, recursionLevel, pages):
    if recursionLevel > 4:
        return

    pageId = insertPageIfNotExists(pageUrl)
    html = urlopen('http://en.wikipedia.org{}'.format(pageUrl))
    bs = BeautifulSoup(html, 'html.parser')
    links = bs.findAll('a', href=re.compile('^(/wiki/)((?!:).)*$'))
    links = [link.attrs['href'] for link in links]
```

```

for link in links:
    insertLink(pageId, insertPageIfNotExists(link))
    if link not in pages:
        # Encontramos uma nova página, acrescente-a e procure
        # links aí
        pages.append(link)
        getLinks(link, recursionLevel+1, pages)

getLinks('/wiki/Kevin_Bacon', 0, loadPages())
cur.close()
conn.close()

```

Nesse código, três funções usam o PyMySQL como interface para o banco de dados:

### `insertPageIfNotExists`

Como diz o nome, essa função insere um novo registro de página caso ela ainda não exista. Essa informação, junto com a lista dinâmica de todas as páginas coletadas armazenada em `pages`, garante que os registros das páginas não estejam duplicados. A função também serve para consultar números de `pageId` na criação de novos links.

### `insertLink`

Cria um novo registro de link no banco de dados. O link não será criado caso já exista. Mesmo que *haja* dois ou mais links idênticos na página, em nosso caso, eles serão iguais, representam o mesmo relacionamento e serão contabilizados como um único registro. Essa função também ajuda a manter a integridade do banco de dados caso o programa seja executado várias vezes, ainda que seja nas mesmas páginas.

### `loadPages`

Carrega todas as páginas que estão no banco de dados no momento em uma lista para determinar se uma nova página deve ser visitada. As páginas também são coletadas em tempo de execução, portanto, se esse crawler for executado apenas uma vez, começando com um banco de dados vazio, teoricamente, `loadPages` não seria necessária. Na prática, porém, pode haver problemas. A rede pode cair, ou talvez você queira coletar links em vários períodos de tempo, e é importante que o crawler seja capaz de carregar a si mesmo novamente, sem perder o controle.

Você deve estar ciente de um detalhe sutil que poderia ser problemático no uso de `loadPages` e da lista `pages` gerada por ela para determinar se uma página deve ou não ser visitada: assim que cada página é carregada, todos

os links dessa página são armazenados como páginas, mesmo que não tenham sido visitadas ainda – somente seus links foram vistos. Se o crawler for interrompido e reiniciado, todas essas páginas “vistas mas não visitadas” jamais serão visitadas, e os links que aí estão não serão registrados. Podemos corrigir esse problema acrescentando uma variável booleana `visited` em cada registro de página e definindo-a com `True` apenas se essa página tiver sido carregada e seus próprios links de saída tiverem sido registrados.

Em nosso caso, porém, a solução é adequada como está. Se for possível garantir que haja tempos de execução bem demorados (ou apenas uma execução) e que um conjunto completo de links não seja necessário (somente um conjunto de dados grande para fazer um experimento), não será preciso acrescentar a variável `visited`.

Para ver a continuação desse problema e a solução final para sair de [Kevin Bacon](https://en.wikipedia.org/wiki/Kevin_Bacon) ([https://en.wikipedia.org/wiki/Kevin\\_Bacon](https://en.wikipedia.org/wiki/Kevin_Bacon)) e chegar em [Eric Idle](https://en.wikipedia.org/wiki/Eric_Idle) ([https://en.wikipedia.org/wiki/Eric\\_Idle](https://en.wikipedia.org/wiki/Eric_Idle)), consulte a seção “*Six Degrees of Wikipedia: conclusão*”, que descreve como solucionar problemas de grafos direcionados.

## Email

Assim como as páginas web são enviadas por meio de HTTP, um email é enviado via SMTP (Simple Mail Transfer Protocol, ou Protocolo Simples de Transferência de Correio). E, assim como usamos um cliente de servidor web para cuidar do envio de páginas web via HTTP, os servidores usam diversos clientes de email, como Sendmail, Postfix ou Mailman para enviar e receber emails.

Embora enviar email com Python seja relativamente simples, isso exige que você tenha acesso a um servidor executando SMTP. Configurar um cliente SMTP em seu servidor ou computador local é complicado e está fora do escopo deste livro, mas vários recursos excelentes podem ajudar nessa tarefa, em particular se você estiver executando Linux ou macOS.

Os códigos de exemplo a seguir partem do pressuposto de que você está executando um cliente SMTP localmente. (Para modificar este código e usar um cliente SMTP remoto, mude `localhost` para o endereço de seu servidor remoto.)

Enviar um email com Python exige apenas nove linhas de código:

```

import smtplib
from email.mime.text import MIMEText

msg = MIMEText('The body of the email is here')

msg['Subject'] = 'An Email Alert'
msg['From'] = 'ryan@pythonscraping.com'
msg['To'] = 'webmaster@pythonscraping.com'

s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()

```

Python tem dois pacotes importantes para envio de email: *smtplib* e *email*.

O módulo de email de Python contém funções convenientes de formatação para criar pacotes de email a serem enviados. O objeto `MIMEText` usado no exemplo cria um email vazio, formatado para ser transferido com o protocolo de baixo nível MIME (Multipurpose Internet Mail Extensions, Extensões de Email para Internet com Múltiplos Propósitos) por meio do qual as conexões SMTP de nível mais alto são feitas. O objeto `MIMEText`, `msg`, contém endereços de origem e de destino do email, assim como um corpo e um cabeçalho, usados por Python para criar um email formatado de modo apropriado.

O pacote `smtplib` contém informações para cuidar da conexão com o servidor. Assim como uma conexão com um servidor MySQL, essa conexão deve ser encerrada sempre que for criada, a fim de evitar criar conexões em excesso.

A função básica de email pode ser ampliada e se tornar mais útil se for incluída em uma função:

```

import smtplib
from email.mime.text import MIMEText
from bs4 import BeautifulSoup
from urllib.request import urlopen
import time

def sendMail(subject, body):
    msg = MIMEText(body)
    msg['Subject'] = subject
    msg['From'] = 'christmas_alerts@pythonscraping.com'
    msg['To'] = 'ryan@pythonscraping.com'

    s = smtplib.SMTP('localhost')
    s.send_message(msg)
    s.quit()

bs = BeautifulSoup(urlopen('https://isitchristmas.com/'), 'html.parser')

```

```
while(bs.find('a', {'id':'answer'}).attrs['title'] == 'NO'):
    print('It is not Christmas yet.')
    time.sleep(3600)
    bs = BeautifulSoup(urlopen('https://isitchristmas.com/'), 'html.parser')

sendMail('It\'s Christmas!',
        'According to http://itischristmas.com, it is Christmas!')
```

Esse script em particular verifica o site <https://isitchristmas.com> (cuja característica principal é apresentar um YES ou NO gigante, dependendo do dia do ano) uma vez por hora. Se algo diferente de NO for visto, você receberá um email avisando que é Natal.

Embora esse programa em particular talvez não pareça mais útil que um calendário pendurado na parede, ele pode ser um pouco ajustado para fazer diversas tarefas extremamente úteis. O programa pode enviar emails de alerta em resposta a interrupções de serviço em sites, a falhas em testes ou até mesmo em resposta à disponibilidade de um produto que você estava esperando na Amazon, cujo estoque havia se esgotado – seu calendário na parede não é capaz de fazer nada disso.

---

<sup>1</sup> Joab Jackson, “*YouTube Scales MySQL with Go Code*” (YouTube escala MySQL com código Go, <http://bit.ly/1LWVmc8>), PCWorld, 15 de dezembro de 2012.

<sup>2</sup> Jeremy Cole e Davi Arnaut, “*MySQL at Twitter*” (MySQL no Twitter, <http://bit.ly/1KHDKns>), The Twitter Engineering Blog, 9 de abril de 2012.

<sup>3</sup> “*MySQL and Database Engineering: Mark Callaghan*” (MySQL e engenharia de banco de dados: Mark Callaghan, <http://on.fb.me/1RFMqvww>), Facebook Engineering, 4 de março de 2012.

## PARTE II

# Coleta de dados avançada

Adquirimos um conhecimento básico sobre web scraping; agora vem a parte divertida. Até o momento, nossos web scrapers são relativamente burros. São incapazes de obter informações a menos que lhes sejam apresentadas de imediato pelo servidor em um formato apropriado. Aceitam todas as informações como lhes são apresentadas e as armazenam sem fazer qualquer análise. Formulários, interação com sites e até mesmo JavaScript podem ser um obstáculo para eles. Em suma, não são apropriados para obter informações a menos que estas realmente se disponham a ser coletadas.

Esta parte do livro ajudará você a analisar dados brutos a fim de conhecer a história por trás deles – a história que, com frequência, os sites ocultam embaixo de camadas de JavaScript, formulários de login e medidas antiscraping. Veremos como usar web scrapers para testar seus sites, automatizar processos e acessar a internet em larga escala. No final desta seção, você terá as ferramentas necessárias para coletar e manipular praticamente qualquer tipo de dado, em qualquer formato, em toda a internet.

# Lendo documentos

É tentador pensar na internet essencialmente como uma coleção de sites baseados em texto, intercalados com conteúdo multimídia supermoderno para web 2.0, o qual, em sua maior parte, poderia ser ignorado quando se trata de web scraping. No entanto, isso seria ignorar o aspecto mais fundamental da internet: ser um veículo de transmissão de arquivos, não importa o conteúdo.

Embora, de uma forma ou de outra, a internet esteja presente desde o final dos anos 1960, o HTML fez sua estreia apenas por volta de 1992. Até então, a internet era constituída principalmente de emails e transmissão de arquivos; o conceito de páginas web conforme o conhecemos hoje em dia ainda não existia. Em outras palavras, a internet não é uma coleção de arquivos HTML. É uma coleção de vários tipos de documentos, com os arquivos HTML sendo usados com frequência como uma moldura para exibí-los. Sem a capacidade de ler diversos tipos de documentos, incluindo texto, PDF, imagens, vídeos, emails e outros, estaríamos deixando de lado uma parte enorme dos dados disponíveis.

Este capítulo aborda como lidar com documentos, seja fazendo download deles em uma pasta local ou lendo e extraíndo seus dados. Veremos também como lidar com vários tipos de codificação de texto, que possibilitam ler até mesmo páginas HTML em idiomas estrangeiros.

## Codificação de documentos

A codificação de um documento diz às aplicações – sejam elas o sistema operacional de seu computador ou o seu próprio código Python – como ele deve ser lido. Em geral, essa codificação pode ser deduzida a partir da extensão do arquivo, embora essa extensão não seja exigida pela codificação. Por exemplo, eu poderia salvar *minhaImagem.jpg* como *minhaImagem.txt* sem que houvesse problemas – pelo menos até que meu editor de texto tentasse abri-lo. Felizmente, essa situação é rara e a

extensão de arquivo de um documento em geral é tudo que é preciso saber para lê-lo de forma correta.

No nível básico, todos os documentos são codificados com 0s e 1s. Acima disso, os algoritmos de codificação definem detalhes como “quantos bits por caractere” ou “quantos bits representam a cor de cada pixel” (no caso de arquivos de imagens). No próximo nível, pode haver uma camada de compactação ou algum algoritmo para redução de espaço, como no caso dos arquivos PNG.

Embora lidar com arquivos que não sejam HTML pareça assustador à primeira vista, fique tranquilo porque, com a biblioteca correta, Python estará equipado de forma adequada para lidar com qualquer formato de informações que você queira lhe passar. A única diferença entre um arquivo-texto, um arquivo de vídeo e um arquivo de imagem é o modo como seus 0s e 1s são interpretados. Este capítulo descreve vários tipos de arquivos comuns: texto, CSV, PDFs e documentos Word.

Note que, fundamentalmente, todos esses arquivos armazenam texto. Para obter informações sobre como trabalhar com imagens, recomendo ler este capítulo para se habituar com diferentes tipos de arquivos e aprender a armazená-los, e então consulte o *Capítulo 13*, que contém mais informações sobre processamento de imagens!

## Texto

De certo modo, não é comum ter arquivos online armazenados como texto simples, mas é frequente que sites antigos ou mais simples tenham repositórios grandes de arquivos-texto. Por exemplo, o IETF (Internet Engineering Task Force, ou Força-tarefa de Engenharia da Internet) armazena todos os seus documentos publicados como HTML, PDF e arquivos-texto (veja <https://www.ietf.org/rfc/rfc1149.txt> como exemplo). A maioria dos navegadores não terá problemas para exibir esses arquivos-texto, e você poderá coletar seus dados.

Para documentos mais básicos em formato texto, por exemplo, o arquivo usado como exercício em <http://www.pythonscraping.com/pages/warandpeace/chapter1.txt>, podemos usar o método a seguir:

```
from urllib.request import urlopen
textPage = urlopen('http://www.pythonscraping.com/')
```

```
'pages/warandpeace/chapter1.txt')  
print(textPage.read())
```

Em geral, quando obtemos uma página com `urlopen`, nós a transformamos em um objeto `BeautifulSoup` para fazer parse do HTML. Nesse caso, podemos ler a página de forma direta. Transformá-la em um objeto `BeautifulSoup`, embora seja perfeitamente possível, seria contraproducente – não há nenhum HTML para fazer parse, portanto a biblioteca seria inútil. Depois que o arquivo-texto é lido na forma de string, basta analisá-lo como faríamos com qualquer string lida em Python. Está claro que a desvantagem, nesse caso, está no fato de não podermos usar tags HTML como pistas do contexto para saber onde estão os textos que realmente queremos e excluir os textos indesejados. Pode ser um desafio se estivermos tentando extrair determinadas informações dos arquivos-texto.

## **Codificação de texto e a internet global**

Você se lembra do que eu disse antes sobre uma extensão de arquivo ser tudo que é necessário para ler um arquivo de forma correta? Bem, por mais estranho que seja, essa regra não se aplica ao documento mais básico de todos: o arquivo `.txt`.

Nove de cada dez vezes, ler um texto usando os métodos descritos antes funcionará bem. No entanto, lidar com textos da internet pode ser um negócio complicado. A seguir, veremos o básico sobre a codificação do inglês e de idiomas estrangeiros, de ASCII a Unicode ou ISO, e como lidar com eles.

### **História da codificação de textos**

O ASCII foi inicialmente desenvolvido nos anos 1960, quando os bits eram caros e não havia razão para codificar nada além do alfabeto latino e alguns caracteres de pontuação. Por esse motivo, somente 7 bits eram usados para codificar um total de 128 letras maiúsculas, letras minúsculas e sinais de pontuação. Mesmo com toda essa criatividade, ainda restavam 33 caracteres não usados para impressão, dos quais alguns foram utilizados, substituídos e/ou se tornaram obsoletos à medida que as tecnologias mudaram ao longo dos anos. Bastante espaço para todos, certo?

Como qualquer programador sabe, 7 é um número estranho. Não é uma bela potência de 2, mas é sedutoramente próximo. Nos anos 1960, cientistas de computação discutiram de forma acirrada entre o acréscimo de um bit extra pela conveniência de ter um belo número redondo *versus* a

praticidade de os arquivos exigirem menos espaço de armazenagem. No final, os 7 bits venceram. Contudo, na computação moderna, cada sequência de 7 bits é preenchida com um 0 extra no início<sup>1</sup>, o que nos deixou com o pior dos dois mundos – arquivos 14% maiores e a falta de flexibilidade de haver apenas 128 caracteres.

No início dos anos 1990, as pessoas perceberam que havia outros idiomas além do inglês, e que seria realmente interessante se os computadores pudessem exibí-los. Uma organização sem fins lucrativos, The Unicode Consortium (Consórcio Unicode), tentou criar um codificador de texto universal definindo codificações para todos os caracteres que tivessem de ser usados em qualquer documento de texto, em qualquer idioma. O objetivo era incluir tudo, do alfabeto latino com o qual este livro está escrito ao cirílico (кириллица), pictogramas chineses (象形), símbolos matemáticos e lógicos ( $\Sigma$ ,  $\geq$ ) e até mesmo emoticons e símbolos variados, por exemplo, o sinal de risco biológico (☣) e o símbolo da paz (☺).

O codificador resultante, como você talvez já saiba, recebeu o nome de UTF-8, o qual, de modo confuso, significa “*Universal Character Set – Transformation Format 8 bit*” (Conjunto Universal de Caracteres – Formato de Transformação de 8 bits). Os 8 *bits* nesse caso não se referem ao tamanho de cada caractere, mas ao menor tamanho exigido por um caractere para que seja exibido.

O tamanho de um caractere UTF-8 é flexível. Esses caracteres podem variar de 1 a 4 bytes, dependendo de sua posição na lista de possíveis caracteres (caracteres mais conhecidos são codificados com menos byte, caracteres mais obscuros exigem mais bytes).

Como essa codificação flexível é possível? O uso de 7 bits com um eventual 0 inútil na frente parecia ser uma falha de design do ASCII à primeira vista, mas se provou ser uma vantagem enorme para o UTF-8. Como o ASCII era muito popular, o Unicode decidiu tirar proveito desse bit 0 na frente ao declarar que todos os bytes que começam com 0 sinalizem que apenas um byte é usado no caractere, tornando os dois esquemas de codificação para ASCII e UTF-8 idênticos. Assim, os caracteres a seguir são válidos tanto para UTF-8 como para ASCII:

01000001 - A

01000010 - B

01000011 - C

Os caracteres a seguir são válidos apenas em UTF-8, e serão renderizados como caracteres que não podem ser exibidos caso o documento seja interpretado como ASCII:

11000011 10000000 - À

11000011 10011111 - ß

11000011 10100111 - ç

Além do UTF-8, há outros padrões UTF, como UTF-16, UTF-24 e UTF-32, embora documentos codificados nesses formatos sejam raros, exceto em circunstâncias incomuns, as quais estão além do escopo deste livro.

Apesar de essa “falha de design” original do ASCII ter representado uma grande vantagem para o UTF-8, a desvantagem não deixou de existir por completo. Os primeiros 8 bits de informação de cada caractere ainda conseguem codificar apenas 128 caracteres, e não 256. Em um caractere UTF-8 que exija vários bytes, bits adicionais na frente são usados, não na codificação do caractere, mas como bits de verificação para evitar dados corrompidos. Dos 32 (8 x 4) bits em caracteres de 4 bytes, apenas 21 bits são usados para codificação dos caracteres, resultando em um total de 2.097.152 caracteres possíveis, dos quais há 1.114.112 alocados atualmente.

É evidente que o problema com qualquer padrão universal de codificação de idiomas é o fato de um documento escrito em um único idioma estrangeiro poder ser muito maior do que deveria. Embora esse idioma seja constituído de apenas 100 caracteres ou mais, serão necessários 16 bits para cada caractere, em vez de somente 8, como é o caso do ASCII específico para o inglês. Isso faz com que documentos de texto em idiomas estrangeiros em UTF-8 tenham cerca do dobro do tamanho dos documentos de texto em inglês, pelo menos para os idiomas estrangeiros que não usem o conjunto de caracteres latino.

A ISO resolve esse problema criando codificações específicas para cada idioma. Assim como o Unicode, os padrões ISO têm as mesmas codificações do ASCII, mas usam o bit 0 de preenchimento no início de qualquer caractere para permitir a criação de 128 caracteres especiais para todos os idiomas que precisem deles. Essa solução é melhor para idiomas europeus, que também dependem bastante do alfabeto latino (cujos caracteres continuam nas posições 0-127 na codificação), mas exigem caracteres especiais extras. Isso permite que o ISO-8859-1 (para o alfabeto latino) tenha símbolos como de frações (por exemplo, ½) ou o sinal de

copyright (©).

Outros conjuntos de caracteres ISO, por exemplo, ISO-8859-9 (turco), ISO-8859-2 (alemão, entre outros idiomas) e ISO-8859-15 (francês, entre outros idiomas), também podem ser encontrados na internet com certa regularidade.

Apesar de a popularidade dos documentos com codificação ISO ter diminuído nos últimos anos, cerca de 9% dos sites da internet continuam codificados com alguma variante de ISO<sup>2</sup>, fazendo com que seja essencial conhecê-los e verificar as codificações antes de coletar dados de um site.

### Codificações em ação

Na seção anterior, usamos as configurações default de `urlopen` para ler documentos de texto que poderiam ser encontrados na internet. Isso funciona muito bem para a maioria dos textos em inglês. No entanto, no instante em que deparar com russo, árabe, ou até mesmo com uma palavra como “résumé”, você poderá ter problemas.

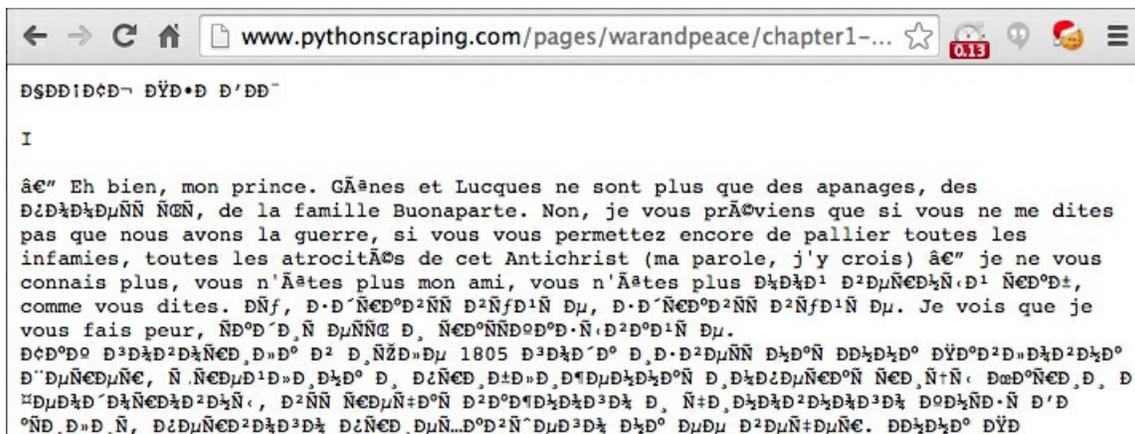
Considere o código a seguir, por exemplo:

```
from urllib.request import urlopen
textPage = urlopen('http://www.pythonscraping.com/'\
    'pages/warandpeace/chapter1-ru.txt')
print(textPage.read())
```

Esse código lê o primeiro capítulo do livro *Guerra e Paz* original (escrito em russo e em francês) e o exibe na tela. Eis uma parte do texto na tela:

```
b"\xd0\xa7\xd0\x90\xd0\xa1\xd0\xa2\xd0\xac \xd0\x9f\xd0\x95\xd0\xa0\xd0\x92\xd0\x90\xd0\xaf\n\nI\n\n\xe2\x80\x9cEh bien, mon prince. G\u00e9nes et Lucques ne sont plus que des apanages, des
```

Além disso, acessar essa página na maioria dos navegadores resultará em um texto sem nexos (veja a Figura 7.1).



*Figura 7.1 – Texto em francês e em cirílico, codificado em ISO-8859-1, a codificação de texto default em muitos navegadores.*

Mesmo para falantes nativos de russo, pode ser um pouco difícil de entender. O problema é que Python tenta ler o documento como ASCII, enquanto o navegador tenta lê-lo como um documento codificado com ISO-8859-1. Nenhum deles, é claro, percebe que é um documento em UTF-8.

Podemos definir a string como UTF-8 de modo explícito, formatando corretamente a saída com caracteres cirílicos:

```
from urllib.request import urlopen

textPage = urlopen('http://www.pythonscraping.com/\
    'pages/warandpeace/chapter1-ru.txt')
print(str(textPage.read(), 'utf-8'))
```

O código terá o seguinte aspecto se esse conceito for usado com o BeautifulSoup e Python 3.x:

```
html = urlopen('http://en.wikipedia.org/wiki/Python_(programming_language)')
bs = BeautifulSoup(html, 'html.parser')
content = bs.find('div', {'id':'mw-content-text'}).get_text()
content = bytes(content, 'UTF-8')
content = content.decode('UTF-8')
```

Python 3.x codifica todos os caracteres em UTF-8 por padrão. Você pode se sentir tentado a deixar de lado essa questão e usar a codificação UTF-8 em todo web scraper que escrever. Afinal de contas, o UTF-8 também lidará tranquilamente tanto com caracteres ASCII quanto com idiomas estrangeiros. No entanto, é importante lembrar dos 9% de sites por aí que usam alguma versão da codificação ISO também, portanto não será possível evitar esse problema por completo.

Infelizmente, no caso de documentos de texto, é impossível determinar a codificação usada por um documento de forma concreta. Algumas bibliotecas são capazes de analisar o documento e dar um bom palpite (usando um pouco de lógica para perceber que “Ñ€Ð°Ñ?Ñ?Ð°Ð°Ð·Ñ” provavelmente não é uma palavra); porém, muitas vezes, estarão erradas.

Felizmente, no caso de páginas HTML, em geral a codificação está contida em uma tag que se encontra na seção <head> do site. A maioria dos sites, em particular os sites em inglês, têm esta tag:

```
<meta charset="utf-8" />
```

Por outro lado, o [site da ECMA International](http://www.ecma-international.org) (<http://www.ecma-international.org>)

[international.org/](http://international.org/)) tem esta tag<sup>3</sup>:

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
```

Se você planeja fazer muito web scraping, particularmente de sites internacionais, procurar essa tag meta e usar a codificação recomendada por ela ao ler o conteúdo da página pode ser uma atitude inteligente.

## CSV

Ao fazer web scraping, é provável que você encontre um arquivo CSV ou um colega de trabalho que goste de dados formatados dessa maneira. Felizmente, Python tem uma [biblioteca incrível](https://docs.python.org/3.4/library/csv.html) (<https://docs.python.org/3.4/library/csv.html>) tanto para ler quanto para escrever arquivos CSV. Embora essa biblioteca seja capaz de lidar com muitas variações de CSV, esta seção tem como foco principal o formato padrão. Se houver algum caso especial com o qual você tenha de lidar, consulte a documentação.

### Lendo arquivos CSV

A biblioteca `csv` de Python está voltada essencialmente para trabalhar com arquivos locais, partindo do pressuposto de que os dados CSV necessários estarão armazenados em seu computador. Infelizmente, esse nem sempre será o caso, sobretudo quando se trata de web scraping. Há várias maneiras de contornar essa situação:

- Faça download do arquivo localmente, de modo manual, e informe a localização do arquivo para Python.
- Escreva um script Python para fazer download do arquivo, leia-o e (opcionalmente) apague o arquivo após obter seus dados.
- Obtenha o arquivo da web na forma de uma string e encapsule a string com um objeto `stringIO` de modo que ela se comporte como um arquivo.

Embora as duas primeiras opções sejam factíveis, ocupar espaço no disco rígido com arquivos quando seria fácil mantê-los em memória não é uma boa prática. É muito melhor ler o arquivo como uma string e encapsulá-la em um objeto que permita a Python tratá-la como um arquivo, sem jamais o salvar. O script a seguir obtém um arquivo CSV da internet (nesse caso, uma lista de álbuns do Monty Python em <http://pythonscraping.com/files/MontyPythonAlbums.csv>) e o exibe, linha a

linha, no terminal:

```
from urllib.request import urlopen
from io import StringIO
import csv

data = urlopen('http://pythonscraping.com/files/MontyPythonAlbums.csv')
        .read().decode('ascii', 'ignore')
dataFile = StringIO(data)
csvReader = csv.reader(dataFile)

for row in csvReader:
    print(row)
```

Eis a saída do código:

```
['Name', 'Year']
["Monty Python's Flying Circus", '1970']
['Another Monty Python Record', '1971']
["Monty Python's Previous Record", '1972']
...
```

Como podemos ver a partir do código de exemplo, o objeto de leitura devolvido por `csv.reader` é iterável e é composto de objetos lista de Python. Por causa disso, cada linha do objeto `csvReader` é acessível da seguinte maneira:

```
for row in csvReader:
    print('The album "' + row[0] + '" was released in ' + str(row[1]))
```

Eis a saída:

```
The album "Name" was released in Year
The album "Monty Python's Flying Circus" was released in 1970
The album "Another Monty Python Record" was released in 1971
The album "Monty Python's Previous Record" was released in 1972
...
```

Observe a primeira linha: The album "Name" was released in Year. Embora esse seja um resultado que pode ser ignorado com facilidade ao escrever um código de exemplo, você não iria querer que ele fosse incluído em seus dados no mundo real. Um programador mais inexperiente poderia simplesmente ignorar a primeira linha do objeto `csvReader`, ou escrever um caso especial para tratá-la. Felizmente, uma alternativa à função `csv.reader` cuida de tudo isso para você de modo automático. Entra em cena um

`DictReader`:

```
from urllib.request import urlopen
from io import StringIO
import csv

data = urlopen('http://pythonscraping.com/files/MontyPythonAlbums.csv')
```

```

        .read().decode('ascii', 'ignore')
dataFile = StringIO(data)
dictReader = csv.DictReader(dataFile)

print(dictReader.fieldnames)

for row in dictReader:
    print(row)

```

`csv.DictReader` devolve os valores de cada linha do arquivo CSV na forma de objetos dicionário, em vez de objetos lista, com os nomes dos campos armazenados na variável `dictReader.fieldnames` e como chaves em cada objeto dicionário:

```

['Name', 'Year']
{'Name': 'Monty Python's Flying Circus', 'Year': '1970'}
{'Name': 'Another Monty Python Record', 'Year': '1971'}
{'Name': 'Monty Python's Previous Record', 'Year': '1972'}

```

A desvantagem, evidentemente, é que demora um pouco mais para criar, processar e exibir esses objetos `DictReader`, em comparação com `csvReader`, porém a conveniência e a usabilidade muitas vezes compensam o overhead adicional. Tenha em mente também que, quando se trata de web scraping, o overhead necessário para requisitar e obter dados dos sites de um servidor externo quase sempre será o fator limitante inevitável para qualquer programa que você escrever, portanto preocupar-se com qual técnica proporcionará uma redução de alguns microssegundos no tempo total de execução muitas vezes será uma questão discutível!

## PDF

Como usuária de Linux, sei do sofrimento de receber um arquivo *.docx* que meu software que não é Microsoft mutila, e da luta para tentar encontrar os codecs para interpretar algum novo formato de mídia da Apple. Em certos aspectos, a Adobe foi revolucionária ao criar seu PDF (Portable Document Format, ou Formato de Documento Portável) em 1993. Os PDFs permitiram que usuários de diferentes plataformas vissem imagens e documentos de texto exatamente do mesmo modo, não importando a plataforma em que fossem visualizados.

Embora armazenar PDFs na web seja, de certo modo, ultrapassado (por que armazenar conteúdo em um formato estático, lento para carregar, quando ele poderia ser escrito em HTML?), os PDFs continuam presentes em todos os lugares, particularmente quando lidamos com formulários

oficiais e arquivamentos.

Em 2009, um britânico chamado Nick Innes foi destaque nos noticiários quando solicitou informações sobre resultados de testes de estudantes da rede pública ao Buckinghamshire City Council (Câmara Municipal de Buckinghamshire), disponíveis por conta da Freedom of Information Act (Lei da Liberdade de Informação) do Reino Unido. Depois de algumas requisições e recusas repetidas, ele finalmente recebeu a informação que procurava – na forma de 184 documentos PDF.

Embora Innes tenha persistido e, em algum momento, recebido um banco de dados formatado de modo mais apropriado, se fosse expert em web scraper, provavelmente teria evitado muito desperdício de tempo nos tribunais e usado os documentos PDF de forma direta, com um dos muitos módulos de parsing de PDF de Python.

Infelizmente, muitas das bibliotecas de parsing de PDF disponíveis para Python 2.x não foram atualizadas com o lançamento de Python 3.x. Entretanto, como o PDF é um formato de documento relativamente simples e de código aberto, muitas bibliotecas Python boas, mesmo em Python 3.x, são capazes de lê-lo.

O PDFMiner3K é uma dessas bibliotecas relativamente fáceis de usar. É flexível e pode ser usada na linha de comando ou integrada com um código existente. Também é capaz de lidar com várias codificações de idiomas – mais uma vez, é um recurso que, com frequência, é conveniente na web.

Podemos instalar a biblioteca como de costume, usando pip ou fazendo download do [módulo Python](https://pypi.org/project/pdfminer3k/) em <https://pypi.org/project/pdfminer3k/>; instale-o descompactando a pasta e executando o seguinte comando:

```
$ python setup.py install
```

A documentação está disponível em [/pdfminer3k-1.3.0/docs/index.html](https://pdfminer3k-1.3.0/docs/index.html) na pasta extraída, embora a documentação atual esteja mais voltada para a interface de linha de comando do que uma integração com código Python.

Eis uma implementação básica que permite ler PDFs arbitrários em uma string, dado um objeto de arquivo local:

```
from urllib.request import urlopen
from pdfminer.pdfinterp import PDFResourceManager, process_pdf
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from io import StringIO
from io import open
```

```

def readPDF(pdfFile):
    rsrcmgr = PDFResourceManager()
    retstr = StringIO()
    laparams = LAParams()
    device = TextConverter(rsrcmgr, retstr, laparams=laparams)

    process_pdf(rsrcmgr, device, pdfFile)
    device.close()

    content = retstr.getvalue()
    retstr.close()
    return content

pdfFile = urlopen('http://pythonscraping.com/'
    'pages/warandpeace/chapter1.pdf')
outputString = readPDF(pdfFile)
print(outputString)
pdfFile.close()

```

Esse código tem como saída o texto simples que já conhecemos:

```
CHAPTER I
```

```
"Well, Prince, so Genoa and Lucca are now just family estates of
the Buonapartes. But I warn you, if you don't tell me that this
means war, if you still try to defend the infamies and horrors
perpetrated by that Antichrist- I really believe he is Antichrist- I will
```

O aspecto interessante sobre esse leitor de PDF é que, se estivermos trabalhando com arquivos locais, podemos substituir um objeto comum de arquivo Python pelo objeto devolvido por `urlopen` e usar esta linha:

```
pdfFile = open('../pages/warandpeace/chapter1.pdf', 'rb')
```

A saída talvez não seja perfeita, em especial para PDFs com imagens, textos formatados de maneira inusitada ou organizados em tabelas ou gráficos. No entanto, para a maioria dos PDFs contendo apenas texto, a saída não deverá ser diferente de um PDF que fosse um arquivo-texto.

## Microsoft Word e .docx

Correndo o risco de ofender meus amigos na Microsoft: não gosto do Microsoft Word. Não é porque seja necessariamente um software ruim, mas por causa do modo como seus usuários o utilizam de forma indevida. Ele tem um talento específico para transformar o que deveriam ser documentos textuais ou PDFs simples em criaturas enormes, lentas e difíceis de abrir, que muitas vezes perdem toda a formatação de uma máquina para outra; além disso, por alguma razão, os documentos são editáveis, quando o conteúdo em geral deveria ser estático.

Arquivos Word foram projetados para criação de conteúdo, não para compartilhamento. Apesar disso, eles estão por toda parte em certos sites, contendo documentos importantes, informações e até mesmo gráficos e multimídia; em suma, tudo que poderia e deveria ser criado com HTML.

Até aproximadamente 2008, os produtos Microsoft Office usavam o formato de arquivo proprietário *.doc*. Esse formato de arquivo binário era difícil de ler e tinha pouco suporte por parte de outros processadores de texto. Em um esforço para se atualizar e adotar um padrão que fosse usado por vários outros softwares, a Microsoft decidiu usar o padrão Open Office baseado em XML, o que tornou os arquivos compatíveis com softwares de código aberto e outros.

Infelizmente, o suporte de Python para esse formato de arquivo, usado pelo Google Docs, Open Office e Microsoft Office, ainda não é muito bom. Há uma [biblioteca python-docx](http://python-docx.readthedocs.io/en/latest/) (<http://python-docx.readthedocs.io/en/latest/>), mas ela somente oferece aos usuários a capacidade de criar documentos e ler dados básicos do arquivo, como o tamanho e o título, e não o conteúdo em si. Para ler o conteúdo de um arquivo Microsoft Office, é necessário desenvolver a sua própria solução.

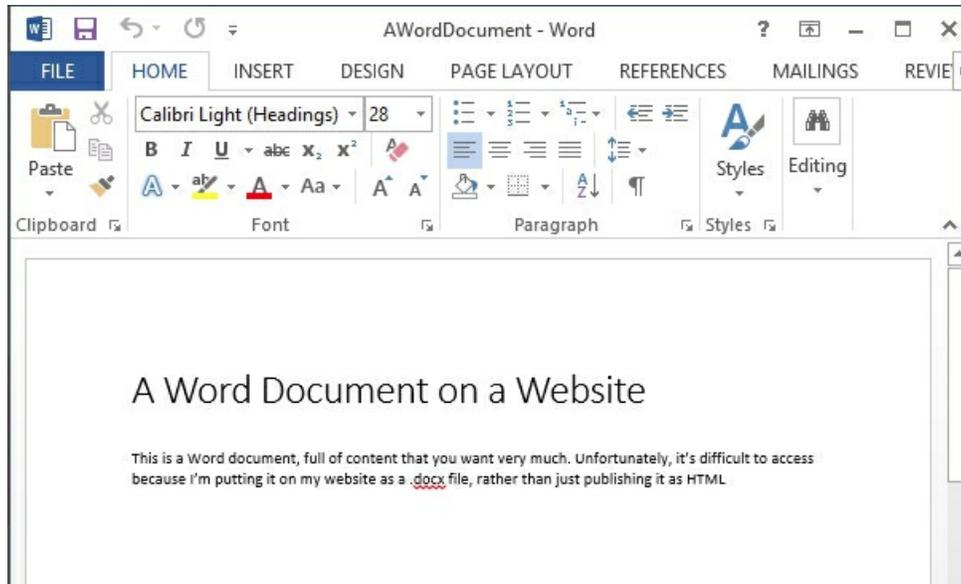
O primeiro passo é ler o XML do arquivo:

```
from zipfile import ZipFile
from urllib.request import urlopen
from io import BytesIO

wordFile = urlopen('http://pythonscraping.com/pages/AWordDocument.docx').read()
wordFile = BytesIO(wordFile)
document = ZipFile(wordFile)
xml_content = document.read('word/document.xml')
print(xml_content.decode('utf-8'))
```

Esse código lê um documento Word remoto como um objeto de arquivo binário (`BytesIO` é análogo a `StringIO`, usado antes neste capítulo), descompacta o arquivo usando a biblioteca `zipfile` de Python (todos os arquivos *.docx* são compactados para economizar espaço) e, em seguida, lê o arquivo descompactado, que está em XML.

A Figura 7.2 mostra o documento Word que está em <http://pythonscraping.com/pages/AWordDocument.docx>.



*Figura 7.2 – Este é um documento Word cheio de conteúdo que você quer muito, mas é difícil acessá-lo, pois eu o coloquei em meu site como um arquivo .docx em vez de publicá-lo em HTML.*

Eis a saída do script Python que lê meu documento Word simples:

```
<!--?xml version="1.0" encoding="UTF-8" standalone="yes"?->
<w:document mc:ignorable="w14 w15 wp14" xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/math" xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" xmlns:o="urn:schemas-microsoft-com:office:office" xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships" xmlns:v="urn:schemas-microsoft-com:vm" xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main" xmlns:w10="urn:schemas-microsoft-com:office:word" xmlns:w14="http://schemas.microsoft.com/office/word/2010/wordml" xmlns:w15="http://schemas.microsoft.com/office/word/2012/wordml" xmlns:wne="http://schemas.microsoft.com/office/word/2006/wordml" xmlns:wp="http://schemas.openxmlformats.org/drawingml/2006/wordprocessingDrawing" xmlns:wp14="http://schemas.microsoft.com/office/word/2010/wordprocessingDrawing" xmlns:wpc="http://schemas.microsoft.com/office/word/2010/wordprocessingCanvas" xmlns:wpg="http://schemas.microsoft.com/office/word/2010/wordprocessingGroup" xmlns:wpi="http://schemas.microsoft.com/office/word/2010/wordprocessingInk" xmlns:wps="http://schemas.microsoft.com/office/word/2010/wordprocessingShape"><w:body><w:p w:rsidp="00764658" w:rsidr="00764658" w:rsidrdefault="00764658"><w:ppr><w:pstyle w:val="Title"></w:pstyle></w:ppr><w:r><w:t>A Word Document on a Website</w:t></w:r><w:bookmarkstart w:id="0" w:name="_GoBack"></w:bookmarkstart><w:bookmarkend w:id="0"></w:bookmarkend></w:p><w:p w:rsidp="00764658" w:rsidr="00764658" w:rsidrdefault="00764658"></w:p><w:p w:rsidp="00764658" w:rsidr="00764658" w:rsidrdefault="00764658" w:rsidrpr="00764658"><w:r><w:t>This is a Word document, full of content that you want very much. Unfortunately, it's difficult to access because I'm putting
```

```

it on my website as a .</w:t></w:r><w:prooferr w:type="spellStart"></
w:prooferr><w:r><w:t>docx</w:t></w:r><w:prooferr w:type="spellEnd"></
w:prooferr> <w:r> <w:t xml:space="preserve"> file, rather than just p
ublishing it as HTML</w:t> </w:r> </w:p> <w:sectpr w:rsidr="00764658"
w:rsidrpr="00764658"> <w:pgszw:h="15840" w:w="12240"></w:pgsz><w:pgm
ar w:bottom="1440" w:footer="720" w:gutter="0" w:header="720" w:left=
"1440" w:right="1440" w:top="1440"></w:pgmar> <w:cols w:space="720"><
/w:cols&g; <w:docgrid w:linepitch="360"></w:docgrid> </w:sectpr> </w:
body> </w:document>

```

É evidente que há muitos metadados nessa saída, mas o conteúdo textual que você quer está escondido no meio dessas informações. Felizmente, todos os textos que estão no documento, inclusive o título no início, estão contidos em tags `w:t`, e isso facilita obtê-los:

```

from zipfile import ZipFile
from urllib.request import urlopen
from io import BytesIO
from bs4 import BeautifulSoup

wordFile = urlopen('http://pythonscraping.com/pages/AWordDocument.docx').read()
wordFile = BytesIO(wordFile)
document = ZipFile(wordFile)
xml_content = document.read('word/document.xml')

wordObj = BeautifulSoup(xml_content.decode('utf-8'), 'xml')
textStrings = wordObj.find_all('w:t')

for textElem in textStrings:
    print(textElem.text)

```

Observe que, em vez do parser *html.parser* que comumente usamos com o BeautifulSoup, passamos o parser *xml*. Isso porque os dois-pontos não são padrão nos nomes de tags HTML como `w:t`, e o *html.parser* não os reconhece.

A saída não é perfeita, mas está quase lá, e exibir cada tag `w:t` em uma nova linha facilita ver como o Word separa o texto:

```

A Word Document on a Website
This is a Word document, full of content that you want very much. Unfortunately,
it's difficult to access because I'm putting it on my website as a .
docx
file, rather than just publishing it as HTML

```

Note que a palavra “docx” está em sua própria linha. No XML original, ela está cercada pela tag `<w:proofErr w:type="spellStart"/>`. Essa é a maneira como o Word destaca “docx”, com o underline vermelho irregular, informando que ele acredita haver um erro de ortografia no nome de seu próprio formato de arquivo.

O título do documento é precedido pela tag de descrição de estilo `<w:pstyle w:val="Title">`. Embora isso não torne extremamente fácil para nós identificar títulos (ou outros textos estilizados) como esse, usar os recursos de navegação do BeautifulSoup pode ser conveniente:

```
textStrings = wordObj.find_all('w:t')

for textElem in textStrings:
    style = textElem.parent.parent.find('w:pStyle')
    if style is not None and style['w:val'] == 'Title':
        print('Title is: {}'.format(textElem.text))
    else:
        print(textElem.text)
```

É fácil expandir essa função de modo a exibir tags em torno de vários estilos de texto ou nomeá-las de alguma outra forma.

---

- 1 Esse bit de “preenchimento” voltará a nos assombrar com os padrões ISO um pouco mais adiante.
- 2 De acordo com a [W3Techs](http://w3techs.com/technologies/history_overview/character_encoding) ([http://w3techs.com/technologies/history\\_overview/character\\_encoding](http://w3techs.com/technologies/history_overview/character_encoding)), que usa web crawlers para obter esses tipos de estatística.
- 3 A ECMA era uma das colaboradoras originais do padrão ISO, portanto não é nenhuma surpresa que seu site esteja codificado com uma variante do ISO.

# Limpendo dados sujos

Até agora no livro, ignoramos o problema de dados mal formatados usando fontes de dados em geral com boa formatação, descartando totalmente os dados caso não correspondessem ao que esperávamos. Com frequência, porém, em web scraping, não podemos ser exigentes demais quanto ao local onde os dados são obtidos ou à sua aparência.

Como consequência de pontuação incorreta, uso inconsistente de letras maiúsculas, quebras de linha e erros de ortografia, dados sujos podem ser um grande problema na web. Este capítulo apresenta algumas ferramentas e técnicas para ajudar a evitar o problema na origem, mudando o modo de escrever o código e limpando os dados depois que estiverem no banco de dados.

## Código para limpeza de dados

Assim como escrevemos código para tratar as exceções que se manifestarem, devemos usar um código defensivo para lidar com o inesperado.

Em linguística, um *n-grama* é uma sequência de  $n$  palavras usadas em um texto ou discurso. Ao fazer análise de idiomas naturais, muitas vezes pode ser conveniente separar um texto procurando  $n$ -gramas comuns ou conjuntos recorrentes de palavras usadas com frequência juntas.

Esta seção visa obter  $n$ -gramas formatados de modo apropriado, em vez de usá-los para alguma análise. Mais adiante, no *Capítulo 9*, veremos bigramas (2-gramas) e trigamas (3-gramas) em ação para fazer resumo e análise de texto.

O código a seguir devolve uma lista de bigramas encontrados no artigo da Wikipédia sobre a linguagem de programação Python:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

def getNgrams(content, n):
```



```
return output
```

Esse código substitui todas as instâncias do caractere de quebra de linha por um espaço, remove citações como [123] e elimina todas as strings vazias, resultantes de vários espaços em branco sequenciais. Em seguida, os caracteres de escape são eliminados ao codificar o conteúdo com UTF-8.

Esses passos melhoram bastante o resultado da função, mas alguns problemas persistem:

```
['years', 'ago('), ['ago(', '-'], ['- ', '-'], ['- ', ')'], [')', 'Stable']
```

É possível melhorar a saída removendo todos os caracteres de pontuação antes e depois de cada palavra (eliminado a pontuação). Com isso, preservamos sinais como hifens no interior das palavras, mas eliminamos strings compostas de apenas um único sinal de pontuação depois que as strings vazias foram eliminadas.

É claro que a pontuação em si tem sentido, e simplesmente a remover pode resultar na perda de algumas informações importantes. Por exemplo, podemos supor que um ponto final seguido de um espaço significa o fim de uma sentença ou frase completa. Talvez você queira descartar n-gramas que cruzem um ponto final, como nesse caso, e considerar apenas os n-gramas criados dentro da sentença.

Por exemplo, dado o texto:

```
Python features a dynamic type system and automatic memory management.  
It supports multiple programming paradigms...
```

o bigrama ['memory', 'management'] seria válido, mas não o bigrama ['management', 'It'].

Agora que temos uma lista mais longa de “tarefas de limpeza”, introduzimos o conceito de “sentenças” e o programa, de modo geral, se tornou mais complexo, é melhor separar o código em quatro funções diferentes:

```
from urllib.request import urlopen  
from bs4 import BeautifulSoup  
import re  
import string  
  
def cleanSentence(sentence):  
    sentence = sentence.split(' ')  
    sentence = [word.strip(string.punctuation+string.whitespace)  
                for word in sentence]  
    sentence = [word for word in sentence if len(word) > 1  
                or (word.lower() == 'a' or word.lower() == 'i')]  
    return sentence
```

```

def cleanInput(content):
    content = re.sub('\n|[[\d+\\]]', ' ', content)
    content = bytes(content, "UTF-8")
    content = content.decode("ascii", "ignore")
    sentences = content.split('. ')
    return [cleanSentence(sentence) for sentence in sentences]

def getNgramsFromSentence(content, n):
    output = []
    for i in range(len(content)-n+1):
        output.append(content[i:i+n])
    return output

def getNgrams(content, n):
    content = cleanInput(content)
    ngrams = []
    for sentence in content:
        ngrams.extend(getNgramsFromSentence(sentence, n))
    return(ngrams)

```

`getNgrams` continua o ponto de entrada básico para o programa. `cleanInput` remove quebras de linha e citações, como antes, mas também separa o texto em “sentenças” com base na posição dos pontos finais seguidos de um espaço. A função também chama `cleanSentence`, que separa a sentença em palavras, remove pontuações e espaços em branco, além de eliminar palavras com um único caractere que não sejam *I* e *a*.

As linhas principais que criam os n-gramas foram passadas para `getNgramsFromSentence`, que é chamada em cada sentença por `getNgrams`. Isso garante que n-gramas que se estendam por várias sentenças não sejam criados.

Observe o uso de `string.punctuation` e `string.whitespace` para obter uma lista de todos os caracteres de pontuação em Python. A saída de `string.punctuation` pode ser vista em um terminal Python:

```

>>> import string
>>> print(string.punctuation)
! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~

```

`print(string.whitespace)` gera uma saída muito menos interessante (afinal de contas, são apenas espaços em branco), mas contém caracteres de espaços incluindo espaços sem quebras, tabulações e quebras de linha.

Ao usar `item.strip(string.punctuation+string.whitespace)` em um laço que itera por todas as palavras do conteúdo, qualquer caractere de pontuação de qualquer lado da palavra será removido, porém palavras com hífen (em que o caractere de pontuação está cercado por letras de ambos os lados)



```
    ngrams.update(newNgrams)
return(ngrams)
```

Há várias maneiras diferentes de fazer isso, por exemplo, adicionar n-gramas em um objeto dicionário no qual o valor da lista aponte para um contador do número de vezes em que um n-grama foi visto. Essa solução tem a desvantagem de exigir um pouco mais de gerenciamento e complicar a ordenação. No entanto, usar um objeto `counter` também tem uma desvantagem: ele não é capaz de armazenar listas (não é possível ter hashes com listas), portanto é necessário fazer antes uma conversão para strings usando um `' '.join(ngram)` em uma list comprehension para cada n-grama.

Eis o resultado:

```
Counter({'Python Software': 37, 'Software Foundation': 37, 'of the': 34,
'of Python': 28, 'in Python': 24, 'in the': 23, 'van Rossum': 20, 'to the':
20, 'such as': 19, 'Retrieved February': 19, 'is a': 16, 'from the': 16,
'Python Enhancement': 15,...
```

Quando escrevi este livro, havia um total de 7.275 bigramas e 5.628 bigramas únicos; o bigrama mais popular era “Software Foundation”, seguido de “Python Software”. No entanto, a análise do resultado mostra que “Python Software” aparece na forma de “Python software” mais duas vezes. De modo semelhante, “van Rossum” e “Van Rossum” aparecem de forma separada na lista.

Acrescentar a linha:

```
content = content.upper()
```

na função `cleanInput` mantém constante o número total de bigramas encontrado, 7.275, enquanto reduz o número de bigramas únicos para 5.479.

Além disso, em geral é bom parar e considerar a capacidade de processamento que você está disposto a usar na normalização dos dados. Há uma série de situações em que palavras grafadas de modo diferente são equivalentes, mas, para resolver essa equivalência, é necessário executar uma verificação em cada uma das palavras a fim de ver se ela corresponde a algum de seus equivalentes previamente programados.

Por exemplo, tanto “Python 1st” quanto “Python first” aparecem na lista de bigramas. No entanto, fazer uma regra abrangente que determine que “todo *first, second, third* etc. será resolvido para *1st, 2nd, 3rd* etc. (ou vice-versa)” resultaria em cerca de dez verificações adicionais por palavra.

De modo semelhante, o uso inconsistente de hifens (“co-ordinated” *versus*

“coordinated”), erros de ortografia e outras incongruências das línguas naturais afetarão o agrupamento de n-gramas e poderão deturpar o resultado caso as incongruências sejam muito comuns.

Uma solução, no caso de palavras com hifens, seria removê-los totalmente e tratar a palavra como uma única string, o que exigiria apenas uma operação. Contudo, isso significaria também que expressões com hifens (uma ocorrência comum em inglês, como em *all-too-common occurrence*) seriam tratadas como uma única palavra. Por outro lado, tratar hifens como espaços poderia ser uma solução mais apropriada. Basta ficar preparado para o ocasional surgimento de um “coordinated” e um “ordinated attack”!

## **Limpeza dos dados após a coleta**

Há um número limitado de tarefas que você pode (e quer) fazer no código. Além disso, o conjunto de dados com o qual está lidando talvez não tenha sido criado por você, ou é um conjunto para o qual seria difícil até mesmo saber como limpar sem vê-lo antes.

Uma reação instintiva de muitos programadores a esse tipo de situação seria “escrever um script”, o que pode ser uma excelente solução. No entanto, ferramentas de terceiros, como o OpenRefine, não apenas são capazes de limpar os dados de modo rápido e fácil como também permitem que seus dados sejam facilmente vistos e usados por pessoas que não são programadores.

### **OpenRefine**

O [OpenRefine](http://openrefine.org/) (<http://openrefine.org/>) é um projeto de código aberto, iniciado por uma empresa chamada Metaweb em 2009. A Google adquiriu a Metaweb em 2010, alterando o nome do projeto de Freebase Gridworks para Google Refine. Em 2012, a Google deixou de dar suporte para o Refine e mudou o nome de novo, agora para OpenRefine, e qualquer pessoa é bem-vinda para contribuir com o desenvolvimento do projeto.

### **Instalação**

O OpenRefine é peculiar porque, embora sua interface seja executada em um navegador, do ponto de vista técnico, é uma aplicação desktop que deve ser baixada e instalada. Faça download da aplicação para Linux, Windows e macOS a partir do [site](http://openrefine.org/download.html) (<http://openrefine.org/download.html>).

Se você é usuário de Mac e deparar com algum problema para abrir o arquivo, acesse System Preferences → Security & Privacy → General (Preferências do sistema → Segurança e Privacidade → Geral). Em “Allow apps downloaded from” (Permitir apps transferidos de), selecione Anywhere (Qualquer lugar). Infelizmente, durante a transição de um projeto Google para um projeto de código aberto, o OpenRefine parece ter perdido sua legitimidade aos olhos da Apple.

Para usar o OpenRefine, é necessário salvar seus dados como um arquivo CSV (consulte a seção “*Armazenando dados no formato CSV*” caso precise lembrar como fazer isso). De modo alternativo, se seus dados estiverem armazenados em um banco de dados, talvez seja possível exportá-los para um arquivo CSV.

## Usando o OpenRefine

Nos exemplos a seguir, usaremos dados coletados da tabela de comparação entre editores de texto da Wikipédia: “[Comparison of Text Editors](https://en.wikipedia.org/wiki/Comparison_of_text_editors)” ([https://en.wikipedia.org/wiki/Comparison\\_of\\_text\\_editors](https://en.wikipedia.org/wiki/Comparison_of_text_editors)) – veja a Figura 8.1. Apesar de estar relativamente bem formatada, essa tabela contém muitas modificações feitas pelas pessoas com o passar do tempo, portanto apresenta pequenas inconsistências de formatação. Além disso, como a intenção é que os dados sejam lidos por seres humanos e não por máquinas, algumas das opções de formatação (por exemplo, o uso de “Free” [Gratuito] em vez de “\$0.00”) não são apropriadas como entrada para programas.

All	Name	Creator	First public release	Latest stable version	Programming language	Cost (US\$)	Software license	Open source
	1. Acme	Rob Pike	1993	Plan 9 and Inferno	C	\$0	LPL (OSI approved)	Yes
	2. AkelPad	Alexey Kuznetsov, Alexander Shengalts	2003	4.9.0	C	\$0	BSD	Yes
	3. Alphatik	Vince Darley	1999	8.3.3		\$40	Proprietary, with BSD components	No
	4. Aquamacs	David Reitter	2005	3.0a	C, Emacs Lisp	\$0	GPL	Yes
	5. Atom	GitHub	2014	0.132.0	HTML, CSS, JavaScript, C++	\$0	MIT	Yes
	6. BBEEdit	Rich Siegel	1992-04	10.5.12	Objective-C, Objective-C++	\$49.99	Proprietary	No
	7. Bluefish	Bluefish Development Team	1999	2.2.6	C	\$0	GPL	Yes
	8. Coda	Panic	2007	2.0.12	Objective-C	\$99	Proprietary	No
	9. ConTEXT	ConTEXT Project Ltd	1999	0.98.6	Object Pascal (Delphi)	\$0	BSD	Yes
	10. Crimson editor	Ingyu Kang, Emerald editor team	1999	3.7Z	C++	\$0	GPL	Yes
	11. Diakonos	Pistos	2004	0.9.2	Ruby	\$0	MIT	Yes
	12. E Text Editor	Alexander Sligsen	2005	2.0.2		\$46.95	Proprietary, with BSD components	No
	13. ed	Ken Thompson	1970	unchanged from original	C	\$0	?	Yes
	14. EditPlus	Sangil Kim	1998	3.5	C++	\$35	Shareware	No
	15. Editra	Cody Precord	2007	0.6.77	Python	\$0	wxWindows license	Yes

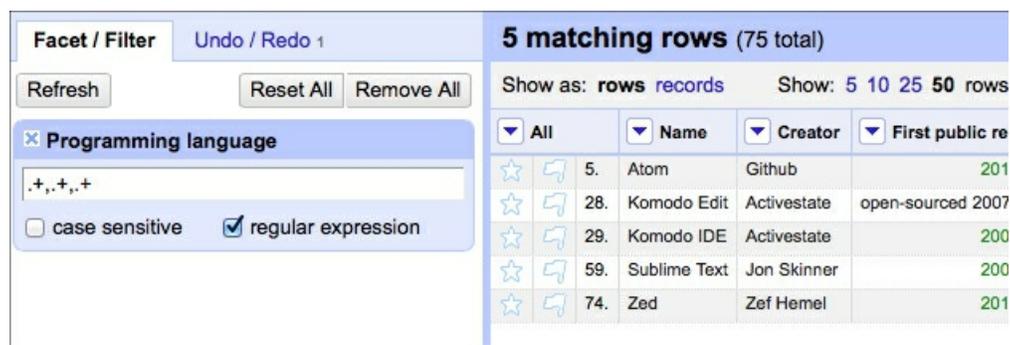
Figura 8.1 – Dados de “*Comparison of Text Editors*” (Comparação entre editores de texto) da Wikipédia, como mostra a tela principal do OpenRefine. O primeiro detalhe a ser observado sobre o OpenRefine é que há uma seta ao lado de cada nome de coluna, a qual oferece um menu de ferramentas

que podem ser usadas na respectiva coluna para filtrar, ordenar, transformar ou remover dados.

### Filtragem

A filtragem de dados pode ser feita com dois métodos: filtros e facetas (facets). Os filtros são convenientes quando usamos expressões regulares para filtrar os dados; por exemplo, “Mostre-me apenas os dados que contenham três ou mais linguagem de programação separadas por vírgula na coluna *Programming language*”, como vemos na Figura 8.2.

Os filtros podem ser combinados, editados e acrescentados com facilidade manipulando os blocos da coluna à direita. Também podem ser combinados com facetas (facets).



The screenshot shows a data table with a filter applied to the 'Programming language' column. The filter is a regular expression: `.+,.,.+`. The interface includes buttons for 'Refresh', 'Reset All', and 'Remove All'. The table shows 5 matching rows out of 75 total. The columns are 'All', 'Name', 'Creator', and 'First public re'. The rows are:

All	Name	Creator	First public re
5.	Atom	Github	201
28.	Komodo Edit	Activestate	open-sourced 2007
29.	Komodo IDE	Activestate	200
59.	Sublime Text	Jon Skinner	200
74.	Zed	Zef Hemel	201

Figura 8.2 – A expressão regular “.+,.,.+” seleciona valores que tenham pelo menos três itens separados por vírgula.

As facetas são ótimas para incluir ou excluir dados com base em todo o conteúdo da coluna. (Por exemplo, “Mostre todas as linhas que usem a licença GPL ou MIT, cujo lançamento inicial tenha sido após 2005”, como vemos na Figura 8.3). Elas têm ferramentas de filtragem embutidas. Por exemplo, filtrar com base em um valor numérico provê barras deslizantes para selecionar o intervalo de valores que você quer incluir.

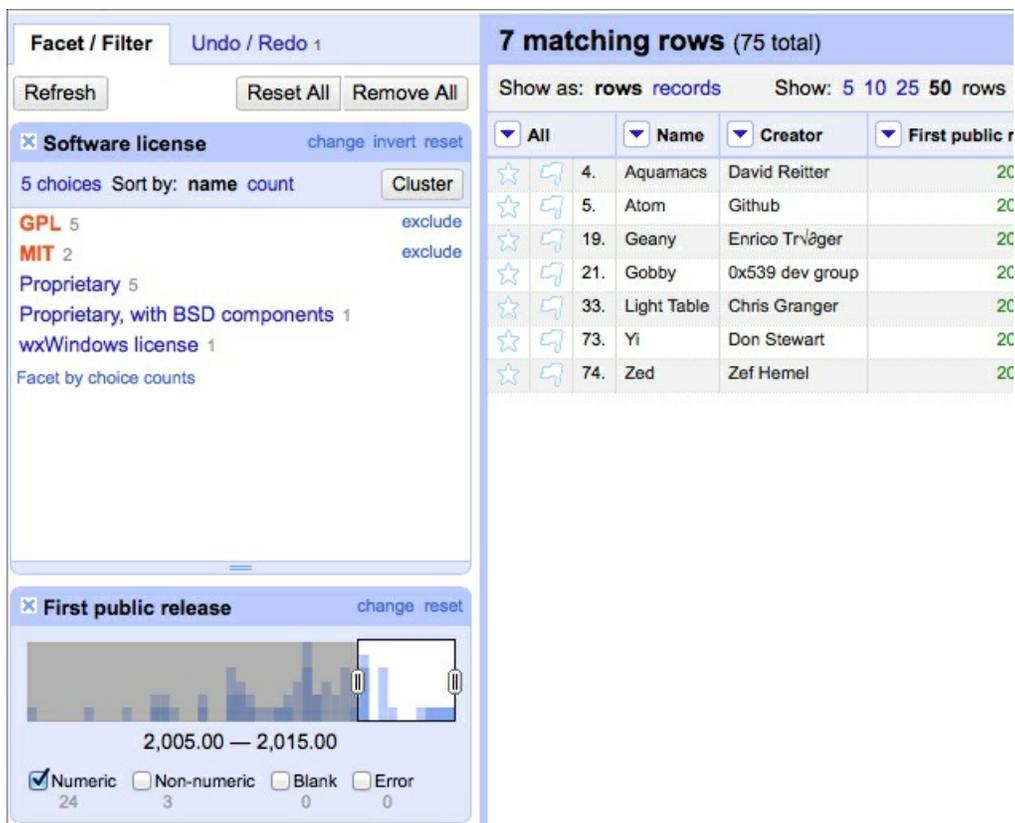


Figura 8.3 – Exibe todos os editores de texto que usam licença GPL ou MIT, cujo lançamento inicial ao público tenha ocorrido após 2005.

Não importa como você filtre seus dados, eles poderão ser exportados a qualquer momento para um dos vários tipos de formatos aceitos pelo OpenRefine. Esses formatos incluem CSV, HTML (uma tabela HTML), Excel e vários outros.

### Limpeza

A filtragem de dados poderá ser feita com sucesso apenas se os dados estiverem relativamente limpos, para começar. No exemplo de faceta na seção anterior, um editor de texto que tivesse uma data de lançamento igual a 01-01-2006 não teria sido selecionado na faceta “First public release” (Primeira versão pública), que procurava um valor igual a 2006 e ignorava valores que não se parecessem com esse.

A transformação de dados é feita no OpenRefine usando a OpenRefine Expression Language (Linguagem de Expressão do OpenRefine), chamada GREL (o G é remanescente do nome anterior do OpenRefine, que era Google Refine). Essa linguagem é usada para criar funções lambda pequenas que transformam os valores das células com base em regras

simples. Por exemplo:

```
if(value.length() != 4, "invalid", value)
```

Quando essa função é aplicada na coluna “First stable release” (Primeira versão estável), ela preserva os valores das células em que a data esteja no formato AAAA e marca todas as demais colunas como `invalid` (Figura 8.4).

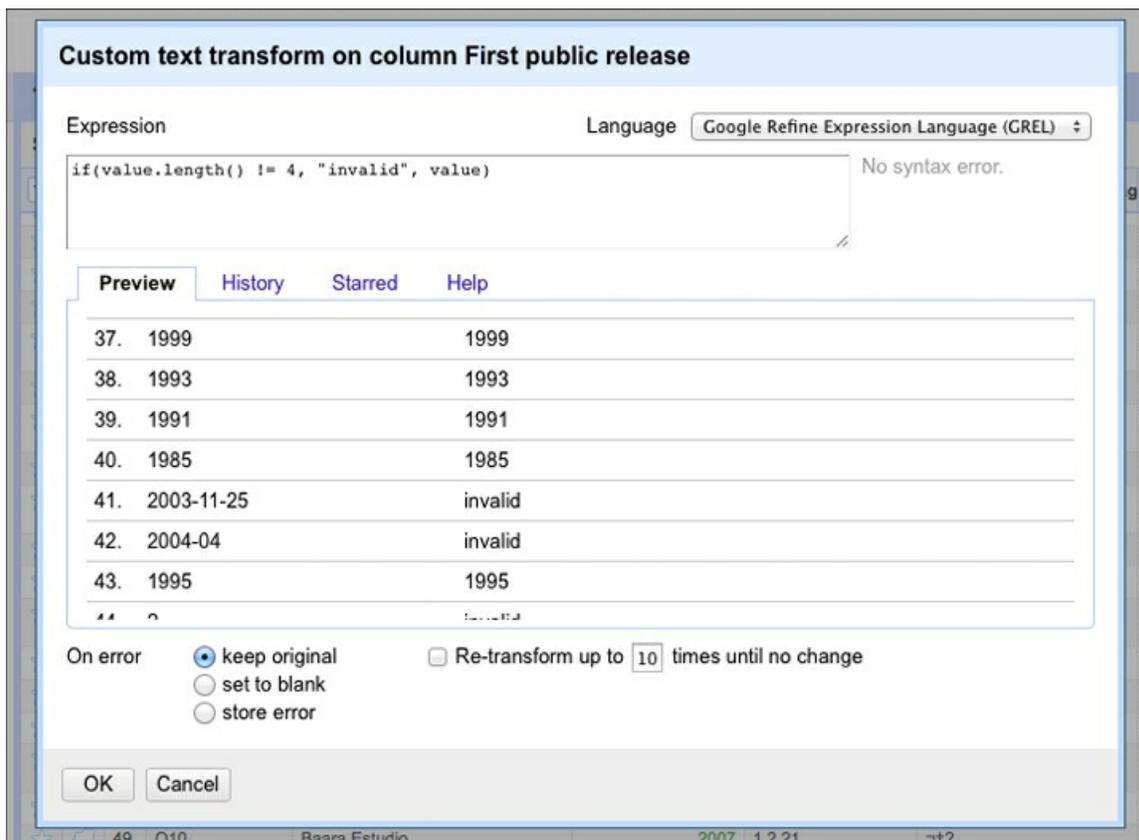


Figura 8.4 – Inserindo uma instrução GREL em um projeto (uma visualização prévia é exibida abaixo da instrução).

Instruções GREL arbitrárias podem ser aplicadas clicando na seta para baixo ao lado do nome de qualquer coluna e selecionando Edit cells → Transform (Editar células → Transformar).

No entanto, marcar todos os valores que não sejam ideais como inválidos, embora facilite identificá-los, não traz muitas vantagens. Seria melhor, se possível, tentar aproveitar as informações desses valores mal formatados. Podemos fazer isso usando a função `match` da GREL:

```
value.match(".*([0-9]{4}).*").get(0)
```

Essa instrução tenta fazer uma correspondência entre o valor da string e a expressão regular especificada. Se a expressão regular corresponder à

string, um array será devolvido. Qualquer substring que corresponda ao “grupo de captura” da expressão regular (demarcado pelos parênteses na expressão; nesse exemplo, é `[0-9]{4}`) será devolvida como valores do array.

Com efeito, esse código encontra todas as ocorrências de quatro decimais seguidos e devolve o primeiro. Em geral, isso é suficiente para extrair anos de um texto ou de datas mal formatadas. Também tem a vantagem de devolver `null` para datas inexistentes. (A GREL não lança uma exceção de ponteiro nulo quando executa operações em uma variável nula.)

Várias outras transformações de dados são possíveis com edição de células e a GREL. Na [\*página do OpenRefine no GitHub\*](#) há um guia completo da linguagem

(<https://github.com/OpenRefine/OpenRefine/wiki/Documentation-For-Users>).

## Lendo e escrevendo em idiomas naturais

Até agora, os dados com os quais vínhamos trabalhando, de modo geral, estavam na forma de números ou de valores contáveis. Na maioria dos casos, simplesmente armazenávamos os dados sem efetuar qualquer análise *a posteriori*. Este capítulo procura abordar o complicado assunto da língua inglesa<sup>1</sup>.

Como o Google sabe o que você está procurando quando digita “cute kitten” (gatinho bonitinho) na Pesquisa de Imagens? Ele sabe por causa do texto que acompanha as imagens de gatinhos bonitinhos. Como o YouTube sabe que deve apresentar determinado esquete do Monty Python quando você digita “dead parrot” (papagaio morto) na caixa de pesquisa? Por causa do título e do texto de descrição que acompanha cada vídeo armazenado.

De fato, até mesmo digitar termos como “deceased bird monty python” (pássaro morto monty python) fará o mesmo esquete, “Dead Parrot”, ser apresentado, apesar de a página propriamente dita não conter nenhuma menção às palavras “deceased” (morto) ou “bird” (pássaro). O Google sabe que um “hot dog” (cachorro-quente) é um alimento e que “boiling puppy” (cachorrinho agitado) é algo totalmente diferente. Como? É tudo uma questão de estatística!

Embora você ache que análise de texto não tenha nada a ver com seu projeto, entender seus conceitos básicos pode ser extremamente útil para todo tipo de aprendizado de máquina (machine learning), assim como para a capacidade genérica de modelar problemas do mundo real em termos de probabilidade e de algoritmos.

Por exemplo, o serviço de música Shazam é capaz de identificar um áudio contendo determinada gravação musical, mesmo que esse áudio contenha som ambiente ou alguma distorção. O Google está trabalhando em legendar imagens de modo automático com base apenas na própria imagem<sup>2</sup>. Ao comparar imagens conhecidas de, por exemplo, cachorros-

quentes, com outras imagens de cachorros-quentes, a ferramenta de pesquisa pode gradualmente aprender como é a aparência de um cachorro- quente e observar esses padrões em outras imagens que lhe forem mostradas.

## Resumindo dados

No *Capítulo 8*, vimos como separar um conteúdo de texto em n-gramas, isto é, conjuntos de expressões com  $n$  palavras. No nível básico, isso pode ser usado para determinar quais conjuntos de palavras e expressões tendem a ser mais comuns em uma seção de texto. Além disso, o código pode ser usado para criar resumos de dados que pareçam naturais se retornarmos ao texto original e extrairmos sentenças que acompanham essas expressões mais populares.

Um texto que usaremos como exemplo para isso é o discurso de posse do nono presidente dos Estados Unidos, William Henry Harrison. O governo de Harrison estabeleceu dois recordes na história da Presidência: um para o discurso de posse mais longo e outro para o período mais breve na presidência – 32 dias.

Usaremos o texto completo de seu *discurso* (<http://pythonscraping.com/files/inaugurationSpeech.txt>) como fonte de dados para muitos dos códigos de exemplo neste capítulo.

Modificando um pouco o código usado para encontrar n-gramas no *Capítulo 8*, podemos gerar um código que procure conjuntos de bigramas e devolva um objeto `counter` com todos eles:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import string
from collections import Counter

def cleanSentence(sentence):
    sentence = sentence.split(' ')
    sentence = [word.strip(string.punctuation+string.whitespace)
                for word in sentence]
    sentence = [word for word in sentence if len(word) > 1
                or (word.lower() == 'a' or word.lower() == 'i')]
    return sentence

def cleanInput(content):
    content = content.upper()
```

```

content = re.sub('\n', ' ', content)
content = bytes(content, "UTF-8")
content = content.decode("ascii", "ignore")
sentences = content.split('. ')
return [cleanSentence(sentence) for sentence in sentences]

def getNgramsFromSentence(content, n):
    output = []
    for i in range(len(content)-n+1):
        output.append(content[i:i+n])
    return output

def getNgrams(content, n):
    content = cleanInput(content)
    ngrams = Counter()
    ngrams_list = []
    for sentence in content:
        newNgrams = [' '.join(gram) for gram in
            getNgramsFromSentence(sentence, 2)]
        ngrams_list.extend(newNgrams)
        ngrams.update(newNgrams)
    return(ngrams)

content = str(
    urlopen('http://pythonscraping.com/files/inaugurationSpeech.txt')
    .read(), 'utf-8')
ngrams = getNgrams(content, 2)
print(ngrams)

```

Eis uma parte da saída gerada:

```

Counter({'OF THE': 213, 'IN THE': 65, 'TO THE': 61, 'BY THE': 41,
'THE CONSTITUTION': 34, 'OF OUR': 29, 'TO BE': 26, 'THE PEOPLE': 24,
'FROM THE': 24, 'THAT THE': 23,...

```

Entre esses bigramas, “the constitution” parece um assunto razoavelmente comum no discurso, mas “of the”, “in the” e “to the” não parecem ter especial relevância. Como podemos nos livrar de palavras indesejadas de modo preciso e automático?

Felizmente, há pessoas por aí que estudam com afinco as diferenças entre palavras “interessantes” e “não interessantes”, e o trabalho delas pode nos ajudar a fazer exatamente isso. Mark Davies, um professor de linguística da Brigham Young University, mantém o [Corpus of Contemporary American English](http://corpus.byu.edu/coca/) (*Corpus* do inglês americano contemporâneo, <http://corpus.byu.edu/coca/>): uma coleção com mais de 450 milhões de palavras de publicações norte-americanas conhecidas, aproximadamente da última década.

A lista das 5 mil palavras encontradas com mais frequência está disponível de forma gratuita e, felizmente, ela é muito mais que suficiente como um filtro básico para eliminar os bigramas mais comuns. Somente com as 100 primeiras palavras, junto com o acréscimo de uma função `isCommon`, melhoramos bastante o resultado:

```
def isCommon(ngram):
    commonWords = ['THE', 'BE', 'AND', 'OF', 'A', 'IN', 'TO', 'HAVE', 'IT', 'I',
        'THAT', 'FOR', 'YOU', 'HE', 'WITH', 'ON', 'DO', 'SAY', 'THIS', 'THEY',
        'IS', 'AN', 'AT', 'BUT', 'WE', 'HIS', 'FROM', 'THAT', 'NOT', 'BY',
        'SHE', 'OR', 'AS', 'WHAT', 'GO', 'THEIR', 'CAN', 'WHO', 'GET', 'IF',
        'WOULD', 'HER', 'ALL', 'MY', 'MAKE', 'ABOUT', 'KNOW', 'WILL', 'AS',
        'UP', 'ONE', 'TIME', 'HAS', 'BEEN', 'THERE', 'YEAR', 'SO', 'THINK',
        'WHEN', 'WHICH', 'THEM', 'SOME', 'ME', 'PEOPLE', 'TAKE', 'OUT', 'INTO',
        'JUST', 'SEE', 'HIM', 'YOUR', 'COME', 'COULD', 'NOW', 'THAN', 'LIKE',
        'OTHER', 'HOW', 'THEN', 'ITS', 'OUR', 'TWO', 'MORE', 'THESE', 'WANT',
        'WAY', 'LOOK', 'FIRST', 'ALSO', 'NEW', 'BECAUSE', 'DAY', 'MORE', 'USE',
        'NO', 'MAN', 'FIND', 'HERE', 'THING', 'GIVE', 'MANY', 'WELL']
    for word in ngram:
        if word in commonWords:
            return True
    return False
```

Com isso, geramos os bigramas a seguir, encontrados mais de duas vezes no corpo do texto:

```
Counter({'UNITED STATES': 10, 'EXECUTIVE DEPARTMENT': 4,
    'GENERAL GOVERNMENT': 4, 'CALLED UPON': 3, 'CHIEF MAGISTRATE': 3,
    'LEGISLATIVE BODY': 3, 'SAME CAUSES': 3, 'GOVERNMENT SHOULD': 3,
    'WHOLE COUNTRY': 3, ...
```

De modo apropriado, os dois primeiros itens da lista são “United States” (Estados Unidos) e “executive department” (Poder Executivo), que seriam esperados no discurso de posse de um presidente.

É importante observar que estamos usando uma lista de palavras comuns de uma era relativamente moderna para filtrar o resultado, o que talvez não seja apropriado, considerando que o texto foi escrito em 1841. No entanto, como estamos usando aproximadamente apenas as 100 primeiras palavras da lista – podemos supor que elas são mais estáveis no tempo do que, por exemplo, as últimas 100 palavras – e parece que os resultados são satisfatórios, é provável que possamos evitar o esforço de identificar ou criar uma lista das palavras mais comuns em 1841 (ainda que um esforço como esse fosse interessante).

Agora que alguns tópicos essenciais foram extraídos do texto, como isso nos ajudaria a escrever um resumo desse texto? Uma maneira é procurar a

primeira sentença que contenha cada n-grama “popular”, com base na teoria de que a primeira ocorrência produzirá uma visão geral satisfatória do conteúdo. Os primeiros cinco bigramas mais populares resultam nas sentenças listadas a seguir:

- *The Constitution of the United States is the instrument containing this grant of power to the several departments composing the Government.* (A Constituição dos Estados Unidos é o instrumento que contém essa concessão de poder aos vários departamentos que compõem o Governo.)
- *Such a one was afforded by the executive department constituted by the Constitution.* (Isso foi possibilitado pelo Poder Executivo estabelecido pela Constituição.)
- *The General Government has seized upon none of the reserved rights of the States.* (O governo federal não coibiu nenhum dos direitos reservados aos estados.)
- *Called from a retirement which I had supposed was to continue for the residue of my life to fill the chief executive office of this great and free nation, I appear before you, fellow-citizens, to take the oaths which the constitution prescribes as a necessary qualification for the performance of its duties; and in obedience to a custom coeval with our government and what I believe to be your expectations I proceed to present to you a summary of the principles which will govern me in the discharge of the duties which I shall be called upon to perform.* (Chamado a trabalhar enquanto desfrutava uma aposentadoria que supunha perdurar pelo resto de minha vida para ocupar a Presidência desta grande e livre nação, eu me apresento diante de vocês, caros cidadãos, para fazer os juramentos prescritos pela Constituição como uma qualificação necessária para cumprir o que ela determina; em obediência a uma tradição coetânea de nosso governo e àquilo que acredito ser as vossas expectativas, prossigo apresentando uma síntese dos princípios que governarão a mim no cumprimento das obrigações a que me caberão.)
- *The presses in the necessary employment of the Government should never be used to “clear the guilty or to varnish crime”.* (A imprensa no emprego necessário do Governo jamais deve ser usada para “exonerar a culpa ou encobrir um crime”.)

Claro que isso não será publicado no CliffsNotes tão cedo, mas, considerando que o documento original continha 217 sentenças, e que a quarta sentença (“Called from a retirement...”) sintetiza de modo bem razoável o tópico principal, não foi ruim para uma primeira execução.

Com blocos de texto mais longos ou mais variados, talvez valha a pena procurar trigramas ou até mesmo 4-gramas para obter as sentenças “mais importantes” de uma passagem. Nesse exemplo, apenas um trigrama é usado várias vezes: “exclusive metallic currency” (moeda exclusivamente metálica) – dificilmente seria uma expressão definidora de um discurso de posse presidencial. Em passagens mais longas, usar trigramas poderia ser apropriado.

Outra abordagem é procurar sentenças que contenham os n-gramas mais populares. É claro que eles tenderão a ser sentenças mais longas, portanto, se isso se tornar um problema, você poderá procurar sentenças com os maiores percentuais de palavras que sejam n-gramas populares, ou criar uma métrica própria de pontuação, combinando diversas técnicas.

## Modelos de Markov

Talvez você já tenha ouvido falar dos geradores de texto de Markov. Eles se tornaram populares para entretenimento, como no aplicativo [“That can be my next tweet!”](http://yes.thatcan.be/my/next/tweet/) (<http://yes.thatcan.be/my/next/tweet/>), assim como pelo seu uso para gerar emails spam que pareçam reais a fim de enganar sistemas de detecção.

Todos esses geradores de texto são baseados no modelo de Markov, muitas vezes usado para analisar grandes conjuntos de eventos aleatórios, em que um evento discreto é seguido de outro evento discreto com determinada probabilidade.

Por exemplo, poderíamos construir um modelo de Markov de um sistema de previsão do tempo, conforme mostra a Figura 9.1.

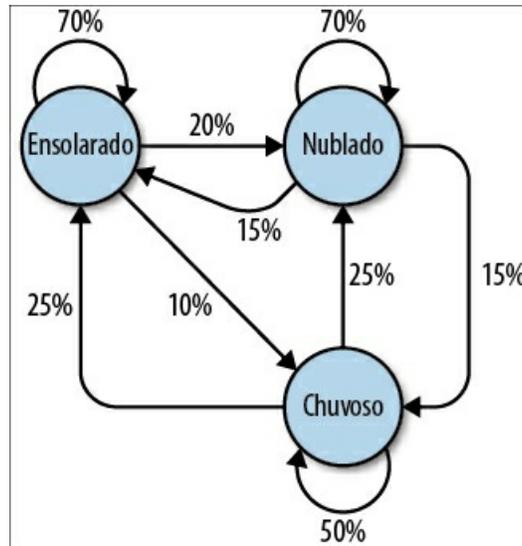


Figura 9.1 – Modelo de Markov que descreve um sistema teórico de previsão do tempo.

Nesse modelo, para cada dia ensolarado, há 70% de chances de o dia seguinte também ser ensolarado, com 20% de chances de ser nublado e apenas 10% de chances de chuva. Se o dia for chuvoso, há 50% de chances de chover no dia seguinte, 25% de chances de estar ensolarado e 25% de chances de estar nublado.

Podemos notar diversas propriedades nesse modelo de Markov:

- Todos os percentuais que saem de um único nó devem somar exatamente 100%. Não importa quão complicado é o sistema, sempre deve haver uma chance de 100% de ele ser levado a outro ponto no próximo passo.
- Embora haja apenas três possibilidades para o clima em qualquer dado instante, esse modelo pode ser usado para gerar uma lista infinita de estados meteorológicos.
- Somente o estado do nó atual em que você estiver influenciará o estado no qual você estará a seguir. Se você estiver no nó Ensolarado, não importa se os 100 dias anteriores tenham sido ensolarados ou chuvosos – as chances de ter sol no próximo dia serão exatamente as mesmas: 70%.
- Pode ser mais difícil alcançar alguns nós do que outros. A matemática por trás disso é um tanto quanto complicada, mas deve ser razoavelmente fácil ver que Chuvoso (com menos de “100%” de setas que apontam para ele) é um estado muito menos provável de ser

alcançado nesse sistema, em qualquer instante, do que Ensolarado ou Nublado.

É evidente que esse é um sistema simples, e os modelos de Markov podem se tornar arbitrariamente maiores. O algoritmo de classificação de páginas do Google é, em parte, baseado em um modelo de Markov, com sites representados como nós e links de entrada/saída representados como conexões entre os nós. A “probabilidade” de chegar em um nó em particular representa a relativa popularidade do site. Isso significa que, se nosso sistema de previsão do tempo representasse uma internet extremamente pequena, “chuvoso” seria uma página com uma posição mais baixa na classificação, enquanto “nublado” estaria em uma posição mais alta.

Com tudo isso em mente, vamos considerar um exemplo mais concreto: analisar e escrever texto.

Mais uma vez, usando o discurso de posse de William Henry Harrison analisado no exemplo anterior, podemos escrever o código a seguir, que gera cadeias de Markov arbitrariamente longas (com o tamanho da cadeia definida com 100) com base na estrutura do texto:

```
from urllib.request import urlopen
from random import randint

def wordListSum(wordList):
    sum = 0
    for word, value in wordList.items():
        sum += value
    return sum

def retrieveRandomWord(wordList):
    randIndex = randint(1, wordListSum(wordList))
    for word, value in wordList.items():
        randIndex -= value
        if randIndex <= 0:
            return word

def buildWordDict(text):
    # Remove quebras de linha e aspas
    text = text.replace('\n', ' ');
    text = text.replace('"', '');

    # Garante que sinais de pontuação sejam tratados como "palavras" próprias,
    # de modo que sejam incluídos na cadeia de Markov
    punctuation = [',', '.', ';', ':']
    for symbol in punctuation:
        text = text.replace(symbol, ' {} '.format(symbol));
```

```

words = text.split(' ')
# Filtra palavras vazias
words = [word for word in words if word != '']

wordDict = {}
for i in range(1, len(words)):
    if words[i-1] not in wordDict:
        # Cria um novo dicionário para essa palavra
        wordDict[words[i-1]] = {}
    if words[i] not in wordDict[words[i-1]]:
        wordDict[words[i-1]][words[i]] = 0
    wordDict[words[i-1]][words[i]] += 1
return wordDict

text = str(urlopen('http://pythonscraping.com/files/inaugurationSpeech.txt')
          .read(), 'utf-8')
wordDict = buildWordDict(text)

# Gera uma cadeia de Markov de tamanho 100
length = 100
chain = ['I']
for i in range(0, length):
    newWord = retrieveRandomWord(wordDict[chain[-1]])
    chain.append(newWord)

print(' '.join(chain))

```

A saída muda sempre que esse código é executado, mas eis um exemplo do misterioso texto sem sentido que ele gera:

```

I sincerely believe in Chief Magistrate to make all necessary sacrifices and
oppression of the remedies which we may have occurred to me in the arrangement
and disbursement of the democratic claims them , consolatory to have been best
political power in fervently commending every other addition of legislation , by
the interests which violate that the Government would compare our aboriginal
neighbors the people to its accomplishment . The latter also susceptible of the
Constitution not much mischief , disputes have left to betray . The maxim which
may sometimes be an impartial and to prevent the adoption or

```

O que está acontecendo no código?

A função `buildWordDict` recebe a string de texto, obtida da internet. Ela então faz algumas limpezas e um pouco de formatação, removendo aspas e inserindo espaços em torno de outros sinais de pontuação para que sejam efetivamente tratados como uma palavra separada. Depois disso, um dicionário bidimensional é construído – um dicionário de dicionários – com o seguinte formato:

```

{word_a : {word_b : 2, word_c : 1, word_d : 1},
 word_e : {word_b : 5, word_d : 2},...}

```

Nesse dicionário de exemplo, “word\_a” foi encontrada quatro vezes, das

quais duas ocorrências foram seguidas da palavra “word\_b”, uma ocorrência seguida de “word\_c” e outra seguida de “word\_d”. “Word\_e” foi seguida sete vezes, cinco vezes por “word\_b” e duas por “word\_d”.

Se desenhassemos um modelo de nós desse resultado, o nó representando word\_a teria uma seta de 50% apontando para “word\_b” (que a seguiu duas de quatro vezes), uma seta de 25% apontando para “word\_c” e 25% apontando para “word\_d”.

Depois de construído, esse dicionário pode ser usado como uma tabela de consulta para saber aonde ir depois, independentemente da palavra do texto em que você por acaso estiver<sup>3</sup>. Usando o exemplo do dicionário de dicionários, você poderia estar em “word\_e” no momento, o que significa que o dicionário {word\_b : 5, word\_d: 2} seria passado para a função `retrieveRandomWord`. Essa função, por sua vez, obtém uma palavra aleatória do dicionário, levando em consideração o número de vezes que ela ocorre.

Ao começar com uma palavra inicial aleatória (nesse caso, a palavra “I” que está em toda parte), podemos percorrer a cadeia de Markov com facilidade, gerando quantas palavras quisermos.

Essas cadeias de Markov tendem a melhorar no que concerne ao seu “realismo” quanto mais textos forem coletados, sobretudo de fontes com estilos semelhantes de escrita. Embora esse exemplo tenha usado bigramas para criar a cadeia (em que a palavra anterior prevê a próxima palavra), trigramas ou n-gramas de mais alta ordem podem ser utilizados, nos quais duas ou mais palavras preveem a próxima palavra.

Apesar de entreter e ser um bom uso para os megabytes de texto que você tenha acumulado durante seu web scraping, aplicações como essas podem dificultar ver o lado prático das cadeias de Markov. Conforme mencionamos antes nesta seção, as cadeias de Markov modelam a ligação entre os sites, de uma página para a próxima. Grandes coleções desses links como ponteiros podem compor grafos em forma de teias, úteis para serem armazenados, percorridos e analisados. Nesse sentido, as cadeias de Markov compõem a base tanto para pensar sobre web crawling quanto para saber como seus web crawlers podem pensar.

## **Six Degrees of Wikipedia: conclusão**

No *Capítulo 3*, criamos um scraper que coleta links de um artigo da Wikipédia para o próximo, começando com o artigo sobre Kevin Bacon e,

no *Capítulo 6*, armazenamos esses links em um banco de dados. Por que estou trazendo esse assunto de volta? Porque o fato é que o problema de escolher um caminho de links que comece em uma página e termine na página desejada (isto é, encontrar uma cadeia de páginas entre [https://en.wikipedia.org/wiki/Kevin\\_Bacon](https://en.wikipedia.org/wiki/Kevin_Bacon) e [https://en.wikipedia.org/wiki/Eric\\_Idle](https://en.wikipedia.org/wiki/Eric_Idle)) é o mesmo que encontrar uma cadeia de Markov em que tanto a primeira quanto a última palavra estão definidas.

Esses tipos de problemas são problemas de *grafos direcionados*, em que  $A \rightarrow B$  não significa necessariamente  $B \rightarrow A$ . A palavra “seleção” pode vir muitas vezes seguida da palavra “brasileira”, mas você verá que a palavra “brasileira” é seguida com muito menos frequência da palavra “seleção”. Embora o artigo da Wikipédia sobre Kevin Bacon tenha links para o artigo sobre a cidade natal do ator, Filadélfia, o artigo sobre Filadélfia não é recíproco, não apresentando ligações de volta a ele.

Em oposição, o jogo original Six Degrees of Kevin Bacon é um problema de *grafo não direcionado*. Se Kevin Bacon atuou em *Linha mortal* (Flatliners) com Julia Roberts, então Julia Roberts necessariamente atuou em *Linha mortal* com Kevin Bacon, portanto o relacionamento é bidirecional (não tem “direção”). Problemas de grafos não direcionados tendem a ser menos comuns em ciência da computação do que problemas de grafos direcionados, e ambos são difíceis de resolver do ponto de vista computacional.

Embora muito trabalho tenha sido dedicado a esses tipos de problemas e à multiplicidade de suas variantes, uma das maneiras mais apropriadas e comuns para encontrar os caminhos mais curtos em um grafo direcionado – e, desse modo, encontrar caminhos entre o artigo da Wikipédia sobre Kevin Bacon e todos os outros artigos da Wikipédia – é fazer uma busca em largura (breadth-first search).

Uma *busca em largura* é efetuada procurando primeiro todos os links associados de forma direta à página inicial. Se esses links não contiverem a página visada (a página que você está procurando), um segundo nível de links – páginas ligadas por uma página ligada pela página inicial – será pesquisado. Esse processo continua até que o limite de profundidade (seis, nesse caso) seja alcançado ou a página visada seja encontrada.

Apresentamos a seguir uma solução completa para a busca em largura,

usando uma tabela de links conforme descrito no *Capítulo 6*:

```
import pymysql

conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
    user='', passwd='', db='mysql', charset='utf8')
cur = conn.cursor()
cur.execute('USE wikipedia')

def getUrl(pageId):
    cur.execute('SELECT url FROM pages WHERE id = %s', (int(pageId)))
    return cur.fetchone()[0]

def getLinks(fromPageId):
    cur.execute('SELECT toPageId FROM links WHERE fromPageId = %s',
        (int(fromPageId)))
    if cur.rowcount == 0:
        return []
    return [x[0] for x in cur.fetchall()]

def searchBreadth(targetPageId, paths=[[1]]):
    newPaths = []
    for path in paths:
        links = getLinks(path[-1])
        for link in links:
            if link == targetPageId:
                return path + [link]
            else:
                newPaths.append(path+[link])
    return searchBreadth(targetPageId, newPaths)

nodes = getLinks(1)
targetPageId = 28624
pageIds = searchBreadth(targetPageId)
for pageId in pageIds:
    print(getUrl(pageId))
```

`getUrl` é uma função auxiliar que obtém URLs do banco de dados, dado um ID de página. De modo semelhante, `getLinks` recebe um `fromPageId` que representa o ID inteiro da página atual e busca uma lista com todos os IDs inteiros das páginas às quais ela está ligada.

A função principal, `searchBreadth`, funciona de modo recursivo, construindo uma lista de todos os caminhos possíveis a partir da página pesquisada, e parando quando encontra um caminho que tenha alcançado a página desejada:

- A função começa com um caminho único, `[1]`, que representa um caminho no qual o usuário está na página-alvo com ID 1 (Kevin Bacon) e não segue nenhum link.

- Para cada caminho na lista de caminhos (na primeira passagem, há apenas um caminho, portanto, esse passo é rápido), todos os links referenciados pela última página do caminho são obtidos.
- Para cada um desses links de saída, o código verifica se ele corresponde a `targetPageId`. Se corresponder, esse caminho será devolvido.
- Se não houver correspondência, um novo caminho é adicionado em uma nova lista de caminhos (agora mais longa), constituída do caminho anterior + o novo link de saída da página.
- Se `targetPageId` não for encontrada nesse nível, haverá uma recursão e `searchBreadth` será chamada com o mesmo `targetPageId` e uma nova lista mais longa de caminhos.

Depois que a lista de IDs de página contendo um caminho entre as duas páginas for encontrada, cada ID é resolvido para o seu URL e será exibido.

Eis a saída da procura de um link entre a página de Kevin Bacon (ID de página 1 nesse banco de dados) e a página de Eric Idle (ID de página 28624 nesse banco de dados):

```
/wiki/Kevin_Bacon
/wiki/Primetime_Emy_Award_for_Outstanding_Lead_Actor_in_a_Miniseries_or_a_Movie
/wiki/Gary_Gilmore
/wiki/Eric_Idle
```

Isso se traduz neste relacionamento de links: Kevin Bacon → Primetime Emmy Award → Gary Gilmore → Eric Idle.

Além de resolver os problemas de Six Degree e modelar quais palavras tendem a seguir outras palavras nas sentenças, grafos direcionados e não direcionados podem ser usados para modelar diversas situações encontradas em web scraping. Quais sites estão ligados a quais outros sites? Quais artigos de pesquisa citam quais outros artigos de pesquisa? Quais produtos tendem a ser exibidos com quais outros produtos em um site de vendas? Qual é a força desse link? É um link recíproco?

Reconhecer esses tipos fundamentais de relacionamentos pode ser extremamente útil para criar modelos e fazer visualizações ou previsões com base nos dados coletados.

## Natural Language Toolkit

Até agora, este capítulo teve como foco essencialmente a análise estatística

de palavras em textos. Quais são as palavras mais populares? Quais palavras são incomuns? Quais palavras têm mais chances de estar presentes depois de quais outras palavras? Como elas são agrupadas? O que falta agora, na medida do possível, é entender o que as palavras representam.

O *NLTK* (Natural Language Toolkit) é um pacote de bibliotecas Python projetado para identificar e atribuir tags a partes do discurso encontradas em textos em língua inglesa. Seu desenvolvimento teve início em 2000, e, nos últimos 15 anos, dezenas de desenvolvedores em todo o mundo contribuíram com o projeto. Embora as funcionalidades que ele oferece sejam incríveis (livros inteiros foram dedicados ao NLTK), esta seção se concentra em apenas alguns de seus usos.

## Instalação e configuração

O módulo `nltk` pode ser instalado do mesmo modo que outros módulos Python, seja fazendo download do pacote direto do site do NLTK ou usando qualquer um dos instaladores de terceiros com a palavra-chave “`nltk`”. Para ver instruções completas sobre a instalação, consulte o [site do NLTK](http://www.nltk.org/install.html) (<http://www.nltk.org/install.html>).

Depois de instalar o módulo, é uma boa ideia fazer download de seus repositórios de texto predefinidos para que você teste as funcionalidades de modo mais fácil. Digite o seguinte na linha de comando Python:

```
>>> import nltk
>>> nltk.download()
```

O NLTK Downloader (Figura 9.2) será iniciado.

Recomendo instalar todos os pacotes disponíveis ao testar o *corpus* do NLTK pela primeira vez. Você pode desinstalar facilmente os pacotes a qualquer momento.

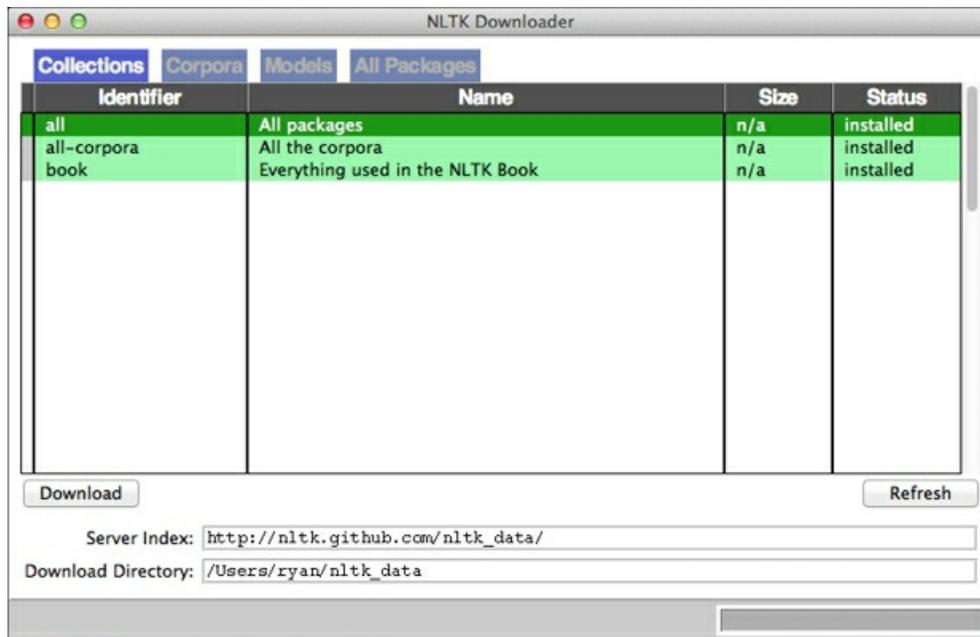


Figura 9.2 – O NLTK Downloader permite navegar pelos pacotes opcionais e bibliotecas de texto associadas ao módulo `nltk` e fazer o seu download.

## Análise estatística com o NLTK

O NLTK é ótimo para gerar informações estatísticas sobre contadores, frequência e diversidade de palavras em seções de texto. Se tudo de que você precisa é um cálculo relativamente simples (por exemplo, o número de palavras únicas usadas em uma seção de texto), importar o `nltk` talvez seja um exagero – é um módulo grande. No entanto, se for necessário fazer uma análise relativamente ampla de um texto, você terá funções ao alcance das mãos, as quais calcularão praticamente qualquer métrica desejada.

A análise no NLTK sempre começa com o objeto `Text`. Objetos `Text` podem ser criados a partir de strings Python simples, assim:

```
from nltk import word_tokenize
from nltk import Text

tokens = word_tokenize('Here is some not very interesting text')
text = Text(tokens)
```

A entrada da função `word_tokenize` pode ser qualquer string de texto Python. Se você não tiver strings longas disponíveis, mas quer brincar com as funcionalidades, o NLTK tem alguns livros já incluídos na biblioteca, que podem ser acessados com a função `import`:

```
from nltk.book import *
```

Esse comando carrega os nove livros:

```

*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908

```

Trabalharemos com `text6`, “Monty Python and the Holy Grail” (Monty Python – Em busca do cálice sagrado; é o roteiro do filme de 1975), em todos os exemplos a seguir.

Objetos texto podem ser manipulados de modo muito semelhante a arrays Python, como se fossem um array contendo as palavras do texto. Usando essa propriedade, podemos contar o número de palavras únicas em um texto e compará-lo com o número total de palavras (lembre-se de que um `set` Python armazena somente valores únicos):

```

>>> len(text6)/len(set(text6))
7.833333333333333

```

O resultado anterior mostra que cada palavra do roteiro foi usada aproximadamente oito vezes. Também podemos colocar o texto em um objeto de distribuição de frequências a fim de determinar algumas das palavras mais comuns e as frequências de diversas palavras:

```

>>> from nltk import FreqDist
>>> fdist = FreqDist(text6)
>>> fdist.most_common(10)
[(':', 1197), ('.', 816), ('!', 801), (',', 731), ('"', 421), ('[', 319), (']', 312), ('the', 299), ('I', 255), ('ARTHUR', 225)]
>>> fdist["Grail"]
34

```

Por ser um roteiro, alguns detalhes sobre o modo como está escrito podem se destacar. Por exemplo, “ARTHUR” com letras maiúsculas surge com frequência porque aparece antes de cada uma das falas de King Arthur (Rei Artur) no roteiro. Além disso, dois-pontos (:) aparecem antes de cada fala, atuando como um separador entre o nome da personagem e a sua fala. Usando esse fato, podemos ver que há 1.197 falas no filme!

O NLTK usa o termo *bigrama* (ocasionalmente, o termo bigrama pode ser

visto como 2-grama, e o termo trigrama, como 3-grama). Podemos criar, procurar e listar bigramas de modo muito fácil:

```
>>> from nltk import bigrams
>>> bigrams = bigrams(text6)
>>> bigramsDist = FreqDist(bigrams)
>>> bigramsDist[('Sir', 'Robin')]
18
```

Para procurar os bigramas “Sir Robin”, é necessário separá-lo na tupla (“Sir”, “Robin”), para que coincida com o modo como os bigramas são representados na distribuição de frequência. Há também um módulo `trigrams` que funciona exatamente do mesmo modo. Para o caso geral, também podemos importar o módulo `ngrams`:

```
>>> from nltk import ngrams
>>> fourgrams = ngrams(text6, 4)
>>> fourgramsDist = FreqDist(fourgrams)
>>> fourgramsDist[('father', 'smelt', 'of', 'elderberries')]
1
```

Nesse caso, a função `ngrams` é chamada para separar um objeto texto em n-gramas de qualquer tamanho, definido conforme o segundo parâmetro. No exemplo, estamos separando o texto em 4-gramas. Em seguida, podemos demonstrar que a expressão “father smelt of elderberries” (pai cheirava a sabugueiro) ocorre exatamente uma vez no roteiro.

Também é possível iterar e atuar sobre distribuições de frequência, objetos texto e n-gramas em um laço. O código a seguir exhibe todos os 4-gramas que começam com a palavra “coconut” (coco), por exemplo:

```
from nltk.book import *
from nltk import ngrams
fourgrams = ngrams(text6, 4)
for fourgram in fourgrams:
    if fourgram[0] == 'coconut':
        print(fourgram)
```

A biblioteca NLTK tem diversas ferramentas e objetos projetados para organizar, contar, ordenar e avaliar porções grandes de texto. Embora mal tenhamos tocado a superfície quanto a seus usos, a maioria dessas ferramentas tem um bom design e funciona de modo bem intuitivo para alguém que tenha familiaridade com Python.

## **Análise lexicográfica com o NLTK**

Até agora, comparamos e classificamos todas as palavras que encontramos com base apenas no valor que elas representam por si só. Não há nenhuma

diferenciação entre homônimos ou pelo contexto em que as palavras são usadas.

Embora algumas pessoas se sintam tentadas a menosprezar os homônimos considerando-os raramente problemáticos, você poderia se surpreender com a frequência com que eles ocorrem. É provável que a maioria dos falantes nativos de inglês em geral não perceba que uma palavra é um homônimo, muito menos considere que ela poderia ser confundida com outra palavra em um contexto diferente.

“He was objective in achieving his objective of writing an objective philosophy, primarily using verbs in the objective case” (Ele era objetivo em alcançar seu objetivo de escrever uma filosofia objetiva, usando essencialmente verbos com pronomes no caso oblíquo) é fácil de ser interpretada por seres humanos, mas um web scraper pode achar que a mesma palavra está sendo usada quatro vezes e descartar todas as informações sobre o significado de cada palavra.

Além de compreender as partes do discurso, ser capaz de distinguir uma palavra usada de uma forma em oposição a outra pode ser conveniente. Por exemplo, poderíamos procurar nomes de empresa compostos de palavras comuns em inglês ou analisar as opiniões de alguém sobre uma empresa. “ACME Products é boa” e “ACME Products não é ruim” podem ter o mesmo significado básico, ainda que uma sentença use “boa” e a outra use “ruim.”

### Tags do Penn Treebank

Por padrão, o NLTK usa um sistema conhecido para marcação de partes do discurso, desenvolvido pelo [Projeto Penn Treebank](https://catalog.ldc.upenn.edu/ldc99t42) (<https://catalog.ldc.upenn.edu/ldc99t42>) da Universidade da Pensilvânia. Embora algumas tags façam sentido (por exemplo, CC é uma conjunção coordenativa), outras podem ser confusas (por exemplo, RP é uma partícula). Use a tabela a seguir como uma referência para as tags mencionadas nesta seção:

CC	Conjunção coordenativa
CD	Número cardinal
DT	Determinante
EX	“There” (haver) existencial
FW	Palavra estrangeira
IN	Preposição, conjunção subordinativa
JJ	Adjetivo
JJR	Adjetivo, comparativo
JJS	Adjetivo, superlativo
LS	Marcador de item de lista

MD	Modal
NN	Substantivo, singular ou incontável
NNS	Substantivo, plural
NNP	Substantivo próprio, singular
NNPS	Substantivo próprio, plural
PDT	Pré-determinante
POS	Terminação de possessivo
PRP	Pronome pessoal
PRP\$	Pronome possessivo
RB	Advérbio
RBR	Advérbio, comparativo
RBS	Advérbio, superlativo
RP	Partícula
SYM	Símbolo
TO	“to” (para)
UH	Interjeição
VB	Verbo, forma básica
VBD	Verbo, passado
VBG	Verbo, gerúndio ou particípio presente
VBN	Verbo, particípio passado
VBP	Verbo, diferente de terceira pessoa do singular, presente
VBZ	Verbo, terceira pessoa do singular, presente
WDT	Determinante iniciado com wh
WP	Pronome iniciado com wh
WP\$	Pronome possessivo iniciado com wh
WRB	Advérbio iniciado com wh

Além de avaliar a linguagem, o NLTK oferece assistência para encontrar o significado das palavras com base no contexto e nos próprios dicionários de tamanhos consideráveis. No nível básico, o NLTK é capaz de identificar partes do discurso:

```
>>> from nltk.book import *
>>> from nltk import word_tokenize
>>> text = word_tokenize('Strange women lying in ponds distributing swords'\
'is no basis for a system of government.')
>>> from nltk import pos_tag
>>> pos_tag(text)
[('Strange', 'NNP'), ('women', 'NNS'), ('lying', 'VBG'), ('in', 'IN')
, ('ponds', 'NNS'), ('distributing', 'VBG'), ('swords', 'NNS'), ('is'
```

```
, 'VBZ'), ('no', 'DT'), ('basis', 'NN'), ('for', 'IN'), ('a', 'DT'),  
('system', 'NN'), ('of', 'IN'), ('government', 'NN'), ('.', '.')] ]
```

Cada palavra está separada em uma *tupla* contendo a palavra e uma tag que identifica a parte do discurso (veja a caixa de texto anterior para obter mais informações sobre essas tags). Embora pareça uma consulta simples, a complexidade exigida para realizar a tarefa de forma correta torna-se evidente no exemplo a seguir:

```
>>> text = word_tokenize('The dust was thick so he had to dust')  
>>> pos_tag(text)  
[('The', 'DT'), ('dust', 'NN'), ('was', 'VBD'), ('thick', 'JJ'), ('so', 'RB'),  
 ('he', 'PRP'), ('had', 'VBD'), ('to', 'TO'), ('dust', 'VB')]
```

Observe que a palavra “dust” é usada duas vezes na sentença: uma como substantivo (poeira) e outra como verbo (remover a poeira). O NLTK identifica os dois usos de modo correto, com base em seu contexto na sentença. Ele identifica partes do discurso usando uma gramática livre de contexto, definida pela língua inglesa. *Gramáticas livres de contexto* são conjuntos de regras que definem quais elementos podem vir depois de quais outros elementos em listas ordenadas. Nesse caso, elas definem quais partes do discurso podem vir depois de quais outras partes do discurso. Sempre que uma palavra ambígua como “dust” é encontrada, as regras da gramática livre de contexto são consultadas, e uma parte do discurso apropriada, que siga as regras, é selecionada.

## Machine learning e machine training

Podemos fazer o NLTK gerar gramáticas livres de contexto totalmente novas, quando fizermos o seu treinamento, por exemplo, com uma língua estrangeira. Se você aplicar tags em porções grandes de texto manualmente na língua, usando as tags apropriadas do Penn Treebank, é possível alimentar o NLTK e treiná-lo para que atribua tags de forma adequada a outros textos que ele encontrar. Esse tipo de treinamento é um componente necessário para qualquer atividade de aprendizado de máquina (machine learning); veremos esse assunto novamente no *Capítulo 14*, quando treinaremos scrapers para reconhecer caracteres CAPTCHA.

Qual é o sentido de saber se uma palavra é um verbo ou um substantivo em um dado contexto? Pode parecer interessante em um laboratório de pesquisa de ciência da computação, mas como isso ajuda no web scraping? Um problema comum em web scraping tem a ver com pesquisa. Podemos coletar textos de um site e querer procurar aí as ocorrências da palavra “google”, mas apenas quando ela for usada como verbo, e não como um substantivo próprio. Ou talvez estejamos procurando somente as ocorrências da empresa Google, e não queremos depender de as pessoas usarem a letra maiúscula de forma correta para encontrar essas ocorrências. Nesse caso, a função `pos_tag` pode ser muito útil:

```

from nltk import word_tokenize, sent_tokenize, pos_tag
sentences = sent_tokenize('Google is one of the best companies in the world.'\
' I constantly google myself to see what I\'m up to.')
nouns = ['NN', 'NNS', 'NNP', 'NNPS']

for sentence in sentences:
    if 'google' in sentence.lower():
        taggedWords = pos_tag(word_tokenize(sentence))
        for word in taggedWords:
            if word[0].lower() == 'google' and word[1] in nouns:
                print(sentence)

```

Esse código exibe apenas as sentenças que contêm a palavra “google” (ou “Google”) como algum tipo de substantivo, não como um verbo. É claro que poderíamos ser mais específicos e exigir que somente ocorrências de Google marcadas com “NNP” (um substantivo próprio) sejam exibidas, mas até mesmo o NLTK comete erros ocasionalmente, e pode ser bom ter um pouco de jogo de cintura, conforme a aplicação.

Boa parte da ambiguidade das línguas naturais pode ser resolvida com a função `pos_tag` do NLTK. Ao pesquisar textos em busca não só das ocorrências da palavra ou expressão que você deseja, mas da palavra ou expressão desejada *mais* a sua tag, é possível melhorar muito a precisão e a eficácia das buscas de seu scraper.

## Recursos adicionais

Processar, analisar e compreender idiomas naturais por computador é uma das tarefas mais difíceis em ciência da computação, e inúmeros volumes e artigos de pesquisa já foram escritos sobre o assunto. Espero que o assunto discutido neste livro inspire você a pensar além do web scraping convencional ou, pelo menos, possa lhe dar uma direção inicial para saber por onde começar quando assumir um projeto que exija análise de línguas naturais.

Muitos recursos excelentes estão disponíveis sobre introdução ao processamento de idiomas e sobre o Natural Language Toolkit de Python. Em particular, o livro [Natural Language Processing with Python](http://oreil.ly/1HYt3vV) (O’Reilly, <http://oreil.ly/1HYt3vV>) de Steven Bird, Ewan Klein e Edward Loper apresenta uma abordagem ampla e ao mesmo tempo introdutória ao assunto.

Além disso, o livro [Natural Language Annotations for Machine Learning](http://oreil.ly/S3BudT) de James Pustejovsky e Amber Stubbs (O’Reilly, <http://oreil.ly/S3BudT>) é um

guia teórico um pouco mais avançado. É necessário ter conhecimento de Python para implementar os exercícios; os tópicos discutidos funcionam perfeitamente com o Natural Language Toolkit de Python.

---

- 1 Embora muitas das técnicas descritas neste capítulo possam ser aplicadas a todos os idiomas, ou à maioria deles, por enquanto não há problema em manter o foco do processamento de idiomas naturais apenas no inglês. Ferramentas como o Natural Language Toolkit de Python, por exemplo, têm o inglês como foco. Cinquenta e seis por cento da internet ainda usa inglês (seguido do alemão, com apenas 6%, de acordo com o [W3Techs](https://w3techs.com/technologies/overview/content_language/all) ([https://w3techs.com/technologies/overview/content\\_language/all](https://w3techs.com/technologies/overview/content_language/all))). Mas quem sabe? É quase certo que a predominância do inglês na maior parte da internet sofrerá mudanças no futuro, e outras atualizações talvez sejam necessárias nos próximos anos.
- 2 Oriol Vinyals et al, [“A Picture Is Worth a Thousand \(Coherent\) Words: Building a Natural Description of Images”](#) (Uma imagem vale mais que mil palavras (coerentes): construindo uma descrição natural de imagens, <http://bit.ly/1HEJ8kX>), Google Research Blog, 17 de novembro de 2014.
- 3 A exceção é a última palavra do texto, pois nada vem depois dela. Em nosso texto de exemplo, a última palavra é um ponto final (.), que é conveniente, pois há outras 215 ocorrências no texto e, desse modo, não representa um beco sem saída. Contudo, em implementações do gerador de Markov no mundo real, a última palavra do texto talvez seja algo que se deva levado em consideração.

# Rastreando formulários e logins

Uma das primeiras perguntas que surgem quando começamos a ir além do básico em web scraping é: “Como acesso informações que estão atrás de uma tela de login?”. A internet avança cada vez mais em direção à interação, mídias sociais e conteúdo gerado por usuários. Formulários e logins são parte integrante desses tipos de sites, e é quase impossível evitá-los. A boa notícia é que também é relativamente fácil lidar com eles.

Até agora, a maior parte de nossas interações com servidores web em nossos exemplos de scrapers consistia no uso de HTTP `GET` para requisitar informações. Este capítulo tem como foco o método `POST`, que envia informações a um servidor web para armazenagem e análise.

Os formulários basicamente oferecem aos usuários uma maneira de submeter uma requisição `POST` que o servidor web é capaz de entender e usar. Assim como as tags de link em um site ajudam os usuários a formatar requisições `GET`, os formulários HTML ajudam a formatar requisições `POST`. É claro que, com um pouco de código, é possível criar essas requisições por conta própria e submetê-las com um scraper.

## Biblioteca Python Requests

Embora seja possível navegar por formulários web usando apenas as bibliotecas básicas de Python, às vezes um pouco de açúcar sintático deixa a vida muito mais doce. Se começarmos a fazer mais que uma requisição `GET` básica com a `urllib`, olhar para além das bibliotecas básicas de Python pode ajudar.

A *biblioteca Requests* (<http://www.python-requests.org>) é excelente para lidar com requisições HTTP complicadas, cookies, cabeçalhos e outros detalhes. Eis o que Kenneth Reitz, criador da Requests, tem a dizer sobre as ferramentas básicas de Python:

O módulo padrão `urllib2` de Python disponibiliza a maior parte dos recursos de HTTP de que você precisa, mas a API é totalmente

inconveniente. Ela foi implementada para uma época diferente – e para uma internet diferente. Um volume de trabalho enorme (até mesmo sobrescrita de métodos) é exigido para executar as mais simples das tarefas.

Não deveria ser assim. Não em Python.

Como qualquer biblioteca Python, a biblioteca *Requests* pode ser instalada com um gerenciador de bibliotecas Python de terceiros, por exemplo, o pip, ou por meio do download e da instalação do [arquivo-fonte \(https://github.com/kennethreitz/requests/tarball/master\)](https://github.com/kennethreitz/requests/tarball/master).

## Submetendo um formulário básico

A maior parte dos formulários web é constituída de alguns campos HTML, um botão de submissão e uma página de ação na qual o processamento do formulário é feito. Em geral, os campos HTML são compostos de texto, mas também podem conter um upload de arquivo ou outro conteúdo que não seja textual.

A maioria dos sites populares bloqueia o acesso aos seus formulários de login em seus arquivos *robots.txt* (o *Capítulo 18* discute a legalidade de fazer scraping em formulários desse tipo), portanto, por segurança, criei uma série de tipos diferentes de formulários e logins em *pythonscraping.com*, nos quais você poderá executar seus web scrapers. <http://pythonscraping.com/pages/files/form.html> é o local em que se encontra o mais básico desses formulários.

Eis o formulário completo:

```
<form method="post" action="processing.php">  
First name: <input type="text" name="firstname"><br>  
Last name: <input type="text" name="lastname"><br>  
<input type="submit" value="Submit">  
</form>
```

Dois detalhes devem ser observados nesse caso: em primeiro lugar, os nomes dos dois campos de entrada são `firstname` e `lastname`. Isso é importante. Os nomes desses campos determinam os nomes dos parâmetros variáveis enviados via `POST` ao servidor quando o formulário é submetido. Se quiser simular a ação executada pelo formulário quando você fizer `POST` dos próprios dados, é preciso garantir que os nomes de suas variáveis sejam os mesmos.

O segundo ponto a ser observado é que a ação do formulário está em *processing.php* (o path absoluto é <http://pythonscraping.com/files/processing.php>). Qualquer requisição POST para o formulário deve ser feita *nessa* página, e não na página em que se encontra o formulário propriamente dito. Lembre-se de que o propósito dos formulários HTML é apenas ajudar os visitantes do site a formatar requisições apropriadas a fim de enviá-las para a página que executa a verdadeira ação. A menos que esteja fazendo uma pesquisa para formatar a requisição, você não terá de se preocupar muito com a página na qual o formulário pode ser encontrado.

Submeter um formulário com a biblioteca *Requests* pode ser feito com quatro linhas, incluindo a importação e a instrução para exibir o conteúdo (sim, é simples assim).

```
import requests

params = {'firstname': 'Ryan', 'lastname': 'Mitchell'}
r = requests.post("http://pythonscraping.com/pages/processing.php", data=params)
print(r.text)
```

Depois que o formulário é submetido, o script devolverá o conteúdo da página:

```
Hello there, Ryan Mitchell!
```

Esse script pode ser aplicado a vários formulários simples encontrados na internet. O formulário para assinar a newsletter da O'Reilly Media, por exemplo, tem o seguinte aspecto:

```
<form action="http://post.oreilly.com/client/o/oreilly/forms/
    quicksignup.cgi" id="example_form2" method="POST">
  <input name="client_token" type="hidden" value="oreilly" />
  <input name="subscribe" type="hidden" value="optin" />
  <input name="success_url" type="hidden" value="http://oreilly.com/store/
    newsletter-thankyou.html" />
  <input name="error_url" type="hidden" value="http://oreilly.com/store/
    newsletter-signup-error.html" />
  <input name="topic_or_dod" type="hidden" value="1" />
  <input name="source" type="hidden" value="orm-home-t1-dotd" />
  <fieldset>
    <input class="email_address long" maxlength="200" name=
      "email_addr" size="25" type="text" value=
      "Enter your email here" />
    <button alt="Join" class="skinny" name="submit" onclick=
      "return addClickTracking('orm','ebook','rightrail','dod'
        );" value="submit">Join</button>
  </fieldset>
```

</form>

Embora pareça intimidador à primeira vista, lembre-se de que, na maioria dos casos (discutiremos depois as exceções), você estará procurando apenas duas informações:

- o nome do campo (ou dos campos) que você quer submeter com os dados (nesse caso, o nome é `email_addr`);
- o atributo de ação do formulário, isto é, a página à qual o formulário faz um `post` (nesse caso, é <http://post.oreilly.com/client/o/oreilly/forms/quicksignup.cgi>).

Basta acrescentar as informações necessárias e executar:

```
import requests
params = {'email_addr': 'ryan.e.mitchell@gmail.com'}
r =
    requests.post("http://post.oreilly.com/client/o/oreilly/forms/quicksignup.cgi",
                  data=params)
print(r.text)
```

Nesse exemplo, o site devolvido é outro formulário a ser preenchido antes de conseguir entrar na lista de distribuição da O'Reilly, mas o mesmo conceito poderia ser aplicado a esse formulário também. Entretanto, peço a você que use seus poderes para o bem e não envie spams à editora com inscrições inválidas, se quiser testar esse código em casa.

## Botões de rádio, caixas de seleção e outras entradas

É claro que nem todos os formulários web são uma coleção de campos de texto seguidos de um botão de submissão. O HTML padrão contém diversos campos de entrada possíveis para formulários: botões de rádio, caixas de seleção, listas suspensas, para citar apenas alguns. O HTML5 acrescenta barras deslizantes (campos de entrada com intervalo), email, datas e outros campos. Com campos JavaScript personalizados, as possibilidades são infinitas, com selecionadores de cores, calendários e tudo mais que os desenvolvedores inventarem.

Não importa a aparente complexidade de qualquer tipo de campo de formulário, é preciso se preocupar apenas com duas informações: o nome do elemento e o seu valor. O nome do elemento pode ser determinado com facilidade olhando o código-fonte e encontrando o atributo `name`. O valor às vezes pode ser mais complicado, pois pode ser preenchido por JavaScript logo antes de o formulário ser submetido. Selecionadores de cores, como

exemplo de um campo de formulário um tanto quanto exótico, provavelmente terão um valor como #F03030.

Se não tiver certeza do formato do valor de um campo de entrada, várias ferramentas podem ser usadas para monitorar as requisições GET e POST que seu navegador troca com os sites. A melhor maneira, e talvez a mais óbvia, de monitorar requisições GET, conforme mencionamos antes, é observar o URL de um site. Se o URL se assemelhar a:

```
http://domainname.com?thing1=foo&thing2=bar
```

você saberá que ele corresponde a um formulário do tipo:

```
<form method="GET" action="someProcessor.php">
<input type="someCrazyInputType" name="thing1" value="foo" />
<input type="anotherCrazyInputType" name="thing2" value="bar" />
<input type="submit" value="Submit" />
</form>
```

que corresponde ao objeto parâmetro de Python:

```
{'thing1': 'foo', 'thing2': 'bar'}
```

Se tiver dificuldade de entender um formulário de POST de aspecto complicado e quiser ver exatamente quais parâmetros seu navegador está enviando para o servidor, o modo mais fácil será usar a ferramenta de inspeção ou as ferramentas de desenvolvedor de seu navegador para visualizá-los (veja a Figura 10.1).

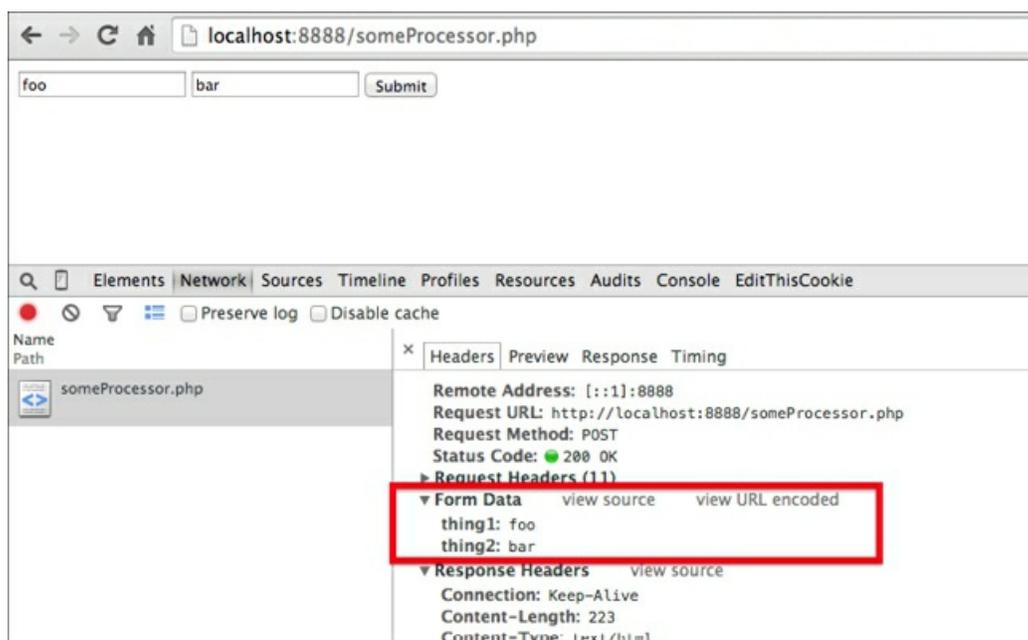


Figura 10.1 – A seção Form Data (Dados de formulário), em destaque na caixa, mostra os parâmetros “thing1” e “thing2” de POST, com valores “foo”

e “bar”.

As ferramentas do desenvolvedor no Chrome podem ser acessadas no menu selecionando View → Developer → Developer Tools (Visualizar → Desenvolvedor → Ferramentas do desenvolvedor). Ela apresenta uma lista de todas as consultas geradas pelo seu navegador enquanto interage com o site atual, e pode ser uma boa maneira de visualizar o conteúdo dessas consultas em detalhes.

## Submetendo arquivos e imagens

Apesar de serem comuns na internet, os uploads de arquivo não são usados com frequência em web scraping. Contudo, é possível que você queira escrever um teste para o próprio site envolvendo um upload de arquivo. De qualquer modo, é interessante saber como fazer isso.

Há um formulário para o exercício de upload de arquivo em <http://pythonscraping/files/form2.html>. Ele contém a seguinte marcação:

```
<form action="processing2.php" method="post" enctype="multipart/form-data">
  Submit a jpg, png, or gif: <input type="file" name="uploadFile"><br>
  <input type="submit" value="Upload File">
</form>
```

Exceto pela tag `<input>` com o atributo de tipo `file`, o formulário parece essencialmente igual aos formulários baseados em texto, usados nos exemplos anteriores. Felizmente, o modo como os formulários são utilizados pela biblioteca Requests de Python também é semelhante:

```
import requests

files = {'uploadFile': open('files/python.png', 'rb')}
r = requests.post('http://pythonscraping.com/pages/processing2.php',
                  files=files)
print(r.text)
```

Observe que, em vez de uma string simples, o valor submetido ao campo do formulário (de nome `uploadFile`) agora é um objeto `File` de Python, conforme devolvido pela função `open`. Nesse exemplo, estamos submetendo um arquivo de imagem, armazenado no computador local, no path `../files/Python-logo.png`, relativo ao local em que o script Python está sendo executado.

Sim, é realmente muito fácil!

## Lidando com logins e cookies

Até agora, discutimos basicamente os formulários que permitem submeter informações a um site ou visualizar informações necessárias na página logo depois do formulário. Como isso difere de um formulário de login, que lhe permite existir em um estado permanente de “login efetuado” durante o acesso ao site?

A maioria dos sites modernos usa cookies para controlar quem está logado e quem não está. Depois que um site autentica suas credenciais de login, ele as armazena no cookie de seu navegador, o qual, em geral, contém um token gerado pelo servidor, um timeout e informações de controle. O site então usa esse cookie como uma espécie de prova de autenticação, que é exibida a cada página que você acessar durante o tempo que estiver no site. Antes do uso disseminado dos cookies em meados dos anos 1990, manter os usuários autenticados de forma segura e monitorá-los era um problema enorme para os sites.

Embora os cookies sejam uma ótima solução para desenvolvedores web, eles podem ser problemáticos para os web scrapers. Podemos submeter um formulário de login o dia todo, mas, se não mantivermos o controle sobre o cookie que o formulário envia de volta, a próxima página acessada agirá como se jamais tivéssemos feito login.

Criei um formulário simples de login em <http://pythonscraping.com/pages/cookies/login.html> (o nome do usuário pode ter qualquer valor, mas a senha deve ser “password”). Esse formulário é processado em <http://pythonscraping.com/pages/cookies/welcome.php>, que contém um link para a página principal, <http://pythonscraping.com/pages/cookies/profile.php>.

Se tentar acessar a página de boas-vindas ou a página de perfil sem fazer login antes, você verá uma mensagem de erro e instruções para fazer login antes de prosseguir. Na página de perfil, uma verificação é feita nos cookies de seu navegador para saber se seu cookie foi definido na página de login.

Manter controle dos cookies é fácil com a biblioteca *Requests*:

```
import requests

params = {'username': 'Ryan', 'password': 'password'}
r = requests.post('http://pythonscraping.com/pages/cookies/welcome.php', params)
print('Cookie is set to:')
```

```

print(r.cookies.get_dict())
print('Going to profile page...')
r = requests.get('http://pythonscraping.com/pages/cookies/profile.php',
                 cookies=r.cookies)
print(r.text)

```

Nesse caso, estamos enviando os parâmetros de login para a página de boas-vindas, que atua como processador do formulário de login. Obtemos os cookies dos resultados da última requisição, exibimos o resultado para conferir e então enviamos esses dados para a página de perfil definindo o argumento `cookies`.

Isso funciona bem em situações simples, mas e se estivéssemos lidando com um site mais complicado, que modificasse os cookies com frequência, sem avisar, ou se preferíssemos, para começar, nem sequer pensar nos cookies? A função `session` da Requests é perfeita para essa situação:

```

import requests

session = requests.Session()

params = {'username': 'username', 'password': 'password'}
s = session.post('http://pythonscraping.com/pages/cookies/welcome.php', params)
print('Cookie is set:')
print(s.cookies.get_dict())
print('Going to profile page...')
s = session.get('http://pythonscraping.com/pages/cookies/profile.php')
print(s.text)

```

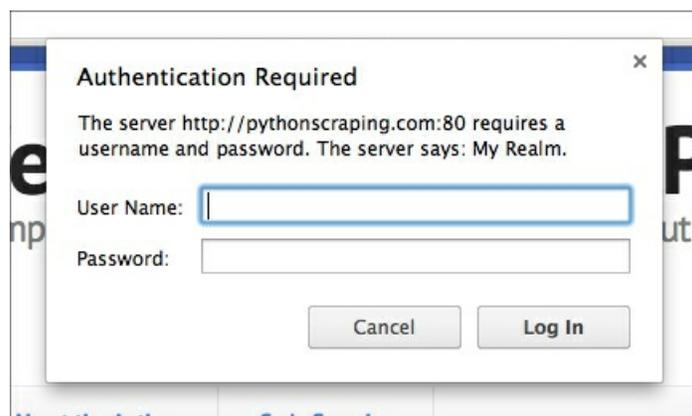
Nesse exemplo, o objeto sessão (obtido por meio da chamada a `requests.Session()`) mantém o controle das informações da sessão, por exemplo, cookies, cabeçalhos e até mesmo informações sobre os protocolos que podem executar sobre o HTTP, como o `HTTPAdapters`.

A *Requests* é uma biblioteca incrível, que perde apenas para o Selenium (será discutido no *Capítulo 11*) no que diz respeito ao tratamento completo dado às tarefas de que trata, sem que os programadores tenham de pensar nela ou escrever o código por conta própria. Embora seja tentador ficar sentado e deixar que a biblioteca faça todo o trabalho, é muito importante estar sempre ciente da aparência dos cookies e saber o que eles controlam quando escrever seus web scrapers. Isso lhe possibilita evitar muitas horas difíceis para depurar o código ou descobrir por que um site está se comportando de modo estranho.

## Autenticação de acesso básica do HTTP

Antes do surgimento dos cookies, um modo conhecido de lidar com logins

era com a *autenticação de acesso básica do HTTP*. Você ainda a verá ocasionalmente, sobretudo em sites com alto nível de segurança ou sites corporativos, e em algumas APIs. Criei uma página em <http://pythonscraping.com/pages/auth/login.php> que usa esse tipo de autenticação (Figura 10.2).



*Figura 10.2 – O usuário deve fornecer um nome de usuário e uma senha para acessar a página protegida pela autenticação de acesso básica.*

Como sempre nesses exemplos, você pode fazer login com qualquer nome de usuário, mas a senha deve ser “password”.

O pacote Requests contém um módulo `auth` especificamente concebido para lidar com autenticação HTTP:

```
import requests
from requests.auth import AuthBase
from requests.auth import HTTPBasicAuth

auth = HTTPBasicAuth('ryan', 'password')
r = requests.post(url='http://pythonscraping.com/pages/auth/login.php',
                  auth=auth)
print(r.text)
```

Embora essa pareça uma requisição `POST` comum, um objeto `HTTPBasicAuth` é passado como o argumento `auth` na requisição. O texto resultante será a página protegida pelo nome do usuário e a senha (ou uma página `Access Denied` [Acesso negado], caso a requisição falhe).

## Outros problemas de formulário

Os formulários web são um ótimo ponto de entrada para bots maliciosos. Não queremos bots criando contas de usuário, utilizando tempo de processamento valioso do servidor nem submetendo comentários spam em

um blog. Por esse motivo, recursos de segurança com frequência são incorporados nos formulários HTML em sites modernos, e talvez não sejam aparentes à primeira vista.

Para obter ajuda com os CAPTCHAs, consulte o *Capítulo 13*, que aborda processamento de imagens e reconhecimento de textos em Python.

Se você deparar com um erro misterioso, o servidor pode estar rejeitando sua submissão de formulário por algum motivo desconhecido, consulte o *Capítulo 14*, que aborda os honeypots, campos ocultos e outras medidas de segurança adotadas pelos sites para proteger seus formulários.

# Scraping de JavaScript

Linguagens de scripting do lado cliente são linguagens que executam no próprio navegador em vez de executar em um servidor web. O sucesso de uma linguagem do lado cliente depende da capacidade do navegador de interpretá-la e executá-la da forma correta. (É por isso que é tão fácil desativar o JavaScript em seu navegador.)

Em parte pela dificuldade de fazer todos os desenvolvedores de navegadores concordarem com um padrão, há muito menos linguagens do lado cliente do que do lado do servidor. Quando se trata de web scraping, isso é bom: quanto menos linguagens houver para tratar, melhor.

Na maioria das vezes, você verá apenas duas linguagens online: ActionScript (usado por aplicações Flash) e JavaScript. O ActionScript é usado com muito menos frequência hoje do que há dez anos e, em geral, é utilizado para streaming de arquivos multimídia, como plataforma para jogos online ou para exibição de páginas de apresentação de sites que ainda não se deram conta de que ninguém quer assistir a uma página de apresentação. De qualquer modo, como não há muita demanda para scraping de páginas Flash, este capítulo terá como foco a linguagem do lado cliente que está presente em todas as páginas web modernas: o JavaScript.

O JavaScript é, sem dúvida, a linguagem de scripting do lado cliente mais comum e mais bem aceita na web atualmente. Pode ser usado para coletar informações de monitoração de usuários, submeter formulários sem recarregar a página, incluir multimídia e também em jogos online completos. Até mesmo páginas de aspecto simples podem conter muitas porções de JavaScript. Podemos encontrá-lo entre tags `script` no código-fonte da página:

```
<script>  
  alert("This creates a pop-up using JavaScript");  
</script>
```

## Introdução rápida ao JavaScript

Ter pelo menos uma ideia do que está acontecendo no código do qual você está coletando dados pode ser muito útil. Com isso em mente, ter familiaridade com JavaScript é uma boa ideia.

*JavaScript* é uma linguagem com tipagem fraca, cuja sintaxe muitas vezes é comparada à sintaxe de C++ e Java. Embora determinados elementos da sintaxe como operadores, laços e arrays sejam parecidos, a tipagem fraca e a natureza da linguagem – que a torna semelhante a uma linguagem de script – podem fazer com que, para alguns programadores, seja difícil lidar com ela.

Por exemplo, o código a seguir calcula valores da sequência de Fibonacci usando recursão, e os exibe no console do desenvolvedor no navegador:

```
<script>
function fibonacci(a, b){
    var nextNum = a + b;
    console.log(nextNum+" is in the Fibonacci sequence");
    if(nextNum < 100){
        fibonacci(b, nextNum);
    }
}
fibonacci(1, 1);
</script>
```

Observe que todas as variáveis são identificadas com um `var` na frente. Isso é semelhante ao sinal `$` em PHP ou à declaração de tipo (`int`, `string`, `List` etc.) em Java ou em C++. Python é diferente por não ter esse tipo de declaração explícita de variável.

JavaScript também é uma ótima linguagem para passar funções como se fossem variáveis:

```
<script>
var fibonacci = function() {
    var a = 1;
    var b = 1;
    return function () {
        var temp = b;
        b = a + b;
        a = temp;
        return b;
    }
}
var fibInstance = fibonacci();
console.log(fibInstance()+" is in the Fibonacci sequence");
```

```
console.log(fibInstance()+" is in the Fibonacci sequence");
console.log(fibInstance()+" is in the Fibonacci sequence");
</script>
```

Isso pode causar espanto à primeira vista, mas será simples se você pensar em termos de expressões lambda (discutidas no *Capítulo 2*). A variável `fibonacci` é definida como uma função que devolve outra função, a qual exibe valores cada vez maiores da sequência de Fibonacci. Sempre que é chamada, ela devolve a função de cálculo de Fibonacci, a qual é executada novamente e incrementa os valores da função.

Embora pareça confuso à primeira vista, alguns problemas, como calcular valores da sequência de Fibonacci, tendem a seguir padrões como esse. Passar funções como variáveis também é muito útil quando se trata de lidar com ações de usuário e callbacks, e vale a pena se sentir à vontade com esse estilo de programação ao ler JavaScript.

## Bibliotecas JavaScript comuns

Embora seja importante conhecer o núcleo da linguagem JavaScript, não é possível ir longe na web moderna sem usar pelo menos uma das muitas bibliotecas de terceiros da linguagem. Podemos ver uma ou mais dessas bibliotecas comuns ao observar o código-fonte de uma página.

Executar JavaScript ao usar Python pode consumir bastante tempo e exigir do processador, sobretudo se feito em larga escala. Saber lidar com JavaScript e ser capaz de interpretá-lo diretamente (sem a necessidade de executá-lo para obter as informações) pode ser bastante conveniente e evitar muitas dores de cabeça.

### jQuery

A *jQuery* é uma biblioteca muito comum, usada por cerca de 70% dos sites mais populares na internet e por aproximadamente 30% do resto da internet<sup>1</sup>. Um site que use a jQuery é prontamente identificável porque faz a importação dessa biblioteca em algum ponto de seu código:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
```

Se a jQuery for encontrada em um site, tome cuidado ao fazer seu scraping, pois ela tende a criar conteúdo HTML dinâmico, o qual aparecerá somente depois que o JavaScript for executado. Se fizer scraping da página usando os métodos tradicionais, você obterá apenas a página previamente carregada, exibida antes de o JavaScript ter criado o conteúdo (esse

problema de scraping será discutido com mais detalhes na seção “*Ajax e HTML dinâmico*”.

Além disso, é provável que essas páginas contenham animações, conteúdo interativo e mídia embutida, que poderão tornar o scraping desafiador.

## Google Analytics

O *Google Analytics* é usado por cerca de 50% de todos os sites<sup>2</sup>, o que talvez o torne a biblioteca JavaScript mais comum e a ferramenta de monitoração de usuários mais popular na internet. Tanto <http://pythonscraping.com> como <http://www.oreilly.com/> usam o Google Analytics.

Determinar se uma página usa o Google Analytics é fácil. Ela terá um código JavaScript no final, semelhante a este (extraído do site da O’Reilly Media):

```
<!-- Google Analytics -->
<script type="text/javascript">

var _gaq = _gaq || [];
_gaq.push(['_setAccount', 'UA-4591498-1']);
_gaq.push(['_setDomainName', 'oreilly.com']);
_gaq.push(['_addIgnoredRef', 'oreilly.com']);
_gaq.push(['_setSiteSpeedSampleRate', 50]);
_gaq.push(['_trackPageview']);

(function() { var ga = document.createElement('script'); ga.type =
'text/javascript'; ga.async = true; ga.src = ('https:' ==
document.location.protocol ? 'https://ssl' : 'http://www') +
'.google-analytics.com/ga.js'; var s =
document.getElementsByTagName('script')[0];
s.parentNode.insertBefore(ga, s); })();

</script>
```

Esse script trata cookies específicos do Google Analytics, usados para monitorar o seu acesso de página em página. Às vezes, isso pode ser um problema para os web scrapers criados a fim de executar JavaScript e tratar cookies (como aqueles que usam o Selenium, discutido mais adiante neste capítulo).

Se um site usa Google Analytics ou um sistema semelhante de análise web (web analytics), e você não quer que ele saiba que está sendo rastreado ou que seus dados estão sendo coletados, não se esqueça de descartar todos os cookies usados na análise – ou todos os cookies.

## Google Maps

Se você já passou um tempo na internet, é quase certo que já tenha visto o *Google Maps* incluído em um site. Sua API facilita bastante disponibilizar mapas com informações personalizadas em qualquer site.

Se estiver coletando qualquer tipo de dado de localização, entender como o Google Maps funciona facilitará obter coordenadas bem formatadas com latitudes/longitudes e até mesmo endereços. Um dos modos mais comuns de representar uma localização no Google Maps é usar um *marcador* (também conhecido como *pino*).

Os marcadores podem ser inseridos em qualquer Google Map usando um código como este:

```
var marker = new google.maps.Marker({
    position: new google.maps.LatLng(-25.363882,131.044922),
    map: map,
    title: 'Some marker text'
});
```

Python facilita extrair todas as ocorrências de coordenadas que estiverem entre `google.maps.LatLng( e )` a fim de obter uma lista de coordenadas com latitudes/longitudes.

Com a *API Reverse Geocoding do Google* (<https://developers.google.com/maps/documentation/javascript/examples/geoc-reverse>), é possível converter esses pares de coordenadas em endereços bem formatados para armazenagem e análise.

## Ajax e HTML dinâmico

Até agora, a única maneira que tínhamos para fazer a comunicação com um servidor web era enviar-lhe algum tipo de requisição HTTP para obtenção de uma nova página. Se você já submeteu um formulário ou adquiriu informações de um servidor sem recarregar a página, é provável que tenha usado um site com Ajax.

Contrário ao que algumas pessoas pensam, o Ajax não é uma linguagem, mas um grupo de tecnologias utilizadas para executar determinada tarefa (pensando bem, é muito semelhante ao web scraping). *Ajax* quer dizer *Asynchronous JavaScript and XML* (JavaScript Assíncrono e XML), e é usado para enviar e receber informações de um servidor web sem fazer outra requisição de página.

Nunca diga “Este site será escrito em Ajax”. O correto seria dizer “Este formulário usará Ajax para se comunicar com o servidor web”.

Assim como o Ajax, o *DHTML* (Dynamic HTML, ou HTML Dinâmico) é um conjunto de tecnologias usado com um propósito comum. O DHTML é composto de código HTML ou linguagem CSS – ou ambos – que mudam à medida que scripts do lado cliente modificam elementos HTML da página. Um botão pode aparecer somente depois que o usuário mover o cursor, uma cor de fundo pode mudar com um clique ou uma requisição Ajax pode disparar a carga de um novo bloco de conteúdo.

Note que, embora a palavra “dinâmico” em geral seja associada a palavras como “mover” ou “alterar”, a presença de componentes HTML interativos, imagens que se movem ou mídia embutida não necessariamente faz com que uma página seja DHTML, apesar de parecer dinâmica. Além do mais, algumas das páginas de aspecto estático mais sem graça da internet podem ter processos DHTML executando internamente, os quais dependem do uso de JavaScript para manipular o HTML e o CSS.

Se fizer scraping de muitos sites, logo você vai deparar com uma situação em que o conteúdo visualizado em seu navegador não será igual ao conteúdo visto no código-fonte obtido do site. Você verá o resultado de seu scraper e coçará a cabeça, tentando entender onde foi parar tudo que você está vendo exatamente na mesma página em seu navegador.

A página web também pode ter uma página de carga que aparece para redirecioná-lo para outra página de resultados, mas você perceberá que o URL da página não muda quando esse redirecionamento acontece.

As duas situações são causadas porque seu scraper não executa o JavaScript que faz a mágica acontecer na página. Sem o JavaScript, o HTML simplesmente fica lá parado, e o site parecerá muito diferente de como será visto em seu navegador web, o qual executa o JavaScript sem problemas.

Há muitas pistas que sinalizam que uma página pode estar usando Ajax ou DHTML para modificar/carregar o conteúdo, mas, em situações como essas, há apenas duas soluções: fazer scraping do conteúdo diretamente do JavaScript ou usar pacotes Python capazes de executar o JavaScript e fazer scraping do site conforme visualizado em seu navegador.

## **Executando JavaScript em Python com o Selenium**

O [Selenium](http://www.seleniumhq.org/) (<http://www.seleniumhq.org/>) é uma ferramenta eficaz para web scraping, desenvolvida originalmente para teste de sites. Atualmente, é usado também quando se precisa de uma imagem exata dos sites – conforme são apresentados em um navegador. O Selenium funciona fazendo os navegadores carregarem o site de modo automático, obtendo os dados necessários e até mesmo capturando imagens de tela ou verificando se determinadas ações ocorrem no site.

O Selenium não contém o próprio navegador web; uma integração com navegadores de terceiros é necessária para a execução. Se o Selenium fosse executado com o Firefox, por exemplo, você veria uma instância do Firefox ser aberta em sua tela, ele acessaria o site, e as ações que você tivesse especificado no código seriam executadas. Embora pareça interessante ver isso, prefiro que meus scripts executem de modo silencioso em segundo plano, portanto uso uma ferramenta chamada [PhantomJS](http://phantomjs.org/) (<http://phantomjs.org/>) em vez de utilizar um navegador de verdade.

O PhantomJS é o que conhecemos como um *navegador headless* (sem cabeça). Ele carrega os sites na memória e executa o JavaScript da página, mas faz isso sem nenhuma renderização de imagens do site para o usuário. Ao combinar o Selenium com o PhantomJS, podemos executar um web scraper bastante eficaz, que lida com cookies, JavaScript, cabeçalhos e tudo que for necessário, com facilidade.

A biblioteca Selenium pode ser baixada a partir de [seu site](https://pypi.python.org/pypi/selenium) (<https://pypi.python.org/pypi/selenium>) ou podemos usar um instalador de terceiros, por exemplo, o pip, na linha de comandos.

O PhantomJS pode ser baixado do [site](http://phantomjs.org/download.html) (<http://phantomjs.org/download.html>). Por ser um navegador completo (apesar de ser headless), e não uma biblioteca Python, o PhantomJS exige download e instalação para ser usado, e não é possível instalá-lo com o pip. Embora muitas páginas utilizem Ajax para carregar dados (em especial, o Google), criei uma página de exemplo em <http://pythonscraping.com/pages/javascript/ajaxDemo.html> na qual seus scrapers podem ser executados. Essa página contém uma amostra de texto, definido no HTML da página, que é substituída por um conteúdo gerado por Ajax depois de dois segundos. Se fôssemos fazer o scraping dos dados dessa página usando métodos tradicionais, obteríamos apenas a página de carga, sem os dados desejados.

A biblioteca Selenium é uma API chamada no objeto WebDriver. O WebDriver é um pouco parecido com um navegador quanto à sua capacidade de carregar sites, mas também pode ser usado como um objeto BeautifulSoup para encontrar elementos da página, interagir com eles (enviar texto, clicar etc.) e executar outras ações para direcionar os web scrapers. O código a seguir obtém o texto que está por trás de uma “parede” Ajax na página de teste:

```
from selenium import webdriver
import time

driver = webdriver.PhantomJS(executable_path='<PhantomJS Path Here>')
driver.get('http://pythonscraping.com/pages/javascript/ajaxDemo.html')
time.sleep(3)
print(driver.find_element_by_id('content').text)
driver.close()
```

## Seletores do Selenium

Nos capítulos anteriores, selecionamos elementos da página usando seletores do BeautifulSoup, como `find` e `find_all`. O Selenium utiliza um conjunto totalmente novo de seletores para encontrar um elemento no DOM de um WebDriver, apesar de terem nomes muito simples.

No exemplo, usamos o seletor `find_element_by_id`, embora os outros seletores a seguir também sejam apropriados:

```
driver.find_element_by_css_selector('#content')
driver.find_element_by_tag_name('div')
```

É claro que, se você quiser selecionar vários elementos da página, a maioria desses seletores é capaz de devolver uma lista de elementos Python se usarmos `elements` (isto é, no plural):

```
driver.find_elements_by_css_selector('#content')
driver.find_elements_by_css_selector('div')
```

Se quiser continuar usando o BeautifulSoup para fazer parse desse conteúdo, use a função `page_source` do WebDriver, que devolve o código-fonte da página, conforme visto pelo DOM naquele momento, na forma de string:

```
pageSource = driver.page_source
bs = BeautifulSoup(pageSource, 'html.parser')
print(bs.find(id='content').get_text())
```

Esse código cria um novo WebDriver Selenium, usando a biblioteca *PhantomJS*, que diz ao WebDriver que carregue uma página e, então, faça uma pausa de três segundos na execução antes de olhar para a página e obter o conteúdo (que esperamos que esteja carregado).

Conforme o local em que estiver a sua instalação do PhantomJS, talvez seja necessário apontar explicitamente a direção correta ao Selenium quando criar um novo WebDriver PhantomJS:

```
driver = webdriver.PhantomJS(executable_path='path/to/driver/'\
                              'phantomjs-1.9.8-macosx/bin/phantomjs')
```

Se tudo estiver configurado da forma correta, o script deve demorar alguns

segundos para executar, e então o resultado será o texto a seguir:

```
Here is some important text you want to retrieve!  
A button to click!
```

Observe que, embora a página em si contenha um botão HTML, a função `.text` do Selenium obtém o valor de texto do botão do mesmo modo que obtém todos os demais conteúdos da página.

Se a pausa em `time.sleep` passar a ser de um segundo em vez de três, o texto devolvido será o texto original:

```
This is some content that will appear on the page while it's loading.  
You don't care about scraping this.
```

Apesar de funcionar, essa solução, de certo modo, é ineficiente, e implementá-la poderia causar problemas em um sistema de grande porte. Os tempos de carga de página são inconsistentes e dependem da carga do servidor em qualquer instante específico; além disso, há variações naturais na velocidade de conexão. Embora a carga dessa página deva demorar pouco mais que dois segundos, estamos lhe concedendo três segundos inteiros para garantir que ela seja totalmente carregada. Uma solução mais eficiente seria verificar repetidamente a existência de um elemento específico em uma página carregada por completo e retornar apenas quando esse elemento existir.

O código a seguir usa a presença do botão cujo ID é `loadedButton` para declarar que a página foi totalmente carregada:

```
from selenium import webdriver  
from selenium.webdriver.common.by import By  
from selenium.webdriver.support.ui import WebDriverWait  
from selenium.webdriver.support import expected_conditions as EC  
  
driver = webdriver.PhantomJS(executable_path='')  
driver.get('http://pythonscraping.com/pages/javascript/ajaxDemo.html')  
try:  
    element = WebDriverWait(driver, 10).until(  
        EC.presence_of_element_located((By.ID, 'loadedButton')))  
finally:  
    print(driver.find_element_by_id('content').text)  
    driver.close()
```

Esse script inclui várias importações novas, com destaque para `WebDriverWait` e `expected_conditions`, ambos combinados nesse caso para compor o que o Selenium chama de *espera implícita*.

Uma espera implícita difere de uma espera explícita por esperar que um determinado estado no DOM ocorra antes de prosseguir, enquanto uma

espera explícita define um tempo fixo, como no exemplo anterior, em que a espera era de três segundos. Em uma espera implícita, o estado do DOM a ser detectado é definido por `expected_condition` (observe que há um cast para `EC` na importação; essa é uma convenção comum, usada para ser mais conciso). As condições esperadas podem variar bastante na biblioteca Selenium, incluindo:

- uma caixa de alerta é exibida;
- um elemento (por exemplo, uma caixa de texto) é colocado em um estado *selecionado*;
- há uma mudança no título da página, ou um texto agora é exibido na página ou em um elemento específico;
- um elemento agora está visível no DOM, ou um elemento desapareceu do DOM.

A maioria dessas condições esperadas exige que você especifique um elemento a ser observado, antes de tudo. Os elementos são especificados com localizadores (locators). Observe que os localizadores não são iguais aos seletores (consulte a seção “*Seletores do Selenium*” para ver mais informações sobre os seletores). Um *localizador* é uma linguagem de consulta abstrata, que usa o objeto `By`; esse objeto pode ser utilizado de diversas maneiras, inclusive na criação de seletores.

No código a seguir, um localizador é usado para encontrar elementos cujo ID é `loadedButton`:

```
EC.presence_of_element_located((By.ID, 'loadedButton'))
```

Os localizadores também podem ser usados para criar seletores, com a função `find_element` do `WebDriver`:

```
print(driver.find_element(By.ID, 'content').text)
```

É claro que, do ponto de vista da funcionalidade, isso equivale à linha do código de exemplo:

```
print(driver.find_element_by_id('content').text)
```

Se não houver necessidade de usar um localizador, não use; você evitará uma importação. No entanto, essa ferramenta conveniente é utilizada em diversas aplicações e tem um alto grau de flexibilidade.

As estratégias de seleção de localizadores a seguir podem ser usadas com o objeto `By`:

**ID**

Usado no exemplo; encontra elementos com base no atributo `id` do HTML.

### CLASS\_NAME

Usado para encontrar elementos com base no atributo `class` do HTML. Por que essa função se chama `CLASS_NAME`, e não apenas `CLASS`? Usar o formato `object.CLASS` criaria problemas para a biblioteca Java do Selenium, na qual `.class` é um método reservado. Para manter a sintaxe do Selenium consistente entre as linguagens, `CLASS_NAME` foi usada.

### CSS\_SELECTOR

Encontra elementos com base no nome de sua `class`, `id` ou `tag`, usando a convenção `#idName`, `.className`, `tagName`.

### LINK\_TEXT

Encontra tags HTML `<a>` com base no texto que contêm. Por exemplo, um link chamado “Next” pode ser selecionado com `(By.LINK_TEXT, "Next")`.

### PARTIAL\_LINK\_TEXT

Semelhante a `LINK_TEXT`, mas faz a correspondência com uma string parcial.

### NAME

Encontra tags HTML com base em seu atributo `name`. É conveniente para formulários HTML.

### TAG\_NAME

Encontra tags HTML com base no nome das tags.

### XPATH

Usa uma expressão `xPath` (cuja sintaxe está descrita na caixa de texto a seguir) para selecionar elementos correspondentes.

## Sintaxe do XPath

*XPath* (forma abreviada de *XML Path*) é uma linguagem de consulta usada para navegar em partes de um documento XML e selecioná-las. Criada pelo W3C em 1999, é ocasionalmente usada em linguagens como Python, Java e C# para lidar com documentos XML.

O BeautifulSoup não tem suporte para XPath, mas muitas das demais bibliotecas usadas neste livro, como Scrapy e Selenium, têm. Com frequência, o XPath pode ser usado do mesmo modo que os seletores CSS (como `mytag#idname`), apesar de ter sido projetado para trabalhar com documentos XML mais genéricos, em vez de documentos HTML em particular.

A sintaxe do XPath apresenta quatro conceitos principais:

- *Nós raiz versus nós que não são raiz*
  - `/div` selecionará o nó `div` somente se estiver na raiz do documento.
  - `//div` seleciona todos os `divs` em qualquer ponto do documento.

- *Seleção de atributos*
  - `//@href` seleciona qualquer nó com o atributo `href`.
  - `//a[@href='http://google.com']` seleciona todos os links do documento que apontem para o Google.
- *Seleção de nós com base na posição*
  - `//a[3]` seleciona o terceiro link do documento.
  - `//table[last()]` seleciona a última tabela do documento.
  - `//a[position() < 3]` seleciona os três primeiros links do documento.
- *Asteriscos (\*) correspondem a qualquer conjunto de caracteres ou nós, e podem ser usados em várias situações.*
  - `//table/tr/*` seleciona todos os filhos de tags `tr` em todas as tabelas (é conveniente para selecionar células que usem tags tanto `th` como `td`).
  - `//div[@*]` seleciona todas as tags `div` com qualquer atributo.

A sintaxe do XPath também tem muitos recursos avançados. Ao longo dos anos, ela se desenvolveu e se transformou em uma linguagem de consulta relativamente complicada, com lógica booleana, funções (como `position()`) e diversos operadores não discutidos nesta seção.

Se você tiver algum problema de seleção de HTML ou de XML que não possa ser resolvido com as funções apresentadas, consulte a [página da Microsoft sobre a sintaxe do XPath \(https://msdn.microsoft.com/en-us/enu/library/ms256471\)](https://msdn.microsoft.com/en-us/enu/library/ms256471).

## Webdrivers adicionais do Selenium

Na seção anterior, o driver do PhantomJS foi usado com o Selenium. Na maioria dos casos, há poucos motivos para que um navegador seja exibido na tela a fim de iniciar o scraping da web, portanto webdrivers headless como o PhantomJS podem ser convenientes. Contudo, usar um tipo diferente de navegador web talvez seja apropriado para executar seus scrapers por algumas razões:

- Resolução de problemas. Se seu código estiver executando PhantomJS e falhar, talvez seja difícil diagnosticar o erro sem que a página esteja diante de seus olhos. Você também pode fazer uma pausa na execução do código e interagir com a página web, como sempre, a qualquer momento.
- Os testes podem depender de um navegador específico para executar.
- Um site ou script incomum, que tenha suas idiossincrasias, pode ter um comportamento um pouco diferente em navegadores distintos. Seu código talvez não funcione no PhantomJS.

Muitos grupos, tanto oficiais como não oficiais, estão envolvidos na criação e na manutenção dos webdrivers Selenium para todos os principais navegadores atuais. O grupo do Selenium mantém um [conjunto desses webdrivers \(http://www.seleniumhq.org/download/\)](http://www.seleniumhq.org/download/) para uma referência fácil.

```
firefox_driver = webdriver.Firefox('<path to Firefox webdriver>')
chrome_driver = webdriver.Chrome('<path to Chrome webdriver>')
safari_driver = webdriver.Safari('<path to Safari webdriver>')
ie_driver = webdriver.Ie('<path to Internet Explorer webdriver>')
```

## Lidando com redirecionamentos

Redirecionamentos do lado cliente são redirecionamentos de página executados em seu navegador pelo JavaScript, em vez de serem executados no servidor antes de o conteúdo da página ser enviado. Às vezes, pode ser complicado fazer a distinção ao acessar uma página em seu navegador web. O redirecionamento pode ocorrer tão rápido que você não perceberá nenhum atraso no tempo de carga, e acreditará que um redirecionamento do lado cliente na verdade é um redirecionamento do lado do servidor.

Entretanto, ao fazer web scraping, a diferença será óbvia. Um redirecionamento do lado do servidor, conforme o modo como é tratado, pode ser facilmente resolvido pela biblioteca Python `urllib`, sem qualquer ajuda do Selenium (para mais informações sobre como fazer isso, veja o *Capítulo 3*). Os redirecionamentos do lado cliente não serão tratados, a menos que alguém execute o JavaScript.

O Selenium é capaz de lidar com esses redirecionamentos do mesmo modo que cuida de outras execuções de JavaScript; contudo, o principal problema com esses redirecionamentos ocorre quando parar a execução da página – isto é, como saber quando uma página terminou o redirecionamento. Uma página de demonstração em <http://pythonscraping.com/pages/javascript/redirectDemo1.html> dá um exemplo desse tipo de redirecionamento, com uma pausa de dois segundos.

Podemos detectar esse redirecionamento de modo inteligente, “observando” um elemento do DOM quando a página é inicialmente carregada, e então chamando esse elemento de forma repetida, até o Selenium lançar uma `StaleElementReferenceException`; o elemento não estará mais vinculado ao DOM da página e o site terá sido redirecionado:

```
from selenium import webdriver
import time
from selenium.webdriver.remote.webelement import WebElement
from selenium.common.exceptions import StaleElementReferenceException

def waitForLoad(driver):
    elem = driver.find_element_by_tag_name("html")
```

```

count = 0
while True:
    count += 1
    if count > 20:
        print('Timing out after 10 seconds and returning')
        return
    time.sleep(.5)
    try:
        elem == driver.find_element_by_tag_name('html')
    except StaleElementReferenceException:
        return

driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get('http://pythonscraping.com/pages/javascript/redirectDemo1.html')
waitForLoad(driver)
print(driver.page_source)

```

Esse script verifica a página a cada meio segundo, com um timeout de 10 segundos, porém os tempos usados para verificação e timeout podem ser ajustados com facilidade, para cima ou para baixo, conforme necessário.

Como alternativa, podemos escrever um laço semelhante que verifique o URL atual da página até que ele mude, ou até que corresponda a um URL específico que estamos procurando.

Esperar que elementos apareçam e desapareçam é uma tarefa comum no Selenium, e a mesma função `WebDriverWait` do exemplo anterior da carga do botão pode ser usada. Nesse caso, estamos especificando um timeout de 15 segundos e um seletor XPath que procura o conteúdo do corpo da página para executar a mesma tarefa:

```

from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import TimeoutException

driver = webdriver.PhantomJS(executable_path=
    'drivers/phantomjs/phantomjs-2.1.1-macosx/bin/phantomjs')
driver.get('http://pythonscraping.com/pages/javascript/redirectDemo1.html')
try:
    bodyElement = WebDriverWait(driver, 15).until(EC.presence_of_element_located(
        (By.XPATH, '//body[contains(text(),
            "This is the page you are looking for!)]"))))
    print(bodyElement.text)
except TimeoutException:
    print('Did not find the element')

```

## Última observação sobre o JavaScript

A maioria dos sites na internet hoje em dia usa JavaScript<sup>3</sup>. Felizmente, para nós, em muitos casos, esse uso de JavaScript não afetará o modo de fazer scraping da página. O JavaScript pode estar limitado às ferramentas de monitoração, ao controle de uma pequena seção do site ou à manipulação de um menu suspenso, por exemplo. Nos casos em que ele exerça impactos sobre o modo de fazer scraping do site, o JavaScript pode ser facilmente executado com ferramentas como o Selenium, a fim de gerar a página com HTML simples cujo scraping aprendemos a fazer na primeira parte do livro.

Lembre-se de que só porque um site usa JavaScript não significa que todas as ferramentas tradicionais de web scraping devam ser jogadas pela janela. O propósito do JavaScript, em última análise, é gerar código HTML e CSS que seja renderizado pelo navegador, ou fazer uma comunicação dinâmica com o servidor por meio de requisições e respostas HTTP. Se o Selenium for usado, o HTML e o CSS da página podem ser lidos e interpretados como você faria com o código de qualquer outro site; as requisições e respostas HTTP podem ser enviadas e tratadas pelo seu código com as técnicas apresentadas nos capítulos anteriores, mesmo sem o uso do Selenium.

Além disso, o JavaScript pode até mesmo ser vantajoso para os web scrapers, pois seu uso como um “sistema de gerenciamento de conteúdo do lado do navegador” pode expor APIs convenientes para o mundo externo, permitindo-lhe obter os dados de modo mais direto. Para outras informações sobre esse assunto, veja o *Capítulo 12*.

Se você continua tendo dificuldades com alguma situação específica e complicada envolvendo JavaScript, o *Capítulo 14* contém informações sobre o Selenium e como interagir diretamente com sites dinâmicos, que incluam interfaces com operações de arrastar e soltar.

---

<sup>1</sup> A postagem de blog de Dave Methvin, “[The State of jQuery 2014](http://bitly.com/2pry8aU/)” (O estado da jQuery em 2014, <http://bitly.com/2pry8aU/>), de 13 de janeiro de 2014, contém um detalhamento das estatísticas.

<sup>2</sup> W3Techs, “[Usage Statistics and Market Share of Google Analytics for Websites](http://w3techs.com/technologies/details/t-googleanalytics/all/all)” (Estatísticas de uso e fatia de mercado do Google Analytics em sites, <http://w3techs.com/technologies/details/t-googleanalytics/all/all>),

<sup>3</sup> W3Techs, “[Usage of JavaScript for Websites](http://w3techs.com/technologies/details/cp-javascript/all/all)” (Uso de JavaScript em sites, <http://w3techs.com/technologies/details/cp-javascript/all/all>).

# Rastreando por meio de APIs

Tradicionalmente, o JavaScript tem sido um obstáculo para os web crawlers em toda parte. Em algum momento na história antiga da internet, podíamos ter a garantia de que uma requisição ao servidor web devolveria os mesmos dados que o usuário veria no navegador se fizesse essa mesma requisição.

À medida que o JavaScript e a geração e carga de conteúdo via Ajax se tornaram mais presentes, essa situação passou a ser menos comum. No *Capítulo 11*, vimos uma maneira de resolver o problema: usar o Selenium para automatizar um navegador e buscar os dados. É uma solução simples. Quase sempre funciona.

O problema é que, quando temos um “martelo” tão potente e eficaz como o Selenium, todo problema de web scraping começa a se parecer bastante com um prego.

Neste capítulo, veremos como contornar totalmente o JavaScript (sem a necessidade de executá-lo ou sequer carregá-lo!) e ir direto à fonte dos dados: as APIs que os geram.

## Introdução rápida às APIs

Apesar de inúmeros livros, palestras e manuais terem sido escritos sobre os detalhes intrincados de APIs REST, GraphQL, JSON e XML, em sua essência, são todos baseados em um conceito simples. Uma *API* define uma sintaxe padronizada que permite a um software se comunicar com outro, mesmo que tenham sido escritos em linguagens diferentes ou estejam estruturados de modo distinto.

Esta seção tem como foco as APIs web (em particular, as APIs que permitem a um servidor web se comunicar com um navegador) e usa o termo *API* para se referir especificamente a esse tipo. Todavia, você deve ter em mente que, em outros contextos, *API* também é um termo genérico, que pode ser usado, por exemplo, para permitir a um programa Java se

comunicar com um programa Python executando no mesmo computador. Uma API nem sempre precisa ser usada “pela internet”, e não necessariamente envolve qualquer tecnologia web.

APIs web com frequência são usadas por desenvolvedores que utilizam um serviço público bastante divulgado e bem documentado. Por exemplo, a ESPN disponibiliza [APIs](http://www.espn.com/apis/devcenter/docs/) (<http://www.espn.com/apis/devcenter/docs/>) para informações sobre atletas, resultados de jogos e outros dados. O Google tem dezenas de APIs em sua [seção para Desenvolvedores](https://console.developers.google.com) (<https://console.developers.google.com>) para tradução de idiomas, análise de dados (analytics) e geolocalização.

A documentação dessas APIs em geral descreve rotas ou *endpoints* como URLs que podem ser requisitados, com parâmetros variáveis, seja no path do URL ou como parâmetros de GET.

Por exemplo, o URL a seguir fornece `pathparam` como um parâmetro da rota:

```
http://example.com/the-api-route/pathparam
```

No caso a seguir, `pathparam` é especificado como valor do parâmetro `param1`:

```
http://example.com/the-api-route?param1=pathparam
```

Os dois métodos para passar dados variáveis à API são usados frequentemente; apesar disso, como em muitas áreas da ciência da computação, o debate filosófico sobre quando e em que locais essas variáveis devem ser passadas no path ou por meio de parâmetros é acirrado.

A resposta da API em geral é devolvida em formato JSON ou XML. O formato JSON é muito mais conhecido que o XML na era moderna, mas você ainda verá algumas respostas XML. Muitas APIs permitem mudar o tipo da resposta, em geral com o uso de outro parâmetro para definir o tipo da resposta que você gostaria de receber.

Eis um exemplo de uma resposta de API em formato JSON:

```
{"user":{"id": 123, "name": "Ryan Mitchell", "city": "Boston"}}
```

A seguir, vemos um exemplo de uma resposta de API em formato XML:

```
<user><id>123</id><name>Ryan Mitchell</name><city>Boston</city></user>
```

O [FreeGeoIP](http://freegeoip.net) (<http://freegeoip.net>) disponibiliza uma API simples e fácil de usar, que traduz endereços IP para endereços físicos. Podemos testar uma requisição simples de API usando o seguinte no navegador<sup>1</sup>:

```
http://freegeoip.net/json/50.78.253.58
```

Eis a resposta gerada:

```
{"ip": "50.78.253.58", "country_code": "US", "country_name": "United States",  
  "region_code": "MA", "region_name": "Massachusetts", "city": "Boston",  
  "zip_code": "02116", "time_zone": "America/New_York", "latitude": 42.3496,  
  "longitude": -71.0746, "metro_code": 506}
```

Observe que a requisição contém o parâmetro `json` no path. Podemos solicitar uma resposta XML ou CSV modificando esse parâmetro conforme desejado:

```
http://freegeoip.net/xml/50.78.253.58
```

```
http://freegeoip.net/csv/50.78.253.58
```

## Métodos HTTP e APIs

Na seção anterior, vimos APIs fazendo requisição `GET` ao servidor em busca de informações. Há quatro maneiras (ou *métodos*) principais de requisitar informações de um servidor web via HTTP:

- GET
- POST
- PUT
- DELETE

Tecnicamente, há outros além desses quatro (como `HEAD`, `OPTIONS` e `CONNECT`), mas são raramente usados em APIs, e é pouco provável que você chegue a vê-los. A grande maioria das APIs se limita a esses quatro métodos, ou até mesmo a um subconjunto deles. É comum ver APIs que usem apenas `GET`, ou apenas `GET` e `POST`.

`GET` é usado quando acessamos um site com a barra de endereço no navegador. É o método empregado quando fazemos uma chamada a <http://freegeoip.net/json/50.78.253.58>. Podemos pensar em `GET` como uma requisição que dissesse: “Ei, servidor web, por favor, obtenha/me dê essas informações”.

Uma requisição `GET`, por definição, não modifica nenhuma informação no banco de dados do servidor. Nada é armazenado; nada é modificado. A informação é apenas lida.

`POST` é usado quando preenchemos um formulário ou submetemos informações, supostamente para um script de backend no servidor. Sempre que fizermos login em um site, estaremos enviando uma requisição `POST` com o nome do usuário e uma senha criptografada (ou esperamos que esteja). Se fizermos uma requisição `POST` com uma API, estaremos dizendo: “Por favor, armazene essas informações em seu banco de dados”.

PUT é menos comum ao interagir com sites, mas é usado de vez em quando em APIs. Uma requisição PUT é utilizada para atualizar um objeto ou uma informação. Uma API pode exigir uma requisição POST para criar um novo usuário, por exemplo, mas poderá pedir uma requisição PUT para atualizar o endereço de email desse usuário<sup>2</sup>:

DELETE é usada, como você deve imaginar, para apagar um objeto. Por exemplo, se uma requisição DELETE for enviada para `http://myapi.com/user/23`, o usuário cujo ID é 23 será apagado. Métodos DELETE não são encontrados com frequência em APIs públicas, criadas essencialmente para disseminar informações ou para permitir aos usuários criar ou atualizar informações, e não para removê-las do banco de dados.

De modo diferente das requisições GET, as requisições POST, PUT e DELETE permitem enviar informações no corpo de uma requisição, além do URL ou da rota da qual os dados estão sendo requisitados.

Assim como a resposta que recebemos do servidor web, em geral esses dados no corpo têm formato JSON; o formato XML é menos comum, e o formato desses dados é definido pela sintaxe da API. Por exemplo, se estivermos usando uma API que crie comentários em postagens de blog, poderemos fazer uma requisição PUT para:

```
http://example.com/comments?post=123
```

com o seguinte corpo na requisição:

```
{"title": "Great post about APIs!", "body": "Very informative. Really helped me out with a tricky technical challenge I was facing. Thanks for taking the time to write such a detailed blog post about PUT requests!", "author": {"name": "Ryan Mitchell", "website": "http://pythonscraping.com", "company": "O'Reilly Media"}}
```

Observe que o ID da postagem de blog (123) é passado como parâmetro no URL, enquanto o conteúdo do novo comentário que estamos criando é incluído no corpo da requisição. Parâmetros e dados podem ser passados tanto no parâmetro como no corpo. Mais uma vez, os parâmetros necessários e o local em que devem ser passados são determinados pela sintaxe da API.

## Mais sobre respostas de APIs<sup>o</sup>

Como vimos no exemplo com o FreeGeoIP no início do capítulo, uma característica importante das APIs é o fato de terem respostas bem formatadas. Os tipos mais comuns de formatação das respostas são XML (eXtensible Markup Language, ou Linguagem de Marcação Extensível) e

JSON (JavaScript Object Notation, ou Notação de Objetos JavaScript).

Nos últimos anos, o JSON se tornou muito mais popular que o XML por dois motivos principais. Em primeiro lugar, arquivos JSON em geral são menores que arquivos XML com um bom design. Compare, por exemplo, os dados XML a seguir, que contêm exatamente 98 caracteres:

```
<user><firstname>Ryan</firstname><lastname>Mitchell</lastname><username>Kludgist</username></user>
```

Observe agora os mesmos dados em JSON:

```
{"user":{"firstname":"Ryan","lastname":"Mitchell","username":"Kludgist"}}
```

São apenas 73 caracteres, ou seja, 36% menor que o XML equivalente.

É claro que seria possível argumentar que o XML poderia estar formatado assim:

```
<user firstname="ryan" lastname="mitchell" username="Kludgist"></user>
```

Contudo, essa opção é considerada uma prática ruim, pois não permite que os dados sejam aninhados com mais níveis de profundidade. Apesar disso, esse formato ainda exige 71 caracteres – cerca do mesmo tamanho que o equivalente em JSON.

Outro motivo pelo qual JSON está se tornando cada vez mais popular que o XML se deve a uma mudança nas tecnologias web. No passado, era mais comum que um script do lado do servidor, como PHP ou .NET, estivesse no lado receptor de uma API. Atualmente, é provável que um framework, por exemplo, o Angular ou o Backbone, envie e receba chamadas de API. Tecnologias do lado do servidor, de certo modo, são independentes do formato com que os dados chegam. No entanto, bibliotecas JavaScript como o Backbone acham mais fácil lidar com JSON.

Embora, em geral, imaginemos as API com uma resposta XML ou JSON, qualquer formato é possível. O tipo de resposta da API está limitado apenas pela imaginação do programador que a criou. O CSV é outro tipo de resposta típico (como vimos no exemplo com o FreeGeoIP). Algumas APIs podem ser projetadas até mesmo para gerar arquivos. Uma requisição pode ser feita a um servidor para gerar uma imagem com algum texto específico sobreposto ou para solicitar um arquivo XLSX ou PDF em particular.

Algumas APIs não devolvem nenhuma resposta. Por exemplo, se estivéssemos fazendo uma requisição para um servidor a fim de criar um novo comentário para uma postagem de blog, apenas um código de resposta HTTP 200 poderia ser devolvido, cujo significado seria: “Postei o

comentário; tudo vai bem!”. Outras APIs podem devolver uma resposta mínima, como esta:

```
{"success": true}
```

Se houver um erro, uma resposta como esta poderia ser recebida:

```
{"error": {"message": "Something super bad happened"}}
```

Ou, se a API não estiver particularmente bem configurada, você poderia receber um stack trace que não permitiria parsing ou um texto em inglês puro. Ao fazer uma requisição a uma API, em geral, verificar antes se a resposta obtida é realmente JSON (ou XML, CSV ou qualquer formato que você esteja esperando) é uma atitude inteligente.

## Parsing de JSON

Neste capítulo, vimos vários tipos de APIs e como funcionam, além de exemplos de respostas JSON dessas APIs. Vamos ver agora como fazer parse dessas informações e usá-las.

No início do capítulo, vimos o exemplo de *freegeoip.net* IP, que converte endereços IP em endereços físicos:

```
http://freegeoip.net/json/50.78.253.58
```

Podemos tomar o resultado dessa requisição e usar as funções de parsing de JSON de Python para decodificá-lo:

```
import json
from urllib.request import urlopen

def getCountry(ipAddress):
    response = urlopen('http://freegeoip.net/json/'+ipAddress).read()
    .decode('utf-8')
    responseJson = json.loads(response)
    return responseJson.get('country_code')

print(getCountry('50.78.253.58'))
```

O código do país para o endereço IP 50.78.253.58 é exibido.

A biblioteca de parsing de JSON usada faz parte da biblioteca nuclear de Python. Basta digitar `import json` no início, e pronto! De modo diferente de muitas linguagens capazes de fazer parse de JSON e gerar um objeto especial ou um nó JSON, Python usa uma abordagem mais flexível e transforma objetos JSON em dicionários, arrays JSON em listas, strings JSON em strings, e assim por diante. Desse modo, é muito fácil acessar e manipular valores armazenados em JSON.

O código a seguir faz uma demonstração rápida de como a biblioteca JSON de Python trata os valores que podem ser encontrados em uma string JSON:

```
import json

jsonString = '{"arrayOfNums":[{"number":0},{"number":1},{"number":2}],
              "arrayOfFruits":[{"fruit":"apple"},{"fruit":"banana"},
                              {"fruit":"pear"}]}'

jsonObj = json.loads(jsonString)

print(jsonObj.get('arrayOfNums'))
print(jsonObj.get('arrayOfNums')[1])
print(jsonObj.get('arrayOfNums')[1].get('number') +
      jsonObj.get('arrayOfNums')[2].get('number'))
print(jsonObj.get('arrayOfFruits')[2].get('fruit'))
```

Eis a saída:

```
[{'number': 0}, {'number': 1}, {'number': 2}]
{'number': 1}
3
pear
```

A linha 1 é uma lista de objetos dicionário, a linha 2 é um objeto dicionário, a linha 3 é um inteiro (a soma dos inteiros acessados nos dicionários) e a linha 4 é uma string.

## APIs não documentadas

Até agora neste capítulo, discutimos apenas as APIs documentadas. A intenção dos desenvolvedores é que as APIs sejam usadas pelo público, e os desenvolvedores publicam informações sobre elas supondo que serão utilizadas por outros desenvolvedores. Entretanto, a grande maioria das APIs não tem nenhuma documentação publicada.

Por que você criaria uma API sem qualquer documentação pública? Conforme mencionamos no início deste capítulo, isso tem tudo a ver com JavaScript.

Tradicionalmente, os servidores web para sites dinâmicos executavam várias tarefas sempre que um usuário requisitava uma página:

- tratavam requisições GET dos usuários que solicitassem uma página de um site;
- obtinham do banco de dados os dados apresentados nessa página;
- formatavam os dados no template HTML da página;

- enviavam esse HTML formatado para o usuário.

À medida que os frameworks JavaScript se tornaram mais comuns, muitas das tarefas de criação de HTML tratadas pelo servidor passaram para o navegador. O servidor pode enviar um template HTML fixo ao navegador do usuário, mas requisições Ajax separadas seriam feitas para carregar o conteúdo e colocá-lo nos lugares corretos nesse template. Tudo isso ocorreria do lado do navegador/cliente.

Inicialmente, isso era um problema para os web scrapers. Eles estavam habituados a fazer a requisição de uma página HTML e obter exatamente isso – uma página HTML, já com todo o conteúdo no lugar certo. Em vez disso, eles passaram a receber um template HTML sem conteúdo.

O Selenium foi usado para resolver esse problema. A partir de agora, o web scraper do programador poderia se transformar no navegador, requisitar o template HTML, executar qualquer código JavaScript, permitir que todos os dados fossem carregados em seus lugares e somente, *então*, fariam scraping dos dados da página. Como o HTML era todo carregado, tudo se reduzia essencialmente a um problema já solucionado – o problema de fazer parsing e formatar um HTML existente.

Contudo, pelo fato de todo o sistema de gerenciamento de conteúdo (que costumava estar apenas no servidor web) ter essencialmente passado para o cliente, isto é, o navegador, até mesmo o mais simples dos sites poderia exigir vários megabytes de conteúdo e uma dúzia de requisições HTTP.

Além disso, quando o Selenium é usado, todos os “extras” que não necessariamente interessam ao usuário são carregados. Chamadas para programas de monitoração, carga de anúncios em barras laterais, chamadas de programas de monitoração para anúncios nas barras laterais. Imagens, CSS, dados de fontes de terceiros – tudo isso tem de ser carregado. Pode parecer ótimo se estivermos usando um navegador para acessar a web, mas, se estivermos escrevendo um web scraper que deva ser rápido, coletar dados específicos e impor o mínimo de carga possível ao servidor web, poderíamos estar carregando cem vezes mais dados do que o necessário.

Contudo, há um lado bom do JavaScript, Ajax e todas essas modernizações na web: como não formatam mais os dados em HTML, em geral os servidores atuam como wrappers finos em torno do próprio banco de dados. Esse wrapper simplesmente extrai dados do banco de dados e os

devolve para a página por meio de uma API.

É claro que essas APIs não foram criadas com o intuito de serem usadas por nada nem ninguém além da própria página e, desse modo, os desenvolvedores não as documentam, e supõem (ou esperam) que ninguém as notará. Porém, elas existem.

O [site do The Nova York Times](http://nytimes.com) (<http://nytimes.com>), por exemplo, carrega todos os seus resultados de pesquisa com JSON. Se você acessar o link:

```
https://query.nytimes.com/search/sitesearch/#/python
```

artigos de notícias recentes para o termo pesquisado “python” serão exibidos. Se fizer scraping dessa página usando a biblioteca `urllib` ou a `Requests`, você não verá nenhum resultado de pesquisa. Eles são carregados de forma separada, por meio de uma chamada de API:

```
https://query.nytimes.com/svc/add/v1/sitesearch.json  
?q=python&spotlight=true&facet=true
```

Se carregássemos essa página com o Selenium, faríamos cerca de 100 requisições e transferiríamos de 600 a 700 kB de dados a cada pesquisa. Ao usar diretamente a API, faríamos apenas uma requisição e transferiríamos apenas os cerca de 60 kb de dados necessários e bem formatados.

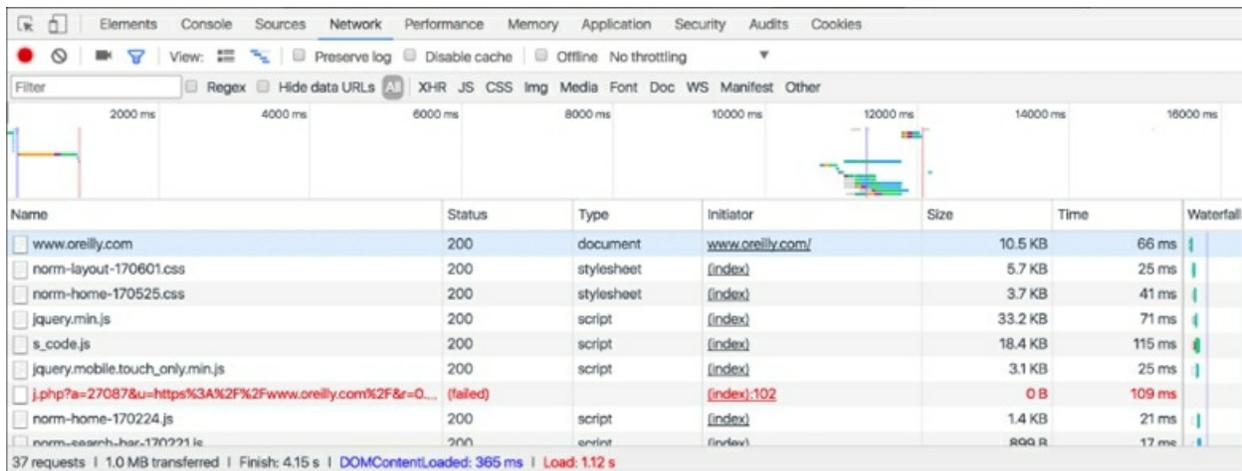
## Encontrando APIs não documentadas

Usamos a ferramenta de inspeção do Chrome nos capítulos anteriores para analisar o conteúdo de uma página HTML, mas agora nós a empregaremos com um propósito um pouco diferente: analisar as requisições e as respostas das chamadas efetuadas para construir essa página.

Para isso, abra a janela de inspeção do Chrome e clique na aba Network (Rede), exibida na Figura 12.1.

Note que é necessário abrir essa janela antes de a página ser carregada. Chamadas de rede não serão monitoradas se a janela estiver fechada.

Enquanto a página estiver carregando, veremos uma linha ser exibida em tempo real sempre que o navegador fizer uma chamada para o servidor web requisitando informações adicionais a fim de renderizar a página. Uma chamada de API pode estar incluída aí.



*Figura 12.1 – A ferramenta de inspeção de rede do Chrome exibe todas as chamadas que seu navegador faz e recebe.*

Encontrar APIs não documentadas talvez exija um pouco de trabalho de detetive (para eliminar o trabalho de detetive, consulte a seção “*Encontrando e documentando APIs de modo automático*”), sobretudo em sites maiores, com muitas chamadas de rede. Em geral, porém, você as reconhecerá ao vê-las.

As chamadas de API tendem a ter várias características convenientes para localizá-las na lista de chamadas de rede:

- Com frequência, contêm JSON ou XML. Podemos filtrar a lista de requisições usando o campo de pesquisa/filtragem.
- Em requisições GET, o URL conterà os valores dos parâmetros passados. Isso será útil, por exemplo, se estivermos procurando uma chamada de API que devolva o resultado de uma pesquisa ou que carregue dados de uma página específica. Basta filtrar o resultado com o termo de pesquisa usado, o ID da página ou outras informações de identificação.
- Em geral, elas serão do tipo XHR.

As APIs talvez nem sempre sejam óbvias, particularmente em sites grandes, com muitas funcionalidades que poderiam gerar centenas de chamadas durante a carga de uma única página. No entanto, encontrar a agulha metafórica no palheiro será muito mais fácil com um pouco de prática.

## Documentando APIs não documentadas

Depois de ter encontrado uma chamada de API, em geral será conveniente documentá-la até certo ponto, sobretudo se seus scrapers dependerem

bastante dessa chamada. Talvez você queira carregar várias páginas do site, filtrando a chamada de API desejada na aba de rede no console da ferramenta de inspeção. Ao fazer isso, poderemos ver como as chamadas mudam de página para página, e identificar os campos que elas aceitam e devolvem.

Qualquer chamada de API pode ser identificada e documentada se prestarmos atenção nos seguintes campos:

- Método HTTP usado
- Entradas
  - Parâmetros do path
  - Cabeçalhos (incluindo cookies)
  - Conteúdo do corpo (para chamadas PUT e POST)
- Saídas
  - Cabeçalhos da resposta (incluindo os cookies definidos)
  - Tipo do corpo da resposta
  - Campos do corpo da resposta

## **Encontrando e documentando APIs de modo automático**

A tarefa de localizar e documentar APIs pode parecer um pouco enfadonha e algorítmica. Isso ocorre porque, em sua maior parte, é assim mesmo. Enquanto alguns sites podem tentar ofuscar o modo como o navegador obtém os dados, o que torna a tarefa um pouco mais complicada, encontrar e documentar APIs é essencialmente uma tarefa de programação.

Criei um repositório no GitHub em <https://github.com/REMitchell/apiscraper> que procura eliminar parte do trabalho pesado.

O programa usa Selenium, ChromeDriver e uma biblioteca chamada BrowserMob Proxy para carregar páginas, rastreá-las em um domínio, analisar o tráfego de rede durante a carga das páginas e organizar essas requisições em chamadas de API legíveis.

Várias partes são necessárias para que esse projeto execute. A primeira é o software propriamente dito.

Crie um clone do projeto [apiscraper](https://github.com/REMitchell/apiscraper) (<https://github.com/REMitchell/apiscraper>) que está no GitHub. O projeto clonado deve conter os seguintes arquivos:

## **apicall.py**

Contém atributos que definem uma chamada de API (path, parâmetros etc.), assim como a lógica para decidir se duas chamadas de API são iguais.

## **apiFinder.py**

Classe principal de rastreamento. Usada por *webservice.py* e *consoleservice.py* para disparar o processo de encontrar APIs.

## **browser.py**

Tem apenas três métodos – `initialize`, `get` e `close` –, mas engloba funcionalidades relativamente complicadas para fazer a associação entre o servidor BrowserMob Proxy e o Selenium. Faz rolagem nas páginas para garantir que a página completa seja carregada e salva arquivos HAR (HTTP Archive) no local apropriado para processamento.

## **consoleservice.py**

Trata comandos do console e inicia a classe principal `APIFinder`.

## **harParser.py**

Faz parsing de arquivos HAR e extrai as chamadas de API.

## **html\_template.html**

Fornece um template para exibir as chamadas de API no navegador.

## **README.md**

Página readme do Git.

Faça download dos arquivos binários do BrowserMob Proxy a partir de <https://bmp.lightbody.net/> e coloque os arquivos extraídos no diretório do projeto *apiscraper*.

Atualmente (quando este livro foi escrito), o BrowserMob Proxy está na versão 2.1.4, portanto esse script supõe que os arquivos binários estão em *browsermob-proxy-2.1.4/bin/browsermob-proxy*, relativo ao diretório-raiz do projeto. Se esse não for o caso, você poderá especificar um diretório diferente em tempo de execução, ou modificar o código em *apiFinder.py* (talvez seja mais fácil).

Faça download do [ChromeDriver](https://sites.google.com/a/chromium.org/chromedriver/downloads) (<https://sites.google.com/a/chromium.org/chromedriver/downloads>) e coloque-o no diretório do projeto *apiscraper*.

As seguintes bibliotecas Python devem estar instaladas:

- tldextract
- selenium
- browsermob-proxy

Depois que tiver essa configuração, você estará pronto para começar a coletar chamadas de API. Digitar:

```
$ python consoleservice.py -h
```

apresentará uma lista de opções:

```
usage: consoleservice.py [-h] [-u [U]] [-d [D]] [-s [S]] [-c [C]] [-i [I]]
                        [--p]
```

optional arguments:

- h, --help show this help message and exit
- u [U] Target URL. If not provided, target directory will be scanned for har files.
- d [D] Target directory (default is "hars"). If URL is provided, directory will store har files. If URL is not provided, directory will be scanned.
- s [S] Search term
- c [C] File containing JSON formatted cookies to set in driver (with target URL only)
- i [I] Count of pages to crawl (with target URL only)
- p Flag, remove unnecessary parameters (may dramatically increase runtime)

Podemos procurar chamadas de API feitas em uma única página para um único termo de pesquisa. Por exemplo, podemos pesquisar uma página em <https://target.com> em busca de uma API que devolva dados de produto para preencher a página de um produto:

```
$ python consoleservice.py -u https://www.target.com/p/rogue-one-a-star-wars-
story-blu-ray-dvd-digital-3-disc/-/A-52030319 -s "Rogue One: A Star Wars Story"
```

Esse comando devolve informações, incluindo um URL, sobre uma API que devolve os dados de produto dessa página:

```
URL: https://redsky.target.com/v2/pdp/tcin/52030319
METHOD: GET
AVG RESPONSE SIZE: 34834
SEARCH TERM CONTEXT: c:"786936852318","product_description":{"title":
"Rogue One: A Star Wars Story (Blu-ray + DVD + Digital) 3 Disc",
"long_description":...
```

Usando a flag `-i`, várias páginas podem ser rastreadas (o default é apenas uma página), começando pelo URL especificado. Isso pode ser conveniente para pesquisar todo o tráfego de rede em busca de palavras-chaves específicas ou, se a flag de termo de pesquisa `-s` for omitida, coletar todo o tráfego de API ao carregar cada página.

Todos os dados coletados são armazenados em um arquivo HAR, no diretório default `/har` na raiz do projeto, embora seja possível alterar esse diretório com a flag `-d`.

Se nenhum URL for fornecido, também é possível passar um diretório com arquivos HAR previamente coletados para pesquisa e análise.

Esse projeto oferece várias outras funcionalidades, incluindo:

- remoção de parâmetros desnecessários (remoção de parâmetros de `GET` ou `POST` que não influenciam o valor de retorno da chamada de API);
- vários formatos de saída da API (linha de comando, HTML, JSON);
- distinção entre parâmetros de path que sinalizam uma rota de API diferente e parâmetros de path que atuam apenas como parâmetros de `GET` para a mesma rota de API.

Outros desenvolvimentos também estão nos planos à medida que eu e outras pessoas continuarmos usando o projeto para web scraping e coleta de APIs.

## **Combinando APIs com outras fontes de dados**

Ainda que a razão de ser de muitas aplicações web modernas seja obter dados existentes e formatá-los de modo mais atraente, eu diria que, na maioria das vezes, essa não é uma tarefa interessante. Se estiver usando uma API como única fonte de dados, o máximo que você fará é copiar o banco de dados já existente de alguém, o qual, essencialmente, já está publicado. Uma tarefa muito mais interessante seria usar duas ou mais fontes de dados e combiná-las de uma nova maneira, ou usar uma API como uma ferramenta para analisar os dados coletados a partir de uma nova perspectiva.

Vamos observar um exemplo de como dados de APIs podem ser usados em conjunto com web scraping para ver quais partes do mundo contribuem mais com a Wikipédia.

Se você já passou bastante tempo na Wikipédia, é provável que já tenha

deparado com a página de histórico de revisões do artigo, a qual exibe uma lista das modificações recentes. Se os usuários estiverem logados na Wikipédia quando fizerem a modificação, seus nomes de usuário serão exibidos. Se não estiverem, seu endereço IP será registrado, como vemos na Figura 12.2.

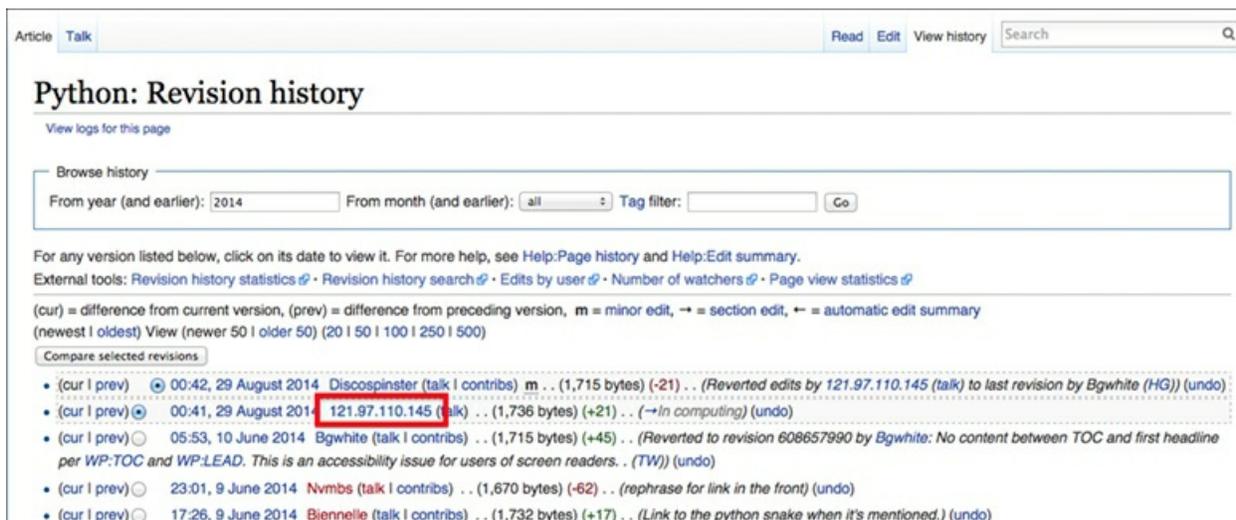


Figura 12.2 – Endereço IP de um autor anônimo que fez uma modificação, na página de histórico de revisões da entrada associada a Python na Wikipédia.

O endereço IP exibido na página de histórico é 121.97.110.145. Ao usar a API de *freegeoip.net*, vemos que esse endereço IP é de Quezon nas Filipinas, quando este livro foi escrito (endereços IP podem, às vezes, mudar geograficamente).

Por si só, essa informação não é muito interessante, mas e se pudéssemos reunir vários dados geográficos sobre as modificações na Wikipédia e os locais em que ocorreram? Alguns anos atrás, fiz exatamente isso e usei a [biblioteca GeoChart do Google](https://developers.google.com/chart/interactive/docs/gallery/geochart) (<https://developers.google.com/chart/interactive/docs/gallery/geochart>) para criar um [mapa interessante](http://www.pythonscraping.com/pages/wikipedia.html) (<http://www.pythonscraping.com/pages/wikipedia.html>) que mostra os lugares de onde as modificações foram feitas; fiz isso para a Wikipédia de língua inglesa, além da Wikipédia escrita em outros idiomas (Figura 12.3).

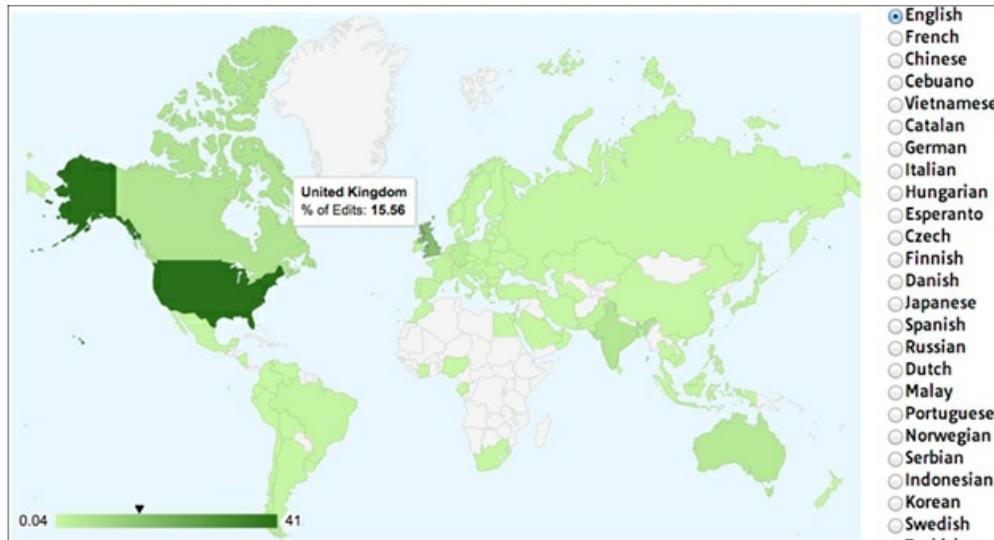


Figura 12.3 – Visualização das modificações na Wikipédia, criada com a biblioteca GeoChart do Google.

Criar um script básico que rastreie a Wikipédia, procure as páginas de histórico de revisões e então procure aí os endereços IP não é difícil. Usando um código do *Capítulo 3* modificado, o script a seguir faz exatamente isso:

```

from urllib.request import urlopen
from bs4 import BeautifulSoup
import json
import datetime
import random
import re

random.seed(datetime.datetime.now())
def getLinks(articleUrl):
    html = urlopen('http://en.wikipedia.org{}'.format(articleUrl))
    bs = BeautifulSoup(html, 'html.parser')
    return bs.find('div', {'id':'bodyContent'}).findAll('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))

def getHistoryIPs(pageUrl):
    # Este é o formato das páginas de histórico de revisões:
    # http://en.wikipedia.org/w/index.php?title=Title_in_URL&action=history
    pageUrl = pageUrl.replace('/wiki/', '')
    historyUrl = 'http://en.wikipedia.org/w/index.php?title={}&action=history'
        .format(pageUrl)
    print('history url is: {}'.format(historyUrl))
    html = urlopen(historyUrl)
    bs = BeautifulSoup(html, 'html.parser')
    # encontra apenas os links cuja classe seja "mw-anonuserlink" e
    # tenha endereços IP em vez de nomes de usuário

```

```

ipAddresses = bs.findAll('a', {'class': 'mw-anonuserlink'})
addressList = set()
for ipAddress in ipAddresses:
    addressList.add(ipAddress.get_text())
return addressList

links = getLinks('/wiki/Python_(programming_language)')

while(len(links) > 0):
    for link in links:
        print('- '*20)
        historyIPs = getHistoryIPs(link.attrs['href'])
        for historyIP in historyIPs:
            print(historyIP)

    newLink = links[random.randint(0, len(links)-1)].attrs['href']
    links = getLinks(newLink)

```

Esse programa usa duas funções principais: `getLinks` (também utilizada no *Capítulo 3*), e a nova função `getHistoryIPs`, que pesquisa o conteúdo de todos os links cuja classe é `mw-anonuserlink` (indicando que é um usuário anônimo com um endereço IP, em vez de ter um nome de usuário) e os devolve na forma de um conjunto.

Esse código também usa um padrão de pesquisa, de certo modo, arbitrário (apesar de ser eficaz neste exemplo) para procurar artigos dos quais os históricos de revisão serão obtidos. Ele começa obtendo os históricos de todos os artigos da Wikipédia para os quais a página inicial aponta (nesse caso, o artigo sobre a linguagem de programação Python). Depois disso, o código seleciona aleatoriamente uma nova página inicial e obtém todas as páginas de histórico de revisões dos artigos para os quais essa página aponta. O programa continuará até alcançar uma página sem links.

Agora que temos o código capaz de obter endereços IP na forma de string, podemos combiná-lo com a função `getCountry` da seção anterior a fim de converter esses endereços IP em países. Modificaremos um pouco o código de `getCountry` para que leve em consideração endereços IP inválidos ou mal formatados, e que resultarão em um erro 404 Not Found (Não encontrado) – quando este livro foi escrito, o FreeGeoIP não fazia conversão de IPv6, por exemplo, o que poderia gerar um erro deste tipo:

```

def getCountry(ipAddress):
    try:
        response = urlopen(
            'http://freegeoip.net/json/{}'.format(ipAddress)).read().decode('utf-8')
    except HTTPError:
        return None

```

```

responseJson = json.loads(response)
return responseJson.get('country_code')

links = getLinks('/wiki/Python_(programming_language)')

while(len(links) > 0):
    for link in links:
        print('- '*20)
        historyIPs = getHistoryIPs(link.attrs["href"])
        for historyIP in historyIPs:
            country = getCountry(historyIP)
            if country is not None:
                print('{} is from {}'.format(historyIP, country))

        newLink = links[random.randint(0, len(links)-1)].attrs['href']
        links = getLinks(newLink)

```

Eis o resultado do exemplo:

```

-----
history url is: http://en.wikipedia.org/w/index.php?title=Programming_
paradigm&action=history
68.183.108.13 is from US
86.155.0.186 is from GB
188.55.200.254 is from SA
108.221.18.208 is from US
141.117.232.168 is from CA
76.105.209.39 is from US
182.184.123.106 is from PK
212.219.47.52 is from GB
72.27.184.57 is from JM
49.147.183.43 is from PH
209.197.41.132 is from US
174.66.150.151 is from US

```

## Mais sobre APIs

Este capítulo apresentou algumas maneiras comuns de usar APIs modernas para acessar dados na internet, e como essas APIs podem ser empregadas para construir web scrapers mais rápidos e mais eficazes. Se você pretende construir APIs em vez de apenas usá-las, ou se quiser saber mais sobre a teoria associada à sua construção e sintaxe, recomendo o livro [RESTful Web APIs](http://bit.ly/RESTful-Web-APIs) de Leonard Richardson, Mike Amundsen e Sam Ruby (O'Reilly, <http://bit.ly/RESTful-Web-APIs>). Esse livro contém uma visão geral teórica e prática bem robusta para usar APIs na internet. Além disso, Mike Amundsen tem uma série incrível de vídeos, [Designing APIs for the Web](http://oreil.ly/1GOXNhE) (O'Reilly, <http://oreil.ly/1GOXNhE>), que ensina você a criar as próprias

APIs – um conhecimento útil caso decida deixar seus dados coletados disponíveis ao público em um formato conveniente.

Embora algumas pessoas lamentem a presença de JavaScript e de sites dinâmicos por todo lado, o que torna as práticas tradicionais de “obter e fazer parse da página HTML” desatualizadas, eu, particularmente, dou as boas-vindas aos novos senhores. Pelo fato de os sites dinâmicos dependerem menos de páginas HTML para consumo humano e mais de arquivos JSON rigorosamente formatados para consumo de HTML, isso é muito vantajoso para qualquer pessoa que esteja tentando obter dados limpos e bem formatados.

A internet não é mais uma coleção de páginas HTML com dados ocasionais de multimídia e adornos CSS, e, sim, uma coleção de centenas de tipos de arquivos e de formatos de dados, trocados rapidamente às centenas para compor páginas a serem consumidas pelo seu navegador. O verdadeiro truque, em geral, é olhar para além da página que está diante de você e obter os dados na fonte.

---

[1](#) Essa API converte endereços IP para localizações geográficas e é uma das APIs que usaremos mais adiante neste capítulo.

[2](#) Na verdade, muitas APIs usam requisições **POST** no lugar de **PUT** para atualizar informações. Se uma nova entidade será criada ou é uma antiga que está sendo atualizada em geral fica a cargo de como a própria requisição da API está estruturada. Apesar disso, é bom saber a diferença e, com frequência, você verá requisições **PUT** em APIs comumente usadas.

# Processamento de imagens e reconhecimento de texto

Dos carros autônomos do Google às máquinas de venda automáticas que reconhecem notas falsificadas, a visão computacional (machine vision) é um campo enorme, com metas e implicações de longo alcance. Este capítulo tem como foco um dos pequenos aspectos desse campo: o reconhecimento de textos – especificamente, como reconhecer e usar imagens baseadas em texto encontradas online utilizando diversas bibliotecas Python.

Usar uma imagem no lugar de um texto é uma técnica comum quando não queremos que um texto seja encontrado e lido por bots. Com frequência, isso é visto em formulários de contato, quando um endereço de email é renderizado de modo parcial ou total como uma imagem. Conforme o nível de habilidade empregado, isso talvez nem seja perceptível aos seres humanos, mas os bots têm dificuldade para ler essas imagens, e a técnica é suficiente para impedir que a maioria dos geradores de spams adquira os endereços de email.

Os CAPTCHAs, é claro, tiram proveito do fato de os usuários lerem imagens de segurança, enquanto a maioria dos bots não é capaz de fazê-lo. Alguns CAPTCHAs são mais difíceis que outros – um problema que abordaremos mais adiante neste livro.

Os CAPTCHAs, porém, não são o único recurso da web nos quais os scrapers precisam da assistência de uma tradução de imagem para texto. Mesmo nos dias de hoje, muitos documentos são digitalizados a partir de cópias impressas e colocados na internet, tornando-os inacessíveis à maior parte da internet, apesar de estarem “ocultos bem diante de nossos olhos”. Sem recursos para tradução de imagens em texto, a única forma de deixar esses documentos acessíveis seria digitá-los manualmente – e ninguém tem tempo para isso.

Traduzir imagens para texto é um processo que se chama OCR (Optical Character Recognition, ou Reconhecimento Óptico de Caracteres). Algumas bibliotecas grandes são capazes de fazer OCR, e várias outras têm suporte para elas ou foram desenvolvidas com base nelas. Esse sistema de bibliotecas às vezes pode se tornar bem complicado, portanto recomendo que você leia a próxima seção antes de tentar fazer qualquer um dos exercícios deste capítulo.

## Visão geral das bibliotecas

Python é uma linguagem incrível para processamento e leitura de imagens, aprendizado de máquina baseado em imagens e até mesmo para a criação delas. Ainda que seja possível usar muitas bibliotecas para processamento de imagens, manteremos o foco em duas: o Pillow e o Tesseract.

Essas duas bibliotecas formam uma dupla complementar eficaz quando se trata de processamento e OCR em imagens da internet. O primeiro passo fica a cargo do *Pillow*, que limpa e filtra as imagens, enquanto o *Tesseract* procura fazer uma correspondência entre as formas encontradas nessas imagens e sua biblioteca de textos conhecidos.

Este capítulo descreve a instalação e o uso básico dessas bibliotecas, além de apresentar vários exemplos dessa dupla atuando em conjunto. Discutiremos também como fazer um treinamento avançado com o Tesseract, de modo que você o treine para fazer OCR de fontes e idiomas adicionais (ou até mesmo de CAPTCHAs) possíveis de serem encontradas na internet.

### Pillow

Apesar de o Pillow talvez não ser a mais completa das bibliotecas de processamento de imagens, ele tem todos os recursos de que você provavelmente precisará e mais alguns – a menos que esteja planejando reescrever o Photoshop em Python e, nesse caso, você estaria lendo o livro errado! O Pillow também tem a vantagem de ser uma das bibliotecas de terceiros mais bem documentada, e é muito simples de usar de imediato.

Tendo originado da PIL (Python Imaging Library, ou Biblioteca de Imagens de Python) para Python 2.x, o Pillow acrescenta suporte para Python 3.x. Assim como seu antecessor, o Pillow permite importar e manipular imagens com facilidade, e tem diversos filtros, máscaras e até mesmo

transformações específicas para pixels:

```
from PIL import Image, ImageFilter

kitten = Image.open('kitten.jpg')
blurryKitten = kitten.filter(ImageFilter.GaussianBlur)
blurryKitten.save('kitten_blurred.jpg')
blurryKitten.show()
```

No exemplo anterior, a imagem *kitten.jpg* será aberta em seu visualizador de imagens padrão, de forma desfocada, e também será salva nesse estado como *kitten\_blurred.jpg* no mesmo diretório.

Usaremos o Pillow para fazer um pré-processamento das imagens de modo a deixá-las mais legíveis para o computador, mas, conforme mencionamos antes, é possível fazer muito mais com a biblioteca além dessas aplicações simples de filtros. Para outras informações, consulte a [documentação do Pillow](http://pillow.readthedocs.org/) (<http://pillow.readthedocs.org/>).

## Tesseract

O Tesseract, uma biblioteca de OCR patrocinada pelo Google (uma empresa obviamente muito conhecida por suas tecnologias de OCR e de aprendizado de máquina), é considerado por muitos o melhor e mais preciso sistema de OCR de código aberto à disposição.

Além de ser preciso, o Tesseract é extremamente flexível. Pode ser treinado para reconhecer qualquer quantidade de fontes (desde que elas sejam relativamente consistentes entre si, como veremos em breve), e pode ser expandido para reconhecer qualquer caractere Unicode.

Este capítulo usa tanto o programa de linha de comando *Tesseract* como o seu wrapper de terceiros para Python, o *pytesseract*. Ambos serão explicitamente nomeados desse modo, portanto, saiba que, quando “Tesseract” for usado, estarei me referindo ao software da linha de comando, e, quando usar “pytesseract”, estarei falando especificamente do wrapper de terceiros para Python.

### Instalando o Tesseract

Para os usuários de Windows, há um [instalador executável](https://code.google.com/p/tesseract-ocr/downloads/list) (<https://code.google.com/p/tesseract-ocr/downloads/list>) conveniente. Atualmente (quando este livro foi escrito), a versão é a 3.02, embora versões mais recentes também deverão ser apropriadas.

Usuários de Linux podem instalar o Tesseract com o `apt-get`:

```
$ sudo apt-get tesseract-ocr
```

A instalação do Tesseract em um Mac é um pouco mais complicada, embora possa ser feita com facilidade com um dos vários instaladores de terceiros, como o [Homebrew](http://brew.sh/) (<http://brew.sh/>), que usamos no *Capítulo 6* para instalar o MySQL. Por exemplo, podemos instalar o Homebrew e usá-lo para instalar o Tesseract com duas linhas:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/ \
install/master/install)"
$ brew install tesseract
```

O Tesseract também pode ser instalado a partir do código-fonte, que está na [página de download do projeto](https://code.google.com/p/tesseract-ocr/downloads/list) (<https://code.google.com/p/tesseract-ocr/downloads/list>).

Para usar alguns recursos do Tesseract, como treinamento do software para reconhecer novos caracteres, que veremos mais adiante nesta seção, é necessário definir também uma nova variável de ambiente, `TESSDATA_PREFIX`, para que ele saiba em que local os arquivos de dados estão armazenados.

Podemos fazer isso na maioria dos sistemas Linux e no macOS da seguinte maneira:

```
$ export TESSDATA_PREFIX=/usr/local/share/
```

Observe que `/usr/local/share/` é o local default para os dados do Tesseract, embora você deva verificar se isso vale para a sua instalação.

De modo semelhante, no Windows, o comando a seguir pode ser usado para definir a variável de ambiente:

```
# setx TESSDATA_PREFIX C:\Program Files\Tesseract OCR\
```

## **pytesseract**

Depois que o Tesseract estiver instalado, estaremos prontos para instalar a biblioteca wrapper para Python, *pytesseract*, que usa a sua instalação de Tesseract para ler arquivos de imagens e gerar strings e objetos que poderão ser utilizados em scripts Python.

Os códigos de exemplo exigem o *pytesseract* 0.1.9

Saiba que algumas mudanças significativas ocorreram (com contribuições da autora) entre as versões 0.1.8 e 0.1.9 do *pytesseract*. Esta seção inclui recursos que se encontram somente na versão 0.1.9 da biblioteca. Por favor, certifique-se de instalar a versão correta quando executar os códigos de exemplo deste capítulo.

Como sempre, podemos instalar o *pytesseract* usando pip, ou fazendo um download da [página do projeto pytesseract](https://pypi.org/project/pytesseract/) (<https://pypi.org/project/pytesseract/>) e executando o seguinte:

```
$ python setup.py install
```

O `pytesseract` pode ser usado em conjunto com a `PIL` para ler texto de imagens:

```
from PIL import Image
import pytesseract
```

```
print(pytesseract.image_to_string(Image.open('files/test.png')))
```

Se a biblioteca *Tesseract* estiver instalada em seu `path` Python, é possível apontar o local ao `pytesseract` incluindo esta linha:

```
pytesseract.pytesseract.tesseract_cmd = '/path/to/tesseract'
```

O *pytesseract* tem vários recursos úteis, além de devolver os resultados de OCR para uma imagem, como no código do exemplo anterior. Ele é capaz de estimar arquivos de caixa (box files) contendo as localizações dos pixels nas fronteiras de cada caractere:

```
print(pytesseract.image_to_boxes(Image.open('files/test.png')))
```

Também é capaz de devolver uma saída completa com todos os dados, por exemplo, valores de confiança, números de páginas e linhas, dados de caixa, além de outras informações:

```
print(pytesseract.image_to_data(Image.open('files/test.png')))
```

A saída default para os dois últimos arquivos são arquivos de strings delimitadas com espaço ou tabulação, mas o resultado também pode ser obtido na forma de dicionários (se a decodificação em UTF-8 não for suficiente) ou cadeias de bytes:

```
from PIL import Image
import pytesseract
from pytesseract import Output
```

```
print(pytesseract.image_to_data(Image.open('files/test.png'),
    output_type=Output.DICT))
print(pytesseract.image_to_string(Image.open('files/test.png'),
    output_type=Output.BYTES))
```

Este capítulo usa uma combinação da biblioteca *pytesseract*, assim como do *Tesseract* para linha de comando, e o disparo do *Tesseract* a partir de Python usando a biblioteca `subprocess`. Apesar de a biblioteca `pytesseract` ser útil e conveniente, há algumas funcionalidades do *Tesseract* que ela não inclui, portanto é bom ter familiaridade com todos os métodos.

## NumPy

Embora o `NumPy` não seja necessário em um OCR simples, você precisará dele se quiser treinar o *Tesseract* para que reconheça conjuntos de

caracteres ou fontes adicionais, introduzidos mais adiante neste capítulo. Também usaremos o NumPy para tarefas matemáticas simples (por exemplo, cálculos de médias ponderadas) em alguns dos exemplos de código mais tarde.

O NumPy é uma biblioteca eficaz, usada para álgebra linear e outras aplicações matemáticas de larga escala. Funciona bem com o Tesseract em razão de sua capacidade de representar e manipular as imagens matematicamente, na forma de arrays grandes de pixels.

O NumPy pode ser instalado com qualquer instalador Python de terceiros, como o pip, ou podemos fazer [download do pacote](https://pypi.org/project/numpy/) (<https://pypi.org/project/numpy/>) e instalá-lo com `$ python setup.py install`.

Mesmo que você não pretenda executar nenhum dos exemplos de código que o utilizam, recomendo instalá-lo ou acrescentá-lo ao seu arsenal Python. Ele serve para complementar a biblioteca matemática embutida de Python, e tem muitos recursos úteis, sobretudo para operações com listas de números.

Por convenção, o NumPy é importado como `np`, e pode ser usado assim:

```
import numpy as np

numbers = [100, 102, 98, 97, 103]
print(np.std(numbers))
print(np.mean(numbers))
```

Esse exemplo exibe o desvio-padrão e a média do conjunto de números fornecido a ele.

## Processando textos bem formatados

Com um pouco de sorte, a maior parte do texto que teremos de processar estará relativamente limpa e bem formatada. Na maioria das vezes, textos bem formatados atendem a vários requisitos, embora a linha entre ser “confuso” e estar “bem formatado” seja subjetiva.

De modo geral, um texto bem formatado:

- está escrito em uma fonte padrão (excluem fontes cursivas ou excessivamente decorativas);
- se for copiado ou fotografado, tem linhas extremamente nítidas, sem detalhes adicionados pela cópia ou pontos escuros;
- está bem alinhado, sem letras inclinadas;

- não avança para fora da imagem nem apresenta texto ou margens cortados nas bordas.

Alguns desses problemas podem ser corrigidos durante o pré-processamento. Por exemplo, é possível converter imagens em escala de cinza, brilho e contraste podem ser ajustados e a imagem pode ser recortada e girada conforme necessário. No entanto, algumas limitações básicas podem exigir um treinamento mais intenso. Veja a seção “*Lendo CAPTCHAs e treinando o Tesseract*”.

A Figura 13.1 mostra um exemplo ideal de um texto bem formatado.

This is some text, written in Arial, that will be read by Tesseract. Here are some symbols: !@#\$%^&amp;\*()

*Figura 13.1 – Amostra de texto salvo como um arquivo .tiff, para ser lido pelo Tesseract.*

O Tesseract pode ser executado na linha de comando para que leia esse arquivo e escreva o resultado em um arquivo-texto:

```
$ tesseract text.tif textoutput | cat textoutput.txt
```

O resultado é uma linha com informações sobre a biblioteca *Tesseract* para informar que ela está executando, seguida do conteúdo do arquivo *textoutput.txt* recém-criado:

```
Tesseract Open Source OCR Engine v3.02.02 with Leptonica  
This is some text, written in Arial, that will be read by  
Tesseract. Here are some symbols: !@#$%"&'()
```

Podemos ver que o resultado, em sua maior parte, é exato, apesar de os símbolos `^` e `*` terem sido interpretados com um caractere de aspas duplas e aspas simples, respectivamente. De modo geral, porém, isso permitiu que o texto fosse lido com muita facilidade.

Depois de desfocar o texto da imagem, criar alguns artefatos de compactação JPG e adicionar um pequeno gradiente no plano de fundo, o resultado piora bastante (veja a Figura 13.2).

This is some text, written in Arial, that will be read by Tesseract. Here are some symbols: !@#\$%^&amp;\*()

*Figura 13.2 – Infelizmente, muitos dos documentos que encontraremos na*

*internet se parecerão mais com essa imagem do que com a imagem do exemplo anterior.*

O Tesseract não é capaz de lidar tão bem com essa imagem, principalmente por causa do gradiente do plano de fundo, e gera a saída a seguir:

```
This is some text, written In Arlal, that"
Tesseract. Here are some symbols: _
```

Note que há um corte no texto assim que o gradiente do plano de fundo dificulta distinguir o texto, e que o último caractere de cada linha está incorreto, pois o Tesseract tenta interpretá-lo sem sucesso. Além disso, os artefatos de JPG e a falta de nitidez fazem com que seja difícil para o Tesseract realizar a distinção entre a letra *i* minúscula, a letra *I* maiúscula e o número *1*.

É nesse caso que o uso de um script Python para limpar suas imagens antes pode ser conveniente. Usando a biblioteca Pillow, podemos criar um filtro de limiar (threshold filter) para nos livrarmos do cinza no plano de fundo, dar destaque ao texto e deixar a imagem mais clara para o Tesseract ler.

Além disso, em vez de usar o Tesseract na linha de comando, podemos usar a biblioteca pytesseract para executar os comandos do Tesseract e ler o arquivo resultante:

```
from PIL import Image
import pytesseract

def cleanFile(filePath, newFilePath):
    image = Image.open(filePath)

    # Define um valor de limiar para a imagem, e salva
    image = image.point(lambda x: 0 if x < 143 else 255)
    image.save(newFilePath)
    return image

image = cleanFile('files/textBad.png', 'files/textCleaned.png')

# chama o tesseract para fazer o OCR na imagem recém-criada
print(pytesseract.image_to_string(image))
```

A Figura 13.3 mostra a imagem resultante, criada como *textCleaned.png* de modo automático.

```
This is some text, written in Arial, that will be read by
Tesseract Here are some symbols: !@#$%^&*()
```

*Figura 13.3 – Essa imagem foi criada passando a versão anterior “confusa” da imagem por um filtro de limiar.*

Exceto por algum sinal de pontuação pouco legível ou faltando, o texto está legível, pelo menos para nós. O Tesseract oferece o seu melhor palpite:

```
This us some text' written In Anal, that will be read by  
Tesseract Here are some symbols: !@#$$%"&'()
```

Os pontos e as vírgulas, por serem muito pequenos, são as primeiras vítimas desse tratamento da imagem e quase desapareceram, tanto de nossas vistas como para o Tesseract. Há também a infeliz interpretação de “Arial” como “Anal”, resultante de o Tesseract ter interpretado o *r* e o *i* como um único caractere *n*.

Apesar disso, houve uma melhoria em relação à versão anterior, na qual quase metade do texto havia sido cortado.

O principal ponto fraco do Tesseract parecem ser os planos de fundo com brilho variado. Os algoritmos do Tesseract tentam ajustar o contraste da imagem de forma automática antes de ler o texto, mas é provável que você obtenha melhores resultados se fizer isso por conta própria, com uma ferramenta como a biblioteca Pillow.

As imagens que certamente você deverá corrigir antes de submeter ao Tesseract são as que estiverem inclinadas, tiverem grandes áreas que contenham algo diferente de texto ou que apresentarem outros problemas.

### **Ajustes automáticos nas imagens**

No exemplo anterior, o valor 143 foi escolhido de forma experimental como o limiar “ideal” para ajustar todos os pixels da imagem para preto ou branco, de modo que o Tesseract consiga ler a imagem. O que aconteceria se tivéssemos muitas imagens, todas com problemas de escala de cinza um pouco diferentes, e não fosse possível ajustá-las de modo razoável manualmente?

Uma maneira de encontrar a melhor solução (ou, pelo menos, uma solução muito boa) é executar o Tesseract em várias imagens ajustadas com valores diferentes e usar um algoritmo para selecionar aquela com o melhor resultado, conforme avaliado por alguma combinação entre o número de caracteres e/ou strings que o Tesseract é capaz de ler e o nível de “confiança” com que esses caracteres foram lidos.

O algoritmo exato a ser usado pode variar um pouco de aplicação para

aplicação, mas o que vemos a seguir é um exemplo de como iterar por limiares no processamento de imagens a fim de descobrir a “melhor” configuração:

```
import pytesseract
from pytesseract import Output
from PIL import Image
import numpy as np

def cleanFile(filePath, threshold):
    image = Image.open(filePath)
    # Define um valor de limiar para a imagem, e salva
    image = image.point(lambda x: 0 if x < threshold else 255)
    return image

def getConfidence(image):
    data = pytesseract.image_to_data(image, output_type=Output.DICT)
    text = data['text']
    confidences = []
    numChars = []

    for i in range(len(text)):
        if data['conf'][i] > -1:
            confidences.append(data['conf'][i])
            numChars.append(len(text[i]))

    return np.average(confidences, weights=numChars, sum(numChars))

filePath = 'files/textBad.png'

start = 80
step = 5
end = 200

for threshold in range(start, end, step):
    image = cleanFile(filePath, threshold)
    scores = getConfidence(image)
    print("threshold: " + str(threshold) + ", confidence: "
          + str(scores[0]) + " numChars " + str(scores[1]))
```

Esse script tem duas funções:

### **cleanFile**

Recebe um arquivo original “ruim” e uma variável de limiar com os quais executa a ferramenta de limiar da PIL. Processa o arquivo e devolve o objeto de imagem da PIL.

### **getConfidence**

Recebe o objeto de imagem limpo da PIL e o submete ao Tesseract. Calcula o nível de confiança médio para cada string reconhecida

(ponderado pelo número de caracteres dessa string), assim como o número de caracteres reconhecidos.

Variando o valor do limiar e obtendo o nível de confiança e a quantidade de caracteres reconhecidos para cada valor, teremos a seguinte saída:

```
threshold: 80, confidence: 61.8333333333 numChars 18
threshold: 85, confidence: 64.9130434783 numChars 23
threshold: 90, confidence: 62.2564102564 numChars 39
threshold: 95, confidence: 64.5135135135 numChars 37
threshold: 100, confidence: 60.7878787879 numChars 66
threshold: 105, confidence: 61.9078947368 numChars 76
threshold: 110, confidence: 64.6329113924 numChars 79
threshold: 115, confidence: 69.7397260274 numChars 73
threshold: 120, confidence: 72.9078947368 numChars 76
threshold: 125, confidence: 73.582278481 numChars 79
threshold: 130, confidence: 75.6708860759 numChars 79
threshold: 135, confidence: 76.8292682927 numChars 82
threshold: 140, confidence: 72.1686746988 numChars 83
threshold: 145, confidence: 75.5662650602 numChars 83
threshold: 150, confidence: 77.5443037975 numChars 79
threshold: 155, confidence: 79.1066666667 numChars 75
threshold: 160, confidence: 78.4666666667 numChars 75
threshold: 165, confidence: 80.1428571429 numChars 70
threshold: 170, confidence: 78.4285714286 numChars 70
threshold: 175, confidence: 76.3731343284 numChars 67
threshold: 180, confidence: 76.7575757576 numChars 66
threshold: 185, confidence: 79.4920634921 numChars 63
threshold: 190, confidence: 76.0793650794 numChars 63
threshold: 195, confidence: 70.6153846154 numChars 65
```

Há uma tendência clara tanto para o nível médio de confiança no resultado como para o número de caracteres reconhecidos. Ambos tendem a ter um pico em torno de um limiar igual a 145, que é próximo do resultado “ideal” de 143, encontrado manualmente.

Os limiares 140 e 145 proporcionam o número máximo de caracteres reconhecidos (83), mas um limiar de 145 resulta no nível de confiança mais alto para os caracteres encontrados, portanto talvez você queira optar por esse resultado e devolver o texto reconhecido com esse limiar como o “melhor palpite” para o texto contido na imagem.

É claro que apenas encontrar a “maioria” dos caracteres não significa necessariamente que todos esses caracteres sejam reais. Com alguns limiares, o Tesseract poderia separar caracteres únicos em vários, ou interpretar um ruído aleatório na imagem como um caractere de texto que na verdade não existe. Nesse caso, talvez você queira levar mais em

consideração o nível médio de confiança de cada valor.

Por exemplo, se encontrar um resultado em que se lê (em parte) o seguinte:

```
threshold: 145, confidence: 75.5662650602 numChars 83
```

```
threshold: 150, confidence: 97.1234567890 numChars 82
```

é provável que não precisássemos pensar duas vezes para optar pelo resultado que oferece mais de 20% de confiança adicional, com perda de apenas um caractere, e supor que o resultado com um limiar de 145 estava simplesmente incorreto ou, quem sabe, houve um caractere separado ou algo que não estava presente foi encontrado.

Em casos como esse, um pouco de experimentos prévios para aperfeiçoar o algoritmo de seleção de limiar poderá ser conveniente. Por exemplo, talvez você queira selecionar o valor para a qual o *produto* entre o nível de confiança e o número de caracteres seja maximizado (nesse caso, 145 continuaria vencendo, com um produto igual a 6272; em nosso exemplo imaginário, o limiar de 150 venceria, com um produto igual a 7964), ou decida usar alguma outra métrica.

Observe que esse tipo de algoritmo de seleção também funciona com outros valores arbitrários da ferramenta PIL diferentes do `threshold`. Além do mais, podemos usá-lo para selecionar dois ou mais valores, variando-os e selecionando o resultado com a melhor pontuação, de modo semelhante.

Obviamente, esse tipo de algoritmo de seleção exige um processamento intenso. Executaremos tanto a PIL como o Tesseract muitas vezes em cada imagem, enquanto, se conhecêssemos previamente os valores “ideais” para os limiares, seria necessário executá-los apenas uma vez.

Tenha em mente que, à medida que começar a trabalhar com imagens a serem processadas, você poderá começar a perceber padrões nos valores “ideais” encontrados. Em vez de tentar todos os limiares de 80 a 200, sendo realista, talvez fosse necessário testar apenas limiares de 130 a 180.

Você poderia até mesmo adotar outra abordagem e escolher limiares, por exemplo, com um intervalo de 20 na primeira execução, e então usar um algoritmo guloso (greedy algorithm) para aprimorar o melhor resultado, decrementando o tamanho de seu passo para limiares entre as “melhores” soluções encontradas na iteração anterior. Isso também pode funcionar melhor se estivermos lidando com diversas variáveis.

## **Coletando texto de imagens em sites**

Usar o Tesseract para ler texto de uma imagem em seu disco rígido, embora não pareça tão empolgante, pode ser uma ferramenta eficaz se usado com um web scraper. Imagens podem inadvertidamente ofuscar um texto em sites (por exemplo, uma cópia JPG do cardápio no site de um restaurante local), mas também podem ocultar um texto de propósito, como será mostrado no próximo exemplo.

Embora o arquivo *robots.txt* da Amazon permita fazer scraping das páginas de produto do site, visualizações prévias de livros em geral não são obtidas por bots que estejam de passagem. Isso ocorre porque essas visualizações prévias são carregadas por meio de scripts Ajax disparados pelos usuários, e as imagens são cuidadosamente ocultadas sob camadas de divs. Para visitantes comuns do site, é provável que essas visualizações se pareçam mais como apresentações Flash do que com arquivos de imagens. É claro que, mesmo que fosse possível obter as imagens, há ainda o problema não tão pequeno de lê-las como texto.

O script a seguir faz essa proeza: acessa a edição com fontes grandes<sup>1</sup> do livro *The Death of Ivan Ilyich* (A morte de Ivan Illich) de Tolstói, abre a ferramenta de leitura, coleta URLs das imagens e então, sistematicamente, faz download, lê e exibe o texto de cada um deles.

Observe que esse código depende de uma exibição ao vivo da Amazon, assim como de diversos recursos de arquitetura desse site estarem executando de forma correta. Se essa exibição for desativada ou substituída, sinta-se à vontade para trocar pelo URL de outro livro com recurso de Visualização Prévia (acho que fontes sans-serif grandes funcionam bem).

Por ser um código relativamente complexo, que se baseia em vários conceitos dos capítulos anteriores, acrescentei comentários para que fosse um pouco mais fácil entender o que está acontecendo:

```
import time
from urllib.request import urlretrieve
from PIL import Image
import tesseract
from selenium import webdriver

def getImageText(imageUrl):
    urlretrieve(image, 'page.jpg')
    p = subprocess.Popen(['tesseract', 'page.jpg', 'page'],
        stdout=subprocess.PIPE,stderr=subprocess.PIPE)
    p.wait()
```

```

f = open('page.txt', 'r')
print(f.read())

# Cria um novo driver do Selenium
driver = webdriver.Chrome(executable_path='<Path to chromedriver>')

driver.get('https://www.amazon.com/Death-Ivan-Ilyich\'
-Nikolayevich-Tolstoy/dp/1427027277')
time.sleep(2)

# Clica no botão para visualização prévia do livro
driver.find_element_by_id('imgBlkFront').click()
imageList = []

# Espera a página ser carregada
time.sleep(5)

while 'pointer' in driver.find_element_by_id(
'sitbReaderRightPageTurner').get_attribute('style'):
# Enquanto a seta da direita estiver disponível para clicar, vira as páginas
driver.find_element_by_id('sitbReaderRightPageTurner').click()
time.sleep(2)
# Obtém qualquer página nova carregada (várias páginas podem ser
# carregadas de uma só vez, mas páginas duplicadas não são
# adicionadas em um conjunto)
pages = driver.find_elements_by_xpath('///div[@class=\'pageImage\']/div/img')
if not len(pages):
print("No pages found")
for page in pages:
image = page.get_attribute('src')
print('Found image: {}'.format(image))
if image not in imageList:
imageList.append(image)
getImageText(image)

driver.quit()

```

Apesar de esse script teoricamente poder ser executado com qualquer tipo de webdriver Selenium, percebi que, no momento, ele é mais confiável com o Chrome.

Como já vimos antes com a ferramenta de leitura do Tesseract, esse código exibe várias passagens longas do livro que, em sua maior parte, são legíveis, como vemos na prévia do primeiro capítulo:

Chapter I

During an Interval In the Melvmskl trial In the large building of the Law Courts the members and public prosecutor met in [van Egorowch Shebek's private room, where the conversation turned on the celebrated Krasovski case. Fedor Vasillevich warmly maintained

that it was not subject to their jurisdiction, Ivan Egorovich maintained the contrary, while Peter ivanowch, not havmg entered into the discussmn at the start, took no part in it but looked through the Gazette which had Just been handed in.

"Gentlemen," he said, "Ivan Ilych has died!"

No entanto, muitas palavras têm erros óbvios, como “Melvmskl” no lugar do nome “Melvinski” e “discusmn” em vez de “discussion”. Vários erros desse tipo podem ser corrigidos com palpites baseados em uma lista de palavras de dicionário (talvez com acréscimos de nomes próprios relevantes como “Melvinski”).

Ocasionalmente, um erro pode se estender por uma palavra inteira, como na página 3 do texto:

it is he who is dead and not 1.

Nesse exemplo, a palavra “I” foi substituída pelo caractere “1”. Uma análise de cadeia de Markov poderia ser útil nesse caso, além do acréscimo de um dicionário de palavras. Se alguma parte do texto contiver uma expressão muito incomum (“and not 1”), poderíamos supor que o texto, na verdade, será a expressão mais comum (“and not I”).

É claro que o fato de essas substituições de caractere seguirem padrões previsíveis ajuda: “vi” torna-se “w” e “I” torna-se “1”. Se essas substituições ocorrerem com frequência em seu texto, você poderia criar uma lista delas e usá-las para “testar” novas palavras e expressões, escolhendo a solução que fizer mais sentido. Uma abordagem seria substituir caracteres confundidos com frequência, e usar uma solução que faça a correspondência com uma palavra do dicionário ou com um n-grama reconhecido (ou mais comum).

Se você adotar essa abordagem, não se esqueça de ler o *Capítulo 9*, que contém outras informações sobre como trabalhar com textos e fazer processamento de idiomas naturais.

Embora o texto nesse exemplo tenha uma fonte sans-serif comum e o Tesseract deva ser capaz de reconhecê-la com relativa facilidade, às vezes um novo treinamento também pode ajudar a melhorar a precisão. A próxima seção discute outra abordagem para resolver o problema de um texto deturpado, exigindo um pequeno investimento prévio de tempo.

Se fornecermos ao Tesseract uma coleção grande de imagens de texto com valores conhecidos, ele poderá ser “ensinado” a reconhecer a mesma fonte

no futuro, com muito mais minúcia e precisão, mesmo que haja problemas ocasionais com o plano de fundo e com o posicionamento no texto.

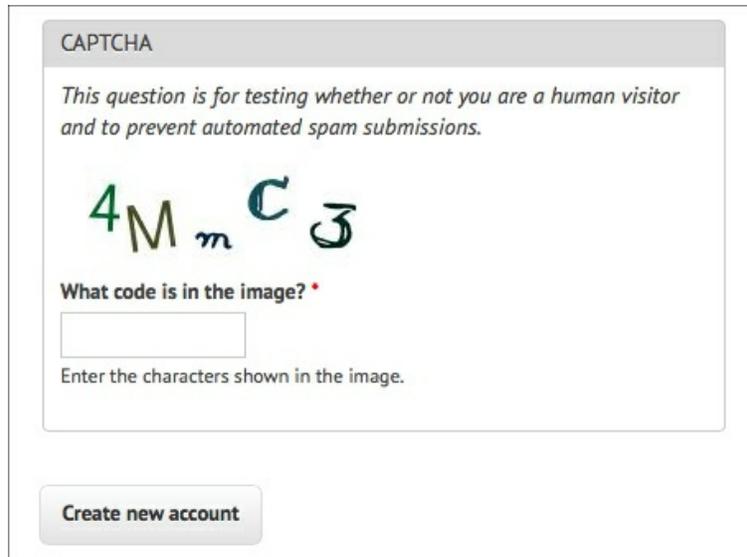
## Lendo CAPTCHAs e treinando o Tesseract

Embora a palavra *CAPTCHA* seja conhecida da maioria, poucas pessoas sabem o que ela significa: *Completely Automated Public Turing Test to Tell Computers and Humans Apart* (Teste de Turing Público Completamente Automatizado para Diferenciação entre Computadores e Humanos). Seu acrônimo inusitado dá uma pista de sua função para obstruir interfaces web que, do contrário, seriam perfeitamente utilizáveis, uma vez que tanto os seres humanos quanto os robôs muitas vezes têm dificuldades para resolver testes CAPTCHA.

O teste de Turing foi descrito pela primeira vez por Alan Turing em seu artigo “Computing Machinery and Intelligence” (Computação e inteligência) de 1950. No artigo, ele descreve uma configuração em que um ser humano poderia se comunicar tanto com outros seres humanos como com programas de inteligência artificial por meio de um terminal de computador. Se o ser humano não fosse capaz de fazer a distinção entre seres humanos e os programas de IA durante uma conversa casual, os programas de IA seriam considerados como aprovados pelo teste de Turing, e a inteligência artificial, Turing raciocinava, estaria genuinamente “pensando” em todas as intenções e propósitos.

O fato de que, nos últimos 60 anos, esses testes usados para testar máquinas passaram a ser usados para testar a nós mesmos, com resultados variados, é irônico. Recentemente, o Google acabou com seu renomado reCAPTCHA, que era difícil, e o motivo principal era a sua tendência de bloquear usuários legítimos do site<sup>2</sup>.

A maioria dos outros CAPTCHAs é um pouco mais simples. O Drupal, um sistema popular de gerenciamento de conteúdo baseado em PHP, por exemplo, tem um [módulo CAPTCHA](https://www.drupal.org/project/captcha) (<https://www.drupal.org/project/captcha>) conhecido, capaz de gerar imagens CAPTCHA com dificuldade variada. A imagem default tem a aparência exibida na Figura 13.4.



*Figura 13.4 – Um exemplo do texto CAPTCHA default do projeto CAPTCHA do Drupal.*

O que faz com que esse CAPTCHA seja tão fácil de ler, tanto para os seres humanos como para as máquinas, em comparação com outros CAPTCHAs?

- Os caracteres não se sobrepõem nem cruzam o espaço um do outro horizontalmente. Isso significa que é possível desenhar um retângulo bem claro em torno de cada caractere sem que esse se sobreponha a outros caracteres.
- Não há imagens, linhas nem outros lixos que desviem a atenção no plano de fundo, que pudessem confundir um programa de OCR.
- Não é óbvio nessa imagem, mas há algumas variações na fonte usada pelo CAPTCHA. Ela alterna entre uma fonte sans-serif clara (como vemos nos caracteres “4” e “M”) e uma fonte com estilo cursivo (como vemos nos caracteres “m” “C” e “3”).
- Há um alto nível de contraste entre o plano de fundo branco e os caracteres com cores escuras.

Esse CAPTCHA, porém, faz uso de algumas curvas, o que o torna desafiador para os programas de OCR lerem:

- Tanto letras como números são usados, aumentando o número de caracteres possíveis.
- A inclinação aleatória das letras pode confundir um software de OCR, mas, para os seres humanos, continua sendo fácil ler.

- A fonte cursiva, de certo modo estranha, apresenta desafios específicos, com linhas extras no “C” e no “3” e uma letra “m” excepcionalmente pequena, exigindo um treinamento adicional para que os computadores possam compreender.

Ao executar o Tesseract nessa imagem com o comando:

```
$ tesseract captchaExample.png output
```

o arquivo *output.txt* a seguir é obtido:

```
4N\,, ,C<3
```

Ele leu 4, C e 3 da forma correta, mas, obviamente, não será capaz de preencher um campo protegido por CAPTCHA tão cedo.

## Treinando o Tesseract

Para treinar o Tesseract de modo que ele reconheça uma escrita, seja de uma fonte obscura e difícil de ler seja de um CAPTCHA, é necessário lhe dar vários exemplos de cada caractere.

É nessa parte que você vai querer ouvir um bom podcast ou assistir a um filme porque serão algumas horas de trabalho um tanto quanto tedioso. O primeiro passo é fazer download de vários exemplos de seu CAPTCHA em um único diretório. O número de exemplos que você reunir dependerá da complexidade do CAPTCHA; usei uma amostra com 100 arquivos (um total de 500 caracteres, ou aproximadamente 8 exemplos por símbolo, na média) para o treinamento de meu CAPTCHA, e isso parece ter funcionado muito bem.

Recomendo nomear a imagem com base na solução do CAPTCHA que ele representa (por exemplo, *4MmC3.jpg*). Percebi que isso ajuda a fazer uma verificação rápida de erro em um grande número de arquivos de uma só vez; você pode visualizar todos os arquivos como miniaturas e comparar facilmente as imagens com seus nomes. Além do mais, isso ajudará bastante na verificação de erros nos passos subsequentes.

O segundo passo é dizer ao Tesseract o que é exatamente cada caractere e onde ele está na imagem. Esse passo envolve criar arquivos de caixa (box files), um para cada imagem CAPTCHA. Um arquivo de caixa tem o seguinte aspecto:

```
4 15 26 33 55 0
M 38 13 67 45 0
m 79 15 101 26 0
C 111 33 136 60 0
3 147 17 176 45 0
```

O primeiro símbolo é o caractere representado, os quatro números

seguintes representam coordenadas para uma caixa retangular que contorna a imagem, e o último número é um “número de página” usado para treinamento com documentos de várias páginas (0 para nós).

É claro que não é divertido criar esses arquivos de caixa manualmente, mas diversas ferramentas podem ajudar. Gosto da ferramenta online [Tesseract OCR Chopper](https://pp19dd.com/tesseract-ocr-chopper/) (<https://pp19dd.com/tesseract-ocr-chopper/>) porque ela não exige instalação nem bibliotecas adicionais, executa em qualquer máquina que tenha um navegador e é relativamente fácil de usar. Faça o upload da imagem, clique no botão Add (Adicionar) na parte inferior se precisar de caixas adicionais, ajuste o tamanho das caixas se necessário e copie e cole o texto em um novo arquivo *.box*.

Os arquivos de caixa devem ser salvos em formato texto simples, com a extensão *.box*. Como no caso dos arquivos de imagem, é conveniente nomear os arquivos de caixa de acordo com as soluções dos CAPTCHAs que representam (por exemplo, *4MmC3.box*). Mais uma vez, isso facilita fazer uma comparação entre o conteúdo do arquivo *.box* e o nome do arquivo e, novamente, com o arquivo de imagem associado se você ordenar todos os arquivos de seu diretório de dados com base em seus nomes.

Você deverá criar cerca de 100 desses arquivos para garantir que terá dados suficientes. Além disso, o Tesseract ocasionalmente descartará arquivos por estarem ilegíveis, portanto talvez você queira ter um pouco de folga quanto a esse número. Se achar que seus resultados de OCR não são tão bons quanto gostaria, ou o Tesseract está tendo dificuldades com determinados caracteres, criar dados de treinamento adicionais e tentar novamente é um bom passo para depuração.

Depois de criar uma pasta de dados cheia de arquivos *.box* e arquivos de imagens, copie esses dados para uma pasta de backup antes de fazer novas manipulações. Embora seja pouco provável que executar scripts de treinamento nos dados vá apagar alguma informação, é melhor prevenir do que remediar quando horas de trabalho investidas na criação de arquivos *.box* estão em jogo. Além do mais, é bom poder se desfazer de um diretório confuso, cheio de dados, e tentar novamente.

Há meia dúzia de passos para fazer todas as análises de dados e criar os arquivos de treinamento necessários ao Tesseract. Algumas ferramentas fazem isso se você lhes fornecer a imagem original e os arquivos *.box* correspondentes, mas nenhuma delas, até agora, funciona para o Tesseract

3.02, infelizmente.

Escrevi *uma solução em Python* (<https://github.com/REMitchell/tesseract-trainer>) que atua em um diretório contendo tanto os arquivos de imagens quanto os arquivos de caixa; todos os arquivos de treinamento necessários serão criados de modo automático.

As configurações iniciais e os passos executados por esse programa podem ser vistos nos métodos `__init__` e `runAll` da classe:

```
def __init__(self):
    languageName = 'eng'
    fontName = 'captchaFont'
    directory = '<path to images>'

def runAll(self):
    self.createFontFile()
    self.cleanImages()
    self.renameFiles()
    self.extractUnicode()
    self.runShapeClustering()
    self.runMfTraining()
    self.runCnTraining()
    self.createTessData()
```

As três únicas variáveis que você deverá definir nesse programa são bem simples:

### **languageName**

É o código de três letras que o Tesseract usa para saber qual é o idioma que está vendo. Na maioria das vezes, é provável que você queira usar `eng` para inglês.

### **fontName**

É o nome da fonte escolhida. Pode ter qualquer valor, mas deve ser uma palavra única, sem espaços.

### **directory**

É o diretório que contém todos os seus arquivos de imagens e os arquivos de caixa. Recomendo que seja um path absoluto, mas, se usar um path relativo, esse deverá ser relativo ao local em que você estiver executando o seu código Python. Se for um path absoluto, o código poderá ser executado de qualquer lugar em seu computador.

Vamos analisar as funções individuais usadas.

`createFontFile` cria um arquivo necessário, `font_properties`, que permite que

o Tesseract saiba qual é a nova fonte que você está criando:

```
captchaFont 0 0 0 0 0
```

Esse arquivo é composto do nome da fonte, seguido de 1s e 0s que indicam se itálico, negrito ou outras versões da fonte devem ser consideradas. (Treinar fontes com essas propriedades é um exercício interessante, mas, infelizmente, está além do escopo deste livro.)

`cleanImages` cria versões com mais contraste de todos os arquivos de imagem encontrados, converte-os para escalas de cinza e executa outras operações que deixam os arquivos de imagem mais fáceis de ler para os programas de OCR. Se você estiver lidando com imagens CAPTCHA com lixo visual que possa ser mais facilmente filtrado no pós-processamento, este seria o local para acrescentar esse processamento adicional.

`renameFiles` renomeia todos os seus arquivos `.box` e os arquivos de imagem correspondentes com os nomes exigidos pelo Tesseract (os números dos arquivos, nesse caso, são dígitos sequenciais para manter os vários arquivos separados):

- `<nomeDoIdioma>.<nomeDaFonte>.exp<númeroDoArquivo>.box`
- `<nomeDoIdioma>.<nomeDaFonte>.exp<númeroDoArquivo>.tiff`

`extractUnicode` olha para todos os arquivos `.box` criados e determina o conjunto total de caracteres disponíveis para treinamento. O arquivo Unicode resultante informará quantos caracteres diferentes foram encontrados, e poderia ser uma ótima maneira de ver rapidamente se está faltando algo.

As três próximas funções, `runShapeClustering`, `runMfTraining` e `runCtTraining`, criam os arquivos `shapetable`, `pfhtable` e `normproto`, respectivamente. Todas elas fornecem informações sobre a geometria e o formato de cada caractere, bem como informações estatísticas usadas pelo Tesseract para calcular a probabilidade de um dado caractere ser de um tipo ou de outro.

Por fim, o Tesseract renomeia cada uma das pastas com dados reunidos para que sejam prefixadas com o nome do idioma necessário (por exemplo, `shapetable` é renomeado para `eng.shapetable`) e processa todos esses arquivos gerando o arquivo de dados final de treinamento, `eng.traineddata`.

O único passo que você deve executar manualmente é mover o arquivo `eng.traineddata` criado para a sua pasta-raiz `tessdata` usando os comandos a seguir no Linux ou no Mac:

```
$cp /path/to/data/eng.traineddata $TESSDATA_PREFIX/tessdata
```

Depois desses passos, você não deverá ter problemas para solucionar os CAPTCHAs de cujo tipo o Tesseract foi treinado. A partir de agora, quando pedir ao Tesseract que leia a imagem de exemplo, a resposta correta será obtida:

```
$ tesseract captchaExample.png output|cat output.txt
4MmC3
```

Sucesso! Uma melhoria significativa em relação à interpretação anterior da imagem como 4N\,, ,c<3.

Essa é apenas uma visão geral rápida de todo o potencial do treinamento de fontes e dos recursos de reconhecimento do Tesseract. Se estiver interessado em treinar intensivamente o Tesseract, quem sabe dando início à sua própria biblioteca de arquivos de treinamento de CAPTCHAs, ou compartilhando recursos de reconhecimento de novas fontes com o mundo, recomendo que consulte a [documentação \(https://github.com/tesseract-ocr/\)](https://github.com/tesseract-ocr/).

## Lendo CAPTCHAs e enviando soluções

Muitos sistemas populares de gerenciamento de conteúdo são alvos frequentes de inscrições spams feitas por bots pré-programados com o endereço bem conhecido dessas páginas de inscrição de usuários. Em <http://pythonscraping.com>, por exemplo, até mesmo um CAPTCHA (devo admitir que é um CAPTCHA frágil) não faz muito para reduzir o fluxo contínuo de inscrições.

Como esses bots fazem isso? Fomos bem-sucedidos para resolver os CAPTCHAs em imagens que estavam em nosso disco rígido, mas como criamos um bot totalmente funcional? Esta seção consolida as várias técnicas abordadas nos capítulos anteriores. Se você ainda não leu o *Capítulo 10*, pelo menos passe os olhos nele rapidamente.

A maioria dos CAPTCHAs baseados em imagens tem várias propriedades:

- São imagens geradas de modo dinâmico, criadas por um programa do lado do servidor. Podem ter imagens cujas fontes não se pareçam com imagens tradicionais, como ``, mas elas podem ser baixadas e manipuladas como qualquer outra imagem.
- A solução para a imagem está armazenada em um banco de dados no servidor.
- Muitos CAPTCHAs expiram se você demorar muito para solucioná-los.

Em geral, isso não é um problema para os bots, mas guardar soluções de CAPTCHA para usar depois, ou utilizar outras práticas que causem demoras entre o instante em que o CAPTCHA foi requisitado e a solução é enviada, podem significar uma falha.

A abordagem genérica consiste em fazer download do arquivo de imagem do CAPTCHA em seu disco rígido, limpá-lo, usar o Tesseract para fazer parse da imagem e devolver a solução no parâmetro apropriado do formulário.

Criei uma página em <http://pythonscraping.com/humans-only> com um formulário de comentário protegido por CAPTCHA, com o intuito de escrever um bot para derrotá-lo. O bot usa a biblioteca de linha de comando Tesseract, em vez do wrapper pytesseract (embora pudesse usar qualquer um deles facilmente), e tem o seguinte aspecto:

```
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup
import subprocess
import requests
from PIL import Image
from PIL import ImageOps

def cleanImage(imagePath):
    image = Image.open(imagePath)
    image = image.point(lambda x: 0 if x<143 else 255)
    borderImage = ImageOps.expand(image,border=20,fill='white')
    borderImage.save(imagePath)

html = urlopen('http://www.pythonscraping.com/humans-only')
bs = BeautifulSoup(html, 'html.parser')
# Coleta valores do formulário previamente preenchidos
imageLocation = bs.find('img', {'title': 'Image CAPTCHA'})['src']
formBuildId = bs.find('input', {'name':'form_build_id'})['value']
captchaSid = bs.find('input', {'name':'captcha_sid'})['value']
captchaToken = bs.find('input', {'name':'captcha_token'})['value']

captchaUrl = 'http://pythonscraping.com'+imageLocation
urlretrieve(captchaUrl, 'captcha.jpg')
cleanImage('captcha.jpg')
p = subprocess.Popen(['tesseract', 'captcha.jpg', 'captcha'], stdout=
    subprocess.PIPE,stderr=subprocess.PIPE)
p.wait()
f = open('captcha.txt', 'r')

# Limpa qualquer caractere de espaço em branco
captchaResponse = f.read().replace(' ', '').replace('\n', '')
```

```

print('Captcha solution attempt: '+captchaResponse)

if len(captchaResponse) == 5:
    params = {'captcha_token':captchaToken, 'captcha_sid':captchaSid,
              'form_id':'comment_node_page_form', 'form_build_id': formBuildId,
              'captcha_response':captchaResponse, 'name':'Ryan Mitchell',
              'subject': 'I come to seek the Grail',
              'comment_body[und][0][value]':
                '...and I am definitely not a bot'}
    r = requests.post('http://www.pythonscraping.com/comment/reply/10',
                      data=params)
    responseObj = BeautifulSoup(r.text, 'html.parser')
    if responseObj.find('div', {'class':'messages'}) is not None:
        print(responseObj.find('div', {'class':'messages'}).get_text())
    else:
        print('There was a problem reading the CAPTCHA correctly!')

```

Observe que esse script falha em duas condições: se o Tesseract não extrair exatamente cinco caracteres da imagem (porque sabemos que todas as soluções válidas para esse CAPTCHA devem ter cinco caracteres), ou se o formulário é submetido, mas o CAPTCHA não foi resolvido corretamente. O primeiro caso ocorre cerca de 50% das vezes; nesse caso, ele não se preocupa em submeter o formulário e falha com uma mensagem de erro. O segundo caso ocorre aproximadamente 20% das vezes, com uma taxa de precisão total de cerca de 30% (ou cerca de 80% de precisão para cada caractere encontrado, em cinco caracteres).

Embora esse valor pareça baixo, tenha em mente que, em geral, nenhum limite é estabelecido para o número de vezes que os usuários podem fazer tentativas de resolver o CAPTCHA, e que a maioria dessas tentativas incorretas pode ser abortada sem a necessidade de realmente enviar o formulário. Quando um formulário é enviado, o CAPTCHA estará correto na maioria das vezes. Se isso não convencer você, não se esqueça também de que um palpite simples resultaria em uma taxa de acerto de 0,0000001%. Executar um programa três ou quatro vezes em vez de dar 900 milhões de palpites representa uma economia significativa de tempo!

---

<sup>1</sup> Quando se trata de processar um texto para o qual não tenha sido treinado, o Tesseract se sai muito melhor com edições de livros que usem fontes grandes, sobretudo se as imagens forem pequenas. A próxima seção discute como treinar o Tesseract com fontes diferentes, o que poderá ajudá-lo a ler tamanhos muito menores de fontes, incluindo visualizações prévias de edições de livros cujas fontes não sejam grandes!

<sup>2</sup> Veja <https://gizmodo.com/google-has-finally-killed-the-captcha-1793190374>.

## Evitando armadilhas no scraping

Poucas coisas são mais frustrantes que fazer scraping de um site, visualizar o resultado e não ver os dados que estão claramente visíveis em seu navegador. Ou submeter um formulário que deveria estar perfeitamente correto, mas ser recusado pelo servidor web. Ou ter seu endereço IP bloqueado por um site por razões desconhecidas.

Esses são alguns dos bugs mais difíceis de solucionar, não só porque podem ser tão inesperados (um script que funciona muito bem em um site pode não funcionar em outro, aparentemente idêntico), mas porque não têm propositalmente nenhuma mensagem de erro esclarecedora nem stack traces para serem usados. Você foi identificado como um bot, foi rejeitado e não sabe por quê.

Neste livro, descrevi várias maneiras de fazer tarefas intrincadas em sites (submeter formulários, extrair e limpar dados complicados, executar JavaScript etc.). Este capítulo é uma espécie de guarda-chuva, pois aborda várias técnicas com raízes em uma ampla gama de assuntos (cabeçalhos HTTP, CSS e formulários HTML, para mencionar alguns). No entanto, todas têm um ponto em comum: foram criadas para superar um obstáculo colocado somente com o propósito de impedir um web scraping automatizado em um site.

Não importa até que ponto essa informação seja útil de imediato no momento para você, porém recomendo que ao menos passe os olhos neste capítulo. Nunca se sabe quando ele poderá ajudá-lo a resolver um bug difícil ou evitar totalmente um problema.

### **Uma observação sobre ética**

Nos primeiros capítulos deste livro, discutimos a área cinzenta legal em que se encontra o web scraping, assim como algumas das diretrizes éticas a serem seguidas no scraping. Para ser franca, este capítulo, do ponto de vista ético, talvez seja o mais difícil de escrever. Meus sites estão infestados

de bots, geradores de spams, web scrapers e tudo quanto é tipo de visitantes virtuais indesejados, como talvez você tenha visto. Então por que ensinar as pessoas a escrever bots melhores?

Acredito que a inclusão deste capítulo seja importante por alguns motivos:

- Há razões perfeitamente éticas e legalmente bem amparadas para fazer scraping de alguns sites que não querem ter dados coletados. Em um trabalho anterior que fiz como web scraper, realizei uma coleta automática de informações de sites que publicavam nomes, endereços, números de telefone e outras informações pessoais de clientes na internet sem o consentimento deles. Usei as informações coletadas para fazer requisições formais aos sites a fim de que removessem essas informações. Para evitar a concorrência, esses sites protegiam zelosamente essas informações contra os scrapers. No entanto, meu trabalho para garantir o anonimato dos clientes de minha empresa (alguns dos quais tinham perseguidores, foram vítimas de violência doméstica ou tinham outros bons motivos para se manter discretos) constituía um caso atraente para o uso de web scraping, e me senti grata por ter as habilidades necessárias para fazer o serviço.
- Embora seja quase impossível construir um site “à prova de scrapers” (ou pelo menos um site que ainda seja acessado facilmente por usuários legítimos), espero que as informações neste capítulo ajudem aqueles que queiram defender seus sites contra ataques maliciosos. Ao longo do capítulo, destacarei alguns dos pontos fracos de cada técnica de web scraping, os quais você poderá usar para defender seu próprio site. Tenha em mente que a maioria dos bots na internet hoje em dia apenas faz uma ampla varredura em busca de informações e vulnerabilidades, e é provável que empregar algumas das técnicas simples descritas neste capítulo frustrará 99% deles. Contudo, os bots estão se tornando cada vez mais sofisticados à medida que o tempo passa, e é melhor estar preparado.
- Assim como a maioria dos programadores, não acredito que reter qualquer tipo de informação educativa tenha um saldo positivo.

Enquanto estiver lendo este capítulo, não se esqueça de que muitos desses scripts e as técnicas descritas não devem ser executados em todos os sites que você encontrar. Não é uma boa atitude e, além do mais, você poderia acabar recebendo uma carta de “cessar e desistir” (cease-and-desist letter)

ou algo pior (para mais informações sobre o que fazer caso receba uma carta desse tipo, veja o *Capítulo 18*). Entretanto, não insistirei nesse assunto sempre que discutirmos uma nova técnica. Assim, para o resto deste capítulo, como disse certa vez o filósofo Gump: “Isso é tudo que tenho a dizer sobre isso”.

## Parecendo um ser humano

O desafio básico para os sites que não queiram ter dados coletados é descobrir como diferenciar os bots dos seres humanos. Embora seja difícil enganar muitas das técnicas usadas pelos sites (como os CAPTCHAs), é possível tomar algumas atitudes simples para fazer seu bot se assemelhar a um ser humano.

### Ajuste seus cabeçalhos

Neste livro, fizemos uso da biblioteca *Requests* de Python para criar, enviar e receber requisições HTTP, por exemplo, quando tratamos formulários em um site no *Capítulo 10*. A biblioteca *Requests* também é excelente para configurar cabeçalhos. Os cabeçalhos HTTP são listas de atributos – ou preferências – enviadas por você sempre que fizer uma requisição a um servidor web. O HTTP define dezenas de tipos de cabeçalho obscuros, a maioria dos quais não é comum. Os sete campos a seguir, porém, são usados de modo consistente pela maioria dos principais navegadores ao iniciar qualquer conexão (os dados de exemplo exibidos são do meu próprio navegador):

Host	<code>https://www.google.com/</code>
Connection	<code>keep-alive</code>
Accept	<code>text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8</code>
User-Agent	<code>Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36</code>
Referrer	<code>https://www.google.com/</code>
Accept-Encoding	<code>gzip, deflate, sdch</code>
Accept-Language	<code>en-US,en;q=0.8</code>

E eis os cabeçalhos que um scraper Python típico usando a biblioteca `urllib` default poderia enviar:

```
_____
```

Accept-Encoding	identity
User-Agent	Python-urllib/3.4

Se você é administrador de um site e está tentando bloquear scrapers, qual deles é mais provável que você deixaria passar?

Felizmente, os cabeçalhos podem ser totalmente personalizados com a biblioteca *Requests*. O site <https://www.whatismybrowser.com> é ótimo para testar as propriedades dos navegadores que são visíveis aos servidores. Você fará o scraping desse site para conferir suas configurações de cookies com o script a seguir:

```
import requests
from bs4 import BeautifulSoup

session = requests.Session()
headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5)'
          'AppleWebKit 537.36 (KHTML, like Gecko) Chrome',
          'Accept': 'text/html,application/xhtml+xml,application/xml;'
          'q=0.9,image/webp,*/*;q=0.8'}
url = 'https://www.whatismybrowser.com/'
      'developers/what-http-headers-is-my-browser-sending'
req = session.get(url, headers=headers)

bs = BeautifulSoup(req.text, 'html.parser')
print(bs.find('table', {'class': 'table-striped'}).get_text)
```

O resultado deve mostrar que os cabeçalhos agora são os mesmos definidos no objeto dicionário `headers` no código.

Embora seja possível que os sites verifiquem se “é um ser humano” com base em qualquer uma das propriedades nos cabeçalhos HTTP, percebi que, em geral, a única configuração que realmente importa é a de `User-Agent`. É uma boa ideia manter esse cabeçalho configurado com um valor mais discreto que `Python-urllib/3.4`, não importa o projeto no qual você estiver trabalhando. Além do mais, se algum dia você deparar com um site muito desconfiado, preencher um dos cabeçalhos de uso comum, porém raramente verificado, por exemplo, `Accept-Language`, poderia ser a chave para convencê-lo de que você é um ser humano.

## Cabeçalhos mudam o modo como vemos o mundo

Suponha que queremos escrever um tradutor de idiomas usando aprendizado de máquina para um projeto de pesquisa, mas não temos grandes quantidades de textos traduzidos para testá-lo. Muitos sites grandes apresentam traduções diferentes para o mesmo conteúdo, com base nas preferências de idioma informadas em seus cabeçalhos. Apenas mudar `Accept-Language: en-US` para `Accept-Language: fr` em seus cabeçalhos poderá fazer você receber um “Bonjour” dos sites com escala e orçamento para lidar com traduções (grandes empresas internacionais em geral são uma boa aposta).

Os cabeçalhos também podem fazer os sites mudarem o formato do conteúdo que apresentam. Por exemplo, dispositivos móveis navegando pela internet muitas vezes veem uma versão reduzida dos sites, sem banners de anúncios, Flash e outras distrações. Se você tentar modificar seu **User-Agent** para outro valor, como o que vemos a seguir, talvez perceba que será um pouco mais fácil fazer scraping dos sites!

```
User-Agent:Mozilla/5.0 (iPhone; CPU iPhone OS 7_1_2 like Mac OS X)
AppleWebKit/537.51.2 (KHTML, like Gecko) Version/7.0 Mobile/11D257
Safari/9537.53
```

## Lidando com cookies em JavaScript

Lidar com cookies da forma correta pode atenuar muitos problemas de scraping, embora os cookies também possam ser uma faca de dois gumes. Sites que monitoram sua progressão usando cookies podem tentar recusar scrapers que exibam um comportamento anormal, por exemplo, que completem formulários muito depressa ou acessem muitas páginas. Apesar de esses comportamentos poderem ser disfarçados fechando e reabrindo as conexões com o site, ou até mesmo alterando seu endereço IP (veja o *Capítulo 17* para mais informações sobre como fazer isso), se seu cookie revelar a sua identidade, seus esforços para disfarçar poderão ser fúteis.

Os cookies também podem ser necessários para fazer scraping de um site. Como vimos no *Capítulo 10*, permanecer logado em um site exige que você seja capaz de armazenar e apresentar um cookie em cada página. Alguns sites nem mesmo exigem que você realmente faça login e sempre obtenha uma nova versão de um cookie – basta armazenar uma cópia antiga de um cookie “logado” e acessar o site.

Se estiver fazendo scraping de um único site ou de um pequeno número de sites desejados, recomendo analisar os cookies gerados por esses sites e considerar quais deles você vai querer que seu scraper trate. Vários plug-ins de navegadores podem mostrar como seus cookies são configurados quando você acessa um site e o percorre. [EditThisCookie \(www.editthiscookie.com\)](http://www.editthiscookie.com), uma extensão do Chrome, é um de meus favoritos.

Observe os códigos de exemplo da seção “*Lidando com logins e cookies*” no *Capítulo 10* para mais informações sobre como lidar com cookies usando a biblioteca *Requests*. É claro que, por não executar JavaScript, a biblioteca *Requests* não será capaz de lidar com muitos dos cookies gerados por um software de monitoração moderno, como o Google Analytics, os quais são configurados somente depois da execução de scripts do lado cliente (ou, às vezes, com base em eventos da página, como cliques de botão durante a

navegação na página). Para cuidar disso, é preciso usar os pacotes Selenium ou PhantomJS (discutimos sua instalação e o uso básico no *Capítulo 11*).

Podemos visualizar os cookies acessando qualquer site (<http://pythonscraping.com>, no exemplo a seguir) e chamando `get_cookies()` no webdriver:

```
from selenium import webdriver
driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get('http://pythonscraping.com')
driver.implicitly_wait(1)
print(driver.get_cookies())
```

Esse código gera o array bem típico dos cookies do Google Analytics:

```
[{'value': '1', 'httponly': False, 'name': '_gat', 'path': '/', 'expiry': 1422806785, 'expires': 'Sun, 01 Feb 2015 16:06:25 GMT', 'secure': False, 'domain': '.pythonscraping.com'}, {'value': 'GA1.2.1619525062.1422806186', 'httponly': False, 'name': '_ga', 'path': '/', 'expiry': 1485878185, 'expires': 'Tue, 31 Jan 2017 15:56:25 GMT', 'secure': False, 'domain': '.pythonscraping.com'}, {'value': '1', 'httponly': False, 'name': 'has_js', 'path': '/', 'expiry': 1485878185, 'expires': 'Tue, 31 Jan 2017 15:56:25 GMT', 'secure': False, 'domain': 'pythonscraping.com'}]
```

Para manipular cookies, as funções `delete_cookie()`, `add_cookie()` e `delete_all_cookies()` podem ser chamadas. Além do mais, é possível salvar e armazenar cookies para usá-los com outros web scrapers. Eis um exemplo para que você tenha uma ideia de como essas funções atuam juntas:

```
from selenium import webdriver

phantomPath = '<Path to Phantom JS>'
driver = webdriver.PhantomJS(executable_path=phantomPath)
driver.get('http://pythonscraping.com')
driver.implicitly_wait(1)

savedCookies = driver.get_cookies()
print(savedCookies)

driver2 = webdriver.PhantomJS(executable_path=phantomPath)
driver2.get('http://pythonscraping.com')
driver2.delete_all_cookies()
for cookie in savedCookies:
    if not cookie['domain'].startswith('.'):
        cookie['domain'] = '.{}'.format(cookie['domain'])
    driver2.add_cookie(cookie)

driver2.get('http://pythonscraping.com')
driver.implicitly_wait(1)
```

```
print(driver2.get_cookies())
```

Nesse exemplo, o primeiro webdriver lê um site, exibe os cookies e então os armazena na variável `savedCookies`. O segundo webdriver carrega o mesmo site, apaga os próprios cookies e adiciona os cookies do primeiro webdriver. A seguir, apresentamos duas notas de caráter técnico:

- O segundo webdriver deve carregar o site antes de os cookies serem adicionados. Isso serve para que o Selenium saiba a qual domínio os cookies pertencem, mesmo que o ato de carregar o site não tenha utilidade nenhuma para o scraper.
- Uma verificação é feita antes de carregar cada cookie para conferir se o domínio começa com um caractere de ponto (.). Essa é uma idiossincrasia do PhantomJS – todos os domínios em cookies adicionados devem começar com um ponto (por exemplo, `.pythonscraping.com`), apesar de nem todos os cookies do webdriver do PhantomJS realmente seguirem essa regra. Se você estiver usando um driver para outro navegador, como Chrome ou Firefox, isso não será necessário.

Depois de executar o código, o segundo webdriver deverá ter os mesmos cookies que o primeiro. De acordo com o Google Analytics, esse segundo webdriver agora é idêntico ao primeiro, e ambos serão monitorados do mesmo modo. Se o primeiro webdriver estivesse logado em um site, o segundo webdriver também estaria.

## Tempo é tudo

Alguns sites bem protegidos podem impedir que você submeta formulários ou interaja com o site se você fizer isso com muita rapidez. Mesmo se esses recursos de segurança não estiverem presentes, fazer download de muitas informações de um site de modo significativamente mais rápido do que um ser humano comum faria é uma boa maneira de ser notado e bloqueado.

Assim, embora a programação multithreaded seja uma ótima maneira de carregar páginas rapidamente – permitindo processar dados em uma thread enquanto as páginas são carregadas de forma contínua em outra –, é uma estratégia ruim para escrever bons scrapers. Procure sempre manter as cargas de página individuais e as requisições de dados em um nível mínimo. Se for possível, procure fazê-los com intervalos de alguns segundos, mesmo que seja necessário acrescentar um código extra:

```
import time
```

```
time.sleep(3)
```

O fato de precisar desses segundos extras entre as cargas de páginas em geral é determinado de modo experimental. Muitas vezes, já tive dificuldades para coletar dados de um site, tendo que provar que “eu não era um robô” a intervalos de alguns minutos (resolvendo um CAPTCHA manualmente, colando meus cookies recém-obtidos no scraper para que o site considerasse que ele havia “provado que é um ser humano”), mas acrescentar um `time.sleep` resolvia meus problemas e permitia que eu fizesse scraping por tempo indeterminado.

Às vezes é preciso reduzir a velocidade para ir mais rápido!

## Recursos de segurança comuns em formulários

Muitos testes de diferenciação têm sido usados ao longo dos anos, e continuam sendo empregados, com graus variados de sucesso, para distinguir web scrapers de seres humanos utilizando navegadores. Embora não seja um grande problema se um bot fizer download de alguns artigos e postagens de blog que, de qualquer modo, já estariam disponíveis ao público, será um problema se um bot criar milhares de contas de usuários e começar a enviar spams para todos os membros de seu site. Os formulários web, sobretudo aqueles que lidam com a criação de contas e com logins, representam uma ameaça significativa para a segurança, e podem causar overhead de processamento se forem vulneráveis a um uso indiscriminado por parte dos bots, portanto, tentar limitar o acesso ao site é do interesse da maioria de seus proprietários (ou, pelo menos, é o que eles acham).

Essas medidas de segurança contra bots centradas em formulários e logins podem representar um desafio significativo para os web scrapers.

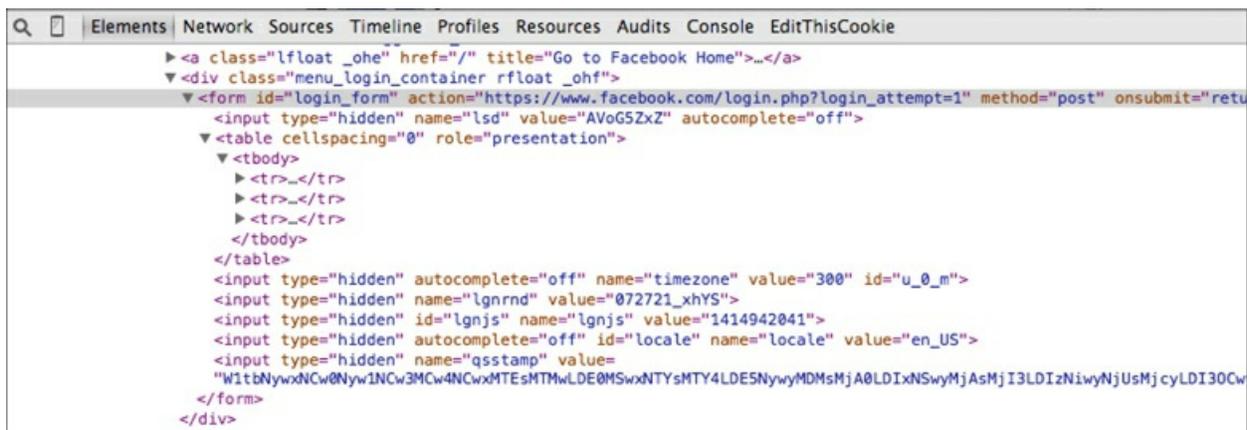
Tenha em mente que essa é apenas uma visão parcial de algumas das medidas de segurança que poderão ser encontradas ao criar bots automatizados para esses formulários. Reveja o *Capítulo 13*, que descreve como lidar com CAPTCHAs e processamento de imagens, e consulte também o *Capítulo 17*, que trata dos cabeçalhos e endereços IP, para obter mais informações sobre como lidar com formulários bem protegidos.

## Valores de campos de entrada ocultos

Campos “ocultos” em formulários HTML permitem que o valor contido no campo seja visível apenas ao navegador, mas seja invisível aos usuários

(a menos que eles vejam o código-fonte do site). Com o aumento do uso de cookies para armazenar variáveis e passá-las para os sites, os campos ocultos foram postos de lado por um tempo até que outro propósito excelente lhes foi atribuído: impedir que scrapers submetessem formulários.

A Figura 14.1 mostra um exemplo desses campos ocultos em atuação em uma página de login do Facebook. Apesar de haver apenas três campos visíveis (Nome do usuário, Senha e um botão de Submissão), o formulário envia muitas informações ao servidor internamente.



```
Q Elements Network Sources Timeline Profiles Resources Audits Console EditThisCookie
▶ <a class="lfloat_ohf" href="/" title="Go to Facebook Home">...</a>
▼ <div class="menu_login_container rfloat_ohf">
  ▼ <form id="login_form" action="https://www.facebook.com/login.php?login_attempt=1" method="post" onsubmit="retu
    <input type="hidden" name="lsd" value="AVoG5ZxZ" autocomplete="off">
    ▼ <table cellpadding="0" role="presentation">
      ▼ <tbody>
        ▶ <tr>...</tr>
        ▶ <tr>...</tr>
        ▶ <tr>...</tr>
      </tbody>
    </table>
    <input type="hidden" autocomplete="off" name="timezone" value="300" id="u_0_m">
    <input type="hidden" name="lgnrnd" value="072721_xhYS">
    <input type="hidden" id="lgajs" name="lgajs" value="1414942041">
    <input type="hidden" autocomplete="off" id="locale" name="locale" value="en_US">
    <input type="hidden" name="qsstamp" value="
      "W1tbNywxNCw0Nyw1NCw3MCw4NCwMTEsMTMwLDE0MSwxNTYsMTY4LDE5NywyMDMsMjA0LDIxNSwyMjAsMjI3LDIzNiwyNjUsMjcyLDI3OCw
    </form>
  </div>
```

Figura 14.1 – O formulário de login do Facebook tem vários campos ocultos.

Os campos ocultos são usados para evitar web scraping de duas maneiras principais: um campo pode ser preenchido com uma variável gerada de modo aleatório na página de formulário, e o servidor espera que seu post seja feito para a página de processamento do formulário. Se esse valor não estiver presente no formulário, o servidor poderá supor, de modo razoável, que a submissão não teve origem orgânica, na página do formulário, mas o post foi feito diretamente para a página de processamento por um bot. A melhor maneira de contornar essa medida é fazer o scraping da página de formulário antes, coletar a variável gerada aleatoriamente e então fazer o post para a página de processamento.

O segundo método é uma espécie de “honeypot”. Se um formulário tiver um campo oculto com um nome inócuo, por exemplo, Nome do Usuário ou Endereço de Email, um bot mal escrito poderá preencher o campo e tentar submetê-lo, sem se importar com o fato de ele estar oculto ao usuário. Qualquer campo oculto contendo um valor (ou valores diferentes de seus defaults na página de submissão do formulário) deve ser

descartado, e o usuário poderá até mesmo ser bloqueado no site.

Em suma, às vezes é necessário verificar a página em que está o formulário para ver se você deixou de notar algo que o servidor está esperando. Se vir vários campos ocultos, em geral com variáveis do tipo string grandes, geradas aleatoriamente, é provável que o servidor web verificará a existência delas no formulário de submissão. Além disso, pode haver outras verificações para garantir que as variáveis do formulário tenham sido usadas apenas uma vez, foram geradas há pouco tempo (isso elimina a possibilidade de simplesmente as armazenar em um script e usá-las de forma repetida com o passar do tempo), ou ambos.

## Evitando honeypots

Apesar de, na maioria das vezes, o CSS facilitar bastante a vida quando se trata de diferenciar entre informações úteis e inúteis (por exemplo, pela leitura das tags `id` e `class`), ocasionalmente, ele pode ser um problema para os web scrapers. Se um campo em um formulário web estiver oculto a um usuário via CSS, é razoável supor que um usuário comum acessando o site não seja capaz de preenchê-lo porque esse campo não será exibido no navegador. Se o formulário *for* preenchido, é provável que haja um bot atuando e o post será descartado.

Isso se aplica não só aos formulários, mas também a links, imagens, arquivos e qualquer outro item no site que seja lido por um bot, mas esteja oculto a um usuário comum acessando o site com um navegador. Um acesso de página a um link “oculto” em um site pode facilmente disparar um script do lado do servidor que bloqueará o endereço IP do usuário, fará logout do usuário nesse site ou tomará alguma outra atitude para evitar novos acessos. De fato, muitos modelos de negócio estão baseados exatamente nesse conceito.

Tome, por exemplo, a página que está em <http://pythonscraping.com/pages/itsatrap.html>. Essa página contém dois links, um oculto por CSS e outro visível. Além disso, ela contém um formulário com dois campos ocultos:

```
<html>
<head>
  <title>A bot-proof form</title>
</head>
<style>
  body {
```

```

        overflow-x:hidden;
    }
    .customHidden {
        position:absolute;
        right:50000px;
    }
</style>
<body>
    <h2>A bot-proof form</h2>
    <a href=
        "http://pythonscraping.com/dontgohere" style="display:none;">Go here!</a>
    <a href="http://pythonscraping.com">Click me!</a>
    <form>
        <input type="hidden" name="phone" value="valueShouldNotBeModified"/><p/>
        <input type="text" name="email" class="customHidden"
            value="intentionallyBlank"/><p/>
        <input type="text" name="firstName"/><p/>
        <input type="text" name="lastName"/><p/>
        <input type="submit" value="Submit"/><p/>
    </form>
</body>
</html>

```

Os três elementos a seguir estão ocultos ao usuário usando três métodos:

- O primeiro link está oculto com um simples atributo `display:none` do CSS.
- O campo de número de telefone é um campo de entrada oculto.
- O campo de email está oculto por estar a 50 mil pixels à direita (presumivelmente fora da tela dos monitores de qualquer pessoa), e a barra de rolagem que poderia denunciá-lo está oculta.

Felizmente, como o Selenium renderiza as páginas que acessa, ele é capaz de fazer a distinção entre os elementos visíveis na página e os que não estão. O fato de um elemento estar visível na página pode ser determinado com a função `is_displayed()`.

Por exemplo, o código a seguir obtém a página descrita antes e procura links e campos de entrada ocultos no formulário:

```

from selenium import webdriver
from selenium.webdriver.remote.webelement import WebElement

driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get('http://pythonscraping.com/pages/itsatrap.html')
links = driver.find_elements_by_tag_name('a')
for link in links:
    if not link.is_displayed():
        print('The link {} is a trap'.format(link.get_attribute('href')))

```

```
fields = driver.find_elements_by_tag_name('input')
for field in fields:
    if not field.is_displayed():
        print('Do not change value of {}'.format(field.get_attribute('name')))
```

O Selenium identifica todos os campos ocultos e gera o resultado a seguir:

```
The link http://pythonscraping.com/dontgohere is a trap
Do not change value of phone
Do not change value of email
```

Embora seja improvável que você acesse qualquer link oculto que encontrar, deve garantir que submeterá valores ocultos previamente preenchidos (ou que o Selenium os submeterá para você), com o resto do formulário. Em suma, apenas ignorar os campos ocultos é perigoso, embora você deva ter cuidado ao interagir com eles.

## Lista de verificação para parecer um ser humano

Há muitas informações neste capítulo – na verdade, neste livro – sobre como construir um scraper que se pareça menos com um scraper e mais com um ser humano. Se você é constantemente bloqueado pelos sites e não sabe o motivo, eis uma lista de verificação que pode ser usada para resolver o problema:

- Em primeiro lugar, se a página recebida do servidor web estiver em branco, tiver informações faltando ou não for a página que você espera (ou é diferente da página que você viu em seu navegador), é provável que isso se deva a um JavaScript executado no site para criar a página. Leia novamente o *Capítulo 11*.
- Se estiver submetendo um formulário ou fazendo uma requisição `POST` para um site, verifique a página a fim de garantir que tudo que o site espera que você submeta está sendo submetido, e no formato correto. Use uma ferramenta como o painel de Inspeção do Chrome para visualizar uma requisição `POST` enviada ao site e garantir que tudo esteja presente, e que uma requisição “orgânica” seja igual àquelas que seu bot está enviando.
- Se estiver tentando fazer login em um site e não conseguir fazê-lo “durar”, ou se o site estiver apresentando outros comportamentos de “estado” anormais, verifique seus cookies. Certifique-se de que os cookies estão sendo persistidos da forma correta entre cada carga de página e que estão sendo enviados ao site a cada requisição.

- Se estiver obtendo erros de HTTP do cliente, em especial, erros 403 Forbidden (Proibido), isso pode indicar que o site identificou seu endereço IP como um bot e não está disposto a aceitar novas requisições. Você terá de esperar até seu endereço IP ser removido da lista ou deverá adquirir um novo endereço IP (vá a um Starbucks ou consulte o *Capítulo 17*). Para garantir que não será bloqueado novamente, experimente fazer o seguinte:
  - Certifique-se de que seus scrapers não estejam se movendo rápido demais pelo site. Um scraping rápido não é uma boa prática, e impõe uma carga pesada aos servidores de administração web, além de poder criar um problema do ponto de vista legal para você e constituir o principal motivo para os scrapers serem colocados em listas negras. Adicione pausas em seus scrapers e deixe que executem durante a noite toda. Lembre-se de que escrever programas ou coletar dados com pressa são sinais de um gerenciamento de projeto ruim; planeje com antecedência para evitar confusões como essa.
  - A providência óbvia é: mude seus cabeçalhos! Alguns sites bloquearão qualquer comunicação que se anunciar como um scraper. Copie os cabeçalhos de seu próprio navegador caso não esteja certo sobre o que seriam alguns valores razoáveis para o cabeçalho.
  - Certifique-se de que não clicará nem acessará nada que um ser humano em geral não seria capaz de clicar ou acessar (consulte a seção “*Evitando honeypots*” para ver outras informações).
  - Se você se vir passando por uma série de dificuldades para ter acesso, considere entrar em contato com o administrador do site para que ele saiba o que você está fazendo. Experimente enviar um email para *webmaster@<nome do domínio>* ou *admin@<nome do domínio>* a fim de pedir permissão para usar seus scrapers. Os administradores também são pessoas, e você poderá se surpreender ao saber como eles podem ser prestativos para compartilhar seus dados.

# Testando seu site com scrapers

Ao trabalhar com projetos web que tenham uma pilha de desenvolvimento grande, em geral, testa-se apenas a parte “de trás” da pilha regularmente. A maioria das linguagens de programação atuais (inclusive Python) tem algum tipo de framework de teste, mas os frontends dos sites muitas vezes ficam de fora desses testes automatizados, ainda que talvez sejam a única parte do projeto vista pelos clientes.

Parte do problema é que, com frequência, os sites são uma mistura de várias linguagens de marcação e de programação. Você pode escrever testes de unidade para suas seções de JavaScript, mas será inútil se o HTML com o qual elas interagem mudar, de modo que o código não exerça mais a ação esperada na página, mesmo que ele funcione corretamente.

O problema dos testes de frontend dos sites muitas vezes acaba ficando para ser pensado *a posteriori*, ou é atribuído a programadores de nível mais baixo, de posse, no máximo, de uma lista de verificação e uma ferramenta para rastrear bugs. No entanto, com apenas um pouco mais de esforço prévio, essa lista de verificação pode ser substituída por uma série de testes de unidade, e os olhos humanos podem ser substituídos por um web scraper.

Pense nisto: desenvolvimento orientado a testes aplicado ao desenvolvimento web. Testes diários para garantir que todas as partes da interface web estão funcionando conforme esperado. Uma suíte de testes executada sempre que alguém acrescentar uma nova funcionalidade no site ou alterar a posição de um elemento. Este capítulo discute o básico sobre testes e como testar todo tipo de sites, dos simples aos complicados, com web scrapers Python.

## Introdução aos testes

Se você nunca escreveu testes para seu código, não há melhor hora do que agora para começar. Ter uma suíte de testes que seja executada a fim de

garantir que seu código funcione conforme esperado (pelo menos, de acordo com os testes escritos) fará você poupar tempo e evitará preocupações, facilitando o lançamento de novas atualizações.

## O que são testes de unidade?

As expressões *teste* e *teste de unidade* muitas vezes são usadas de forma indistinta. Em geral, quando os programadores se referem a “escrever testes”, o que eles realmente querem dizer é “escrever testes de unidade”. Por outro lado, quando alguns programadores se referem a escrever testes de unidade, na verdade, estão escrevendo algum outro tipo de teste.

Apesar de as definições e as práticas tenderem a variar de empresa para empresa, um teste de unidade, de modo geral, tem as seguintes características:

- Cada teste de unidade testa um único aspecto da funcionalidade de um componente. Por exemplo, o teste pode garantir que a mensagem de erro apropriada será gerada se um valor monetário negativo for sacado de uma conta bancária.

Com frequência, os testes de unidade são agrupados na mesma classe, de acordo com o componente que estiverem testando. Poderíamos ter um teste para um valor monetário negativo sendo sacado de uma conta bancária, seguido de um teste de unidade para o comportamento de uma conta bancária que ficasse negativa.

- Cada teste de unidade pode ser executado de modo totalmente independente, e qualquer configuração ou desmontagem (teardown) necessárias ao teste de unidade devem ser tratadas pelo próprio teste de unidade. De modo semelhante, os testes de unidade não devem interferir no sucesso ou na falha de outros testes, e deve ser possível executá-los com sucesso em qualquer ordem.
- Cada teste de unidade em geral contém pelo menos uma *asserção*. Por exemplo, um teste de unidade poderia confirmar se a resposta de  $2 + 2$  é 4. Ocasionalmente, um teste de unidade pode conter apenas um estado de falha. Por exemplo, o teste pode falhar se uma exceção for lançada, mas, por padrão, passará se tudo correr bem.
- Os testes de unidade ficam separados do resto do código. Embora tenham necessariamente de importar e usar o código que testam, em geral, os testes são mantidos em classes e em diretórios separados.

Embora haja vários outros tipos de testes – testes de integração e testes de validação, por exemplo –, este capítulo tem como foco principal os testes de unidade. Estes não só se tornaram muito conhecidos, com tendências recentes em direção ao desenvolvimento orientado a testes, como também seu tamanho e a flexibilidade fazem com que seja fácil trabalhar com eles como exemplos; Python tem alguns recursos embutidos para testes de unidade, como veremos na próxima seção.

## Módulo unittest de Python

O módulo de teste de unidade `unittest` de Python acompanha todas as instalações padrões da linguagem. Basta importar e estender `unittest.TestCase`, e ele fará o seguinte:

- disponibilizará funções `setUp` e `tearDown` executadas antes e depois de cada teste de unidade;
- disponibilizará vários tipos de instruções de “asserção” para permitir que os testes passem ou falhem;
- executará todas as funções que começam com `test_` como testes de unidade e ignorará as funções que não tenham esse prefixo.

O código a seguir contém um único teste de unidade simples para garantir que  $2 + 2 = 4$ , de acordo com Python:

```
import unittest

class TestAddition(unittest.TestCase):
    def setUp(self):
        print('Setting up the test')

    def tearDown(self):
        print('Tearing down the test')

    def test_twoPlusTwo(self):
        total = 2+2
        self.assertEqual(4, total);

if __name__ == '__main__':
    unittest.main()
```

Embora `setUp` e `tearDown` não ofereçam nenhuma funcionalidade útil nesse caso, elas foram incluídas para ilustrar. Observe que essas funções são executadas antes e depois de cada teste individual, e não antes e depois de todos os testes da classe.

A saída da função de teste, quando executada da linha de comando, deverá

ser:

```
Setting up the test
Tearing down the test
```

.

```
-----
Ran 1 test in 0.000s
```

OK

Esse resultado mostra que o teste executou com sucesso, e que  $2 + 2$  é realmente igual a 4.

## Executando o unittest em notebooks Jupyter

Os scripts de teste de unidade neste capítulo são todos disparados com:

```
if __name__ == '__main__':
    unittest.main()
```

A linha `if __name__ == '__main__'` será verdadeira somente se a linha for executada diretamente em Python, e não por meio de uma instrução de importação. Isso permite a você executar o seu teste de unidade, com a classe `unittest.TestCase` que ele estende, diretamente da linha de comando.

Em um notebook Jupyter, a situação é um pouco diferente. Os parâmetros `argv` criados pelo Jupyter podem causar erros no teste de unidade; como o framework `unittest`, por padrão, sai de Python depois que o teste é executado (isso causa problemas no kernel do notebook), também é necessário evitar que isso aconteça.

Nos notebooks Jupyter, usaremos o seguinte para iniciar os testes de unidade:

```
if __name__ == '__main__':
    unittest.main(argv=[''], exit=False)
%reset
```

A segunda linha define todas as variáveis de `argv` (argumentos da linha de comando) com uma string vazia, que é ignorada por `unittest.main`. Ela também evita que `unittest` saia depois que o teste é executado.

A linha `%reset` é útil porque reinicia a memória e destrói todas as variáveis criadas pelo usuário no notebook Jupyter. Sem ela, cada teste de unidade que você escrever no notebook conterà todos os métodos de todos os testes executados antes, que também herdaram de `unittest.TestCase`, incluindo os métodos `setUp` e `tearDown`. Isso também significa que cada teste de unidade executaria todos os métodos dos testes de unidade antes dele!

Usar `%reset`, porém, cria um passo manual extra para o usuário na execução dos testes. Ao executar o teste, o notebook perguntará se o usuário tem certeza de que quer reiniciar a memória. Basta digitar `y` e teclar Enter para fazer isso.

## Testando a Wikipédia

Testar o frontend de seu site (excluindo o JavaScript, que será discutido depois) é simples e basta combinar a biblioteca Python `unittest` com um web scraper:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import unittest
```

```
class TestWikipedia(unittest.TestCase):
```

```

bs = None
def setUpClass():
    url = 'http://en.wikipedia.org/wiki/Monty_Python'
    TestWikipedia.bs = BeautifulSoup(urlopen(url), 'html.parser')

def test_titleText(self):
    pageTitle = TestWikipedia.bs.find('h1').get_text()
    self.assertEqual('Monty Python', pageTitle);

def test_contentExists(self):
    content = TestWikipedia.bs.find('div',{'id':'mw-content-text'})
    self.assertIsNotNone(content)

if __name__ == '__main__':
    unittest.main()

```

Há dois testes desta vez: o primeiro verifica se o título da página é “Monty Python” conforme esperado, enquanto o segundo garante que a página tem uma div de conteúdo.

Observe que o conteúdo da página é carregado apenas uma vez e o objeto global `bs` é compartilhado entre os testes. Isso é feito com o uso da função `setUpClass` especificada pelo `unittest`, a qual é executada apenas uma vez no início da classe (de modo diferente de `setUp`, executada antes de cada teste individual). Usar `setUpClass` em vez de `setUp` evita cargas desnecessárias da página; podemos adquirir o conteúdo uma vez e executar vários testes com ele.

Uma grande diferença de arquitetura entre `setUpClass` e `setUp`, além de quando e com qual frequência são executadas, consiste em `setUpClass` ser um método estático que “pertence” à própria classe e tem variáveis globais de classe, enquanto `setUp` é uma função de instância, que pertence a uma instância da classe em particular. É por isso que `setUp` pode definir atributos em `self` – a instância em particular dessa classe – enquanto `setUpClass` pode acessar somente atributos estáticos da classe `TestWikipedia`.

Embora testar uma única página de cada vez talvez não pareça tão eficaz nem muito interessante, como você deve se lembrar do que vimos no *Capítulo 3*, é relativamente fácil construir web crawlers capazes de percorrer as páginas de um site de modo interativo. O que acontecerá se combinarmos um web crawler com um teste de unidade que faça uma asserção sobre cada página?

Há muitas maneiras de executar um teste de forma repetida, mas você deve tomar cuidado e carregar cada página apenas uma vez para cada conjunto

de testes que quiser executar nela, e deve evitar também a armazenagem de grandes quantidades de informações na memória ao mesmo tempo. A configuração a seguir faz exatamente isso:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import unittest
import re
import random
from urllib.parse import unquote

class TestWikipedia(unittest.TestCase):

    def test_PageProperties(self):
        self.url = 'http://en.wikipedia.org/wiki/Monty_Python'
        # Testa as 10 primeiras páginas encontradas
        for i in range(1, 10):
            self.bs = BeautifulSoup(urlopen(self.url), 'html.parser')
            titles = self.titleMatchesURL()
            self.assertEqual(titles[0], titles[1])
            self.assertTrue(self.contentExists())
            self.url = self.getNextLink()
        print('Done!')

    def titleMatchesURL(self):
        pageTitle = self.bs.find('h1').get_text()
        urlTitle = self.url[(self.url.index('/wiki/')+6):]
        urlTitle = urlTitle.replace('_', ' ')
        urlTitle = unquote(urlTitle)
        return [pageTitle.lower(), urlTitle.lower()]

    def contentExists(self):
        content = self.bs.find('div',{'id':'mw-content-text'})
        if content is not None:
            return True
        return False

    def getNextLink(self):
        # Devolve um link aleatório da página, usando a técnica mostrada na
        Capítulo 3
        links = self.bs.find('div', {'id':'bodyContent'}).find_all(
            'a', href=re.compile('^(/wiki/)((?!:).)*$'))
        randomLink = random.SystemRandom().choice(links)
        return 'https://wikipedia.org{}'.format(randomLink.attrs['href'])

if __name__ == '__main__':
    unittest.main()
```

Há alguns pontos a serem observados. Em primeiro lugar, há apenas um teste nessa classe. As outras funções, tecnicamente, são apenas funções

auxiliares, apesar de estarem fazendo a maior parte do trabalho de processamento para determinar se um teste passa. Como a função de teste executa as instruções de asserção, os resultados dos testes são passados de volta para a função na qual a asserção ocorre.

Além disso, enquanto `contentExists` devolve um booleano, `titleMatchesURL` devolve os próprios valores para avaliação. Para ver por que deveríamos passar os valores e não apenas um booleano, compare o resultado de uma asserção booleana:

```
=====
FAIL: test_PageProperties (__main__.TestWikipedia)
-----
Traceback (most recent call last):
  File "15-3.py", line 22, in test_PageProperties
    self.assertTrue(self.titleMatchesURL())
AssertionError: False is not true
```

com o resultado de uma instrução `assertEquals`:

```
=====
FAIL: test_PageProperties (__main__.TestWikipedia)
-----
Traceback (most recent call last):
  File "15-3.py", line 23, in test_PageProperties
    self.assertEqual(titles[0], titles[1])
AssertionError: 'lockheed u-2' != 'u-2 spy plane'
```

Qual delas é mais fácil de depurar? (Nesse caso, o erro ocorre por causa de um redirecionamento, quando o artigo <http://wikipedia.org/wiki/u-2%20spy%20plane> é redirecionado para um artigo cujo título é “Lockheed U-2”).

## Testando com o Selenium

Como no caso do scraping de Ajax que vimos no *Capítulo 11*, JavaScript apresenta desafios específicos quando se trata de testar sites. Felizmente, o Selenium tem um framework excelente para lidar com sites complicados; de fato, a biblioteca foi originalmente projetada para testar sites!

Apesar de estarem escritos na mesma linguagem, você pode ficar surpreso com o fato de a sintaxe dos testes de unidade com Python e com o Selenium ter pouco em comum. O Selenium não exige que seus testes de

unidade estejam contidos em funções dentro de classes, suas instruções `assert` não exigem parênteses e os testes transcorrem silenciosamente, gerando algum tipo de mensagem apenas se houver uma falha:

```
driver = webdriver.PhantomJS()
driver.get('http://en.wikipedia.org/wiki/Monty_Python')
assert 'Monty Python' in driver.title
driver.close()
```

Quando for executado, esse teste não deverá gerar nenhuma saída

Desse modo, os testes com o Selenium podem ser escritos de modo mais casual que os testes de unidade Python, e instruções `assert` podem até mesmo ser integradas no código comum, nos pontos em que for desejável que a execução do código termine caso alguma condição não seja atendida.

## Interagindo com o site

Recentemente, eu queria entrar em contato com um pequeno negócio local por meio do formulário de contato do site dele, mas percebi que o formulário HTML estava com problemas; nada acontecia quando eu clicava no botão de envio. Depois de investigar um pouco, vi que eles usavam um formulário mailto simples, criado para lhes enviar um email com o conteúdo do formulário. Felizmente, pude usar essa informação para enviar-lhes um email, explicar o problema com o formulário e contratá-los, apesar do problema técnico.

Se eu fosse escrever um scraper tradicional que usasse ou testasse esse formulário, é provável que meu scraper apenas copiasse o layout do formulário e enviasse um email diretamente – ignorando por completo o formulário. Como eu poderia testar a funcionalidade do formulário e garantir que estivesse funcionando perfeitamente com um navegador?

Apesar de os capítulos anteriores terem discutido como navegar em links, submeter formulários e fazer outros tipos de atividades que envolvam interação, em sua essência, tudo que fizemos foi concebido para *ignorar* a interface do navegador, e não para usá-la. O Selenium, por outro lado, é capaz de literalmente inserir texto, clicar em botões e fazer tudo por meio do navegador (nesse caso, o navegador headless PhantomJS), detectando problemas como formulários inoperantes, código JavaScript ruim, erros de digitação no HTML e outros problemas que sejam um obstáculo aos clientes.

O conceito de `elements` do Selenium é fundamental nesse tipo de teste.



```

actions.perform()
#####

print(driver.find_element_by_tag_name('body').text)

driver.close()

```

O Método 1 chama `send_keys` nos dois campos e então clica no botão de submissão. O Método 2 usa uma única cadeia de ações para clicar e inserir um texto em cada campo, o que acontece em uma sequência depois que o método `perform` é chamado. Esse script atua do mesmo modo, não importa se o primeiro ou o segundo método é usado, e exibe a linha a seguir:

```
Hello there, Ryan Mitchell!
```

Há outra diferença entre os dois métodos, além dos objetos que usam para tratar os comandos: observe que o primeiro método clica no botão Submit, e o segundo utiliza a tecla Return para submeter o formulário enquanto a caixa de texto é submetida. Como há várias maneiras de pensar na sequência de eventos que completam a mesma ação, há diversas maneiras de executar a mesma ação com o Selenium.

### Arrastar e soltar

Clicar em botões e inserir texto é um dos recursos, mas o Selenium realmente se destaca na capacidade de lidar com formas relativamente novas de interação na web. O Selenium permite manipular interfaces do tipo arrastar-e-soltar (drag-and-drop) com facilidade. Usar sua função de arrastar-e-soltar exige que você especifique um elemento de *origem* (o elemento que será arrastado) e um offset para arrastá-lo ou um elemento-alvo sobre o qual ele será arrastado.

A página de demonstração em <http://pythonscraping.com/pages/javascript/draggableDemo.html> contém um exemplo desse tipo de interface:

```

from selenium import webdriver
from selenium.webdriver.remote.webelement import WebElement
from selenium.webdriver import ActionChains

driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get('http://pythonscraping.com/pages/javascript/draggableDemo.html')

print(driver.find_element_by_id('message').text)

element = driver.find_element_by_id('draggable')
target = driver.find_element_by_id('div2')
actions = ActionChains(driver)
actions.drag_and_drop(element, target).perform()

```

```
print(driver.find_element_by_id('message').text)
```

Duas mensagens são exibidas na div `message` da página de demonstração. A primeira contém:

```
Prove you are not a bot, by dragging the square from the blue area to the red area!
```

Então, logo depois que a tarefa é concluída, o conteúdo é exibido mais uma vez, no qual agora se lê:

```
You are definitely not a bot!
```

Conforme a página de demonstração sugere, arrastar elementos para provar que você não é um bot é recorrente em muitos CAPTCHAs. Embora os bots sejam capazes de arrastar objetos por aí há muito tempo (é apenas uma questão de clicar, segurar e mover), de certo modo, a ideia de usar “arraste isto” como uma verificação para saber se é um ser humano fazendo a operação simplesmente não morre.

Além do mais, essas bibliotecas de CAPTCHAs para arrastar raramente usam alguma tarefa difícil para os bots, como “arraste a imagem do gatinho sobre a imagem da vaca” (que exige identificar as figuras como “um gatinho” e “uma vaca” ao interpretar as instruções); em vez disso, em geral, elas envolvem ordenação de números ou outra tarefa razoavelmente trivial, como no exemplo anterior.

É claro que sua robustez está no fato de haver muitas variações e elas não serem usadas com frequência; é improvável que alguém se preocupe em criar um bot capaz de derrotar todas elas. De qualquer modo, esse exemplo deve bastar para mostrar por que jamais devemos usar essa técnica em sites de grande porte.

### **Capturando imagens de tela**

Além dos recursos usuais de teste, o Selenium tem um truque interessante na manga, o qual pode facilitar um pouco seus testes (ou impressionar seu chefe): imagens de tela. Sim, evidências fotográficas da execução dos testes de unidade podem ser geradas sem a necessidade de pressionar a tecla `PrtScn`:

```
driver = webdriver.PhantomJS()
driver.get('http://www.pythonscraping.com/')
driver.get_screenshot_as_file('tmp/pythonscraping.png')
```

Esse script acessa <http://pythonscraping.com> e então armazena uma imagem da tela da página inicial na pasta `tmp` local (a pasta já deve existir para que a imagem seja armazenada corretamente). As imagens de tela podem ser

salvas em diversos formatos.

## unittest ou Selenium?

O rigor sintático e a verbosidade do `unittest` Python talvez sejam desejáveis à maioria das suítes de teste grandes, enquanto a flexibilidade e a eficácia de um teste Selenium talvez sejam a sua única opção para testar algumas funcionalidades dos sites. Qual deles devemos usar?

Eis o segredo: você não precisa escolher. O Selenium pode ser usado com facilidade para obter informações sobre um site, e o `unittest` pode avaliar se essas informações atendem aos critérios para passar no teste. Não há motivos para não importar as ferramentas do Selenium no `unittest` de Python, combinando o melhor dos dois mundos.

Por exemplo, o script a seguir cria um teste de unidade para uma interface com a operação de arrastar em um site, confirmando se ele mostra que “You are not a bot!” (Você não é um bot!) corretamente, depois que um elemento é arrastado para outro:

```
from selenium import webdriver
from selenium.webdriver.remote.webelement import WebElement
from selenium.webdriver import ActionChains
import unittest

class TestDragAndDrop(unittest.TestCase):
    driver = None
    def setUp(self):
        self.driver = webdriver.PhantomJS(executable_path='<Path to PhantomJS>')
        url = 'http://pythonscraping.com/pages/javascript/draggableDemo.html'
        self.driver.get(url)

    def tearDown(self):
        print("Tearing down the test")

    def test_drag(self):
        element = self.driver.find_element_by_id('draggable')
        target = self.driver.find_element_by_id('div2')
        actions = ActionChains(self.driver)
        actions.drag_and_drop(element, target).perform()
        self.assertEqual('You are definitely not a bot!',
            self.driver.find_element_by_id('message').text)

if __name__ == '__main__':
    unittest.main(argv=[''], exit=False)
```

Praticamente tudo em um site pode ser testado com a combinação entre o `unittest` Python e o Selenium. Com efeito, se os combinarmos com algumas

das bibliotecas de processamento de imagens do *Capítulo 13*, podemos até mesmo obter uma imagem da tela do site e testar o que ela deve conter, pixel a pixel!

# Web Crawling em paralelo

O web crawling é rápido. Pelo menos, em geral, é muito mais rápido do que contratar uma dúzia de estagiários para copiar dados da internet manualmente! É claro que o progresso da tecnologia e a tendência ao hedonismo farão com que, em determinado momento, nem ele seja “rápido o suficiente”. É nessa hora que as pessoas geralmente começam a olhar para o processamento distribuído.

De modo diferente da maioria das demais áreas de tecnologia, muitas vezes não é possível melhorar o web crawling apenas “jogando mais ciclos para o problema”. Executar um processo é rápido; executar dois processos não é necessariamente duas vezes mais rápido. Executar três processos pode fazer você ser banido do servidor remoto sobrecarregando com todas as suas requisições!

No entanto, em algumas situações, um web crawling em paralelo, isto é, a execução de threads/processos em paralelo, ainda pode ser vantajosa para:

- coletar dados de várias fontes (vários servidores remotos) em vez de uma só;
- executar operações longas/complexas nos dados coletados (como análise de imagens ou OCR) que poderiam ser feitas em paralelo com a busca dos dados;
- coletar dados de um web service grande para o qual você paga por consulta, ou no qual criar várias conexões com o serviço esteja dentro dos limites de seu contrato de uso.

## Processos versus threads

Python aceita programação tanto com multiprocesso como com multithreading. Em última análise, ambas têm o mesmo objetivo: executar duas tarefas de programação ao mesmo tempo, em vez de executar o programa do modo linear mais tradicional.

Em ciência da computação, cada processo que executa em um sistema operacional pode ter várias threads. Cada processo tem a própria memória alocada, o que significa que várias threads podem acessar essa mesma memória, enquanto vários processos não podem fazer isso, e devem transmitir suas informações de forma explícita.

Usar programação multithreaded para executar tarefas em threads separadas com memória compartilhada em geral é considerado mais fácil que fazer programação com multiprocesso. Todavia, essa conveniência tem um custo.

O GIL (Global Interpreter Lock, ou Trava Global do Interpretador) de Python atua de modo a evitar que as threads executem a mesma linha de código ao mesmo tempo. O GIL garante que a memória comum compartilhada por todos os processos não seja corrompida (por exemplo, com bytes na memória gravados pela metade com um valor e com outro valor na outra metade). Essa trava possibilita escrever um programa multithreaded e saber o que teremos na mesma linha, mas também tem o potencial de criar gargalos.

## Crawling com várias threads

Python 3.x usa o módulo `_thread`, considerado obsoleto.

O exemplo a seguir mostra o uso de várias threads para executar uma tarefa:

```
import _thread
import time

def print_time(threadName, delay, iterations):
    start = int(time.time())
    for i in range(0, iterations):
        time.sleep(delay)
        seconds_elapsed = str(int(time.time()) - start)
        print("{} {}".format(seconds_elapsed, threadName))

try:
    _thread.start_new_thread(print_time, ('Fizz', 3, 33))
    _thread.start_new_thread(print_time, ('Buzz', 5, 20))
    _thread.start_new_thread(print_time, ('Counter', 1, 100))
except:
    print('Error: unable to start thread')

while 1:
    pass
```

Essa é uma referência ao clássico [teste de programação FizzBuzz](http://wiki.c2.com/?FizzBuzzTest) (<http://wiki.c2.com/?FizzBuzzTest>), com uma saída um pouco mais extensa:

```
1 Counter
2 Counter
3 Fizz
3 Counter
4 Counter
5 Buzz
5 Counter
6 Fizz
6 Counter
```

O script inicia três threads, uma que exhibe “Fizz” a cada três segundos, outra que exhibe “Buzz” a cada cinco segundos e uma terceira que exhibe “Counter” a cada segundo.

Depois que as threads são iniciadas, a thread principal de execução alcança um laço `while 1` que mantém o programa (e suas threads filhas) em execução até o usuário teclar Ctrl-C para interrompê-lo.

Em vez de exhibir fizzes e buzzes, podemos executar uma tarefa útil nas threads, por exemplo, rastrear um site:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import random

import _thread
import time

def get_links(thread_name, bs):
    print('Getting links in {}'.format(thread_name))
    return bs.find('div', {'id':'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))

# Define uma função para a thread
def scrape_article(thread_name, path):
    html = urlopen('http://en.wikipedia.org{}'.format(path))
    time.sleep(5)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    print('Scraping {} in thread {}'.format(title, thread_name))
    links = get_links(thread_name, bs)
    if len(links) > 0:
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        print(newArticle)
        scrape_article(thread_name, newArticle)

# Cria duas threads conforme definidas a seguir
```

```

try:
    _thread.start_new_thread(scrape_article, ('Thread 1', '/wiki/Kevin_Bacon',))
    _thread.start_new_thread(scrape_article, ('Thread 2', '/wiki/Monty_Python',))
except:
    print ('Error: unable to start threads')

while 1:
    pass

```

Observe a inclusão desta linha:

```
time.sleep(5)
```

Por estarmos rastreando a Wikipédia quase duas vezes mais rápido do que faríamos com apenas uma thread, a inclusão dessa linha evita que o script imponha uma carga muito alta aos servidores da Wikipédia. Na prática, se a execução for em um servidor para o qual o número de requisições não seja um problema, essa linha deve ser removida.

E se quiséssemos modificar um pouco esse código para manter o controle dos artigos que as threads acessaram em conjunto até agora, de modo que nenhum artigo seja acessado duas vezes? Podemos usar uma lista em um ambiente com várias threads, do mesmo modo que a usamos em um ambiente com uma só thread:

```

visited = []
def get_links(thread_name, bs):
    print('Getting links in {}'.format(thread_name))
    links = bs.find('div', {'id': 'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link for link in links if link not in visited]

def scrape_article(thread_name, path):
    visited.append(path)

```

Observe que estamos concatenando o path à lista de paths visitados como a primeira ação executada por `scrape_article`. Isso reduz, mas não elimina totalmente as chances de que o scraping do artigo seja feito duas vezes.

Se você estivesse sem sorte, as duas threads ainda poderiam deparar com o mesmo path no mesmo instante, ambas veriam que ele não está na lista de paths visitados e, na sequência, acrescentariam esse path na lista e fariam o seu scraping ao mesmo tempo. Na prática, porém, isso é improvável por causa da velocidade da execução e do número de páginas que a Wikipédia contém.

Esse é um exemplo de uma *condição de concorrência* (race condition). Condições de concorrência podem ser complicadas de depurar, mesmo para programadores experientes, portanto é importante avaliar o código

para ver se há situações como essa em potencial, estimar sua probabilidade e antecipar-se à gravidade de seu impacto.

No caso dessa condição de concorrência em particular, em que o scraper acessa a mesma página duas vezes, talvez não compense reescrever o código.

## Condições de concorrência e filas

Apesar de ser possível fazer uma comunicação entre threads usando listas, elas não foram concebidas especificamente para isso, e seu uso indevido pode facilmente causar lentidão na execução do programa ou até mesmo erros resultantes de condições de concorrência.

As listas são ótimas para concatenação ou leitura, mas não são tão boas quando se trata de remover itens de posições arbitrárias, sobretudo do início. Usar uma linha como:

```
myList.pop(0)
```

exige, na verdade, que Python reescreva a lista toda, deixando mais lenta a execução do programa.

O mais perigoso, porém, é que as listas favorecem a escrita acidental de uma linha que não é segura para thread (thread-safe). Por exemplo:

```
myList[len(myList)-1]
```

na verdade talvez não devolva o último item da lista em um ambiente com várias threads, ou poderá até mesmo lançar uma exceção se o valor de `len(myList)-1` for calculado logo antes de outra operação que modifique a lista.

Pode-se argumentar que a instrução anterior deveria ter sido escrita de modo mais “pythônico” como `myList[-1]` e, é claro, ninguém *já* escreveu acidentalmente um código não pythônico em um momento de fraqueza (em particular, não os desenvolvedores Java que se recordem da época em que usavam padrões como `myList[myList.length-1]` )! Contudo, mesmo que seu código seja irrepreensível, considere estas outras formas de linhas não seguras para thread envolvendo listas:

```
my_list[i] = my_list[i] + 1  
my_list.append(my_list[-1])
```

Ambas podem resultar em uma condição de concorrência capaz de gerar resultados inesperados. Portanto, vamos abandonar as listas e passar mensagens às threads usando variáveis que não sejam listas!

```
# Lê a mensagem da lista global
```

```
my_message = global_message
# Escreve uma mensagem de volta
global_message = 'I've retrieved the message'
# faz algo com my_message
```

Parece razoável, até você perceber que poderia ter inadvertidamente sobrescrito outra mensagem que estivesse chegando de outra thread, no instante entre a primeira e a segunda linha, com o texto “I’ve got your message” (Recebi sua mensagem). Agora, então, você só precisa construir uma série sofisticada de objetos de mensagem pessoais para cada thread, com alguma lógica para identificar quem recebe o quê... Ou poderíamos usar o módulo embutido `queue` exatamente para isso.

As filas (queues) são objetos semelhantes a listas, que funcionam com uma abordagem FIFO (First In First Out, ou o Primeiro que Entra é o Primeiro que Sai) ou LIFO (Last In First Out, ou o Último que Entra é o Primeiro que Sai). Uma fila recebe mensagens de qualquer thread por meio de `queue.put('My message')` e pode transmitir a mensagem para qualquer thread que chame `queue.get()`.

As filas não foram projetadas para armazenar dados estáticos, mas para transmiti-los de forma segura para threads. Depois de serem obtidos da fila, os dados só existirão na thread que os adquiriu. Por esse motivo, as filas são comumente usadas para delegar tarefas ou enviar notificações temporárias.

Isso pode ser útil no web crawling. Por exemplo, suponha que você queira fazer a persistência dos dados coletados pelo seu scraper em um banco de dados, e queira que cada thread faça essa persistência com rapidez. Uma única conexão compartilhada entre todas as threads pode causar problemas (uma única conexão não possibilita tratar requisições em paralelo), mas não faz sentido conceder a cada thread de scraping uma conexão própria com o banco de dados). À medida que seu scraper crescer (em algum momento, você pode coletar dados de uma centena de sites diferentes com cem threads diferentes), isso poderá significar muitas conexões com o banco de dados, na maior parte do tempo ociosas, fazendo apenas escritas ocasionais depois que uma página é carregada.

Em vez disso, poderíamos ter um número menor de threads para o banco de dados, cada uma com a própria conexão, esperando obter itens de uma fila para armazená-los. Com isso, teremos um conjunto de conexões com o banco de dados muito mais fácil de administrar.

```
from urllib.request import urlopen
```

```

from bs4 import BeautifulSoup
import re
import random
import _thread
from queue import Queue
import time
import pymysql

def storage(queue):
    conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
        user='root', passwd='', db='mysql', charset='utf8')
    cur = conn.cursor()
    cur.execute('USE wiki_threads')
    while 1:
        if not queue.empty():
            article = queue.get()
            cur.execute('SELECT * FROM pages WHERE path = %s',
                (article["path"]))
            if cur.rowcount == 0:
                print("Storing article {}".format(article["title"]))
                cur.execute('INSERT INTO pages (title, path) VALUES (%s, %s)', \
                    (article["title"], article["path"]))
                conn.commit()
            else:
                print("Article already exists: {}".format(article['title']))

visited = []
def getLinks(thread_name, bs):
    print('Getting links in {}'.format(thread_name))
    links = bs.find('div', {'id': 'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link for link in links if link not in visited]

def scrape_article(thread_name, path, queue):
    visited.append(path)
    html = urlopen('http://en.wikipedia.org{}'.format(path))
    time.sleep(5)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    print('Added {} for storage in thread {}'.format(title, thread_name))
    queue.put({"title":title, "path":path})
    links = getLinks(thread_name, bs)
    if len(links) > 0:
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        scrape_article(thread_name, newArticle, queue)

queue = Queue()
try:
    _thread.start_new_thread(scrape_article, ('Thread 1',
        '/wiki/Kevin_Bacon', queue,))

```

```

        _thread.start_new_thread(scrape_article, ('Thread 2',
        '/wiki/Monty_Python', queue,))
        _thread.start_new_thread(storage, (queue,))
except:
    print ('Error: unable to start threads')

while 1:
    pass

```

Esse script cria três threads: duas para coletar dados de páginas da Wikipédia em um percurso aleatório, e uma terceira para armazenar os dados coletados em um banco de dados MySQL. Para mais informações sobre MySQL e armazenagem de dados, consulte o *Capítulo 6*.

## Módulo threading

O módulo Python `_thread` é um módulo de nível razoavelmente baixo, que permite administrar suas threads de forma minuciosa, mas não oferece muitas funções de nível alto para facilitar a sua vida. O módulo `threading` é uma interface de nível mais alto que permite usar threads de modo mais organizado, ao mesmo tempo que ainda expõe todos os recursos do módulo `_thread` subjacente.

Por exemplo, podemos usar funções estáticas como `enumerate` para obter uma lista de todas as threads ativas inicializadas com o módulo `threading` sem a necessidade de manter o controle delas por conta própria. A função `activeCount`, de modo semelhante, informa o número total de threads. Muitas funções de `_thread` recebem nomes mais convenientes ou fáceis de lembrar, como `currentThread` em vez de `get_ident` para obter o nome da thread atual.

Eis um exemplo simples de uso de `threading`:

```

import threading
import time

def print_time(threadName, delay, iterations):
    start = int(time.time())
    for i in range(0,iterations):
        time.sleep(delay)
        seconds_elapsed = str(int(time.time()) - start)
        print ('{} {}'.format(seconds_elapsed, threadName))

threading.Thread(target=print_time, args=('Fizz', 3, 33)).start()
threading.Thread(target=print_time, args=('Buzz', 5, 20)).start()
threading.Thread(target=print_time, args=('Counter', 1, 100)).start()

```

A mesma saída “FizzBuzz” do exemplo anterior simples que usava `_thread` é

gerada.

Um dos aspectos interessantes sobre o módulo `threading` é a facilidade de criar dados locais de thread indisponíveis a outras threads. Pode ser um recurso conveniente se houver várias threads, cada uma fazendo scraping de um site diferente, e cada uma mantendo a própria lista local das páginas acessadas.

Esses dados locais podem ser criados em qualquer ponto da função de thread com uma chamada a `threading.local()`:

```
import threading

def crawler(url):
    data = threading.local()
    data.visited = []
    # Rastreia o site

threading.Thread(target=crawler, args=('http://brookings.edu')).start()
```

Isso resolve o problema das condições de concorrência que ocorrem entre objetos compartilhados nas threads. Sempre que um objeto não tiver de ser compartilhado, não o compartilhe e mantenha-o na memória local da thread. Para compartilhar objetos de modo seguro entre as threads, podemos fazer uso de `Queue`, que vimos na seção anterior.

O módulo `threading` atua como uma espécie de babá de threads, e pode ser bastante personalizado para definir o que essa atividade de babá implica. A função `isAlive`, por padrão, verifica se a thread continua ativa. Será verdadeira até uma thread acabar de fazer seu rastreamento (ou falhar).

Com frequência, os crawlers são projetados para executar por muito tempo. O método `isAlive` pode garantir que, se uma thread falhar, ela seja reiniciada:

```
threading.Thread(target=crawler)
t.start()

while True:
    time.sleep(1)
    if not t.isAlive():
        t = threading.Thread(target=crawler)
        t.start()
```

Outros métodos de monitoração podem ser acrescentados se o objeto `threading.Thread` for estendido:

```
import threading
import time
```

```

class Crawler(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.done = False

    def isDone(self):
        return self.done

    def run(self):
        time.sleep(5)
        self.done = True
        raise Exception('Something bad happened!')

t = Crawler()
t.start()

while True:
    time.sleep(1)
    if t.isDone():
        print('Done')
        break
    if not t.isAlive():
        t = Crawler()
        t.start()

```

Essa nova classe `crawler` contém um método `isDone` que pode ser usado para verificar se o crawler terminou de fazer o rastreamento. Pode ser útil se houver alguns métodos adicionais de logging que devam ser terminados para que a thread seja encerrada, mas o trabalho de rastreamento já tenha sido feito. Em geral, `isDone` pode ser substituído por algum tipo de status ou medida de progresso – quantas páginas foram registradas, ou a página atual, por exemplo.

Qualquer exceção lançada por `crawler.run` fará a classe ser reiniciada até `isDone` ser `True` e o programa terminar.

Estender `threading.Thread` em suas classes de crawler pode melhorar sua robustez e flexibilidade, bem como sua capacidade de monitorar qualquer propriedade de vários crawlers ao mesmo tempo.

## Rastreamento com multiprocessamento

O módulo `Processing` de Python cria novos objetos de processo que podem ser iniciados e cuja junção (`join`) pode ser feita a partir do processo principal. O código a seguir usa o exemplo de FizzBuzz da seção sobre processos com `threading` para uma demonstração.

```

from multiprocessing import Process

```

```

import time

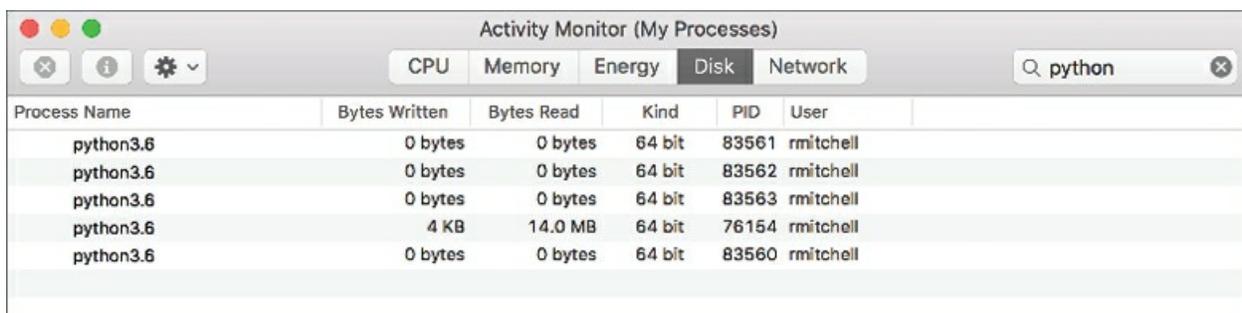
def print_time(threadName, delay, iterations):
    start = int(time.time())
    for i in range(0,iterations):
        time.sleep(delay)
        seconds_elapsed = str(int(time.time()) - start)
        print (threadName if threadName else seconds_elapsed)

processes = []
processes.append(Process(target=print_time, args=('Counter', 1, 100)))
processes.append(Process(target=print_time, args=('Fizz', 3, 33)))
processes.append(Process(target=print_time, args=('Buzz', 5, 20)))

for p in processes:
    p.start()
for p in processes:
    p.join()

```

Lembre-se de que cada processo é tratado como um programa individual independente pelo sistema operacional. Se observar seus processos com o monitor de atividades de seu sistema operacional ou com o gerenciador de tarefas, você deverá ver o reflexo disso, como mostra a Figura 16.1.



Process Name	Bytes Written	Bytes Read	Kind	PID	User
python3.6	0 bytes	0 bytes	64 bit	83561	rmitchell
python3.6	0 bytes	0 bytes	64 bit	83562	rmitchell
python3.6	0 bytes	0 bytes	64 bit	83563	rmitchell
python3.6	4 KB	14.0 MB	64 bit	76154	rmitchell
python3.6	0 bytes	0 bytes	64 bit	83560	rmitchell

*Figura 16.1 – Cinco processos Python em execução enquanto o FizzBuzz executa.*

O quarto processo com PID 76154 é uma instância do notebook Jupyter em execução, que deverá ser vista caso esse código seja executado a partir do notebook iPython. O quinto processo, 83560, é a thread principal de execução, iniciada quando o programa é executado. Os PIDs são alocados sequencialmente pelo sistema operacional. A menos que haja outro programa que aloque rapidamente um PID enquanto o script FizzBuzz estiver executando, você deverá ver três outros PIDs sequenciais – nesse caso, 83561, 83562 e 83563.

Esses PIDs também podem ser encontrados com um código que use o

módulo os:

```
import os
...
# exibe o PID filho
os.getpid()
# exibe o PID pai
os.getppid()
```

Cada processo em seu programa deve exibir um PID diferente para a linha `os.getpid()`, mas exibirá o mesmo PID pai em `os.getppid()`.

Tecnicamente, há duas linhas de código desnecessárias no programa anterior em particular. Se a instrução final de junção não for incluída:

```
for p in processes:
    p.join()
```

o processo pai ainda terminará e encerrará os processos filhos com ele de modo automático. No entanto, essa junção é necessária caso você queira executar um código depois que esses processos filhos forem concluídos.

Por exemplo:

```
for p in processes:
    p.start()
print('Program complete')
```

Se a instrução de junção não for incluída, o resultado será este:

```
Program complete
1
2
```

Se for incluída, o programa espera que cada um dos processos termine antes de prosseguir:

```
for p in processes:
    p.start()

for p in processes:
    p.join()
print('Program complete')
...
Fizz
99
Buzz
100
Program complete
```

Se quiser interromper prematuramente a execução do programa, é claro que podemos usar Ctrl-C para encerrar o processo pai. O término do processo pai também fará com que qualquer processo filho gerado a partir dele termine, portanto usar Ctrl-C é seguro, e você não tem de se

preocupar com o fato de restar acidentalmente algum processo em execução em segundo plano.

## Rastreamento da Wikipédia com multiprocessamento

O exemplo de rastreamento da Wikipedia com várias threads pode ser modificado de modo a usar processos separados em vez de threads distintas:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import random

from multiprocessing import Process
import os
import time

visited = []
def get_links(bs):
    print('Getting links in {}'.format(os.getpid()))
    links = bs.find('div', {'id': 'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link for link in links if link not in visited]

def scrape_article(path):
    visited.append(path)
    html = urlopen('http://en.wikipedia.org{}'.format(path))
    time.sleep(5)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    print('Scraping {} in process {}'.format(title, os.getpid()))
    links = get_links(bs)
    if len(links) > 0:
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        print(newArticle)
        scrape_article(newArticle)

processes = []
processes.append(Process(target=scrape_article, args=('/wiki/Kevin_Bacon',)))
processes.append(Process(target=scrape_article, args=('/wiki/Monty_Python',)))

for p in processes:
    p.start()
```

Novamente, estamos atrasando o processo do scraper de modo artificial incluindo um `time.sleep(5)`, de modo que esse código seja usado como exemplo, sem impor uma carga absurdamente alta aos servidores da Wikipédia.

Nesse caso, estamos substituindo o `thread_name` definido pelo usuário, passado como argumento, por `os.getpid()`, que não precisa ser passado como argumento e pode ser acessado de qualquer lugar.

Uma saída como esta é gerada:

```
Scraping Kevin Bacon in process 84275
Getting links in 84275
/wiki/Philadelphia
Scraping Monty Python in process 84276
Getting links in 84276
/wiki/BBC
Scraping BBC in process 84276
Getting links in 84276
/wiki/Television_Centre,_Newcastle_upon_Tyne
Scraping Philadelphia in process 84275
```

Teoricamente, fazer crawling em processos separados é um pouco mais rápido do que usar threads diferentes por dois motivos principais:

- Os processos não estão sujeitos a travar por causa do GIL, e podem executar as mesmas linhas de código e modificar o mesmo objeto (na verdade, instâncias diferentes do mesmo objeto) ao mesmo tempo.
- Os processos podem executar em vários núcleos (cores) de CPU, o que pode significar vantagens quanto à velocidade se cada um de seus processos ou threads fizer uso intenso do processador.

No entanto, essas vantagens vêm acompanhadas de uma grande desvantagem. No programa anterior, todos os URLs encontrados são armazenados em uma lista `visited` global. Quando estávamos usando várias threads, essa lista era compartilhada entre elas; e uma thread, na ausência de uma condição de concorrência rara, não podia acessar uma página que já tivesse sido acessada por outra thread. No entanto, cada processo agora tem a própria versão independente da lista de páginas acessadas, e está livre para visitar páginas que já tenham sido acessadas por outros processos.

## Comunicação entre processos

Os processos atuam em sua própria memória independente, e isso pode causar problemas se você quiser que elas compartilhem informações.

Ao modificar o exemplo anterior para que exiba a lista atual de páginas visitadas, podemos ver este princípio em ação:

```
def scrape_article(path):
    visited.append(path)
```

```
print("Process {} list is now: {}".format(os.getpid(), visited))
```

O resultado é uma saída como esta:

```
Process 84552 list is now: ['/wiki/Kevin_Bacon']
Process 84553 list is now: ['/wiki/Monty_Python']
Scraping Kevin Bacon in process 84552
Getting links in 84552
/wiki/Desert_Storm
Process 84552 list is now: ['/wiki/Kevin_Bacon', '/wiki/Desert_Storm']
Scraping Monty Python in process 84553
Getting links in 84553
/wiki/David_Jason
Process 84553 list is now: ['/wiki/Monty_Python', '/wiki/David_Jason']
```

Contudo, há uma forma de compartilhar informações entre processos na mesma máquina, usando dois tipos de objetos Python: filas e pipes.

Uma *fila* (queue) é parecida com a fila de threading que vimos antes. Informações podem ser inseridas nela por um processo e removidas por outro. Depois que essas informações forem removidas, elas desaparecem da fila. Como foram projetadas como um método para “transmissão de dados temporários”, as filas não são muito apropriadas para armazenar uma referência estática, como uma “lista de páginas web que já foram visitadas”.

Mas e se essa lista de páginas web fosse substituída por algum tipo de delegação para scraping? Os scrapers poderiam retirar uma tarefa de uma fila na forma de um path para coletar dados (por exemplo, */wiki/Monty\_Python*) e, por sua vez, adicionar uma lista de “URLs encontrados” em uma fila separada, que seria processada pelo código de delegação de scraping, de modo que somente novos URLs seriam adicionados na primeira fila de tarefas:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import random
from multiprocessing import Process, Queue
import os
import time

def task_delegator(taskQueue, urlsQueue):
    # Inicializa com uma tarefa para cada processo
    visited = ['/wiki/Kevin_Bacon', '/wiki/Monty_Python']
    taskQueue.put('/wiki/Kevin_Bacon')
    taskQueue.put('/wiki/Monty_Python')
```

```

while 1:
    # Verifica se há novos links em urlsQueue
    # para serem processados
    if not urlsQueue.empty():
        links = [link for link in urlsQueue.get() if link not in visited]
        for link in links:
            # Adiciona um novo link em taskQueue
            taskQueue.put(link)

def get_links(bs):
    links = bs.find('div', {'id':'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link.attrs['href'] for link in links]

def scrape_article(taskQueue, urlsQueue):
    while 1:
        while taskQueue.empty():
            # Dorme por 100 ms enquanto espera a fila de tarefas
            # Isso deve ser raro
            time.sleep(.1)
        path = taskQueue.get()
        html = urlopen('http://en.wikipedia.org{}'.format(path))
        time.sleep(5)
        bs = BeautifulSoup(html, 'html.parser')
        title = bs.find('h1').get_text()
        print('Scraping {} in process {}'.format(title, os.getpid()))
        links = get_links(bs)
        # Envia ao código de delegação para processamento
        urlsQueue.put(links)

processes = []
taskQueue = Queue()
urlsQueue = Queue()
processes.append(Process(target=task_delegator, args=(taskQueue, urlsQueue,)))
processes.append(Process(target=scrape_article, args=(taskQueue, urlsQueue,)))
processes.append(Process(target=scrape_article, args=(taskQueue, urlsQueue,)))

for p in processes:
    p.start()

```

Há algumas diferenças estruturais entre esse scraper e aqueles criados originalmente. Em vez de cada processo ou thread seguir o próprio percurso aleatório começando pelo ponto de partida que receberam, eles atuam em conjunto para fazer um rastreamento completo do site. Cada processo pode extrair qualquer “tarefa” da fila, e não apenas os links que eles mesmos encontraram.

## Rastreamento com multiprocesso – outra abordagem

Todas as abordagens discutidas para rastreamento com várias threads e com vários processos partem do pressuposto de que você precisa de alguma espécie de “orientação parental” sobre as threads e processos filhos. Podemos iniciar ou terminar todos de uma só vez, e podemos enviar mensagens ou compartilhar memória entre eles.

Mas e se seu scraper for projetado de modo que nenhuma orientação ou comunicação seja necessária? Talvez ainda não haja muitos motivos para começar a enlouquecer com `import _thread`.

Por exemplo, suponha que você queira rastrear dois sites semelhantes em paralelo. Temos um crawler implementado, capaz de rastrear qualquer um desses sites, e isso é determinado por uma pequena diferença na configuração ou talvez por um argumento na linha de comando. Não há absolutamente motivo algum para que você não possa fazer simplesmente o seguinte:

```
$ python my_crawler.py website1  
$ python my_crawler.py website2
```

E *voilà*, você acabou de iniciar um web crawler com multiprocesso, ao mesmo tempo em que evitou um overhead na CPU para manter um processo pai a fim de iniciá-los!

É claro que essa abordagem tem suas desvantagens. Se quiser executar dois web crawlers no *mesmo* site dessa maneira, será necessário ter alguma forma de garantir que eles não comecem acidentalmente a fazer scraping das mesmas páginas. A solução poderia ser a criação de uma regra de URL (“o crawler 1 faz scraping das páginas de blog, o crawler 2 faz scraping das páginas de produto”) ou dividir o site de alguma maneira.

Como alternativa, poderíamos cuidar dessa coordenação com algum tipo de banco de dados intermediário. Antes de acessar um novo link, o crawler poderia fazer uma requisição ao banco de dados para perguntar: “Essa página já foi rastreada?”. O crawler usa o banco de dados como um sistema de comunicação entre processos. É claro que, sem uma apreciação cuidadosa, esse método pode resultar em condições de concorrência ou em atrasos se a conexão com o banco de dados for lenta (provavelmente só será um problema se a conexão for feita com um banco de dados remoto).

Talvez você perceba também que esse método não é muito escalável. Usar o módulo `Process` permite aumentar ou diminuir dinamicamente o número

de processos rastreando o site, ou até mesmo armazenando dados. Inicialmente manualmente exige que uma pessoa execute fisicamente o script ou que haja um script de gerenciamento separado (seja um script bash, um cron job ou outro meio) para isso.

No entanto, esse é um método que já usei com muito sucesso no passado. Para projetos pequenos, executados uma só vez, é uma ótima maneira de obter muitas informações com rapidez, sobretudo se houver vários sites.

# Fazendo scraping remotamente

No último capítulo, vimos como executar web scrapers com várias threads e processos, em que a comunicação estava um tanto quanto limitada ou tinha de ser planejada com cuidado. Este capítulo leva esse conceito à sua conclusão lógica – executar crawlers não só em processos separados, mas em máquinas totalmente distintas.

O fato de este capítulo ser um dos últimos do livro, de certo modo, é apropriado. Até agora, vínhamos executando todas as aplicações Python a partir da linha de comando, confinados ao seu computador local. É claro que você pode ter instalado o MySQL em uma tentativa de reproduzir o ambiente de um servidor na vida real. A situação, porém, não é a mesma. Como diz o ditado: “Se você ama alguém, conceda-lhe a liberdade”.

Este capítulo descreve vários métodos para executar scripts a partir de máquinas diferentes, ou apenas de endereços IP distintos em sua própria máquina. Embora possa se sentir tentado a menosprezar este passo como não *necessário* neste momento, você poderia se surpreender ao saber como é fácil começar a trabalhar com as ferramentas de que já dispõe (por exemplo, um site pessoal em uma conta de hospedagem paga), e como sua vida será muito mais simples se você parar de tentar executar seus scrapers Python em seu notebook.

## Por que usar servidores remotos?

Embora usar um servidor remoto pareça um passo óbvio quando lançamos uma aplicação web com vistas a um público mais amplo, com frequência, executamos as ferramentas que construímos para nós mesmos localmente. As pessoas que decidem enviá-las para uma plataforma remota em geral baseiam sua decisão em dois motivos principais: necessidade de ter mais potência e flexibilidade e de usar um endereço IP alternativo.

## Evitando o bloqueio de endereços IP

Ao construir web scrapers, a regra geral é: quase tudo pode ser forjado. Você pode enviar emails de endereços que não são seus, automatizar dados de movimentação de mouse a partir da linha de comando ou até mesmo horrorizar os administradores web enviando-lhes tráfego de site do Internet Explorer 5.0.

A única informação que não pode ser forjada é o seu endereço IP. Qualquer pessoa pode enviar uma carta cujo remetente seja: “Presidente, 1600 Pennsylvania Avenue Northwest, Washington, DC 20500”. No entanto, se a carta tiver sido postada em Albuquerque no Novo México, é quase certo que você não estará se correspondendo com o Presidente dos Estados Unidos<sup>1</sup>.

A maior parte dos esforços para impedir que os scrapers acessem sites tem como foco detectar as diferenças entre seres humanos e bots. Ir tão longe a ponto de bloquear endereços IP é um pouco como um fazendeiro abrir mão de usar pesticidas em favor de simplesmente fazer uma queimada no campo. É um método usado como último recurso, porém eficaz, para descartar pacotes enviados por endereços IP que causem problemas. No entanto, essa solução apresenta problemas:

- É difícil manter listas de acesso com endereços IP. Embora sites grandes geralmente tenham seus próprios programas para automatizar parte do gerenciamento rotineiro dessas listas (bots bloqueando bots!), alguém precisa ocasionalmente as verificar ou, no mínimo, monitorar seu crescimento para saber se há problemas.
- Cada endereço acrescenta uma pequena quantidade de tempo de processamento na recepção dos pacotes, pois é necessário que o servidor verifique os pacotes recebidos junto à lista para decidir se deve aprová-los. Vários endereços multiplicados por vários pacotes podem resultar em uma quantidade elevada rapidamente. Para poupar tempo de processamento e evitar complexidades, os administradores, com frequência, agrupam esses endereços IP em blocos e criam regras como “todos os 256 endereços desta faixa estão bloqueados” caso haja infratores com endereços próximos. Isso nos leva ao terceiro ponto.
- O bloqueio de endereços IP pode levar ao bloqueio dos “mocinhos” também. Por exemplo, enquanto fazia minha graduação no Olin College of Engineering, um aluno escreveu um software que tentava manipular votos para conteúdos populares no [digg.com](http://digg.com) (foi antes de o Reddit estar

em alta). Um único endereço IP bloqueado resultou em um dormitório inteiro incapaz de acessar o site. O aluno apenas moveu seu software para outro servidor; nesse ínterim, as páginas do Digg perderam acessos de muitos usuários regulares de seu público-alvo principal.

Apesar das desvantagens, o bloqueio de endereços IP ainda é um método muito comum para os administradores impedirem que web scrapers suspeitos acessem seus servidores. Se um endereço IP for bloqueado, a única solução verdadeira é fazer scraping a partir de um endereço IP diferente. Isso pode ser feito movendo o scraper para um novo servidor ou encaminhando o tráfego por um servidor diferente, com um serviço como o Tor.

### **Portabilidade e extensibilidade**

Algumas tarefas são grandes demais para um computador doméstico e uma conexão com a internet. Ainda que não queira impor uma carga pesada a um único site, você poderia coletar dados de vários deles e precisar de muito mais largura de banda e área de armazenagem do que sua configuração atual é capaz de oferecer.

Além disso, ao se livrar de cargas intensas de processamento, ciclos de máquina de seu computador doméstico estarão livres para tarefas mais importantes (alguém aí quer jogar *World of Warcraft*?) Não será preciso se preocupar em economizar energia elétrica nem manter uma conexão com a internet (inicie sua aplicação em um Starbucks, pegue o notebook e saiba que tudo continuará executando de modo seguro), e os dados coletados poderão ser acessados de qualquer lugar em que houver uma conexão com a internet.

Se você tiver uma aplicação que exija muita capacidade de processamento a ponto de um único computador extragrande não o satisfazer, dê uma olhada também no *processamento distribuído*. Isso permite que várias máquinas trabalhem em paralelo para que seus objetivos sejam atingidos. Como um exemplo simples, poderíamos ter um computador rastreando um conjunto de sites e outro rastreando um segundo conjunto, e ambos armazenariam os dados coletados no mesmo banco de dados.

É claro que, conforme mencionamos nos capítulos anteriores, muitas pessoas podem imitar o que uma pesquisa do Google faz, mas poucas serão capazes de replicar a escala com que ela é feita. O processamento distribuído envolve uma área grande da ciência da computação e está além

do escopo deste livro. No entanto, aprender a iniciar sua aplicação em um servidor remoto é um primeiro passo necessário, e você se surpreenderá com o que os computadores são capazes de fazer hoje em dia.

## Tor

A rede Onion Router (Roteador Cebola), mais conhecida pelo acrônimo *Tor*, é uma rede de servidores voluntários, configurada para encaminhar e reencaminhar tráfego por várias camadas (daí a referência à cebola) de diferentes servidores a fim de ocultar sua origem. Os dados são criptografados antes de entrarem na rede, de modo que, caso algum servidor em particular seja espionado, a natureza da comunicação não poderá ser revelada. Além disso, embora as comunicações de entrada e de saída de qualquer servidor em particular possam ser comprometidas, seria necessário que alguém conhecesse os detalhes da comunicação de entrada e de saída de *todos* os servidores no percurso da comunicação para decifrar os verdadeiros pontos inicial e final de uma comunicação – uma proeza quase impossível.

O Tor é comumente usado por quem trabalha com direitos humanos e denúncias políticas para se comunicar com jornalistas, e recebe boa parte de seus financiamentos do governo norte-americano. É claro que ele também é frequentemente utilizado para atividades ilegais, sendo, desse modo, alvo constante da vigilância do governo (apesar disso, até hoje, a vigilância teve sucesso apenas parcial).

### Limitações ao anonimato do Tor

Embora o motivo de usar o Tor neste livro não seja permanecer totalmente no anonimato, mas alterar o endereço IP, vale a pena investir um pouco de tempo para compreender alguns dos pontos fortes e as limitações do Tor a fim de deixar o tráfego anônimo.

Ainda que, ao usar o Tor, seja possível supor que seu endereço IP de origem, de acordo com um servidor web, não é um endereço IP que seja rastreado até você, qualquer informação compartilhada com esse servidor web poderá expô-lo. Por exemplo, se fizer login em sua conta Gmail e então executar pesquisas incriminadoras no Google, essas pesquisas poderão ser agora associadas à sua identidade.

Indo além do óbvio, porém, mesmo o ato de fazer login no Tor pode ser prejudicial ao seu anonimato. Em dezembro de 2013, um aluno de graduação de Harvard, em uma tentativa de se livrar dos exames finais, enviou uma ameaça de bomba por email para a universidade usando a rede Tor, com uma conta de email anônima. Quando a equipe de TI de Harvard analisou os logs, eles encontraram tráfego de saída para a rede Tor de um único computador, registrado a um aluno conhecido, durante o período em que a ameaça de bomba foi enviada. Embora não pudessem identificar o destino eventual desse tráfego (apenas que ele fora enviado pelo Tor), o fato de os horários terem coincidido e de haver apenas uma máquina logada naquele momento, sem dúvida, foram suficientes para processar o aluno.

Fazer login no Tor não garante a você uma capa de invisibilidade de modo automático nem lhe dá total liberdade para fazer o que bem entender na internet. Embora seja uma ferramenta útil, lembre-se de usá-la com cautela, inteligência e, é claro, moralidade.

Ter o Tor instalado e executando é um requisito para usar Python com o Tor, como veremos na próxima seção. Felizmente, o serviço Tor é muito fácil de instalar e executar. Basta acessar a [página de download do Tor](https://www.torproject.org/download/download) (<https://www.torproject.org/download/download>) e fazer seu download, instalar, abrir e conectar! Tenha em mente que sua velocidade de internet poderá parecer mais lenta quando estiver usando o Tor. Seja paciente – seus dados poderão estar dando a volta ao mundo várias vezes!

## PySocks

O PySocks é um módulo Python excepcionalmente simples, capaz de encaminhar tráfego por meio de servidores proxy, e funciona de modo incrível em conjunto com o Tor. Você pode fazer seu download a partir do [site](https://pypi.org/project/PySocks/1.5.0/) (<https://pypi.org/project/PySocks/1.5.0/>) ou usar qualquer um dos gerenciadores de módulos de terceiros para instalá-lo.

Apesar de não haver muita documentação para esse módulo, é muito simples usá-lo. O serviço Tor deve estar executando na porta 9150 (a porta default) enquanto o código a seguir é executado:

```
import socks
import socket
from urllib.request import urlopen

socks.set_default_proxy(socks.SOCKS5, "localhost", 9150)
socket.socket = socks.socksocket
print(urlopen('http://icanhazip.com').read())
```

O site <http://icanhazip.com> exhibe apenas o endereço IP do cliente conectado ao servidor, e pode ser útil para testes. Quando for executado, esse script deverá exibir um endereço IP que não é o seu.

Se quiser usar o Selenium e o PhantomJS com o Tor, não será necessário ter o PySocks – basta garantir que o Tor esteja executando e acrescentar os parâmetros opcionais em `service_args`, especificando que o Selenium deve se conectar por meio da porta 9150:

```
from selenium import webdriver
service_args = [ '--proxy=localhost:9150', '--proxy-type=socks5', ]
driver = webdriver.PhantomJS(executable_path='<path to PhantomJS>',
                             service_args=service_args)

driver.get('http://icanhazip.com')
print(driver.page_source)
```

```
driver.close()
```

Mais uma vez, esse código deve exibir um endereço IP que não é o seu, mas aquele que seu cliente Tor em execução está usando no momento.

## Hospedagem remota

Apesar de um anonimato completo ter deixado de existir assim que você usou seu cartão de crédito, hospedar seus web scrapers remotamente pode melhorar muito a sua velocidade. Isso ocorre tanto porque você pode comprar tempo em máquinas provavelmente muito mais potentes que a sua, mas também porque a conexão não precisa mais passar pelas camadas de uma rede Tor para alcançar seu destino.

### Executando de uma conta que hospeda sites

Se você tem um site pessoal ou comercial, é provável que já tenha os meios para executar seus web scrapers a partir de um servidor externo. Mesmo com servidores web relativamente bloqueados, nos quais você não tenha acesso à linha de comando, é possível disparar scripts que iniciem e terminem por meio de uma interface web.

Se seu site estiver hospedado em um servidor Linux, o servidor provavelmente já estará executando Python. Se estiver hospedado em um servidor Windows, talvez você não tenha tanta sorte; será necessário verificar especificamente se Python está instalado, ou se o administrador do servidor está disposto a instalá-lo.

A maioria dos pequenos provedores de hospedagem na web tem um software chamado *cPanel*, usado para disponibilizar serviços básicos de administração e informações sobre seu site e os serviços relacionados. Se tiver acesso ao cPanel, certifique-se de que Python está configurado para executar em seu servidor acessando o Apache Handlers e acrescentando um novo handler (caso não houver):

```
Handler: cgi-script  
Extension(s): .py
```

Isso diz ao seu servidor que todos os scripts Python devem ser executados como um *script CGI*. CGI, que significa que *Common Gateway Interface* (Interface Comum de Gateway) é qualquer programa que possa ser executado em um servidor e gere conteúdo a ser exibido em um site de modo dinâmico. Ao definir explicitamente os scripts Python como CGI, você estará dando permissão ao servidor para executá-los, em vez de

simplesmente os exibir em um navegador ou enviar um download ao usuário.

Escreva seu script Python, faça o upload para o servidor e configure as permissões do arquivo com 755 para permitir que seja executado. Para executar o script, vá para o local no qual você fez o upload usando o seu navegador (ou, melhor ainda, escreva um scraper que faça isso). Se estiver preocupado com o fato de o público em geral poder acessar e executar o script, há duas opções:

- Armazene o script em um URL obscuro ou oculto e certifique-se de que não haja links para o script de nenhum outro URL acessível de modo a evitar que as ferramentas de pesquisa o indexem.
- Proteja o script com uma senha, ou exija que uma senha ou um token secreto seja enviado antes que ele seja executado.

É claro que executar um script Python a partir de um serviço especificamente projetado para exibir sites é uma espécie de hack. Por exemplo, é provável que você perceba que seu web scraper-com-site seja um pouco lento para carregar. Com efeito, a página não carrega realmente (de modo completo, com a saída de todas as instruções `print` que tenham sido escritas) até que todo o scraping seja concluído. Isso pode demorar minutos, horas ou jamais terminar, dependendo de como foi escrito. Ainda que, sem dúvida, o script faça o serviço, talvez você queira obter o resultado um pouco mais em tempo real. Para isso, será necessário um servidor projetado para outros usos que não apenas a web.

## **Executando a partir da nuvem**

De volta aos velhos tempos da computação, os programadores pagavam ou reservavam tempo nos computadores para executar seu código. Com o advento dos computadores pessoais, isso se tornou desnecessário – basta escrever e executar um código em seu próprio computador. Atualmente, as ambições das aplicações superaram o desenvolvimento dos microprocessadores a tal ponto que os programadores estão, mais uma vez, passando a usar instâncias de processamento pagas por hora.

Dessa vez, porém, os usuários não estão pagando pelo tempo em uma única máquina física, mas em sua capacidade de processamento equivalente, em geral distribuída em várias máquinas. A estrutura nebulosa desse sistema permite que a capacidade de processamento seja cobrada de

acordo com a demanda dos horários de pico. Por exemplo, a Amazon permite a aquisição de “instâncias spot” (spot instances) quando custos mais baixos são mais importantes do que o imediatismo.

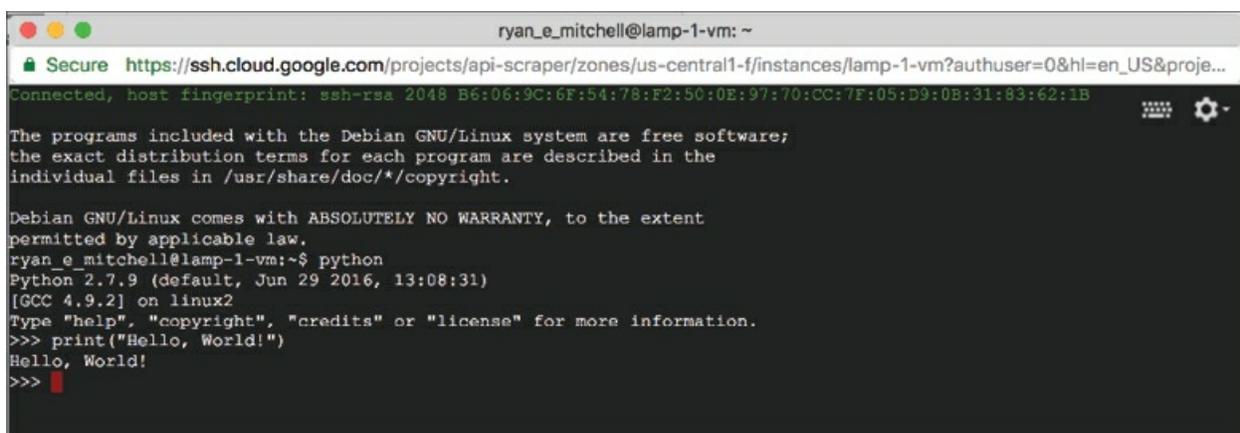
As instâncias de processamento também são mais especializadas, e podem ser selecionadas com base nas necessidades de sua aplicação, com opções como “mais memória” (high memory), “computação acelerada” (fast computing) e “otimizadas para armazenamento” (large storage). Embora os web scrapers em geral não usem muita memória, talvez você queira considerar as opções de otimização para armazenamento ou computação acelerada no lugar de uma instância mais genérica para a sua aplicação de scraping. Se estiver fazendo processamento intensivo de idiomas naturais, trabalho de OCR ou encontrando caminhos (como no problema do Six Degrees of Wikipedia), uma instância de computação acelerada seria apropriada. Se estiver fazendo scraping de grandes quantidades de dados, armazenando arquivos ou efetuando análises de dados em larga escala, talvez queira optar por uma instância com otimização para armazenamento.

Apesar de o céu ser o limite no que concerne aos gastos, atualmente (quando este livro foi escrito), o preço das instâncias começa em apenas 1,3 centavo de dólar por hora (para uma microinstância EC2 da Amazon), e a instância mais barata do Google custa 4,5 centavos de dólar por hora, com um mínimo de somente 10 minutos. Graças à economia de escala, comprar uma pequena instância de processamento de uma empresa grande é praticamente o mesmo que comprar a própria máquina física e dedicada – exceto que, agora, você não terá mais de contratar um profissional de TI para mantê-la em execução.

É claro que instruções passo a passo para configurar e executar instâncias de processamento na nuvem estão um pouco além do escopo deste livro, mas é provável que você descubra que essas instruções passo a passo não são necessárias. Com a Amazon e o Google (sem mencionar as inúmeras empresas menores no mercado) competindo pelos dólares da computação na nuvem, eles facilitaram a configuração de novas instâncias, bastando, para isso, seguir um prompt simples, pensar em um nome para a aplicação e fornecer um número de cartão de crédito. Atualmente (quando este livro foi escrito), tanto a Amazon quanto o Google também oferecem horas de processamento gratuito no valor de centenas de dólares na tentativa de atrair novos clientes.

Depois que tiver uma instância configurada, você será um novo proprietário orgulhoso de um endereço IP, um nome de usuário e chaves públicas/privadas, que poderão ser usados para se conectar com a sua instância usando SSH. A partir daí, tudo deve se passar como se você estivesse usando um servidor que fosse fisicamente seu – exceto, é claro, que não será mais necessário se preocupar com manutenção de hardware nem executar seu próprio conjunto enorme de ferramentas avançadas de monitoração.

Para trabalhos rápidos e simples, em especial se você não tem muita experiência para lidar com SSH e com pares de chaves, creio que as instâncias do Cloud Platform do Google podem ser mais fáceis de utilizar de imediato. Elas têm um launcher simples e, após a inicialização, disponibilizam até mesmo um botão para visualizar um terminal SSH no próprio navegador, como vemos na Figura 17.1.



```
ryan_e_mitchell@lamp-1-vm: ~
Secure https://ssh.cloud.google.com/projects/api-scraper/zones/us-central1-f/instances/lamp-1-vm?authuser=0&hl=en_US&proje...
Connected, host fingerprint: ssh-rsa 2048 B6:06:9C:6F:54:78:F2:50:0E:97:70:CC:7F:05:D9:0B:31:83:62:1B
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
ryan_e_mitchell@lamp-1-vm:~$ python
Python 2.7.9 (default, Jun 29 2016, 13:08:31)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
>>>
```

*Figura 17.1 – Terminal para navegador, em uma instância de máquina virtual do Cloud Platform do Google.*

## Recursos adicionais

Muitos anos atrás, executar “na nuvem” era essencialmente o domínio daqueles que estivessem dispostos a mergulhar fundo na documentação e já tivessem alguma experiência com administração de servidores. Hoje em dia, porém, as ferramentas melhoraram bastante em razão da maior popularidade e da concorrência entre provedores de computação na nuvem.

Apesar disso, para construir scrapers e crawlers de grande porte ou mais complexos, talvez você queira mais orientações sobre como criar uma

plataforma para coletar e armazenar dados.

O livro *Google Compute Engine* de Marc Cohen, Kathryn Hurley e Paul Newson (O'Reilly, <http://oreil.ly/1FVOW6y>) é um recurso simples que descreve como usar o Google Cloud Computing tanto com Python como com JavaScript. Ele não só aborda a interface de usuário do Google como também as ferramentas de linha de comando e de scripting que podem ser usadas para dar mais flexibilidade à sua aplicação.

Se preferir trabalhar com a Amazon, o livro *Python and AWS Cookbook* de Mitch Garnaat (O'Reilly, <http://oreil.ly/VSctQP>) é um guia conciso, porém muito útil, que lhe permitirá trabalhar com o Amazon Web Services e mostrará como ter uma aplicação escalável pronta e executando.

---

<sup>1</sup> Tecnicamente, os endereços IP podem ser forjados em pacotes de saída; essa é uma técnica usada em ataques distribuídos de negação de serviço (distributed denial-of-service attacks), em que os invasores não se preocupam em receber pacotes de volta (os quais, se houver, serão enviados para o endereço errado). Por definição, porém, o web scraping é uma atividade na qual uma resposta do servidor web é necessária, portanto consideramos que os endereços IP são uma informação que não pode ser falsificada.

## Aspectos legais e éticos do web scraping

Em 2010, o engenheiro de software Pete Warden construiu um web crawler para coletar dados do Facebook. Ele coletou dados de aproximadamente 200 milhões de usuários do Facebook – nomes, informações de localização, amigos e interesses. É claro que o Facebook percebeu e lhe enviou cartas de cessar e desistir (cease-and-desist letters), às quais ele obedeceu. Quando questionado sobre o motivo pelo qual havia concordado com elas, ele disse: “Big data? São baratos. Advogados? Nem tanto”.

Neste capítulo, veremos as leis norte-americanas (e algumas leis internacionais) que são relevantes para o web scraping, e aprenderemos a analisar os aspectos legais e éticos de uma dada situação de web scraping.

Antes de ler esta seção, considere o óbvio: sou uma engenheira de software, não uma advogada. Não interprete nada que ler neste capítulo ou em nenhum dos demais como se fossem um aconselhamento jurídico profissional nem trabalhe com essas informações dessa forma. Embora eu acredite que seja capaz de discutir os aspectos legais e éticos do web scraping com conhecimento de causa, você deve consultar um advogado (e não um engenheiro de software) antes de se responsabilizar por qualquer projeto de web scraping que seja ambíguo do ponto de vista legal.

O objetivo deste capítulo é apresentar um modelo para que você entenda e discuta diversos aspectos legais relacionados ao web scraping, por exemplo, propriedade intelectual, acesso não autorizado a computadores e uso de servidores; todavia, essas informações não devem substituir um verdadeiro aconselhamento jurídico.

### **Marcas registradas, direitos autorais, patentes, oh, céus!**

É hora de um curso básico de Propriedade Intelectual! Há três tipos básicos de propriedade intelectual (PI): marcas registradas (indicadas pelos símbolos <sup>™</sup> ou ®), direitos autorais (é o símbolo © presente em todos os

lugares) e as patentes (às vezes sinalizadas com um texto informando que a invenção é protegida por patente ou por um número de patente; muitas vezes, porém, não haverá nenhuma informação).

As patentes são usadas apenas para declarar a posse de invenções. Não é possível patentear imagens, texto ou qualquer informação por si só. Embora algumas patentes, como de software, sejam menos tangíveis do que aquilo que imaginamos como “invenções”, lembre-se de que é o *artefato* (ou a técnica) que é patenteado – e não a informação contida na patente. A menos que você construa algo a partir de projetos coletados na internet, ou alguém patenteie um método de web scraping, é improvável que você vá infringir uma patente inadvertidamente ao fazer scraping da web.

É pouco provável também que as marcas registradas sejam um problema, mas, mesmo assim, elas devem ser levadas em consideração. De acordo com o US Patent and Trademark Office (Departamento de Patentes e Marcas Registradas dos Estados Unidos):

Uma *marca registrada* (trademark) é uma palavra, frase, símbolo e/ou design que identifica e distingue a origem dos produtos como sendo de uma companhia e não de outra. Uma *marca de serviço* (service mark) é uma palavra, frase, símbolo e/ou design que identifica e distingue a origem de um serviço em vez de um produto. O termo “marca registrada” muitas vezes é usado para se referir tanto às marcas registradas como às marcas de serviço.

Além das tradicionais marcas com palavras/símbolos em que pensamos quando as marcas registradas nos vêm à mente, outros atributos descritivos podem ter marca registrada. Esses incluem, por exemplo, o formato de uma embalagem (pense nas garrafas de Coca-Cola) ou até mesmo uma cor (um caso que merece destaque é a cor rosa do isolamento de fibra de vidro da Pantera Cor-de-Rosa do Owens Corning).

De modo diferente das patentes, a propriedade de uma marca registrada depende muito do contexto no qual ela é usada. Por exemplo, se quisesse publicar uma postagem de blog com uma imagem do logo da Coca-Cola, eu poderia fazer isso (desde que não houvesse nenhuma implicação de que minha postagem de blog fosse patrocinada ou publicada pela Coca-Cola). Se eu quisesse fabricar um novo refrigerante exibindo o mesmo logo da Coca-Cola na garrafa, claramente essa seria uma infração da marca

registrada. De modo semelhante, embora eu pudesse engarrafar meu novo refrigerante com o cor-de-rosa da Pantera Cor-de-Rosa, eu não poderia usar a mesma cor para criar um produto para isolamento de residências.

## **Lei de direitos autorais**

Tanto as marcas registradas como as patentes têm um aspecto em comum: ambas devem ser formalmente registradas para serem reconhecidas. Contrário à crença popular, isso não vale para conteúdos com direitos autorais. O que faz com que imagens, texto, música etc. tenham direitos autorais? Não é o aviso de Todos os Direitos Reservados na parte inferior da página, nem nada especial quanto a conteúdo “publicado” *versus* “não publicado”. Todo conteúdo que você criar estará automaticamente sujeito à lei de direitos autorais assim que passar a existir.

A Convenção de Berna para Proteção de Obras Literárias e Artísticas, que recebeu o nome por causa da cidade de Berna na Suíça, onde foi adotada pela primeira vez em 1886, determina o padrão internacional para direitos autorais. A convenção, em sua essência, afirma que todos os países-membros devem reconhecer a proteção dos direitos autorais das obras dos cidadãos de outros países-membros como se fossem cidadãos de seu próprio país. Na prática, isso significa que, se você for um cidadão norte-americano, poderá ser responsabilizado nos Estados Unidos por violação de direitos autorais de um conteúdo escrito por alguém, por exemplo, na França (e vice-versa).

Obviamente, o direito autoral é uma preocupação para os web scrapers. Se eu fizer scraping do conteúdo do blog de alguém e publicá-lo em meu próprio blog, poderia muito bem estar sujeita a ser processada. Felizmente, há várias camadas de proteção que podem deixar meu projeto de scraping de blog defensável, dependendo de como funcionar.

Em primeiro lugar, a proteção de direitos autorais engloba apenas trabalhos criativos. Não inclui estatísticas ou fatos. Felizmente, boa parte do que os web scrapers procuram *são* estatísticas e fatos. Embora um web scraper que colete poesias da internet e as exiba em seu próprio site esteja violando a lei de direitos autorais, um web scraper que colete informações sobre a frequência de postagens sobre poesia ao longo do tempo não estará. A poesia, em seu formato básico, é um trabalho criativo. O número médio de palavras em poemas publicados em um site mensalmente é um dado factual, e não um trabalho criativo.

Um conteúdo postado literalmente (em oposição a um conteúdo agregado/calculado a partir de dados brutos coletados) talvez não viole a lei de direitos autorais se esses dados forem sobre preços, nomes de executivos de empresas ou alguma outra informação factual.

Até mesmo conteúdos com direitos autorais podem ser usados de forma direta, dentro de limites razoáveis, sob a DMCA (Digital Millennium Copyright Act, ou Lei dos Direitos Autorais do Milênio Digital). A DMCA define algumas regras para o tratamento automatizado de conteúdo com direitos autorais. É uma lei extensa, com muitas regras específicas para tudo, de ebooks a telefones. Contudo, dois pontos principais podem ser de particular relevância para o web scraping:

- Sob a proteção do mecanismo de “safe harbor” (porto seguro), se fizer scraping de um conteúdo de uma fonte que você foi levado a crer que continha apenas conteúdo não protegido por direitos autorais, porém um usuário havia submetido ali um conteúdo com direito autoral, você estará protegido, desde que remova o conteúdo com direito autoral quando for notificado.
- Você não pode se desviar das medidas de segurança (por exemplo, proteção com senha) para acessar um conteúdo.

Além do mais, a DMCA também reconhece que o conceito de *fair use* (uso justo) conforme a legislação norte-americana se aplica, e que notificações de remoção do conteúdo não podem ser geradas com base na proteção do “safe harbor” se o uso do conteúdo com direitos autorais se enquadrar no *fair use*.

Em suma, você não deve jamais publicar conteúdo protegido por direitos autorais sem permissão do autor original ou de quem detém esses direitos. Se estiver armazenando um conteúdo protegido, ao qual você tem livre acesso em seu próprio banco de dados não público para análise, não há problemas. Se publicar esse banco de dados em seu site para visualização ou download, será um problema. Se estiver analisando esse banco de dados e publicando estatísticas sobre contagens de palavras, uma lista dos autores baseada na produtividade ou uma outra meta-análise dos dados, tudo bem. Se inserir algumas citações selecionadas ou pequenas amostras dos dados para acompanhar sua meta-análise e a sua argumentação, provavelmente também não haverá problemas, mas talvez seja bom analisar a cláusula de *fair use* da legislação americana por garantia.

## Invasão de bens móveis

A *invasão de bens móveis* (trespass to chattels), em sua essência, é diferente do que imaginamos quando pensamos em “leis de invasão”, pois ela não se aplica a propriedades ou terrenos, mas a bens móveis (por exemplo, um servidor). Essa lei se aplica quando seu acesso a um bem sofre interferência de tal modo que impeça você de acessá-lo ou usá-lo.

Na atual era da computação em nuvem, é tentador não pensar nos servidores web como recursos reais e tangíveis. No entanto, os servidores não só são constituídos de componentes caros como também precisam ser guardados, monitorados, refrigerados e alimentados com muita energia elétrica. De acordo com algumas estimativas, os computadores são responsáveis por 10% do consumo global de eletricidade<sup>1</sup>. (Se uma inspeção em seus próprios eletrônicos não o convencer, considere as enormes fazendas de servidores (server farms) do Google, que têm de estar conectadas a grandes estações elétricas.)

Apesar de serem recursos caros, os servidores são interessantes do ponto de vista legal, pois os webmasters, em geral, *querem* que as pessoas consumam seus recursos (isto é, acessem seus sites) – só não querem que as pessoas consumam seus recursos *em excesso*. Acessar um site com o seu navegador não é um problema; lançar um ataque DDOS (Distributed Denial of Service, ou Negação de Serviço Distribuído) completo contra ele obviamente é.

Três critérios devem ser atendidos para um web scraper violar a lei de invasão de bens móveis:

### *Falta de consentimento*

Como os servidores web estão abertos a todos, em geral eles “dão consentimento” aos web scrapers também. No entanto, muitos Termos de Serviço dos sites proíbem especificamente o uso de scrapers. Além disso, qualquer notificação de cessar-e-desistir entregue a você evidentemente revogará esse consentimento.

### *Dano real*

Os servidores são caros. Além dos custos do servidor, se seus scrapers derrubarem um site ou limitarem sua capacidade de atender a outros usuários, isso pode ser incluído no “dano” causado por você.

## Intenção

Se você escreve o código, sabe o que ele faz!

Os três critérios devem ser atendidos para que a lei de invasão de bens móveis se aplique. No entanto, se estiver violando um Termo de Serviço, mas não estiver causando nenhum dano real, não pense que você estará imune a uma ação legal. É bem possível que esteja violando a lei de direitos autorais, a DMCA, a CFAA (Computer Fraud and Abuse Act, ou Lei de Fraude e Abuso de Computadores), sobre a qual darei mais explicações depois, ou uma das outras inúmeras leis que se apliquem aos web scrapers.

## Restringindo seus bots

Nos velhos tempos, os servidores web eram muito mais potentes que os computadores pessoais. Com efeito, parte da definição de *servidor* era *computador de grande porte*. Agora, houve uma espécie de virada no jogo. Meu computador pessoal, por exemplo, tem um processador de 3.5 GHz e 8 GB de RAM. Uma instância média da Amazon, para comparar (atualmente, quando este livro foi escrito), tem 4 GB de RAM e cerca de 3 GHz de capacidade de processamento.

Com uma conexão de internet razoável e uma máquina dedicada, até mesmo um único computador pessoal é capaz de impor uma carga pesada a vários sites, prejudicando-os ou tirando-os totalmente do ar. A menos que haja uma emergência médica e a única cura seja agregar todos os dados do site de Joe Schmo em apenas dois segundos, não há realmente motivos para sobrecarregar um site.

Um bot observado jamais termina. Às vezes, é melhor deixar os crawlers executando durante a noite, em vez de deixá-los no meio da tarde ou no final do dia, por alguns motivos:

- Se você tiver aproximadamente oito horas, até mesmo no passo lentíssimo de dois segundos por página, será possível rastrear mais de 14 mil páginas. Se o tempo não for um problema, você não se sentirá tentado a aumentar a velocidade de seus crawlers.
- Supondo que o público-alvo do site, de modo geral, esteja em sua localidade (faça ajustes caso o público-alvo seja remoto), a carga de tráfego do site provavelmente será muito menor durante a noite, e isso significa que seu rastreamento não contribuirá com um congestionamento no horário de pico de tráfego.
- Você economizará tempo se estiver dormindo, em vez de verificar constantemente seus logs para ver novas informações. Pense em como será empolgante acordar de manhã com dados totalmente novos!

Considere os seguintes cenários:

- Seu web crawler percorre o site de Joe Schmo coletando alguns ou todos os dados.
- Seu web crawler percorre centenas de sites pequenos coletando alguns ou todos os dados.
- Seu web crawler percorre um site bem grande, por exemplo, a Wikipédia.

No primeiro cenário, é melhor deixar o bot executando lentamente, durante a noite.

No segundo cenário, é melhor rastrear cada site usando o algoritmo de round-robin, em vez de rastreá-los lentamente, um de cada vez. Dependendo da quantidade de sites rastreados, isso significa que você pode coletar dados tão rápido quanto sua conexão com a internet e seu computador permitirem, e, mesmo assim, a carga será razoável para cada servidor remoto individual. Isso pode ser feito por meio de programação, seja usando várias threads (em que cada thread individual rastreia um único site e faz uma pausa em sua própria execução), ou usando listas Python para manter o controle dos sites.

No terceiro cenário, é improvável que a carga que a sua conexão com a internet e o computador pessoal impõem a um site como a Wikipédia seja percebida ou seja motivo de preocupação. No entanto, se estiver usando uma rede de computadores distribuídos, obviamente a questão será

diferente. Seja cauteloso e pergunte a um representante da empresa sempre que possível.

## **Lei de Fraude e Abuso de Computadores**

No início dos anos 1980, os computadores começaram a sair do ambiente acadêmico e migraram para o mundo dos negócios. Pela primeira vez, vírus e worms passaram a ser vistos não apenas como uma inconveniência (ou um passatempo divertido), mas como uma questão criminal séria, capaz de causar prejuízos financeiros. Em resposta, a CFAA (Computer Fraud and Abuse Act, ou Lei de Fraude e Abuso de Computadores) foi criada em 1986.

Embora você ache que a lei se aplique apenas a uma versão estereotipada de um hacker mal-intencionado disseminando vírus, ela tem sérias implicações para os web scrapers também. Pense em um scraper que faça uma varredura na internet à procura de formulários de login com senhas fáceis de descobrir, ou que colete segredos do governo, acidentalmente deixados em um local oculto, porém público. Todas essas atividades são ilegais (e com razão) sob a CFAA.

A lei define sete principais infrações criminais, que podem ser sintetizadas da seguinte maneira:

- acesso consciente e não autorizado a computadores do governo norte-americano e obtenção de informações desses computadores;
- acesso consciente e não autorizado a um computador, com obtenção de informações financeiras;
- acesso consciente e não autorizado a um computador do governo norte-americano, afetando o uso desse computador pelo governo;
- acesso consciente a qualquer computador protegido com tentativa de fraude;
- acesso consciente a um computador sem autorização, causando danos a ele;
- compartilhamento ou tráfico de senhas ou informações de autorização de computadores usados pelo governo norte-americano ou que afetem o comércio interestadual ou estrangeiro;
- tentativas de extorquir dinheiro ou “qualquer item de valor” por causar prejuízos, ou ameaçar causar prejuízos, a qualquer computador protegido.

Em suma, fique longe dos computadores protegidos, não acesse computadores (incluindo servidores web) aos quais você não tenha acesso concedido e, sobretudo, fique longe dos computadores do governo e dos computadores financeiros.

## **robots.txt e Termos de Serviço**

Os termos de serviço e os arquivos *robots.txt* de um site são um território interessante do ponto de vista legal. Se um site é acessível publicamente, o direito que o webmaster tem de declarar quais softwares podem e quais não podem acessá-lo é discutível. Dizer “tudo bem se você usar seu navegador para visualizar este site, mas não se usar um programa escrito por você” é complicado.

A maioria dos sites tem um link para seus Termos de Serviço (TOS) no rodapé de cada página. O TOS contém mais que apenas regras para os web crawlers e acessos automáticos; com frequência, ele contém informações sobre o tipo de dados que o site coleta, o que ele faz com eles e, em geral, uma declaração legal informando que os serviços oferecidos pelo site não têm nenhuma garantia explícita ou implícita.

Se estiver interessado em SEO (Search Engine Optimization, ou Otimização para Mecanismos de Busca) ou em tecnologia para ferramentas de pesquisa, é provável que já tenha ouvido falar do arquivo *robots.txt*. Se acessar praticamente qualquer site grande e procurar seu arquivo *robots.txt*, você o encontrará na pasta web raiz: <http://website.com/robots.txt>.

A sintaxe dos arquivos *robots.txt* foi desenvolvida em 1994, durante a explosão inicial da tecnologia para ferramenta de pesquisas web. Foi mais ou menos nessa época que as ferramentas de pesquisa que varriam toda a internet, como AltaVista e DogPile, começaram a competir seriamente com listas simples de sites organizadas por assunto, por exemplo, uma lista organizada pelo Yahoo!. Esse aumento da pesquisa na internet significou uma explosão não só no número de web crawlers, mas na disponibilidade das informações coletadas por esses crawlers ao cidadão comum.

Embora talvez consideremos esse tipo de disponibilidade natural hoje em dia, alguns webmasters ficaram chocados quando informações publicadas em níveis profundos da estrutura de arquivos de seus sites se tornaram disponíveis na primeira página dos resultados de pesquisa das principais ferramentas de busca. Em resposta, a sintaxe dos arquivos *robots.txt*,

chamada Robots Exclusion Standard (Padrão de Exclusão de Robôs), foi desenvolvida.

De modo diferente do TOS, que em geral se refere aos web crawlers em termos amplos e em linguagem bem humana, é possível fazer parse dos arquivos *robots.txt* e usá-los em programas automáticos com muita facilidade. Ainda que pareça o sistema perfeito para resolver o problema de bots indesejados de uma vez por todas, lembre-se de que:

- Não há nenhuma entidade oficial que determine a sintaxe de *robots.txt*. É uma convenção comum e, de modo geral, amplamente adotada, mas não há nada que impeça alguém de criar sua própria versão de um arquivo *robots.txt* (exceto pelo fato de que nenhum bot o reconhecerá nem obedecerá até que se torne popular). Apesar disso, é uma convenção bastante aceita, sobretudo por ser relativamente simples, e não há motivações para que as empresas inventem o seu próprio padrão ou tentem melhorá-lo.
- Não há qualquer maneira de garantir o uso de um arquivo *robots.txt*. É apenas um sinal para dizer: “Por favor, não acesse essas partes do site”. Há muitas bibliotecas de web scraping que obedecem ao arquivo *robots.txt* (embora essa seja apenas uma configuração default, possível de ser sobrescrita). Além disso, com frequência, há mais barreiras para seguir um arquivo *robots.txt* (afinal de contas, é necessário fazer scraping, parse e aplicar o conteúdo da página à lógica de seu código) do que para simplesmente ir em frente e fazer scraping de qualquer página que você quiser.

A sintaxe do Padrão de Exclusão de Robôs é bem simples. Como em Python (e em várias outras linguagens), os comentários começam com um símbolo #, terminam com uma quebra de linha e podem ser usados em qualquer lugar no arquivo.

A primeira linha do arquivo, sem considerar os comentários, é iniciada com `User-agent:`, que especifica o usuário ao qual as regras seguintes se aplicam. Na sequência, há um conjunto de regras, que podem ser `Allow:` ou `Disallow:`, conforme o bot tenha permissão ou não na respectiva seção do site. Um asterisco (\*) representa um caractere-curinga, e pode ser usado para descrever um `User-agent` ou um URL.

Se uma regra vier depois de outra que pareça contradizê-la, a última regra tem precedência. Por exemplo:

```
#Bem-vindo ao meu arquivo robots.txt!  
User-agent: *  
Disallow: *
```

```
User-agent: Googlebot  
Allow: *  
Disallow: /private
```

Nesse caso, todos os bots são proibidos em qualquer parte do site, exceto o Googlebot, que tem permissão para qualquer lugar, com exceção do diretório */private*.

O arquivo *robots.txt* do Twitter tem instruções explícitas para os bots do Google, Yahoo!, Yandex (uma ferramenta de pesquisa popular na Rússia), Microsoft e outros bots ou ferramentas de pesquisa não incluídos nas categorias anteriores. A seção do Google (que parece idêntica às permissões para todas as demais categorias de bots) tem o seguinte aspecto:

```
#Google Search Engine Robot  
User-agent: Googlebot  
Allow: /?_escaped_fragment_  
  
Allow: /?lang=  
Allow: /hashtag/*?src=  
Allow: /search?q=%23  
Disallow: /search/realtime  
Disallow: /search/users  
Disallow: /search/*/grid  
  
Disallow: /*?  
Disallow: /*/followers  
Disallow: /*/following
```

Observe que o Twitter restringe o acesso às partes de seu site para as quais há uma API. Como o Twitter tem uma API bem organizada (e uma com a qual ele pode ganhar dinheiro licenciando), é interesse da empresa não permitir que haja qualquer “API caseira” que colete informações rastreando o site de modo independente.

Embora um arquivo informando ao seu crawler quais são as partes que ele não pode acessar pareça restritivo à primeira vista, talvez seja uma bênção disfarçada para o desenvolvimento de um web crawler. Se você encontrar um arquivo *robots.txt* que não permita o rastreamento de uma parte específica do site, o webmaster está essencialmente dizendo que aceita crawlers em todas as demais seções (afinal de contas, se não concordasse com isso, não teria restringido o acesso quando escreveu o *robots.txt*, para

começar).

Por exemplo, a seção do arquivo *robots.txt* da Wikipédia que se aplica aos web scrapers em geral (em oposição às ferramentas de pesquisa) é extremamente permissiva. Ela chega a conter um texto legível aos seres humanos para dar boas-vindas aos bots (somos nós!) e bloqueia o acesso somente a algumas páginas, como a página de login, a página de pesquisa e a “Página aleatória” (random article):

```
#
# Friendly, low-speed bots are welcome viewing article pages, but not
# dynamically generated pages please.
#
# Inktomi's "Slurp" can read a minimum delay between hits; if your bot supports
# such a thing using the 'Crawl-delay' or another instruction, please let us
# know.
#
# There is a special exception for API mobileview to allow dynamic mobile web &
# app views to load section content.
# These views aren't HTTP-cached but use parser cache aggressively and don't
# expose special: pages etc.
#
User-agent: *
Allow: /w/api.php?action=mobileview&
Disallow: /w/
Disallow: /trap/
Disallow: /wiki/Especial:Search
Disallow: /wiki/Especial%3ASearch
Disallow: /wiki/Special:Collection
Disallow: /wiki/Spezial:Sammlung
Disallow: /wiki/Special:Random
Disallow: /wiki/Special%3ARandom
Disallow: /wiki/Special:Search
Disallow: /wiki/Special%3ASearch
Disallow: /wiki/Spesial:Search
Disallow: /wiki/Spesial%3ASearch
Disallow: /wiki/Spezial:Search
Disallow: /wiki/Spezial%3ASearch
Disallow: /wiki/Specjalna:Search
Disallow: /wiki/Specjalna%3ASearch
Disallow: /wiki/Speciaal:Search
Disallow: /wiki/Speciaal%3ASearch
Disallow: /wiki/Speciaal:Random
Disallow: /wiki/Speciaal%3ARandom
Disallow: /wiki/Speciel:Search
Disallow: /wiki/Speciel%3ASearch
Disallow: /wiki/Speciale:Search
Disallow: /wiki/Speciale%3ASearch
```

```
Disallow: /wiki/Istimewa:Search
Disallow: /wiki/Istimewa%3ASearch
Disallow: /wiki/Toiminnot:Search
Disallow: /wiki/Toiminnot%3ASearch
```

A decisão de escrever web crawlers que obedecem ao arquivo *robots.txt* cabe a você, mas recomendo enfaticamente que isso seja feito, sobretudo se seus crawlers rastreiam a web de modo indiscriminado.

## Três web scrapers

Pelo fato de o web scraping ser um campo tão ilimitado, há um número enorme de maneiras de se colocar em apuros legais. Esta seção apresenta três casos que tiveram envolvimento com algum tipo de lei em geral aplicada aos web scrapers, e como ela foi usada em cada caso.

### **eBay versus Bidder's Edge e transgressão a bens móveis**

Em 1997, o mercado do Beanie Baby estava em alta, o setor de tecnologia fervilhava e sites de leilão eram a grande novidade na internet. Uma empresa chamada Bidder's Edge concebeu e criou um novo tipo de site de metaleilões. Em vez de forçar você a ir de um site de leilão a outro comparando preços, ela agregaria os dados de todos os leilões no momento para um produto específico (por exemplo, um novíssimo boneco Furby ou uma cópia do filme *O mundo das Spice Girls*) e apontava o site que tivesse o menor preço.

O Bidder's Edge fazia isso com um batalhão de web scrapers, efetuando requisições constantes aos servidores web dos vários sites de leilão a fim de obter informações sobre preços e produtos. De todos os sites de leilão, o eBay era o maior, e o Bidder's Edge acessava os servidores do eBay aproximadamente 100 mil vezes por dia. Até mesmo para os padrões atuais, é muito tráfego. De acordo com o eBay, isso representava 1,53% de todo o seu tráfego de internet na época, e a empresa, sem dúvida, não estava satisfeita com isso.

O eBay enviou ao Bidder's Edge uma carta de cessar e desistir (cease-and-desist letter), junto com uma oferta para licenciar seus dados. No entanto, as negociações para essa licença falharam e o Bidder's Edge continuou a rastrear o site do eBay.

O eBay tentou bloquear os endereços IP usados pelo Bidder's Edge – eram 169 –, porém, o Bidder's Edge conseguiu contornar o problema usando

servidores proxy (servidores que encaminham requisições em nome de outra máquina, mas usam o próprio endereço IP do servidor proxy). Como certamente você deve imaginar, era uma solução frustrante e insustentável para as duas partes – o Bidder’s Edge ficava constantemente tentando encontrar novos servidores proxy e comprar novos endereços IP enquanto os antigos estavam bloqueados, e o eBay era forçado a manter listas longas no firewall (e acrescentar um overhead a fim de comparar endereços IP para cada pacote verificado, o que exigia bastante do processador).

Por fim, em dezembro de 1999, o eBay processou o Bidder’s Edge por invasão de bens móveis (trespass to chattels).

Como os servidores do eBay eram recursos reais e tangíveis, e a empresa não estava satisfeita com o uso anormal deles pelo Bidder’s Edge, a lei de invasão de bens móveis parecia a lei ideal a ser usada. Com efeito, na era moderna, a invasão de bens móveis anda de mãos dadas com os processos de web scraping, e muitas vezes é considerada uma lei de TI.

Os tribunais determinaram que, para que o eBay ganhasse a causa usando a lei de invasão de bens móveis, a empresa deveria comprovar dois fatos:

- o Bidder’s Edge não tinha permissão para usar os recursos do eBay;
- o eBay havia sofrido perdas financeiras como resultado das ações do Bidder’s Edge.

Considerando a evidência das cartas de cessar e desistir do eBay, junto com os registros de TI mostrando o uso dos servidores e os custos associados a eles, essa foi uma tarefa relativamente fácil para o eBay. É claro que nenhuma grande batalha judicial termina facilmente: novas acusações foram feitas, muitos advogados foram pagos e, em março de 2001, houve um acordo fora dos tribunais envolvendo uma quantia não revelada.

Isso significa que qualquer uso não autorizado do servidor de outra pessoa é automaticamente uma violação da lei de invasão de bens móveis? Não necessariamente. O Bidder’s Edge foi um caso extremo; eles estavam usando muitos recursos do eBay, fazendo com que a empresa tivesse de comprar servidores adicionais, pagar mais pela energia elétrica e talvez contratar mais funcionários (embora 1,53% não pareça muito, em empresas grandes, pode chegar a uma quantia significativa).

Em 2003, a Suprema Corte da Califórnia tomou uma decisão, em outro caso, Intel Corp *versus* Hamidi, no qual um ex-funcionário da Intel (Hamidi) havia enviado emails que a Intel não gostou, usando os

servidores da Intel, aos funcionários dessa empresa. O tribunal determinou o seguinte:

A reivindicação da Intel é injustificável, não porque e-mails transmitidos pela internet desfrutam imunidade exclusiva, mas porque o dano por invasão de bens móveis – diferente das causas de ação mencionadas – não pode, na Califórnia, ser comprovado sem evidências de um prejuízo à propriedade pessoal ou aos interesses legais da parte queixosa.

Essencialmente, a Intel havia falhado em provar que os custos de transmitir os seis emails enviados por Hamidi a todos os funcionários (o interessante é que cada funcionário tinha a opção de ser removido da lista de distribuição de Hamidi – pelo menos, ele foi bem-educado!) haviam contribuído para qualquer prejuízo financeiro à empresa. Isso não havia privado a Intel de qualquer propriedade ou do uso de sua propriedade.

### **Estados Unidos versus Auernheimer e a Lei de Fraude e Abuso de Computadores**

Se as informações estiverem prontamente acessíveis na internet a um ser humano usando um navegador web, é improvável que acessar exatamente o mesmo conteúdo de forma automatizada colocaria você em uma situação complicada junto aos agentes federais. No entanto, por mais fácil que seja a uma pessoa suficientemente curiosa encontrar uma pequena brecha de segurança, essa pequena brecha pode se transformar bem rápido em uma brecha muito maior e mais perigosa se scrapers automatizados entrarem em cena.

Em 2010, Andrew Auernheimer e Daniel Spitler perceberam uma funcionalidade interessante nos iPads: se você acessasse o site da AT&T com eles, a empresa o redirecionava para um URL contendo o número de ID único de seu iPad:

<https://dcp2.att.com/OEPClient/openPage?ICCID=<idNumber>&IMEI=>

Essa página continha um formulário de login, com o endereço de email do usuário cujo número de ID estivesse no URL. Isso permitia aos usuários ter acesso às suas contas simplesmente fornecendo a senha.

Embora houvesse uma quantidade enorme de números de ID de iPads em potencial, era possível, considerando que houvesse web scrapers suficientes, iterar pelos possíveis números e obter endereços de email durante esse processo. Ao oferecer aos usuários esse recurso conveniente de login, a AT&T basicamente tornou públicos os endereços de email de

seus clientes na internet.

Auernheimer e Spitler criaram um scraper que coletou 114 mil desses endereços de email, entre eles os endereços de email privados de celebridades, CEOs e funcionários do governo. Auernheimer (mas não Spitler) então enviou a lista, e informações sobre como a obtivera, à Gawker Media, que publicou a história (mas não a lista) com a manchete: “Apple’s Worst Security Breach: 114,000 iPad Owners Exposed” (Pior falha de segurança da Apple: 114 mil proprietários de iPad expostos).

Em junho de 2011, a residência de Auernheimer foi invadida pelo FBI em conexão com a coleta de endereços de email, apesar de ele ter sido detido por drogas. Em novembro de 2012, ele foi considerado culpado por falsificação de identidade e conspiração por acessar um computador sem autorização e, mais tarde, foi sentenciado a 41 meses em uma penitenciária federal e condenado a pagar 73 mil dólares de restituição.

Seu caso chamou a atenção do advogado de direitos civis Orin Kerr, que se juntou à sua equipe jurídica e fez uma apelação ao Terceiro Circuito de Cortes de Apelações (Third Circuit Court of Appeals). Em 11 de abril de 2014 (esses processos legais podem demorar bastante), o Terceiro Circuito concordou com a apelação, afirmando o seguinte:

A condenação de Auernheimer na Acusação 1 deve ser revogada porque acessar um site publicamente disponível não é um acesso não autorizado sob a Lei de Fraude e Abuso de Computadores, 18 U.S.C. § 1030(a)(2) (C). A AT&T optou por não empregar senhas nem outras medidas de proteção para controlar o acesso aos endereços de email de seus clientes. É irrelevante o fato de que a AT&T desejasse subjetivamente que pessoas externas não deparassem com os dados ou que Auernheimer tivesse hiperbolicamente caracterizado o acesso como um “roubo”. A empresa configurou seus servidores de modo a deixar as informações disponíveis para qualquer pessoa, e, desse modo, autorizou o público em geral a visualizá-las. Acessar os endereços de email por meio do site público da AT&T era permitido pela CFAA e, desse modo, não constituiu um crime.

Assim, a sanidade prevaleceu no sistema jurídico, Auernheimer foi libertado da prisão naquele mesmo dia e todos viveram felizes para sempre. Apesar de, em última análise, o tribunal ter decidido que Auernheimer não violou a Lei de Fraude e Abuso de Computadores, ele teve a casa invadida

pelo FBI, gastou milhares de dólares com despesas legais e passou três anos entrando e saindo de tribunais e prisões. Como web scrapers, quais lições podemos aprender com esse caso para evitar situações semelhantes?

Fazer scraping de qualquer espécie de informações sensíveis, sejam dados pessoais (nesse caso, endereços de email), sejam segredos comerciais ou governamentais, provavelmente não é algo que você deva fazer sem ter um advogado de prontidão. Mesmo que as informações estejam publicamente disponíveis, pense no seguinte: “Um usuário comum de computador seria capaz de acessar facilmente essas informações se quisesse vê-las?” ou “É uma informação que a empresa quer que seus usuários vejam?”.

Em várias ocasiões, já liguei para algumas empresas a fim de informar que havia vulnerabilidades de segurança em seus sites e aplicações web. A fala a seguir faz maravilhas: “Oi, sou profissional de segurança e descobri uma possível vulnerabilidade de segurança em seu site. Você poderia me encaminhar a alguém para que eu possa informá-lo de modo que o problema seja resolvido?”. Além da satisfação imediata por ter sua genialidade como hacker (white hat) reconhecida, talvez você ganhe assinaturas gratuitas, recompensas em dinheiro e outros produtos como consequência!

Além do mais, a divulgação das informações de Auernheimer para a Gawker Media (antes de notificar a AT&T) e o alarde feito em torno da exploração da vulnerabilidade também fizeram com que ele se tornasse um alvo atraente para os advogados da AT&T.

Se encontrar vulnerabilidades de segurança em um site, o melhor a fazer é alertar os proprietários desse site, e não a mídia. Você pode se sentir tentado a escrever uma postagem de blog e anunciar esse fato ao mundo, sobretudo se uma correção para o problema não for feita de imediato. No entanto, é preciso lembrar que essa é uma responsabilidade da empresa, e não sua. O melhor que você pode fazer é levar seus web scrapers (e, se aplicável, o seu negócio) para longe do site!

### **Field versus Google: direitos autorais e robots.txt**

Um advogado chamado Blake Field processou a Google argumentando que o recurso de caching de site violava a lei de direitos autorais por exibir uma cópia de seu livro depois que ele o havia removido de seu site. A lei de direitos autorais permite ao criador de um trabalho criativo original ter controle sobre a distribuição desse trabalho. Field argumentava que o

caching do Google (depois que ele havia removido o livro do site) o destituía de sua capacidade de controlar a distribuição.

### Cache de web do Google

Quando os web scrapers do Google (também conhecidos como *bots do Google*) rastreiam os sites, eles fazem uma cópia do site e o hospedam na internet. Qualquer um pode acessar esse cache usando este formato de URL:

```
http://webcache.googleusercontent.com/search?q=cache:http://pythonscraping.com/
```

Se um site que você estiver procurando, ou do qual está fazendo scraping, estiver indisponível, esse URL poderia ser verificado para saber se há uma cópia que possa ser usada!

Conhecer o recurso de caching do Google e não ter tomado nenhuma providência não ajudou no caso de Field. Afinal de contas, ele poderia ter evitado que os bots do Google fizessem caching de seu site apenas adicionando o arquivo *robots.txt*, com diretivas simples informando quais páginas poderiam e quais não poderiam ter dados coletados.

O importante é que o tribunal considerou que a determinação de Safe Harbor da DMCA permitia ao Google fazer cache e exibir sites como o de Field legalmente: “[um] provedor de serviço não deve ser responsabilizado por perdas monetárias... por infração de direitos autorais em virtude da armazenagem intermediária e temporária de conteúdo em um sistema ou rede controlado ou operado pelo provedor de serviço ou por ele”.

## Seguindo em frente

A internet está mudando constantemente. As tecnologias que nos trazem imagens, vídeos, texto e outros arquivos de dados estão sendo atualizadas e reinventadas o tempo todo. Para acompanhar esse ritmo, o conjunto de tecnologias usado para scraping de dados da internet também deve mudar.

Não sabemos o que pode acontecer. Versões futuras deste texto poderão omitir por completo o JavaScript, considerando-o uma tecnologia obsoleta e de uso raro e, em vez disso, poderia ter como foco o parsing de hologramas HTML8. No entanto, o que não mudará é o modo de pensar e a abordagem geral necessários para coletar dados de qualquer site com sucesso (ou o que quer que seja usado como “sites” no futuro).

Ao deparar com qualquer projeto de web scraping, sempre faça as seguintes perguntas:

- Qual é a pergunta que eu quero que seja respondida, ou qual é o problema que eu quero que seja resolvido?

- Quais dados me ajudarão a fazer isso e onde estão?
- Como o site exibe esses dados? Sou capaz de identificar exatamente qual parte do código do site contém essas informações?
- Como posso isolar os dados e obtê-los?
- Que tipo de processamento ou análise devem ser feitos para deixar os dados mais convenientes?
- Como posso tornar esse processo melhor, mais rápido e mais robusto?

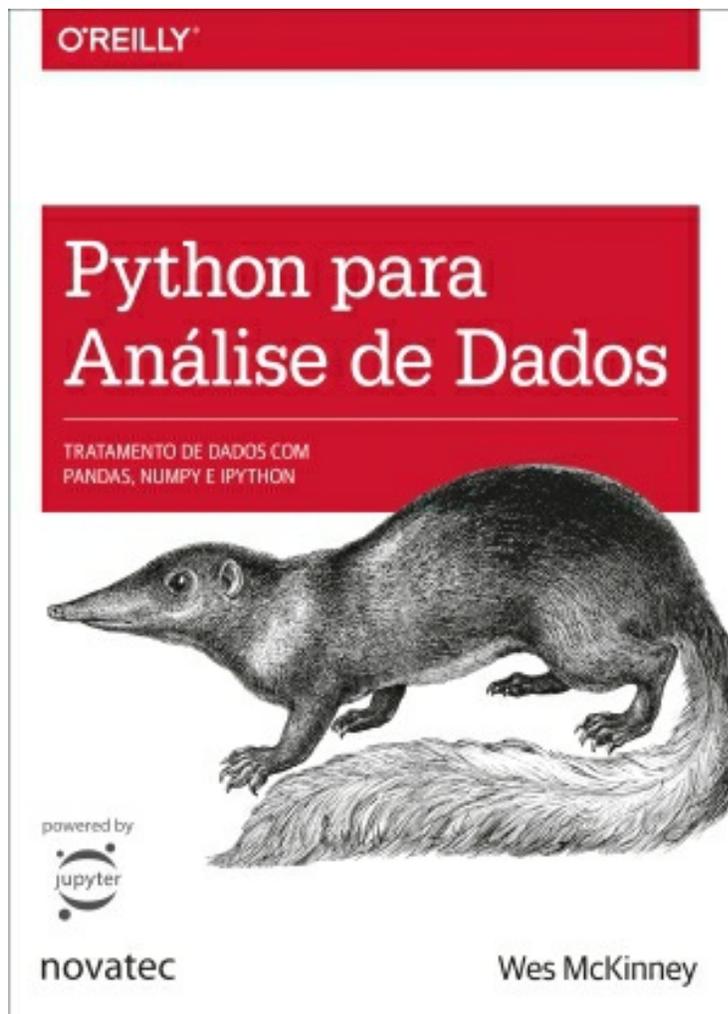
Além do mais, é necessário aprender não só a usar as ferramentas apresentadas neste livro de forma isolada, mas saber como podem atuar em conjunto para resolver um problema maior. Às vezes, os dados estarão facilmente disponíveis e bem formatados, permitindo que um scraper simples dê conta do serviço. Em outras ocasiões, será preciso pôr a cabeça para funcionar.

No *Capítulo 11*, por exemplo, combinamos a biblioteca Selenium para identificar imagens carregadas com Ajax na Amazon, e o Tesseract para usar OCR a fim de lê-las. No problema do Six Degrees of Wikipedia, usamos expressões regulares para escrever um crawler que armazenava informações de links em um banco de dados, e então utilizamos um algoritmo de resolução de grafos para responder à pergunta: “Qual é o caminho mais curto com links entre Kevin Bacon e Eric Idle?”.

Raramente haverá um problema sem solução quando se trata de coletar dados de modo automático na internet. Basta lembrar que a internet é uma API gigante, com uma interface de usuário um tanto quanto precária.

---

<sup>1</sup> Bryan Walsh, [“The Surprisingly Large Energy Footprint of the Digital Economy \[UPDATE\]”](#) (O surpreendentemente elevado consumo de energia na economia digital, <http://ti.me/21FOF3E>), TIME.com, 14 de agosto de 2013.



# Python para análise de dados

McKinney, Wes

9788575227510

616 páginas

[Compre agora e leia](#)

Obtenha instruções completas para manipular, processar, limpar e extrair informações de conjuntos de dados em Python. Atualizada

para Python 3.6, este guia prático está repleto de casos de estudo práticos que mostram como resolver um amplo conjunto de problemas de análise de dados de forma eficiente. Você conhecerá as versões mais recentes do pandas, da NumPy, do IPython e do Jupyter no processo. Escrito por Wes McKinney, criador do projeto Python pandas, este livro contém uma introdução prática e moderna às ferramentas de ciência de dados em Python. É ideal para analistas, para quem Python é uma novidade, e para programadores Python iniciantes nas áreas de ciência de dados e processamento científico. Os arquivos de dados e os materiais relacionados ao livro estão disponíveis no GitHub.

- utilize o shell IPython e o Jupyter Notebook para processamentos exploratórios;
- conheça os recursos básicos e avançados da NumPy (Numerical Python);
- comece a trabalhar com ferramentas de análise de dados da biblioteca pandas;
- utilize ferramentas flexíveis para carregar, limpar, transformar, combinar e reformatar dados;
- crie visualizações informativas com a matplotlib;
- aplique o recurso groupby do pandas para processar e sintetizar conjuntos de dados;
- analise e manipule dados de séries temporais regulares e irregulares;
- aprenda a resolver problemas de análise de dados do mundo real com exemplos completos e detalhados.

[Compre agora e leia](#)

O'REILLY

# Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações nativas de nuvem



novatec

Bilgin Ibryam  
Roland Huß

## Padrões para Kubernetes

Ibryam, Bilgin

9788575228159

272 páginas

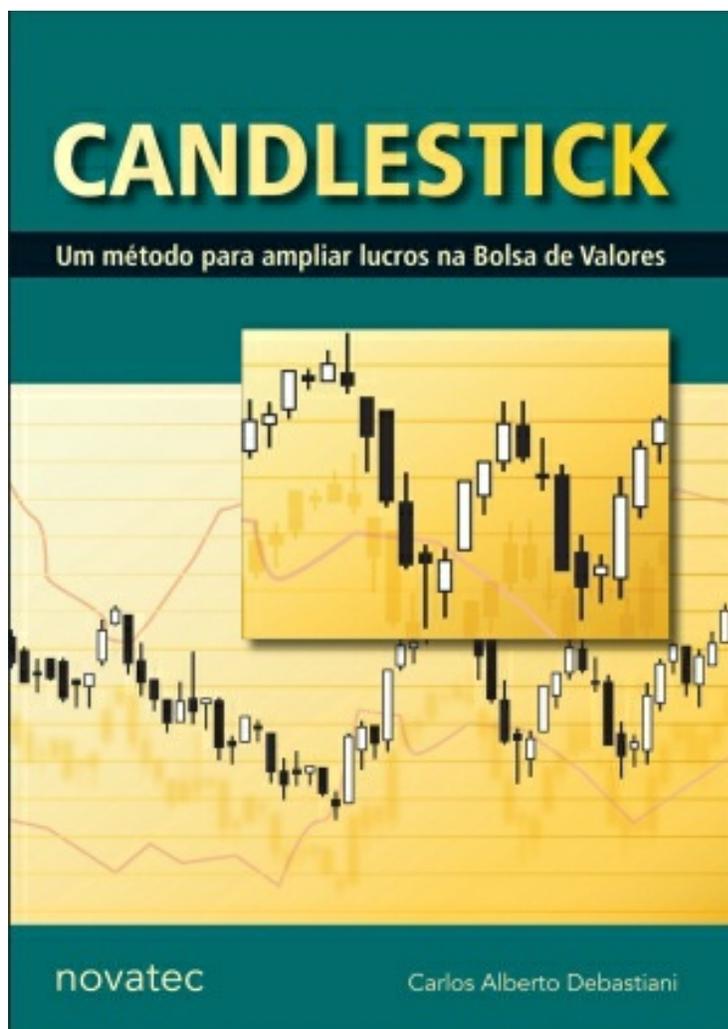
[Compre agora e leia](#)

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos

microserviços 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).

[Compre agora e leia](#)



# Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

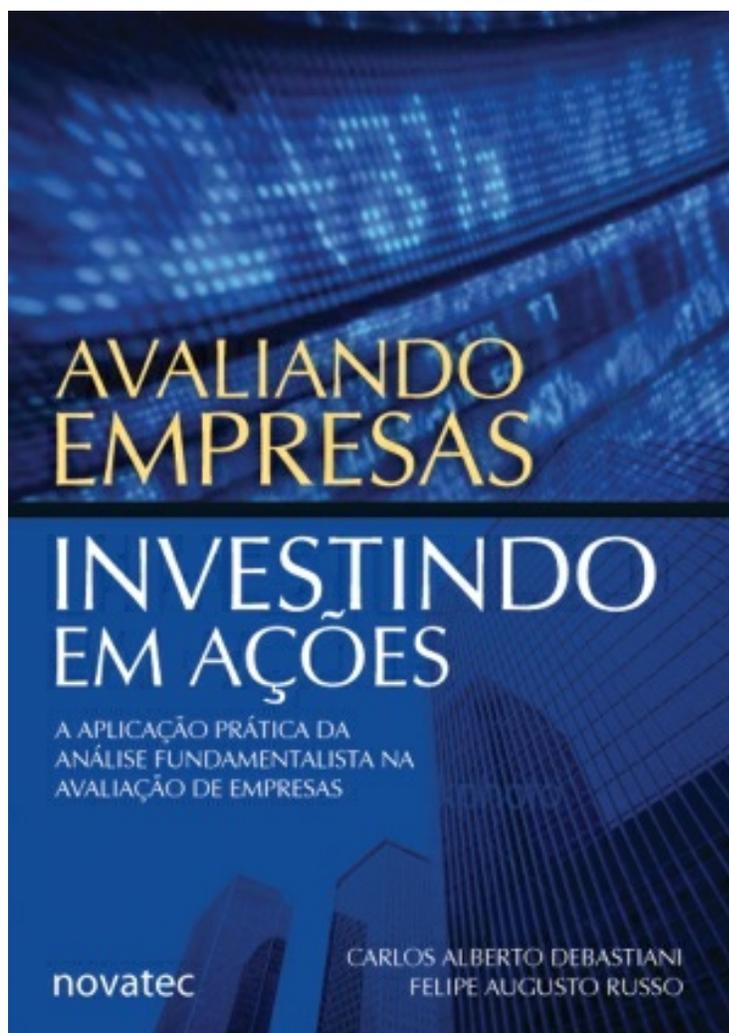
[Compre agora e leia](#)

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

Debastiani, Carlos Alberto

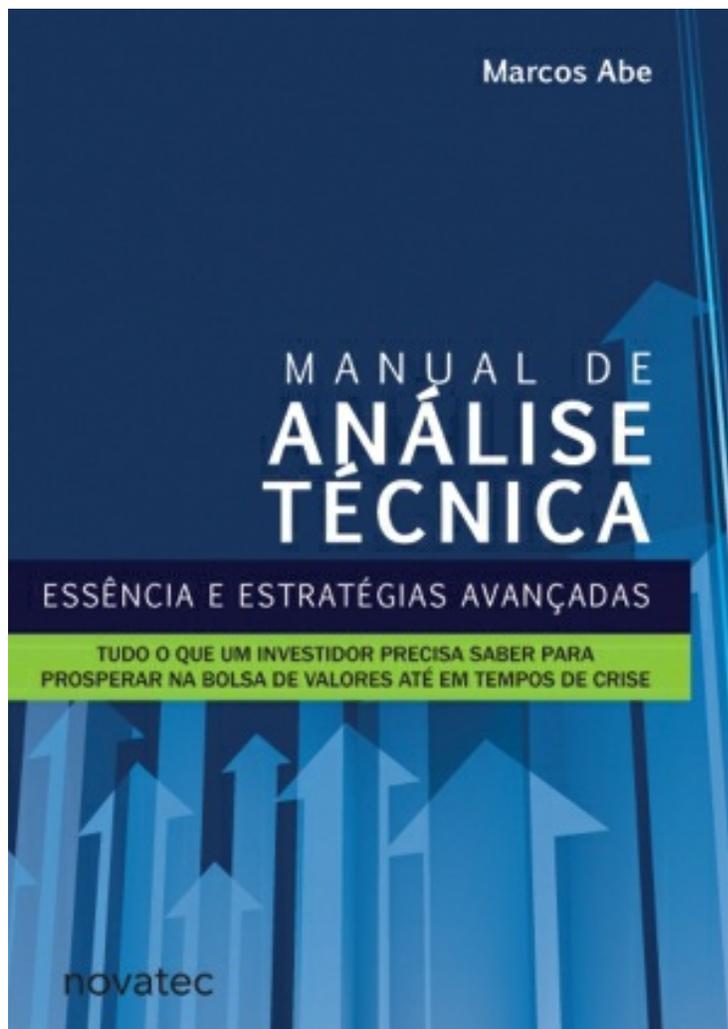
9788575225974

224 páginas

[Compre agora e leia](#)

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](#)



# Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

[Compre agora e leia](#)

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.

[Compre agora e leia](#)