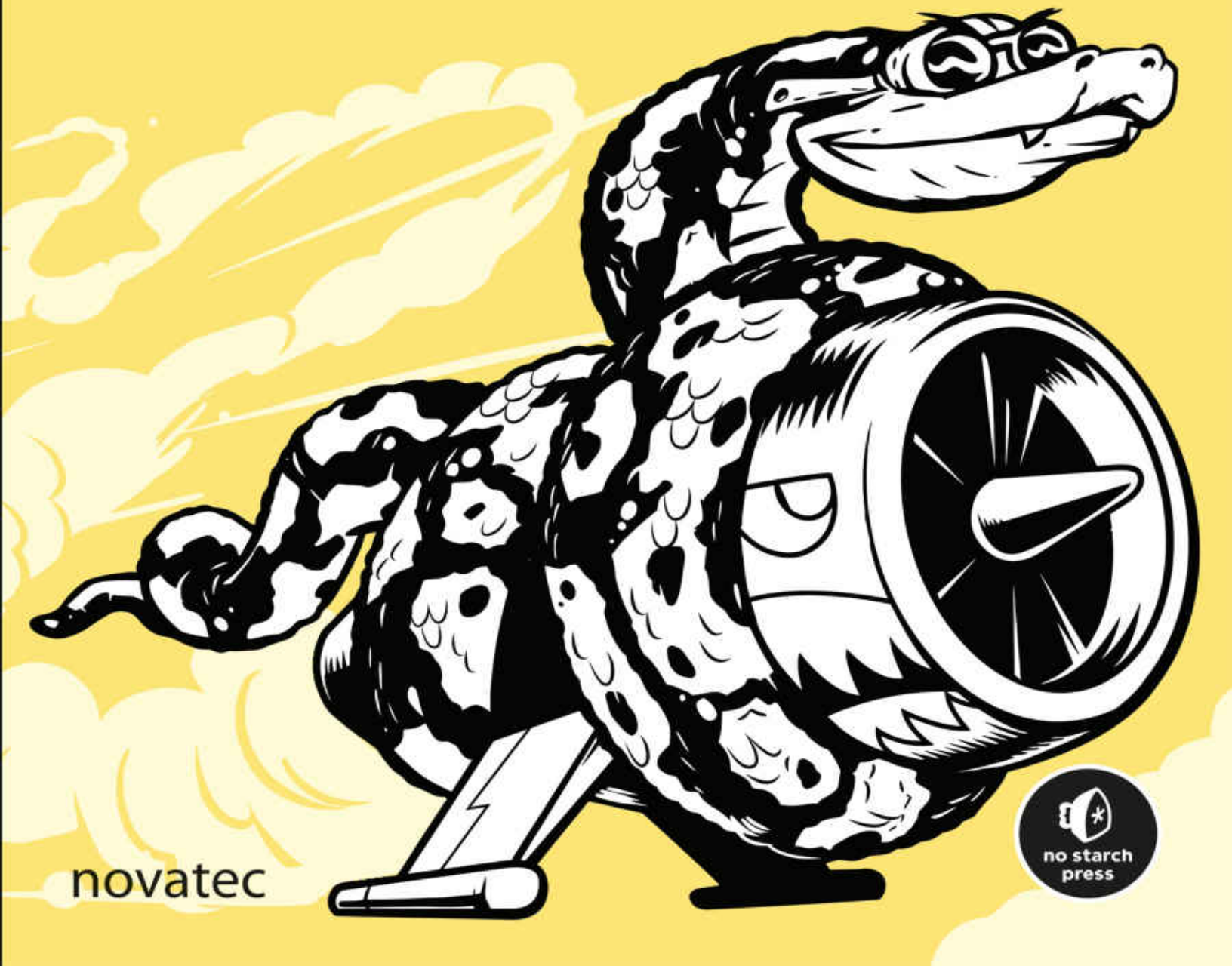
TERCEIRA EDIÇÃO

CURSO INTENSIVO DE PYTHON

UMA INTRODUÇÃO PRÁTICA E BASEADA EM PROJETOS À PROGRAMAÇÃO

ERIC MATTHES



novatec

no starch press

ELOGIOS AO

CURSO INTENSIVO DE PYTHON

"Foi interessante ver a No Starch Press produzindo futuros clássicos que deveriam andar de mãos dadas com os livros de programação mais tradicionais. *Curso Intensivo de Python* é um desses livros."

– Greg Laden, ScienceBlogs

"Aborda alguns projetos bastante complexos e os apresenta de forma consistente, lógica e amigável, atraindo o interesse do leitor."

– Full Circle Magazine

"Bem elaborado com trechos de códigos bem explicados. Você e o livro trabalham juntos: um pequeno passo de cada vez, desenvolvendo códigos mais complexos, com explicações sobre o que está acontecendo durante toda a jornada."

– FlickThrough Reviews

"Aprender Python com o *Curso Intensivo de Python* foi uma experiência extremamente positiva! Ótima escolha, caso seja iniciante em Python."

– Mikke Goes Coding

"Faz o que se propõe, e muito bem... Apresenta grande quantidade de exercícios práticos, bem como três projetos desafiadores e divertidos."

– RealPython.com

"Com uma introdução dinâmica, mas abrangente, à programação com Python, o *Curso Intensivo de Python* é outro livro excepcional para se ter em sua biblioteca e ajudá-lo a finalmente dominar o Python."

– TutorialEdge.net

"Opção genial aos marinheiros de primeira viagem em programação. Se está procurando uma introdução consolidada e simples para aprender minuciosamente Python, recomendo muito esse livro."

– WhatPixel.com

"Abrange tudo o que você precisa saber sobre Python e ainda mais, literalmente."

– FireBearStudio.com

"À medida que usa Python para ensiná-lo a programar, o *Curso Intensivo de Python* também ensina habilidades de programação limpas, usadas na maioria das outras linguagens."

– Great Lakes Geek

Curso Intensivo de Python

Uma introdução prática e baseada em projetos à programação

3ª Edição

Eric Matthes

Novatec

Editor: Rubens Prates
Tradução: Cibelle Ravaglia
Revisão gramatical: Alexandra Resende

ISBN do impresso: 978-85-7522-843-2

ISBN do ebook: 978-85-7522-844-9

Histórico de impressões:

Maio/2023 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
E-mail: novatec@novatec.com.br
Site: https://novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec
GRA20230404

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

Matthes, Eric
Curso intensivo de Python / Eric Matthes ; [tradução Cibelle Ravaglia]. -- 3. ed. -- São Paulo : Novatec Editora, 2023.

Título original: Python Crash Course
ISBN 978-85-7522-843-2

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

23-151031 CDD-005.133

Índices para catálogo sistemático:

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

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Ao meu pai, que sempre teve tempo para responder minhas perguntas sobre programação, e a Ever, que está começando a me fazer perguntas.

Sumário

Parte I Noções básicas

Capítulo 1 Primeiros passos

Capítulo 2 Variáveis e tipos de dados simples

Parte II Projetos

Capítulo 12 Uma espaçonave que dispara balas

Sobre o autor

Eric Matthes foi professor de matemática e ciências do Ensino Médio durante 25 anos e ministrava aulas introdutórias de Python sempre que encontrava uma forma de encaixá-las no conteúdo programático. Atualmente, é escritor e programador em tempo integral, e está engajado com diversos projetos open source. Seus projetos apresentam um leque multifacetado de objetivos, desde ajudar a predizer atividades de deslizamento de terra em regiões montanhosas até simplificar o processo de implantação de projetos Django. Quando não está escrevendo ou programando, gosta de escalar montanhas e passar tempo com sua família.

Sobre o revisor técnico

Kenneth Love mora no Noroeste Pacífico com sua família e com seus gatos. Kenneth é um programador Python de longa data, colaborador open source, professor e conferencista.

Prefácio da terceira edição

A resposta à primeira e segunda edições do *Curso Intensivo de Python* tem sido majoritariamente positivas. Mais de um milhão de exemplares foram impressos, incluindo traduções em mais de 10 idiomas. Recebi cartas e e-mails de leitores a partir dos 10 anos, bem como de aposentados que querem aprender a programar em seu tempo livre. O *Curso Intensivo de Python* está sendo usado em escolas de Ensino Fundamental e Médio, e também em aulas do ensino superior. Alunos que recebem livros didáticos mais avançados estão usando o *Curso Intensivo de Python* como texto complementar em suas aulas, encontrando nele valiosa complementação. As pessoas estão usando este livro para aprimorar suas habilidades no trabalho, mudar de carreira e começar a trabalhar nos próprios projetos paralelos. Em suma, as pessoas estão usando esta obra para todos os propósitos que eu esperava, e muito mais.

A oportunidade de escrever a terceira edição do *Curso Intensivo de Python* foi extremamente divertida. Embora seja uma linguagem madura, o Python continua a evoluir como todas as linguagens. Ao revisar este livro, meu principal objetivo foi mantê-lo como curso introdutório de Python minuciosamente organizado. Ao lê-lo, você aprenderá tudo o que é necessário para começar a trabalhar nos próprios projetos e também estabelecerá os alicerces para todo seu aprendizado futuro. Atualizei algumas seções a fim de retratar formas mais novas e mais simples de fazer as coisas em Python. Elucidei também algumas seções em que determinados detalhes da linguagem não foram apresentados com a precisão que poderiam ter sido. Todos os projetos foram completamente atualizados com bibliotecas populares e bem mantidas que você pode usar

tranquilamente para criar os próprios projetos.

Vejamos a seguir um resumo das mudanças específicas feitas na terceira edição:

- Agora, o Capítulo 1 apresenta o editor de texto VS Code, popular entre programadores iniciantes e profissionais e que funciona bem em todos os sistemas operacionais.
- O Capítulo 2 inclui os métodos removeprefix() e removesufix(), úteis ao trabalhar com arquivos e URLs. Este capítulo também apresenta as mensagens de erro recém-aprimoradas do Python, que fornecem informações bem mais específicas para ajudá-lo a solucionar problemas de seu código quando algo der errado.
- O Capítulo 10 usa o módulo pathlib para trabalhar com arquivos. Trata-se de uma abordagem mais simples para ler e gravar em arquivos.
- O Capítulo 11 usa o pytest a fim de escrever testes automatizados para o código que você programou. A biblioteca pytest se tornou a ferramenta padrão da mercado para escrever testes em Python. É amigável o bastante para usar em seus primeiros testes e, se você seguir uma carreira como programador Python, também a usará em configurações profissionais.
- Nos capítulos 12 a 14, o projeto do *Jogo Invasão Alienígena* inclui uma configuração para controlar a taxa de frames, o que faz com que o jogo seja executado de modo mais consistente em diferentes sistemas operacionais. Uma abordagem mais simples é utilizada para desenvolver a frota de alienígenas, e a organização geral do projeto também foi limpa.
- Nos capítulos 15 a 17, os projetos de Visualização de Dados usam as funcionalidades mais recentes do Matplotlib e do Plotly. As visualizações do Matplotlib apresentam configurações de estilo atualizadas. O projeto de passeio aleatório (o famoso random walk) apresenta uma pequena melhoria que aumenta a acurácia dos gráficos. Ou seja: você verá uma variedade maior de padrões surgir cada vez que gerar um novo passeio. Agora, todos os projetos com Plotly usam o módulo Plotly Express, possibilitando

gerar visualizações iniciais a partir de somente algumas linhas de código. É possível explorar com facilidade uma diversidade de visualizações antes de se comprometer com um tipo de gráfico e, depois, focar o refinamento dos elementos individuais desse gráfico.

- Nos capítulos 18 a 20, o projeto Registro de Aprendizagem é desenvolvido com a versão mais recente do Django e estilizado com a versão mais recente do Bootstrap. Algumas partes do projeto foram renomeadas visando facilitar o acompanhamento da organização geral do projeto. Agora, o projeto está implantado no Platform.sh, serviço moderno de hospedagem para projetos Django. O processo de implantação [de deploy] é controlado por arquivos de configuração YAML, que oferecem alto grau de controle sobre como é feito o deploy do seu projeto. Essa abordagem condiz com o modo dos programadores profissionais implantarem projetos modernos de Django.
- O Apêndice A foi inteiramente atualizado com as recomendações das melhores práticas atuais para instalar o Python em todos os principais sistemas operacionais. O Apêndice B compreende instruções detalhadas para configurar o VS Code e breves descrições da maioria dos principais editores de texto e IDEs atualmente em uso. O Apêndice C direciona os leitores aos diversos dos recursos online mais populares para obter ajuda. O Apêndice D prossegue fornecendo um minicurso intensivo sobre o uso do Git para controle de versão. O Apêndice E é novinho em folha, foi escrito para a terceira edição. Mesmo com um bom conjunto de instruções para implantar as aplicações que você cria, muitas coisas podem dar errado. Este apêndice oferece um guia aprofundado de resolução de problemas que você pode usar quando o processo de implantação não funcionar na primeira tentativa.
- O índice foi completamente atualizado, possibilitando que você use o *Curso Intensivo de Python* como referência para todos os seus futuros projetos Python.

Obrigado por ler o *Curso Intensivo de Python*! Se tiver algum

comentário ou dúvida, sinta-se à vontade para entrar em contato; sou *@ehmatthes* no Twitter.

Agradecimentos

Este livro não teria sido possível sem a equipe maravilhosa e imensamente profissional da No Starch Press. Bill Pollock me convidou para escrever um livro introdutório, sou profundamente grato a essa proposta inicial. Liz Chadwick trabalhou em todas as três edições, e o livro é melhor devido à sua participação contínua. Esta nova edição contou com as perspectivas renovadas de Eva Morrow, e suas ideias contribuíram com a melhoria do livro. Agradeço a orientação de Doug McNair em relação à gramática adequada, para que a obra não ficasse demasiadamente formal. Jennifer Kepler supervisionou o trabalho de produção, transformando meus muitos arquivos em um produto final refinado.

Não tive a oportunidade de trabalhar diretamente com muitas pessoas da No Starch Press que ajudaram a fazer deste livro um sucesso. A No Starch tem uma equipe de marketing formidável, que vai além da venda de livros; essas pessoas garantem que os leitores encontrem os livros que podem lhes ajudar a alcançar seus objetivos. A No Starch também tem um departamento sólido de direitos estrangeiros. O *Curso Intensivo de Python* alcançou leitores de todo o mundo, falantes de diversos idiomas, devido ao empenho dessa equipe. A todas essas pessoas com quem não trabalhei individualmente, obrigado por o ajudarem o *Curso Intensivo de Python* encontrar seu público-alvo.

Gostaria de agradecer a Kenneth Love, revisor técnico de todas as três edições do *Curso Intensivo de Python*. Conheci Kenneth na conferência anual PyCon e, desde então, seu entusiasmo pela linguagem e pela comunidade Python tem sido fonte constante de inspiração profissional. Kenneth, como de costume, foi além da mera

verificação de fatos e revisou o livro com o intuito de ajudar os programadores mais novos a desenvolver entendimento sólido da linguagem Python e da programação como um todo. Todos sempre estiveram atentos às seções que tiveram um bom resultado nas edições anteriores, mas que poderiam ser melhoradas, dada a oportunidade de reescrita completa. Sendo assim, sou completamente responsável por quaisquer inexatidões remanescentes.

Gostaria também de expressar minha gratidão a todos os leitores que compartilharam suas experiências bem-sucedidas com o *Curso Intensivo de Python*. Aprender os princípios básicos da programação pode mudar sua perspectiva sobre o mundo e, não raro, isso impacta profundamente as pessoas. Ao ouvir essas histórias, me senti extremamente lisonjeado; agradeço a todos que compartilharam suas experiências de maneira tão aberta.

Gostaria de agradecer ao meu pai por ter me apresentado à programação ainda jovem e por não ter medo de que eu quebrasse seu computador. Gostaria de agradecer minha esposa, Erin, por me apoiar e me incentivar enquanto eu escrevia esta obra, e por todo o trabalho exigido para que se materialize em várias edições. Gostaria também de agradecer ao meu filho Ever, cuja curiosidade sempre me serve de fonte de inspiração.

Introdução

Todo programador tem uma história a respeito de como aprendeu a escrever o primeiro programa. Comecei a programar ainda na infância, quando meu pai trabalhava para a Digital Equipment Corporation, uma das empresas pioneiras da era da computação moderna. Escrevi meu primeiro programa em um computador que meu pai montou em nosso porão. O computador não era mais nada do que uma reles placa-mãe conectada a um teclado sem gabinete, e seu monitor era um tubo simples de raios catódicos. Meu programa inicial foi um mero jogo de adivinhação de números, mais ou menos assim:

```
Estou pensando em um número! Tente adivinhar o número em que estou pensando: 25
Muito baixo! Tente de novo: 50
Muito alto! ente de novo: 42
É isso! Gostaria de jogar novamente? (sim/não) não
Obrigado por jogar!
```

Nunca me esquecerei de como fiquei contente ao ver minha família jogar um jogo que desenvolvi e que funcionou conforme o previsto.

Essa primeira experiência inicial me impactou de forma intemporal. É uma verdadeira satisfação desenvolver algo com um propósito, que resolva um problema. O software que escrevo atualmente atende às necessidades mais importantes do que meus empenhos infantis, porém, a sensação de satisfação que tenho ao desenvolver um programa que funciona ainda é basicamente a mesma.

A quem este livro se destina?

O intuito deste livro é prepará-lo o mais rápido possível para programar em Python, de modo que você consiga criar programas

que funcionem – jogos, visualizações de dados e aplicações web – ao mesmo tempo em que assimila o conhecimento básico em programação que lhe servirá pelo resto da vida. O *Curso Intensivo de Python* foi redigido para pessoas de qualquer idade que nunca programaram em Python ou nunca programaram. Este livro é para aqueles que querem aprender com rapidez os conceitos básicos de programação, para que, assim, possam focar os projetos interessantes, e para aqueles que gostam de testar o raciocínio aprendendo conceitos e resolvendo problemas relevantes. O *Curso Intensivo de Python* também é ideal para professores de todos os níveis de ensino que queiram apresentar a seus alunos uma introdução à programação baseada em projetos. Caso seja universitário e queira uma introdução mais amigável ao Python do que o conteúdo que lhe foi atribuído, este livro pode facilitar sua aprendizagem no ensino superior. Se quer mudar de carreira, o *Curso Intensivo de Python* pode ajudá-lo a fazer a transição para uma carreira mais gratificante. Os resultados têm sido positivos para um leque amplo e variado de leitores, com uma ampla gama de objetivos.

O que você pode esperar do livro?

O intuito deste livro é fazer de você um bom programador em geral, sobretudo, um bom programador Python. À medida que adquire uma base sólida em conceitos gerais de programação, você efetivamente aprenderá e adotará bons hábitos. Após evoluir em sua aprendizagem com o *Curso Intensivo de Python*, você estará preparado para aprender técnicas mais avançadas de Python. Ou seja, será ainda mais fácil de aprender sua próxima linguagem de programação.

Na Parte I deste livro você aprenderá os conceitos básicos de programação necessários para escrever programas em Python. Você basicamente aprenderia os mesmos conceitos com qualquer outra

linguagem de programação. Aprenderá mais sobre diferentes tipos de dados e como pode armazená-los em seus programas. Além disso, você criará coleções de dados, como listas e dicionários, e aprenderá como usar essas coleções de modo eficiente. Aprenderá a usar loops `while` e instruções `if` com o objetivo de testar determinadas condições, de modo que consiga executar seções específicas de código, verificando se suas condições são `True` ou `False` – técnica que ajuda a automatizar muitos processos.

Você vai aprender a aceitar entradas de usuários, de modo que seus programas fiquem interativos e sejam executados pelo tempo que o usuário desejar. Você descobrirá como escrever funções para que consiga reutilizar partes de seus programas: basta escrever blocos de código que executam determinadas ações uma vez, de modo que possa usá-lo quantas vezes precisar. Em seguida, você ampliará esse conceito a um comportamento mais complexo com classes, desenvolvendo programas relativamente simples que respondam a uma variedade de situações. Você aprenderá a escrever programas que lidam elegantemente com erros comuns. Após dominar cada um desses conceitos básicos, você escreverá uma série de programas cada vez mais complexos usando o que aprendeu. Por último, dará o primeiro passo rumo à programação intermediária, aprendendo como escrever testes para seu código, a fim de conseguir desenvolver ainda melhor seus programas sem se preocupar com o aparecimento de bugs. Com todas as informações da Parte I, você estará preparado para assumir projetos maiores e mais complexos.

Na Parte II, você usará o que aprendeu na Parte I em três projetos. Você pode escolher qualquer um ou todos os projetos, na ordem que bem entender. No primeiro projeto, nos capítulos 12 a 14, você desenvolverá um jogo de tiro no estilo *Space Invaders* chamado *Invasão Alienígena*, com diversos níveis progressivos de dificuldade. Depois de finalizar este projeto, você deve estar mais do que apto para desenvolver os próprios jogos 2D. Ainda que não pretenda se tornar um programador de jogos, desenvolver este projeto é uma

forma divertida de conciliar muito do que você aprendeu na Parte I.

O segundo projeto, nos capítulos 15 a 17, apresenta a visualização de dados. Os cientistas de dados usam uma variedade de técnicas de visualização para ajudá-los a entender a quantidade colossal de informações que lhes são disponibilizadas. Você trabalhará com conjuntos de dados gerados por meio de código, com conjuntos de dados que você fará download de fontes online e conjuntos de dados cujo download será feito automaticamente por seus programas. Após finalizar este projeto, você será capaz de escrever programas que analisam grandes conjuntos de dados e geram representações visuais de muitos tipos diferentes de informações.

No terceiro projeto, nos capítulos 18 a 20, você desenvolverá uma pequena aplicação web chamada Registro de Aprendizagem. Esse projeto possibilita que mantenha um registro organizado de informações que você aprendeu sobre um tópico específico. É possível manter registros separados para diferentes tópicos, possibilitando que outras pessoas criem uma conta e comecem a registrar suas jornadas de aprendizagem. Você também aprenderá a implantar seu projeto para que qualquer pessoa em qualquer lugar do mundo consiga acessá-lo.

Recursos online

A No Starch Press tem mais informações disponíveis online sobre este livro em https://nostarch.com/python-crash-course-3rd-edition.

Disponibilizo também um conjunto abrangente de recursos complementares em https://ehmatthes.github.io/pcc_3e. Esses recursos incluem:

- I**nstruções de configuração** – Apesar de serem idênticas às do livro, as instruções online de configuração incluem links ativos de todas as etapas diferentes em que você pode clicar. Se tiver problemas com a configuração, consulte este recurso.
- **Atualizações** – O Python, como todas as linguagens, está em

constante evolução. Eu mantenho um conjunto detalhado de atualizações. Então, se algo não estiver funcionando, confira aqui se as instruções foram alteradas.

- **Soluções dos exercícios** – Você deve passar um tempão tentando resolver os exercícios nas seções "Faça você mesmo". No entanto, se ficar empacado e não conseguir fazer nenhum progresso, confira as soluções online para a maioria dos exercícios.
- **Folhas de dicas** – Disponibilizo online um conjunto completo de folhas de dicas para download e breve referência dos principais conceitos.

Por que Python?

Todos os anos eu pondero se devo continuar usando Python ou se devo me dedicar a uma linguagem diferente, talvez uma linguagem mais recente no mundo da programação. No entanto, meu foco continua sendo o Python por muitas razões. Python é uma linguagem tremendamente eficiente: seus programas farão mais com poucas linhas de código do que o exigido por muitas outras linguagens. A sintaxe do Python também o ajudará a escrever um código "limpo", mais fácil de ler, depurar e de estender e receber incrementos, em comparação com outras linguagens.

As pessoas utilizam o Python para diversas finalidades: criar jogos, aplicações web, resolver problemas de negócios e desenvolver ferramentas internas em todos os tipos de empresas diferentes. O Python também é bastante utilizado em áreas de conhecimento científico, em pesquisas acadêmicas e trabalhos aplicados.

Uma das razões mais importantes pelas quais continuo usando Python é devido à comunidade Python, que envolve um grupo de pessoas extremamente diversificado e receptivo. A comunidade é indispensável aos programadores, já que a programação não é um esforço solitário. A maioria de nós, até os programadores mais experientes, precisa pedir conselhos a outras pessoas que já

resolveram problemas parecidos. Uma comunidade bem relacionada e prestativa é determinante para ajudá-lo a resolver problemas, e a comunidade Python presta total apoio às pessoas que estão aprendendo Python como primeira linguagem de programação ou que migram para o Python com experiência em outras linguagens.

Python é uma ótima linguagem para se aprender, então vamos começar!

PARTE I

Noções básicas

A **Parte I** deste livro ensina os conceitos básicos que você precisará para escrever programas Python. Vários desses conceitos são comuns em muitas linguagens de programação. Ou seja, serão úteis durante toda a sua vida como programador.

No **Capítulo 1** instalaremos o Python no computador e executaremos nosso primeiro programa, que exibe a mensagem *Hello world!* na tela.

No **Capítulo 2** vamos aprender a atribuir informações às variáveis e a usar valores numéricos e de texto.

Os **capítulos 3** e **4** apresentam as listas. As listas podem armazenar todas as informações que queremos em um único lugar, possibilitando que você trabalhe com esses dados de modo eficiente. É possível usar centenas, milhares e até milhões de valores em apenas algumas linhas de código.

No **Capítulo 5** usaremos as instruções `if` para escrever código que responde distintivamente em duas situações: se as condições forem verdadeiras e se não forem verdadeiras.

O **Capítulo 6** mostra como usar os dicionários do Python, que possibilitam estabelecer relações entre diferentes informações. Assim como as listas, os dicionários podem conter o máximo de informações que você precisa armazenar.

No **Capítulo 7** aprenderemos como aceitar entradas de usuários para que nossos programas fiquem interativos. Você também aprenderá sobre loops `while` que executam blocos de código

repetidamente, desde que determinadas condições permaneçam verdadeiras.

No **Capítulo 8** escreveremos funções, blocos nomeados de código que realizam uma tarefa específica e podem ser executados sempre que você precisar.

O **Capítulo 9** apresenta as classes que possibilitam modelar objetos do mundo real. Escreveremos um código que representa cães, gatos, pessoas, carros, espaçonaves e muito mais.

O **Capítulo 10** demonstra como trabalhar com arquivos e lidar com erros para que seus programas não travem de forma inesperada. Vamos armazenar dados antes de o programa encerrar e vamos reler os dados quando o programa for executado novamente. Aprenderemos sobre as exceções do Python, que possibilitam prever erros, fazendo com que seus programas lidem elegantemente com esses erros.

No **Capítulo 11** aprenderemos a escrever testes para nosso código, a fim de verificarmos se os programas funcionam conforme esperado. Como resultado, você será capaz de expandir seus programas sem se preocupar com a introdução de novos bugs. Testar o código é uma das primeiras habilidades que o ajuda na transição de programador iniciante para intermediário.

CAPÍTULO 1

Primeiros passos

Neste capítulo, você executará seu primeiro programa Python, *hello_world.py*. Primeiro, é necessário verificar se uma versão recente do Python está instalada em seu computador; caso contrário, você vai instalá-lo. Instalaremos também um editor de texto para trabalhar com os programas Python. Editores de texto reconhecem código Python e realçam as seções à medida que você escreve, facilitando a compreensão da estrutura do seu código.

Configurando seu ambiente de programação

O Python difere um pouco em sistemas operacionais diferentes, assim precisamos levar algumas coisas em consideração. Nas seções a seguir, garantiremos que o Python esteja configurado adequadamente em seu sistema.

Versões do Python

Toda linguagem de programação evolui à medida que ideias e novas tecnologias surgem, e os desenvolvedores Python têm reiteradamente contribuído para que a linguagem fique mais versátil e poderosa. No momento em que eu escrevia este livro, a versão mais recente era o Python 3.11. No entanto, tudo aqui deve ser executado no Python 3.9 ou posterior. Nesta seção, descobriremos se o Python já está instalado em seu sistema e se é necessário instalar uma versão mais recente. O Apêndice A também contém informações complementares sobre como instalar a versão mais

recente do Python em cada sistema operacional principal.

Executando trechos de código Python

É possível executar o interpretador Python em uma janela de terminal. Desse jeito, podemos testar partes de código Python sem precisar salvar e executar um programa inteiro.

No decorrer deste livro, você verá trechos de código parecidos com este:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
```

O prompt com três parênteses angulares (>>>), que chamaremos de *prompt do Python*, sinaliza que você deve usar a janela do terminal. O texto em negrito é o código que devemos digitar e executar pressionando a tecla ENTER. Aqui, a maioria dos exemplos são programas pequenos e independentes, que executaremos no editor de texto em vez de no terminal, pois escreveremos a maior parte do código no editor de texto. Mas, vez ou outra, conceitos básicos serão apresentados em uma série de trechos, executados em uma sessão de terminal Python. Assim, é possível demonstrar conceitos específicos com mais eficiência. Quando a linha de um código tiver três parênteses angulares, você está vendo o código e a saída de uma sessão de terminal. Testaremos o programa no interpretador de seu sistema logo mais.

Usaremos também um editor de texto para criar um programa simples chamado *Hello World!* que se tornou o verdadeiro clássico para aprender a programar. No mundo da programação há uma tradição de longa data: exibir a mensagem Hello world! na tela, para que seu programa em uma nova linguagem lhe traga boa sorte. Um programa tão simples tem um propósito sofisticado. Se esse programa for devidamente executado em seu sistema, qualquer programa Python que você escrever também funcionará.

Sobre o editor VS Code

O *VS Code* é um editor de texto poderoso, de qualidade profissional, gratuito e amigável para iniciantes. O VS Code é ótimo para projetos simples e complexos, portanto, caso se sinta à vontade em usá-lo enquanto aprende Python, poderá usá-lo também à medida que avança para projetos maiores e mais complicados. O VS Code pode ser instalado em todos os sistemas operacionais modernos, sendo compatível com a maioria das linguagens de programação, inclusive Python.

O Apêndice B fornece informações sobre outros editores de texto. Se estiver curioso sobre as outras opções, talvez queira dar uma olhada nesse apêndice. Se quer começar a programar logo, você pode usar o VS Code para iniciar. Posteriormente, você pode considerar outros editores, assim que adquirir um pouco mais de experiência como programador. Neste capítulo, vou orientá-lo na instalação do VS Code em seu sistema operacional.

NOTA *Caso já tenha um editor de texto instalado e saiba como configurá-lo para executar programas Python, você pode usá-lo.*

Python em diferentes sistemas operacionais

O Python é uma linguagem de programação multiplataforma. Ou seja, é executado em todos os principais sistemas operacionais. Qualquer programa Python que você escrever pode ser executado em qualquer computador moderno que tenha o instalado. No entanto, os métodos para configurar o Python em diferentes sistemas operacionais variam um pouco.

Nesta seção, aprenderemos como configurar o Python em seu sistema. Primeiro, confira se uma versão recente do Python está instalada em seu sistema. Caso contrário, instale. Depois,

instalaremos o VS Code. Essas são as duas únicas etapas diferentes para cada sistema operacional.

Nas seções a seguir, você executará *hello_world.py*, e vamos solucionar qualquer coisa que tenha dado errado. Vou orientá-lo no processo de cada sistema operacional, para que tenha um ambiente de programação Python no qual possa confiar.

Python no Windows

Em geral, o Windows não vem com o Python, portanto é bem provável que precisemos instalá-lo e depois instalar o VS Code.

Instalando o Python

Primeiro, verifique se o Python está instalado em seu sistema. Para abrir a janela de comando, digite **command** no menu Iniciar, depois clique no aplicativo **Prompt de Comando**. Na janela do terminal, digite **python** em letras minúsculas. Se receber como resposta um prompt do Python (>>>), o Python já está instalado em seu sistema. Se você vir uma mensagem de erro informando que o python não é um comando reconhecido ou se a Microsoft Store for aberta, o Python não está instalado. Feche a Microsoft Store se abrir automaticamente; é melhor fazer o download de um instalador oficial do que usar a versão da Microsoft.

Se o Python não estiver instalado em seu sistema ou se você vir uma versão anterior ao Python 3.9, será necessário fazer o download de um instalador Python para Windows. Acesse https://python.org e passe o mouse sobre o link **Downloads**. Você verá um botão de download com a versão mais recente do Python. Clique no botão para iniciar automaticamente o download do instalador correto para o seu sistema. Após o download do arquivo, execute o instalador. Não se esqueça de selecionar a opção **Add Python to PATH**, pois isso facilita a configuração adequada do seu sistema. A Figura 1.1 mostra esta opção selecionada.

Figura 1.1: Não se esqueça de selecionar a caixa de seleção Add Python to PATH.

Executando o Python em uma sessão de terminal

Abra uma nova janela do Prompt de Comando e digite **python** em letras minúsculas. Você deve ver um prompt do Python (>>>), isso significa que o Windows encontrou a versão do Python que acabamos de instalar.

```
C:\> python
Python 3.x.x (main, Jun . . . , 13:29:14) [MSC v.1932 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

NOTA *Se você não ver essa saída ou algo parecido, confira as instruções mais detalhadas de configuração no Apêndice A.*

Digite a seguinte linha em sua sessão do Python:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Você verá a saída Hello Python interpreter! Sempre que quiser executar um trecho de código Python, abra uma janela do Prompt de Comando e inicie uma sessão de terminal Python. Para fechar a

sessão do terminal, pressione CTRL+Z e, em seguida, pressione ENTER ou digite o comando **exit()**.

Instalando o VS Code

Você pode fazer o download do instalador do VS Code em https://code.visualstudio.com. Clique no botão **Download for Windows** e execute o instalador. Ignore as seções a seguir sobre macOS e Linux e siga as etapas em "*Executando um programa Hello World*" na página 40.

Python no macOS

O Python não vem instalado por padrão nas versões mais recentes do macOS. Assim, você precisará instalá-lo caso ainda não tenha instalado. Nesta seção, instalaremos a versão mais recente do Python e, em seguida, instalaremos o VS Code e verificaremos se está adequadamente configurado.

NOTA *O Python 2 era incluído em versões mais antigas do macOS, mas é uma versão desatualizada, não devemos usá-la.*

Verificando se o Python 3 está instalado

Abra uma janela de terminal clicando em **Aplicativos**4**Utilidades**4**Terminal**. Você também pode pressionar ⌘+**barra de espaço**, tipo **terminal**, em seguida, pressione ENTER. Para ver se existe uma versão recente o suficiente do Python instalada, digite **python3**. É bem provável que você veja uma mensagem sobre como instalar as *command line developer tools*. É melhor instalar essas ferramentas depois de instalar o Python, logo se esta mensagem aparecer, cancele a janela pop-up.

Se a saída mostrar que você tem o Python 3.9 ou uma versão posterior instalada, pule a próxima seção e vá para "Executando o Python em uma sessão de terminal". Se vir alguma versão anterior ao Python 3.9, siga as instruções na próxima seção para instalar a

versão mais recente.

Repare que no macOS, sempre que você vir o comando python neste livro, precisará usar o comando python3 para ter certeza de que está usando o Python 3. Na maioria dos sistemas macOS, o comando python aponta para uma versão desatualizada do Python que deve ser usada somente por ferramentas internas do sistema ou aponta para nada, gerando uma mensagem de erro.

Instalando a última versão do Python

Podemos encontrar um instalador Python para o seu sistema em https://python.org. Passe o mouse sobre o link **Download** e você verá um botão para baixar a versão mais recente do Python. Clique no botão para iniciar automaticamente o download do instalador correto para o seu sistema. Após o download do arquivo, execute o instalador.

Após a execução do instalador, uma janela Finder deve aparecer. Clique duas vezes no arquivo *Install Certificates.command*. A execução desse arquivo possibilitará a instalação mais fácil de bibliotecas adicionais necessárias para projetos do mundo real, incluindo os projetos da segunda metade deste livro

Executando o Python em uma sessão de terminal

Agora, podemos tentar executar trechos de código Python abrindo uma nova janela de terminal e digitando **python3**:

```
$ python3
Python 3.x.x (v3.11.0:eb0004c271, Jun . . . , 10:03:01)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Este comando inicia uma sessão de terminal Python. Você deve ver um prompt do Python (>>>), isso significa que o macOS encontrou a versão do Python que acabamos de instalar.

Digite a seguinte linha na sessão do terminal:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Você deve ver a mensagem Hello Python interpreter!, exibida diretamente na janela do terminal atual. É possível fechar o interpretador Python pressionando CTRL+D ou digitando o comando **exit()**.

NOTA *Em sistemas macOS mais recentes, você verá um sinal de porcentagem (%) como prompt de terminal em vez de um cifrão ($).*

Instalando o VS Code

Para instalar o editor VS Code é necessário fazer o download do instalador em https://code.visualstudio.com. Clique na seta à direita do botão **Download**, e vá para a pasta **Downloads**. Arraste o instalador do **Visual Studio Code** para a pasta Aplicativos e clique duas vezes no instalador para executá-lo.

Ignore a seção a seguir sobre Python no Linux e siga as etapas em "*Executando um programa Hello World*" na página 40.

Python no Linux

Como os sistemas Linux são arquitetados para programação, o Python já está instalado na maioria dos computadores Linux. As pessoas que escrevem e mantêm o Linux esperam que você faça sua própria programação em algum momento e o encorajam a fazê-lo. Por esta razão, há muito pouco para instalar e apenas algumas configurações a se fazer para começar a programar.

Verificando sua versão do Python

Abra uma janela de terminal executando o aplicativo Terminal em seu sistema (no Ubuntu, pressione CTRL+ALT+T). Para descobrir qual versão do Python está instalada, digite **python3** com *p* minúsculo. Quando o Python é instalado, este comando inicia o interpretador Python. Você deve ver a saída indicando qual versão

do Python está instalada. Você também deve ver um prompt do Python (>>>), em que pode começar a digitar comandos Python:

```
$ python3
Python 3.10.4 (main, Apr  . . . , 09:04:19) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Essa saída indica que o Python 3.10.4 é a versão atual e padrão do Python instalada no computador. Quando ver essa saída, pressione CTRL+D ou digite **exit()** para sair do prompt do Python e retornar ao prompt do terminal. Aqui, sempre que você vir o comando python, digite **python3**.

É necessário o Python 3.9 ou posterior para executar o código deste livro. Se a versão do Python instalada em seu sistema for anterior ao Python 3.9, ou caso queira atualizar para a versão mais recente disponível, confira as instruções no Apêndice A.

Executando o Python em uma sessão de terminal

Você pode tentar executar trechos de código Python abrindo um terminal e digitando **python3**, como fez ao verificar sua versão. Faça isso mais uma vez e, quando o Python for executado, digite a seguinte linha na sessão do terminal:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

A mensagem deve ser exibida diretamente na janela do terminal atual. Lembre-se de fechar o interpretador Python pressionando CTRL+D ou digitando o comando **exit()**.

Instalando o VS Code

No Ubuntu Linux é possível instalar o VS Code no Ubuntu Software Center. Clique no ícone do Ubuntu Software em seu menu e procure por *vscode*. Clique no aplicativo **Visual Studio Code** (às vezes chamado de *code*) e clique em **Instalar**. Uma vez instalado, procure em seu sistema por *VS Code* e inicie o aplicativo.

Executando um programa Hello World

Com uma versão recente do Python e do VS Code instalada, você está quase pronto para executar seu primeiro programa Python escrito em um editor de texto. Mas antes precisamos instalar uma extensão Python para VS Code.

Instalando a extensão Python para VS Code

O VS Code é compatível com muitas linguagens de programação diferentes; para aproveitar ao máximo como programador Python, é necessário instalar a extensão Python. Essa extensão inclui compatibilidade para escrever, editar e executar programas Python.

Para instalar a extensão Python clique no ícone Gerenciar, que se parece com uma engrenagem no canto inferior esquerdo do aplicativo VS Code. No menu que aparece, clique em **Extensões**. Digite **python** na caixa de pesquisa e clique em **Python extension**. (Caso veja mais de uma extensão chamada *Python*, escolha aquela fornecida pela Microsoft.) Clique **Instalar** e instale quaisquer ferramentas adicionais que seu sistema precise para completar a instalação. Se você vir uma mensagem de que precisa instalar o Python e já o fez, pode ignorá-la.

> **NOTA** *Se estiver usando o macOS e um pop-up solicitar que instale* command line developer tools*, clique em* ***Instalar.*** *Talvez você veja uma mensagem de que levará um tempo excessivamente longo para instalar, mas deve levar apenas cerca de 10 ou 20 minutos com uma.*

Executando hello_world.py

Antes de escrever seu primeiro programa, crie uma pasta chamada *python_work* em sua área de trabalho para seus projetos. É melhor usar letras minúsculas e underscores em vez de espaços em nomes de arquivos e pastas, já que o Python usa essas convenções de nomenclatura. É possível criar essa pasta em outro lugar que não

seja a área de trabalho, porém, será mais fácil seguir algumas etapas posteriores se você salvar a pasta *python_work* direto na área de trabalho.

Abra o VS Code e feche a guia **Introdução** se ainda estiver aberta. Crie um novo arquivo clicando em **Arquivo**4**Novo Arquivo** ou pressionando CTRL+N (⌘+N no macOS). Salve o arquivo como *hello_world.py* em sua pasta *python_work*. A extensão *.py* informa ao VS Code que seu arquivo está escrito em Python e como executar o programa e realçar o texto de maneira útil.

Após salvar o arquivo, digite a seguinte linha no editor:

hello_world.py

```
print("Hello Python world!")
```

Para executar seu programa, selecione **Executar**□**Executar sem Depuração** ou pressione CTRL+F5. Uma tela de terminal deve aparecer na parte inferior da janela do VS Code, mostrando a saída do seu programa:

```
Hello Python world!
```

Provavelmente, você verá uma saída adicional mostrando o interpretador Python usado para executar seu programa. Caso queira simplificar as informações exibidas para ver somente a saída do seu programa, consulte o Apêndice B. No Apêndice B, você também pode encontrar sugestões úteis sobre como utilizar o VS Code com mais eficiência.

Caso não veja essa saída, algo pode ter dado errado no programa. Verifique todos os caracteres digitados na linha. Você digitou sem querer o print com P maiúsculo? Esqueceu de uma ou ambas as aspas ou parênteses? As linguagens de programação trabalham com uma sintaxe muito específica e, caso não digite tudo de forma correta, receberá erros. Se não conseguir executar o programa, veja as sugestões na próxima seção.

Resolução de problemas

Se não conseguir executar o *hello_world.py*, veja a seguir algumas sugestões que você pode tentar e que também são boas soluções gerais para qualquer problema de programação:

- Quando um programa tem um erro significativo, o Python exibe uma *traceback*, um relatório de erros. O Python examina o arquivo e tenta identificar o problema. Verifique o traceback; isso pode lhe fornecer uma pista sobre o problema que está impedindo a execução do programa.
- Saia um pouco da frente do computador, faça uma breve pausa e tente novamente. Lembre-se de que a sintaxe é indispensável na programação. Ou seja, algo tão simples como aspas ou parênteses incompatíveis pode impedir que um programa seja executado de forma correta. Releia as partes relevantes deste capítulo, examine seu código e tente identificar o erro.
- Recomece tudo. É bem provável que não seja necessário desinstalar nenhum software, mas talvez seja uma boa excluir seu arquivo *hello_world.py* e recriá-lo do zero.
- Peça a outra pessoa para seguir as etapas deste capítulo, em seu computador ou em um computador diferente, e observe atentamente o que ela faz. Talvez você tenha se esquecido de um pequeno passo que outra pessoa não esqueceu.
- Confira as instruções adicionais de instalação no Apêndice A; alguns dos detalhes incluídos no Apêndice podem ajudá-lo a resolver seu problema.
- Ache alguém que conheça Python e peça para ajudá-lo a configurar. Caso pergunte às pessoas, pode descobrir que talvez conheça alguém que use Python.
- As instruções de configuração deste capítulo também estão disponíveis no site que acompanha este livro em https://ehmatthes.github.io/pcc_3e. Talvez a versão online dessas instruções lhe sirva melhor, pois você pode simplesmente recortar e colar o código e clicar nos links para os recursos necessários.
- Peça ajuda online. O Apêndice C disponibiliza uma série de

recursos, como fóruns e sites de bate-papo ao vivo, em que é possível pedir ajuda de soluções para pessoas que já resolveram o problema que você está enfrentando.

Nunca se preocupe em incomodar programadores experientes. Todo programador já se sentiu perdido em algum momento, e a maioria deles fica contente em ajudá-lo a configurar devidamente seu sistema. Contanto que você consiga expor claramente o que está tentando fazer, o que já tentou e os resultados que está obtendo, há uma boa chance de alguém ajudá-lo. Conforme mencionado na introdução, a comunidade Python é muito amigável e prestativa com iniciantes.

O Python deve executar sem problemas em qualquer computador moderno. Problemas iniciais de configuração podem ser frustrantes, mas vale a pena resolvê-los. Assim que executar o *hello_world.py*, você pode começar a aprender Python e suas atividades de programação se tornarão mais interessantes e satisfatórias.

Executando programas Python em um terminal

Você executará a maioria dos programas diretamente em seu editor de texto. No entanto, às vezes é interessante executar programas em um terminal. Por exemplo, talvez você queira executar um programa existente sem abri-lo para edição.

Isso é possível em qualquer sistema com o Python instalado, caso você saiba como acessar o diretório em que o arquivo do programa está armazenado. Para isso, verifique se salvou o arquivo *hello_world.py* na pasta *python_work* em sua área de trabalho.

No Windows

Você pode usar o comando **cd**, de *alterar o diretório*, navegar pelo sistema de arquivos em uma janela de comando. O comando **dir**, de *diretório*, mostra todos os arquivos que existem no diretório atual.

Abra uma nova janela de terminal e digite os seguintes comandos para executar *hello_world.py*:

```
C:\> cd Desktop\python_work
C:\Desktop\python_work> dir
hello_world.py
C:\Desktop\python_work> python hello_world.py
Hello Python world!
```

Primeiro, use o comando **cd** para navegar até a pasta *python_work*, que está na pasta *Desktop*. Em seguida, use o comando **dir** para verificar se *hello_world.py* está nessa pasta. Depois, execute o arquivo usando o comando **python hello_world.py**.

A maioria dos seus programas executará perfeita e diretamente em seu editor. Apesar disso, conforme suas atividades ficam mais complexas, você desejará executar alguns de seus programas em um terminal.

No macOS e no Linux

Executar um programa Python em uma sessão de terminal é a mesma coisa no Linux e no macOS. Você pode usar o comando **cd**, de *alterar o diretório*, navegar pelo sistema de arquivos em uma janela de comando. O comando **ls**, de *Lista*, mostra todos os arquivos não ocultos e existentes do diretório atual.

Abra uma nova janela de terminal e digite os seguintes comandos para executar *hello_world.py*:

```
~$ cd Desktop/python_work/
~/Desktop/python_work$ ls
hello_world.py
~/Desktop/python_work$ python3 hello_world.py
Hello Python world!
```

Primeiro, use o comando **cd** para navegar até a pasta *python_work*, que está na pasta *Desktop*. Em seguida, use o comando **ls** para verificar se *hello_world.py* está nessa pasta. Depois, execute o arquivo usando o comando **python3 hello_world.py**.

A maioria dos seus programas executará perfeita e diretamente em

seu editor. Apesar disso, conforme suas atividades ficam mais complexas, você desejará executar alguns de seus programas em um terminal.

FAÇA VOCÊ MESMO

Os exercícios deste capítulo são de natureza exploratória. A partir do Capítulo 2, os desafios que resolverá serão baseados no que você aprendeu.

1.1 python.org: Explore a página inicial do Python (https://python.org) para encontrar tópicos de seu interesse. À medida que se familiarizar com o Python, diferentes partes do site serão mais proveitosas para você.

1.2 Erros de digitação do Hello World: Abra o arquivo *hello_world.py* que acabou de criar. Cometa de propósito um erro de digitação em algum lugar da linha e execute o programa novamente. Você consegue cometer um erro de digitação que gera um erro? Consegue entender a mensagem de erro? Consegue cometer um erro de digitação que não gere um erro? Por que você acha que não gerou erro?

1.3 Habilidades infinitas: Caso tivesse habilidades infinitas de programação, o que você desenvolveria? Você está prestes a aprender a programar. Caso tenha um objetivo final em mente, você usará imediatamente suas habilidades; agora é um ótimo momento para descrever de forma sucinta o que você deseja criar. É um bom hábito manter um caderno de "ideias" que você possa consultar sempre que quiser começar um projeto novo. Agora, reserve alguns minutos para registrar três programas que você deseja criar.

Recapitulando

Neste capítulo, você aprendeu um pouco sobre o Python e instalou o Python em seu sistema, se não tivesse instalado. Você também instalou um editor de texto para escrever com facilidade código Python. Executamos trechos de código Python em uma sessão de terminal, e você executou seu primeiro programa, *hello_world.py*. Provavelmente, você aprendeu também um pouco sobre solução de problemas.

No próximo capítulo, você aprenderá sobre os diferentes tipos de dados com os quais pode trabalhar em seus programas Python e começará também a usar variáveis.

CAPÍTULO 2

Variáveis e tipos de dados simples

Neste capítulo, aprenderemos sobre os diferentes tipos de dados com os quais podemos trabalhar em nossos programas Python. Aprenderemos também a utilizar variáveis para representar dados em nossos programas.

O que realmente ocorre quando executamos o hello_world.py

Vamos analisar melhor o que o Python faz quando executamos o *hello_world.py*. Como se pode ver, o Python realiza um volume considerável de tarefas, mesmo quando executa um simples programa:

hello_world.py

```
print("Hello Python world!")
```

Ao executar esse código, você deve ver a seguinte saída:

```
Hello Python world!
```

Ao executarmos o arquivo *hello_world.py,* o final *.py* indica que o arquivo é um programa Python. Assim, o editor passa o programa ao *interpretador Python*, que lê o programa e verifica o que cada palavra significa. Por exemplo, quando vê a palavra print seguida por parênteses, o interpretador exibe na tela seja lá o que estiver dentro dos parênteses.

Conforme escreve seus programas, o editor destaca diferentes

partes do programa de formas distintas. Por exemplo, o editor reconhece que `print()` é o nome de uma função e exibe essa palavra em uma cor. Reconhece que `"Hello Python world!"` não é um código Python e exibe essa frase em uma cor diferente. Esse recurso se chama *realce de sintaxe* e é bastante útil quando você começa a escrever os próprios programas.

Variáveis

Vamos tentar usar uma variável com o *hello_world.py*. Adicione uma nova linha no início do arquivo e modifique a segunda linha:

hello_world.py

```
message = "Hello Python world!"
print(message)
```

Execute esse programa para ver o que acontece. Você deve ver a mesma saída que antes:

```
Hello Python world!
```

Adicionamos uma *variável* chamada `message`. Toda variável guarda um *valor*, informação associada a essa variável. Nesse caso, o valor é o texto `"Hello Python world!"`.

Adicionar uma variável ocasiona um pouco mais de trabalho para o interpretador Python. Ao processar a primeira linha, o interpretador associa a variável `message` com o texto `"Hello Python world!"`. Ao chegar à segunda linha, exibe na tela o valor associado a `message`.

Vamos incrementar esse programa modificando o *hello_world.py* para que exiba uma segunda mensagem. Adicione uma linha em branco ao *hello_world.py* e, em seguida, acrescente duas novas linhas de código:

```
message = "Hello Python world!"
print(message)

message = "Hello Python Crash Course world!"
print(message)
```

Agora, ao executar o *hello_world.py*, você deverá ver duas linhas como saída:

```
Hello Python world!
Hello Python Crash Course world!
```

É possível mudar o valor de uma variável em seu programa a qualquer momento, e o Python sempre manterá o registro do valor atual.

Nomeando e usando variáveis

No Python, ao usar variáveis, é necessário seguir algumas regras e diretrizes. Infringir algumas dessas regras causará erros; outras diretrizes apenas o ajudam a escrever um código mais fácil de ler e entender. Ao trabalhar com variáveis, não se esqueça das seguintes regras:

- Nomes de variáveis podem ter somente letras, números e underscores. Podem começar com uma letra ou um underscore, mas não com um número. Por exemplo, podemos nomear uma variável como `message_1`, mas não como `1_message`.
- Não são permitidos espaços em nomes de variáveis, mas são permitidos underscores para separar palavras em nomes de variáveis. Por exemplo, `greeting_message` executa, mas `greeting message` causará erros.
- Evite utilizar palavras reservadas e nomes de funções Python como nomes de variáveis. Por exemplo, não use a palavra `print` como nome de variável; o Python a reservou para um propósito específico de programação. (Veja a seção "*Palavras Reservadas e Funções Built-in do Python*" na página 573.)
- Nomes de variáveis devem ser sucintos, mas descritivos. Por exemplo, `name` é melhor que `n`, `student_name` é melhor que `s_n`, e `name_length` é melhor que `length_of_persons_name`.
- Atenção ao usar a letra minúscula *I* e a letra maiúscula *O*, já que podem ser confundidas com os números *1* e *0*.

Aprender a criar bons nomes de variáveis, sobretudo quando seus

programas se tornam mais interessantes e complicados, requer um pouco de prática. À medida que escreve mais programas e começa a ler o código de outras pessoas, você se torna mais apto em criar nomes pertinentes.

NOTA *As variáveis Python que você está usando neste momento devem estar em letras minúsculas. Caso use letras maiúsculas, o Python não retornará erros, porém, letras maiúsculas em nomes de variáveis têm significados especiais que abordaremos nos próximos capítulos.*

Evitando erros em nomes ao usar variáveis

Todo programador comete erros, e a maioria comete erros diariamente. Embora possam cometer erros, bons programadores também sabem como responder a esses erros com eficiência. Vamos analisar um erro que você provavelmente cometerá no início e vamos aprender a corrigi-lo.

Vamos escrever um código que gera intencionalmente um erro. Digite o seguinte código, inclusive a palavra com erro ortográfico mesage, em negrito:

```
message = "Hello Python Crash Course reader!"
print(mesage)
```

Quando seu programa apresenta um erro, o interpretador Python faz o possível para ajudá-lo a descobrir onde está o problema. O interpretador fornece um traceback quando um programa não pode ser executado com êxito. Um *traceback* é um registro de onde o interpretador encontrou problemas ao tentar rodar seu código. Vejamos um exemplo do traceback que o Python fornece depois que o nome de uma variável foi digitado incorretamente sem querer:

```
Traceback (most recent call last):
1  File "hello_world.py", line 2, in <module>
2    print(mesage)
           ^^^^^^
3 NameError: name 'mesage' is not defined. Did you mean: 'message'?
```

A saída informa que ocorre um erro na linha 2 do arquivo

hello_world.py 1. O interpretador mostra essa linha 2 para nos ajudar a identificar de imediato o erro e nos informar que tipo de erro foi encontrado 3. Nesse caso, encontrou um *name error*, informando que a variável que está sendo exibida, mesage, não foi definida. O Python não pode identificar o nome fornecido da variável. Em geral, um erro de nome significa que esquecemos de definir o valor de uma variável antes de usá-la ou cometemos um erro de ortografia ao digitar o nome da variável. Se encontrar um nome de variável semelhante ao que não reconhece, o Python perguntará se esse é o nome que você pretendia usar.

Nesse exemplo, omitimos a letra *s* no nome da variável message na segunda linha. O Python não verifica a ortografia do seu código, mas assegura que os nomes das variáveis sejam escritos de modo coerente. Por exemplo, observe o que acontece quando escrevemos *message* de forma incorreta na linha que define a variável:

```
mesage = "Hello Python Crash Course reader!"
print(mesage)
```

Neste caso, o programa executou com sucesso!

```
Hello Python Crash Course reader!
```

Os nomes das variáveis correspondem, logo o Python não vê problemas. Apesar de serem meticulosas, as linguagens de programação desconsideram erros ortográficos. Como resultado, não é necessário considerar as regras de ortografia e gramática em inglês ou em português ao tentar criar nomes de variáveis e escrever código.

Muitos erros de programação são erros simples de digitação de um caractere na linha de um programa. Caso passe um tempão procurando por um desses erros, saiba que você não é o único. Muitos programadores experientes e talentosos passam horas procurando esses tipos de errinhos. Leve na brincadeira e siga em frente, saiba que isso acontecerá com frequência durante sua jornada de programação.

Variáveis são rótulos

Não raro, variáveis são descritas como caixas nas quais podemos armazenar valores. Talvez essa ideia possa ajudá-lo nas primeiras vezes em que usa uma variável, mesmo não sendo uma forma precisa de descrever como as variáveis são representadas internamente no Python. É bem melhor considerar variáveis como rótulos que podemos atribuir a valores. Podemos dizer também que uma variável referencia determinado valor.

Possivelmente, essa distinção pouco importa em seus primeiros programas, mas vale a pena aprender isso agora do que depois. Em dado momento, você verá um comportamento inesperado de uma variável e compreender exatamente como as variáveis funcionam o ajudará a identificar o que está acontecendo em seu código.

NOTA *A melhor forma de entender novos conceitos de programação é tentar usá-los em seus programas. Se você ficar empacado enquanto faz um exercício deste livro, tente fazer outra coisa por um tempo. Caso continue empacado, revise a parte relevante deste capítulo. Se ainda precisar de ajuda, confira as sugestões no Apêndice C.*

FAÇA VOCÊ MESMO

Escreva um programa separado para cada um desses exercícios. Salve cada programa com um nome de arquivo que siga as convenções padrão do Python, usando letras minúsculas e underscores, como *simple_message.py* e *simple_messages.py*.

2.1 Simple Message: Atribua uma mensagem a uma variável e exiba essa mensagem.

2.2 Simple Messages: Atribua uma mensagem a uma variável e exiba essa mensagem. Em seguida, mude o valor da variável para uma nova mensagem e mostre a nova mensagem.

Strings

Como a maioria dos programas define e agrega algum tipo de dado e, em seguida, faz algo de útil com ele, isso ajuda a classificar diferentes tipos de dados. O primeiro tipo de dados que veremos é a string. À primeira vista, strings são bem simples, mas podemos usá-

las das mais variadas formas.

Uma *string* é uma série de caracteres. No Python, seja lá o que estiver entre aspas é considerado uma string, e podemos usar aspas simples ou duplas em torno das strings desse jeito:

```
"This is a string."
'This is also a string.'
```

Essa flexibilidade possibilita que usemos aspas e apóstrofos em nossas strings:

```
'I told my friend, "Python is my favorite language!"'
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive community."
```

Vamos explorar algumas formas de usar as strings.

Alterando letras maiúsculas e minúsculas em uma string com métodos

Uma das tarefas mais simples que podemos fazer com strings é alterar as letras maiúsculas e minúsculas das palavras. Veja o código a seguir e tente descobrir o que está acontecendo:

name.py

```
name = "ada lovelace"
print(name.title())
```

Salve este arquivo como *name.py* e depois execute-o. Você deve ver essa saída:

```
Ada Lovelace
```

Neste exemplo, a variável `name` se refere à string em letra minúscula `"ada lovelace"`. O método `title()` aparece após chamar a variável no `print()`. Um *método* é uma ação que o Python pode executar em um dado. O ponto (`.`) após `name` no `name.title()` comunica ao Python que o método `title()` deve trabalhar na variável `name`. Todo método é seguido por um conjunto de parênteses, já que os métodos geralmente precisam de informações adicionais para realizar suas tarefas. Essas informações são fornecidas entre parênteses. A função `title()` não precisa de nenhuma informação adicional, logo seus parênteses estão vazios.

O método title() altera a primeira letra de cada palavra para maiúsculas, cada palavra começa com uma letra maiúscula. Isso é prático, já que muitas vezes queremos atribuir um nome como uma informação. Por exemplo, talvez você queira que seu programa reconheça os valores de entrada Ada, ADA, e ada como o mesmo nome e exiba todos como Ada.

Diversos outros métodos úteis também estão disponíveis para manipular letras maiúsculas e minúsculas. Por exemplo, é possível alterar uma string para que contenha letras maiúsculas ou letras minúsculas assim:

```
name = "Ada Lovelace"
print(name.upper())
print(name.lower())
```

O código exibirá:

```
ADA LOVELACE
ada lovelace
```

O método lower() é bastante útil para armazenar dados. Via de regra, não podemos confiar nas letras maiúsculas e minúsculas fornecidas pelos usuários. Desse modo, converteremos as strings em letras minúsculas antes de armazená-las. Assim, quando quisermos exibir as informações, usaremos as letras mais inteligíveis para cada string.

Usando variáveis em strings

Em determinadas situações queremos usar o valor de uma variável dentro de uma string. Por exemplo, talvez você queira usar duas variáveis para representar um nome e um sobrenome, respectivamente, e depois queira combiná-las para exibir o nome completo de alguém:

full_ame. py

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
print(full_name)
```

Para inserir o valor de uma variável em uma string, coloque a letra f

imediatamente antes da aspa inicial 1. Coloque chaves ao redor do nome ou nomes de qualquer variável que você quer usar dentro da string. O Python substituirá cada variável por seu valor quando a string for exibida.

Essas strings se chamam *f-strings*. O *f* é de *formato*, pois o Python formata a string substituindo o nome de qualquer variável entre chaves por seu valor. A saída do código anterior é:

```
ada lovelace
```

É possível fazer muitas coisas com as f-strings. Por exemplo, você pode usar f-strings para compor mensagens completas usando as informações associadas a uma variável, conforme mostrado a seguir:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
1 print(f"Hello, {full_name.title()}!")
```

full_name é usado em uma frase que cumprimenta o usuário 1, e o método title() altera o nome para letras maiúsculas e minúsculas. O código retorna um simples cumprimento, elegantemente formatado:

```
Hello, Ada Lovelace!
```

É possível também usar f-strings para compor uma mensagem e, em seguida, atribuir a mensagem inteira a uma variável:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
1 message = f"Hello, {full_name.title()}!"
2 print(message)
```

Este código exibe a mensagem Hello, Ada Lovelace! também, mas ao atribuir a mensagem a uma variável 1, simplificamos a chamada do print() 2.

Adicionando espaço em branco a strings com tabs ou quebras de linhas

Em programação, *espaço em branco* se refere a quaisquer caracteres não exibíveis, como espaços, tabulações e símbolos de

fim de linha. Podemos utilizar espaços em branco para organizar saídas, de modo que seja facilmente legíveis aos usuários.

Para adicionar uma tabulação em seu texto, use a combinação de caracteres \t:

```
>>> print("Python")
Python
>>> print("\tPython")
    Python
```

Para adicionar uma nova quebra de linha em uma string, use a combinação de caracteres \n:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

É possível também combinar tabulações e novas quebras de linhas em uma única string. A string "\n\t" informa ao Python para passar para uma nova linha e iniciar a próxima linha com uma tabulação. O exemplo a seguir demonstra como você pode usar uma string de uma linha para gerar uma saída com quatro linhas:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

Nos próximos dois capítulos, quebras de linhas e tabulações serão de grande serventia, já que começaremos a gerar a saída de muitas linhas a partir de algumas linhas de código.

Removendo espaços em branco com o strip()

Espaços em branco extras podem deixar nossos programas confusos. Para os programadores, 'python ' e 'python ' parecem basicamente iguais. Mas para um programa são duas strings diferentes. O Python identifica o espaço extra em 'python ' e o considera importante, a menos que você escreva o contrário.

É essencial considerar espaços em branco porque muitas vezes queremos comparar duas strings para determinar se são iguais. Por exemplo, podemos ter uma situação importante envolvendo verificação dos nomes de usuário das pessoas que logaram em um site. Espaços em branco extras também podem gerar confusão em situações mais simples. Felizmente, o Python facilita a remoção de espaços em branco extras dos dados inseridos pelas pessoas.

O Python pode procurar espaços em branco extras nos lados direito e esquerdo de uma string. A fim de garantir que não tenha nenhum espaço em branco no lado direito de uma string, use o método rstrip():

```
>>> favorite_language = 'python '
>>> favorite_language
'python '
>>> favorite_language.rstrip()
'python'
>>> favorite_language
'python '
```

O valor associado ao favorite_language 1 contém espaço em branco extra no final da string. Ao solicitarmos esse valor ao Python em uma sessão de terminal, podemos ver o espaço no final do valor 2. Quando o método rstrip() itera na variável favorite_language 3, esse espaço extra é removido. No entanto, é removido apenas temporariamente. Caso solicite o valor de favorite_language mais uma vez, a string terá a mesma aparência de quando foi inserida, incluindo o espaço em branco extra 4.

Para remover o espaço em branco da string de forma definitiva, é necessário associar o valor removido ao nome da variável:

```
>>> favorite_language = 'python '
>>> favorite_language = favorite_language.rstrip()
>>> favorite_language
'python'
```

A fim de remover o espaço em branco da string, retiramos o espaço em branco do lado direito da string e associamos esse novo valor à variável original 1. Em programação é comum alterar o valor de uma

variável. É assim que o valor de uma variável pode ser atualizado à medida que um programa é executado ou em resposta à entrada do usuário.

É possível também removermos o espaço em branco do lado esquerdo de uma string com o método lstrip(), ou de ambos os lados ao mesmo tempo usando o strip():

```
1 >>> favorite_language = ' python '
2 >>> favorite_language.rstrip()
  ' python'
3 >>> favorite_language.lstrip()
  'python '
4 >>> favorite_language.strip()
  'python'
```

Nesse exemplo, começamos com um valor que tem espaços em branco no início e no final 1. Em seguida, removemos o espaço extra do lado direito 2, do lado esquerdo 3, e de ambos os lados 4. Testar essas funções de remoção pode ajudá-lo a se familiarizar com a manipulação de strings. Em um contexto real, essas funções de remoção são usadas quase sempre para limpar entradas de usuário antes que sejam armazenadas em um programa.

Removendo prefixos

Ao trabalhar com strings, outra tarefa comum é remover um prefixo. Considere um URL com o típico prefixo *https://*. Queremos remover esse prefixo para que possamos focar somente a parte da URL que os usuários precisam inserir em uma barra de endereços. Vejamos como fazer isso:

```
>>> nostarch_url = 'https://nostarch.com'
>>> nostarch_url.removeprefix('https://')
'nostarch.com'
```

Digite o nome da variável seguido por um ponto e, em seguida, o método removeprefix(). Dentro dos parênteses insira o prefixo que deseja remover da string original.

Assim como os métodos para remover espaços em branco, o

removeprefix() deixa a string original inalterada. Se quiser manter o novo valor com o prefixo removido, reatribua-o à variável original ou atribua-o a uma nova variável:

```
>>> simple_url = nostarch_url.removeprefix('https://')
```

Quando vemos um URL em uma barra de endereço e a parte *https://* não é mostrada, o navegador provavelmente está usando um método como o removeprefix() nos bastidores.

Evitando erros de sintaxe com strings

Um tipo de erro que podemos ver com certa frequência é um erro de sintaxe. Um *erro de sintaxe* ocorre quando o Python não reconhece uma seção do seu programa como código Python válido. Por exemplo, caso use um apóstrofo entre aspas simples, você gerará um erro. Isso ocorre porque o Python interpreta tudo dentro da primeira aspa simples e do apóstrofo como uma string. Em seguida, o Python tenta interpretar o restante do texto como código Python, provocando erros.

Vejamos como usar devidamente aspas simples e duplas. Salve este programa como *apostrophe.py* e depois execute-o:

apostrophe.py

```
message = "One of Python's strengths is its diverse community."
print(message)
```

O apóstrofo aparece dentro de um conjunto de aspas duplas, assim o interpretador Python não tem problemas para ler adequadamente a string:

```
One of Python's strengths is its diverse community.
```

Contudo, se utilizarmos aspas simples, o Python não conseguirá identificar onde a string deve terminar:

```
message = 'One of Python's strengths is its diverse community.'
print(message)
```

Você verá a seguinte saída:

```
  File "apostrophe.py", line 1
    message = 'One of Python's strengths is its diverse community.'
```

```
1 ^
SyntaxError: unterminated string literal (detected at line 1)
```

Na saída, vemos que o erro ocorre logo após a aspa simples final `1`. Esse erro de sintaxe sinaliza que o interpretador não reconhece algo no código como código Python válido, acreditando que o problema possa ser uma string que não está devidamente entre aspas. Erros podem se originar de diversas fontes, e ressaltarei aquelas mais comuns à medida que forem surgindo. Você pode ver erros de sintaxe com frequência à medida que aprende a escrever adequadamente código Python. Os erros de sintaxe também são o tipo menos específico de erro. Por isso, podem ser difíceis e frustrantes de identificar e corrigir. Caso fique empacado em um erro persistente, confira as sugestões no Apêndice C.

NOTA *O recurso de realce de sintaxe do seu editor deve ajudá-lo a identificar alguns erros de sintaxe de imediato à medida que escreve seus programas. Caso veja o código Python destacado como se fosse inglês ou inglês destacado como se fosse código Python, provavelmente há uma aspa incompatível em algum lugar do arquivo.*

FAÇA VOCÊ MESMO

Salve cada um dos exercícios a seguir em um arquivo separado, com um *nome como name_cases.py*. Se ficar empacado, faça uma pausa ou confira as sugestões no Apêndice C.

2.3 Mensagem pessoal: Use uma variável para representar o nome de uma pessoa e exiba uma mensagem para essa pessoa. Sua mensagem deve ser simples, como "Olá Eric, gostaria de aprender um pouco de Python hoje?"

2.4 Maiúsculas e minúsculas: Use uma variável para representar o nome de uma pessoa e, em seguida, exiba o nome dessa pessoa em letras minúsculas, maiúsculas e as primeiras letras maiúsculas.

2.5 Citação famosa: Encontre uma citação de uma pessoa famosa que você admira. Exiba a citação e o nome do autor. Sua saída deve se parecer com a seguinte, incluindo as aspas:

Albert Einstein disse uma vez: "Uma pessoa que nunca cometeu um erro nunca tentou nada de novo".

2.6 Citação famosa 2: Repita o Exercício 2.5, mas desta vez represente o nome da pessoa famosa usando uma variável chamada famous_person. Depois, escreva sua

mensagem e a represente com uma nova variável chamada message. Printe sua mensagem.

2.7 Removendo nomes: Use uma variável para representar o nome de uma pessoa e inclua alguns caracteres de espaço em branco no início e no final do nome. Lembre-se de usar cada combinação de caracteres, "\t" e "\n", pelo menos uma vez.

Exiba o nome uma vez para que o espaço em branco ao redor do nome seja mostrado. Em seguida, printe o nome usando cada uma das três funções de remoção, lstrip(), rstrip(), e strip().

2.8 Extensões de arquivo: O Python tem um método removesuffix() que funciona exatamente como removeprefix(). Atribua o valor 'python_notes.txt' a uma variável chamada filename. Depois, utilize o método removesuffix() para exibir o nome do arquivo sem a extensão do arquivo, como alguns navegadores de arquivos fazem.

Números

Na programação, os números são usados com bastante frequência para armazenar pontuação de jogos, representar dados em visualizações, armazenar informações em aplicativos web e assim por diante. O Python trata os números de diversas formas diferentes, dependendo de como estão sendo usados. Analisaremos primeiramente como o Python lida com números inteiros, já que os inteiros são os mais simples de se trabalhar.

Inteiros

No Python é possível somar (+), subtrair (-), multiplicar (*) e dividir (/) números inteiros.

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

Caso execute essas operações na sessão do terminal, o Python simplesmente retorna o resultado da operação. O Python usa dois símbolos de multiplicação para representar expoentes:

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

O Python também suporta a ordem de precedência das operações, logo podemos usar inúmeras operações em uma expressão. Podemos também utilizar parênteses para modificar a ordem das operações para que o Python possa avaliar a expressão na ordem especificada. Por exemplo:

```
>>> 2 + 3*4
14
>>> (2 + 3) * 4
20
```

Nesses exemplos, o espaçamento não impacta como o Python avalia as expressões; os espaços simplesmente nos ajudam a identificar mais rápido as operações que têm prioridade quando estamos lendo o código.

Floats

O Python chama qualquer número com um ponto decimal de número de ponto flutuante [*float*]. Esse termo é usado na maioria das linguagens de programação e refere-se ao fato de que um ponto decimal pode aparecer em qualquer posição de um número. Deve-se arquitetar minuciosamente toda linguagem de programação para lidar de forma adequada com os números decimais, de modo que tenham o devido comportamento, independentemente de onde o ponto decimal apareça.

Na maioria dos casos podemos utilizar floats sem nos preocuparmos como se comportam. Basta digitar os números que quer usar, e o Python provavelmente fará o que você espera:

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
```

```
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

No entanto, fique atento de que às vezes você pode obter um número arbitrário de casas decimais como resposta:

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

Isso ocorre em todas as linguagens, sendo pouco preocupante. O Python tenta encontrar uma maneira de representar o resultado com a maior precisão possível, o que às vezes é difícil, pois os computadores precisam representar números internamente. Por ora, apenas ignore as casas decimais extras; nos projetos da Parte II, aprenderemos como lidar com as casas extras quando for necessário.

Inteiros e floats

Ao realizar a divisão de dois números quaisquer, mesmo que sejam inteiros que resultem em um número inteiro, sempre obteremos um float:

```
>>> 4/2
2.0
```

Caso misture um inteiro e um float em qualquer outra operação, você também obterá um float:

```
>>> 1 + 2.0
3.0
>>> 2 * 3.0
6.0
>>> 3.0 ** 2
9.0
```

O padrão Python é um float em qualquer operação que use um float, mesmo que a saída seja um número inteiro.

Underscores em números

Ao escrever números grandes, é possível agrupar dígitos usando underscores para tornar os números grandes mais legíveis:

```
>>> universe_age = 14_000_000_000
```

Ao printar um número que foi definido com underscores, o Python exibe apenas os dígitos:

```
>>> print(universe_age)
14000000000
```

O Python ignora os underscore ao armazenar esses tipos de valores. Mesmo se não agruparmos os dígitos em três, o valor ainda não será afetado. Para o Python, `1000` é o mesmo que `1_000`, que é o mesmo que `10_00`. Esse recurso funciona com números inteiros e floats.

Atribuição múltipla

Podemos atribuir valores a mais de uma variável usando somente uma única linha de código. Isso pode ajudar a sintetizar seus programas, facilitando a legibilidade deles; você usará essa técnica com mais frequência ao inicializar um conjunto de números.

Por exemplo, veja como inicializar as variáveis `x`, `y`, e `z` para zero:

```
>>> x, y, z = 0, 0, 0
```

É necessário separar os nomes das variáveis com vírgulas e fazer o mesmo com os valores, e o Python atribuirá cada valor à sua respectiva variável. Desde que o número de valores corresponda ao número de variáveis, o Python irá combiná-las de forma correta.

Constantes

Uma *constante* é uma variável cujo valor permanece o mesmo durante a vida de um programa. O Python não tem tipos de constantes built-in, mas os programadores Python usam letras maiúsculas para indicar que uma variável deve ser tratada como uma constante e nunca ser alterada:

```
MAX_CONNECTIONS = 5000
```

Quando quiser tratar uma variável como uma constante em seu código, escreva o nome da variável com todas as letras maiúsculas.

FAÇA VOCÊ MESMO

2.9 Número Oito: Escreva operações de adição, subtração, multiplicação e divisão que resultem cada uma no número 8. Não se esqueça de incluir suas operações em um print() para conferir os resultados. Você deve criar quatro linhas mais ou menos assim:

```
print(5+3)
```

Sua saída deve ter quatro linhas, com o número 8 aparecendo uma vez em cada linha.

2.10 Número favorito: Use uma variável para representar seu número favorito. Em seguida, usando essa variável, crie uma mensagem que revele seu número favorito. Printe essa mensagem.

Comentários

Os comentários são um recurso de grande utilidade na maioria das linguagens de programação. Até agora, tudo o que você escreveu em seus programas foi código Python. À medida que seus programas ficam mais extensos e mais complicados, é necessário adicionar notas em seus programas que descrevam sua abordagem geral para o problema que está resolvendo. Um *comentário* possibilita que você escreva notas em seu próprio idioma, dentro de seus programas.

Como escrever comentários?

No Python, a marca de hash (#) indica um comentário. Qualquer coisa após uma marca de hash em seu código é ignorada pelo interpretador Python. Por exemplo:

comment.py

```
# diga olá a todos
print("Hello Python people!")
```

O Python ignora a primeira linha e executa a segunda linha.

```
Hello Python people!
```

Quais tipos de comentários você deve escrever?

A principal justificativa para escrever comentários é explicar o que seu código deve fazer e como você faz o código rodar. Ao trabalhar em um projeto, você entende como todas as peças se encaixam. Mas ao retomar um projeto depois de algum tempo ausente, você provavelmente esquece alguns detalhes. Sempre é possível estudar seu código por um tempo e entender como os segmentos devem rodar, porém, escrever bons comentários pode lhe poupar tempo, já que você sintetiza de forma clara sua abordagem geral.

Caso queira se tornar um programador profissional ou colaborar com outros programadores, você deve escrever comentários pertinentes. Hoje, a maioria dos softwares é escrita de maneira colaborativa, seja por um grupo de funcionários de uma empresa ou por um grupo de pessoas que trabalham juntas em um projeto open source. Programadores habilidosos esperam ver comentários no código. Ou seja, é melhor começar a adicionar comentários descritivos aos seus programas agora mesmo. Escrever comentários claros e concisos em seu código é um dos hábitos mais positivos que você pode adquirir como programador iniciante.

Ao decidir se deve escrever um comentário ou não, pergunte a si mesmo se teve que considerar diversas abordagens antes de recorrer a uma maneira razoável de fazer algo funcionar; se for o caso, escreva um comentário sobre sua solução. É mais fácil excluir comentários extras posteriormente do que voltar e escrever comentários em um programa pouco comentado. De agora em diante usarei comentários em exemplos no decorrer deste livro para ajudar a explicar seções de código.

FAÇA VOCÊ MESMO

2.11 Adicionando comentários: Escolha dois dos programas que você escreveu e adicione pelo menos um comentário a cada um. Caso não tenha nada específico para escrever porque até agora seus programas são muito simples, basta acrescentar seu nome e a data atual no início de cada arquivo do programa. Em seguida, escreva uma frase descrevendo o que o programa faz.

Zen do Python

Programadores Python experientes o incentivarão a evitar a complexidade e buscar a simplicidade sempre que possível. A filosofia da comunidade Python está no " Zen do Python" de Tim Peters. Você pode acessar esse breve conjunto de princípios para escrever um bom código Python digitando import this em seu interpretador. Não reproduzirei todo o "Zen do Python" aqui, mas compartilharei algumas linhas para ajudá-lo a compreender como esses princípios são de suma importância para os programadores iniciantes em Python.

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
```

[Bonito é melhor que feio] Os programadores Python abraçam a noção de que o código pode ser bonito e elegante. Na programação, as pessoas resolvem problemas. Os programadores sempre respeitam soluções bem desenvolvidas, eficientes e até bonitas para os problemas. Conforme você aprende mais sobre o Python e escreve mais código, um dia, talvez uma pessoa espie seu programa e diga: "Uau, que código lindo!".

```
Simple is better than complex.
```

[Simples é melhor que complexo] Se tiver a chance de escolher entre uma solução simples e uma complexa, e ambas funcionarem, use a solução simples. A manutenção de seu código será mais fácil, e será mais fácil para você e outras pessoas usarem esse código mais tarde.

```
Complex is better than complicated.
```

[Complexo é melhor que complicado] A vida real é caótica e, às vezes, uma solução simples para um problema é impossível. Nesse caso, use a solução mais simples que funcione.

```
Readability counts.
```

[A legibilidade conta] Ainda que seu código seja complexo, procure

torná-lo legível. Quando estiver trabalhando em um projeto que envolva codificação complexa, foque os comentários elucidativos.

> There should be one-- and preferably only one --obvious way to do it.

[Deve haver uma – e, de preferência, somente uma – maneira óbvia de fazer algo] Se pedirem a dois programadores para resolver o mesmo problema, eles deverão apresentar soluções bastante compatíveis. Isso não significa que não há espaço para criatividade na programação. Pelo contrário, há muito espaço para a criatividade! No entanto, boa parte da programação consiste em usar abordagens incrementais e comuns para situações simples, dentro de um projeto maior e mais criativo. Outros programadores Python devem entender os detalhes práticos de seus programas.

> Now is better than never.

[Agora é melhor do que nunca] Você até poderia passar o resto de sua vida aprendendo todos as especificidades do Python e da programação em geral, mas nunca concluiria nenhum projeto. Não tente escrever um código perfeito; escreva um código que funcione e, depois, decida se quer melhorar seu código para esse projeto ou partir para algo novo.

Conforme você avança para o próximo capítulo e começa a se aprofundar em tópicos mais complexos, tente manter essa filosofia de simplicidade e clareza em mente. Programadores experientes respeitarão mais seu código e ficarão contente em lhe fornecer feedback e colaborar com você em projetos interessantes.

FAÇA VOCÊ MESMO

2.12 Zen do Python: Digite import this em uma sessão do terminal Python e leia rapidamente os princípios adicionais.

Recapitulando

Neste capítulo, aprendemos como trabalhar com variáveis e a usar nomes de variáveis descritivos e resolver erros de nome e de sintaxe quando surgem. Aprendemos o que são strings e como exibi-las

usando letras minúsculas, maiúsculas e inicias maiúsculas. A princípio, utilizamos espaços em branco para organizar a saída de forma ordenada e aprendemos a remover elementos desnecessários de uma string. Depois, começamos a trabalhar com números inteiros e floats e aprendemos algumas forma de trabalhar com dados numéricos. Além disso, aprendemos a escrever comentários explicativos para que você e outras pessoas leiam seu código de modo mais fácil. Por último, lemos a filosofia de manter seu código o mais simples possível, sempre que possível.

No Capítulo 3 aprenderemos como armazenar coleções de informações em estruturas de dados chamadas *listas*, e também a trabalhar com uma lista, manipulando qualquer informação que esteja nela.

CAPÍTULO 3
Introdução às listas

Neste capítulo e no próximo aprenderemos o que são listas e como começar a trabalhar com os elementos de uma lista. As listas possibilitam armazenar conjuntos de informações em um só lugar, mesmo se tivermos apenas alguns ou milhões de itens. As listas são um dos recursos mais poderosos do Python, de fácil acesso a novos programadores e relacionam muitos conceitos importantes da programação.

O que é uma lista?

Uma *lista* é uma coleção de itens em uma ordem específica. É possível criar uma lista incluindo as letras do alfabeto, os algarismos de 0 a 9, ou o nome de todos os seus familiares. Você pode inserir o que bem entender em uma lista, e os itens em sua lista não precisam estar relacionados de nenhuma forma especial. Via de regra, como uma lista contém mais de um elemento, é uma boa ideia nomeá-la no plural como letters, digits ou names.

No Python, colchetes ([]) designam uma lista, e os elementos individuais de uma lista são separados por vírgulas. Vejamos um exemplo simples de uma lista contendo alguns tipos de bicicletas:

bicycles.py

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

Se pedir para exibir uma lista, o Python retornará sua representação da lista, incluindo os colchetes:

```
['trek', 'cannondale', 'redline', 'specialized']
```

Como essa não é a saída que queremos que os usuários vejam, aprenderemos como acessar os itens individuais em uma lista.

Acessando os elementos em uma lista

Listas são coleções ordenadas, ou seja, podemos acessar qualquer elemento em uma lista informando a posição ou *índice* do item desejado ao Python. Para acessar um elemento em uma lista, escreva o nome dela seguido do índice do item entre colchetes.

Por exemplo, vamos extrair a primeira bicicleta da lista `bicycles`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0])
```

Quando pedimos um único item de uma lista, o Python retorna somente esse elemento sem colchetes:

```
trek
```

Este é o resultado que queremos que os usuários vejam: uma saída limpa e ordenadamente formatada.

Podemos também utilizar os métodos string do Capítulo 2 em qualquer elemento desta lista. Por exemplo, podemos formatar o elemento `'trek'` para parecer mais apresentável usando o método `title()`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0].title())
```

Esse exemplo gera a mesma saída que o exemplo anterior, a salvo que `'Trek'` está com a primeira letra maiúscula.

As posições do índice começam em 0, não em 1

O Python considera que o primeiro item de uma lista está na posição 0, não na posição 1. Isso se aplica à maioria das linguagens de programação, devido ao fato de como as operações de lista são implementadas em um nível mais baixo. Se estiver recebendo resultados inesperados, pergunte a si mesmo se está cometendo um simples erro off-by-one (erro de lógica).

O segundo item em uma lista tem um índice 1. Ao usar esse sistema numérico, podemos acessar qualquer elemento que quisermos de uma lista subtraindo um de sua posição na lista. Por exemplo, para acessar o quarto item em uma lista, consulte o item no índice 3.

O código a seguir consulta as bicicletas nos índices `1` e `3`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])
```

Esse código retorna a segunda e a quarta bicicleta na lista:

```
cannondale
specialized
```

O Python tem uma sintaxe singular para acessar o último elemento de uma lista. Se pedirmos o elemento no índice `-1`, sempre retorna o último elemento da lista:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

Esse código retorna o valor `'specialized'`. Essa sintaxe é muito prática, pois muitas vezes queremos acessar os últimos elementos de uma lista, sem sabermos exatamente qual é o tamanho dela. Essa convenção também se aplica a outros valores negativos de índice. O índice `-2` retorna o segundo elemento do final da lista, o índice `-3` retorna o terceiro elemento do final, e assim por diante.

Usando valores individuais de uma lista

Podemos usar valores individuais de uma lista, como faríamos com qualquer outra variável. Por exemplo, você pode utilizar f-strings para criar uma mensagem com base em um valor de uma lista.

Vamos tentar acessar a primeira bicicleta da lista e compor uma mensagem utilizando este valor:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
message = f"My first bicycle was a {bicycles[0].title()}."
print(message)
```

Criamos uma frase usando o valor em `bicycles[0]` e atribuindo-o à variável `message`. A saída é uma frase simples com a primeira bicicleta

da lista:

```
My first bicycle was a Trek.
```

FAÇA VOCÊ MESMO

Tente criar os seguintes programas curtos a fim de adquirir um pouco de experiência pessoal com as listas do Python. Talvez você queira criar uma nova pasta para os exercícios de cada capítulo, assim pode mantê-los organizados.

3.1 Nomes: Armazene o nome de alguns de seus amigos em uma lista chamada names. Exiba o nome de cada pessoa acessando cada elemento da lista, um de cada vez.

3.2 Cumprimentos: Comece com a lista usada no Exercício 3.1, mas em vez de apenas exibir o nome de cada pessoa, exiba também uma mensagem para elas. O texto de cada mensagem deve ser o mesmo, porém, cada mensagem deve ser personalizada com o nome da pessoa.

3.3 Sua própria lista: Pense em seu meio de transporte favorito, como uma moto ou um carro, e crie uma lista que armazene diversos exemplos. Use sua lista para exibir uma série de declarações sobre esses itens, como "Gostaria de ter uma moto da Honda".

Modificando, adicionando e removendo elementos

A maioria das listas que criar será *dinâmica*, ou seja, você criará uma lista e, em seguida, adicionará e removerá elementos dela à medida que seu programa evoluir. Por exemplo, podemos criar um jogo no qual um jogador tem que atirar em alienígenas que rasgam o céu. Poderíamos armazenar o conjunto inicial de alienígenas em uma lista e, depois, remover um alienígena da lista cada vez que um deles for abatido. Sempre que um novo alienígena aparecer na tela, você o adiciona à lista. Ao longo do jogo, sua lista de alienígenas aumentará e diminuirá.

Modificando elementos em uma lista

A sintaxe para modificar um elemento se assemelha à sintaxe para acessar um elemento em uma lista. Para alterar um elemento, use o nome da lista seguido pelo índice do elemento que queira alterar e forneça o valor novo que quer que esse elemento tenha.

Por exemplo, digamos que temos uma lista de motos e o primeiro elemento da lista é `'honda'`. É possível mudar o valor deste primeiro elemento após criar a lista:

motorcycles.py

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

motorcycles[0] = 'ducati'
print(motorcycles)
```

Aqui, definimos a lista `motorcycles`, com `'honda'` sendo o primeiro elemento. Em seguida, mudamos o valor do primeiro elemento para `'ducati'`. A saída mostra que o primeiro elemento foi alterado, ao passo que o restante da lista permanece o mesmo:

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

É possível alterar o valor de qualquer elemento em uma lista, não somente do primeiro item.

Adicionando elementos a uma lista

Talvez você queira adicionar um elemento novo a uma lista, independentemente do motivo. Por exemplo, talvez você deseje criar alienígenas novos para um jogo, adicionar dados novos a uma visualização ou usuários novos registrados a um site que criou. O Python disponibiliza inúmeras maneiras de adicionar dados novos para listas existentes.

Anexando elementos ao final de uma lista

A forma mais simples de adicionar um elemento novo a uma lista é *anexar* o elemento nela. Ao anexar um elemento a uma lista, esse elemento novo é adicionado ao final dela. Usando a mesma lista do exemplo anterior, adicionaremos o elemento novo `'ducati'` ao final dela:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
```

```
motorcycles.append('ducati')
print(motorcycles)
```

Aqui, o método append() adiciona 'ducati' ao final da lista, sem afetar nenhum dos outros elementos dela:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

O método append() facilita criar listas dinamicamente. Por exemplo, é possível começar com uma lista vazia e depois adicionar elementos nela usando uma série de métodos append(). Em uma lista vazia adicionaremos os elementos 'honda', 'yamaha' e 'suzuki' nela:

```
motorcycles = []

motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

A lista decorrente é exatamente igual às listas dos exemplos anteriores:

```
['honda', 'yamaha', 'suzuki']
```

Criar listas assim é bastante comum, porque muitas vezes não saberemos os dados que os usuários querem armazenar em um programa até que o programa seja executado. Para que seus usuários assumam o controle, comece definindo uma lista vazia para armazenar os valores dos usuários. Em seguida, anexe com o método append() cada valor novo fornecido à lista que acabou de criar.

Inserindo elementos em uma lista

Podemos adicionar um elemento novo em qualquer posição em sua lista usando o método insert(). Faremos isso especificando o índice do elemento novo e o valor do elemento novo:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

motorcycles.insert(0, 'ducati')
print(motorcycles)
```

Nesse exemplo, inserimos o valor 'ducati' no início da lista. O método insert() abre um espaço na posição 0 e armazena o valor 'ducati' nesse local:

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

Essa operação desloca todos os outros valores uma posição à direita na lista.

Removendo elementos de uma lista

Não raro, queremos remover um elemento ou um conjunto de elementos de uma lista. Por exemplo, quando um jogador abate um alienígena do céu, você provavelmente vai querer removê-lo da lista de alienígenas ativos. Ou quando um usuário decidir cancelar a conta em sua aplicação web, talvez você queira removê-lo da lista de usuários ativos. Podemos remover um elemento de acordo com sua posição na lista ou seu valor.

Removendo um elemento usando a instrução del

Caso saiba a posição do elemento que deseja remover de uma lista, você pode usar a instrução del:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[0]
print(motorcycles)
```

Aqui, usamos a instrução del para remover o primeiro elemento, 'honda', da lista de motos:

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

É possível remover um elemento de qualquer posição em uma lista usando a instrução del se você souber seu índice. Por exemplo, veja como remover o segundo elemento, 'yamaha', da lista:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
```

```
del motorcycles[1]
print(motorcycles)
```

A segunda moto é deletada da lista:

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

Em ambos os exemplos, não podemos mais acessar o valor que foi removido da lista após usarmos a instrução `del`.

Removendo um elemento com o método pop()

Às vezes, queremos utilizar o valor de um elemento depois de removê-lo de uma lista. Por exemplo, talvez você queira acessar a posição *x* e *y* de um alienígena que acabou de ser abatido, de modo que consiga desenhar uma explosão nessa posição. Em uma aplicação web, talvez você queira remover um usuário de uma lista de membros ativos e, em seguida, adicionar esse usuário a uma lista de membros inativos.

O método `pop()` remove o último elemento de uma lista, mas possibilita que você trabalhe com esse elemento após removê-lo. O termo *pop* se origina da ideia de considerar uma lista como uma pilha de itens, removendo um item do topo da pilha. Analogicamente, o topo de uma pilha corresponde ao final de uma lista.

Usaremos o método pop() para remover uma moto da lista:

```
1 motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

2 popped_motorcycle = motorcycles.pop()
3 print(motorcycles)
4 print(popped_motorcycle)
```

Começamos definindo e mostrando a lista `motorcycles` `1`. Depois, removemos um valor da lista e atribuímos esse valor à variável `popped_motorcycle` `2`. Exibimos a lista `3` a fim de mostrar um valor removido dela. Em seguida, exibimos o valor removido `4` para provar que ainda temos acesso ao valor.

A saída demonstra que o valor 'suzuki' foi removido do final da lista e agora está atribuído à variável popped_motorcycle:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

Qual é a serventia desse método pop()? Imagine que as motos da lista estão armazenadas em ordem cronológica, conforme a data de compra. Nesse caso, podemos utilizar o método pop() para exibir explicações sobre a última moto que compramos:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print(f"The last motorcycle I owned was a {last_owned.title()}.")
```

A saída é uma frase simples sobre a moto recém-comprada:

```
The last motorcycle I owned was a Suzuki.
```

Removendo elementos de qualquer posição em uma lista

É possível usar o pop() para remover um elemento de qualquer posição em uma lista, incluindo o índice do elemento que queira remover entre parênteses:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

first_owned = motorcycles.pop(0)
print(f"The first motorcycle I owned was a {first_owned.title()}.")
```

Começamos removendo a primeira moto da lista e, em seguida, exibimos uma mensagem a respeito. A saída é uma frase simples com a primeira moto comprada:

```
The first motorcycle I owned was a Honda.
```

Lembre-se de que sempre que usar o método pop(), o elemento com o qual está trabalhando não fica mais armazenado na lista.

Caso esteja inseguro se deve usar a instrução del ou o método pop(), pense o seguinte: quando quiser deletar um elemento de uma lista e não for nunca mais for usá-lo, recorra à instrução del. Agora, se

desejar usar o elemento mesmo ao removê-lo, recorra ao método pop().

Removendo um elemento por valor

Em alguns casos, não sabemos a posição do valor que queremos remover de uma lista. Se souber apenas o valor do elemento que quer remover, é possível usar o método remove().

Por exemplo, digamos que queremos remover o valor 'ducati' da lista de motos:

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

motorcycles.remove('ducati')
print(motorcycles)
```

Aqui, o método remove() solicita ao Python que identifique onde 'ducati' aparece na lista e remova esse elemento:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

Podemos também utilizar o método remove() para trabalhar com um valor que está sendo removido de uma lista. Removeremos o valor 'ducati' e exibiremos a justificativa para removê-lo da lista:

```
1 motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
  print(motorcycles)

2 too_expensive = 'ducati'
3 motorcycles.remove(too_expensive)
  print(motorcycles)
4 print(f"\nA {too_expensive.title()} is too expensive for me.")
```

Após definir a lista 1, atribuímos o valor 'ducati' a uma variável chamada too_expensive 2. Depois, usamos essa variável para informar ao Python qual valor remover da lista 3. O valor 'ducati' foi removido da lista 4, mas ainda está acessível pela variável too_expensive, possibilitando exibir uma justificativa sobre porque removemos 'ducati' da lista de motos:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

A Ducati is too expensive for me.

NOTA *O método* remove() *deleta somente a primeira ocorrência do valor especificado. Se existir a possibilidade de o valor aparecer mais de uma vez na lista, é necessário usar um loop a fim de garantir que todas as ocorrências do valor sejam removidas. Aprenderemos como fazer isso no Capítulo 7.*

FAÇA VOCÊ MESMO

Os exercícios seguintes são um pouco mais complexos do que os do Capítulo 2, mas lhe possibilitam usar as listas de todas as formas descritas.

3.4 Lista de convidados: Se pudesse convidar qualquer pessoa, viva ou falecida, para um jantar, quem você convidaria? Crie uma lista que tenha pelo menos três pessoas que gostaria de convidar para um jantar. Em seguida, use sua lista a fim de exibir uma mensagem para cada pessoa, convidando-a para o jantar.

3.5 Mudando a lista de convidados: Você acabou de ficar sabendo que um dos convidados não conseguirá ir ao jantar, assim precisa enviar um conjunto novo de convites. É necessário convidar outra pessoa.

- Comece com o programa do Exercício 3.4. No final do programa, adicione um print(), informando o nome do convidado que não irá ao jantar.
- Modifique sua lista substituindo o nome do convidado que não pode comparecer pelo nome da pessoa nova que você está convidando.
- Exiba um segundo conjunto de mensagens de convite, uma para cada pessoa que ainda não consta em sua lista.

3.6 Mais convidados: Você acabou de encontrar uma mesa maior de jantar, agora há mais espaço disponível. Convide mais três pessoas para o jantar.

- Comece com o programa do Exercício 3.4 ou 3.5. No final do programa, adicione um print(), informando às pessoas que encontrou uma mesa maior.
- Use um insert() para adicionar um convidado novo ao início de sua lista.
- Use um insert() para adicionar um convidado novo no meio de sua lista.
- Use um append() para adicionar um convidado novo no final de sua lista.
- Exiba um conjunto novo de mensagens de convite, um para cada pessoa em sua lista.

3.7 Reduzindo a lista de convidados: Você acabou de descobrir que sua mesa nova de jantar não chegará a tempo e agora tem espaço somente para dois convidados.

- Comece com o programa do Exercício 3.6. Adicione uma linha nova que exiba uma mensagem que você pode convidar apenas duas pessoas para o jantar.
- Use o pop() para remover convidados de sua lista, um de cada vez, até que restem

somente dois nomes nela. Sempre que remover um nome de sua lista, exiba uma mensagem para essa pessoa informando que lamenta por não poder convidá-la para o jantar.

- Exiba uma mensagem para cada uma das duas pessoas que ainda estão na sua lista, informando que ainda estão convidadas.
- Use o del para remover os dois últimos nomes de sua lista, para que ela fique vazia. Exiba sua lista para ter certeza de que você realmente tem uma lista vazia no final do seu programa.

Organizando uma lista

Não raro, as listas serão criadas em uma ordem imprevisível, já que você nem sempre pode controlar a ordem em que seus usuários fornecem os dados. Apesar de isso ser inevitável na maioria das circunstâncias, muitas vezes queremos apresentar as informações em uma ordem específica. Às vezes queremos preservar a ordem original da lista, já outras vezes queremos alterar a ordem original. O Python disponibiliza uma série de formas diferentes para organizar suas listas, dependendo da situação.

Ordenando uma lista permanentemente com o método sort()

O método sort() do Python facilita relativamente a ordenação de uma lista. Imagine que temos uma lista de carros e queremos mudar sua ordem para armazená-los em ordem alfabética. Simplificando as coisas: vamos supor que todos os valores da lista estejam em letras minúsculas:

cars.py

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort()
print(cars)
```

O método sort() altera permanentemente a ordem da lista. Agora, os carros estão em ordem alfabética, e nunca podemos reverter para a ordem original:

```
['audi', 'bmw', 'subaru', 'toyota']
```

Podemos também ordenar essa lista em ordem alfabética reversa passando o argumento reverse=True para o método sort(). O exemplo a seguir ordena a lista de carros em ordem alfabética reversa:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True)
print(cars)
```

Mais uma vez, a ordem da lista é permanentemente alterada:

```
['toyota', 'subaru', 'bmw', 'audi']
```

Ordenando uma lista temporariamente com a função sorted()

A fim de conservar a ordem original de uma lista, mas apresentá-la em ordem ordenada, podemos utilizar a função sorted(). A função sorted() possibilita exibir sua lista em uma ordem específica, porém, não afeta a ordem original da lista.

Testaremos essa função com a lista de carros.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

print("Here is the original list:")
print(cars)

print("\nHere is the sorted list:")
print(sorted(cars))

print("\nHere is the original list again:")
print(cars)
```

Primeiro, exibimos a lista em sua ordem original 1 e, em seguida, em ordem alfabética 2. Após a lista ser exibida em uma ordem nova, mostramos que a lista ainda está armazenada em sua ordem original 3:

```
Here is the original list:
['bmw', 'audi', 'toyota', 'subaru']

Here is the sorted list:
['audi', 'bmw', 'subaru', 'toyota']
```

```
1 Here is the original list again:
  ['bmw', 'audi', 'toyota', 'subaru']
```

Repare que a lista ainda existe em sua ordem original 1 após a função sorted() ter sido usada. A função sorted() também pode aceitar um argumento reverse=True, caso queira exibir uma lista em ordem alfabética reversa.

NOTA *Ordenar uma lista em ordem alfabética é um pouco mais complicado quando todos os valores não estão em letras minúsculas. Ao definir uma sequência de ordenação, temos diversas formas de interpretar letras maiúsculas. No entanto, especificar a ordem exata pode ser mais complexo do que queremos lidar neste momento. Apesar disso, a maioria das abordagens de ordenação terá como base o que você aprendeu nesta seção.*

Exibindo uma lista em ordem inversa

Para reverter a ordem original de uma lista, podemos utilizar o método reverse(). Originalmente, se armazenamos a lista de carros na ordem cronológica de compra, poderíamos facilmente reordenar a lista em ordem cronológica inversa:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)

cars.reverse()
print(cars)
```

Veja que o reverse() não reordena em ordem alfabética; o método simplesmente inverte a ordem da lista:

```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
```

O método reverse() altera permanentemente a ordem de uma lista, mas podemos reverter para a ordem original a qualquer momento aplicando reverse() à mesma lista uma segunda vez.

Identificando o tamanho de uma lista

É possível encontrar rapidamente o tamanho de uma lista usando a

função len(). Neste exemplo, a lista tem quatro elementos, logo seu tamanho é 4:

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
4
```

A função len() será conveniente quando for necessário identificarmos o número de alienígenas que ainda precisam ser abatidos em um jogo, determinar a quantidade de dados que precisamos manipular em uma visualização ou descobrir o número de usuários registrados em um site, entre outras tarefas.

NOTA *O Python calcula os itens em uma lista começando com um. Logo, você não deve se deparar com nenhum erro ao definir o tamanho de uma lista.*

FAÇA VOCÊ MESMO

3.8 Conhecendo o mundo: Pense em pelo menos cinco lugares do mundo que você gostaria de conhecer.

- Armazene esses locais em uma lista. Verifique se ela não está em ordem alfabética.
- Exiba sua lista na ordem original. Não se preocupe em exibir a lista ordenadamente; basta exibi-la como uma lista crua do Python.
- Use sorted() para exibir sua lista em ordem alfabética, sem alterar a lista original.
- Mostre que sua lista ainda está na ordem original exibindo-a.
- Use o sorted() para exibir sua lista em ordem alfabética reversa, sem alterar a ordem original dela.
- Demonstre que sua lista ainda está na ordem original exibindo-a mais uma vez.
- Use o reverse() para alterar a ordem de sua lista. Exiba essa lista para mostrar que sua ordem foi alterada.
- Use o reverse() para alterar a ordem de sua lista novamente. Exiba-a a fim de mostrar que voltou à ordem original.
- Use o sort() para alterar sua lista para que ela seja armazenada em ordem alfabética. Exiba a lista para mostrar que sua ordem foi alterada.
- Use sort() para alterar sua lista, de modo que ela seja armazenada em ordem alfabética inversa. Exiba a lista para mostrar que sua ordem foi alterada.

3.9 Convidados para o jantar: Recorra a um dos programas dos exercícios 3.4 a 3.7 (páginas *75*-*76*), e use len() para exibir uma mensagem indicando o número de pessoas que você está convidando para jantar.

3.10 Funções: Pense em coisas que você conseguiria armazenar em uma lista. Por exemplo, você pode criar uma lista de montanhas, rios, países, cidades, idiomas, ou qualquer outra coisa que quiser. Crie um programa com uma lista contendo esses itens e, em seguida, use cada função apresentada neste capítulo, pelo menos, uma vez.

Evitando erros de índice ao trabalhar com listas

Quando trabalhamos com listas pela primeira vez, podemos cometer um tipo de erro comum. Digamos que você tenha uma lista com três elementos e tenta acessar um quarto item:

motorcycles.py

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[3])
```

Esse exemplo gera um *erro de índice*:

```
Traceback (most recent call last):
  File "motorcycles.py", line 2, in <module>
    print(motorcycles[3])
          ~~~~~~~~~~~^^^
IndexError: list index out of range
```

O Python tenta acessar o elemento no índice 3. Mas quando pesquisa na lista, nenhum elemento de `motorcycles` tem índice 3. Devido à natureza off-by-one da indexação em listas, temos aqui um típico erro. As pessoas costumam achar que o terceiro elemento é o elemento número 3, pois começam a contar a partir de 1. Só que em Python o terceiro elemento é o número 2, porque a indexação começa a partir de 0.

Um erro de índice significa que o Python não consegue identificar um elemento no índice solicitado. Caso um erro de índice aconteça em seu programa, tente ajustar o índice que você está querendo acessar em um elemento. Depois, execute o programa mais uma vez para ver se os resultados estão corretos.

Lembre-se sempre de que, se quiser acessar o último elemento de uma lista, use o índice `-1`. Sempre funcionará, ainda que sua lista tenha mudado de tamanho desde a última vez que você a acessou:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1])
```

O índice `-1` sempre retorna o último item de uma lista, neste caso o valor `'suzuki'`:

```
suzuki
```

O único momento que essa abordagem causará um erro é quando tentamos acessar o último elemento de uma lista vazia:

```
motorcycles = []
print(motorcycles[-1])
```

Como não existe nenhum elemento em `motorcycles`, o Python retorna outro erro de índice:

```
Traceback (most recent call last):
  File "motorcyles.py", line 3, in <module>
    print(motorcycles[-1])
          ~~~~~~~~~~~^^^^
IndexError: list index out of range
```

Se um erro de índice acontecer e você não conseguir descobrir como resolvê-lo, tente exibir sua lista ou somente o tamanho de sua lista. Talvez sua lista pareça bem diferente do que você pensava, ainda mais se foi manipulada dinamicamente pelo seu programa. Ver a lista original ou o número exato de elementos nela pode ajudá-lo a resolver esses erros lógicos.

FAÇA VOCÊ MESMO

3.11 Erro intencional: Se você ainda não recebeu um erro de índice em um de seus programas, tente gerar um. Mude um índice em um de seus programas para gerar um erro de índice. Faça questão de corrigir o erro antes de fechar o programa.

Recapitulando

Neste capítulo, aprendemos o que são listas e como trabalhar com os elementos individuais de uma lista. Aprendemos como definir uma lista e como adicionar e remover elementos dela, a ordenar listas permanente e temporariamente para exibi-las, e também como encontrar o tamanho de uma lista e como evitar erros de índice ao trabalhar com listas.

No Capítulo 4, você aprenderá a trabalhar com elementos em uma lista de modo mais eficiente. Ao percorrer cada elemento em uma lista usando somente algumas linhas de código, você conseguirá trabalhar com eficiência, mesmo quando sua lista contiver milhares ou milhões de elementos.

CAPÍTULO 4
Trabalhando com listas

No Capítulo 3, aprendemos a criar uma simples lista e a trabalhar com os elementos individuais dessa lista. Neste capítulo, aprenderemos como usar os loops para percorrer uma lista inteira somente com algumas linhas de código, indiferentemente do tamanho da lista. Os *loops* possibilitam que executemos a mesma ação, ou conjunto de ações, com todos os elementos de uma lista. Desse modo, é possível trabalhar efetivamente com listas de qualquer tamanho, inclusive aquelas com milhares ou mesmo milhões de elementos.

Loops: percorrendo uma lista inteira

Queremos muitas vezes percorrer todos os itens de uma lista, executando a mesma tarefa em cada elemento. Por exemplo, em um jogo, talvez você queira deslocar todos os elementos na tela para que fiquem na mesma proporção. Com uma lista de números, talvez você queira realizar a mesma operação estatística em todos os elementos. Ou talvez queira exibir cada destaque de uma lista de artigos em um site. Quando quiser realizar a mesma ação com todos os elementos em uma lista, você pode usar o loop `for` do Python.

Digamos que temos uma lista com nomes de mágicos e queremos exibir cada nome dessa lista. É possível fazer isso acessando cada nome individualmente, mas essa abordagem ocasionaria muitos problemas. Por exemplo, fazer isso com uma lista extensa de nomes seria repetitivo. Além do mais teríamos que mudar nosso código sempre que o tamanho da lista mudasse. O uso do loop `for` evita

esses dois problemas, permitindo que o Python se encarregue internamente desses problemas.

Vamos utilizar um loop `for` para exibir cada nome em uma lista de mágicos:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician)
```

Começamos definindo uma lista, assim como fizemos no Capítulo 3. Em seguida, definimos um loop `for`. Essa linha solicita ao Python que extraia um nome da lista `magicians`, associando-o à variável `magician`. Depois, dizemos ao Python para exibir o nome que acabou de ser atribuído à `magician`. O Python então repete essas duas últimas linhas, uma vez para cada nome na lista. Talvez ajude mais ler esse código assim: "Para cada mágico na lista de mágicos, exiba o nome do mágico". A saída é uma simples exibição de cada nome na lista.

```
alice
david
carolina
```

Análise minuciosa dos loops

Loops são essenciais, já que é por meio deles que um computador normalmente automatiza tarefas repetitivas. Por exemplo, em um loop simples, como aquele que usamos em *magicians.py*, o Python inicialmente lê a primeira linha do loop:

```
for magician in magicians:
```

Essa linha informa ao Python para acessar o primeiro valor da lista `magicians` e associá-lo à variável `magician`. O primeiro valor é `'alice'`. Em seguida, o Python lê a próxima linha:

```
    print(magician)
```

O Python exibe o valor atual de `magicians`, que ainda é `'alice'`. Como a lista contém mais valores, o Python retorna à primeira linha do loop:

```
for magician in magicians:
```

Python acessa o próximo nome da lista, 'david', e associa esse valor à variável magician. O Python então executa a linha:

```
print(magician)
```

O Python exibe o valor atual de magician mais uma vez, que agora é 'david'. Então, o Python percorre todo o loop mais uma vez com o último valor da lista, 'carolina'. Como a lista não tem mais valores, o Python passa para a próxima linha do programa. Nesse caso, como não temos nada depois do loop for, o programa termina.

Ao usar loops pela primeira vez, lembre-se de que o conjunto de etapas é repetido uma vez para cada elemento da lista, independentemente de quantos elementos a lista tiver. Caso tenha um milhão de elementos em sua lista, o Python repetirá essas etapas um milhão de vezes – com extrema rapidez.

Lembre-se também de que ao escrever o próprio loop for, é possível escolher qualquer nome que quiser para a variável temporária que será associada a cada valor na lista. No entanto, é adequado optar por um nome relevante que represente um único elemento da lista. Por exemplo, vejamos uma boa forma de iniciar um loop for para uma lista de gatos, uma lista de cães e uma lista geral de itens:

```
for cat in cats:
for dog in dogs:
for item in list_of_items:
```

Essas convenções de nomenclatura podem ajudá-lo a acompanhar a ação que está sendo executada em cada elemento dentro do loop for. O uso de nomes singulares e plurais pode ajudar a identificar se uma seção de código está funcionando com um único elemento da lista ou com a lista inteira.

Fazendo mais tarefas dentro de um loop for

Podemos fazer praticamente qualquer coisa com cada elemento dentro de um loop for. Vamos desenvolver mais o exemplo anterior exibindo uma mensagem para cada mágico, comunicando-lhes que realizaram um belo truque:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
```

Nesse código, a única diferença é onde escrevemos uma mensagem para cada mágico, começando com o nome do mágico. Na primeira vez que percorremos o loop, o valor de `magician` é `'alice'`, então o Python inicia a primeira mensagem com o nome `'Alice'`. Na segunda vez, a mensagem começará com `'David'` e, na terceira vez, a mensagem começará com `'Carolina'`.

A saída mostra uma mensagem personalizada para cada mágico na lista:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
```

Podemos também escrever quantas linhas de código quisermos no loop `for`. Cada linha indentada após a linha `magician in magicians` está *dentro do loop*, e cada linha indentada é executada uma vez para cada valor na lista. Ou seja, você pode trabalhar o quanto quiser com cada valor da lista.

Adicionaremos uma segunda linha à nossa mensagem, dizendo a cada mágico que estamos ansiosos pelo próximo truque:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

Como indentamos ambos os `print()`, cada linha será executada uma vez para cada mágico da lista. A quebra de linha (`"\n"`) no segundo `print()` insere uma linha em branco após cada passagem pelo loop, o que acaba criando um conjunto de mensagens ordenadas para cada pessoa na lista:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
```

```
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

É possível usar quantas linhas quiser em seus loops for. Na prática, você muitas vezes achará conveniente realizar diversas operações diferentes com cada elemento de uma lista ao usar um loop for.

Fazendo mais tarefas após usar um loop for

O que acontece quando termina a execução de um loop for? Normalmente, queremos sintetizar um bloco de saída ou passarmos para outra tarefa que nosso programa tem que realizar.

Quaisquer linhas de código após o loop for não indentadas são executados uma vez sem repetição. Vamos escrever um agradecimento geral ao grupo de mágicos, agradecendo-lhes pelo excelente espetáculo. Para exibir essa mensagem em grupo, após a exibição de todas as mensagens individuais, inserimos a mensagem após o loop for, sem indentá-la:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")

print("Thank you, everyone. That was a great magic show!")
```

Os dois primeiros print() são repetidos uma vez para cada mágico da lista, conforme visto anteriormente. No entanto, como não está indentada, a última linha é exibida apenas uma vez:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!
```

Ao processarmos dados usando um loop for, descobrimos que é uma boa forma de sintetizar uma operação executada em um conjunto inteiro de dados. Por exemplo, é possível usar um loop for para inicializar um jogo executando uma lista de personagens, exibindo cada personagem na tela. Em seguida, após o loop, você pode escrever um pouco de código adicional que exiba um botão *Play Now* depois de todos os personagens terem sido desenhados na tela.

Evitando erros de indentação

O Python usa indentação para determinar como uma linha, ou grupo de linhas, está relacionada ao resto do programa. Nos exemplos anteriores, as linhas que exibiram mensagens para mágicos individuais eram parte do loop for, porque estavam indentadas. A indentação do Python facilita e muito a leitura do código. Basicamente, a indentação usa espaços em branco para forçá-lo a escrever um código formatado com uma estrutura visual clara. Em programas Python mais extensos, você verá blocos de código indentados em alguns níveis diferentes. Esses níveis de indentação o ajudam a obter uma ideia geral da organização total do programa.

Ao começar a escrever código que depende de indentação adequada, será necessário prestar atenção em alguns *erros de indentação*. Por exemplo, as pessoas às vezes indentam linhas de código que não precisam ser indentadas ou se esquecem de indentar linhas que precisam ser indentadas. Ver exemplos desses tipos de erros o ajudará a evitá-los no futuro, assim como a corrigi-los, quando aparecerem em seus programas.

Vamos examinar alguns dos erros de indentação mais comuns.

Esquecendo da indentação

Sempre indente a linha após a instrução for em um loop. Você pode até se esquecer da indentação, mas o Python não:

magiians.py

```
  magicians = ['alice', 'david', 'carolina']
  for magician in magicians:
1 print(magician)
```

O `print() 1` deveria estar indentado, mas não está. Ao esperar um bloco indentado e não encontrá-lo, o Python o alerta em qual linha houve um problema:

```
  File "magicians.py", line 3
    print(magician)
    ^
IndentationError: expected an indented block after 'for' statement on line 2
```

Em geral, você consegue resolver esse tipo de erro de indentação fazendo a indentação da linha ou das linhas imediatamente após a instrução `for`.

Esquecendo de indentar linhas adicionais

Não raro, o loop será executado sem erros, mas não gerará o resultado esperado. Talvez isso ocorra quando você está tentando executar diversas tarefas em um loop e acaba esquecendo de indentar algumas linhas.

Por exemplo, vejamos o que acontece quando nos esquecemos de indentar a segunda linha no loop que informa a cada mágico que estamos ansiosos pelo próximo truque:

```
  magicians = ['alice', 'david', 'carolina']
  for magician in magicians:
      print(f"{magician.title()}, that was a great trick!")
1 print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

O segundo `print() 1` deveria estar indentado, mas como encontra pelo menos uma linha indentada depois da instrução `for`, o Python não relata um erro. Assim sendo, o primeiro `print()` é executado uma vez para cada nome na lista porque está indentado. Como não está indentado, o segundo `print()` é executado somente uma vez após o término da execução do loop. Como o valor final associado a `magician` é `'carolina'`, ela é a única que recebe a mensagem "looking forward to

the next trick”:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Trata-se de um *erro lógico*. Apesar de a sintaxe ser um código Python válido, o código não gera o resultado desejado, já que ocorre um problema em sua lógica. Caso uma ação seja executada apenas uma vez, mas o esperado é que seja repetida uma vez para cada elemento em uma lista, averigue se é necessário indentar uma linha ou um grupo de linhas.

Indentando desnecessariamente

Se você indentar sem querer uma linha que não precisa ser indentada, o Python o alerta sobre a indentação inesperada:

hello_world.py

```
message = "Hello Python world!"
    print(message)
```

Não precisamos indentar o `print()`, pois não faz parte de um loop; logo, o Python relata o seguinte erro:

```
  File "hello_world.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

Faça a indentação somente quando tiver uma razão específica para tal, assim você evita erros de indentação inesperada. Até agora, nos programas que estamos escrevendo, as únicas linhas que devemos indentar são as ações que queremos repetir para cada elemento dentro de um loop `for`.

Indentando desnecessariamente após o loop

Caso indente sem querer o código que deve ser executado após o término de um loop, esse código será repetido uma vez para cada elemento da lista. Vez ou outra, isso faz com que o Python gere um

erro, mas normalmente, gera um erro lógico.

Por exemplo, vejamos o que acontece quando indentamos sem querer a linha que agradeceu aos grupo de mágicos por fazerem um bom espetáculo:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")

    print("Thank you everyone, that was a great magic show!")
```

Como está indentada 1, a última linha é exibida para cada pessoa:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

Thank you everyone, that was a great magic show!
David, that was a great trick!
I can't wait to see your next trick, David.

Thank you everyone, that was a great magic show!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you everyone, that was a great magic show!
```

Trata-se de outro erro lógico, semelhante ao erro da seção "*Esquecendo de indentar linhas adicionais*" na página 87. Já que não faz ideia do que você está tentando fazer em seu código, o Python executará todo o código escrito em uma sintaxe válida. Se uma ação for repetida muitas vezes quando deveria ser executada somente uma vez, provavelmente será necessário remover a indentação do código dessa ação.

Esquecendo os dois-pontos

Os dois-pontos no final de uma instrução `for` informam ao Python para interpretar a próxima linha como o início de um loop:

```
  magicians = ['alice', 'david', 'carolina']
1 for magician in magicians
    print(magician)
```

Se, por um acaso, você esquecer os dois-pontos `1`, receberá um erro de sintaxe porque o Python não sabe exatamente o que você está tentando fazer:

```
  File "magicians.py", line 2
    for magician in magicians
                          ^
SyntaxError: expected ':'
```

O Python não sabe se você simplesmente esqueceu os dois-pontos ou se pretendia escrever um pouco mais de código para um loop mais complexo. Se conseguir identificar uma possível correção, o interpretador sugerirá uma, como adicionar dois-pontos no final de uma linha, como fez com a resposta `expected ':'`. Alguns erros podem ser corrigidos fácil e claramente, graças às sugestões dos tracebacks do Python. Alguns erros são mais difíceis de resolver, mesmo quando a eventual correção envolve um único caractere. Não se sinta mal quando levar um tempo para corrigir um erro; lembre-se: você não é o único que está passando por isso.

FAÇA VOCÊ MESMO

4.1 Pizzas: Pense em, pelo menos, três tipos que você gosta. Armazene esses nomes de pizza em uma lista e use um loop for para exibir o nome de cada uma.

- Modifique seu loop for a fim de que exiba uma frase usando o nome da pizza, em vez de exibir apenas o nome da pizza. Para cada pizza, você deve gerar uma linha de saída com uma simples afirmação como: *Gosto de pizza de pepperoni*.
- Adicione uma linha no final do seu programa, fora do loop for, que ressalte o quanto você gosta de pizza. A saída deve ter três ou mais linhas sobre os tipos de pizza que você gosta e, em seguida, uma frase adicional, como *Eu amo pizza!*

4.2 Animais: Pense em, pelo menos, três animais diferentes que compartilhem uma característica comum. Armazene o nome desses animais em uma lista e, em seguida, use um loop for para exibir o nome de cada animal.

- Modifique seu programa a fim de exibir uma afirmação sobre cada animal, como *Um cachorro seria um ótimo animal de estimação (pet).*
- Adicione uma linha no final do seu programa, indicando o que esses animais compartilham em comum. Você pode exibir uma frase, como *Qualquer um desses animais daria um ótimo animal de estimação!*

Criando listas numéricas

Não faltam razões para armazenar um conjunto de números. Por exemplo, suponha que você precise manter os registros das posições de cada personagem em um jogo e também dos escores mais altos de um jogador. Com as visualizações de dados, quase sempre trabalharemos com conjuntos de números, como temperaturas, distâncias, tamanhos de população ou valores de latitude e longitude, dentre outros tipos de conjuntos numéricos.

As listas são perfeitas para armazenar conjuntos de números, e o Python fornece uma variedade de ferramentas para ajudá-lo a trabalhar efetivamente com listas de números. Após entender como usar essas ferramentas de modo eficaz, seu código funcionará bem, mesmo quando suas listas contiverem milhões de elementos.

Usando a função range

A função `range()` do Python facilita gerar uma série de números. Por exemplo, podemos utilizar a função `range()` para exibir uma série de números como esse:

first_numbers.py

```
for value in range(1, 5):
    print(value)
```

Aparentemente, o código deveria exibir os números de 1 a 5, mas o número 5 não é exibido:

```
1
2
3
4
```

Neste exemplo, o `range()` exibe somente os números de 1 a 4. Trata-se de outro resultado do comportamento off-by-one, dentre muitos, que você verá com frequência nas linguagens de programação. Aqui, quando fornecemos dois valores, a função `range()` faz com que o Python comece a calcular o primeiro valor fornecido, parando no segundo valor fornecido. Como o Python parou o cálculo nesse

segundo valor, a saída nunca contém o valor final que, nesse caso, seria 5.

Para exibir os números de 1 a 5, devemos usar range(1, 6):

```
for value in range(1, 6):
    print(value)
```

Desta vez, a saída começa em 1 e termina em 5:

```
1
2
3
4
5
```

Se você usar o range() e sua saída for diferente do esperado, tente ajustar o valor final por 1.

Você também pode passar somente um argumento para o range(), assim, iniciará a sequência de números em 0. Por exemplo, range(6) retornaria os números de 0 a 5.

Usando o range() para criar uma lista de números

Caso queira criar uma lista de números, basta converter os resultados do range() diretamente em uma lista com a função list(). Quando envolvemos list() em torno da função range(), chamamos isso de wrapper, a saída será uma lista de números.

No exemplo da seção anterior, simplesmente exibimos uma série de números. Podemos utilizar list() para converter o mesmo conjunto de números em uma lista:

```
numbers = list(range(1, 6))
print(numbers)
```

Vejamos o resultado:

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

Podemos também usar a função range() para informar ao Python que desconsidere números em um determinado intervalo. Se passarmos um terceiro argumento para o range(), o Python usará esse valor como tamanho de passo, também chamado de step ou intervalo

numérico, ao gerar os números.

Por exemplo, veja como listar os números pares entre 1 e 10:

even_numbers.py

```
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

Nesse exemplo, a função `range()` começa com o valor 2 e, em seguida, soma 2 a esse valor. A função adiciona 2 repetidamente até atingir ou ultrapassar o valor final, 11, gerando o seguinte resultado:

```
[2, 4, 6, 8, 10]
```

É possível criar basicamente qualquer conjunto de números que quisermos usando a função `range()`. Por exemplo, como você criaria uma lista com os dez primeiros números quadrados (ou seja, o quadrado de cada número inteiro de 1 a 10). No Python, dois asteriscos (`**`) representam a operação de exponenciação ou potenciação. Vejamos como você pode inserir os dez primeiros números quadrados em uma lista:

square_numbers.py

```
squares = []
for value in range(1, 11):
    square = value ** 2
    squares.append(square)

print(squares)
```

Começamos com uma lista vazia chamada `squares`. Em seguida, solicitamos ao Python que percorra cada valor de 1 a 10 usando a função `range()`. Dentro do loop, o valor atual é elevado à segunda potência, ou elevado ao quadrado, e atribuído à variável `square` 1. Cada novo valor de `square` é então anexado à lista `squares` 2. Por último, quando o loop termina de executar, a lista de quadrados é exibida:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Para escrever esse código de forma mais concisa, omita a variável temporária `square` e anexe cada novo valor diretamente à lista:

```
squares = []
for value in range(1,11):
    squares.append(value**2)

print(squares)
```

Essa linha faz o mesmo trabalho que as linhas dentro do loop `for` na listagem anterior. Cada valor no loop é elevado ao quadrado e imediatamente anexado à lista de quadrados.

Você pode recorrer a qualquer uma dessas abordagens quando estiver criando listas mais complexas. Às vezes, usar uma variável temporária facilita a leitura do seu código; outras vezes, aumenta extensiva e desnecessariamente o código. Foque primeiro em escrever um código que você claramente entenda e execute o que você quer. Em seguida, procure abordagens mais eficientes conforme revisar seu código.

Estatísticas simples com uma lista de números

Algumas funções do Python são vantajosas ao trabalhar com listas de números. Por exemplo, podemos encontrar facilmente o mínimo, máximo, e a soma de uma lista de números:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

NOTA *Nesta seção, os exemplos usam listas pequenas de números que cabem facilmente na página. Esses exemplos executariam perfeitamente se sua lista contivesse um milhão ou mais números.*

List comprehensions

A abordagem anterior para gerar a lista `square` consistia em usar três ou quatro linhas de código. Uma *list comprehension* possibilita gerar

essa mesma lista com somente uma linha de código. Uma list comprehension combina o loop `for` e a criação de elementos novos em uma linha e anexa automaticamente cada elemento novo. Nem sempre as lists comprehensions são apresentadas aos iniciantes, mas resolvi inclui-las porque você provavelmente as verá assim que começar a examinar o código de outras pessoas.

O próximo exemplo cria a mesma lista de números quadrados que vimos anteriormente, mas com uma list comprehension:

squares.py

```
squares = [value**2 for value in range(1, 11)]
print(squares)
```

Para usar essa sintaxe, comece com um nome descritivo para a lista, como `squares`. Em seguida, abra um conjunto de colchetes e defina a expressão para os valores que quer armazenar na lista nova. Neste exemplo, a expressão é `value**2`, que eleva o valor ao quadrado. Depois, escreva um loop `for` para gerar os números desejados que quer inserir na expressão e feche os colchetes. Neste exemplo, o loop `for` é `for value in range(1, 11)`, que fornece os valores de 1 a 10 na expressão `value**2`. Repare que não temos dois-pontos no final da instrução `for`.

O resultado é a mesma lista de números quadrados que vimos antes:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

É necessário prática para escrever as próprias lists comprehensions, mas com o tempo, assim que estiver familiarizado com as listas comuns, você perceberá que vale a pena. Quando escrever três ou quatro linhas de código para gerar listas começar a se tornar repetitivo, pense em escrever as próprias list comprehensions.

FAÇA VOCÊ MESMO

4.3 Contando até vinte: Use um loop for para exibir os números de 1 a 20, todos juntos.

4.4 Um milhão: Crie uma lista com números de um a um milhão e, em seguida, utilize

um loop for para exibi-los. (Se a saída estiver demorando muito, interrompa-a pressionando CTRL+C ou fechando a janela de saída.)

4.5 Somando um milhão: crie uma lista com números de um a um milhão e, em seguida, use min() e max() a fim de garantir que sua lista realmente comece em um e termine em um milhão. Além disso, use a função sum() para ver a rapidez com que o Python pode efetuar a soma de um milhão de números.

4.6 Números ímpares: Use o terceiro argumento da função range() para criar uma lista com números ímpares de 1 a 20. Use o loop for para exibir cada número.

4.7 Três: Crie uma lista dos múltiplos de 3, de 3 a 30. Use um loop for para exibir os números em sua lista.

4.8 Cubos: Um número elevado à terceira potência é chamado de *cubo*. Por exemplo, no Python, o cubo de 2 é escrito como 2**3. Escreva uma lista dos primeiros 10 cubos (ou seja, o cubo de cada número inteiro de 1 a 10) e use um loop for para exibir o valor de cada cubo.

4.9 Cube Comprehension: Use uma list comprehension para gerar uma lista dos primeiros 10 cubos.

Trabalhando com parte de uma lista

No Capítulo 3, aprendemos como acessar elementos únicos em uma lista e, neste capítulo, aprendemos como trabalhar com todos os elementos em uma lista. No Python, podemos trabalhar também com um grupo específico de elementos em uma lista, chamado de *fatia* (slicing).

Fatiando uma lista

Para criar uma fatia, especifique o índice do primeiro e do último elemento com o qual quer trabalhar. Assim como a função `range()`, o Python para um elemento antes do segundo índice especificado. Para gerar como saída os três primeiros elementos em uma lista, acesse os índices de `0` a `3`, que retornam os elementos `0`, `1` e `2`.

O próximo exemplo é uma lista de jogadores de um time:

players.py

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[0:3])
```

O código exibe uma fatia da lista. A saída retém a estrutura da lista e inclui os três primeiros jogadores da lista:

```
['charles', 'martina', 'michael']
```

É possível gerar qualquer subconjunto de uma lista. Por exemplo, se quiser o segundo, terceiro e quarto elementos em uma lista, comece a fatia no índice `1` e termine no índice `4`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

Desta vez, a fatia começa com `'martina'` e termina com `'florence'`:

```
['martina', 'michael', 'florence']
```

Caso omita o primeiro índice em uma fatia, o Python iniciará automaticamente sua fatia no início da lista:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

Sem um primeiro índice, o Python começa no início da lista:

```
['charles', 'martina', 'michael', 'florence']
```

É possível usar uma sintaxe parecida se quisermos uma fatia com o final de uma lista. Por exemplo, caso queira todos os elementos do terceiro ao último, você pode começar com o índice `2` e omitir o segundo índice:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

O Python retorna todos os elementos do terceiro até o final da lista:

```
['michael', 'florence', 'eli']
```

Essa sintaxe possibilita gerar como saída todos os elementos de qualquer ponto da lista até o final, independentemente do tamanho da lista. Lembre-se de que um índice negativo retorna um elemento a uma certa distância do final de uma lista; desse modo, podemos gerar como saída qualquer fatia do final de uma lista. Por exemplo, se quisermos mostrar os últimos três jogadores da lista, podemos usar a fatia `players [-3:]`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

O código exibe os nomes dos últimos três jogadores e continuará a funcionar conforme a lista de jogadores muda de tamanho.

NOTA *É possível incluir um terceiro valor entre colchetes indicando uma fatia. Se incluído, o terceiro valor informa ao Python quantos elementos desconsiderar entre os elementos no intervalo especificado.*

Percorrendo uma fatia com um loop

Podemos utilizar uma fatia em um loop `for` se quisermos percorrer um subconjunto dos elementos em uma lista. No exemplo a seguir, percorremos os três primeiros jogadores e exibimos seus nomes como parte de uma equipe simples:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print("Here are the first three players on my team:")
for player in players[:3]:
    print(player.title())
```

Em vez de percorrer toda a lista de jogadores com o loop, o Python percorre somente os três primeiros nomes `1`:

```
Here are the first three players on my team:
Charles
Martina
Michael
```

Fatias são de grande ajuda em diversas situações. Por exemplo, ao criar um jogo, é possível adicionar a escore final de um jogador a uma lista sempre que esse jogador terminar de jogar. Assim, poderíamos obter os três melhores escores de um jogador ordenando a lista em ordem decrescente, obtendo uma fatia com apenas os três primeiros escores. Ao trabalhar com dados, você pode usar fatias para processar seus dados em partes com tamanho específico. Ou, ao desenvolver uma aplicação web, você pode usar fatias para exibir informações em uma série de páginas contendo a quantidade necessária de informações em cada página.

Copiando uma lista

Não raro, você quer começar com uma lista existente, criado uma lista completamente nova com base na primeira. Vamos explorar

como funciona a cópia de uma lista e examinar uma situação em que copiar uma lista é útil.

Para copiar uma lista, podemos criar uma fatia que inclua toda a lista original, omitindo o primeiro e o segundo índice (`[:]`). Isso informa ao Python para criar uma fatia que comece no primeiro elemento e termine no último elemento, gerando uma cópia de toda a lista.

Por exemplo, imagine que temos uma lista de nossas comidas favoritas e queremos criar uma lista separada das comidas que um amigo gosta. Como até agora nosso amigo gostou de tudo na lista, podemos criar a lista dele copiando a nossa:

foods.py

```
  my_foods = ['pizza', 'falafel', 'carrot cake']
1 friend_foods = my_foods[:]

  print("My favorite foods are:")
  print(my_foods)

  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

Primeiro, criamos uma lista das comidas que gostamos chamada `my_foods`. Em seguida, criamos uma nova lista chamada `friend_foods`. Fazemos uma cópia de `my_foods` criando uma fatia de `my_foods` sem especificar nenhum índice `1`, e atribuímos a cópia a `friend_foods`. Ao exibir cada lista, vemos que ambas contêm as mesmas comidas.

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']
```

A fim de comprovar de que realmente temos duas listas separadas, adicionaremos uma nova comida a cada lista e mostraremos que cada lista registra as comidas favoritas de cada pessoa:

```
  my_foods = ['pizza', 'falafel', 'carrot cake']
1 friend_foods = my_foods[:]
```

```
2 my_foods.append('cannoli')
3 friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

Copiamos os elementos originais em `my_foods` para a lista nova `friend_foods`, como no exemplo anterior `1`. Em seguida, adicionamos uma nova comida a cada lista: adicionamos `'cannoli'` a `my_foods` `2` e `'ice cream'` a `friend_foods` `3`. Depois, exibimos as duas listas para conferir se cada uma dessas comidas consta na lista apropriada:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'ice cream']
```

A saída mostra que agora `'cannoli'` aparece em nossa lista de comidas favoritas, mas `'ice cream'` não. Podemos ver que `'ice cream'` agora aparece na lista de nosso amigo, mas `'cannoli'` não. Se tivéssemos simplesmente definido `friend_foods` igual a `my_foods`, não teríamos gerado a saída de duas listas separadas. Por exemplo, vejamos o que acontece quando tentamos copiar uma lista sem usar uma fatia:

```
my_foods = ['pizza', 'falafel', 'carrot cake']

# Isso não funciona:
friend_foods = my_foods

my_foods.append('cannoli')
friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

Em vez de atribuir uma cópia de `my_foods` a `friend_foods`, definimos `friend_foods` igual a `my_foods`. Na realidade, essa sintaxe informa ao

Python para associar a nova variável friend_foods à lista que já está associada a my_foods, assim ambas as variáveis apontam para a mesma lista. Como resultado, quando adicionamos 'cannoli' a my_foods, 'cannoli' também aparecerá em friend_foods. Da mesma forma, 'ice cream' aparecerá em ambas as listas, mesmo que possa parecer que foi adicionado somente a friend_foods.

Mesmo a saída comprovando que ambas as listas são as mesmas agora, não queríamos esse resultado:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

NOTA *Por ora, não se preocupe com detalhes deste exemplo. Caso esteja tentando trabalhar com uma cópia de uma lista e observa um comportamento inesperado, verifique se está copiando a lista com uma fatia, como no primeiro exemplo.*

FAÇA VOCÊ MESMO

4.10 Fatias: Use um dos programas que escreveu neste capítulo, adicione diversas linhas ao final do programa para executarem o seguinte:

- Exiba a mensagem: *Os três primeiros elementos da lista são:*. Em seguida, use uma fatia para exibir os três primeiros elementos da lista desse programa.
- Exiba a mensagem: *O três elementos que ficam no meio da lista são:*. Depois, use uma fatia para exibir os três elementos do meio da lista.
- Exiba a mensagem: *Os três últimos elementos da lista são:* Em seguida, utilize uma fatia para exibir os três últimos elementos da lista.

4.11 Minhas pizzas, suas pizzas: Comece com o programa do Exercício 4.1 (página *90*). Faça uma cópia da lista de pizzas e a nomeie como friend_pizzas. Em seguida, siga as etapas:

- Adicione uma pizza nova à lista original.
- Adicione uma pizza diferente à lista friend_pizzas.
- Prove que tem duas listas separadas. Exiba a mensagem: *Minhas pizzas favoritas são:*. E, em seguida, use um loop for para exibir a primeira lista. Exiba a mensagem: *Minhas pizzas favoritas são:*. E, em seguida, use um loop for para exibir a segunda lista. Garanta que cada pizza nova seja armazenada na lista adequada.

4.12 Mais loops: Nesta seção, todas as versões de food.py evitaram o uso de loops for

para exibição, a fim de economizar espaço. Escolha uma versão de food.py e escreva dois loops for para exibir cada lista de alimentos.

Tuplas

As listas funcionam perfeitamente para armazenar coleções de elementos que podem mudar ao longo da vida de um programa. A capacidade de modificar listas é indispensável quando estamos trabalhando com uma lista de usuários em um site ou uma lista de personagens em um jogo. No entanto, às vezes queremos criar uma lista de elementos que não podem ser modificados. Com as tuplas podemos fazer exatamente isso. O Python trata valores que não podem mudar como *imutáveis*, e uma lista imutável se chama *tupla*.

Definindo uma tupla

Uma tupla se parece com uma lista, exceto que usamos parênteses em vez de colchetes. Após definir uma tupla, é possível acessar elementos individuais usando o índice de cada um deles, exatamente como faríamos em uma lista.

Por exemplo, se tivermos um retângulo que sempre deve ter determinado tamanho, podemos assegurar que seu tamanho não mude inserindo as dimensões em uma tupla:

dimensions.py

```
dimensions = (200, 50)
print(dimensions[0])
print(dimensions[1])
```

Definimos a tupla `dimensions`, usando parênteses em vez de colchetes. Em seguida, exibimos cada elemento na tupla individualmente, utilizando a mesma sintaxe que usamos para acessar elementos em uma lista:

```
200
50
```

Vejamos o que acontece se tentarmos alterar um dos elementos da tupla `dimensions`:

```
dimensions = (200, 50)
dimensions[0] = 250
```

Esse código tenta alterar o valor da primeira dimensão, mas o Python retorna um erro de tipo. Como estamos tentando modificar uma tupla, ação impossível com esse tipo de objeto, o Python nos informa que não podemos atribuir um valor novo a um elemento em uma tupla:

```
Traceback (most recent call last):
  File "dimensions.py", line 2, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

Isso é bom porque queremos que o Python gere um erro quando uma linha de código tentar alterar as dimensões do retângulo.

NOTA *Em termos técnicos, as tuplas são definidas pela presença de uma vírgula; os parênteses proporcionam mais clareza e legibilidade. Caso queira definir uma tupla com um elemento, basta incluir uma vírgula à direita:*

```
my_t = (3,)
```

Em geral, não faz sentido criar uma tupla com apenas um elemento, mas isso pode ocorrer quando as tuplas são geradas automaticamente.

Percorrendo todos os valores em uma tupla com um loop

Podemos percorrer todos os valores em uma tupla com um loop `for`, justamente como fizemos com uma lista:

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

O Python retorna todos os elementos na tupla, assim como retornaria em uma lista:

```
200
50
```

Sobrescrevendo uma tupla

Mesmo sendo impossível modificar uma tupla, podemos atribuir um valor novo a uma variável que represente uma tupla. Por exemplo, se quiséssemos alterar as dimensões deste retângulo, poderíamos redefinir toda a tupla:

```
dimensions = (200, 50)
print("Original dimensions:")
for dimension in dimensions:
    print(dimension)

dimensions = (400, 100)
print("\nModified dimensions:")
for dimension in dimensions:
    print(dimension)
```

As primeiras quatro linhas definem a tupla original e exibem as dimensões iniciais. Depois, associamos uma tupla nova à variável `dimensions` e exibimos os valores novos. Desta vez, o Python não gera nenhum erro, pois a reatribuição de uma variável é válida:

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

Quando comparadas às listas, as tuplas são estruturas de dados simples. Use as tuplas quando quiser armazenar um conjunto de valores que não devem ser alterados ao longo da vida de um programa.

FAÇA VOCÊ MESMO

4.13 Buffet: Um restaurante com serviço de buffet oferece somente cinco refeições básicas. Pense em cinco refeições simples e armazene-as em uma tupla.

- Use um loop for para exibir cada refeição que o restaurante oferece.
- Tente modificar um dos elementos e verifique se o Python rejeita a mudança.
- O restaurante muda seu cardápio, substituindo dois dos elementos por refeições diferentes. Adicione uma linha que reescreva a tupla e, depois, use um loop for para

exibir cada um dos elementos no menu reformulado.

Estilizando seu código

Agora que você está escrevendo programas mais extensos, é uma boa ideia aprender como estilizar seu código de forma consistente. Reserve um tempinho para que seu código fique mais fácil de ler. Escrever um código fácil de ler o ajuda a acompanhar o que seus programas estão fazendo e ajuda também outras pessoas a entenderem seu código.

Os programadores Python adotam uma série de convenções a fim de garantir que o código de todo mundo seja estruturado da mesma forma. Após aprender a escrever código Python limpo, você será capaz de entender a estrutura geral do código Python de qualquer outra pessoa, desde que adote as mesmas diretrizes. Caso pretenda se tornar um programador profissional um dia, comece a adotar agora e o mais rápido possível essas diretrizes para que, assim, desenvolva bons hábitos de programação.

Guia de estilo

Quando alguém quer realizar uma mudança na linguagem Python, é necessário recorrer a uma *Python Enhancement Proposal (PEP), Proposta de Aprimoramento do Python*. Uma das PEPS mais antigas é a *PEP 8*, que orienta os programadores sobre como estilizar seu código. Apesar de ser relativamente extensa, boa parte da PEP 8 diz respeito à estruturas de codificação mais complexas do que as que vimos até agora.

O guia de estilo Python foi elaborado com base no entendimento de que o código é lido com mais frequência do que escrito. Escrevemos o código uma vez e começamos a lê-lo ao depurá-lo. Ao adicionar uma funcionalidade a um programa, você passará mais tempo lendo seu código. Outros programadores também leem seu código quando você o compartilha.

Dada a escolha entre um código mais fácil de escrever ou um código mais fácil de ler, os programadores Python quase sempre irão incentivá-lo a escrever um código mais fácil de ler. As diretrizes a seguir o ajudarão a escrever um código claro desde o início.

Indentação

A PEP 8 nos recomenda quatro espaços por nível de indentação. O uso de quatro espaços aumenta a legibilidade, deixando espaço para diversos níveis de indentação em cada linha.

Em um documento de processamento de texto, como um documento word, as pessoas costumam usar tabulações em vez de espaços para o recuo. Apesar de isso funcionar perfeitamente com documentos de texto, o Python se confunde quando misturamos tabulações com espaços. Todo editor de texto fornece configuração que possibilita usar a tecla TAB, mas depois acaba convertendo cada tabulação em um número definido de espaços. Sem sombra de dúvidas, você deve usar a tecla TAB. No entanto, verifique a configuração do seu editor para inserir espaços em vez de tabulações em seu documento.

Misturar tabulações e espaços em seu arquivo pode ocasionar problemas dificílimos de identificar. Caso ache que tem uma mistura de tabulações e espaços, você pode converter as tabulações de um arquivo em espaços com a maioria dos editores.

Comprimento da linha

Muitos programadores Python recomendam que cada linha deve ter menos de 80 caracteres. Essa diretriz se deve ao fato de que, antigamente, a maioria dos computadores comportava apenas 79 caracteres em uma única linha de terminal. Hoje em dia, é possível inserir linhas mais extensas, porém há outras razões para se adotar o comprimento de linha padrão de 79 caracteres.

Em geral, os programadores profissionais abrem diversos arquivos

na mesma tela, assim o comprimento de linha padrão possibilita com que consigam enxergar todas as linhas de dois ou três arquivos abertos lado a lado na mesma tela. A PEP 8 recomenda também limitar todos os comentários a 72 caracteres por linha, já que algumas das ferramentas que geram documentação automática para projetos maiores adicionam caracteres de formatação no início de cada linha comentada.

As diretrizes do PEP 8 para comprimento de linha não são imutáveis como as tuplas, algumas equipes preferem um limite de 99 caracteres. Não se preocupe tanto com o comprimento das linhas em seu código enquanto estiver aprendendo, mas esteja ciente de que as pessoas que trabalham de forma colaborativa quase sempre seguem as diretrizes da PEP 8. A maioria dos editores possibilita que você configure um realce visual, normalmente uma linha vertical na tela, que mostra onde se estabelecem esses limites.

NOTA *O Apêndice B mostra como configurar seu editor de texto para que sempre insira quatro espaços sempre que você pressiona a tecla TAB, assim como uma linha-guia vertical para ajudá-lo com o limite de 79 caracteres.*

Linhas em branco

Para agrupar partes de seu programa visualmente, use linhas em branco. Devemos usar linhas em branco para organizar arquivos, mas não podemos usá-las em excesso. Seguindo os exemplos deste livro, você deve encontrar o ponto adequado de equilíbrio. Por exemplo, caso tenha cinco linhas de código que criam uma lista e outras três que fazem alguma coisa com essa lista, é apropriado inserir uma linha em branco entre essas duas seções. No entanto, recomenda-se não inserir três ou quatro linhas em branco entre as duas seções.

Embora não influenciem a execução do código, as linhas em branco influenciam a legibilidade dele. O interpretador Python utiliza a

indentação horizontal para interpretar o significado do código, mas desconsidera o espaçamento vertical.

Outros guias de estilo

A PEP 8 apresenta diversas recomendações adicionais de estilo, mas boa parte dessas diretrizes se referem a programas mais complexos do que os que estamos escrevendo aqui. À medida que aprender estruturas Python mais complexas, compartilharei as partes expressivas das diretrizes do PEP 8.

FAÇA VOCÊ MESMO

4.14 PEP 8: Consulte o guia de estilo PEP 8 original em https://python.org/dev/peps/pep-0008. Você não vai usá-la muito agora, mas seria interessante dar uma olhada.

4.15 Revisão do código: Escolha três dos programas que escreveu neste capítulo e modifique cada um deles para atender às recomendações da PEP 8.

- Utilize quatro espaços para cada nível de indentação. Configure seu editor de texto para inserir quatro espaços sempre que você pressionar a tecla TAB, se ainda não tiver configurado (confira o Apêndice B para instruções sobre como fazer isso).
- Use menos de 80 caracteres em cada linha e configure seu editor para mostrar uma linha-guia vertical na posição do 80º caractere.
- Não use excessivamente linhas em branco em seus arquivos de programa.

Recapitulando

Neste capítulo, aprendemos como trabalhar eficientemente com os elementos de uma lista. Aprendemos como percorrer uma lista com um loop `for`, como o Python utiliza a indentação para estruturar um programa e como evitar alguns erros comuns de indentação. Você aprendeu a criar listas numéricas simples, bem como algumas operações que podem ser efetuadas em listas numéricas. Aprendeu também como fatiar uma lista para trabalhar com um subconjunto de elementos e como usar uma fatia para copiarmos devidamente uma lista. Estudamos também as tuplas, que fornecem grau de proteção a um conjunto de valores que não devem ser alterados, e como estilizar seu código cada vez mais complexo para facilitar a

leitura.

No Capítulo 5, aprendermos como responder adequadamente a diferentes condições com a instrução `if`. Veremos como organizar conjuntos relativamente complexos de testes condicionais para responder adequada e precisamente ao tipo de situação ou informação desejadas. Aprenderemos também a usar as instruções `if` percorrendo uma lista para executar ações específicas com elementos selecionados dessa mesma lista.

CAPÍTULO 5
Instruções if

A programação geralmente implica examinar um conjunto de condições e decidir como proceder com base nessas condições. A instrução `if` do Python possibilita avaliar o estado atual de um programa e responder apropriadamente a esse estado.

Neste capítulo, aprenderemos a escrever testes condicionais, que viabilizam analisar qualquer condição de interesse. Estudaremos como escrever simples instruções `if` e como criar uma série mais complexa de instruções `if` visando identificar se as condições exatas que você quer são especificadas. Em seguida, aplicaremos esse conceito às listas, assim seremos capazes de escrever um loop `for` que processa a maioria dos elementos em uma lista de uma forma e determinados elementos com valores específicos de outra.

Um simples exemplo

O exemplo a seguir mostra como os testes `if` possibilitam responder devidamente à situações especiais. Imagine que você tem uma lista de carros e quer exibir o nome de cada carro. Como os carros são nomes próprios, a maioria dos nomes deve ser exibida com a primeira letra em maiúsculo. Apesar disso, o valor `'bmw'` deve ser exibido em letras maiúsculas. O código a seguir percorre uma lista de nomes de carros com um loop `for` e procura o valor `'bmw'`. Sempre que for `'bmw'`, o valor será exibido com todas as letras maiúsculas em vez de ser exibido com a primeira letra maiúscula:

cars.py

```
cars = ['audi', 'bmw', 'subaru', 'toyota']

for car in cars:
    if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

Neste exemplo, o loop verifica primeiro se o valor atual de `car` é `'bmw'` 1. Se for, o valor é exibido em letras maiúsculas. Se `car` for diferente de `'bmw'`, o valor será exibido com a primeira letra maiúscula:

```
Audi
BMW
Subaru
Toyota
```

Esse exemplo combina uma série de conceitos que aprenderemos neste capítulo. Começaremos nossos estudos analisando os tipos de testes que podemos usar para examinar as condições de um programa.

Testes condicionais

No cerne de cada instrução `if` reside uma expressão que pode ser avaliada como `True` ou `False`, chamada de *teste condicional*. O Python utiliza os valores `True` e `False` a fim de decidir se o código em uma instrução `if` deve ser executado. Se um teste condicional for avaliado como `True`, o Python executará o código após a instrução `if`. Se o teste for avaliado como `False`, o Python desconsiderará o código após a instrução `if`.

Verificando a igualdade

Boa parte dos testes condicionais compara o valor atual de uma variável com um valor específico de interesse. O teste condicional mais simples analisa se o valor de uma variável é igual ao valor de interesse:

```
>>> car = 'bmw'
>>> car == 'bmw'
True
```

A primeira linha define o valor de `car` como `'bmw'` recorrendo a um único sinal de igual, como já vimos tantas vezes. A próxima linha verifica se o valor de `car` é `'bmw'`, recorrendo a um sinal de igual duplo (`==`). Esse *operador de igualdade* retorna `True` se os valores à esquerda e à direita do operador forem correspondentes e `False` se não forem. Neste exemplo, os valores correspondem, logo o Python retorna `True`.

Se o valor de `car` fosse diferente de `'bmw'`, o teste retornaria `False`:

```
>>> car = 'audi'
>>> car == 'bmw'
False
```

Um único sinal de igual equivale a uma declaração. Podemos ler essa primeira linha de código assim: "O valor atribuído a `car` é igual a `'audi'`". Em contrapartida, um duplo sinal de igual equivale a uma pergunta: "O valor de `car` é igual a `'bmw'`?" A maioria das linguagens de programação utiliza sinais de igual dessa forma.

Ignorando letras maiúsculas e minúsculas ao verificar a igualdade

No Python, o teste de igualdade diferencia as letras maiúsculas de minúsculas. Por exemplo, dois valores com capitalizações diferentes não são considerados iguais:

```
>>> car = 'Audi'
>>> car == 'audi'
False
```

Caso seja necessário diferenciar letras minúsculas e maiúsculas, esse comportamento é positivo. Caso não seja, e você queira apenas testar o valor de um variável, podemos converter o valor da variável em letras minúsculas antes de compará-lo:

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Esse teste retornará True, independentemente da formatação do valor de 'Audi' porque agora o teste não diferencia letras maiúsculas de minúsculas. O método lower() não altera o valor originalmente armazenado em car, assim podemos fazer esse tipo de comparação sem afetar a variável:

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
>>> car
'Audi'
```

De início, atribuímos a string com a primeira letra maiúscula "Audi" à variável car. Em seguida, convertemos o valor de car em letras minúsculas e o comparamos à string 'audi'. As duas strings correspondem, assim o Python retorna True. Podemos ver que o valor armazenado em car não foi afetado pelo método lower().

De forma semelhante, os sites colocam em prática determinadas regras em relação aos dados fornecidos pelos usuários. Por exemplo, um site pode realizar um teste condicional como esse a fim de garantir que cada usuário tenha um nome de usuário verdadeiramente exclusivo, não somente para autenticar a variação de letras maiúsculas e minúsculas do nome de usuário de outra pessoa. Quando alguém registra um nome novo de usuário, esse nome é convertido em letras minúsculas e comparado às variantes de letras minúsculas de todos os nomes de usuário existentes. Durante essa verificação, um nome de usuário como 'John' será rejeitado se qualquer variação de 'john' já estiver em uso.

Verificando a diferença

Quando queremos determinar se dois valores não são iguais, podemos usar o *operador de diferença* (!=). Vejamos outra instrução if para examinar como usar o operador de diferença. Armazenaremos um ingrediente do pedido de uma pizza em uma variável e exibiremos uma mensagem, caso o pedido não contenha anchovas:

toppings.py

```
requested_topping = 'mushrooms'

if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```

Esse código compara o valor de `request_topping` com o valor `'anchovies'`. Se esses dois valores não corresponderem, o Python retornará `True` e executará o código após a instrução `if`. Se os dois valores corresponderem, o Python retornará `False` e não executará o código após a instrução `if`.

Como o valor de `request_topping` não é `'anchovies'`, a função `print()` é executada:

```
Hold the anchovies!
```

A maioria das expressões condicionais testará a igualdade, mas, às vezes, você achará melhor testar a diferença.

Comparações numéricas

Testar valores numéricos é bastante simples. Por exemplo, o código seguinte verifica se uma pessoa tem 18 anos:

```
>>> age = 18
>>> age == 18
True
```

Podemos testar também se dois números não são iguais. Por exemplo, o código a seguir exibe uma mensagem se a resposta fornecida não estiver correta:

magic_number.py

```
answer = 17
if answer != 42:
    print("That is not the correct answer. Please try again!")
```

O teste condicional passa, já o valor de `answer` (`17`) não é igual a `42`. Como o teste foi aprovado, o bloco de código indentado é executado:

```
That is not the correct answer. Please try again!
```

É possível também incluir diferentes comparações matemáticas sem suas declarações condicionais, como menor que, menor ou igual a, maior que e maior ou igual a:

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Cada comparação matemática pode ser usada como parte de uma instrução `if`, que pode ajudá-lo a detectar as condições exatas de interesse.

Verificando múltiplas condições

Talvez você queira verificar múltiplas condições ao mesmo tempo. Por exemplo, às vezes você precisa que duas condições sejam `True` para executar uma ação. Outras vezes, você fica satisfeito com apenas uma condição sendo `True`. As palavras reservadas `and` e `or` podem ajudar nessas situações.

Usando and para verificar múltiplas condições

Para verificar se duas condições são concomitantemente `True`, use a palavra reservada `and` para combinar os dois testes condicionais; se cada teste passar, a expressão geral será avaliada como `True`. Se um dos testes falhar ou ambos os testes falharem, a expressão será avaliada como `False`.

Por exemplo, podemos verificar se duas pessoas têm mais de 21 anos com o seguinte teste:

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 22
```

```
>>> age_0 >= 21 and age_1 >= 21
True
```

Primeiro, definimos duas idades, `age_0` e `age_1`. Em seguida, verificamos se ambas as idades têm 21 anos ou mais `1`. O teste à esquerda passa, mas o teste à direita falha. Logo, a expressão condicional geral é avaliada como `False`. Em seguida, alteramos `age_1` para 22 `2`. Agora, o valor de `age_1` é maior que 21, então ambos os testes individuais são aprovados, fazendo com que a expressão condicional geral seja avaliada como `True`.

Para melhorar a legibilidade, podemos inserir parênteses em torno dos testes individuais, mesmo não sendo obrigatórios. Caso insira os parênteses, seu teste ficará assim:

```
(age_0 >= 21) and (age_1 >= 21)
```

Usando or para verificar múltiplas condições

A palavra reservada `or` também possibilita que verifiquemos múltiplas condições, mas com um detalhe: o teste passa quando um ou ambos os testes individuais passam. Uma expressão `or` falha apenas quando ambos os testes individuais falham.

Vamos usar o exemplo das duas idades novamente, mas, desta vez, queremos apenas uma pessoa acima de 21 anos:

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

Começamos com duas variáveis de idade mais uma vez. Como o teste de `age_0` `1` passa, a expressão geral é avaliada como `True`. Em seguida, reduzimos `age_0` para 18. No teste final `2`, ambos os testes falham e a expressão geral é avaliada como `False`.

Verificando se um valor está em uma lista

Às vezes, é necessário verificar se uma lista contém um determinado valor antes de executar uma ação. Por exemplo, vamos supor que queremos verificar se um novo nome de usuário já existe em uma lista de nomes de usuário atuais antes de concluirmos o cadastro de alguém em um site. Em um projeto de mapeamento, talvez você queira verificar se um local registrado já existe em uma lista de locais conhecidos.

Para averiguar se um determinado valor já está em uma lista, use a palavra reservada `in`. Usaremos como exemplo um código que você poderia escrever para uma pizzaria. Vamos criar uma lista de ingredientes para um cliente que pediu uma pizza e, depois, vamos verificar se determinados ingredientes constam na lista.

```
>>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
>>> 'mushrooms' in requested_toppings
True
>>> 'pepperoni' in requested_toppings
False
```

A palavra reservada `in` solicita ao Python para verificar a existência de `'mushrooms'` e `'pepperoni'` na lista `request_toppings`. Essa técnica é muito poderosa porque podemos criar uma lista de valores importantes e verificar facilmente se o valor que estamos testando corresponde a um dos valores da lista.

Verificando se um valor não está em uma lista

Outras vezes, é necessário saber se um valor não consta em uma lista. Nesse caso, podemos usar a palavra reservada `not`. Por exemplo, considere uma lista de usuários proibidos de comentar em um fórum. Podemos verificar se um usuário foi banido antes de permitir que essa pessoa envie um comentário:

banned_users.py

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'
```

```
if user not in banned_users:
    print(f"{user.title()}, you can post a response if you wish.")
```

Aqui, a instrução `if` é visivelmente clara. Se o valor de `user` não constar na lista `banned_users`, o Python retornará `True` e executará a linha indentada.

A usuária `'marie'` não consta na lista `banned_users`, assim ela vê uma mensagem que a convida a postar uma resposta:

```
Marie, you can post a response if you wish.
```

Expressões booleanas

Em algum momento, à medida que aprende mais sobre programação, você vai se deparar com o termo *expressão booleana*. Expressão booleana é somente outra palavra para teste condicional. Um *valor booleano* é `True` ou `False`, assim como o valor de uma expressão condicional depois de avaliada.

Os valores booleanos costumam ser usados para acompanhar determinadas condições, como se um jogo estiver sendo executado ou se um usuário tem permissão para editar determinado conteúdo em um site:

```
game_active = True
can_edit = False
```

Os valores booleanos fornecem um modo eficaz para rastrear o estado de um programa ou uma condição específica e importante em seu programa.

FAÇA VOCÊ MESMO

5.1 Testes condicionais: Escreva uma série de testes condicionais. Exiba uma afirmação com cada teste descrito e a previsão dos resultados de cada teste. Seu código deve ficar mais ou menos assim:

```
car = 'subaru'
print("Is car == 'subaru'? I predict True.")
print(car == 'subaru')

print("\nIs car == 'audi'? I predict False.")
```

```
print(car == 'audi')
```

- Preste bastante atenção aos seus resultados e procure entender por que cada linha é avaliada como True ou False.
- Crie, pelo menos, 10 testes. Execute, pelo menos, 5 testes avaliados como True e outros 5 testes avaliados como False.

5.2 Mais testes condicionais: não é necessário restringir o número de testes a 10. Caso queira executar mais comparações, escreva mais testes e os adicione a *conditional_tests.py*. Gere, pelo menos, um resultado True e um False para cada condição a seguir:

- Testes com operadores de igualdade e de diferença com strings.
- Testes usando o método lower().
- Testes numéricos com operadores de igualdade e de diferença, maior e menor que, maior ou igual que e menor ou igual a.
- Testes com as palavras reservadas and e or.
- Testes para averiguar se um valor consta em uma lista.
- Testes para averiguar se um valor não consta em uma lista.

Instruções if

Assim que entender os testes condicionais, você será capaz de escrever instruções if. Por mais que existam diversos tipos diferentes de instruções if, escolher qual usar depende do número de condições que precisamos testar. Já vimos e discutimos alguns exemplos de instruções if em testes condicionais, mas, agora, vamos nos aprofundar no assunto.

Instruções if simples

O tipo mais simples de instrução if tem um teste e uma ação:

```
if teste_conditional:
    faça alguma coisa
```

É possível inserir qualquer teste condicional na primeira linha, e basicamente qualquer ação no bloco indentado após o teste. Se o teste condicional for avaliado como True, o Python executará o código após a instrução if. Se o teste for avaliado como False, o Python ignorará o código após a instrução if.

Digamos que temos uma variável representando a idade de uma pessoa e queremos saber se essa pessoa tem idade para votar. O código a seguir testa se a pessoa pode votar:

voting.py

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
```

O Python verifica se o valor de `age` é maior ou igual a 18. Como é maior, o Python executa o `print()` indentado:

```
You are old enough to vote!
```

A indentação desempenha o mesmo papel nas instruções `if` e nos loops `for`. Todas as linhas indentadas depois de uma instrução `if` serão executadas se o teste for aprovado, e todo o bloco de linhas indentadas será ignorado se o teste não for aprovado.

Podemos inserir quantas linhas de código quisermos no bloco seguinte à instrução `if`. Vamos adicionar outra linha de saída se a pessoa tiver idade suficiente para votar, perguntando se já se registrou para votar[1]:

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

O teste condicional passa e ambas instruções `prints()` são indentadas e as linhas são exibidas:

```
You are old enough to vote!
Have you registered to vote yet?
```

Se o valor de `age` for menor que 18, o programa não gerará nenhuma saída.

Instruções if-else

Queremos executar uma ação quando um teste condicional passa, mas não é sempre que queremos executar a mesma ação. Muitas vezes, em outros casos, queremos executar uma ação diferente.

Com a sintaxe if-else do Python isso é possível. Um bloco if-else se parece com uma instrução if simples, exceto que a instrução else permite definir uma ação ou um conjunto de ações que são executadas quando o teste condicional falha.

Mostraremos a mesma mensagem exibida anteriormente se a pessoa tiver idade suficiente para votar, mas, desta vez, adicionaremos uma mensagem para quem não tiver idade para votar:

```
  age = 17
1 if age >= 18:
      print("You are old enough to vote!")
      print("Have you registered to vote yet?")
2 else:
      print("Sorry, you are too young to vote.")
      print("Please register to vote as soon as you turn 18!")
```

Caso o teste condicional 1 passe, o primeiro bloco de print() indentado será executado. Caso o teste seja avaliado como False, o bloco else 2 é executado. Como age é menor que 18 anos, desta vez, o teste condicional falha e o código no bloco else é executado:

```
Sorry, you are too young to vote.
Please register to vote as soon as you turn 18!
```

Esse código executa porque temos somente duas situações possíveis para avaliar: uma pessoa tem idade para votar ou não. A estrutura if-else funciona bem em situações nas quais queremos que o Python sempre execute uma das duas ações possíveis. Em uma sequência if-else simples como essa, uma dessas duas ações sempre será executada

Sequência if-elif-else

Precisaremos com frequência testar mais de duas situações possíveis e, para avaliá-las, podemos usar a sintaxe if-elif-else do Python. O Python executa somente um bloco em uma sequência if-elif-else. E executa cada teste condicional consecutivamente, até que um passe. Quando um teste passa, o código seguinte a esse teste é executado

e o Python desconsidera o restante dos testes.

Muitas situações reais envolvem mais de duas condições possíveis. Por exemplo, imagine um parque de diversões que cobra preços diferentes de entrada para diferentes faixas etárias:

- A entrada para menores de 4 anos é gratuita.
- A entrada para qualquer pessoa entre 4 e 18 anos custa US$25.
- A entrada para maiores de 18 anos custa U$S40.

Como podemos usar uma instrução if para estipular o preço da entrada de uma pessoa? O código a seguir testa a faixa etária de uma pessoa e, em seguida, exibe uma mensagem com preço de entrada:

amusement_park.py

```
  age = 12
1 if age < 4:
      print("Your admission cost is $0.")
2 elif age < 18:
      print("Your admission cost is $25.")
3 else:
      print("Your admission cost is $40.")
```

O teste if 1 verifica se uma pessoa tem menos de 4 anos. Quando o teste passa, uma mensagem adequada é exibida, e o Python desconsidera o resto dos testes. A linha elif 2 é realmente outro teste if, executado apenas se o teste anterior falhar. Nesse ponto da sequência, sabemos que a pessoa tem, pelo menos, 4 anos porque o primeiro teste falhou. Se a pessoa for menor de 18 anos, uma mensagem adequada é exibida, e o Python desconsidera o bloco else. Se ambos os testes if e elif falharem, o Python executa o código no bloco else 3.

Neste exemplo, o teste if 1 é avaliado como False, então seu bloco de código não é executado. No entanto, o teste elif é avaliado como True (12 é menor que 18),então seu código é executado. A saída é uma frase, informando ao usuário o preço da entrada:

```
Your admission cost is $25.
```

Qualquer idade acima 17 anos faria com que os dois primeiros testes falhassem. Nessas situações, o bloco `else` seria executado e o preço da entrada seria de US$40.

Em vez de exibir o preço de entrada dentro do bloco `if-elif-else`, seria mais sucinto definir somente o preço dentro da sequência `if-elif-else` e, em seguida, inserir um único `print()` que execute após a avaliação da sequência.

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
else:
    price = 40

print(f"Your admission cost is ${price}.")
```

As linhas indentadas definem o valor de `price` de acordo com a idade da pessoa, como no exemplo anterior. Após o preço ser definido pela sequência `if-elif-else`, um `print()` separado e não indentado usa esse valor para exibir uma mensagem declarando o preço da entrada por pessoa.

Ainda que o código gere a mesma saída do exemplo anterior, o objetivo da sequência `if-elif-else` é mais limitado. Em vez de estabelecer um preço e exibir uma mensagem, a sequência simplesmente determina o preço do ingresso. Além de ser mais efetivo, esse código revisado é mais fácil de alterar do que a abordagem original. Para alterar o texto da mensagem de saída, teríamos somente que alterar um `print()` em vez de três deles separados.

Usando múltiplos blocos elif

Você pode usar quantos blocos `elif` quiser em seu código. Por exemplo, se o parque de diversões implementasse um desconto para idosos, poderíamos adicionar mais um teste condicional ao código

para determinarmos se alguém obtém desconto. Digamos que qualquer pessoa com 65 anos ou mais pague metade do preço usual da entrada, ou US$20:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
else:
    price = 20

print(f"Your admission cost is ${price}.")
```

A maior parte desse código está inalterada. Agora, o segundo bloco `elif` executa uma verificação para garantir se uma pessoa tem menos de 65 anos antes de lhe atribuir o preço da entrada inteira, de U$$40. Repare que o valor atribuído ao bloco `else` precisa ser alterado para U$$20, porque as únicas idades que passam por esse bloco são pessoas com 65 anos ou mais.

Omitindo o bloco else

O Python não requer um bloco `else` no final de uma sequência `if-elif`. Às vezes, um bloco `else` já é o suficiente. Outras vezes, é mais fácil usar uma instrução adicional `elif` que processe a condição específica de interesse:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
elif age >= 65:
    price = 20

print(f"Your admission cost is ${price}.")
```

O último bloco elif atribui um preço de US$20 para pessoas com 65 anos ou mais, e fica um pouco mais legível do que o bloco else geral. Com isso, cada bloco de código deve passar por um teste específico para ser executado.

O bloco else é uma instrução abrangente. Corresponde a qualquer condição não igualada por um teste específico if ou elif que, ocasionalmente, pode conter dados inválidos ou até maliciosos. Se estiver testando uma condição final específica, considere usar um bloco elif final e omita o bloco else. Como resultado, você ficará mais seguro de que seu código será executado apenas nas condições corretas.

Testando múltiplas condições

Apesar de poderosa, a sequência if-elif-else somente é necessária quando queremos que um teste passe. Assim que processa um teste que passa, o Python desconsidera os testes restantes. É um comportamento positivo, porque é eficiente e possibilita testar uma condição específica.

Ainda assim, é importante às vezes verificar todas as condições de interesse. Nesse caso, use uma série de instruções simples if sem blocos elif ou else. Essa técnica tem lógica quando mais de uma condição pode ser True, e queremos verificar todas as condições True.

Vamos retomar o exemplo da pizzaria. Se alguém pedir uma pizza com dois ingredientes, precisamos inclui-los na pizza:

topings.py

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
if 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
if 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")
```

```
print("\nFinished making your pizza!")
```

Começamos com uma lista contendo os ingredientes do pedido. A primeira instrução `if` verifica se a pessoa quer sua pizza com mushrooms (cogumelos). Se sim, exibe-se uma mensagem confirmando esse ingrediente. O teste para pepperoni `1` é outra instrução simples `if`, não uma instrução `elif` ou `else`, assim esse teste é executado independentemente de o teste anterior ter passado ou não. A última instrução `if` verifica se o pedido tem extra cheese (queijo extra), indiferente dos resultados dos dois primeiros testes. Esses três testes independentes são efetuados sempre que esse programa é executado.

Neste exemplo, como todas as condições são avaliadas, cogumelos e queijo extra são adicionados à pizza:

```
Adding mushrooms.
Adding extra cheese.

Finished making your pizza!
```

Esse código não executaria devidamente se usássemos um bloco `if-elif-else`, porque o código pararia sua execução após somente um teste passasse. Vejamos como ficaria esse código:

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

O teste em `'mushrooms'` é o primeiro a passar. Ou seja, cogumelos são adicionados à pizza. Contudo, os valores `'extra cheese'` e `'pepperoni'` nunca são verificados, porque o Python não executa nenhum teste além do primeiro teste que passa em uma sequência `if-elif-else`. O primeiro ingrediente da pizza pedida será adicionado, mas todos os outros

não:

```
Adding mushrooms.

Finished making your pizza!
```

Em síntese, caso queira que apenas um bloco de código seja executado, use uma sequência if-elif-else. Se for necessário executar mais de um bloco de código, use uma série de instruções if independentes.

FAÇA VOCÊ MESMO

5.3 Cores de alienígenas #1: Imagine que um alienígena acabou de ser abatido em um jogo. Crie uma variável chamada alien_color e lhe atribua um valor 'green', 'yellow' ou 'red'.

- Escreva uma instrução if para testar se a cor do alienígena é verde. Se for, exiba uma mensagem informando que o jogador acabou de ganhar 5 pontos.
- Escreva uma versão desse programa que passe no teste if e outra que falhe. (A versão que falha não gerará saída.)

5.4 Cores de alienígenas #2: Escolha uma cor para um alienígena, como no Exercício 5.3, e escreva uma sequência if-else.

- Se a cor do alienígena for verde, exiba uma afirmação de que o jogador acabou de ganhar 5 pontos por abrir fogo contra um alienígena.
- Se a cor do alienígena não for verde, exiba uma afirmação de que o jogador acabou de ganhar 10 pontos.
- Escreva uma versão desse programa que execute o bloco if e outra que execute o bloco else.

5.5 Cores alienígenas #3: Converta sua sequência if-else do Exercício 5.4 em uma sequência if-elif-else.

- Se o alienígena for verde, exiba uma afirmação de que o jogador ganhou 5 pontos.
- Se o alienígena for amarelo, exiba uma afirmação de que o jogador ganhou 10 pontos.
- Se o alienígena for vermelho, exiba uma afirmação de que o jogador ganhou 15 pontos.
- Escreva três versões desse programa, assegurando que cada afirmação exibida seja correspondente à cor adequada do alienígena

5.6 Fases da vida: Escreva uma sequência if-elif-else que determine a fase da vida de uma pessoa. Defina um valor para a variável age, e depois:

- Se a pessoa tiver menos de 2 anos, exiba uma mensagem informando que a pessoa é um neném.

- Se a pessoa tiver pelo menos 2 anos, e menos de 4, exiba uma mensagem informando que é uma criança.
- Se a pessoa tiver pelo menos 4 anos, e menos de 13, exiba uma mensagem informando que é um(a) garoto(a).
- Se a pessoa tiver pelo menos 13 anos, e menos de 20, exiba uma mensagem informando que é adolescente.
- Se a pessoa tiver pelo menos 20 anos, e menos de 65, exiba uma mensagem informando que é um adulto.
- Se a pessoa tiver 65 anos ou mais, imprima uma mensagem informando que a pessoa é um(a) idoso(a).

5.7 Fruta favorita: Crie uma lista de suas frutas favoritas e, em seguida, escreva uma série de declarações if independentes que verificam determinadas frutas em sua lista.

- Crie uma lista com suas três frutas favoritas e a nomeie como favorite_fruits.
- Escreva cinco instruções if. Cada uma deve verificar se um determinado tipo de fruta consta em sua lista. Se sim, o bloco if deve exibir uma afirmação do tipo: *Você realmente gosta de bananas!*

Usando instruções if com listas

Podemos desenvolver alguns programas interessantes quando combinamos listas com instruções `if`. Podemos acompanhar valores especiais que precisam ser tratados de forma diferente de outros valores na lista. É possível lidar eficientemente com condições variáveis, como a disponibilidade de determinados itens em um restaurante durante um turno. Podemos também começar a provar que o código funciona como esperado em todas as situações possíveis.

Verificando elementos especiais

Neste capítulo, começamos com um simples exemplo que mostrava como lidar com um valor especial como `'bmw'`, que precisava ser exibido em um formato diferente dos outros valores da lista. Agora que você tem uma noção básica dos testes condicionais e das instruções `if`, vamos nos aprofundar em como podemos acompanhar valores especiais em uma lista e manipulá-los de forma adequada.

Vamos continuar com o exemplo da pizzaria. A pizzaria exibe uma mensagem sempre que um ingrediente é adicionado a uma pizza enquanto está sendo preparada. Podemos escrever o código que executa essa ação com eficiência criando uma lista de ingredientes solicitados pelo cliente, com um loop para informar cada ingrediente conforme adicionado à pizza:

toppings.py

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print(f"Adding {requested_topping}.")
print("\nFinished making your pizza!")
```

A saída é simples porque esse código usa um singelo loop `for`:

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.

Finished making your pizza!
```

Mas, e se a pizzaria ficar sem pimentão verde? Uma instrução `if` dentro do loop `for` pode lidar com essa situação de forma adequada:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
    else:
        print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

Desta vez, verificamos cada elemento solicitado antes de adicioná-lo à pizza. A instrução `if` verifica se a pessoa pediu a pizza com pimentão verde. Se pediu, exibimos uma mensagem informando o porquê não se pode pedir pimentão verde. O bloco `else` garante que todos os outros ingredientes sejam adicionados à pizza.

A saída mostra que cada ingrediente solicitado é manipulado de forma adequada.

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!
```

Verificando se uma lista não está vazia

Fizemos uma simples suposição sobre todas as listas com as quais trabalhamos até agora: supomos que cada lista tinha, pelo menos, um elemento. Daqui a pouco, vamos permitir que os usuários forneçam as informações armazenadas em uma lista. Ou seja, não podemos supor que uma lista contém algum elemento sempre que um loop for executado. Nesses casos, é bom verificar se uma lista está vazia antes de executar um loop `for`.

Para exemplificar isso, verificaremos se a lista com ingredientes solicitados está vazia antes de preparar a pizza. Se a lista estiver vazia, avisaremos ao usuário se ele quer uma pizza normal. Se a lista não estiver vazia, prepararemos a pizza da mesma forma que fizemos nos exemplos anteriores:

```
requested_toppings = []

if requested_toppings:
    for requested_topping in requested_toppings:
        print(f"Adding {requested_topping}.")
    print("\nFinished making your pizza!")
else:
    print("Are you sure you want a plain pizza?")
```

Agora, começamos com uma lista vazia dos ingredientes solicitados. Em vez de avançar direto para o loop `for`, primeiro fazemos uma verificação rápida. Quando o nome de uma lista é utilizado em uma instrução `if`, o Python retorna `True` se a lista contiver pelo menos um elemento; uma lista vazia é avaliada como `False`. Se `request_toppings` passar no teste condicional, executamos o mesmo loop `for` do exemplo anterior. Se o teste condicional falhar, exibimos uma mensagem perguntando ao cliente se realmente quer uma pizza normal, sem ingredientes adicionais.

Como nesse caso a lista está vazia, a saída pergunta se o usuário quer uma pizza normal.

```
Are you sure you want a plain pizza?
```

Se a lista não estiver vazia, a saída exibirá cada ingrediente solicitado adicionado à pizza.

Usando múltiplas listas

As pessoas pedirão praticamente qualquer coisa, ainda mais quando se trata de ingredientes de pizza. Mas, e se um cliente pedir uma pizza com batatas fritas? Podemos utilizar listas e instruções `if` para garantir que a entrada faça sentido antes de verificá-la.

Vamos ficar atentos a pedidos com ingredientes inusitados antes de preparar uma pizza. O exemplo seguinte define duas listas. A primeira é uma lista de ingredientes disponíveis na pizzaria, e a segunda é a lista de ingredientes solicitados pelo usuário. Desta vez, cada elemento em `request_toppings` é comparado à lista antes de ser adicionado à pizza:

```
available_toppings = ['mushrooms', 'olives', 'green peppers',
                      'pepperoni', 'pineapple', 'extra cheese']

1 requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

  for requested_topping in requested_toppings:
2     if requested_topping in available_toppings:
          print(f"Adding {requested_topping}.")
3     else:
          print(f"Sorry, we don't have {requested_topping}.")

  print("\nFinished making your pizza!")
```

Primeiro, definimos uma lista de ingredientes disponíveis na pizzaria. Repare que, se a pizzaria tivesse uma seleção estável de ingredientes, poderíamos usar uma tupla. Em seguida, criamos uma lista com os ingredientes solicitados por um cliente. Neste exemplo, temos um pedido inusitado de ingrediente: `french fries'` (batatas fritas) `1`. Depois, percorremos com um loop a lista dos ingredientes

solicitados. Dentro do loop, verificamos se cada ingrediente solicitado consta realmente na lista de ingredientes disponíveis 2. Se constar, adicionamos esse ingrediente à pizza. Se o ingrediente solicitado não constar na lista de ingredientes disponíveis, o bloco else será executado 3. O bloco else exibe uma mensagem informando ao usuário quais ingredientes não estão disponíveis.

A sintaxe desse código gera uma saída limpa e explicativa:

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

Com somente algumas linhas de código, lidamos com uma situação real com bastante eficiência!

FAÇA VOCÊ MESMO

5.8 Olá, admin: Crie uma lista com cinco ou mais nomes de usuários, incluindo o nome 'admin'. Imagine que está escrevendo um código que exibirá uma mensagem de boas-vindas aos usuários, após cada um deles logar em um site. Percorra a lista com um loop e exiba uma mensagem de boas-vindas para cada usuário.

- Se o nome de usuário for 'admin', exiba uma mensagem especial, tipo: *Olá administrador, gostaria de ver um relatório de status?*
- Caso contrário, exiba uma mensagem genérica, como: *Olá Jaden, obrigado por fazer login novamente.*

5.9 Sem usuários: adicione um teste if a hello_admin.py a fim de garantir que a lista de usuários não esteja vazia.

- Se a lista estiver vazia, exiba mensagem: *É necessário encontrar alguns usuários!*
- Remova todos os nomes de usuários de sua lista e verifique se a mensagem correta foi exibida.

5.10 Verificando nomes de usuários: faça o seguinte para criar um programa que simule como os sites garantem que todos tenham um nome de usuário exclusivo.

- Crie lista de cinco ou mais nomes de usuários chamada current_users.
- Crie outra lista com cinco nomes de usuários chamada new_users. Assegure-se de que um ou dois dos nomes novos de usuário também estejam na lista current_users.
- Percorra com um loop a lista new_users para verificar se cada nome novo de usuário já foi usado. Se sim, exiba uma mensagem informando que a pessoa precisará inserir um nome novo de usuário. Se um nome de usuário não foi usado, exiba uma mensagem informando que o nome de usuário está disponível.

- Garanta que sua comparação não diferencie letras maiúsculas de minúsculas. Se 'John' foi usado, 'JOHN' não deve ser aceito. (Para fazer isso, será necessário fazer uma cópia de current_users contendo as versões em minúsculas de todos os usuários existentes.)

5.11 Números ordinais: Os números ordinais designam sua posição em uma lista, como *1º ou 2º. Em inglês, 1st* ou *2nd*. A maioria dos números ordinais em inglês termina em *th*, exceto 1, 2 e 3.

- Armazene os números de 1 a 9 em uma lista.
- Percorra com um loop a lista.
- Use uma sequência if-elif-else dentro do loop para exibir a terminação ordinal adequada para cada número. Sua saída deve ler "1st 2nd 3rd 4th 5th 6th 7th 8th 9th", e cada resultado deve estar em uma linha separada.

Estilizando suas instruções if

Em todos os exemplos deste capítulo, vimos boas práticas se estilização. A única recomendação da PEP 8 para estilizar testes condicionais é usar um único espaço ao redor dos operadores de comparação, como `==`, `>=` e `<=`. Por exemplo:

```
if age < 4:
```

É melhor que:

```
if age<4:
```

Esse espaço não compromete a forma como o Python interpreta seu código; apenas faz com que seja mais fácil para você e outras pessoas lerem seu código.

FAÇA VOCÊ MESMO

5.12 Estilizando instruções if: Revise os programas que escreveu neste capítulo e verifique se os seus testes condicionais foram adequadamente estilizados.

5.13 Suas ideias: Nesse momento, você é um programador mais preparado do que era quando começou a ler este livro. Agora que já tem uma noção melhor de como situações reais são modeladas em programas, talvez você esteja pensando em alguns problemas que poderia solucionar com os próprios programas. Registre qualquer ideia nova que tiver sobre problemas que queira resolver conforme suas habilidades de programação melhoram. Pense nos jogos que pode desenvolver, nos conjuntos de dados que talvez queira explorar e nas aplicações web que gostaria de criar.

Recapitulando

Neste capítulo, aprendemos como escrever testes condicionais, que sempre são avaliados como True ou False. Aprendemos a escrever instruções simples if, sequências if-else e if-elif-else. Começamos a utilizar essas estruturas para identificar condições específicas que precisavam ser testadas e para saber quando essas condições foram especificadas em seus programas. Estudamos como manipular de forma diferente determinados elementos em uma lista com a eficiência do loop for. Revimos também as recomendações de estilo do Python para garantir que seus programas cada vez mais complexos ainda sejam relativamente fáceis de ler e entender.

No Capítulo 6, estudaremos os dicionários Python. Apesar de ser semelhante a uma lista, um dicionário possibilita relacionar informações. Aprenderemos a criar dicionários, percorrê-los com um loop e usá-los com listas e instruções if. Com os dicionários temos um mundo novo de possibilidades: você será capaz de modelar um leque ainda maior de situações reais.

---

1 N.T.: Nos Estados Unidos, em razão da não obrigatoriedade do voto, as pessoas se registram para votar em uma data ou no mesmo dia da votação, dependendo das leis estaduais. Já no Brasil, é só tirar o título de eleitor e votar.

CAPÍTULO 6
Dicionários

Neste capítulo, aprenderemos como usar os dicionários Python, que possibilitam associar informações relacionadas, e como acessar as informações assim que estiverem em um dicionário e como modificá-las. Como os dicionários podem armazenar uma quantidade praticamente ilimitada de informações, aprenderemos como percorrer com um loop os dados em um dicionário. Além disso, veremos como aninhar dicionários em listas, listas em dicionários e até mesmo dicionários em outros dicionários.

Compreender os dicionários possibilita modelar uma variedade de objetos reais com mais acurácia. Será possível criar um dicionário representando uma pessoa e, em seguida, armazenar quantas informações quisermos sobre essa pessoa. Você pode armazenar nome, idade, localização, profissão e qualquer outro aspecto de uma pessoa que possa ser representado. Será possível armazenar quaisquer categorias de informação que possam ser combinadas, como uma lista de palavras e seus significados, uma lista de nomes de pessoas e seus números favoritos, uma lista de montanhas e suas elevações, e assim por diante.

Um dicionário simples

Imagine um jogo de alienígenas com cores e valores de pontos diferentes. O simples dicionário a seguir armazena informações sobre um determinado alienígena:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

O dicionário `alien_0` armazena a cor e o valor dos pontos do alienígena. As duas últimas linhas acessam e exibem essas informações, conforme podemos ver:

```
green
5
```

Como acontece com a maioria dos conceitos novos de programação, utilizar dicionários requer prática. Após trabalhar um pouco com os dicionários, você perceberá como podem efetivamente modelar situações reais.

Trabalhando com dicionários

Um *dicionário* Python é uma coleção de *pares chaves-valor*. Cada *chave* está vinculada a um valor, e podemos usar uma chave para acessar o valor associado a essa chave. O valor de uma chave pode ser um número, uma string, uma lista ou até mesmo outro dicionário. Na verdade, podemos utilizar qualquer objeto criado em Python como um valor de um dicionário.

No Python, um dicionário é envolto entre chaves `{ }`, com uma série de pares chave-valor dentro das chaves, conforme mostrado no exemplo anterior:

```
alien_0 = {'color': 'green', 'points': 5}
```

Um *par chave-valor* é um conjunto de valores associados entre si. Ao fornecer uma chave, o Python retorna o valor associado a essa chave. Cada chave é vinculada ao seu valor por dois-pontos, e os pares chave-valor individuais são separados por vírgulas. Podemos armazenar quantos pares chave-valor quisermos em um dicionário.

O dicionário mais simples tem exatamente um par chave-valor, como podemos ver nesta versão modificada do dicionário `alien_0`:

```
alien_0 = {'color': 'green'}
```

Esse dicionário armazena uma informação sobre alien_0: a cor do alienígena. A string 'color' é uma chave do dicionário, e seu valor associado é 'green'.

Acessando valores em um dicionário

Para acessar o valor associado a uma chave, forneça o nome do dicionário e insira a chave entre colchetes, conforme mostrado a seguir:

alien.py

```
alien_0 = {'color': 'green'}
print(alien_0['color'])
```

O Python retorna o valor associado à chave 'color' do dicionário alien_0:

```
green
```

Um dicionário pode armazenar um número ilimitado de pares chave-valor. Por exemplo, vejamos o dicionário original alien_0 com dois pares chave-valor:

```
alien_0 = {'color': 'green', 'points': 5}
```

Agora, conseguimos acessar a cor ou o valor dos pontos de alien_0. Se um jogador abater esse alienígena, podemos verificar quantos pontos ele deve ganhar com o seguinte código:

```
alien_0 = {'color': 'green', 'points': 5}

new_points = alien_0['points']
print(f"You just earned {new_points} points!")
```

Uma vez definido o dicionário, extraímos o valor associado à chave 'points' do dicionário. Esse valor é então atribuído à variável new_points. A última linha exibe uma afirmação sobre quantos pontos o jogador acabou de ganhar:

```
You just earned 5 points!
```

Caso execute esse código sempre que um alienígena for abatido, poderá acessar o valor desses pontos.

Adicionando novos pares chave-valor

Como são estruturas dinâmicas, podemos adicionar novos pares chave-valor a um dicionário a qualquer momento. Para adicionar um novo par chave-valor, forneça o nome do dicionário seguido pela nova chave entre colchetes, com o valor novo.

Vamos adicionar duas informações novas ao dicionário `alien_0`: as coordenadas x e y do alienígena, que nos ajudarão a apresentá-lo em determinada posição na tela. Vamos posicionar o alienígena na borda esquerda da tela, 25 pixels abaixo da parte superior. Em geral, como as coordenadas da tela começam no canto superior esquerdo, posicionaremos o alienígena na borda esquerda da tela, definindo a coordenada x em 0 e 25 pixels a partir da parte superior, e a coordenada y com o valor 25 positivo, como a seguir:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

alien_0['x_position'] = 0
alien_0['y_position'] = 25
print(alien_0)
```

Começamos definindo o mesmo dicionário com o qual estamos trabalhando. Depois, exibimos esse dicionário, apresentando um snapshot de suas informações. Em seguida, adicionamos um novo par chave-valor ao dicionário: a chave `'x_position'` e o valor `0`. Fazemos o mesmo para a chave `'y_position'`. Quando exibimos o dicionário modificado, vemos os dois pares adicionais de chave-valor:

```
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'x_position': 0, 'y_position': 25}
```

A versão final do dicionário contém quatro pares chave-valor. Os dois pares originais especificam a cor e o valor do ponto, e os outros dois especificam a posição do alienígena.

Os dicionários preservam a ordem em que foram definidos. Ao exibir um dicionário ou percorrer seus elementos, você verá os elementos

na mesma ordem em que foram adicionados ao dicionário.

Começando com um dicionário vazio

Não raro, é conveniente, ou mesmo necessário, começar com um dicionário vazio e, aos poucos, adicionar itens novos. Para começar a preencher um dicionário vazio, defina um dicionário com um conjunto vazio de chaves e, em seguida, adicione cada par chave-valor em sua linha. Por exemplo, vejamos como criar o dicionário `alien_0` com essa abordagem:

alien.py

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Primeiro, definimos um dicionário vazio chamado `alien_0`. Depois, adicionamos valores para cor e pontos. O resultado é o dicionário usado nos exemplos anteriores:

```
{'color': 'green', 'points': 5}
```

Normalmente, recorremos a dicionários vazios quando armazenamos dados fornecidos pelo usuário em um dicionário ou quando escrevemos código que gera automaticamente um número significativo de pares chave-valor.

Modificando valores em um dicionário

Para modificar um valor em um dicionário, forneça o nome do dicionário com a chave entre colchetes e, em seguida, o valor novo que deseja associar a essa chave. Por exemplo, imagine um alienígena que muda da cor verde para a cor amarelo conforme o jogo avança:

alien.py

```
alien_0 = {'color': 'green'}
print(f"The alien is {alien_0['color']}.")
```

```
alien_0['color'] = 'yellow'
print(f"The alien is now {alien_0['color']}.")
```

Primeiro, definimos um dicionário para `alien_0` que contém somente o cor do alienígena; depois alteramos o valor associado à chave `'color'` para `'yellow'`. A saída mostra que o alienígena realmente mudou da cor verde para a cor amarela:

```
The alien is green.
The alien is now yellow.
```

Vejamos um exemplo mais interessante. Rastrearemos a posição de um alienígena que consegue se deslocar em diferentes velocidades. Armazenaremos um valor que representa a velocidade atual do alienígena e o usaremos para estipular a distância que o alienígena deve se deslocar à direita:

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}
print(f"Original position: {alien_0['x_position']}")

# Desloca o alienígena para direita
# Estipula a distância que o alienígena deve percorrer conforme sua velocidade
if alien_0['speed'] == 'slow':
    x_increment = 1
elif alien_0['speed'] == 'medium':
    x_increment = 2
else:
    # Com isso, o alienígena fica veloz
    x_increment = 3

# A posição nova é a posição antiga mais o incremento
alien_0['x_position'] = alien_0['x_position'] + x_increment

print(f"New position: {alien_0['x_position']}")
```

Começamos definindo um alienígena com as posições iniciais *x* e *y*, e a velocidade `'medium'`. Para simplificar as coisas, omitimos os valores da cor e dos pontos. Se você incluísse esses pares chave-valor, o exemplo funcionaria do mesmo jeito. Exibimos também o valor original de `x_position` a fim de conferirmos até onde o alienígena se desloca para a direita.

Uma sequência `if-elif-else` determina a distância que o alienígena deve

se deslocar para a direita e atribui esse valor à variável x_increment 1. Se a velocidade do alienígena for 'slow', ele se moverá uma unidade à direita; se for 'medium', se deslocará duas unidades à direita; e se for 'fast', ele se moverá três unidades à direita. Uma vez calculado, o incremento é adicionado ao valor de x_position 2, e o resultado é armazenado na x_position do dicionário.

Como temos um alienígena com velocidade média, sua posição se desloca duas unidades à direita:

```
Original x-position: 0
New x-position: 2
```

Trata-se de uma técnica bacana: ao alterar um valor no dicionário do alienígena, podemos alterar o comportamento geral do alienígena. Por exemplo, para transformar esse alienígena com velocidade média em um alienígena veloz, basta acrescentar a seguinte linha:

```
alien_0['speed'] = 'fast'
```

O bloco if-elif-else atribuiria um valor maior a x_increment na próxima vez que o código fosse executado.

Removendo pares chave-valor

Quando não precisarmos mais de uma informação armazenada em um dicionário, podemos utilizar o del para remover completamente um par chave-valor. Tudo o que del precisa é o nome do dicionário e a chave que você quer remover.

Por exemplo, removeremos a chave 'points' do dicionário alien_0, com seu valor:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

1 del alien_0['points']
print(alien_0)
```

A instrução del 1 informa ao Python para deletar a chave 'points' do dicionário alien_0 e também remover o valor associado a essa chave.

A saída mostra que a chave 'points' e seu valor de 5 foram excluídos do dicionário, mas o restante do dicionário permanece inalterado:

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

NOTA *Fique ciente de que o par chave-valor deletado é removido permanentemente.*

Um dicionário de objetos parecidos

O exemplo anterior incluía o armazenamento de diferentes tipos de informações sobre um objeto, um alienígena em um jogo. Podemos também utilizar um dicionário para armazenar um tipo de informação sobre muitos objetos. Por exemplo, digamos que você queira fazer uma pesquisa com diversas pessoas e perguntar qual é a linguagem de programação favorita delas. Vejamos como um dicionário ajuda a armazenar os resultados de uma pesquisa simples:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }
```

Como podemos ver, dividimos um dicionário maior em diversas linhas. Cada chave é o nome de uma pessoa que respondeu à pesquisa e cada valor é a escolha da linguagem de programação. Na hipótese de precisar de mais de uma linha para definir um dicionário, pressione ENTER após a chave de abertura. Depois, indente a próxima linha em um nível (quatro espaços) e escreva o primeiro par chave-valor, seguido por uma vírgula. De agora em diante, quando pressionar ENTER, seu editor de texto deve indentar automaticamente todos os próximos pares chave-valor, de modo que correspondam ao primeiro par chave-valor.

Definido o dicionário, adicione uma chave de fechamento em uma

linha nova logo após o último par chave-valor e a indente um nível para que fique alinhada com as chaves no dicionário. É boa prática incluir uma vírgula após o último par chave-valor, assim você estará pronto para adicionar um novo par chave-valor na próxima linha.

NOTA *A maioria dos editores tem funcionalidades que o ajuda a formatar listas e dicionários extensos de maneira semelhante a este exemplo. Outras formas aceitáveis de formatar dicionários extensos também estão disponíveis, logo é possível ver uma formatação ligeiramente diferente em seu editor ou em outros códigos.*

Para usar esse dicionário, dado o nome de uma pessoa que respondeu à pesquisa, podemos consultar com facilidade sua linguagem de programação favorita:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }

language = favorite_languages['sarah'].title()
print(f"Sarah's favorite language is {language}.")
```

Para conferir qual linguagem Sarah escolheu, basta acessar o valor em:

```
favorite_languages['sarah']
```

Usamos essa sintaxe para extrair a linguagem favorita de Sarah do dicionário `1` e atribuí-la à variável `language`. Aqui, como criamos uma variável nova, o `print()` se torna mais limpo. A saída exibe a linguagem favorita de Sarah:

```
Sarah's favorite language is C.
```

Você pode usar essa mesma sintaxe com qualquer pessoa representada no dicionário.

Usando get() para acessar valores

Usar chaves entre colchetes para acessar o valor que estamos interessados em um dicionário pode ocasionar um problema em potencial: se a chave solicitada não existir, receberemos um erro.

Vejamos o que acontece quando tentamos acessar o valor de pontos de um alienígena que não tem valor definido de pontos:

alien_no_points.py

```
alien_0 = {'color': 'green', 'speed': 'slow'}
print(alien_0['points'])
```

O Python retorna um traceback, mostrando um KeyError:

```
Traceback (most recent call last):
  File "alien_no_points.py", line 2, in <module>
    print(alien_0['points'])
          ~~~~~~~^^^^^^^^^^
KeyError: 'points'
```

No Capítulo 10, vamos aprender mais como lidar com esses tipos gerais de erro. Para dicionários especificamente, podemos usar o método get() a fim de definirmos um valor padrão que será retornado se a chave solicitada não existir.

O método get() exige uma chave como primeiro argumento. Como segundo argumento opcional, podemos passar o valor a ser retornado caso a chave não exista:

```
alien_0 = {'color': 'green', 'speed': 'slow'}

point_value = alien_0.get('points', 'No point value assigned.')
print(point_value)
```

Se a chave 'points' existir no dicionário, obteremos o valor correspondente. Se não, obtemos o valor padrão. Nesse caso, points não existe e recebemos uma mensagem limpa em vez de um erro:

```
No point value assigned.
```

Se houver uma chance de a chave que você está tentando acessar não existir, considere usar o método get() em vez de a notação de colchetes.

NOTA *Caso omita o segundo argumento para chamar o método* get() *e a chave não existir, o Python retornará o valor* None*. O valor especial* None *significa "nenhum valor existe". Não se trata de um erro: é um valor especial cujo objetivo é sinalizar a ausência de um valor. No Capítulo ٨, veremos mais detalhes sobre o* None*.*

FAÇA VOCÊ MESMO

6.1 Pessoa: Use um dicionário para armazenar informações sobre uma pessoa que você conhece. Armazene o nome, sobrenome, idade e a cidade onde mora. Nomeie as chaves como first_name, last_name, age e city. Exiba cada informação armazenada em seu dicionário.

6.2 Números favoritos: Use um dicionário para armazenar os números favoritos das pessoas. Pense em cinco nomes e os utilize como chaves em seu dicionário. Pense em um número favorito para cada pessoa e armazene cada um como um valor em seu dicionário. Exiba o nome de cada pessoa e seu número favorito. Para que tudo fique ainda mais divertido, pergunte a alguns amigos e obtenha alguns dados reais para o seu programa.

6.3 Glossário: Um dicionário Python pode ser usado para modelar um dicionário real. Contudo, para evitar confusão, vamos chamá-lo de glossário.

- Pense em cinco palavras do mundo de programação que você aprendeu nos capítulos anteriores. Use essas palavras como chaves em seu glossário e armazene seus significados como valores.
- Exiba cada palavra e seu significado como uma saída elegantemente formatada. É possível até mesmo exibir a palavra seguida por dois-pontos e depois seu significado ou a palavra em uma linha e, em seguida, exibir seu significado indentado em uma segunda linha. Use o caractere quebra de linha (\n) para inserir uma linha em branco entre cada par palavra-significado em sua saída.

Percorrendo um dicionário com um loop

Um único dicionário Python pode conter somente alguns pares chave-valor ou milhões de pares. Visto que um dicionário pode conter grandes volumes de dados, o Python possibilita percorrer um dicionário com um loop. Ou seja, como os dicionários podem ser utilizado para armazenar informações de diferentes formas, existem diferentes formas de percorrê-los com um loop. É possível percorrer todos os pares chave-valor de um dicionário com suas chaves ou com seus valores.

Percorrendo todos os pares chave-valor com um loop

Antes de nos aprofundarmos nas diferentes abordagens de loop, vejamos um dicionário novo, desenvolvido para armazenar informações sobre um usuário em um site. O dicionário a seguir armazenaria o nome de usuário, o nome e o sobrenome de uma pessoa:

user.py

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }
```

Tomando como base o que já aprendemos neste capítulo, podemos acessar qualquer informação sobre `user_0`. Mas e se quiséssemos verificar tudo que foi armazenado no dicionário desse usuário? Para isso, vamos percorrer o dicionário com um loop `for`:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }

for key, value in user_0.items():
    print(f"\nKey: {key}")
    print(f"Value: {value}")
```

A fim de percorrer um dicionário com um loop `for`, criamos nomes para as duas variáveis que armazenarão a chave e o valor em cada par chave-valor. Escolha quaisquer nomes que desejar para essas duas variáveis. Caso opte por abreviações para os nomes das variáveis, o código também executaria perfeitamente:

```
for k, v in user_0.items()
```

A segunda metade da instrução `for` inclui o nome do dicionário seguido pelo método `items()`, que retorna uma sequência de pares chave-valor. Em seguida, o loop `for` atribui cada um desses pares às duas variáveis fornecidas. No exemplo anterior, usamos as variáveis

para exibir cada chave, seguido pelo value associado. No primeiro print(), o "\n" garante que uma linha em branco seja inserida antes de cada par chave-valor na saída:

```
Key: username
Value: efermi

Key: first
Value: enrico

Key: last
Value: fermi
```

Percorrer com um loop todos os pares chave-valor funciona bem, ainda mais com dicionários como o *favorite_languages.py*, exemplo da página *135*, que armazena o mesmo tipo de informação para muitas chaves diferentes. Se percorrermos com um loop o dicionário favorite_languages, obteremos o nome de cada pessoa no dicionário e sua linguagem de programação favorita. Como as chaves sempre referenciam o nome de uma pessoa e o valor é sempre uma linguagem, usaremos as variáveis name e language no loop em vez de key e value, assim fica mais fácil acompanhar o que está acontecendo dentro do loop:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }

for name, language in favorite_languages.items():
    print(f"{name.title()}'s favorite language is {language.title()}.")
```

Esse código informa ao Python para percorrer com um loop cada par chave-valor no dicionário. À medida que percorre cada par, a chave é atribuída à variável name, e o valor é atribuído à variável language. Esses nomes descritivos facilitam e muito ver o que o print() está fazendo.

Agora, em apenas algumas linhas de código, podemos exibir todas as informações da pesquisa:

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Rust.
Phil's favorite language is Python.
```

Se o nosso dicionário armazenasse os resultados de uma pesquisa com mil ou um milhão de pessoas, esse tipo de loop também executaria perfeitamente.

Percorrendo todas as chaves de um dicionário com um loop

O método `keys()` ajuda bastante quando não precisamos trabalhar com todos os valores em um dicionário. Vamos percorrer com um loop o dicionário `favorite_languages` e exibir os nomes de todos os participantes da pequisa.

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }

for name in favorite_languages.keys():
    print(name.title())
```

Esse loop `for` informa ao Python para extrair todas as chaves do dicionário `favorito_languages` e atribui-las uma de cada vez à variável `name`. A saída mostra os nomes de todos os participantes da pesquisa.

```
Jen
Sarah
Edward
Phil
```

Na verdade, percorrer as chaves com um loop é comportamento padrão quando percorremos um dicionário. Ou seja, esse código geraria exatamente a mesma saída se você escrevesse:

```
for name in favorite_languages:
```

Em vez de:

```
for name in favorite_languages.keys():
```

Você pode optar por usar o método `keys()` explicitamente se isso facilitar a legibilidade de seu código ou pode omiti-lo, caso queira.

É possível acessar o valor associado a qualquer chave de interesse dentro do loop, usando a chave atual. Vamos exibir uma mensagem para alguns amigos sobre as linguagens de programação que escolherem. Vamos percorrer com um loop os nomes no dicionário como fizemos anteriormente, mas quando o nome corresponder a um de nossos amigos, exibiremos uma mensagem sobre sua linguagem favorita:

```
favorite_languages = {
    -- trecho de código omitido --
    }

friends = ['phil', 'sarah']
for name in favorite_languages.keys():
    print(f"Hi {name.title()}.")

1   if name in friends:
2       language = favorite_languages[name].title()
        print(f"\t{name.title()}, I see you love {language}!")
```

Primeiro, criamos uma lista de amigos para exibir uma mensagem. Dentro do loop, exibimos o nome de cada pessoa. Em seguida, verificamos se o `name` com o qual estamos trabalhando consta na lista `friends` `1`. Se constar, determinamos a linguagem favorita da pessoa usando o nome do dicionário e o valor atual de `name` como chave `2`. Depois, exibimos uma mensagem especial, incluindo uma referência à linguagem escolhida por cada um. O nome de todos é mostrado, mas nossos amigos recebem uma mensagem especial:

```
Hi Jen.
Hi Sarah.
    Sarah, I see you love C!
Hi Edward.
Hi Phil.
    Phil, I see you love Python!
```

Podemos também utilizar o método keys() para verificar se uma determinada pessoa participou da pesquisa. Desta vez, vamos verificar se Erin participou:

```
favorite_languages = {
    -- trecho de código omitido --
    }

if 'erin' not in favorite_languages.keys():
    print("Erin, please take our poll!")
```

O método keys() não é usado somente com loops: na verdade, retorna uma sequência de todas as chaves, e a instrução if simplesmente verifica se 'erin' está nessa sequência. Como ela não está, exibimos uma mensagem que a convida a participar da pesquisa:

```
Erin, please take our poll!
```

Percorrendo as chaves de um dicionário com um loop em uma ordem específica

Quando percorremos um dicionário com um loop, o Python retorna os elementos na mesma ordem em que foram inseridos. No entanto, queremos às vezes percorrer um dicionário com um loop em uma ordem diferente. Podemos fazer isso ordenando as chaves conforme são retornadas no loop for. Podemos utilizar a função sorted() para obter uma cópia das chaves ordenadas:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }

for name in sorted(favorite_languages.keys()):
    print(f"{name.title()}, thank you for taking the poll.")
```

Essa instrução for é como outras instruções for, exceto pelo fato de que envolvemos em um wrapper a função sorted() no método dictionary.keys(). Isso informa ao Python para obter todas as chaves no

dicionário e ordená-las antes de iniciar o loop. A saída mostra todos os participantes da pesquisa, com os nomes ordenados:

```
Edward, thank you for taking the poll.
Jen, thank you for taking the poll.
Phil, thank you for taking the poll.
Sarah, thank you for taking the poll.
```

Percorrendo todos os valores de um dicionário com um loop

Caso esteja basicamente interessado nos valores contidos em um dicionário, poderá usar o método `values()` para retornar uma sequência de valores sem nenhuma chave. Por exemplo, digamos que queremos apenas uma lista de todas as linguagens de programação escolhida em nossa pesquisa, sem o nome da pessoa que escolheu cada uma delas:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
    }

print("The following languages have been mentioned:")
for language in favorite_languages.values():
    print(language.title())
```

Aqui, a instrução `for` extrai cada valor do dicionário e o atribui à variável `language`. Quando esses valores são exibidos, obtemos uma lista das linguagens escolhidas:

```
The following languages have been mentioned:
Python
C
Rust
Python
```

Com essa abordagem, extraímos os valores do dicionário sem verificar se temos repetições. Isso até funciona bem com um pequeno número de valores, mas em uma pesquisa com um volume grande de participantes teríamos como resultado uma lista muito

repetitiva. Para ver cada linguagem escolhida sem repetição, podemos usar um conjunto. Um *conjunto* é uma coleção na qual cada item deve ser único:

```
favorite_languages = {
    -- trecho de código omitido --
    }

print("The following languages have been mentioned:")
for language in set(favorite_languages.values()):
    print(language.title())
```

Ao fazermos um wrapper de um `set()` em torno de uma coleção de valores que contém itens duplicados, o Python identifica os itens únicos na coleção e cria um conjunto a partir desses itens. Aqui, utilizamos `set()` para extrair as linguagens únicas em `favorite_languages.values()`.

O resultado é uma lista não repetitiva de linguagens escolhidas pelos participantes da pequisa:

```
The following languages have been mentioned:
Python
C
Rust
```

À medida que aprende mais sobre Python, você muitas vezes vai se deparar com um recurso integrado (também chamado de recurso built-in) da linguagem que o ajuda a fazer exatamente o que deseja com seus dados.

NOTA *Você pode criar um conjunto diretamente usando chaves e separando os elementos com vírgulas:*

```
>>> languages = {'python', 'rust', 'python', 'c'}
>>> languages
{'rust', 'python', 'c'}
```

É fácil confundir conjuntos com dicionários porque ambos estão entre chaves. Quando ver chaves, mas nenhum par chave-valor, provavelmente está olhando para um conjunto. Ao contrário de listas e dicionários, conjuntos não retêm itens em nenhuma ordem específica.

FAÇA VOCÊ MESMO

6.4 Glossário 2: Agora você sabe como percorrer um dicionário com um loop, limpe o código do Exercício 6.3 (página *138*) substituindo sua série de print() por um loop que percorre as chaves e os valores do dicionário. Quando tiver certeza de que seu loop funciona, adicione mais cinco termos Python ao seu glossário. Quando executar seu programa novamente, essas palavras e dignificados novos devem ser incluídos automaticamente na saída.

6.5 Rios: Crie um dicionário contendo os três principais rios e o país por onde cada rio passa. Um par chave-valor pode ser 'nile': 'egypt'.

- Use um loop para exibir uma frase sobre cada rio, como: *O Nilo atravessa o Egito*.
- Use um loop para exibir o nome de cada rio incluído no dicionário.
- Use um loop para exibir o nome de cada país incluído no dicionário.

6.6 Pesquisa: Use o código de *favorite_languages.py* (página *135*).

- Crie uma lista de pessoas que deveriam participar da pesquisa de linguagens favoritas. Inclua alguns nomes que já estão no dicionário e outros que não estão.
- Percorra com um loop a lista de pessoas que devem participar da pesquisa. Se já tiverem respondido, exiba uma mensagem agradecendo a resposta. Se ainda não tiverem respondido, exiba uma mensagem as convidando a participar.

Aninhamento

Não raro, queremos armazenar múltiplos dicionários em uma lista ou uma lista de itens como um valor em um dicionário. Chamamos isso de *aninhamento*. Podemos aninhar dicionários dentro de uma lista, uma lista de itens dentro de um dicionário ou até mesmo um dicionário dentro de outro dicionário. O aninhamento é um recurso poderoso, como veremos nos próximos exemplos.

Uma lista de dicionários

O dicionário alien_0 contém uma variedade de informações sobre um alienígena, mas não tem espaço para armazenar informações sobre um segundo alienígena, tampouco uma tela cheia deles. Como podemos lidar com uma frota de alienígenas? Uma forma é criar uma lista de alienígenas na qual cada alienígena é um dicionário de informações sobre ele mesmo. Por exemplo, o código a seguir cria uma lista de três alienígenas:

aliens.py

```
alien_0 = {'color': 'green', 'points': 5}
alien_1 = {'color': 'yellow', 'points': 10}
alien_2 = {'color': 'red', 'points': 15}

aliens = [alien_0, alien_1, alien_2]

for alien in aliens:
    print(alien)
```

Primeiro, criamos três dicionários, cada um representando um alienígena diferente. Armazenamos cada um desses dicionários em uma lista chamada `aliens` `1`. Por último, percorremos a lista com um loop e exibimos cada alienígena:

```
{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}
```

Um exemplo mais realista seria mais de três alienígenas em código que gere automaticamente cada alienígena. No exemplo a seguir, usamos `range()` para criar uma frota com 30 alienígenas:

```
# Cria uma lista vazia para armazenar alienígenas
aliens = []

# Cria 30 alienígenas verdes
for alien_number in range(30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

# Exibe os primeiros 5 alienígenas
for alien in aliens[:5]:
    print(alien)
print("...")

# Exibe quantos alienígenas foram criados
print(f"Total number of aliens: {len(aliens)}")
```

Esse exemplo começa com uma lista vazia a fim de armazenar todos os alienígenas que serão criados. A função `range()` `1` retorna uma série de números, informando ao Python quantas vezes queremos que o loop se repita. Cada vez que o loop é executado, criamos um alienígena novo `2` e anexamos cada alienígena novo à lista `aliens` `3`.

Utilizamos uma fatia para exibir os primeiros cinco alienígenas 4 e, por último, exibimos o comprimento da lista a fim de comprovarmos que realmente geramos a frota completa de 30 alienígenas:

```
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
...

Total number of aliens: 30
```

Embora todos esses alienígenas tenham as mesmas características, o Python considera cada um deles como um objeto independente, possibilitando modificar cada alienígena individualmente.

Como podemos trabalhar com um grupo de alienígenas desse? Imagine que uma das características do jogo seja alienígenas mudando de cor e se deslocando cada vez mais rápido conforme o jogo avança. Na hora de mudar as cores, podemos usar um loop `for` e uma instrução `if` para alterar a cor dos alienígenas. Por exemplo, podemos alterar a cor dos três primeiros alienígenas para amarelo, com velocidade média valendo 10 pontos cada, assim:

```
# Cria uma lista vazia para armazenar alienígenas
aliens = []

# Cria 30 alienígenas verdes
for alien_number in range (30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

# Exibe os primeiros 5 alienígenas
for alien in aliens[:5]:
    print(alien)
print("...")
```

Como queremos modificar os três primeiros alienígenas, iteramos uma fatia com somente os três primeiros alienígenas. Agora, todos os alienígenas são verdes, mas nem sempre será o caso, assim escrevemos uma instrução if para garantir que estamos apenas modificando alienígenas verdes. Se o alienígena for verde, mudamos a cor para 'yellow', a velocidade para 'medium e o valor do ponto para 10, conforme mostrado na saída a seguir:

```
{'color': 'yellow', 'points': 10, 'speed': 'medium'}
{'color': 'yellow', 'points': 10, 'speed': 'medium'}
{'color': 'yellow', 'points': 10, 'speed': 'medium'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
...
```

Podemos ampliar esse loop adicionando um bloco elif para converter alienígenas amarelos em alienígenas vermelhos e rápidos, valendo 15 pontos cada. Sem mostrar todo o programa novamente, esse loop ficaria assim:

```
for alien in aliens[0:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10
    elif alien['color'] == 'yellow':
        alien['color'] = 'red'
        alien['speed'] = 'fast'
        alien['points'] = 15
```

É habitual armazenar diversos dicionários em uma lista quando cada dicionário contém muitos tipos de informações sobre um objeto. Por exemplo, é possível criar um dicionário para cada usuário em um site, como fizemos em *user.py* na página *139*, e armazenar dicionários individuais em uma lista chamada users. Em uma lista, todos os dicionários devem ter estrutura idêntica. Assim, podemos percorrer a lista com um loop e trabalhar com cada objeto de dicionário da mesma forma.

Uma lista em um dicionário

Em vez de inserir um dicionário em uma lista, podemos convenientemente inserir uma lista em um dicionário. Por exemplo, pense em como poderia representar um pedido de uma pizza. Se usar somente uma lista, tudo o que você armazenaria seria uma lista dos ingredientes da pizza. Mas com um dicionário, essa lista de ingredientes pode ser apenas um aspecto da pizza que você está representando.

No exemplo a seguir, dois tipos de informações são armazenados para cada pizza: um tipo de massa e uma lista de ingredientes. A lista de ingredientes é um valor associado à chave `'toppings'`. Se quisermos usar os elementos da lista, basta fornecer o nome do dicionário e a chave `'toppings'`, como faríamos com qualquer valor no dicionário. Em vez de retornar um único valor, obtemos uma lista de ingredientes:

pizza.py

```
# Armazena informações sobre uma pizza que está sendo pedida
pizza = {
    'crust': 'thick',
    'toppings': ['mushrooms', 'extra cheese'],
    }

# Resume o pedido
print(f"You ordered a {pizza['crust']}-crust pizza "
    "with the following toppings:")

for topping in pizza['toppings']:
    print(f"\t{topping}")
```

Primeiro, criamos um dicionário que armazena informações sobre uma pizza que está sendo pedida. No dicionário temos uma chave `'crust'`, e o valor associado é a string `'thick'`. A próxima chave, `'toppings'`, tem como valor uma lista que armazena todos os ingredientes solicitados. Resumimos o pedido antes de preparar a pizza `1`. Caso precise quebrar uma linha extensa em um `print()`, escolha um ponto adequado para quebrar a linha que está sendo exibida e termine a

linha com uma aspa. Indente a próxima linha, adicione uma aspa de abertura e escreva a string. O Python combinará automaticamente todas as strings que encontrar dentro dos parênteses. Para exibir os ingredientes, escrevemos um loop `for` 2. Para acessar a lista de ingredientes, utilizamos a chave `'toppings'`, assim o Python obtém a lista de ingredientes do dicionário.

A saída a seguir exibe um resumo da pizza prepararemos:

```
You ordered a thick-crust pizza with the following toppings:
    mushrooms
    extra cheese
```

É possível aninhar uma lista em um dicionário sempre que desejar que mais de um valor seja associado a uma única chave. No exemplo anterior de linguagens de programação favoritas, se tivéssemos armazenado as respostas de cada pessoa em uma lista, elas poderiam escolher mais de uma linguagem favorita. Ou seja, caso tivéssemos percorrido o dicionário com um loop, o valor associado a cada pessoa seria uma lista de linguagens em vez de apenas uma linguagem. Dentro do loop `for` do dicionário, utilizamos outro loop `for` para percorrer a lista de linguagens associadas a cada pessoa:

favorite_languages.py

```
favorite_languages = {
    'jen': ['python', 'rust'],
    'sarah': ['c'],
    'edward': ['rust', 'go'],
    'phil': ['python', 'haskell'],
    }

for name, languages in favorite_languages.items():
    print(f"\n{name.title()}'s favorite languages are:")
    for language in languages:
        print(f"\t{language.title()}")
```

Agora, o valor associado a cada nome em `favorite_languages` é uma lista. Repare que algumas pessoas têm uma linguagem favorita e outras têm mais. Ao percorrermos o dicionário com um loop 1, usamos a

variável languages para armazenar cada valor do dicionário, porque sabemos que cada valor será uma lista. Dentro do loop de dicionário principal, utilizamos outro loop for 2 a fim de percorrer a lista de linguagens favoritas de cada pessoa. Agora, cada pessoa pode especificar quantas linguagens favoritas quiser:

```
Jen's favorite languages are:
    Python
    Rust

Sarah's favorite languages are:
    C

Edward's favorite languages are:
    Rust
    Go

Phil's favorite languages are:
    Python
    Haskell
```

Para refinar ainda mais esse programa, podemos incluir instrução if no início do loop for do dicionário a fim de verificarmos se cada pessoa tem mais de uma linguagem favorita examinando o valor de len(languages). Se uma pessoa tiver mais de uma linguagem favorita, a saída permanecerá a mesma. Se a pessoa tiver somente uma linguagem favorita, podemos alterar o texto para exibir isso. Por exemplo, poderíamos escrever: "Sarah's favorite language is C".

NOTA *Não devemos ter listas e dicionários profundamente aninhados. Se estiver aninhando elementos em um nível mais profundo do que os exemplos anteriores ou se estiver trabalhando com o código de outra pessoa com níveis significativos de aninhamento, provavelmente existe um jeito mais simples de resolver o problema.*

Um dicionário em um dicionário

Podemos aninhar um dicionário dentro de outro dicionário, mas o código pode se tornar complexo se fizermos isso. Por exemplo, se

tiver um site com diversos usuários, cada um com um nome de usuário único, você poderá usar os nomes de usuário como chaves em um dicionário. Você pode armazenar informações sobre cada usuário usando um dicionário como valor associado ao nome de usuário. Na lista a seguir, armazenamos três informações sobre cada usuário: nome, sobrenome e localização. Acessaremos essas informações percorrendo com um loop os nomes de usuário e o dicionário de informações associado a cada nome de usuário:

many_users.py

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
        },

    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
        },

    }

1 for username, user_info in users.items():
2     print(f"\nUsername: {username}")
3     full_name = f"{user_info['first']} {user_info['last']}"
      location = user_info['location']

4     print(f"\tFull name: {full_name.title()}")
      print(f"\tLocation: {location.title()}")
```

De início, definimos um dicionário chamado `users` com duas chaves: uma para os nomes de usuário `'aeinstein'` e `'mcurie'`. O valor associado a cada chave é um dicionário que inclui o nome, sobrenome e localização de cada usuário. Em seguida, percorremos o dicionário `users` `1`. O Python atribui cada chave à variável `username` e o dicionário associado a cada nome de usuário é atribuído à variável `user_info`. Uma vez dentro do loop principal do dicionário, exibimos o nome de

usuário 2.

Em seguida, passamos a acessar o dicionário interno 3. A variável user_info, que contém o dicionário de informações do usuário, tem três chaves: 'first', 'last' e 'location'. Usamos cada chave para gerar um nome completo e localização elegantemente formatados para cada pessoa e, depois, exibimos um resumo do que sabemos sobre cada usuário 4:

```
Username: aeinstein
    Full name: Albert Einstein
    Location: Princeton

Username: mcurie
    Full name: Marie Curie
    Location: Paris
```

Veja que a estrutura do dicionário de cada usuário é idêntica. Ainda que não seja exigência do Python, essa estrutura facilita trabalhar com dicionários aninhados. Se o dicionário de cada usuário tivesse chaves diferentes, o código dentro do loop for seria mais complexo.

FAÇA VOCÊ MESMO

6.7 Pessoas: comece com o programa escrito para Exercício 6.1 (página *138*). Crie dois dicionários novos representando pessoas diferentes e armazene todos os três dicionários em uma lista chamada people. Percorra sua lista de pessoas com um loop. À medida que percorre a lista, exiba tudo o que sabe sobre cada pessoa.

6.8 Animais de estimação: Crie vários dicionários, em que cada dicionário representa um animal de estimação diferente. Em cada dicionário inclua o tipo de animal e o nome do dono. Armazene esses dicionários em uma lista chamada pets. Depois, percorra sua lista com um loop e, enquanto faz isso, exiba tudo o que sabe sobre cada animal de estimação.

6.9 Lugares favoritos: Crie um dicionário chamado favorite_places. Pense em três nomes para usar como chave no dicionário e armazene de um a três lugares favoritos para cada pessoa. Agora, para que este exercício fique ainda mais interessante, peça a alguns amigos que lhe digam alguns de seus lugares favoritos. Percorra o dicionário com um loop e exiba o nome de cada pessoa e seus lugares favoritos.

6.10 Números favoritos: Modifique seu programa do Exercício 6.2 (página *138*) para que cada pessoa possa ter mais de um número favorito. Depois, exiba o nome de cada pessoa com seus números favoritos.

6.11 Cidades: Crie um dicionário chamado cities. Utilize o nome de três cidades como chaves de seu dicionário. Crie um dicionário de informações sobre cada cidade e inclua o país em que a cidade está, sua população aproximada e um fato sobre essa cidade. O

nome das chaves para o dicionário de cada cidade devem ser alguma coisa como country, population e fact. Exiba o nome de cada cidade e todas as informações que você armazenou a respeito.

6.12 Extensões: Agora que já estamos trabalhando com exemplos complexos o suficiente para que sejam mais desenvolvidos de diferentes maneira, use um dos programas de exemplo deste capítulo e o amplie, adicionando chaves e valores novos, alterando o contexto do programa ou melhorando a formatação da saída.

Recapitulando

Neste capítulo, aprendemos como definir um dicionário e como trabalhar com as informações armazenadas em um dicionário. Estudamos como acessar e modificar elementos individuais em um dicionário e como percorrer todas as informações em um dicionário com um loop. Aprendemos a percorrer os pares chave-valor de um dicionário, suas chaves e seus valores com um loop e também como aninhar diversos dicionários em uma lista, aninhar listas em um dicionário e aninhar um dicionário em outro dicionário.

No próximo capítulo, estudaremos os loops while e como aceitar entradas de pessoas que estão usando seus programas. Será um capítulo incrível, porque aprenderemos como tornar nossos programas interativos: eles serão capazes de responder à entrada do usuário.

CAPÍTULO 7

Entrada do usuário e loops while

Escrevemos a maioria dos programas a fim de resolver problemas dos usuários finais. Para isso, precisamos normalmente obter algumas informações do usuário. Por exemplo, digamos que alguém queira saber se tem idade suficiente para votar. Se escrevermos um programa para responder a essa pergunta, precisaremos saber a idade do usuário antes de fornecer uma resposta. Será necessário que o programa solicite ao usuário seus *dados* ou que ele forneça sua idade. Após receber essa informação como entrada, o programa pode compará-la com a idade de votação para estipular se o usuário tem idade suficiente e, em seguida, relatar o resultado.

Neste capítulo, aprenderemos como aceitar a entrada do usuário para que nossos programas consigam trabalhar com esses dados. Quando programas precisam de nomes, podemos solicitá-los ao usuário. Quando programas precisam de uma lista de nomes, podemos solicitar ao usuário uma série de nomes. Para isso, utilizaremos a função `input()`.

Aprenderemos também como manter os programas em execução pelo tempo que os usuários desejarem, assim eles podem imputar todas as informações necessárias, e seu programa pode trabalhar com essas informações. E para mantermos os programas em execução usaremos o loop `while`, desde que determinadas condições permaneçam verdadeiras.

Com a capacidade de receber os dados de entrada do usuário e

controlar o tempo de execução dos programas, desenvolveremos programas totalmente interativos.

Como a função input() funciona

A função input() pausa um programa e espera que o usuário forneça dados em forma de texto. Após receber a entrada do usuário, o Python atribui essa entrada a uma variável, assim podemos trabalhar de forma prática com esses dados de entrada.

Por exemplo, o programa a seguir solicita que o usuário insira uma entrada em forma de texto e, em seguida, exibe essa mensagem de volta ao usuário:

parrot.py

```
message = input("Tell me something, and I will repeat it back to you: ")
print(message)
```

A função input() recebe um argumento: o *prompt* que queremos exibir ao usuário, para que saiba que tipo de informação inserir. Nesse exemplo, quando o Python executa a primeira linha, o usuário vê o prompt Tell me something, and I will repeat it back to you:. O programa espera enquanto o usuário insere a resposta e continua a execução após o usuário pressionar ENTER. A resposta é atribuída à variável message e, em seguida, o print(message) exibe a entrada de volta para o usuário:

```
Tell me something, and I will repeat it back to you: Hello everyone!
Hello everyone!
```

NOTA *Alguns editores de texto não executam programas que solicitam entrada do usuário. Você pode utilizar esses editores para escrever programas que solicitam entrada, mas precisará executá-los em um terminal. Consulte a seção "Executando programas Python em um terminal" na página 43.*

Escrevendo prompts limpos

Sempre que utilizar a função input(), escreva um prompt limpo e de fácil entendimento, que informe ao usuário exatamente o tipo de

informação que você está procurando. Qualquer frase que diga ao usuário o que inserir deve funcionar. Por exemplo:

greeter.py

```
name = input("Please enter your name: ")
print(f"\nHello, {name}!")
```

Adicione um espaço no final de seus prompts (após os dois-pontos no exemplo anterior) a fim de separar o prompt da resposta do usuário e deixar claro ao usuário onde inserir o texto. Por exemplo:

```
Please enter your name: Eric
Hello, Eric!
```

Às vezes queremos escrever um prompt com mais de uma linha. Por exemplo, talvez você queira informar ao usuário por que está solicitando determinada entrada. É possível atribuir seu prompt a uma variável e passar essa variável à função `input()`. Isso possibilita criar um prompt com diversas linhas e escrever uma instrução `input()` limpa.

greeter.py

```
prompt = "If you share your name, we can personalize the messages you see."
prompt += "\nWhat is your first name? "

name = input(prompt)
print(f"\nHello, {name}!")
```

Esse exemplo nos mostra uma das formas de criar uma string multilinha. A primeira linha atribui a primeira parte da mensagem à variável `prompt`. Na segunda linha, o operador `+=` pega a string atribuída ao `prompt` e adiciona a string nova no final.

Agora, o prompt se estende por duas linhas, novamente com espaço após o ponto de interrogação para maior clareza:

```
If you share your name, we can personalize the messages you see.
What is your first name? Eric

Hello, Eric!
```

Usando int() para receber entradas numéricas

Quando usamos a função input(), o Python interpreta tudo o que o usuário insere como uma string. Vejamos a sessão do interpretador, que pergunta a idade do usuário:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age
'21'
```

Embora o usuário digite o número 21, quando solicitamos o valor de age, o Python retorna '21', a string que representa o valor numérico digitado. Sabemos que o Python interpretou a entrada como uma string, pois o número agora está entre aspas. Caso queira apenas exibir a entrada, o código funciona bem. Mas caso tente usar a entrada como um número, receberá um erro:

```
>>> age = input("How old are you? ")
How old are you? 21
1 >>> age >= 18
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
2 TypeError: '>=' not supported between instances of 'str' and 'int'
```

Como tentamos utilizar a entrada para fazer uma comparação numérica 1, o Python gera um erro porque não consegue comparar uma string com um inteiro: a string '21' atribuída à age não pode ser comparada com o valor numérico 18 2.

É possível resolver esse problema usando a função int(), que converte a string de entrada em um valor numérico. Isso possibilita que a comparação seja executada com sucesso:

```
>>> age = input("How old are you? ")
How old are you? 21
1 >>> age = int(age)
>>> age >= 18
True
```

Neste exemplo, quando inserimos 21 no prompt, o Python interpreta o número como uma string, mas o valor é convertido em uma representação numérica por int() 1. Desse modo, o Python consegue

executar o teste condicional: comparar age (que agora representa o valor numérico 21) e 18 a fim de verificar se age é maior ou igual a 18. O teste é avaliado como True.

Como usamos a função int () em um programa propriamente dito? Imagine um programa que verificar se as pessoas têm altura suficiente para andar em uma montanha-russa:

rollercoaster.py

```
height = input("How tall are you, in inches? ")
height = int(height)

if height >= 48:
    print("\nYou're tall enough to ride!")
else:
    print("\nYou'll be able to ride when you're a little older.")
```

O programa consegue comparar height com 48, já que height = int(height) converte o valor de entrada em uma representação numérica antes que a comparação seja feita. Se o número inserido for maior ou igual a 48, informamos ao usuário se sua altura é o suficiente:

```
How tall are you, in inches? 71

You're tall enough to ride!
```

Ao usar a entrada numérica para realizar cálculos e comparações, não se esqueça de converter primeiro o valor de entrada em uma representação numérica.

Operador módulo

Um mecanismo que ajuda muito a trabalhar com informações numéricas é o *operador módulo* (%), que divide um número por outro número e retorna o resto:

```
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
```

```
>>> 7 % 3
1
```

O operador módulo não informa o resultado da divisão entre dois valores e sim o resto da divisão entre dois valores.

Quando um número é divisível por outro número, o resto dessa divisão é 0. Logo, o operador módulo sempre retorna 0. Podemos usar esse operador para determinar se um número é par ou ímpar:

even_or_odd.py

```
number = input("Enter a number, and I'll tell you if it's even or odd: ")
number = int(number)

if number % 2 == 0:
    print(f"\nThe number {number} is even.")
else:
    print(f"\nThe number {number} is odd.")
```

Como números pares são sempre divisíveis por dois, se o módulo de um número divido por dois for zero (aqui, `if number % 2 == 0`) o número é par. Caso contrário, é ímpar.

```
Enter a number, and I'll tell you if it's even or odd: 42

The number 42 is even.
```

FAÇA VOCÊ MESMO

7.1 Aluguel de carro: Escreva um programa que pergunte ao usuário que tipo de carro ele gostaria de alugar. Exiba uma mensagem sobre esse carro, como: "Vejamos se consigo encontrar um Subaru para você".

7.2 Reservas em restaurante: Crie um programa que pergunte quantos lugares em uma mesa o usuário precisa. Se a resposta for mais de oito, exiba uma mensagem informando que é necessário aguardar por uma mesa. Caso contrário, informe que a mesa já está disponível.

7.3 Múltiplos de dez: Solicite ao usuário um número e informe se o número é múltiplo de 10 ou não.

Apresentando os loops while

O loop `for` itera em uma coleção de itens e executa um bloco de código por vez para cada elemento da coleção. Em contrapartida, o loop `while` é executado desde que, ou *enquanto*, uma determinada

condição for verdadeira.

Usando o loop while

Podemos usar o loop while para contar uma série de números. Por exemplo, o loop while a seguir conta de 1 a 5:

counting.py

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

Na primeira linha começamos a contar a partir de 1, atribuindo o valor 1 à current_number. Com essa definição, o loop while continua executando enquanto o valor de current_number for menor ou igual a 5. O código dentro do loop exibe o valor de current_number e adiciona 1 a esse valor: current_number += 1 (O operador += é um atalho para current_number = current_number + 1).

O Python repete o loop enquanto a condição current_number <= 5 for verdadeira. Como 1 é menor que 5, o Python exibe 1 e adiciona 1, atualizando o número atual em 2. Como 2 é menor que 5, o Python exibe 2 e adiciona 1 novamente, atualizando o número atual em 3, e assim por diante. Assim que o valor de current_number for maior que 5, o loop para de executar e o programa termina:

```
1
2
3
4
5
```

É bem provável que os programas que você usa diariamente tenham loops while. Por exemplo, um jogo precisa de um loop while que continue em execução durante o tempo em que um jogador quiser jogar e que pare assim que o jogador encerrar o jogo. Não seria divertido usar um programa que parasse de executar sem que nós os instruíssemos ou que continuasse executando, mesmo após de encerrá-lo. Por isso, os loops while são de grande utilidade.

Permitindo que o usuário encerre um programa

Podemos fazer com que o programa *parrot.py* execute pelo tempo que o usuário quiser, inserindo boa parte dele em um loop while. Vamos definir um *valor de saída* e, em seguida, vamos manter o programa em execução, desde que o usuário não tenha inserido o valor de saída:

parrot.py

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)
    print(message)
```

Primeiro, definimos um prompt que informa ao usuário duas opções: inserir uma mensagem ou inserir o valor de saída (nesse caso, 'quit'). Depois, definimos uma variável message para acompanhar qualquer valor inserido pelo usuário. Definimos message como uma string vazia, "". Desse modo, o Python consegue realizar uma verificação assim que iterar a linha while. Na primeira vez que o programa é executado e o Python itera a instrução while, é necessário comparar o valor de message com 'quit', ainda que o usuário não tenha fornecido nenhuma entrada. Caso não tenha nada para efetuar a comparação, o Python não será capaz de continuar executando o programa. Para solucionar esse problema, fizemos questão de atribuir um valor inicial à message. Ainda que seja uma string vazia, o Python consegue entendê-la, possibilitando efetuar a comparação que faz o loop while ser executado. Esse loop while é executado enquanto o valor de message não for 'quit'.

Na primeira passagem pelo loop, message é apenas uma string vazia, então o Python entra no loop. Em message = input(prompt), o Python exibe o prompt e espera o usuário fornecer uma entrada. Seja lá o que for fornecido pelo usuário, essa entrada é atribuída à message e exibida; depois, o Python reavalia a condição na instrução while.

Contanto que o usuário não tenha digitado a palavra 'quit', o prompt é exibido mais uma vez e o Python espera por mais entradas. Quando o usuário finalmente digita 'quit', o Python interrompe a execução do loop while e o programa termina:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
quit
```

Ainda que funcione bem, esse programa exibe a palavra 'quit' como se fosse uma mensagem real. Um simples teste if corrige isso:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Agora, o programa executa uma breve verificação, antes de apresentar a mensagem e apenas a exibe se não corresponder ao valor de quit:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

Usando flags

No exemplo anterior fizemos o programa executar determinadas tarefas enquanto uma determinada condição era verdadeira. E quanto aos programas mais complicados, nos quais muitos eventos diferentes podem interromper sua execução?

Por exemplo, inúmeros eventos diferentes podem encerrar um jogo. Um jogo termina quando um jogador perde todas suas espaçonaves, quando o tempo se esgota ou quando as cidades que deveria proteger são destruídas. Se quaisquer eventos desses acontecer, o jogo termina. Como diversos eventos possíveis podem interromper um programa, se tentarmos testar essas condições com uma instrução `while`, as coisas podem ficar complicadas e difíceis de se resolver.

Caso seja necessário que um programa execute apenas enquanto diversas condições forem verdadeiras, podemos definir uma variável a fim de determinar se o programa está ativo ou não. Essa variável, chamada de *flag*, se comporta com uma variável sinalizadora para o programa. Podemos criar programas que sejam executados enquanto uma flag estiver definida como `True` e que parem de executar quando qualquer um dos diversos eventos definir o valor da flag como `False`. Com isso, nossa instrução `while` geral precisa verificar somente uma condição: se a presente flag é `True`. Desse modo, todos os nossos outros testes (a fim de verificar a ocorrência de um evento que deve definir uma flag como `False`) podem ser ordenadamente organizados em todo o programa.

Adicionaremos uma flag no arquivo *parrot.py* da seção anterior. Essa flag, que chamaremos de `active` (embora você possa chamá-la do que quiser) monitorará se o programa deve ou não continuar executando:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

active = True
```

```
1 while active:
    message = input(prompt)

    if message == 'quit':
        active = False
    else:
        print(message)
```

Definimos a variável `active` como `True` para que o programa inicie com o status ativo. Isso simplifica ainda mais a instrução `while` porque nenhuma comparação é feita na instrução; a lógica é calculada em outras partes do programa. Desde que a variável `active` permaneça como `True`, o loop continuará executando `1`.

Na instrução `if` dentro do loop `while`, verificamos o valor de `message` assim que o usuário insere sua entrada. Se o usuário digitar `'quit'`, definimos `active` como `False` e o loop `while` para de executar. Se o usuário inserir algo diferente de `'quit'`, exibimos sua entrada como uma mensagem.

Esse programa tem a mesma saída do exemplo anterior, em que inserimos o teste condicional diretamente na instrução `while`. No entanto, agora que temos uma flag para sinalizar se o programa geral está com status ativo, seria fácil adicionar mais testes (como instruções `elif`) para eventos que devem definir `active` como `False`. Isso é de grande ajuda em programas complicados como jogos, nos quais pode haver muitos eventos que devem interromper a execução do programa. Sempre que qualquer um desses eventos definir a flag active como `False`, o loop principal do jogo será encerrado, uma mensagem *Game Over* poderá ser exibida e o jogador pode ter a opção de jogar mais uma vez.

Usando o break para sair de um loop

Para sair de um loop `while` imediatamente sem executar mais nenhum código do loop, independentemente dos resultados de qualquer teste condicional, use a instrução `break`. A instrução `break` direciona o fluxo do seu programa; é possível utilizá-la para controlar quais

linhas de código são ou não executadas. Assim, o programa execute somente o código que você quer, quando quiser.

Por exemplo, considere um programa que pergunta ao usuário sobre os lugares que visitou. Podemos interromper o loop while chamando o break assim que o usuário inserir o valor 'quit':

cities.py

```
prompt = "\nPlease enter the name of a city you have visited:"
prompt += "\n(Enter 'quit' when you are finished.) "

while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print(f"I'd love to go to {city.title()}!")
```

Um loop que começa com while True será executado incessantemente, a menos que seja interrompido por uma instrução break. Nesse programa, o loop continua solicitando ao usuário que insira o nome das cidades em que esteve até que seja fornecido 'quit'. Quando 'quit' é inserido, a instrução break é executada, fazendo com que o Python saia do loop:

```
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) New York
I'd love to go to New York!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) San Francisco
I'd love to go to San Francisco!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) quit
```

NOTA *É possível usar a instrução* break *em qualquer um dos loops do Python. Por exemplo, podemos utilizar o* break *para sair de um loop* for *que esteja percorrendo uma lista ou dicionário.*

Usando continue em um loop

Em vez de sair completamente de um loop com um break sem executar o restante do código, podemos utilizar a instrução continue para retornar ao início do loop, tomando como base o resultado de um teste condicional. Por exemplo, vejamos um loop que conta de 1 a 10, mas exibe apenas os números ímpares nesse intervalo:

counting.py

```
current_number = 0
while current_number < 10:
1    current_number += 1
    if current_number % 2 == 0:
        continue

    print(current_number)
```

Primeiro, definimos current_number como 0. Por ser menor que 10, o Python entra no loop while. Uma vez dentro do loop, incrementamos o contador em 1 1, assim o valor de current_number é 1. Em seguida, a instrução if verifica o módulo de current_number e 2. Se o módulo for 0 (significando que current_number é divisível por 2), a instrução continue informa ao Python para ignorar o restante do loop e retornar ao início. Se o número atual não for divisível por 2, o restante do loop será executado e o Python exibirá o número atual:

```
1
3
5
7
9
```

Evitando loops infinitos

Todo loop while precisa de um jeito para interromper a execução, caso contrário, é executado continua e ininterruptamente. Por exemplo, o loop a seguir deve contar de 1 a 5:

counting.py

```
x = 1
while x <= 5:
```

```
    print(x)
    x += 1
```

No entanto, caso omita sem querer a linha `x += 1`, o loop será executado ininterruptamente:

```
# Este loop é executado eternamente!
x = 1
while x <= 5:
    print(x)
```

Agora o valor de `x` começará em `1`, mas nunca mudará. Como resultado, o teste condicional `x <= 5` sempre será avaliado como `True` e o loop `while` será executado indefinidamente, exibindo uma série de 1s, assim:

```
1
1
1
1
-- trecho de código omitido --
```

Não existe programador que não escreva por engano um loop `while` infinito de vez em quando, principalmente quando os loops de um programa têm condições sutis. Caso seu programa trave em um loop infinito, pressione CTRL+C ou apenas feche a janela do terminal que exibe a saída do programa.

Para evitar escrever loops infinitos, teste cada loop `while` e garanta que parem conforme o esperado. Se quisermos que o programa encerre assim que o usuário inserir um determinado valor de entrada, execute o programa e insira esse valor. Se não encerrar, examine minuciosamente a maneira como o programa lida com o valor que deve gerar a saída do loop. Não se esqueça de que, pelo menos, uma parte do programa pode definir a condição do loop como `False` ou parar em uma instrução `break`.

NOTA *O VS Code, como muitos editores, exibe a saída em uma janela de terminal integrada. Para cancelar um loop infinito, clique na área de saída do editor antes de pressionar CTRL+C.*

FAÇA VOCÊ MESMO

7.4 Ingredientes de pizza: Escreva um loop que solicite ao usuário uma série de ingredientes de pizza até que ele forneça o valor 'quit'. À medida que cada ingrediente é fornecido, exiba uma mensagem informando que esses ingredientes estão sendo adicionados à pizza.

7.5 Ingressos de cinema: Um cinema cobra preços de ingressos diferentes, dependendo da idade da pessoa. Se a pessoa for menor de 3 anos, o ingresso é gratuito; se tiver entre 3 e 12 anos, o ingresso custa U$S10; e caso tenha mais de 12 anos, o ingresso custa US$15. Escreva um loop que pergunte a idade dos usuários e, em seguida, informe o preço do ingresso do cinema.

7.6 Três saídas: Crie diferentes versões do Exercício 7.4 ou 7.5 que executem cada uma das seguintes tarefas, pelo menos uma vez:

- Use um teste condicional na instrução while para interromper o loop.
- Use uma variável active para controlar o tempo que o loop é executado.
- Use uma instrução break para sair do loop quando o usuário inserir o valor 'quit'.

7.7 Infinito: Escreva e execute um loop infinito. (Para encerrar o loop, pressione CTRL+C ou feche a janela que exibe a saída.)

Usando um loop while com listas e dicionários

Até o momento, trabalhamos somente com uma informação do usuário por vez. Recebemos a entrada do usuário e, em seguida, exibimos a entrada ou uma resposta a ela. E na próxima passagem pelo loop while, recebemos outro valor de entrada e resposta. No entanto, para acompanharmos muitos usuários e informações, precisaremos de listas e dicionários em nossos loops while.

Por mais que um loop for seja eficaz para percorrer uma lista, não devemos modificar uma lista dentro de um loop for porque o Python terá dificuldades de acompanhar os elementos da lista. Para modificar uma lista enquanto trabalha nela, utilize um loop while. O uso de loops while com listas e dicionários possibilita coletar, armazenar e organizar muitas entradas para examiná-las e, posteriormente, convertê-las em informações.

Transferindo elementos de uma lista para outra

Imagine uma lista de usuários recém-registrados, mas não verificados, de um site. Após verificarmos esses usuários, como podemos transferi-los para uma lista separada de usuários confirmados? Uma possibilidade seria usar um loop while para extrair os usuários da lista de usuários não confirmados ao mesmo tempo que os verificamos e, depois, adicioná-los a uma lista separada de usuários confirmados. Vejamos como ficaria esse código:

confirmed_users.py

```
    # Começa com usuários que precisam ser verificados,
    # e uma lista vazia a fim de armazenar os usuários confirmados
 1  unconfirmed_users = ['alice', 'brian', 'candace']
    confirmed_users = []

    # Faz a verificação de cada usuário até que não se tenha mais
    # usuários não confirmados
    # Transfere cada usuário verificado para a lista de usuários confirmados
 2  while unconfirmed_users:
 3      current_user = unconfirmed_users.pop()

        print(f"Verifying user: {current_user.title()}")
 4      confirmed_users.append(current_user)

    # Exibe todos os usuários confirmados
    print("\nThe following users have been confirmed:")
    for confirmed_user in confirmed_users:
        print(confirmed_user.title())
```

Começamos com uma lista de usuários não confirmados 1 (Alice, Brian e Candace) e uma lista vazia a fim de armazenarmos os usuários confirmados. O loop while é executado enquanto a lista unconfirmed_users não estiver vazia 2. Dentro desse loop, o método pop() remove os usuários não verificados, um de cada vez, do final de unconfirmed_users 3. Como Candace é a última na lista unconfirmed_users, seu nome será o primeiro a ser removido, atribuído a current_user e adicionado à lista confirmed_users 4. O próximo é Brian, depois Alice.

Simulamos a confirmação de cada usuário exibindo uma mensagem de verificação e os adicionando à lista de usuários confirmados.

Conforme a lista de usuários não confirmados diminui, a lista de usuários confirmados aumenta. Quando a lista de usuários não confirmados estiver vazia, o loop é interrompido e a lista de usuários confirmados é exibida:

```
Verifying user: Candace
Verifying user: Brian
Verifying user: Alice

The following users have been confirmed:
Candace
Brian
Alice
```

Removendo todas as instâncias de valores específicos de uma lista

No Capítulo 3, usamos `remove()` para remover um valor específico de uma lista. A função `remove()` funcionou, pois o valor em que estávamos interessados aparecia somente uma vez na lista. Mas, e se quiséssemos remover todas as instâncias de um valor de uma lista?

Imagine que temos uma lista de animais de estimação cujo valor `'cat'` está repetido diversas vezes. Para remover todas as instâncias desse valor, podemos executar um loop `while` até que `'cat'` não esteja mais na lista, conforme mostrado a seguir:

pets.py

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

Começamos com uma lista com diversas instâncias de `'cat'`. Depois de exibir a lista, o Python entra no loop `while` porque encontra o valor `'cat'` na lista pelo menos uma vez. Uma vez dentro do loop, o Python remove a primeira instância de `'cat'`, retorna à linha `while` e entra

novamente no loop quando identifica que 'cat' ainda está na lista. O Python remove cada instância de 'cat' até que o valor não esteja mais na lista, ponto em que o sai do loop e exibe a lista mais uma vez:

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']
```

Preenchendo um dicionário com entrada do usuário

Podemos solicitar em um prompt quantas entradas forem necessárias em cada passagem por um loop while. Criaremos um programa com um pesquisa, em que cada passagem pelo loop solicita o nome e a resposta do participante. Armazenaremos os dados que coletamos em um dicionário, pois queremos vincular cada resposta a um usuário específico:

mountain_poll.py

```
responses = {}
# Define uma flag para sinalizar que a pesquisa está ativa
polling_active = True

while polling_active:
    # Solicita o nome e a resposta do participante
    name = input("\nWhat is your name? ")
    response = input("Which mountain would you like to climb someday? ")

    # Armazena a resposta no dicionário
    responses[name] = response

    # Detecta se mais alguém participará da pesquisa
    repeat = input("Would you like to let another person respond? (yes/ no) ")
    if repeat == 'no':
        polling_active = False

# A pesquisa está completa. Mostra os resultados.
print("\n--- Poll Results ---")
for name, response in responses.items():
    print(f"{name} would like to climb {response}.")
```

De início, o programa define um dicionário vazio (responses) e define uma flag (polling_active) a fim de sinalizar que a pesquisa está ativa. Enquanto polling_active for True, o Python executará o código no loop

while.

Dentro do loop, o usuário é solicitado a inserir seu nome e uma montanha que gostaria de escalar 1. Essa informação é armazenada no dicionário responses 2, e se pergunta ao usuário se deseja ou não continuar respondendo a pesquisa 3. Se a resposta for yes, o programa entrará no loop while novamente. Se a resposta for no, a flag polling_active é definida como False, o loop while interrompe sua execução e o bloco de código final 4 exibe os resultados da pesquisa.

Se executarmos esse programa e inserirmos exemplos de respostas, veremos uma saída mais ou menos assim:

```
What is your name? Eric
Which mountain would you like to climb someday? Denali
Would you like to let another person respond? (yes/ no) yes

What is your name? Lynn
Which mountain would you like to climb someday? Devil's Thumb
Would you like to let another person respond? (yes/ no) no

--- Poll Results ---
Eric would like to climb Denali.
Lynn would like to climb Devil's Thumb.
```

FAÇA VOCÊ MESMO

7.8 Lanchonete: Crie uma lista chamada sandwich_orders e a preencha com o nome de diversos sanduíches. Depois, crie uma lista vazia chamada finished_sandwiches. Percorra a lista de pedidos de sanduíches com um loop e exiba uma mensagem para cada pedido, como: Seu lanche de atum está pronto. Conforme cada sanduíche é preparado, passe-os para a lista de sanduíches prontos. Após todos os sanduíches terem sido preparados, exiba uma mensagem enumerando cada um deles.

7.9 Sem pastrami: Usando a lista sandwich_orders do Exercício 7.8, assegure-se de que o sanduíche 'pastrami' apareça na lista pelo menos três vezes. Faça mais um código perto do início de seu programa, exibindo uma mensagem que informe que a lanchonete está sem pastrami e, em seguida, use um loop while para remover todas as ocorrências de 'pastrami' em sandwich_orders. Faça questão de que nenhum sanduíche de pastrami acabe em finished_sandwiches.

7.10 Férias tão sonhadas: Crie uma pesquisa que pergunte aos usuários sobre as férias de seus sonhos. Crie um prompt mais ou *menos assim: Se pudesse visitar qualquer lugar do mundo, para onde iria?* Inclua um bloco de código que exiba os resultados dessa pesquisa.

Recapitulando

Neste capítulo, aprendemos como utilizar a função input() para possibilitar que os usuários forneçam as próprias informações em seus programas. Aprendemos a trabalhar com entrada de texto e numérica e como usar loops while para fazer os programas executarem pelo tempo que usuários desejarem. Vimos diversas formas de controlar o fluxo de um loop while definindo uma flag active, usando a instrução break e a continue. Aprendemos como usar um loop while para transferir elementos de uma lista para outra e como remover todas as instâncias de um valor de uma lista e também como usar loops while com dicionários.

No Capítulo 8 aprenderemos o que são as funções. As *funções* possibilitam dividir os programas em pequenas partes, e cada uma delas realiza uma tarefa específica. É possível chamar uma função quantas vezes for necessário e armazená-las em arquivos separados. Com as funções podemos desenvolver códigos mais efetivos com facilidade de solução de problemas e manutenção e que podem ser reutilizado em muitos programas diferentes.

CAPÍTULO 8
Funções

Neste capítulo, aprenderemos como escrever *funções*, blocos nomeados de código arquitetados para realizar uma tarefa específica. Quando queremos executar uma tarefa específica e definida em uma função, *chamamos* a função responsável por ela. Se for necessário executar uma tarefa diversas vezes em todas as partes de seu programa, você não precisará digitar todo o código para a mesma tarefa repetidas vezes; basta chamar a função destinada a lidar com essa tarefa, e a chamada informa ao Python para executar o código dentro da função. Com o tempo, você perceberá que as funções facilitam a escrita, a legibilidade, o teste e a correção de seus programas.

Neste capítulo, aprenderemos também diversas formas de passar informações às funções. Veremos como escrever determinadas funções cuja tarefa primordial é exibir informações e outras funções arquitetadas para processar dados e retornar um valor ou conjunto de valores. Por último, aprenderemos a armazenar funções em arquivos separados chamados *módulos* com o intuito de ajudar a organizar os principais arquivos de seu programa.

Definindo uma função

Vejamos uma simples função chamada `greet_user()` que exibe um cumprimento:

greeter.py

```
def greet_user():
    """Exibe um simples cumprimento"""
```

```
    print("Hello!")

greet_user()
```

Esse exemplo apresenta a estrutura mais simples de uma função. A primeira linha utiliza a palavra reservada `def` para informar ao Python que estamos definindo uma função. Trata-se da *definição da função*, que informa ao Python o nome da função e, se for o caso, que tipo de informação a função precisa para realizar sua tarefa. Os parênteses armazenam essas informações. Nesse caso, o nome da função é `greet_user()`, e como não precisa de informações para realizar sua tarefa, seus parênteses estão vazios. (Mesmo assim, os parênteses são necessários.) Por último, a definição termina em dois-pontos.

Quaisquer linhas indentadas depois de `def greet_user()` constituem o *corpo* da função. Na segunda linha, o texto é um comentário chamado *docstring*, que explica o que a função faz. Ao gerar documentação para as funções em seus programas, o Python procura uma string logo após a definição da função. Em geral, essas strings são inseridas entre aspas triplas, possibilitando escrever múltiplas linhas.

A linha `print("Hello!")` é a única linha de código concreto no corpo dessa função. Logo, `greet_user()` tem somente uma tarefa: `print("Hello!")`.

E para usarmos essa função, devemos chamá-la. Uma *chamada de função* informa ao Python para executar o código na função. Para *chamar* uma função, escreva o nome da função, seguido por qualquer informação necessária entre parênteses. Aqui, como nenhuma informação é necessária, chamar nossa função é tão simples quanto inserir `greet_user()`. Como esperado, a função exibe `Hello!`:

```
Hello!
```

Passando informações para uma função

Quando ligeiramente modificada, a função `greet_user()` pode

cumprimentar o usuário pelo nome. Para que a função faça isso, insira username entre os parênteses da definição da função em def greet_user(). Quando adicionamos username aqui, possibilitamos que a função aceite qualquer valor especificado de username. Agora, a função espera que um valor seja fornecido para username sempre que for chamada. Ao chamar greet_user(), podemos lhe passar um nome, como 'jesse', entre parênteses:

```
def greet_user(username):
    """Exibe um simples cumprimento"""
    print(f"Hello, {username.title()}!")

greet_user('jesse')
```

A inserção de greet_user('jesse') chama greet_user() e fornece à função as informações necessárias para executar o print(). A função aceita o nome passado, exibindo o cumprimento para esse nome:

```
Hello, Jesse!
```

Da mesma forma, inserir greet_user('sarah') chama greet_user(), passa 'sarah' e exibe Hello, Sarah! Podemos chamar greet_user() e lhe passar qualquer nome quantas vezes quisermos para que a função gere a saída previsível todas as vezes.

Argumentos e parâmetros

Na função anterior greet_user(), definimos greet_user() a fim de exigir um valor para a variável username. Assim que chamamos a função e fornecemos as informações (o nome de uma pessoa), o cumprimento adequado foi exibido.

A variável username na definição de greet_user() é um exemplo de *parâmetro*, uma informação que a função precisa para realizar sua tarefa. O valor 'jesse' em greet_user('jesse') é um exemplo de argumento. Um *argumento* é uma informação passada de uma chamada de função para uma função. Ao chamarmos uma função, inserimos entre parênteses o valor com o qual queremos que a função trabalhe. Nesse caso, o argumento 'jesse' foi passado para a função greet_user(), e o valor foi atribuído ao parâmetro username.

NOTA *As pessoas às vezes falam de argumentos e parâmetros de forma intercambiável. Não se surpreenda se vir as variáveis de uma definição de função serem chamadas de argumentos ou as variáveis em uma chamada de função serem chamadas de parâmetros.*

FAÇA VOCÊ MESMO

8.1 Mensagem: Escreva uma função chamada display_message() que exiba uma frase contando a todo mundo o que você está aprendendo neste capítulo. Chame a função e verifique se a mensagem é adequadamente exibida.

8.2 Livro favorito: Escreva uma função chamada favorite_book() que aceite um parâmetro, title. A função deve exibir uma mensagem como: Um dos meus livros favoritos é Alice no País das Maravilhas. Chame a função e lembre-se de incluir o título de um livro como argumento na chamada da função.

Passando argumentos

Dado que uma definição de função pode ter diversos parâmetros, uma chamada de função pode precisar de vários argumentos. É possível passar argumentos para as funções de inúmeras formas. Pode-se utilizar *argumentos posicionais*, que precisam estar na mesma ordem em que os parâmetros foram escritos. Podemos usar *argumentos* nomeados, em que cada argumento é composto de um nome de variável e de um valor, e listas e dicionários de valores. Examinaremos cada um deles.

Argumentos posicionais

Ao chamarmos uma função, o Python verifica a correspondência de cada argumento na chamada da função com um parâmetro na definição da função. A forma mais simples de fazer isso é tomar como base a ordem dos argumentos fornecidos. Os valores desse tipo de correspondência são chamados de *argumentos posicionais*.

Vejamos como isso funciona. Imagine uma função que exibe informações sobre animais de estimação. A função nos informa o tipo de cada animal e o nome, conforme mostrado aqui:

pets.py

```
1 def describe_pet(animal_type, pet_name):
    """Exibe as informações sobre um animal de estimação"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

2 describe_pet('hamster', 'harry')
```

A definição revela que essa função precisa de um tipo de animal e seu nome `1`. Ao chamarmos `describe_pet()`, precisamos fornecer um tipo de animal e um nome, nessa mesma ordem. Por exemplo, na chamada de função, o argumento `'hamster'` é atribuído ao parâmetro `animal_type` e o argumento `'harry'` é atribuído ao parâmetro `pet_name` `2`. No corpo da função, esses dois parâmetros são utilizados para exibir informações sobre o animal de estimação que está sendo apresentado.

A saída exibe um hamster chamado Harry:

```
I have a hamster.
My hamster's name is Harry.
```

Múltiplas chamadas de função

É possível chamar uma função quantas vezes forem necessárias. Inserir um segundo animal de estimação diferente requer somente mais uma chamada de `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Exibe as informações sobre um animal de estimação"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

Nessa segunda chamada de função, passamos a `describe_pet()` os argumentos `'dog'` e `'willie'`. Como no conjunto anterior de argumentos que usamos, o Python verifica a correspondência de `'dog'` com o parâmetro `animal_type` e `'willie'` com o parâmetro `pet_name`. Como anteriormente, a função realiza sua tarefa, mas, desta vez, exibe valores para um cachorro chamado Willie. Agora, temos um hamster

chamado Harry e um cachorro chamado Willie:

```
I have a hamster.
My hamster's name is Harry.

I have a dog.
My dog's name is Willie.
```

Chamar uma função diversas vezes é uma forma ultraeficiente de trabalhar. Basta escrever uma vez na função o código que representa um animal de estimação. Desse modo, sempre que quiser representar um novo animal de estimação, chame a função com as informações do novo animal de estimação. Mesmo que o código que descreve um animal de estimação fosse incrementado em 10 linhas, ainda poderíamos representar um novo animal de estimação com apenas uma linha chamando a função novamente.

A ordem é importante em argumentos posicionais

Podemos obter resultados inesperados se misturarmos a ordem dos argumentos em uma chamada de função com argumentos posicionais:

```
def describe_pet(animal_type, pet_name):
    """Exibe as informações sobre um animal de estimação"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('harry', 'hamster')
```

Nesta chamada de função, enumeramos primeiro o nome e depois o tipo de animal. Dessa vez, como o argumento `'harry'` é enumerado primeiro, esse valor é atribuído ao parâmetro `animal_type`. Da mesma maneira, `'hamster'` é atribuído à `pet_name`. Agora, temos um "harry" chamado "Hamster":

```
I have a harry.
My harry's name is Hamster.
```

No caso de obter resultados estranhos como esse, confira se a ordem dos argumentos em sua chamada de função corresponde à ordem dos parâmetros na definição da função.

Argumentos nomeados

Um *argumento nomeado* é um par nome-valor que passamos para uma função. Como associamos diretamente o nome e o valor dentro do argumento, quando passamos o argumento para a função, não há confusão (não teremos um harry chamado Hamster). Graças aos argumentos nomeados, não precisamos nos preocupar em ordenar corretamente os argumentos na chamada de função, já que o papel de cada valor na chamada de função fica explícito.

Vamos reescrever *pets.py* com argumentos nomeados para chamar describe_pet():

```
def describe_pet(animal_type, pet_name):
    """Exibe as informações sobre um animal de estimação"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(animal_type='hamster', pet_name='harry')
```

A função describe_pet() não mudou. Mas quando chamamos a função, informamos explicitamente ao Python a correspondência entre cada parâmetro e cada argumento. Ao ler a chamada de função, o Python já sabe atribuir o argumento 'hamster' ao parâmetro animal_type e o argumento 'harry' a pet_name. A saída mostra corretamente que temos um hamster chamado Harry.

A ordem dos argumentos nomeados não é importante porque o Python sabe onde cada valor deve ir. As duas chamadas de função correspondem à:

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

NOTA *Ao usar argumentos nomeados, não se esqueça de usar os nomes exatos dos parâmetros na definição da função.*

Valores default

Ao escrevermos uma função, podemos definir um *valor default* para cada parâmetro. Se um argumento para um parâmetro for fornecido

na chamada da função, o Python usará o valor do argumento. Caso contrário, utilizará o valor default do parâmetro. Portanto, ao definirmos um valor default para um parâmetro, podemos descartar o argumento correspondente que normalmente escreveríamos na chamada da função. O uso de valores default pode simplificar suas chamadas de função e elucidar como suas funções são normalmente usadas.

Por exemplo, se constatar que a maioria das chamadas para describe_pet() estão sendo usadas para representar cachorros, você pode definir o valor default de animal_type como 'dog'. Agora, qualquer um que chamar describe_pet() para um cachorro pode omitir essa informação:

```
def describe_pet(pet_name, animal_type='dog'):
    """Exibe informação sobre o animal de estimação"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(pet_name='willie')
```

Mudamos a definição de describe_pet() para incluir um valor default, 'dog', para animal_type. Agora, quando a função é chamada sem animal_type especificado, o Python sabe usar o valor 'dog' para esse parâmetro:

```
I have a dog.
My dog's name is Willie.
```

Repare que a ordem dos parâmetros na definição da função teve que ser alterada. Como o valor default torna desnecessário especificar uma animal como argumento, o único argumento restante na chamada da função é o nome do animal de estimação. O Python ainda interpreta o valor como um argumento posicional. Ou seja, se a função for chamada apenas com o nome de um animal de estimação, esse argumento fará correspondência com primeiro parâmetro enumerado na definição da função. Por isso, é necessário que o primeiro parâmetro seja pet_name.

Agora, a forma mais simples de usar essa função é fornecer apenas

o nome de um cachorro na chamada da função:

```
describe_pet('willie')
```

Essa chamada de função teria a mesma saída do exemplo anterior. Como o único argumento fornecido é 'willie', a correspondência é feita com o primeiro parâmetro na definição, pet_name. E como nenhum argumento é fornecido a animal_type, o Python usa o valor default 'dog'.

Para representar um animal que não seja um cachorro, podemos usar uma chamada de função como esta:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Como um argumento explícito para animal_type é fornecido, o Python ignorará o valor default do parâmetro.

> **NOTA** *Qualquer parâmetro com valores default precisa ser enumerado após todos os parâmetros que não têm valores default. Isso possibilita que Python continue interpretando corretamente os argumentos posicionais.*

Chamadas de função equivalentes

Assim como argumentos posicionais, argumentos nomeados e valores default podem ser usados juntos. Não raro, recorreremos à diversas maneiras equivalentes para chamar uma função. Vejamos a seguinte definição para describe_pet() com um valor default fornecido:

```
def describe_pet(pet_name, animal_type='dog'):
```

Com essa definição, um argumento sempre precisa ser fornecido para pet_name, e esse valor pode ser fornecido usando o formato posicional ou nomeado. Se o animal que está sendo exibido não for um cachorro, devemos incluir um argumento para animal_type na chamada e esse argumento também pode ser especificado usando o formato posicional ou nominal.

Todas as chamadas a seguir funcionariam com essa função:

```
# Um cachorro chamado Willie
describe_pet('willie')
describe_pet(pet_name='willie')
```

```
# Um hamster chamado Harry
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

Cada uma dessas chamadas de função teria a mesma saída dos exemplos anteriores.

O estilo usado de chamada não importa. Contanto que suas chamadas de função gerem a saída desejada, basta utilizar o estilo que achar mais fácil de entender.

Evitando erros de argumento

Ao começar a usar funções, não fique surpreso se encontrar erros de argumentos sem correspondência. Podemos nos deparar com argumentos sem correspondência quando fornecemos argumentos de mais ou de menos do que uma função precisa para realizar sua tarefa. Por exemplo, vejamos o que ocorre se tentarmos chamar `describe_pet()` sem argumentos:

```
def describe_pet(animal_type, pet_name):
    """Exibe as informações sobre um animal de estimação"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet()
```

O Python reconhece que algumas informações estão faltando na chamada da função e o traceback nos informa que:

```
Traceback (most recent call last):
1  File "pets.py", line 6, in <module>
2    describe_pet()
     ^^^^^^^^^^^^^^
3 TypeError: describe_pet() missing 2 required positional arguments:
   'animal_type' and 'pet_name'
```

Primeiro, o traceback indica a localização do problema `1`, possibilitando-nos reaver e verificar o que deu algo errado com nossa chamada de função. Em seguida, a chamada de função responsável pelo erro é exibida `2`. Por último, o traceback informa que faltam dois argumentos na chamada e apresenta o nome dos

argumentos ausentes 3. Caso essa função estivesse em um arquivo separado, poderíamos possivelmente reescrever a chamada de maneira adequada, sem precisar abrir esse arquivo e ler o código da função.

O Python é tão bom que lê o código da função para nós e nos informa os nomes dos argumentos que precisamos fornecer. Ou seja, mais um incentivo para que você nomeie suas variáveis e funções com nomes descritivos. Caso adote essa prática, as mensagens de erro do Python podem não apenas ajudá-lo como também ajudar outra pessoa que use seu código.

Caso forneça um número excessivo de argumentos, você receberá um traceback parecido que pode ajudá-lo a fazer a correspondência adequada entre a chamada de função e a definição de função.

FAÇA VOCÊ MESMO

8.3 Camiseta: Crie uma função chamada make_shirt() que aceita um tamanho e o texto que deve ser estampado na camiseta. A função deve exibir uma frase resumindo o tamanho da camiseta e a mensagem estampada nela.

Chame a função uma vez usando argumentos posicionais para criar uma camiseta. Chame a função uma segunda vez usando argumentos nomeados.

8.4 Camisetas grandes: Modifique a função make_shirt() para que as camisetas sejam grandes por padrão com a seguinte frase estampada: *Eu amo Python*. Escreva uma camiseta grande e uma média com a mensagem padrão e uma camiseta de qualquer tamanho com uma frase diferente.

8.5 Cidades: Escreva uma função chamada describe_city() que aceite o nome de uma cidade e de seu país. A função deve exibir uma simples frase, como Reykjavik fica na Islândia. Forneça ao parâmetro do país um valor default. Chame sua função para três cidades diferentes e, pelo menos, para uma que não esteja no país default.

Valores de retorno

Nem sempre uma função precisa exibir sua saída diretamente. Em vez disso, pode processar alguns dados e retornar um valor ou conjunto de valores. O valor que a função retorna se chama *valor de retorno*. A instrução return recebe um valor de dentro de uma função e o reenvia para a linha que chamou a função. Os valores de retorno

possibilitam transferir grande parte do trabalho repetitivo às funções, simplificando assim o corpo do seu programa.

Retornando um valor simples

Vejamos uma função que recebe um nome e um sobrenome e retorna um nome completo formatado elegantemente:

formatted_name.py

```
def get_formatted_name(first_name, last_name):
    """Retorna um nome completo, elegantemente formatado"""
1   full_name = f"{first_name} {last_name}"
2   return full_name.title()

3 musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

A definição de `get_formatted_name()` recebe como parâmetros um nome e sobrenome. A função combina esses dois nomes, adiciona um espaço entre eles e atribui o resultado a `full_name` 1. O valor de `full_name` é convertido em letras iniciais maiúsculas e, em seguida, retornado à linha de chamada 2.

Ao chamar uma função que retorna um valor, é necessário fornecer uma variável à qual o valor de retorno possa ser atribuído. Nesse caso, o valor retornado é atribuído à variável `musician` 3. A saída apresenta um nome elegantemente formatado, com as partes do nome de uma pessoa:

```
Jimi Hendrix
```

Talvez pareça trabalhoso receber um nome elegantemente formatado quando poderíamos apenas escrever:

```
print("Jimi Hendrix")
```

No entanto, quando formos trabalhar com programas grandes, em que é necessário armazenar muitos nomes e sobrenomes separadamente, funções como `get_formatted_name()` ajudam muito. Primeiro, basta armazenar nomes e sobrenomes separadamente e, em seguida, chamar essa função sempre que quiser exibir um nome

completo.

Definindo um argumento como opcional

Não raro, faz sentido definir um argumento como opcional. Assim, as pessoas que usarem a função podem optar por fornecer informações extras, se quiserem. É possível utilizar valores default para definir um argumento como opcional.

Por exemplo, digamos que queremos incrementar `get_formatted_name()` para lidar também com nomes do meio. Vejamos uma primeira tentativa de incluir nomes do meio:

```
def get_formatted_name(first_name, middle_name, last_name):
    """Retorna um nome completo, formatado ordenadamente"""
    full_name = f"{first_name} {middle_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

Essa função executa quando recebe um nome, nome do meio e sobrenome. A função recebe todas as três partes de um nome e, em seguida, cria uma string com essas partes. A função adiciona espaços sempre que adequado e converte o nome completo com iniciais maiúsculas:

```
John Lee Hooker
```

Apesar disso, nomes do meio nem sempre são necessários, e do jeito que está essa função não executaria se tentássemos chamá-la apenas com nome e sobrenome. Para definir o nome do meio como opcional, podemos atribuir ao argumento `middle_name` um valor default vazio e desconsiderá-lo, a menos que o usuário forneça um valor. Para fazer `get_formatted_name()` executar sem um nome do meio, definimos o valor default de `middle_name` com uma string vazia e o passamos para o final da lista de parâmetros:

```
def get_formatted_name(first_name, last_name, middle_name=''):
    """Retorna um nome completo, formatado elegantemente"""
1   if middle_name:
        full_name = f"{first_name} {middle_name} {last_name}"
```

```
2     else:
          full_name = f"{first_name} {last_name}"
      return full_name.title()

  musician = get_formatted_name('jimi', 'hendrix')
  print(musician)

3 musician = get_formatted_name('john', 'hooker', 'lee')
  print(musician)
```

Nesse exemplo, o nome é criado a partir de três partes possíveis. Como sempre há um nome seguido do primeiro sobrenome, esses parâmetros são enumerados primeiro na definição da função. Como é opcional, nome do meio é enumerado por último na definição e seu valor default é uma string vazia.

No corpo da função, verificamos se um nome do meio foi fornecido. O Python interpreta strings não vazias como `True`, logo, o teste condicional `if middle_name` avalia como `True` se um argumento de nome do meio estiver na chamada de função `1`. Se um nome do meio for fornecido, o nome, o nome do meio e o sobrenome serão combinados para formar um nome completo. Em seguida, esse nome é alterado em letras iniciais maiúsculas e retornado à linha da chamada de função, onde é atribuído à variável `musician` e exibido. Se nenhum nome do meio for fornecido, a string vazia falha no teste `if` e o bloco `else` é executado `2`. O nome completo é constituído com somente o primeiro nome e sobrenome, e o nome formatado é retornado à linha de chamada onde é atribuído à `musician` e exibido.

Chamar essa função com o primeiro nome e sobrenome é fácil. No entanto, se estivermos usando um nome do meio, temos que garantir que o nome do meio seja o último argumento passado. Desse modo, o Python faz a correspondência dos argumentos posicionais devidamente `3`.

Essa versão modificada da nossa função funciona para pessoas com apenas um nome e sobrenome, e também para pessoas que têm um nome do meio:

```
Jimi Hendrix
John Lee Hooker
```

Os valores opcionais possibilitam que as funções lidem com uma ampla variedade de casos de uso, ao mesmo tempo em que possibilitam simplificar ao máximo as chamadas de função.

Retornando um dicionário

Uma função pode retornar qualquer tipo de valor que quisermos, incluindo estruturas de dados mais complicadas, como listas e dicionários. Por exemplo, a função a seguir recebe partes de um nome e retorna um dicionário que representa uma pessoa:

person.py

```
def build_person(first_name, last_name):
    """Retorna um dicionário de informações sobre uma pessoa"""
1   person = {'first': first_name, 'last': last_name}
2   return person

musician = build_person('jimi', 'hendrix')
3 print(musician)
```

A função `build_person()` recebe o primeiro nome e o sobrenome e insere esses valores em um dicionário `1`. O valor de `first_name` é armazenado com a chave `'first'`, e o valor de `last_name` é armazenado com a chave `'last'`. Em seguida, todo o dicionário que representa a pessoa é retornado `2`. Agora, o valor de retorno é exibido `3` com as duas informações textuais originais, armazenadas em um dicionário:

```
{'first': 'jimi', 'last': 'hendrix'}
```

Essa função recebe informações textuais simples e as insere em uma estrutura de dados mais expressiva que possibilita trabalhar com as informações além de apenas exibi-las.

Agora, as strings `'jimi'` e `'hendrix'` são representadas como primeiro nome e sobrenome. É possível incrementar facilmente essa função para aceitar valores opcionais como um nome do meio, uma idade, uma atividade profissional ou qualquer outra informação que quisermos armazenar sobre uma pessoa. Por exemplo, a alteração a

seguir nos possibilita armazenar também a idade de uma pessoa:

```
def build_person(first_name, last_name, age=None):
    """Retorna um dicionário de informações sobre uma pessoa"""
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

Adicionamos um novo parâmetro opcional `age` à definição da função e atribuímos ao parâmetro o valor especial `None`, usado quando uma variável não tem um valor específico atribuído a ela. Talvez você tenha a impressão de que `None` é um valor de placeholder (marcador de posição). Nos testes condicionais, `None` é avaliado como `False`. Caso a chamada de função inclua um valor para `age`, esse valor será armazenado no dicionário. Essa função sempre armazena o nome de uma pessoa, mas também pode ser modificada para armazenar qualquer outra informação que quisermos sobre uma pessoa.

Usando uma função com um loop while

É possível usar funções com todas as estruturas Python que aprendemos até o momento. Por exemplo, vejamos o uso da função `get_formatted_name()` com um loop `while` para cumprimentar os usuários mais formalmente. Observe a primeira tentativa de cumprimentar as pessoas com seu primeiro nome e sobrenome:

greeter.py

```
def get_formatted_name(first_name, last_name):
    """Retorna um nome completo, formatado elegantemente"""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# Temos um loop infinito aqui!
while True:
    print("\nPlease tell me your name:")
    f_name = input("First name: ")
    l_name = input("Last name: ")
```

```
    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

Nesse exemplo, usamos uma versão simples da `get_formatted_name()` que não tem nomes do meio. O loop `while` pede que o usuário insira seu nome e solicitamos seu primeiro nome e sobrenome separadamente 1.

Mas esse loop `while` tem um problema: não definimos uma condição de saída. Onde inserimos uma condição de saída quando solicitamos uma série de entradas? Queremos que o usuário consiga encerrar o loop de forma mais fácil possível. Ou seja, cada prompt deve disponibilizar uma condição de saída. A instrução `break` viabiliza uma maneira descomplicada de sair do loop em qualquer um dos prompts:

```
def get_formatted_name(first_name, last_name):
    """Retorna um nome completo, formatado elegantemente"""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
        break

    l_name = input("Last name: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

Adicionamos uma mensagem informando ao usuário como sair e, em seguida, saímos do loop se o usuário inserir o valor de saída em qualquer prompt. Agora, o programa continuará cumprimentando as pessoas até que alguém digite `q` em qualquer um dos nomes:

```
Please tell me your name:
(enter 'q' at any time to quit)
```

First name: **eric**
Last name: **matthes**

Hello, Eric Matthes!

Please tell me your name:
(enter 'q' at any time to quit)
First name: **q**

FAÇA VOCÊ MESMO

8.6 Nome de cidades: Escreva uma função chamada city_country() que recebe o nome de uma cidade e seu país. A função deve retornar uma string formatada como esta:

"Santiago, Chile"

Chame sua função com pelo menos três pares cidade-país e exiba os valores retornados.

8.7 Álbum: Escreva uma função chamada make_album() que crie um dicionário representando um álbum de música. A função deve ter o nome de um artista e o título de álbum, e deve retornar um dicionário com essas duas informações. Utilize a função para criar três dicionários representando álbuns distintos. Exiba cada valor de retorno para mostrar que os dicionários estão armazenando adequadamente as informações do álbum.

Use None para adicionar um parâmetro opcional ao make_album() que possibilite armazenar o número de músicas em um álbum. Se a linha chamadora incluir um valor para o número de músicas, adicione esse valor ao dicionário do álbum. Crie, pelo menos, uma nova chamada de função que inclua o número de músicas em um álbum.

8.8 Álbuns de usuários: Comece com seu programa do Exercício 8.7. Escreva um loop while que possibilite aos usuários inserir o artista e o título de um álbum. Após receber essas informações, chame make_album() com a entrada do usuário e exiba o dicionário criado. Não se esqueça de incluir um valor de saída no loop while.

Passando uma lista

Em geral, é conveniente passar uma lista para uma função, seja uma lista de nomes, números ou objetos mais complexos, como dicionários. Ao passarmos uma lista para uma função, a função obtém acesso direto ao conteúdo da lista. Vamos utilizar funções para que as tarefas com listas fiquem mais eficientes.

Digamos que temos uma lista de usuários e queremos exibir uma mensagem de boas-vindas a cada um. O exemplo a seguir envia uma lista de nomes para uma função chamada greet_users(), que

cumprimenta cada pessoa na lista individualmente:

greet_users.py

```
def greet_users(names):
    """Exibe um simples hello para cada usuário na lista"""
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)
usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

Definimos `greet_users()` para que espere uma lista de nomes, atribuída ao parâmetro `name`. A função percorre a lista que recebe com um loop e exibe um cumprimento a cada usuário. Fora da função, definimos uma lista de usuários e, em seguida, passamos a lista `usernames` para `greet_users()` na chamada de função:

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

Essa é a saída que queríamos. Cada usuário vê um cumprimento personalizado, podemos chamar a função sempre que quisermos cumprimentar um conjunto específico de usuários.

Modificando uma lista em uma função

Ao passarmos uma lista para uma função, a função pode modificar essa lista. Quaisquer mudanças feitas na lista dentro do corpo da função são definitivas, possibilitando programar de forma eficiente, mesmo quando estivermos lidando com volumes grandes de dados.

Imagine uma empresa que cria modelos impressos em 3D de designs que os usuários enviam. Os designs que precisam ser impressos são armazenados em uma lista. Após impressos, são passados para uma lista separada. O código a seguir faz justamente isso sem funções:

printing_models.py

```
# Começa com alguns designs que precisam ser impressos.
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
```

```
completed_models = []

# Simula a impressão de cada design, até que não reste nenhum
# Passa cada design para completed_models após impressão
while unprinted_designs:
    current_design = unprinted_designs.pop()
    print(f"Printing model: {current_design}")
    completed_models.append(current_design)

# Exibe todos os modelos concluídos
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)
```

O programa começa com uma lista de designs que precisam ser impressos e uma lista vazia chamada completed_models, à qual cada design será passado após impresso. Desde que os designs permaneçam em unprinted_designs, o loop while simula a impressão de cada um deles, removendo um design do final da lista, armazenando-o em current_design e exibindo uma mensagem de que o design atual está sendo impresso. Em seguida, o design é adicionado à lista de modelos concluídos. Quando o loop terminar de ser executado, o programa exibe uma lista dos designs impressos:

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case

The following models have been printed:
dodecahedron
robot pendant
phone case
```

É possível reorganizar esse código escrevendo duas funções para que cada uma delas execute uma tarefa específica. Não alteraremos a maior parte do código; estamos apenas estruturando-o com mais cuidado. A primeira função lidará com a impressão dos designs e a segunda sintetizará as impressões feitas:

```
1 def print_models(unprinted_designs, completed_models):
    """
    Simula a impressão de cada design, até que não reste nenhum.
    Passa cada design para completed_models após impressão.
```

```
    """
    while unprinted_designs:
        current_design = unprinted_designs.pop()
        print(f"Printing model: {current_design}")
        completed_models.append(current_design)

2 def show_completed_models(completed_models):
    """ Exibe todos os modelos impressos"""
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)

unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Definimos a função print_models() com dois parâmetros: uma lista de designs que precisam ser impressos e uma lista de modelos concluídos 1. Dadas essas duas listas, a função simula a impressão de cada design, deixando a lista de designs não impressos vazia e preenchendo a lista de modelos concluídos. Em seguida, definimos a função show_completed_models() com um parâmetro: a lista de modelos concluídos 2. Com essa lista, o show_completed_models() exibe o nome de cada modelo impresso.

Apesar de esse programa ter a mesma saída que a versão sem funções, o código fica mais estruturado. O código que executa a maior parte da tarefa foi passado para duas funções separadas, facilitando o entendimento da parte principal do programa. Observe o corpo do programa e veja como é mais fácil entender o que está acontecendo:

```
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Definimos uma lista de designs não impressos e uma lista vazia que armazenará os modelos concluídos. Depois, como já definimos

nossas duas funções, tudo o que temos a fazer é chamá-las e passar os argumentos adequados às duas. Chamamos print_models() e passamos as duas listas necessárias a essa função; como esperado, print_models() simula a impressão dos designs. Em seguida, chamamos show_completed_models() e passamos a lista de modelos concluídos para que a função possa informar os modelos impressos. Os nomes descritivos das funções facilitam com que outras pessoas leiam e compreendem esse código, mesmo sem comentários.

É mais fácil incrementar e manter esse programa do que a versão sem funções. No caso de precisarmos imprimir mais designs posteriormente, basta chamar print_models() de novo. Se percebermos que é necessário modificar o código de impressão, podemos alterá-lo apenas uma vez, e nossas alterações serão reproduzidas em todos os trechos em que a função for chamada. Trata-se de uma técnica mais eficiente do que ter que atualizar o código separadamente em todo o programa.

Nesse exemplo, observamos também a ideia de que cada função deve ter uma tarefa específica. A primeira função imprime cada design e a segunda exibe os modelos concluídos. Isso é mais prático do que usar uma função para executar ambas as tarefas. Se estiver escrevendo uma função e perceber que a função está executando muitas tarefas diferentes, tente dividir o código em duas funções. Lembre-se de que sempre podemos chamar uma função a partir de outra função, e isso é vantajoso quando dividimos uma tarefa complexa em uma série de etapas.

Evitando que uma função modifique uma lista

Às vezes, queremos evitar que uma função modifique uma lista. Por exemplo, digamos que você comece com uma lista de designs não impressos e escreva uma função que os passem para uma lista de modelos concluídos, como no exemplo anterior. Talvez você decida que, mesmo tendo impresso todos os designs, quer manter a lista original de designs não impressos em seus registros. No entanto,

como você transferiu todos os nomes de design de unprinted_designs, a lista agora está vazia. E essa lista vazia é a única versão que você tem, já que a lista original se perdeu. Nesse caso, podemos solucionar esse problema passando à função uma cópia da lista, não a original. Quaisquer alterações que a função faça na lista afetarão somente a cópia, deixando a lista original intacta.

Podemos enviar a cópia de uma lista para uma função:

```
nome_função(nome_lista[:])
```

A notação de fatia [:] faz uma cópia da lista para enviar à função. Caso não quiséssemos deixar a lista de designs não impressos vazia em *printing_models.py*, poderíamos chamar print_models() assim:

```
print_models(unprinted_designs[:], completed_models)
```

A função print_models() consegue executar sua tarefa, pois ainda recebe o nome de todos os designs não impressos. Mas, desta vez, a função usa uma cópia da lista de designs originais não impressos, não a lista real unprinted_designs. Como antes, a lista completed_models será preenchida com o nome dos modelos impressos, mas a lista original de designs não impressos não será afetada pela função.

Mesmo que possamos preservar o conteúdo de uma lista passando uma cópia dela às nossas funções, devemos passar a lista original para as funções, a menos que tenhamos um motivo especial para passar uma cópia. Uma função consegue executar melhor uma tarefa com uma lista existente, já que assim evita usar desnecessariamente tempo e memória exigidos para fazer uma cópia separada, ainda mais quando se trabalha com listas grandes.

FAÇA VOCÊ MESMO

8.9 Mensagens: Crie uma lista com uma série de mensagens curtas de texto. Passe a lista para uma função chamada show_messages(), que exiba cada mensagem de texto.

8.10 Enviando mensagens: Comece com uma cópia do seu programa do Exercício 8.9. Escreva uma função chamada send_messages() para exibir cada mensagem de texto e passe cada mensagem para uma nova lista chamada sent_messages à medida que é exibida. Após chamar a função, exiba ambas as listas para ter certeza de que as mensagens foram corretamente transferidas.

8.11 Mensagens arquivadas: Comece sua tarefa a partir do Exercício 8.10. Chame a função send_messages() com uma cópia da lista de mensagens. Após chamar a função, exiba ambas as listas para mostrar que a lista original reteve suas mensagens.

Passando um número arbitrário de argumentos

Às vezes, não sabemos com antecedência quantos argumentos uma função precisa aceitar. Felizmente, o Python possibilita que uma função colete um número arbitrário de argumentos a partir da instrução de chamada.

Por exemplo, imagine uma função que cria uma pizza. É necessário que essa função aceite uma série de ingredientes, mas não sabemos com antecedência quantos ingredientes uma pessoa vai pedir. No exemplo a seguir, a função tem um parâmetro, `*toppings`, que coleta todos os argumentos fornecidos na linha de chamada:

pizza.py

```
def make_pizza(*toppings):
    """Exibe a lista de ingredientes solicitados"""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

O asterisco no nome do parâmetro `*toppings` informa ao Python para criar uma tupla chamada `toppings`, contendo todos os valores que essa função recebe. O `print()` no corpo da função gera a saída mostrando que o Python pode lidar com uma chamada de função de um valor e com uma chamada de três valores. Chamadas diferentes são tratadas de forma parecida. Observe que o Python empacota os argumentos em uma tupla, mesmo que a função receba somente um valor:

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Agora, podemos substituir a chamada `print()` por um loop que percorre a lista de ingredientes e apresenta a pizza que está sendo

pedida:

```
def make_pizza(*toppings):
    """Sintetiza a pizza que estamos prestes a fazer"""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

A função responde adequadamente, quer receba um valor ou três valores:

```
Making a pizza with the following toppings:
- pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Essa sintaxe funciona, independentemente de quantos argumentos a função recebe.

Misturando argumentos posicionais e arbitrários

Se quisermos que uma função aceite diversos tipos diferentes de argumentos, devemos inserir por último o parâmetro que aceita um número arbitrário de argumentos na definição da função. Primeiro, o Python faz a correspondência de argumentos posicionais e nomeados e, em seguida, coleta todos os argumentos restantes no parâmetro final.

Por exemplo, se a função precisar receber um tamanho para a pizza, esse parâmetro deve ficar antes do parâmetro `toppings`:

```
def make_pizza(size, *toppings):
    """ Sintetiza a pizza que estamos prestes a fazer"""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Na definição da função, o Python atribui o primeiro valor que recebe ao parâmetro size. Todos os outros valores que vêm depois são armazenados na tupla toppings. Primeiro, as chamadas de função incluem um argumento para o tamanho, seguido de quantos ingredientes forem necessários.

Agora, cada pizza tem um tamanho e uma quantidade de ingredientes, e cada informação é exibida no local adequado, mostrando o tamanho primeiro e os ingredientes depois:

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

NOTA *Não raro, você verá o nome do parâmetro genérico* *args*, que coleta argumentos posicionais arbitrários como esse.*

Usando argumentos nomeados arbitrários

Vez ou outra queremos aceitar um número arbitrário de argumentos, mas não sabemos antes que tipo de informação será passado à função. Nesse caso, podemos escrever funções que aceitem a quantidade de pares chave-valor fornecidos na chamada. Vejamos um exemplo de criação de perfis de usuário: você sabe que obterá informações sobre um usuário, mas não sabe que tipo de informação receberá. No exemplo a seguir, a função build_profile() sempre recebe um primeiro nome e um sobrenome, mas também aceita um número arbitrário de argumentos nomeados:

user_profile.py

```
def build_profile(first, last, **user_info):
    """Cria um dicionário contendo tudo o que sabemos sobre um usuário"""
1    user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info
```

```
user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')
print(user_profile)
```

A definição de build_profile() espera um primeiro nome e sobrenome e, em seguida, possibilita que o usuário passe quantos pares nome-valor quiser. Os asteriscos duplos antes do parâmetro * user_info fazem com que o Python crie um dicionário chamado user_info contendo todos os pares de nome-valor extras que a função recebe. Dentro da função podemos acessar os pares chave-valor em user_info, como acessaríamos em qualquer dicionário.

No corpo do build_profile() adicionamos o primeiro nome e sobrenome ao dicionário user_info, pois sempre receberemos essas duas informações do usuário 1, mesmo que ainda não tenham sido inseridas no dicionário. Em seguida, retornamos o dicionário user_info à linha de chamada da função.

Chamamos build_profile(), passando o primeiro nome 'albert', o sobrenome 'einstein' e os dois pares chave-valor location='princeton' e field='physics'. Atribuímos o profile retornado a user_profile e o exibimos:

```
{'location': 'princeton', 'field': 'physics',
'first_name': 'albert', 'last_name': 'einstein'}
```

O dicionário retornado contém o primeiro nome e sobrenome do usuário e, nesse caso, a localização e também o campo de estudo. A função executará, independentemente de quantos pares adicionais chave-valor forem fornecidos na chamada de função.

Quando escrevemos nossas funções, podemos misturar valores posicionais, nomeados e arbitrários de muitas formas diferentes. É bom saber que esses tipos de argumentos existem, já que você os verá com frequência assim que começar a ler o código de outras pessoas. É necessário prática para usar corretamente os diferentes tipos e saber quando usar cada tipo. Por ora, lembre-se de usar a abordagem mais simples que executa as tarefas. À medida que progride, você aprenderá a utilizar sempre a abordagem mais

eficiente.

NOTA *Muitas vezes, você verá o nome do parâmetro* **kwargs *usado para coletar argumentos nomeados não específicos.*

FAÇA VOCÊ MESMO

8.12 Sanduíches: Crie uma função que aceite uma lista de itens que uma pessoa quer em um sanduíche. A função deve ter um parâmetro que colete todos os itens fornecidos na chamada de função e deve exibir um resumo do sanduíche que está sendo solicitado. Chame a função três vezes, com um número diferente de argumentos a cada vez.

8.13 Perfil de usuário: Comece com uma cópia do *user_profile.py* da página *194*. Crie um perfil de si mesmo chamando build_profile(), com seu primeiro nome e sobrenome e três outros pares chave-valor que o representem.

8.14 Carros: Crie uma função que armazena informações sobre um carro em um dicionário. A função deve sempre receber um fabricante e um nome de modelo. Em seguida, deve aceitar um número arbitrário de argumentos nomeados. Chame a função com as informações necessárias e dois outros pares nome-valor, como uma cor ou um recurso opcional. Sua função deve funcionar mais ou menos assim:

```
car = make_car('subaru', 'outback', color='blue', tow_package=True)
```

Exiba o dicionário retornado para garantir que todas as informações foram corretamente armazenadas.

Armazenando suas funções em módulos

Uma das vantagens das funções é o modo como separam blocos de código do programa principal. Ao usarmos nomes descritivos para as funções, fica mais fácil de entender nosso programa. Podemos ir mais além, armazenando as funções em um arquivo separado chamado módulo e, em seguida, *importando* esse *módulo* para o programa principal. Uma instrução import informa ao Python para disponibilizar o código em um módulo no arquivo de programa atualmente em execução.

Armazenar funções em um arquivo separado possibilita ocultar os detalhes do código do programa e focar a lógica de alto nível. Possibilita também reutilizar funções em muitos programas diferentes. Quando armazenamos funções em arquivos separados, podemos compartilhar esses arquivos com outros programadores

sem ter que compartilhar todo o programa. Saber importar funções também facilita usar as bibliotecas de funções que outros programadores desenvolveram.

Existem diversas formas de importar um módulo, veremos brevemente cada uma delas.

Importando um módulo inteiro

Para começar a importar funções, primeiro é necessário criar um módulo. Um *módulo* é um arquivo que termina em *.py*, contendo o código que queremos importar para o programa. Criaremos um módulo que contenha a função `make_pizza()`. A fim de criar esse módulo, removeremos tudo do *arquivo pizza.py*, exceto a função `make_pizza()`:

pizza.py

```
def make_pizza(size, *toppings):
    """Sintetiza a pizza que estamos prestes a fazer"""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
```

Agora, vamos criar um arquivo separado chamado *making_pizzas.py* no mesmo diretório que *pizza.py*. Esse arquivo importa o módulo que acabamos de criar e, em seguida, chama duas vezes `make_pizza()`:

making_pizzas.py

```
import pizza

pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Quando lê esse arquivo, a linha `import pizza` informa ao Python para abrir o arquivo *pizza.py* e copiar todas as funções dele para esse programa. Na verdade, não vemos o código sendo copiado entre arquivos, pois o Python copia o código nos bastidores, pouco antes da execução do programa. Tudo o que precisamos saber é que qualquer função definida no *pizza.py* agora estará disponível em *making_pizzas.py*.

Para chamar uma função de um módulo importado, digite o nome do módulo, pizza, seguido pelo nome da função, make_pizza(), separados por um ponto 1. O código a seguir gera a mesma saída que o programa original, que não importou um módulo:

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Essa abordagem inicial de importação, em que simplesmente escrevemos import seguido pelo nome do módulo, disponibiliza todas as funções do módulo no programa. Caso use esse tipo de instrução import para importar um módulo inteiro chamado *module_name.py*, cada função no módulo estará disponível por meio da seguinte sintaxe:

```
nome_módulo.nome_função()
```

Importando funções específicas

Podemos também importar uma função específica de um módulo. Vejamos a sintaxe geral dessa abordagem:

```
from nome_módulo import nome_função
```

É possível importar quantas funções quisermos de um módulo, separando o nome de cada função com uma vírgula:

```
from mnome_módulo import função_0, função_1, função_2
```

Se quiséssemos importar apenas a função que vamos usar, o exemplo *making_pizzas.py* ficaria assim:

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Com essa sintaxe, não é necessário usar a notação de ponto ao chamar uma função. Já que importamos explicitamente a função make_pizza() na instrução import, podemos chamá-la pelo nome quando

usamos a função.

Usando as para atribuir um alias a uma função

Caso o nome da função que está sendo importada possa entrar em conflito com um nome existente do programa ou caso o nome da função seja extenso, podemos utilizar um *alias* curto e exclusivo – um nome alternativo, semelhante a um apelido para a função. Atribuímos esse apelido especial à função ao importá-la.

Aqui, atribuímos à função `make_pizza` um alias, `mp()`, importando `make_pizza` como `mp()`. A palavra reservada `as` renomeia uma função usando o alias fornecido:

```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Nesse programa, a instrução `import` mostrada aqui renomeia a função `make_pizza()` para `mp()`. Sempre que quisermos chamar `make_pizza()`, basta escrever `mp()`, e o Python executará o código em `make_pizza()`, evitando qualquer confusão com outra função `make_pizza()` que possamos ter escrito no arquivo de programa.

Vejamos a sintaxe geral para fornecer um alias:

```
from nome_módulo import nome_função as nf
```

Usando as para atribuir um alias a um módulo

É possível também fornecer um alias para um nome de módulo. Atribuir a um módulo um alias curto, como `p` para `pizza`, possibilita chamar as funções do módulo mais rápido. Chamar `p.make_pizza()` é mais conciso do que chamar `pizza.make_pizza()`:

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

O módulo `pizza` recebe o alias `p` na instrução `import`, mas todas as funções do módulo mantêm seus nomes originais. Chamar as

funções escrevendo p.make_pizza() não é apenas mais conciso do que pizza.make_pizza(), como também redireciona sua atenção do nome do módulo e possibilita que você foque os nomes descritivos de suas funções. Esses nomes de função, que informam claramente o que cada função faz, são mais importantes para a legibilidade do código do que utilizar o nome completo do módulo.

Vejamos a sintaxe geral dessa abordagem:

```
import nome_módulo as nm
```

Importando todas as funções em um módulo

Podemos solicitar ao Python que importe todas as funções em um módulo com o operador asterisco (*):

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Na instrução import, o asterisco solicita ao Python que copie todas as funções do módulo pizza para esse arquivo de programa. Como cada função é importada, podemos chamar cada função pelo nome sem usar a notação de ponto. No entanto, é melhor não usar essa abordagem quando estivermos trabalhando com módulos maiores que não escrevemos: se o módulo tiver um nome de função que corresponda a um nome existente de um projeto, poderemos obter resultados inesperados. Pode ser que o Python veja diversas funções ou variáveis com o mesmo nome e, em vez de importar todas as funções separadamente, sobrescreverá as funções.

A melhor abordagem é importar a função ou funções desejadas, ou importar todo o módulo e usar a notação de ponto. Isso resulta em um código limpo, fácil de ler e entender. Incluí esta seção para que você reconheça instruções import quando as vir no código de outras pessoas:

```
from nome_módulo import *
```

Estilizando funções

É necessário que você se lembre de alguns detalhes quando estiver estilizando funções. Funções devem ter nomes descritivos, e esses nomes devem usar letras minúsculas e underscores. Os nomes descritivos nos ajudam a entender o que o código está tentando fazer. O nome dos módulos também devem seguir essas convenções.

Cada função deve ter um comentário que explique de forma concisa o que faz. Esse comentário deve aparecer logo após a definição da função e deve usar o formato docstring. Quando uma função está bem documentada, outros programadores podem usá-la lendo apenas a descrição na docstring. Os programadores também devem ser capazes de confiar que o código funciona conforme descrito e, desde que saibam o nome da função, os argumentos necessários e o tipo de valor retornado, eles devem conseguir usá-la em seus programas.

Caso especifique um valor default para um parâmetro, nenhum espaço deverá ser usado em ambos os lados do sinal de igual:

```
def nome_função(parâmetro_0, parâmetro_1='valor_default')
```

A mesma convenção deve ser utilizada para argumentos nomeados em chamadas de função:

```
fnome_função(valor_0, parâmetro_1='valor')
```

A PEP 8 (https://www.python.org/dev/peps/pep-0008) recomenda um limite de linhas de código com 79 caracteres para que cada linha fique visível em uma janela do editor de tamanho razoável. Se um conjunto de parâmetros fizer com que a definição de uma função tenha mais de 79 caracteres, pressione ENTER após o parêntese de abertura na linha de definição. Na próxima linha, pressione a tecla TAB duas vezes para separar a lista de argumentos do corpo da função, que só será indentada um nível.

A maioria dos editores alinha automaticamente quaisquer linhas

adicionais de argumentos para corresponder à indentação estabelecida na primeira linha:

```
def nome_função(
        parâmetro_0, parâmetro_1, parâmetro_2,
        parâmetro_3, parâmetro_4, parâmetro_5):
    corpo da função...
```

Se o programa ou módulo tiver mais de uma função, podemos separar cada uma por duas linhas em branco para facilitar a visualização de onde uma função termina e a próxima começa.

Todas as instruções import devem ser escritas no início de um arquivo. A única exceção é quando escrevemos comentários no início do arquivo para descrever o programa como um todo.

FAÇA VOCÊ MESMO

8.15 Imprimindo modelos: Insira as funções do *exemplo printing_models.py* em um arquivo separado chamado *printing_functions.py*. Escreva uma instrução import na parte superior do *printing_models.py* e modifique o arquivo para usar as funções importadas.

8.16 Imports: Usando um programa que você escreveu e que tenha apenas uma função, armazene essa função em um arquivo separado. Importe a função para o arquivo de programa principal e chame a função usando cada uma dessas abordagens:

```
import nome_módulo
from nome_módulo import nome_função
from nome_módulo import nome_função as nf
import nome_módulo as nm
from nome_módulo import *
```

8.17 Estilizando funções: Escolha os três programas que você escreveu para este capítulo e garanta que sigam as diretrizes de estilo descritas nesta seção.

Recapitulando

Neste capítulo, aprendemos a escrever funções e a passar argumentos, assim as funções têm acesso às informações de que precisam para realizar suas tarefas. Estudamos como usar argumentos posicionais e nomeados e também a aceitar um número arbitrário de argumentos. Vimos funções que exibem saída e que retornam valores, e como usar funções com listas, dicionários,

instruções if e loops while. Aprendemos também como armazenar funções em arquivos separados chamados *módulos*, com o intuito de que os arquivos de programa sejam mais simples e fáceis de entender. Por último, vimos como estilizar funções para que os programas fiquem mais estruturados, facilitando a legibilidade de todas as pessoas.

Um dos objetivos como programador é desenvolver um código simples que execute o que queremos. As funções nos ajudam com isso, já que possibilitam escrever blocos de códigos que funcionam para usá-los quando precisarmos. Quando sabemos que uma função executa adequadamente uma tarefa, podemos confiar que continuará funcionando e podemos passar para a próxima tarefa de programação.

Com as funções, temos a possibilidade de escrever o código uma vez e, depois, reutilizá-los quantas vezes bem entendermos. Se precisarmos executar o código em uma função, basta escrevermos uma chamada de linha e a função executa sua tarefa. Se precisarmos modificar o comportamento de uma função, basta modificar um bloco de código, e essa mudança passa a valer em todas as partes que chamamos essa função.

Funções facilitam a legibilidade dos programas, ao passo que bons nomes de função sintetizam o que cada parte de um programa faz. Para termos uma noção mais rápida do funcionamento de um programa, é melhor lermos uma série de chamadas de função do que uma série extensa de blocos de código.

Funções também facilitam o teste e a depuração do código. Quando você atribui a maior parte das tarefas do programa a um conjunto de funções, em que cada uma delas executa uma tarefa específica, fica mais fácil de testar e manter o código escrito. Ou seja, podemos escrever um programa separado para chamar e testar cada função em todas as situações que encontrarmos, pois teremos certeza de que executarão corretamente sempre que as chamarmos.

No Capítulo 9, vamos aprender a escrever classes. As *classes* combinam funções e dados em um pacote elegante que pode ser usado de formas flexíveis e eficientes.

CAPÍTULO 9
Classes

A *programação orientada a objetos (OOP)* é uma das abordagens mais eficazes para desenvolver software. Na programação orientada a objetos escrevemos *classes* que representam coisas e situações do mundo cotidiano, e criamos *objetos* tomando como base essas classes. Ao escrever uma classe, definimos o comportamento geral que uma categoria inteira de objetos pode ter.

Ao criarmos objetos individuais a partir de classes, cada objeto é automaticamente complementado com o comportamento geral; podemos assim atribuir a cada objeto quaisquer características especiais que quisermos. Não se espante com o nível de qualidade em que as situações do mundo cotidiano podem ser modeladas com a OOP.

A criação de um objeto a partir de uma classe se chama *instanciação*, e trabalhamos com as *instâncias* de uma classe. Neste capítulo, escreveremos classes e criaremos instâncias dessas classes. Vamos especificar o tipo de informação que pode ser armazenada em instâncias e vamos definir ações que podem ser executadas com essas instâncias. Escreveremos também classes que incrementam a funcionalidade das classes existentes, para que classes semelhantes possam compartilhar funcionalidades comuns e para que possamos fazer mais com menos código. Vamos armazenar as classes em módulos e vamos importar classes desenvolvidas por outros programadores em nossos arquivos de programa.

Aprender sobre programação orientada a objetos o ajudará a enxergar o mundo como um programador. Ajudará a entender seu

código – não apenas o que está acontecendo linha a linha, como também os principais conceitos que o fundamentam. Quando aprendemos a lógica subjacente das classes, ficamos preparados para pensar logicamente e para desenvolver programas que resolvam de modo efetivo basicamente qualquer problema que encontrarmos.

As classes também facilitam a sua vida e a vida de outros programadores com os quais você trabalhará conforme assume desafios cada vez mais complexos. Quando você e outros programadores programam código com base no mesmo tipo de lógica, é possível entender o trabalho um do outro. As pessoas com que você trabalha entenderão seus programas, possibilitando que todos alcancem um bom desempenho.

Criando e usando uma classe

É possível modelar quase qualquer coisa com as classes. Começaremos escrevendo uma simples classe, `Dog`, para representar um cachorro – não um cachorro específico, mas qualquer cachorro. O que sabemos sobre a maioria dos cachorros de estimação? Bom, todos têm um nome e uma idade. Sabemos também que a maioria deles senta e rola. Essas duas informações (nome e idade) e esses dois comportamentos (sentar e rolar) serão inseridos em nossa classe `Dog` porque são comuns à maioria dos cachorros. Essa classe explicará ao Python como fazer um objeto que represente um cachorro. Após escrever nossa classe, vamos usá-la para criar instâncias individuais, cada uma representando um cachorro específico.

Criando a classe Dog

Cada instância criada a partir da classe `Dog` armazenará `name` e `age`, e atribuiremos a cada cachorro a capacidade `sit()` e `roll_over()`:

dog.py

```
1 class Dog:
    """Simples tentativa de modelar um cachorro"""

2   def __init__(self, name, age):
        """ Inicializa os atributos de nome e idade"""
3       self.name = name
        self.age = age

4   def sit(self):
        """Simula um cachorro sentado em resposta a um comando"""
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        """Simula um cachorro rolando em resposta a um comando"""
        print(f"{self.name} rolled over!")
```

Podemos observar muitos detalhes aqui, mas não se preocupe. No decorrer deste livro, você verá essa estrutura e terá muito tempo para se acostumar. De início, definimos uma classe chamada `Dog` 1. Por convenção, nomes com a primeira letra maiúscula se referem às classes em Python. A definição de classe não leva parênteses porque estamos criando essa classe do zero. Em seguida, escrevemos docstrings para descrever o que a classe faz.

Método __init__()

Uma função que parte de uma classe é um *método*. Tudo o que aprendemos sobre funções também vale para os métodos; por ora, a única diferença prática é a maneira como chamaremos os métodos. O método 2 `__init__()` é um método especial que o Python executa automaticamente sempre que criamos uma instância nova com base na classe `Dog`. Esse método tem dois underscores principais à esquerda e dois à direita, convenção que ajuda a evitar que os nomes de método default do Python entrem em conflito com os nomes criados do método. Não se esqueça de usar dois underscores em cada lado de `__init__()`. Caso use apenas um de cada lado, o método não será chamado automaticamente quando a classe for usada, resultando em erros difíceis de identificar.

Definimos o método `__init__()` com três parâmetros: `self`, `name` e `age`. O

parâmetro self é necessário na definição do método e deve vir primeiro, antes dos outros parâmetros. Deve ser incluído na definição porque quando Python chamar esse método mais tarde (para criar uma instância de Dog), a chamada de método passará automaticamente o argumento self. Toda chamada de método associada a uma instância passa automaticamente self, referência à própria instância; isso concede à instância individual acesso aos atributos e aos métodos da classe. Ao criarmos uma instância de Dog, o Python chamará o método __init__() da classe Dog. Vamos passar um nome e uma idade para Dog() como argumentos; o self é passado automaticamente, não precisamos passá-lo. Sempre que quisermos criar uma instância da classe Dog, basta fornecer valores para os dois últimos parâmetros, name e age.

As duas variáveis definidas no corpo do método __init__() têm o prefixo self 3. Qualquer variável prefixada com self fica disponível para todos os métodos da classe, e também conseguiremos acessar essas variáveis por meio de qualquer instância criada a partir da classe. A linha self.name = name aceita o valor associado ao parâmetro name e o atribui à variável name que, em seguida, é anexada à instância que está sendo criada. O mesmo processo ocorre com self.age = age. Chamamos variáveis como essa, acessíveis por meio de instâncias de *atributos*.

A classe Dog tem dois outros métodos definidos: sit() e roll_over() 4. Já que esses métodos não precisam de informações adicionais para serem executados, apenas os definimos para ter um parâmetro, self. As instâncias que criarmos mais tarde terão acesso a esses métodos. Dito de outro modo, serão capazes de sentar e rolar. Por ora, sit() e roll_over() não ajudam muito. Simplesmente exibem uma mensagem dizendo se o cachorro está sentado ou rolando. Contudo, podemos ampliar esse conceito à situações realistas: se essa classe fizesse parte de um jogo de computador, esses métodos teriam o código para fazer a animação de um cachorro sentar e rolar. Caso essa classe fosse escrita para controlar um robô, esses métodos

orientariam os movimentos para fazer com que um cachorro robótico sentasse e rolasse.

Como criar uma instância a partir de uma classe

Imagine uma classe como um conjunto de instruções para criar uma instância. A classe `Dog` é um conjunto de instruções que informa ao Python como criar instâncias individuais representando cachorros específicos.

Criaremos uma instância para representar um cachorro específico.

```
class Dog:
    -- trecho de código omitido --

1 my_dog = Dog('Willie', 6)

2 print(f"My dog's name is {my_dog.name}.")
3 print(f"My dog is {my_dog.age} years old.")
```

A classe `Dog` que estamos usando é a que acabamos de criar no exemplo anterior. Aqui, solicitamos ao Python que crie um cachorro cujo nome é `'Willie'` e cuja idade é `6` `1`. Ao ler essa linha, o Python chama o método `__init__()` em `Dog` com os argumentos `'Willie'` e `6`. O método `__init__()` cria uma instância que representa esse cachorro específico e define os atributos `name` e `age` com os valores que fornecemos. O Python então retorna uma instância representando esse cachorro. Atribuímos essa instância à variável `my_dog`. Aqui, a convenção de nomenclatura ajuda; podemos normalmente supor que um nome com a primeira letra maiúscula como `Dog` se refere a uma classe, e um nome com letras minúsculas como `my_dog` se refere a uma única instância criada a partir de uma classe.

Acessando atributos

Para acessar os atributos de uma instância, use a notação de ponto. Vejamos como acessamos o valor do atributo `name` `2` de `my_dog`:

```
my_dog.name
```

No Python, usamos regularmente a notação de ponto. Essa sintaxe

demonstra como o Python encontra o valor de um atributo. Aqui, o Python confere a instância my_dog e, em seguida, encontra o atributo name associado a my_dog. É o mesmo atributo referenciado como self.name na classe Dog. Empregamos a mesma abordagem para trabalhar com o atributo age 3.

A saída é um resumo do que sabemos sobre my_dog:

```
My dog's name is Willie.
My dog is 6 years old.
```

Chamando métodos

Após criarmos uma instância a partir da classe Dog, podemos utilizar a notação de ponto para chamar qualquer método definido nessa mesma classe. Vamos fazer com que nosso cachorro sente e role.

```
class Dog:
    -- trecho de código omitido --

my_dog = Dog('Willie', 6)
my_dog.sit()
my_dog.roll_over()
```

Para chamar um método, forneça o nome da instância (nesse caso, my_dog) e o método que quer chamar, separados por um ponto. Ao ler my_dog.sit(), o Python procura o método sit() na classe Dog e executa o código. O Python interpreta a linha my_dog.roll_over() da mesma forma.

Agora, Willie faz o que mandamos:

```
Willie is now sitting.
Willie rolled over!
```

Essa sintaxe é uma mão na roda. Quando atributos e métodos recebem nomes descritivos apropriados como name, age, sit() e roll_over(), podemos facilmente deduzir o que um bloco de código, mesmo um que nunca vimos antes, deve fazer.

Criando múltiplas instâncias

Podemos criar quantas instâncias precisarmos a partir de uma

classe. Criaremos um segundo cachorro chamado `your_dog`:

```
class Dog:
    -- trecho de código omitido --

my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)

print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()

print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

Nesse exemplo, criamos um cachorro chamado Willie e um cadela chamada Lucy. Cada cachorro é uma instância separada com o próprio conjunto de atributos, capaz de executar o mesmo conjunto de ações:

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

Ainda que usássemos o mesmo nome e idade para o segundo cachorro, o Python criaria uma instância separada da classe `Dog`. É possível criar quantas instâncias precisarmos a partir de uma classe, desde que possamos atribuir a cada instância um nome de variável exclusivo ou que ocupe um lugar exclusivo em uma lista ou dicionário.

FAÇA VOCÊ MESMO

9.1 Restaurante: Crie uma classe chamada Restaurant. O método `__init__()` para Restaurant deve armazenar dois atributos: restaurant_name e cuisine_type. Crie um método chamado describe_restaurant() que exiba essas duas informações e um método chamado open_restaurant() que exiba uma mensagem sinalizando que o restaurante está aberto.

Crie uma instância chamada restaurant a partir da sua classe. Exiba os dois atributos individualmente e, em seguida, chame ambos os métodos.

9.2 Três restaurantes: Comece com sua classe do Exercício 9.1. Crie três instâncias diferentes da classe e chame describe_restaurant() para cada instância.

9.3 Usuários: Crie uma classe chamada User. Crie dois atributos chamados first_name e last_name e diversos outros atributos que normalmente são armazenados em um perfil de usuário. Crie um método chamado describe_user() que exiba um resumo das informações do usuário. Crie outro método chamado greet_user() que exiba um cumprimento personalizado ao usuário.

Crie diversas instâncias que representem usuários distintos e chame ambos os métodos para cada um.

Trabalhando com classes e instâncias

É possível utilizar classes para representar muitas situações do mundo cotidiano. Após escrever uma classe, passaremos a maior parte do tempo trabalhando com instâncias criadas a partir dessa classe. E uma das primeiras tarefas que queremos fazer é modificar os atributos associados a uma instância específica. Podemos modificar os atributos de uma instância diretamente ou escrever métodos que atualizam esses atributos de maneiras específicas.

Classe Car

Escreveremos uma classe nova representando um carro. Nossa classe armazenará informações sobre o tipo de carro com o qual estamos trabalhando e terá um método que resume essas informações:

car.py

```
class Car:
    """Simples tentativa de representar um carro"""

1   def __init__(self, make, model, year):
        """Inicializa os atributos para descrever um carro"""
        self.make = make
        self.model = model
        self.year = year

2   def get_descriptive_name(self):
        """Retorna um nome descritivo, formatado elegantemente"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()
```

```
3 my_new_car = Car('audi', 'a4', 2024)
  print(my_new_car.get_descriptive_name())
```

Na classe `Car`, primeiro definimos o método `__init__()` com o parâmetro `self` `1`, assim como fizemos com a classe `Dog`. Fornecemos também três outros parâmetros: `make`, `model` e `year`. O método `__init__()` recebe esses parâmetros e os atribui aos atributos que serão associados às instâncias criadas a partir dessa classe. Ao criarmos uma instância nova de `Car`, é necessário especificar uma marca, modelo e ano para nossa instância.

Definimos um método chamado `get_descriptive_name()` `2` que insere `year`, `make` e `model` de um carro em uma string, representando o carro de forma elegante. Assim, não precisaremos exibir o valor de cada atributo individualmente. Nesse método, para trabalhar com os valores dos atributos, utilizamos `self.make`, `self.model` e `self.year`. Fora da classe, criamos uma instância da classe `Car` e a atribuímos à variável `my_new_car` `3`. Em seguida, chamamos `get_descriptive_name()` para mostrar que tipo de carro temos:

```
2024 Audi A4
```

Vamos deixar as coisas mais interessantes adicionando um atributo que muda ao longo do tempo. Adicionaremos um atributo que armazena a quilometragem geral do carro. No código, representaremos essa quilometragem como milhas.

Definindo um valor default para um atributo

Quando uma instância é criada, os atributos podem ser definidos sem serem passados como parâmetros. Esses atributos podem ser definidos no método `__init__()`, onde lhes são atribuídos um valor default.

Adicionaremos um atributo chamado `odometer_reading` que sempre começa com um valor de 0. Adicionaremos também um método `read_odometer()` que nos ajuda a ler o hodômetro de cada carro:

```
class Car:

    def __init__(self, make, model, year):
        """Inicializa os atributos para descrever um carro"""
        self.make = make
        self.model = model
        self.year = year
1       self.odometer_reading = 0

    def get_descriptive_name(self):
        -- trecho de código omitido --

2   def read_odometer(self):
        """Exibe uma frase mostrando a quilometragem do carro, em milhas"""
        print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

Dessa vez, ao chamar o método `__init__()` para criar uma instância nova, o Python armazena os valores de marca, modelo e ano como atributos, como no exemplo anterior. Depois, o Python cria um atributo novo chamado `odometer_reading` e define seu valor inicial como 0 `1`. Temos também um método novo chamado `read_odometer()` `2` que facilita a leitura da quilometragem de um carro.

O nosso carro começa com a quilometragem 0:

```
2024 Audi A4
This car has 0 miles on it.
```

Como pouquíssimos carros são vendidos com exatamente 0 de quilometragem, precisamos alterar o valor desse atributo de alguma forma.

Modificando valores de atributos

É possível mudar o valor de um atributo de três formas: podemos alterá-lo diretamente por meio de uma instância, defini-lo por meio de um método ou incrementá-lo (adicionar um determinado valor a ele) por meio de um método. Analisaremos cada uma dessas abordagens.

Alterando o valor de um atributo diretamente

A forma mais simples de alterar o valor de um atributo é acessá-lo diretamente por meio de uma instância. Aqui, definimos diretamente a leitura do hodômetro como 23:

```
class Car:
    -- trecho de código omitido --

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

Utilizamos a notação de ponto a fim de acessar o atributo `odometer_reading` do carro e definimos seu valor diretamente. Essa linha informa ao Python para receber a instância `my_new_car`, encontrar o atributo `odometer_reading` associado e definir o valor desse atributo como 23:

```
2024 Audi A4
This car has 23 miles on it.
```

Às vezes, queremos acessar atributos de modo direto como fizemos, mas, outras vezes, queremos escrever um método que atualize seu valor.

Alterando o valor de um atributo por meio de um método

Ajuda bastante ter métodos que atualizem determinados atributos para você. Em vez de acessar os atributos diretamente, basta passar o valor novo para um método que lida internamente com a atualização.

Vejamos um exemplo que mostra um método chamado `update_odometer()`:

```
class Car:
    -- trecho de código omitido --

    def update_odometer(self, mileage):
```

```
        """Define a leitura do hodômetro para o valor fornecido"""
        self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())

1 my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

A única alteração em `Car` é a adição de `update_odometer()`. Esse método recebe um valor de quilometragem e o atribui a `self.odometer_reading`. Com a instância `my_new_car`, chamamos `update_odometer()` com `23` como argumento `1`. Isso define a leitura do hodômetro como 23 e `read_odometer()` exibe a leitura:

```
2024 Audi A4
This car has 23 miles on it.
```

Podemos incrementar o método `update_odometer()` para executar uma tarefa adicional sempre que a leitura do hodômetro for alterada. Adicionaremos um pouco de lógica para garantir que ninguém tente reverter a leitura do hodômetro:

```
class Car:
    -- trecho de código omitido --

    def update_odometer(self, mileage):
        """
        Define a leitura do hodômetro para o valor fornecido.
        Rejeita a alteração se houver tentativas de reverter o hodômetro.
        """
1       if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
2           print("You can't roll back an odometer!")
```

Agora `update_odometer()` verifica se a leitura nova faz sentido, antes de alterar o atributo. Se o valor fornecido para `mileage` for maior ou igual à quilometragem existente, `self.odometer_reading`, podemos atualizar a leitura do hodômetro com a quilometragem nova `1`. Se a quilometragem nova for menor à quilometragem existente, receberemos um aviso de que não podemos reverter o hodômetro `2`.

Incrementando o valor de um atributo por meio de um método

Não raro, queremos incrementar o valor de um atributo em um determinado valor, em vez de definir um valor completamente novo. Digamos que compramos um carro usado e percorremos 100 milhas entre o momento em que o compramos e o momento em que o registramos. Vejamos um método que nos possibilita passar essa quantidade incremental e adicionar esse valor à leitura do hodômetro:

```
class Car:
    -- trecho de código omitido --

    def update_odometer(self, mileage):
        -- trecho de código omitido --

    def increment_odometer(self, miles):
        """Adiciona a quantidade fornecida à leitura do hodômetro"""
        self.odometer_reading += miles

1 my_used_car = Car('subaru', 'outback', 2019)
  print(my_used_car.get_descriptive_name())

2 my_used_car.update_odometer(23_500)
  my_used_car.read_odometer()

  my_used_car.increment_odometer(100)
  my_used_car.read_odometer()
```

O método novo `increment_odometer()` aceita uma quantidade de milhas, e adiciona esse valor a `self.odometer_reading`. Primeiro, criamos um carro usado, `my_used_car` `1`. Definimos o hodômetro 23.500 chamando `update_odometer()` e passando `23_500` `2`. Por último, chamamos `increment_odometer()` e passamos `100` para adicionar as 100 milhas que percorremos entre comprar o carro e registrá-lo:

```
2019 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

Podemos alterar esse método para rejeitar incrementos negativos, assim ninguém consegue usar essa função para reverter o

hodômetro.

NOTA *Podemos utilizar esses métodos para controlar a forma como os usuários do programa atualizam valores como uma leitura do hodômetro, mas qualquer pessoa com acesso ao programa pode definir a leitura do hodômetro em qualquer valor, acessando diretamente o atributo. A segurança eficaz exige extrema atenção aos detalhes, além de verificações básicas como as que acabamos de mostrar.*

FAÇA VOCÊ MESMO

9.4 Pessoas atendidas: Comece com o seu programa do Exercício 9.1 (página *208*). Adicione um atributo chamado number_served com um valor default de 0. Crie uma instância chamada restaurant a partir dessa classe. Exiba o número de clientes que o restaurante atendeu e, em seguida, altere este valor e o exiba novamente.

Adicione um método chamado set_number_served() que possibilita definir o número de clientes atendidos. Chame esse método com um novo número e exiba mais uma vez o valor.

Adicione um método chamado increment_number_served(), o qual possibilita aumentar o número de clientes atendidos. Chame esse método com qualquer número que quiser e que possa representar quantos clientes foram atendidos em, digamos, um dia de atividade comercial.

9.5 Tentativas de login: Adicione um atributo chamado login_attempts à sua classe User do Exercício 9.3 (página *209*). Crie um método chamado increment_login_attempts() que incrementa o valor de login_attempts em 1. Crie outro método chamado reset_login_attempts() que redefine o valor de login_attempts para 0.

Crie uma instância da classe User e chame increment_login_attempts() diversas vezes. Exiba o valor de login_attempts para verificar que foi incrementado corretamente e, em seguida, chame reset_login_attempts(). Exiba login_attempts novamente a fim de ter certeza de que foi redefinido para 0.

Herança

Nem sempre é preciso começar do zero ao escrever uma classe. Se a classe que estiver escrevendo for uma versão especializada de outra classe escrita, recorra à *herança*. Quando uma classe *herda* de outra, assume os atributos e métodos da primeira classe. A classe original é chamada de *classe-pai*, e a classe nova é a *classe-filha*. A classe-filha pode herdar qualquer um, ou todos os atributos e

métodos da classe-pai, mas também pode definir atributos e métodos novos por conta própria.

Método __init__() para uma classe-filha

Ao escrevermos uma classe nova que toma como base uma classe existente, queremos muitas vezes chamar o método `__init__()` da classe-pai. Isso inicializará todos os atributos definidos no método-pai `__init__()` e os disponibilizará na classe-filha.

Para exemplificar, modelaremos um carro elétrico. Como um carro elétrico é apenas um tipo específico de carro, podemos tomar como base a classe `Car` que escrevemos antes para criar nossa classe nova, `ElectricCar`. Desse modo, basta escrever o código para os atributos e comportamentos específicos dos carros elétricos.

Começaremos criando uma versão simples da classe `ElectricCar`, que faz tudo o que a classe `Car` faz:

electric_car.py

```
class Car:
    """Simples tentativa de representar um carro"""

    def __init__(self, make, model, year):
        """Inicializa os atributos para descrever um carro"""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Retorna um nome descritivo, formatado elegantemente"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Exibe uma frase mostrando a quilometragem do carro"""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        """Define a leitura do hodômetro para o valor fornecido"""
        if mileage >= self.odometer_reading:
```

```
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Adiciona a quantidade fornecida à leitura do hodômetro"""
        self.odometer_reading += miles

2 class ElectricCar(Car):
    """Representa aspectos de um carro, específicos para veículos elétricos"""

3     def __init__(self, make, model, year):
        """Inicializa os atributos da classe-pai"""
4         super().__init__(make, model, year)


5 my_leaf = ElectricCar('nissan', 'leaf', 2024)
  print(my_leaf.get_descriptive_name())
```

Começamos com `Car 1`. Ao criarmos uma classe-filha, a classe-pai deve fazer parte do arquivo atual e deve aparecer antes da classe-filha no arquivo. Em seguida, definimos a classe-filha, `ElectricCar 2`. O nome da classe-pai deve ser incluído entre parênteses na definição de uma classe-filha. O método `__init__()` recebe as informações necessárias para criar uma instância de `Car 3`.

A função `super() 4` é uma função especial que possibilita chamar um método da classe-pai. Essa linha solicita ao Python que chame o método `__init__()` da classe `Car`, que, por sua vez, fornece uma instância de `ElectricCar` a todos os atributos definidos nesse método. O nome *super* se origina da convenção de chamar a classe-pai de *superclasse* e a classe-filha de *subclasse*.

Testamos se a herança executa adequadamente quando tentamos criar um carro elétrico com o mesmo tipo de informação que forneceríamos ao criar um carro comum. Criamos uma instância da classe `ElectricCar` e a atribuímos a `my_leaf 5`. Essa linha chama o método `__init__()` definido em `ElectricCar`, que, por sua vez, solicita que o Python chame o método `__init__()` definido na classe-pai `Car`. Fornecemos os argumentos `'nissan'`, `'leaf'` e `2024`.

Tirando o __init__(), ainda não temos atributos ou métodos específicos para um carro elétrico. Nesse momento, estamos apenas garantindo que o carro elétrico tenha os comportamentos apropriados de Car:

```
2024 Nissan Leaf
```

A instância ElectricCar funciona como uma instância de Car. Agora, podemos começar a definir atributos e métodos específicos para nossos carros elétricos.

Definindo atributos e métodos para a classe-filha

Uma vez que temos uma classe-filha que herde de uma classe-pai, podemos adicionar quaisquer novos atributos e métodos necessários para diferenciar a classe-filha da classe-pai.

Adicionaremos um atributo específico para carros elétricos (uma bateria, por exemplo) e um método para apresentar esse atributo. Armazenaremos o tamanho da bateria e escreveremos um método que exiba sua descrição:

```
class Car:
    -- trecho de código omitido --

class ElectricCar(Car):
    """Representa aspectos de um carro, específicos para veículos elétricos"""

    def __init__(self, make, model, year):
        """
        Inicializa os atributos da classe-pai.
        Em seguida, inicializa os atributos específicos para um carro elétrico.
        """
        super().__init__(make, model, year)
1       self.battery_size = 40

2   def describe_battery(self):
        """Exibe uma frase descrevendo o tamanho da bateria"""
        print(f"This car has a {self.battery_size}-kWh battery.")

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.describe_battery()
```

Adicionamos o atributo novo self.battery_size e definimos seu valor

inicial como 40 1. Esse atributo será associado a todas as instâncias criadas a partir da classe ElectricCar, mas não será associado a nenhuma instância de Car. Adicionamos também um método chamado describe_battery() que exibe informações sobre a bateria 2. Ao chamarmos esse método, temos uma descrição clara e específica para um carro elétrico.

```
2024 Nissan Leaf
This car has a 40-kWh battery.
```

Não há limite para o quanto podemos especializar a classe ElectricCar. É possível adicionar quantos atributos e métodos precisarmos a fim de modelar um carro elétrico com qualquer grau de acurácia necessário. Um atributo ou método pertencente a qualquer carro, não específico de um carro elétrico, deve ser adicionado à classe Car em vez de ser adicionado à classe ElectricCar. Dessa forma, qualquer pessoa que use a classe Car terá essa funcionalidade também disponível, e a classe ElectricCar conterá somente o código para as informações e comportamentos específicos de veículos elétricos.

Sobrescrevendo métodos a partir da classe-pai

É possível sobrescrever qualquer método da classe-pai que não corresponda ao que estamos tentando modelar com a classe-filha. Para fazer isso, defina um método na classe-filha com o mesmo nome do método que deseja sobrescrever na classe-pai. O Python desconsiderará o método da classe-pai e somente prestará atenção ao método definido na classe-filha.

Digamos que a classe Car tenha um método chamado fill_gas_tank(). Trata-se de um método irrelevante para um carro totalmente elétrico, talvez você queira sobrescrevê-lo. Vejamos um jeito de fazer isso:

```
class ElectricCar(Car):
    -- trecho de código omitido --

    def fill_gas_tank(self):
```

```
        """Carros elétricos não têm tanques de gasolina"""
        print("This car doesn't have a gas tank!")
```

Agora, se alguém tentar chamar `fill_gas_tank()` com um carro elétrico, o Python ignorará o método `fill_gas_tank()` em `Car` e executará esse código. Ao recorrermos à herança, podemos fazer nossas classes-filhas reterem o que precisamos e sobrescrever qualquer coisa não necessária à classe-pai.

Instâncias como atributos

Ao modelar algo do mundo cotidiano em código, talvez você se dê conta de que está adicionando cada vez mais detalhes a uma classe. Você acabará percebendo que tem uma lista crescente de atributos e métodos e que seus arquivos estão ficando maiores. Nessas situações, talvez você reconheça que parte de uma classe pode ser escrita como uma classe separada. Podemos dividir classes grandes em classes menores que trabalhem juntas; essa abordagem se chama *composição*.

Por exemplo, se continuarmos adicionando detalhes à classe `ElectricCar`, podemos reparar que estamos adicionando muitos atributos e métodos específicos à bateria do carro. Quando isso acontece, podemos parar e transferir esses atributos e métodos para uma classe separada chamada `Battery`. Em seguida, podemos utilizar uma instância de `Battery` como um atributo na classe `ElectricCar`:

```
class Car:
    -- trecho de código omitido --

class Battery:
    """Simples tentativa de modelar uma bateria para um carro elétrico"""

1   def __init__(self, battery_size=40):
        """Inicializa os atributos da bateria"""
        self.battery_size = battery_size

2   def describe_battery(self):
        """Exibe uma frase descrevendo o tamanho da bateria"""
        print(f"This car has a {self.battery_size}-kWh battery.")
```

```
class ElectricCar(Car):
    """Representa aspectos de um carro, específicos para veículos elétricos

    def __init__(self, make, model, year):
        """
        Inicializa os atributos da classe-pai.
        Em seguida, inicializa os atributos específicos para um carro elétrico.
        """
        super().__init__(make, model, year)
3       self.battery = Battery()

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
```

Definimos uma classe nova chamada `Battery` que não herda de nenhuma outra classe. O método `1 __init__()` tem um parâmetro, `battery_size`, além do `self`. Trata-se de um parâmetro opcional que define o tamanho da bateria como 40, caso nenhum valor seja fornecido. O método `describe_battery()` também foi transferido para essa classe `2`.

Agora, na classe `ElectricCar`, adicionamos um atributo chamado `self.battery 3`. Essa linha solicita que o Python crie uma instância nova a partir de `Battery` (com um tamanho default de 40, já que não estamos especificando um valor) e atribua essa instância ao atributo `self.battery`. Isso ocorrerá sempre que o método `__init__()` for chamado; agora, qualquer instância de `ElectricCar` terá uma instância de `Battery` automaticamente criada.

Criamos um carro elétrico e o atribuímos à variável `my_leaf`. Se quisermos descrever a bateria, precisamos trabalhar com o atributo `battery` do carro:

```
my_leaf.battery.describe_battery()
```

Essa linha informa ao Python para verificar a instância `my_leaf`, encontrar seu atributo `battery` e chamar o método `describe_battery()` associado à instância `Battery`, atribuída ao atributo.

A saída é idêntica ao que já vimos:

```
2024 Nissan Leaf
This car has a 40-kWh battery.
```

Embora pareça bastante trabalho, agora podemos descrever a bateria com o máximo de detalhes que quisermos sem bagunçar a classe `ElectricCar`. Vamos adicionar outro método à `Battery` que apresente o alcance da distância que o carro consegue percorrer com base no tamanho da bateria:

```
class Car:
    -- trecho de código omitido --

class Battery:
    -- trecho de código omitido --

    def get_range(self):
        """Exibe frase sobre a distância que o carro percorre com essa bateria"""
        if self.battery_size == 40:
            range = 150
        elif self.battery_size == 65:
            range = 225

        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    -- trecho de código omitido --

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
1 my_leaf.battery.get_range()
```

O método novo `get_range()` executa algumas análises simples. Se a capacidade da bateria for de 40 kWh, `get_range()` define o a distância como 150 milhas, e se a capacidade for de 65 kWh, ele define a distância como 225 milhas. Em seguida, apresenta esse valor. Se quisermos usar esse método, temos que chamá-lo novamente por meio do atributo `1 battery` do carro.

A saída nos apresenta o alcance de distância do carro com base no tamanho da bateria:

```
2024 Nissan Leaf
This car has a 40-kWh battery.
This car can go about 150 miles on a full charge.
```

Modelando objetos do cotidiano

À medida que começar a modelar coisas mais complicadas, como carros elétricos, você se deparará com dúvidas interessantes. O alcance de distância de um carro elétrico é uma propriedade da bateria ou do carro? Se estamos descrevendo somente um carro, provavelmente é bom manter a associação do método `get_range()` com a classe `Battery`. Mas se estamos descrevendo toda a linha de carros de um fabricante, provavelmente queremos transferir `get_range()` para a classe `ElectricCar`. O método `get_range()` ainda verificaria o tamanho da bateria antes de determinar o alcance de distância, mas apresentaria um alcance específico para o tipo de carro ao qual está associado. Outra opção seria manter a associação do método `get_range()` com a bateria, mas lhe passar um parâmetro como `car_model`. O método `get_range()` apresentaria um alcance de distância com base no tamanho da bateria e no modelo do carro.

Isso levanta uma questão interessante, relacionada à sua evolução como programador. Ao se deparar com questões e dúvidas como essas, repare que você já está raciocinando em um nível lógico mais alto em vez de focar a sintaxe. Você não está pensando em Python, mas em como representar o mundo que vê ao seu redor em código. Quando chegar a esse ponto, perceberá que muitas vezes não há abordagens certas ou erradas para modelar situações do mundo cotidiano. Algumas abordagens são mais eficientes do que outras, mas identificar as representações mais eficientes exige prática. Caso seu código esteja funcionando como esperado, você está de parabéns! Não desanime se descobrir que está desmantelando suas classes e as reescrevendo inúmeras vezes com abordagens diferentes. Na busca de escrever um código preciso e eficiente, todos passam por esse processo.

FAÇA VOCÊ MESMO

9.6 Sorveteria: Uma sorveteria é um tipo específico de restaurante. Escreva uma classe chamada IceCreamStand que herde da classe Restaurant do Exercício 9.1 (página *208*) ou Exercício 9.4 (página *214*). Qualquer uma das versões da classe funcionará; basta escolher a que você mais gosta. Adicione um atributo chamado flavors que armazene uma lista de sabores de sorvete. Escreva um método que exiba esses sabores. Crie uma instância a partir de IceCreamStande chame esse método.

9.7 Admin: Um administrador é um tipo especial de usuário. Crie uma classe chamada Admin que herde da classe User escrita no Exercício 9.3 (página *209*) ou Exercício 9.5 (página *214*). Adicione um atributo, privileges, que armazene uma lista de strings como "can add post", "can delete post", "can ban user", e assim por diante. Escreva um método chamado show_privileges() que enumere o conjunto de privilégios do administrador. Crie uma instância Admin e chame seu método.

9.8 Privilégios: Crie uma classe Privileges separada. A classe deve ter um atributo, privileges, que armazene uma lista de strings, conforme descrito no Exercício 9.7. Passe o método show_privileges() para essa classe. Crie uma instância de Privileges como um atributo na classe Admin. Crie uma instância nova de Admin e use seu método para mostrar seus privilégios.

9.9 Trocar bateria: Utilize a versão final do *electric_car.py* dessa seção. Adicione um método à classe Battery chamado upgrade_battery(). Esse método deve verificar o tamanho da bateria e definir a capacidade como 65, caso necessário. Crie um carro elétrico com um tamanho default de bateria, chame get_range() uma vez e, depois, chame get_range() uma segunda vez, após atualizar a bateria. Você deve ver aumento no alcance de distância do carro.

Importando classes

Conforme adiciona mais funcionalidade às suas classes, seus arquivos podem ficar extensos, mesmo quando usa adequadamente a herança e a composição. Segundo a filosofia geral do Python, precisamos manter os arquivos o mais organizados possível. Para ajudar, o Python possibilita armazenar classes em módulos e, em seguida, importar as classes que precisamos para o programa principal.

Importando uma única classe

Criaremos um módulo com apenas uma classe, `Car`. Isso levanta um problema sutil de nomenclatura: já temos um arquivo chamado

car.py neste capítulo, mas esse módulo deve chamar *car.py*, pois contém o código que representa um carro. Resolveremos esse problema de nomenclatura armazenando a classe Car em um módulo chamado *car.py,* substituindo o arquivo *car.py* que estávamos usando antes. A partir de agora, qualquer programa que utilize esse módulo precisará de um nome de arquivo mais específico, como *my_car.py*. Vejamos o *car.py* com apenas o código da classe Car:

car.py

```
"""Classe que pode ser usada para representar um carro"""

class Car:
    """Simples tentativa de representar um carro"""

    def __init__(self, make, model, year):
        """Inicializa os atributos para descrever um carro"""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Retorna um nome descritivo, formatado elegantemente"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """ Exibe uma frase mostrando a quilometragem do carro"""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        """
        Define a leitura do hodômetro para o valor fornecido
        Rejeita a alteração se houver tentativas de reverter o hodômetro
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Adiciona a quantidade fornecida à leitura do hodômetro"""
        self.odometer_reading += miles
```

Incluímos uma docstring no nível do módulo que explica brevemente o conteúdo desse módulo 1. Recomenda-se escrever uma docstring para cada módulo que criar.

Agora, criamos um arquivo separado chamado *my_car.py*. Esse arquivo importará a classe Car e criará uma instância a partir dessa classe:

my_car.py

```
1 from car import Car

  my_new_car = Car('audi', 'a4', 2024)
  print(my_new_car.get_descriptive_name())

  my_new_car.odometer_reading = 23
  my_new_car.read_odometer()
```

A instrução import 1 informa ao Python para abrir o módulo car e importar a classe Car. Agora, podemos utilizar a classe Car como se estivesse definida nesse arquivo. A saída é a mesma que já vimos:

```
2024 Audi A4
This car has 23 miles on it.
```

Importar classes é um jeito eficaz de programar. Imagine o tamanho desse arquivo de programa se toda a classe Car estivesse nele. Ao transferir a classe para um módulo e, em seguida, importá-lo, ainda temos a mesma funcionalidade, mas o arquivo principal do programa fica limpo e fácil de ler. Armazenamos também a maior parte da lógica em arquivos separados; uma vez que nossas classes funcionem como esperado, podemos abrir um pouco mão desses arquivos e focarmos a lógica de alto nível do programa principal.

Armazenando múltiplas classes em um módulo

É possível armazenar quantas classes precisarmos em um único módulo, mesmo que cada classe em um módulo precise estar relacionada de alguma forma. Como as classes Battery e ElectricCar ajudam a representar carros, vamos adicioná-las ao módulo *car.py*.

car.py

```
"""Conjunto de classes usadas para representar carros a gasolina e elétricos"""

class Car:
    -- trecho de código omitido --

class Battery:
    """Simples tentativa de modelar uma bateria para um carro elétrico"""

    def __init__(self, battery_size=40):
        """Inicializa os atributos da bateria"""
        self.battery_size = battery_size

    def describe_battery(self):
        """Exibe uma frase descrevendo o tamanho da bateria"""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
        """Exibe uma frase sobre a distância que o carro
           consegue percorrer com essa bateria"""
        if self.battery_size == 40:
            range = 150
        elif self.battery_size == 65:
            range = 225

        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    """Modela aspectos de um carro, específicos para veículos elétricos"""

    def __init__(self, make, model, year):
        """
        Inicializa os atributos da classe-pai.
        Em seguida, inicializa os atributos específicos para um carro elétrico.
        """
        super().__init__(make, model, year)
        self.battery = Battery()
```

Agora, podemos criar um arquivo novo chamado *my_electric_car.py*, importar a classe ElectricCar e criar um carro elétrico:

my_electric_car.py

```
from car import ElectricCar

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
```

```
my_leaf.battery.describe_battery()
my_leaf.battery.get_range()
```

Vemos a mesma saída de antes, ainda que a maior parte da lógica esteja ocultada em um módulo:

```
2024 Nissan Leaf
This car has a 40-kWh battery.
This car can go about 150 miles on a full charge.
```

Importando múltiplas classes de um módulo

Podemos importar quantas classes precisarmos para um arquivo de programa. Se quisermos criar um carro usual e um carro elétrico no mesmo arquivo, é necessário importar ambas as classes, `Car` e `ElectricCar`:

my_cars.py

```
1 from car import Car, ElectricCar

2 my_mustang = Car('ford', 'mustang', 2024)
  print(my_mustang.get_descriptive_name())
3 my_leaf = ElectricCar('nissan', 'leaf', 2024)
  print(my_leaf.get_descriptive_name())
```

Importamos múltiplas classes de um módulo separando cada uma com uma vírgula `1`. Após importar as classes necessárias, fique à vontade para criar quantas instâncias precisar de cada classe.

Nesse exemplo, criamos um Ford Mustang `2` a gasolina e, depois, um Nissan Leaf `3` elétrico:

```
2024 Ford Mustang
2024 Nissan Leaf
```

Importando um módulo inteiro

Podemos também importar um módulo inteiro e, em seguida, acessar as classes necessárias com a notação de ponto. Essa abordagem é simples e facilita a leitura do código. Visto que cada chamada que cria uma instância de uma classe inclui o nome do módulo, não teremos conflitos de nomes com nenhum nome utilizado no arquivo atual.

Vejamos como é importar o módulo `car` inteiro e, em seguida, como criar um carro usual e um carro elétrico:

my_cars.py

```
1 import car

2 my_mustang = car.Car('ford', 'mustang', 2024)
  print(my_mustang.get_descriptive_name())

3 my_leaf = car.ElectricCar('nissan', 'leaf', 2024)
  print(my_leaf.get_descriptive_name())
```

Primeiro, importamos o módulo `car` inteiro `1`. Em seguida, acessamos as classes necessárias por meio da sintaxe *nome_módulo*.*NomeClasse.* Mais uma vez, criamos um Ford Mustang `2` e um Nissan Leaf `3`.

Importando todas as classes de um módulo

É possível importar todas as classes de um módulo com a seguinte sintaxe:

```
from nome_módulo import *
```

Não se recomenda esse método por duas razões. A primeira, ajuda muito ser capaz de ler as instruções `import` no início de um arquivo e ter uma noção clara de quais classes um programa usa. Com essa abordagem, não fica claro quais classes do módulo estão sendo usadas. Essa abordagem também pode gerar confusão com nomes no arquivo. Caso importe sem querer uma classe com o mesmo nome que outra coisa no arquivo do programa, você pode criar erros difíceis de identificar. Mostro isso aqui porque, embora não seja uma abordagem recomendada, é provável que você a veja no código de outras pessoas em algum momento.

Caso precise importar muitas classes de um módulo, é melhor importar o módulo inteiro e usar a sintaxe *nome_módulo.NomeClasse*. Não será possível ver todas as classes usadas no início do arquivo, mas, no programa, será possível ver claramente onde o módulo é usado. Assim, evitamos também possíveis conflitos de nomenclatura que podem surgir ao importar todas as classes de um módulo.

Importando um módulo para um módulo

Às vezes, queremos distribuir nossas classes em diversos módulos para evitar que qualquer arquivo fique demasiadamente grande e para evitar armazenar classes não relacionadas no mesmo módulo. Talvez você descubra que, ao armazenar classes em diversos módulos, uma classe em um módulo depende de uma classe em outro módulo. Quando isso acontece, é possível importar a classe necessária para o primeiro módulo.

Por exemplo, armazenaremos a classe `Car` em um módulo e as classes `ElectricCar` e `Battery` em outro. Criaremos um módulo novo chamado *electric_car.py* – substituindo o arquivo *electric_car.py* que criamos antes – e copiaremos somente as classes `Battery` e `ElectricCar` para esse arquivo:

electric_car.py

```
"""Conjunto de classes que pode ser usado para representar carros elétricos

from car import Car

class Battery:
    -- trecho de código omitido --

class ElectricCar(Car):
    -- trecho de código omitido --
```

A classe `ElectricCar` precisa de acesso à sua classe-pai `Car`, então importamos `Car` diretamente para o módulo. Se esquecermos essa linha, o Python gerará um erro quando tentarmos importar o módulo `electric_car`. Precisamos também atualizar o módulo `Car` para que contenha somente a classe `Car`:

car.py

```
"""Classe que pode ser usada para representar um carro"""

class Car:
    -- trecho de código omitido --
```

Agora, podemos importar de cada módulo separadamente e criar

qualquer tipo de carro que precisamos:

my_cars.py

```
from car import Car
from electric_car import ElectricCar

my_mustang = Car('ford', 'mustang', 2024)
print(my_mustang.get_descriptive_name())

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
```

Importamos `Car` e `ElectricCar` de seus respectivos módulos. Depois, criamos um carro usual e um carro elétrico. Ambos os carros são corretamente criados:

```
2024 Ford Mustang
2024 Nissan Leaf
```

Usando aliases

Como vimos no Capítulo 8, ao usarmos módulos, os aliases podem ser bastante úteis para organizar o código dos projetos. Podemos também usar aliases ao importar classes.

Como exemplo, imagine um programa em que você queira criar muitos carros elétricos. Talvez seja entediante digitar (e ler) `ElectricCar` repetidas vezes. Podemos associar um alias a `ElectricCar` r quando declaramos o `import`:

```
from electric_car import ElectricCar as EC
```

Agora, podemos usar esse alias sempre que quisermos criar um carro elétrico:

```
my_leaf = EC('nissan', 'leaf', 2024)
```

Podemos também associar um alias a um módulo. Vejamos como importar o módulo `electric_car` inteiro com um alias:

```
import electric_car as ec
```

Agora, é possível usar esse alias de módulo com o nome completo da classe:

```
my_leaf = ec.ElectricCar('nissan', 'leaf', 2024)
```

Definindo o próprio fluxo de estudo e de trabalho

Conforme podemos ver, o Python disponibiliza muitas opções para estruturar o código em um projeto grande. É indispensável conhecer todas essas possibilidades, pois, assim, você consegue estabelecer as melhores formas de organizar seus projetos, bem como entender os projetos de outras pessoas.

De início, mantenha sua estrutura de códigos simples. Tente fazer tudo em um arquivo e transfira suas classes para módulos separados, assim que tudo estiver funcionando. Se gostar de como os módulos e arquivos interagem, tente armazenar suas classes em módulos quando começar um projeto. Encontre uma abordagem que possibilite escrever um código que funcione e comece a partir daí.

FAÇA VOCÊ MESMO

9.10 Importando Restaurant: Use sua última classe Restaurant, e armazene-a em um módulo. Crie um arquivo separado que importe Restaurant. Crie uma instância de Restaurant e chame um de seus métodos para mostrar que a instrução import executa de modo adequado.

9.11 Importando Admin: Comece com o programa do Exercício 9.8 (página *222*). Armazene as classes User, Privileges e Admin em um único módulo. Crie um arquivo separado, uma instância de Admin e chame show_privileges() com o intuito de mostrar que tudo está funcionando corretamente.

9.12 Múltiplos módulos: Armazene a classe User em um módulo e armazene as classes Privileges e Admin em outro. Em um arquivo separado, crie uma instância de Admin e chame show_privileges() com o intuito de mostrar que tudo ainda está funcionando corretamente.

Biblioteca padrão do Python

A *biblioteca padrão do Python* é um conjunto de módulos incluídos em cada instalação Python. Agora que entendemos basicamente como as funções e as classes funcionam, podemos começar a usar esses módulos desenvolvidos por outros programadores. É possível utilizar qualquer função ou classe da biblioteca padrão, incluindo uma simples instrução `import` no início dos arquivos. Vejamos o módulo `random`, que pode ajudar na modelagem de muitas situações

reais.

Uma função interessante do módulo random é a randint(). Essa função aceita dois argumentos inteiros e retorna um número inteiro selecionado aleatoriamente entre (e incluindo) esses números.

Veja como gerar um número aleatório entre 1 e 6:

```
>>> from random import randint
>>> randint(1, 6)
3
```

Outra função útil é a choice(). Essa função aceita uma lista ou tupla e retorna um elemento aleatoriamente escolhido:

```
>>> from random import choice
>>> players = ['charles', 'martina', 'michael', 'florence', 'eli']
>>> first_up = choice(players)
>>> first_up
'florence'
```

Nunca use o módulo random para criar aplicações relacionadas à segurança, apesar de funcionar perfeitamente para muitos projetos divertidos e interessantes.

NOTA *Podemos também fazer download de módulos de fontes externas. Na Parte II veremos muitos desses exemplos, já que precisaremos de módulos externos para finalizar cada projeto.*

FAÇA VOCÊ MESMO

9.13 Dados: Crie uma classe Die com um atributo chamado sides, que tem um valor default de 6. Crie um método chamado roll_die() que exiba um número aleatório entre 1 e o número de lados que o dado tem. Crie um dado com 6 lados e jogue-o 10 vezes.

Crie um dado com 10 lados e um com 20 lados. Jogue cada dado 10 vezes.

9.14 Loteria: Crie uma lista ou tupla contendo uma série de 10 números e 5 letras. Selecione aleatoriamente 4 números ou letras da lista e exiba uma mensagem informando que qualquer bilhete que corresponda a esses 4 números ou letras ganha um prêmio.

9.15 Análise de loteria: Você pode usar um loop a fim de verificar a dificuldade de alguém ganhar o tipo de loteria que acabou de modelar. Crie uma lista ou tupla chamada my_ticket. Escreva um loop que continue analisando números até seu bilhete ganhar. Exiba uma mensagem informando quantas vezes o loop teve que ser executado até sortear um bilhete vencedor.

9.16 Python Module of the Week: Excelente recurso para explorar a biblioteca padrão do Python é o site *Python Module of the Week*. Acesse https://pymotw.com e confira o índice. Encontre um módulo que lhe pareça interessante e leia-o, talvez começando com o módulo random.

Estilizando classes

Vale a pena elucidar algumas questões de estilo relacionadas às classes, sobretudo à medida que seus programas se tornam mais complicados.

Nomes de classes devem seguir o padrão *CamelCase*. Basta escrever a primeira letra do nome em maiúsculo e não usar underscores. Nomes de instâncias e módulos devem ser escritos em letras minúsculas, com underscores entre as palavras.

Crie uma docstring imediatamente após definir cada classe. A docstring deve ser uma breve descrição do que a classe faz, siga as mesmas convenções de formatação que usou para escrever as docstrings das funções. Crie também uma docstring imediatamente após criar cada módulo, descrevendo para que as classes de um módulo podem ser usadas.

Pode-se usar linhas em branco para organizar o código, mas não abuse delas. Dentro de uma classe, é possível usar uma linha em branco entre os módulos e, dentro de um módulo, pode-se usar duas linhas em branco para separar as classes.

Caso seja necessário importar um módulo de uma biblioteca padrão e um módulo que escrever, insira primeiro a instrução import do módulo da biblioteca padrão. Em seguida, adicione uma linha em branco e a instrução `import` do módulo que escreveu. Em programas com diversas instruções `import`, essa convenção facilita ver de onde se origina os diferentes módulos usados no programa.

Recapitulando

Neste capítulo, aprendemos a escrever nossas próprias classes.

Aprendemos a armazenar informações em uma classe usando atributos e como escrever métodos que atribuem às classes o comportamento necessário. Estudamos os métodos `__init__()` que criam instâncias a partir de suas classes justamente com os atributos que queremos. Vimos como alterar os atributos de uma instância diretamente e por meio de métodos. Aprendemos que a herança pode simplificar a criação de classes que estão relacionadas entre si e a usar instâncias de uma classe como atributos em outra classe para manter cada classe simples.

Vimos como armazenar classes em módulos e importar as classes necessárias para os arquivos em que serão usadas, mantendo, assim, os projetos organizados. Começamos a aprender sobre a biblioteca padrão do Python e vimos um exemplo do módulo `random`. Por último, aprendemos a estilizar classes usando as convenções pytônicas.

No Capítulo 10 aprenderemos a trabalhar com arquivos, assim podemos salvar o trabalho que fizemos em um programa e o trabalho que permitimos que os usuários fizessem. Veremos também as *exceções*, classe especial do Python, arquitetada para ajudá-lo a responder a erros, quando eles surgirem.

CAPÍTULO 10

Arquivos e exceções

Agora que você conhece as habilidades básicas e necessárias para escrever programas organizados e fáceis de usar, é hora de pensar na relevância e usabilidade de seus programas. Neste capítulo, aprenderemos a trabalhar com arquivos, assim nossos programas podem analisar rapidamente grandes volumes de dados.

Aprenderemos a resolver erros para que nossos programas não falhem em situações imprevisíveis. Veremos as *exceções*, objetos especiais que o Python cria a fim de gerenciar os erros ocorridos enquanto um programa está em execução. Veremos também o módulo `json`, que possibilita salvar dados do usuário para que não sejam perdidos quando o programa parar de ser executado.

Quando aprendemos a trabalhar com arquivos e a salvar dados, as pessoas conseguem usar mais facilmente nossos programas: os usuários são capazes de escolher quais dados fornecer e quando inseri-los, e as pessoas são capazes de executar o programa ou alguma tarefa e, encerrá-lo, para depois retomar o que estavam fazendo. Quando aprendemos a lidar com exceções, aprendemos a lidar com situações em que os arquivos não existem e com outros problemas que podem ocasionar a falha dos programas. Assim, tornamos nossos programas mais robustos, sobretudo quando temos dados ruins, oriundos de erros inocentes ou de tentativas maliciosas de quebrá-los. Com as habilidades que aprenderá neste capítulo, você desenvolverá programas com melhor aplicabilidade e usabilidade e mais instáveis.

Lendo um arquivo

Volumes impressionantes de dados estão disponíveis em arquivos de textos que, por sua vez, podem conter dados meteorológicos, dados de tráfego, dados socioeconômicos, obras literárias e muito mais. A leitura de um arquivo é sobretudo útil em aplicações de análise de dados, mas também em qualquer situação em que queremos analisar ou modificar informações armazenadas em um arquivo. Por exemplo, podemos desenvolver um programa que lê o conteúdo de um arquivo de texto para rescrevê-lo com uma formatação e exibi-lo em um navegador.

Quando quisermos trabalhar com as informações em um arquivo de texto, o primeiro passo é ler o arquivo na memória. Desse modo, é possível trabalhar com todo o conteúdo do arquivo de uma só vez ou trabalhar com o conteúdo linha a linha.

Lendo o conteúdo de um arquivo

Para ler o conteúdo de arquivos, precisamos de um arquivo com algumas linhas de texto. Começaremos com um arquivo que contém o valor do número *pi* com até 30 casas decimais, 10 casas decimais por linha:

pi_digits.txt

```
3.1415926535
  8979323846
  2643383279
```

Para testar os exemplos a seguir, podemos inserir essas linhas em um editor e salvar o arquivo como *pi_digits.txt*, ou podemos fazer o download do arquivo dos recursos do livro em https://ehmatthes.github.io/pcc_3e. Salve o arquivo no mesmo diretório em que armazenará os programas deste capítulo.

Vejamos um programa que abre esse arquivo, o lê e exibe o conteúdo dele na tela:

file_reader.py

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()
print(contents)
```

Para trabalhar com o conteúdo de um arquivo, é necessário informar ao Python o path (caminho) do arquivo. Um *path* é a localização exata de um arquivo ou pasta em um sistema. O Python fornece um módulo chamado `pathlib` que facilita o trabalho com arquivos e diretórios, independentemente do sistema operacional usado por você ou pelos usuários do seu programa. Em geral, módulos como esse, que fornecem funcionalidades específicas, são chamados de *biblioteca*. O nome `pathlib` se origina das palavras inglesas path e library, caminho e biblioteca, respectivamente.

Começamos importando a classe `Path` do `pathlib`. Podemos fazer muita coisa com um objeto `Path` que aponta para um arquivo. Por exemplo, podemos verificar se o arquivo existe antes de trabalhar com ele, ler o conteúdo do arquivo ou escrever dados novos no arquivo. Aqui, criamos um objeto `Path` representando o arquivo *pi_digits.txt*, que atribuímos à variável `path` `1`. Como esse arquivo é salvo no mesmo diretório que o arquivo *.py* que estamos escrevendo, o nome do arquivo é tudo o que o `Path` precisa para acessá-lo.

NOTA *O VS Code procura arquivos na pasta recentemente mais acessada. Se estiver usando o VS Code, comece abrindo a pasta onde está armazenando os programas deste capítulo. Por exemplo, caso esteja salvando seus arquivos de programa em uma pasta chamada* `chapter_10`*, pressione Ctr+O(⌘+O no macOS) e abra essa pasta.*

Uma vez que temos um objeto `Path` representando *pi_digits.txt*, recorremos ao método `read_text()` para ler todo o conteúdo do arquivo `2`. O conteúdo do arquivo é retornado como uma única string, que atribuímos à variável `contents`. Ao exibirmos o valor de `contents`, vemos todo o conteúdo do arquivo de texto:

```
3.1415926535
  8979323846
  2643383279
```

A única diferença entre essa saída e o arquivo original é a linha em branco extra no final da saída. A linha em branco aparece, pois `read_text()` retorna uma string vazia quando chega ao final do arquivo; essa string vazia aparece como uma linha em branco.

Podemos remover a linha em branco extra usando `rstrip()` na string `contents`:

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()
contents = contents.rstrip()
print(contents)
```

Lembre-se do Capítulo 2, em que o método `rstrip()` do Python remove, ou retira, quaisquer caracteres de espaço em branco do lado direito de uma string. Agora, a saída corresponde exatamente ao conteúdo do arquivo original:

```
3.1415926535
  8979323846
  2643383279
```

É possível remover o caractere de quebra de linha quando lemos o conteúdo do arquivo, usando o método `rstrip()` imediatamente após chamar `read_text()`:

```
contents = path.read_text().rstrip()
```

Essa linha informa ao Python para chamar o método `read_text()` no arquivo com o qual estamos trabalhando. Em seguida, aplica o método `rstrip()` à string que `read_text()` retorna. A string limpa é então atribuída à variável `contents`. Essa abordagem se chama *encadeamento de métodos*, e você a verá com frequência na programação.

Paths relativos e absolutos

Quando passamos um simples nome de arquivo como *pi_digits.txt*

para Path, o Python procura no diretório, onde o arquivo que está sendo executado atualmente (ou seja, seu arquivo de programa *.py*) é armazenado.

Não raro, dependendo de como organizamos nosso trabalho, o arquivo que queremos abrir não estará no mesmo diretório que o arquivo de programa. Por exemplo, é possível armazenar seus arquivos de programa em uma pasta chamada *python_work*; dentro de *python_work*, podemos ter outra pasta chamada *text_files*, assim é possível distinguir seus arquivos de programa dos arquivos de texto que estão sendo manipulados. Ainda que o arquivo *text_files* esteja em *python_work*, passar o Path ao nome de um arquivo em *text_files* não dará certo, já que o Python procurará somente em *python_work*, parando por aí, não procurando o Path em *text_files.* Para que o Python abra arquivos de um diretório diferente daquele onde o arquivo de programa está armazenado, é necessário fornecer o path correto.

Na programação, podemos especificar os paths de suas formas. Um *path relativo de arquivo* solicita que o Python procure um determinado local, relativo ao diretório onde o arquivo de programa atualmente em execução está armazenado. Como o arquivo *text_files* está dentro de *python_work*, precisamos criar um path que comece com o diretório *text_files* e termine com o nome do arquivo. Veja como criar esse path:

```
path = Path('text_files/nome_arquivo.txt')
```

Podemos também informar ao Python exatamente onde o arquivo está em nosso computador, independentemente de onde o programa em execução está armazenado. Isso se chama *path absoluto de arquivo*. É possível usar um path absoluto caso o path relativo não funcione. Por exemplo, caso tenha inserido *text_files* em alguma pasta diferente de *python_work*, simplesmente passar Path ao path 'text_files/*filename*.txt' não funcionará, já que o Python somente procurará esse local dentro de *python_work*. É necessário escrever por extenso um path absoluto a fim de explicitar onde queremos que

o Python procure.

Via de regra, os paths absolutos são maiores do que os paths relativos, pois estão na pasta root do sistema:

```
path = Path('/home/eric/data_files/text_files/nome_arquivo.txt')
```

Quando usamos paths absolutos, podemos ler arquivos de qualquer local do seu sistema. No momento, é mais fácil armazenar arquivos no mesmo diretório que os arquivos de programa ou em uma pasta como *text_files* dentro do diretório que armazena os arquivos de programa.

NOTA *Sistemas Windows usam barra invertida (\) em vez de uma barra (/) ao exibir os paths de arquivos, mas você deve usar barras invertidas em seu código, mesmo no Windows. A biblioteca* pathlib *usará automaticamente a representação correta do path quando interagir com seu sistema ou com o sistema de qualquer usuário.*

Acessando as linhas de um arquivo

Ao trabalharmos com um arquivo, queremos muitas vezes examinar cada linha dele. Talvez você esteja buscando informações específicas no arquivo ou queira modificar o texto do arquivo de alguma forma. Por exemplo, talvez você queira ler um arquivo de dados meteorológicos e trabalhar com qualquer linha que tenha a palavra *ensolarado* na descrição da previsão do tempo desse dia. Em um noticiário, talvez procuraremos por qualquer linha com a tag <headline> para reescrevermos essa linha com um tipo específico de formatação

Podemos utilizar o método splitlines() para transformar uma string extensa em um conjunto de linhas e, em seguida, usar um loop for para examinar cada linha de um arquivo, uma de cada vez:

file_reader.py

```
from pathlib import Path
```

```
path = Path('pi_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
for line in lines:
    print(line)
```

Começamos lendo todo o conteúdo do arquivo, como fizemos antes 1. Caso planeje trabalhar com as linhas individuais em um arquivo, não é necessário remover nenhum espaço em branco ao ler o arquivo. O método splitlines() retorna uma lista de todas as linhas no arquivo, e atribuímos essa lista à variável lines 2. Depois, percorremos essas linhas com o loop e exibimos cada uma delas:

```
3.1415926535
  8979323846
  2643383279
```

Como não modificamos nenhuma das linhas, a saída corresponde exatamente ao arquivo de texto original.

Trabalhando com o conteúdo de um arquivo

Após lermos o conteúdo de um arquivo na memória, podemos fazer o que quisermos com esses dados, logo vamos explorar um pouco os algarismos do número *pi*. Primeiro, tentaremos criar uma única string contendo todos os algarismos do arquivo sem espaço em branco:

pi_string.py

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ''
for line in lines:
    pi_string += line

print(pi_string)
print(len(pi_string))
```

Começamos lendo o arquivo e armazenando cada linha de algarismos em uma lista, assim como no exemplo anterior. Em seguida, criamos uma variável, pi_string, para armazenar os algarismos de *pi*. Escrevemos um loop que adiciona cada linha de algarismos à pi_string 1. Exibimos essa string e também mostramos seu comprimento:

```
3.1415926535  8979323846  2643383279
36
```

A variável pi_string contém o espaço em branco que estava ao lado esquerdo dos algarismos em cada linha, mas podemos removê-los usando lstrip() em cada linha:

```
-- trecho de código omitido --
for line in lines:
    pi_string += line.lstrip()

print(pi_string)
print(len(pi_string))
```

Agora, temos uma string contendo *pi* com até 30 casas decimais. A string tem 32 caracteres, pois também inclui o 3 inicial e um ponto decimal:

```
3.14159265358979323846264338327 9
32
```

NOTA *Ao ler um arquivo de texto, o Python interpreta todo o texto no arquivo como uma string. Se ler um número e quiser trabalhar com esse valor em um contexto numérico, você terá que convertê-lo em um número inteiro com a função* int() *ou em um float com a função* float()*.*

Arquivos gigantes: um milhão de algarismos

Até o momento, nos concentramos em analisar um arquivo de texto com apenas três linhas, mas o código desses exemplos executaria perfeitamente em arquivos maiores. Se começarmos com um arquivo de texto contendo *pi* e até um milhão de casas decimais, em vez de apenas 30, podemos criar uma única string com todos esses

algarismos. Não é necessário mudar nosso programa, exceto para passar um arquivo diferente. Vamos exibir somente as primeiras 50 casas decimais, desse modo, não precisamos ver um milhão de algarismos no terminal:

pi_string.py

```
from pathlib import Path

path = Path('pi_million_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ''
for line in lines:
    pi_string += line.lstrip()

print(f"{pi_string[:52]}...")
print(len(pi_string))
```

A saída mostra que, de fato, temos uma string contendo *pi* com até um milhão de casas decimais.

```
3.14159265358979323846264338327950288419716939937510...
1000002
```

O Python não tem limite inerente à quantidade de dados com os quais podemos trabalhar; é possível trabalhar com a quantidade de dados que a memória do seu sistema conseguir suportar.

NOTA *Para executar esse programa (e muitos dos exemplos a seguir), é necessário fazer o download dos recursos disponíveis em* https://ehmatthes.github.io/pcc_3e*.*

Seu aniversário está contido no número Pi?

Sempre tive curiosidade de saber se a data do meu aniversário aparece em algum lugar entre os algarismos do número *pi*. Usaremos o programa que acabamos de escrever para descobrir se a data de aniversário de alguém aparece em algum lugar entre o primeiro milhão de algarismos do número *pi*. Faremos isso expressando cada aniversário como uma string de algarismos e

analisando se essa string aparece na sequência dos algarismos em pi_string:

pi_birthday.py

```
-- trecho de código omitido --
for line in lines:
    pi_string += line.strip()

birthday = input("Enter your birthday, in the form mmddyy: ")
if birthday in pi_string:
    print("Your birthday appears in the first million digits of pi!")
else:
    print("Your birthday does not appear in the first million digits of pi.")
```

Primeiro, solicitamos a data de aniversário do usuário e, depois, verificamos se essa string está em pi_string. Vamos testar:

```
Enter your birthdate, in the form mmddyy: 120372
Your birthday appears in the first million digits of pi!
```

Minha data de aniversário aparece entre os algarismos do número *pi*! Após ler um arquivo, podemos analisar seu conteúdo de qualquer forma que possamos imaginar.

FAÇA VOCÊ MESMO

10.1 Aprendendo Python: Abra um arquivo em branco no seu editor de texto e escreva algumas linhas resumindo o que aprendeu sobre o Python até agora. Comece cada linha com a frase: *No Python, podemos ...* Salve o arquivo com o nome *learning_python.txt*, no mesmo diretório que seus exercícios deste capítulo. Crie um programa que leia o arquivo e exiba o que você escreveu duas vezes: exiba o conteúdo uma vez, lendo o arquivo inteiro, e uma outra vez, armazenando as linhas em uma lista e, depois, percorrendo cada linha com um loop.

10.2 Aprendendo C: É possível utilizar o método replace() para substituir qualquer palavra em uma string por uma palavra diferente. Veja um breve exemplo de como substituir 'dog' por 'cat' em uma frase:

```
>>> message = "I really like dogs."
>>> message.replace('dog', 'cat')
'I really like cats.'
```

Leia cada linha do arquivo que acabou de criar, *learning_python.txt*, e substitua a palavra *Python* pelo nome de outra linguagem de programação, como *C.* Exiba cada linha modificada na tela.

10.3 Código mais simples: Nesta seção, o programa *file_reader.py* usa uma variável temporária, lines, para mostrar como o splitlines() funciona. Você pode ignorar a variável temporária e percorrer com um loop a lista que splitlines() retorna:

```
for line in contents.splitlines():
```

Remova a variável temporária de cada um dos programas desta seção para que fiquem mais concisos.

Escrevendo em um arquivo

Uma das formas mais simples de salvar dados é escrevê-los em um arquivo. Ao escrevermos um texto em um arquivo, a saída ainda fica disponível, mesmo após fecharmos o terminal com a saída do programa. É possível examinar a saída após um programa terminar de ser executado, e podemos também compartilhar os arquivos de saída com outras pessoas. Podemos também escrever programas que releiam o texto da memória e podemos trabalhar mais uma vez com esses dados posteriormente.

Escrevendo em uma única linha

Após definirmos um path, podemos escrever em um arquivo usando o método `write_text()`. Para ver como isso funciona, vamos escrever uma mensagem simples e vamos armazená-la em um arquivo em vez de exibi-la:

write_message.py

```
from pathlib import Path

path = Path('programming.txt')
path.write_text("I love programming.")
```

O método `write_text()` recebe um único argumento: a string que queremos escrever no arquivo. Esse programa não tem saída de terminal, mas se abrirmos o arquivo *programming.txt*, veremos uma linha:

programming.txt

```
I love programming.
```

Esse arquivo se comporta como qualquer outro arquivo de seu computador. Conseguimos abri-lo, escrever um texto novo nele, copiar dele, colar nele, e assim por diante.

NOTA *O Python só consegue escrever strings em um arquivo de texto. Se quiser armazenar dados numéricos em um arquivo de texto, será necessário primeiro converter os dados para o formato de string com a função* str()*.*

Escrevendo múltiplas linhas

O método write_text() executa algumas coisas nos bastidores. Se o arquivo para o qual o path apontar não existir, o método cria esse arquivo. Além do mais, depois de escrever a string no arquivo, o método garante que ele seja adequadamente fechado. Arquivos que não são adequadamente fechados podem resultar em dados ausentes ou corrompidos.

Para escrever mais de uma linha em um arquivo, é necessário criar uma string contendo todo o conteúdo do arquivo e, em seguida, chamar write_text() com essa string. Escreveremos múltiplas linhas no arquivo *programming.txt*:

```
from pathlib import Path

contents = "I love programming.\n"
contents += "I love creating new games.\n"
contents += "I also love working with data.\n"

path = Path('programming.txt')
path.write_text(contents)
```

Definimos uma variável chamada contents para armazenar todo o conteúdo do arquivo. Na próxima linha, utilizamos o operador += para adicionar essa string à variável. Podemos fazer isso quantas vezes forem necessárias, a fim de criar strings de qualquer comprimento. Nesse caso, incluímos caracteres de quebra de linha no final de cada linha, para garantir que cada instrução apareça na própria linha.

Se executarmos o programa e abrirmos *programming.txt*, veremos cada uma dessas linhas no arquivo de texto:

```
I love programming.
I love creating new games.
I also love working with data.
```

Podemos também usar espaços, tabulações e linhas em branco para formatar a saída, assim como fazemos com a saída do terminal. Não há limite para o comprimento de strings, e é assim que muitos documentos gerados por computador são criados.

NOTA *Seja cuidadoso ao chamar* write_text() *em um objeto path. Se o arquivo já existir,* write_text() *apagará o conteúdo atual do arquivo e escreverá um conteúdo novo nele. Neste capítulo, mais adiante, aprenderemos como verificar se um arquivo existe com o* pathlib*.*

FAÇA VOCÊ MESMO

10.4 Convidados: Escreva um programa que solicite ao usuário seu nome. Quando responder, escreva o nome dele em um arquivo chamado *guest.txt*.

10.5 Livro de convidados: Escreva um loop while que solicite o nome dos usuários. Colete todos os nomes inseridos e, em seguida, escreva esses nomes em um arquivo chamado *guest_book.txt*. Garanta que cada item apareça em uma linha nova no arquivo.

Exceções

O Python utiliza objetos especiais chamados *exceções* para gerenciar erros ocorridos durante a execução de um programa. Sempre que ocorre um erro que faz com que Python fique indeciso sobre como agir em seguida, gera-se um objeto de exceção. Caso desenvolva um código para abordar a exceção, o programa continuará em execução. Se não tentar abordar a exceção, o programa será suspenso e mostrará um *traceback*, incluindo um relatório da exceção lançada.

As exceções são tratadas com blocos try-except. Um bloco *try-except* solicita que Python tome alguma ação, mas também informa ao Python o que fazer se uma exceção for lançada. Quando usamos blocos try-except, os programas continuam executando, mesmo

quando as coisas começam a dar errado. Em vez de tracebacks, que podem deixar os usuários confusos, os usuários verão mensagens amigáveis de erro, escritas por você.

Lidando com a exceção ZeroDivisionError

Vejamos um erro simples que faz com que o Python lance uma exceção. É bem provável que você saiba que é impossível dividir um número por zero, mas, mesmo assim, solicitaremos que o Python faça isso:

division_calculator.py

```
print(5/0)
```

Como o Python não consegue efetuar essa operação, recebemos um traceback:

```
Traceback (most recent call last):
  File "division_calculator.py", line 1, in <module>
    print(5/0)
          ~^~
1 ZeroDivisionError: division by zero
```

O erro relatado no traceback, ZeroDivisionError, é um objeto de exceção 1. O Python cria esse tipo de objeto em resposta a uma situação em que não consegue fazer o que solicitamos. Quando isso ocorre, o Python suspende o programa e nos informa o tipo de exceção lançada. Podemos utilizar essas informações para modificar nosso programa. Diremos ao Python o que fazer quando esse tipo de exceção ocorrer; assim, se acontecer de novo, estaremos preparados.

Usando blocos try-except

Se acharmos que um erro pode ocorrer, podemos escrever um bloco try-except para lidar com a exceção que pode ser lançada. Instruímos o Python para tentar executar um código e como deve agir se o código resultar em um tipo específico de exceção.

Vejamos um bloco try-except que lida com a exceção ZeroDivisionError:

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Inserimos `print(5/0)`, a linha que causou o erro, dentro de um bloco `try`. Se o código em um bloco `try` executar, o Python desconsidera o bloco `except`. Se o código no bloco `try` causar um erro, o Python procurará um bloco `except`, cujo erro corresponda ao que foi lançado e executará o código nesse bloco.

Nesse exemplo, o código no bloco `try` gera um `ZeroDivisionError`. Desse modo, o Python procura um bloco `except` que lhe informe como responder. O Python executa o código nesse bloco e o usuário vê uma mensagem de erro amigável em vez de um traceback:

```
You can't divide by zero!
```

Caso tenha mais código após o bloco `try-except`, o programa continua sendo executado porque instruímos o Python como lidar com o erro. Vejamos um exemplo em que a identificação de um erro pode possibilitar que um programa continue em execução.

Usando exceções para prevenir falhas

Abordar de modo adequado os erros é imprescindível, sobretudo quando o programa tem que executar mais tarefas após o erro ocorrer. Erros acontecem regularmente em programas que solicitam entrada aos usuários. Se responder à entrada inválida adequadamente, o programa consegue solicitar uma entrada mais válida em vez de falhar.

Criaremos uma calculadora simples que efetue somente divisões:

division_calculator.py

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
```

```
2     second_number = input("Second number: ")
      if second_number == 'q':
          break
3     answer = int(first_number) / int(second_number)
      print(answer)
```

Esse programa solicita que o usuário insira um `first_number 1` e, se o usuário não inserir `q` para sair, um `second_number 2`. Em seguida, dividimos esses dois números para obter `answer 3`. Como o programa não toma nenhuma ação para lidar com erros, solicitar uma divisão por zero provocará uma falha:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
Traceback (most recent call last):
  File "division_calculator.py", line 11, in <module>
    answer = int(first_number) / int(second_number)
             ~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~
ZeroDivisionError: division by zero
```

Não é nada bom quando um programa falha, mas é pior permitir que os usuários vejam tracebacks. Usuários não técnicos ficarão confusos e, em um ambiente malicioso, hackers coletarão mais informações do que queremos. Por exemplo, eles saberão o nome do arquivo do programa e verão uma parte do código que não está funcionando corretamente. Às vezes, um hacker habilidoso pode usar essas informações para definir que tipo de ataque lançar contra seu código.

Bloco else

É possível que o programa fique mais resistente a erros, envolvendo a linha que pode gerar erros em um wrapper contendo um bloco `try-except`. Como o erro ocorre na linha que executa a divisão, é nela que vamos inserir o bloco `try-except`. No exemplo a seguir, também podemos ver um bloco `else`. Qualquer código que dependa da execução bem-sucedida do bloco `try` deve ser inserido no bloco `else`:

```
    -- trecho de código omitido --
  while True:
      -- trecho de código omitido --
      if second_number == 'q':
          break
1     try:
          answer = int(first_number) / int(second_number)
2     except ZeroDivisionError:
          print("You can't divide by 0!")
3     else:
          print(answer)
```

Instruímos o Python para tentar finalizar a operação de divisão em um bloco `try` 1, que tem somente o código que pode causar um erro. Qualquer código que dependa da execução bem-sucedida do bloco `try` é adicionado ao bloco `else`. Nesse caso, se a operação de divisão for bem-sucedida, usamos o bloco `else` para exibir o resultado 3.

O bloco `except` informa ao Python como responder quando um `ZeroDivisionError` é lançado 2. Se a execução do bloco `try` não for bem-sucedida devido a um erro de divisão por zero, exibimos uma mensagem amigável, informando ao usuário como evitar esse tipo de erro. O programa continua sendo executado e o usuário nunca vê um traceback:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
You can't divide by 0!

First number: 5
Second number: 2
2.5

First number: q
```

O único código que deve ser inserido em um bloco `try` é aquele que pode fazer com que uma exceção seja lançada. Às vezes, você terá um código adicional que deve ser executado apenas se a execução do bloco `try` for bem-sucedida; esse código deve ser inserido no bloco `else`. O bloco `except` informa ao Python o que fazer se

determinada exceção for lançada quando tenta executar o código no bloco try.

Quando prevemos prováveis fontes de erros, podemos desenvolver programas robustos que continuam a ser executados mesmo quando encontram dados inválidos e recursos ausentes. Assim, o código fica resistente a erros de usuários inocentes e ataques maliciosos.

Lidando com a exceção FileNotFoundError

Um problema comum ao trabalhar com arquivos é lidar com arquivos ausentes. O arquivo que você está procurando pode estar em um local diferente, talvez o nome do arquivo esteja incorreto ou o arquivo não existe. Podemos lidar com todas essas situações com um bloco try-except.

Tentaremos ler um arquivo que não existe. O programa a seguir tenta ler o conteúdo de *Alice in Wonderland*, no entanto, não salvei o arquivo *alice.txt* no mesmo diretório que *alice.py*:

alice.py

```
from pathlib import Path

path = Path('alice.txt')
contents = path.read_text(encoding='utf-8')
```

Repare que estamos usando read_text() de forma ligeiramente diferente do que já vimos antes. O argumento encoding é necessário quando a codificação padrão do sistema não corresponde à codificação do arquivo que está sendo lido. É mais provável que isso aconteça ao ler um arquivo que não foi criado em seu sistema.

Como não consegue ler um arquivo ausente, o Python lança uma exceção:

```
  Traceback (most recent call last):
1   File "alice.py", line 4, in <module>
2     contents = path.read_text(encoding='utf-8')
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    File "/.../pathlib.py", line 1056, in read_text
      with self.open(mode='r', encoding=encoding, errors=errors) as f:
```

```
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^
  File "/.../pathlib.py", line 1042, in open
    return io.open(self, mode, buffering, encoding, errors, newline)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^
3 FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

Temos um traceback mais extenso do que os que vimos anteriormente, assim analisaremos como podemos entender tracebacks mais complexos. Em geral, é melhor começar com o final do traceback. Na última linha, podemos ver que uma exceção FileNotFoundError foi lançada 3. Isso é importante, pois a linha nos informa que tipo de exceção usar no bloco except que vamos escrever.

Se voltarmos, perto do início do traceback 1, podemos ver que o erro ocorreu na linha 4 do arquivo *alice.py*. A próxima linha mostra a linha de código que causou o erro 2. O restante do traceback mostra códigos das bibliotecas envolvidas na abertura e leitura de arquivos. Via de regra, não é necessário ler ou entender todas as linhas de um traceback.

Para lidar com o erro que está sendo lançado, o bloco try começará com a linha identificada como problemática no traceback. Em nosso exemplo, essa é a linha que contém read_text():

```
from pathlib import Path

path = Path('alice.txt')
try:
    contents = path.read_text(encoding='utf-8')
1 except FileNotFoundError:
    print(f"Sorry, the file {path} does not exist.")
```

Nesse exemplo, o código no bloco try gera um FileNotFoundError, portanto, escrevemos um bloco except que corresponde a esse erro 1. O Python executa o código nesse bloco quando o arquivo não pode ser encontrado, e o resultado é uma mensagem de erro amigável em vez de um traceback:

```
Sorry, the file alice.txt does not exist.
```

Como o arquivo não existe, o programa não tem mais nada para

verificar. Por isso, vemos esse tipo de saída. Vamos desenvolver esse exemplo e ver como o tratamento de exceções pode ajudar quando estamos trabalhando com mais de um arquivo.

Analisando textos

É possível analisar arquivos de texto contendo livros inteiros. Muitas obras clássicas da literatura estão disponíveis como arquivos de texto simples porque são de domínio público. Os textos usados nesta seção são provenientes do Projeto Gutenberg (https://gutenberg.org). O Projeto Gutenberg faz a curadoria de uma coleção de obras literárias disponíveis em domínio público. É um excelente recurso, caso esteja interessado em trabalhar com textos literários em seus projetos de programação.

Vamos extrair o texto de *Alice in Wonderland* e tentar contar o número de palavras dele. Para fazer isso, usaremos o método de string `split()`, que por padrão divide uma string onde quer que encontre qualquer espaço em branco:

```
from pathlib import Path

path = Path('alice.txt')
try:
    contents = path.read_text(encoding='utf-8')
except FileNotFoundError:
    print(f"Sorry, the file {path} does not exist.")
else:
    # Conta o número aproximado de palavras no arquivo
    words = contents.split()
    num_words = len(words)
    print(f"The file {path} has about {num_words} words.")
```

Movi o arquivo *alice.txt* para o diretório correto, desta vez, o bloco `try` funcionará. Pegamos a string `contents`, que agora contém todo o texto de *Alice in Wonderland* como uma string extensa, e usamos `split()` a fim de gerar uma lista de todas as palavras do livro `1`. Se usarmos `len()` nesta lista `2`, a função nos fornece uma boa aproximação do número de palavras do texto original. Por último, exibimos uma mensagem informando quantas palavras foram encontradas no

arquivo. Inserimos esse código no bloco `else`, já que esse código só é executado se o código no bloco `try` tiver sido executado com sucesso.

A saída nos informa quantas palavras existem em *alice.txt*:

```
The file alice.txt has about 29594 words.
```

Embora a contagem seja um pouco alta, visto que estamos usando um arquivo de texto com informações extras fornecidas pelo editor, temos uma boa aproximação do tamanho de *Alice in Wonderland.*

Trabalhando com múltiplos arquivos

Analisaremos mais livros, mas, antes disso, passaremos a maior parte deste programa para uma função chamada `count_words()`. Isso facilitará a execução da análise de múltiplos livros:

word_count.py

```
from pathlib import Path

def count_words(path):
    """ Conta o número aproximado de palavras em um arquivo
    try:
        contents = path.read_text(encoding='utf-8')
    except FileNotFoundError:
        print(f"Sorry, the file {path} does not exist.")
    else:
        # Conta o número aproximado de palavras no arquivo:
        words = contents.split()
        num_words = len(words)
        print(f"The file {path} has about {num_words} words.")

path = Path('alice.txt')
count_words(path)
```

A maior parte desse código está inalterada. Apenas indentamos e transferimos o código para o corpo de `count_words()`. Como é bom hábito manter os comentários atualizados quando alteramos um programa, convertemos o comentário em uma docstring e o reescrevemos um pouco 1.

Agora, podemos escrever um breve loop para contar as palavras em qualquer texto que quisermos analisar. Para isso, armazenamos os

nomes dos arquivos que queremos analisar em uma lista e, depois, chamamos count_words() para cada arquivo na lista. Tentaremos contar as palavras dos livros *Alice in Wonderland*, *Siddhartha*, *Moby Dick* e *Little Women*, todos disponíveis em domínio público. Não inclui de propósito *siddhartha.txt* no diretório que contém *word_count.py*, para que vermos como nosso programa lida com um arquivo ausente:

```
from pathlib import Path

def count_words(filename):
    -- trecho de código omitido --

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt',
        'little_women.txt']
for filename in filenames:
1    path = Path(filename)
    count_words(path)
```

Os nomes dos arquivos são armazenados como strings simples. Cada string é então convertida em um objeto Path 1, antes de chamarmos count_words(). O arquivo *siddhartha.txt* ausente não afeta o restante da execução do programa:

```
The file alice.txt has about 29594 words.
Sorry, the file siddhartha.txt does not exist.
The file moby_dick.txt has about 215864 words.
The file little_women.txt has about 189142 words.
```

Nesse exemplo, o uso do bloco try-except fornece duas vantagens consideráveis. Primeiro, evitamos que nossos usuários vejam o traceback, segundo, permitimos que programa continuasse analisando os textos que consegue encontrar. Se não identificássemos o FileNotFoundError que *siddhartha.txt* lança, o usuário veria o traceback inteiro, e o programa interromperia sua execução, depois de tentar analisar *Siddhartha*. O programa também nunca analisaria *Moby Dick* ou *Little Women*.

Falhando sem acusar erros

No exemplo anterior, informamos aos nossos usuários que um dos

arquivos estava indisponível. No entanto, não é necessário relatar todas as exceções identificadas. Vez ou outra, queremos que o programa falhe sem acusar erros quando ocorrer uma exceção e continue executando como se nada tivesse acontecido. O termo falhando sem acusar erros se refere ao termo fail silently, que pode ser encontrado também como *falhe silenciosamente*. Para fazer com que um programe falhe sem acusar erros, basta escrever um bloco `try` como de costume, e dizer explicitamente ao Python para não executar nenhuma ação no bloco `except`. O Python tem uma instrução `pass` que lhe instrui a não fazer nada em um bloco:

```
def count_words(path):
    """Conta o número aproximado de palavras em um arquivo"""
    try:
        -- trecho de código omitido --
    except FileNotFoundError:
        pass
    else:
        -- trecho de código omitido --
```

A única diferença entre essa listagem e a anterior é a instrução `pass` no bloco `except`. Agora, quando um `FileNotFoundError` é lançado, o código no bloco `except` é executado, mas nada acontece. Nenhum traceback é gerado e não há saída em resposta ao erro lançado. Os usuários veem a contagem de palavras para cada arquivo existente, mas não veem nenhum indicativo de que um arquivo não foi encontrado.

```
The file alice.txt has about 29594 words.
The file moby_dick.txt has about 215864 words.
The file little_women.txt has about 189142 words.
```

A declaração `pass` também se comporta como um placeholder. Trata-se de um lembrete de que optamos por não fazer nada em um ponto específico na execução do programa, mas talvez, posteriormente, possamos fazer algo. Por exemplo, nesse programa, podemos registrar qualquer nome de arquivo ausente em um arquivo chamado *missing_files.txt*. Nossos usuários não veriam esse arquivo, mas seríamos capazes de ler o arquivo e lidar com quaisquer textos ausentes.

Decidindo quais erros reportar

Como sabemos quando relatar um erro aos usuários e quando fazer com que um programa falha sem acusar erros? Se souberem quais textos devem ser analisados, talvez os usuários gostem de uma mensagem informando por que alguns textos não foram analisados. Caso esperem por alguns resultados, mas não saibam quais livros devem ser analisados, talvez não seja necessário que os usuários saibam que alguns textos estão indisponíveis. Fornecer aos usuários informações que não estão procurando pode limitar a usabilidade de seu programa. As estruturas de gestão de erros do Python possibilitam controle detalhado sobre o quanto compartilhar com os usuários quando as coisas saírem errado; cabe a você decidir quanta informação compartilhar.

Um código bem escrito e devidamente testado não fica muito propenso a erros internos, como erros de sintaxe ou lógicos. Mas, sempre que seu programa depender de algo externo, como a entrada do usuário, a existência de um arquivo ou a disponibilidade de uma conexão de rede, existe a possibilidade de uma exceção ser lançada. À medida que adquire um pouco mais de experiência, você saberá onde incluir blocos de tratamento de exceções em seu programa e quanto relatar aos usuários sobre erros lançados.

FAÇA VOCÊ MESMO

10.6 Operação de soma: Um problema comum ao solicitar entrada numérica ocorre quando as pessoas fornecem texto em vez de números. Quando tentamos converter a entrada em um int, recebemos um ValueError. Escreva um programa que solicite dois números. Faça a soma deles e exiba o resultado. Identifique o ValueError se qualquer valor de entrada não for um número e exiba uma mensagem de erro amigável. Teste seu programa inserindo dois números e, em seguida, forneça um texto em vez de um número.

10.7 Calculadora de soma: Insira o código do Exercício 10.5 em um loop while para que o usuário possa continuar fornecendo números, mesmo que cometa um erro e insira texto em vez de um número.

10-8 Gatos e cachorros: Crie dois arquivos, *cats.txt* e *dogs.txt*. Armazene, pelo menos, três nomes de gatos no primeiro arquivo e três nomes de cachorros no segundo arquivo. Crie um programa que tente ler esses arquivos e exiba o conteúdo do arquivo na tela. Insira seu código em um bloco try-except para detectar o erro FileNotFound e

exiba uma mensagem amigável se um arquivo estiver ausente. Transfira um dos arquivos para um local diferente em seu sistema e garanta que o código no bloco except seja devidamente executado.

10.9 Gatos e cachorros sem acusar erros: Modifique seu bloco except do Exercício 10.7 para que falhe sem acusar erros se algum arquivo estiver ausente.

10.10 Palavras comuns: Visite o Projeto Gutenberg (https://gutenberg.org/) e encontre alguns textos que você gostaria de analisar. Faça o download dos arquivos de textos dos livros ou copie o texto bruto do seu navegador em um arquivo de texto no seu computador.

Você pode utilizar o método count() para detectar quantas vezes uma palavra ou frase aparece em uma string. Por exemplo, o código a seguir conta o número de vezes que a palavra 'row' aparece em uma string:

```
>>> line = "Row, row, row your boat"
>>> line.count('row')
2
>>> line.lower().count('row')
3
```

Perceba que converter a string em letras minúsculas com a função lower() coleta todas as ocorrências da palavra que você está procurando, independentemente de como estão formatadas.

Crie um programa que leia os arquivos encontrados no Projeto Gutenberg e determine quantas vezes a palavra 'the' aparece em cada texto. Será uma aproximação porque o programa também contará palavras como 'then' e 'there'. Tente inserir 'the ', com um espaço na string, e veja o quanto sua contagem diminui.

Armazenando dados

Muitos de seus programas solicitarão aos usuários que forneçam informações específicas. É possível permitir que os usuários armazenem preferências em um jogo ou forneçam dados para uma visualização. Seja lá qual for o foco do seu programa, você armazenará as informações fornecidas pelos usuários em estruturas de dados, como listas e dicionários. Quando os usuários fecham um programa, quase sempre desejamos salvar as informações inseridas. Uma forma simples de fazer isso é armazenar dados usando o módulo `json`.

O módulo `json` possibilita converter estruturas Python simples de dados em strings formatadas em JSON e, em seguida, possibilita carregar os dados desse arquivo na próxima vez que o programa for

executado. Podemos também utilizar o json para compartilhar dados entre diferentes programas Python. Melhor ainda, o formato de dados JSON não é específico do Python, ou seja, podemos compartilhar dados armazenados no formato JSON com pessoas que trabalham com diferentes linguagens de programação. É um formato conveniente e portátil, e é fácil de aprender.

NOTA *O formato* JSON (JavaScript Object Notation, Notação de Objeto JavaScript, em tradução livre) *foi inicialmente desenvolvido para JavaScript. No entanto, tornou-se um formato comum usado por muitas linguagens, incluindo o Python.*

Usando as funções json.dumps() e json.loads())

Criaremos um breve programa que armazena um conjunto de números e outro programa que relê esses números na memória. O primeiro programa usará a função json.dumps() para armazenar o conjunto de números, e o segundo programa usará a função json.loads().

A função json.dumps () recebe um argumento: um dado que deve ser convertido para o formato JSON. A função retorna uma string, que podemos escrever em um arquivo de dados:

number_writer.py

```
from pathlib import Path
import json

numbers = [2, 3, 5, 7, 11, 13]

path = Path('numbers.json')
contents = json.dumps(numbers)
path.write_text(contents)
```

Primeiro, importamos o módulo json e, em seguida, criamos uma lista de números com a qual trabalhar. Depois, escolhemos um nome de arquivo para armazenar a lista de números 1. É de praxe usar a extensão de arquivo *.json* para sinalizar que os dados no arquivo

estão armazenados no formato JSON. Em seguida, usamos a função json.dumps () 2 para gerar uma string contendo a representação JSON dos dados com os quais estamos trabalhando. Uma vez que temos essa string, a escrevemos no arquivo com o mesmo método write_text() que usamos antes.

Esse programa não tem saída, mas vamos abrir o arquivo *numbers.json* para verificá-lo. Os dados são armazenados em um formato que se parece com o Python:

```
[2, 3, 5, 7, 11, 13]
```

Agora, vamos escrever um programa separado que usa json.loads() para reler a lista na memória:

number_reader.py

```
from pathlib import Path
import json

path = Path('numbers.json')
contents = path.read_text()
numbers = json.loads(contents)

print(numbers)
```

Fizemos questão de ler do mesmo arquivo que escrevemos 1. Como o arquivo de dados é somente um arquivo de texto com formatação específica, podemos lê-lo com o método read_text() 2. Em seguida, passamos o conteúdo do arquivo para json.loads() 3. Essa função aceita uma string formatada em JSON e retorna um objeto Python (nesse caso, uma lista), que atribuímos a numbers. Por último, exibimos a lista de números acessados e verificamos que é a mesma lista criada em *number_writer.py*:

```
[2, 3, 5, 7, 11, 13]
```

É a forma mais simples de compartilhar dados entre dois programas.

Salvando e lendo dados gerados pelo usuário

Salvar dados com o json é útil quando trabalhamos com dados gerados pelo usuário, já que se não armazenarmos as informações

do usuário de alguma forma, as perderemos quando o programa parar de ser executado. Vejamos um exemplo em que solicitamos o nome do usuário na primeira vez que ele executa um programa e, em seguida, lembramos seu nome quando ele executar o programa novamente.

Começaremos armazenando o nome do usuário:

remember_me.py

```
from pathlib import Path
import json

1 username = input("What is your name? ")

2 path = Path('username.json')
  contents = json.dumps(username)
  path.write_text(contents)

3 print(f"We'll remember you when you come back, {username}!")
```

Primeiro, solicitamos um nome de usuário para armazenar 1. Em seguida, escrevemos os dados que acabamos de coletar em arquivo chamado *username.json* 2. Depois, exibimos uma mensagem informando ao usuário que armazenamos suas informações 3:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Agora, criaremos um programa novo que cumprimenta um usuário cujo nome já foi armazenado:

greet_user.py

```
from pathlib import Path
import json

1 path = Path('username.json')
  contents = path.read_text()
2 username = json.loads(contents)

print(f"Welcome back, {username}!")
```

Lemos o conteúdo do arquivo de dados 1 e, depois, usamos json.loads() para atribuir os dados acessados à variável username 2. Como

acessamos o nome de usuário, podemos dar as boas-vindas novamente ao usuário com um cumprimento personalizado:

```
Welcome back, Eric!
```

Precisamos combinar esses dois programas em um arquivo. Quando alguém executar *remember_me.py*, queremos acessar o nome de usuário da memória, se possível. Caso contrário, solicitaremos um nome de usuário e o armazenaremos em *username.json* para a próxima vez. Aqui, poderíamos escrever um bloco `try-except` para responder adequadamente se *username.json* existe ou não, mas em vez disso, recorreremos a um método útil do módulo `pathlib`:

remember_me.py

```
from pathlib import Path
import json

path = Path('username.json')
1 if path.exists():
    contents = path.read_text()
    username = json.loads(contents)
    print(f"Welcome back, {username}!")
2 else:
    username = input("What is your name? ")
    contents = json.dumps(username)
    path.write_text(contents)
    print(f"We'll remember you when you come back, {username}!")
```

Podemos usar muitos métodos úteis com objetos `Path`. O método `exists()` retorna `True` se existir um arquivo ou pasta e `False` se não existir. Aqui, usamos `path.exists()` para descobrir se um nome de usuário já foi armazenado `1`. Se *username.json* existir, carregamos o nome de usuário e exibimos um cumprimento personalizado.

Se o arquivo *username.json* não existir `2`, solicitamos um nome de usuário e armazenamos o valor que o usuário fornece. Exibimos também mensagem familiar de que nos lembraremos deles quando retornarem.

Qualquer que seja o bloco executado, o resultado é um nome de usuário e um cumprimento adequado. Se for a primeira vez que o

programa é executado, a saída será:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Caso contrário:

```
Welcome back, Eric!
```

Essa é a saída que veremos se o programa já foi executado pelo menos uma vez. Mesmo que os dados desta seção sejam apenas uma única string, o programa executaria perfeitamente com quaisquer dados que possam ser convertidos em uma string formatada em JSON.

Refatoração

Não raro, chegamos a um ponto em que nosso código funciona sem problemas, mas reconhecemos que podemos melhorá-lo se o dividirmos em uma série de funções com tarefas específicas. Esse processo se chama *refatoração*. A refatoração contribui para um código mais limpo, mais fácil de entender e de incrementar.

É possível refatorar *remember_me.py* transferindo a maior parte de sua lógica para uma ou mais funções. Como o foco de *remember_me.py* está em cumprimentar o usuário, passaremos todo o nosso código existente para uma função chamada greet_user():

remember_me.py

```
from pathlib import Path
import json

def greet_user():
    """Cumprimenta o usuário pelo nome"""
    path = Path('username.json')
    if path.exists():
        contents = path.read_text()
        username = json.loads(contents)
        print(f"Welcome back, {username}!")
    else:
        username = input("What is your name? ")
        contents = json.dumps(username)
        path.write_text(contents)
```

```
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

Agora, como estamos usando uma função, reescrevemos os comentários em uma docstring que retrata como o programa funciona atualmente 1. Trata-se de um arquivo um pouco mais limpo. Contudo, a função greet_user() está fazendo mais do que apenas cumprimentar o usuário – também está acessando um nome de usuário armazenado, se existir, e solicitando um novo nome de usuário, se não existir.

Refatoraremos greet_user() para que não execute tantas tarefas diferentes. Vamos começar passando o código para acessar um nome de usuário armazenado para uma função separada:

```
from pathlib import Path
import json

def get_stored_username(path):
1    """Obtém o nome de usuário armazenado, se disponível"""
    if path.exists():
        contents = path.read_text()
        username = json.loads(contents)
        return username
    else:
2        return None

def greet_user():
    """Cumprimenta o usuário pelo nome"""
    path = Path('username.json')
    username = get_stored_username(path)
3    if username:
        print(f"Welcome back, {username}!")
    else:
        username = input("What is your name? ")
        contents = json.dumps(username)
        path.write_text(contents)
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

A função nova get_stored_username() 1 tem um propósito claro, como mencionado na docstring. Essa função acessa um nome de usuário

armazenado e o retorna, se encontrar um. Se o path passado para `get_stored_username()` não existir, a função retorna `None` 2. Trata-se de uma prática recomendada: uma função deve retornar o valor esperado, ou deve retornar `None`. Isso nos possibilita realizar um teste simples com o valor de retorno da função. Exibimos uma mensagem de boas-vindas ao usuário se a tentativa de acessar um nome de usuário for bem-sucedida 3 e, se não for, solicitamos um novo nome de usuário.

Devemos fatorar mais um bloco de código de `greet_user()`. Se o nome de usuário não existir, devemos passar o código que solicita um novo nome de usuário para uma função específica:

```
from pathlib import Path
import json

def get_stored_username(path):
    """Obtém o nome de usuário armazenado, se disponível"""
    -- trecho de código omitido --

def get_new_username(path):
    """Solicita um novo nome de usuário"""
    username = input("What is your name? ")
    contents = json.dumps(username)
    path.write_text(contents)
    return username

def greet_user():
    """Cumprimenta o usuário pelo nome"""
    path = Path('username.json')
1   username = get_stored_username(path)
    if username:
        print(f"Welcome back, {username}!")
    else:
2       username = get_new_username(path)
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

Cada função dessa versão final de *remember_me.py* tem um propósito específico e claro. Ao chamarmos `greet_user()`, a função exibe uma mensagem adequada: ou dá as boas-vindas mais uma

vez ao usuário existente ou cumprimenta um novo usuário. A função faz isso chamando get_stored_username() 1, responsável apenas por acessar um nome de usuário armazenado, se existir. Por último, se necessário, greet_user() chama get_new_username() 2, responsável apenas por obter um novo nome de usuário e armazená-lo. Essa compartimentação de tarefas é parte imprescindível para escrever um código nítido, que será fácil de manter e incrementar.

FAÇA VOCÊ MESMO

10.11 Número favorito: Desenvolva um programa que solicite o número favorito do usuário. Use json.dumps() para armazenar esse número em um arquivo. Escreva um programa separado que leia esse valor e exiba a mensagem: "Eu sei o seu número favorito! É _____".

10.12 Relembrando o número favorito: Combine os dois programas que escreveu no Exercício 10.11 em um arquivo. Se o número já estiver armazenado, informe o número favorito ao usuário. Caso contrário, solicite o número favorito do usuário e armazene-o em um arquivo. Execute o programa duas vezes para verificar se funciona.

10.13 Dicionário do usuário: O exemplo *remember_me.py* armazena apenas uma informação, o nome de usuário. Incremente esse exemplo solicitando mais duas informações sobre o usuário e armazene todas as informações coletadas em um dicionário. Escreva esse dicionário em um arquivo com json.dumps(), e o releia usando json.loads(). Exiba um resumo mostrando exatamente o que seu programa relembra sobre o usuário.

10.14 Verificando usuário: A listagem final de *remember_me.py* pressupõe que o usuário já forneceu seu nome de usuário ou que o programa está sendo executado pela primeira vez. Devemos modificá-lo, caso o usuário atual não seja a pessoa que usou o programa pela última vez.

Antes de exibir uma mensagem de boas-vindas em greet_user(), pergunte ao usuário se o seu nome está correto. Caso contrário, chame get_new_username() para obter o nome de usuário correto.

Recapitulando

Neste capítulo, aprendemos a trabalhar com arquivos. Aprendemos a ler todo o conteúdo de um arquivo e, em seguida, analisar o conteúdo uma linha de cada vez, se necessário. Vimos como escrever a quantidade de texto que quisermos em um arquivo e também como lidar com as exceções que você provavelmente verá em seus programas. Por fim, aprendemos como armazenar

estruturas de dados Python para que possamos salvar as informações fornecidas pelo usuário. Assim, os usuários não precisam fazer tudo de novo sempre que executarem um programa.

No Capítulo 11, veremos formas eficientes de testar o código. Isso o ajudará a confiar que o código que desenvolve está correto e a identificar bugs introduzidos conforme incrementa os programas que escreve.

CAPÍTULO 11

Testando seu código

Ao escrevermos uma função ou uma classe, podemos também escrever testes para esse código. Os testes evidenciam que nosso código funciona devidamente em resposta a todos os tipos de entrada às quais foi desenvolvido para receber. Com os testes, podemos ter segurança de que nosso código executará adequadamente conforme mais pessoas começam a usar nossos programas. É possível também testar códigos novos à medida que os adicionamos, garantindo, assim, que nossas mudanças não prejudiquem o comportamento existente do programa. Como todos os programadores cometem erros, é necessário que todo programador teste regularmente seu código a fim de detectar problemas antes que sejam detectados pelos usuários.

Neste capítulo, aprenderemos a testar o código usando o framework pytest. A biblioteca do pytest é uma coleção de ferramentas que nos ajuda a escrever nossos primeiros testes de forma rápida e simples, ao mesmo tempo em que nos auxilia com nossos testes à medida que ficam cada vez mais complexos, como nossos projetos. Por padrão, o pytest não vem instalado no Python. Assim, veremos como instalar bibliotecas externas. Quando sabemos instalar bibliotecas externas, um novo horizonte com ampla variedade de código bem desenvolvidos surge. Com essas bibliotecas, podemos trabalhar com uma infinidade imensa de tipos de projetos.

Aprenderemos a criar uma série de testes e verificar se cada conjunto de entradas resulta na saída desejada. Veremos exemplos de testes que passam e não passam e aprenderemos como um teste

que falha pode nos ajudar a melhorar o código. Aprenderemos a testar funções e classes e começaremos a entender quantos testes devemos escrever para um projeto.

Instalando o pytest com pip

Apesar de o Python incluir muitas funcionalidades na biblioteca padrão, os desenvolvedores Python também dependem muito de pacotes de terceiros. Um *pacote de terceiros* é uma biblioteca desenvolvida fora do núcleo de desenvolvimento da linguagem Python. Desde então, algumas bibliotecas populares de terceiros acabam sendo incluídas na biblioteca padrão da linguagem e na maioria das instalações Python. Isso acontece com mais frequência com bibliotecas que provavelmente não passarão por muitas mudanças posteriores, após seus bugs iniciais serem resolvidos. Esses tipos de bibliotecas podem evoluir no mesmo ritmo que a linguagem como um todo

No entanto, muitos pacotes são mantidos fora da biblioteca padrão para que possam ser desenvolvidos em uma linha do tempo independente da própria linguagem. Esses pacotes costumam ser atualizados com mais frequência do que seriam se estivessem vinculados ao cronograma de desenvolvimento do Python. Isso vale para o pytest e para a maioria das bibliotecas que usaremos na segunda metade deste livro. Não confie cegamente em todos os pacotes de terceiros, mas também não fique desanimado pelo fato de que muitas funcionalidades importantes são implementadas por meio desses pacotes.

Atualizando o pip

O Python vem com uma ferramenta chamada pip, usada para instalar pacotes de terceiros. Como ajuda a instalar pacotes de recursos externos, o pip é atualizado com frequência para resolver possíveis problemas de segurança. Assim sendo, vamos começar

atualizando o pip.

Abra uma nova janela de terminal e digite o seguinte comando:

```
$ python -m pip install --upgrade pip
1 Requirement already satisfied: pip in /.../python3.11/site-packages (22.0.4)
  -- trecho de código omitido --
2 Successfully installed pip-22.1.2
```

A primeira parte deste comando, **python -m pip**, informa ao Python para executar o módulo pip. A segunda parte, **install --upgrade**, solicita que o pip atualize um pacote já instalado. A última parte, **pip**, especifica qual pacote de terceiros deve ser atualizado. A saída mostra que minha versão atual do pip, version 22.0.4 1, foi substituída pela versão mais recente no momento em que eu escrevia este livro, 22.1.2 2.

Podemos utilizar esse comando para atualizar qualquer pacote de terceiros instalado em nosso sistema:

```
$ python -m pip install --upgrade nome_pacote
```

NOTA *Se estiver usando Linux, o pip poderá não estar incluído em sua instalação do Python. Se receber um erro ao tentar atualizar o pip, confira as instruções no Apêndice A.*

Instalando o pytest

Agora que o **pip** está atualizado, podemos instalar o pytest:

```
$ python -m pip install --user pytest
Collecting pytest
  -- trecho de código omitido --
Successfully installed attrs-21.4.0 iniconfig-1.1.1 ...pytest-7.x.x
```

Dessa vez, ainda estamos usando o comando core **pip install**, sem a flag **--upgrade**. Em vez disso, estamos utilizando a flag **--user**, que instrui o Python para instalar esse pacote apenas no usuário atual. A saída mostra que a versão mais recente do pytest foi instalada com sucesso, com uma série de outros pacotes dos quais o pytest depende.

É possível usar o seguinte comando para instalar muitos pacotes de

terceiros:

$ **python -m pip install --user *package_name***

NOTA *Caso tenha alguma dificuldade em executar esse comando, tente executar o mesmo comando sem a opção* --user*.*

Testando uma função

Para aprender a testar, precisamos de código. Vejamos uma simples função que recebe um primeiro nome e um sobrenome e retorna um nome completo elegantemente formatado:

name_function.py

```
def get_formatted_name(first, last):
    """Gera um nome completo, elegantemente formatado"""
    full_name = f"{first} {last}"
    return full_name.title()
```

A função get_formatted_name() combina o primeiro e o último nome com um espaço entre eles para formar um nome completo e, em seguida, converte as primeiras letras do nome em maiúsculas e retorna o nome completo. Para verificar se get_formatted_name() funciona, criaremos um programa que use essa função. O programa *names.py* possibilita que os usuários digitem um primeiro nome e sobrenome e vejam um nome completo, elegantemente formatado:

names.py

```
from name_function import get_formatted_name

print("Enter 'q' at any time to quit.")
while True:
    first = input("\nPlease give me a first name: ")
    if first == 'q':
        break
    last = input("Please give me a last name: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print(f"\tNeatly formatted name: {formatted_name}.")
```

Esse programa importa get_formatted_name() de *name_function.py*. O usuário consegue fornecer uma série de nomes próprios e sobrenomes e visualizar os nomes completos formatados que são gerados:

```
Enter 'q' at any time to quit.

Please give me a first name: janis
Please give me a last name: joplin
      Neatly formatted name: Janis Joplin.

Please give me a first name: bob
Please give me a last name: dylan
       Neatly formatted name: Bob Dylan.

Please give me a first name: q
```

Aqui, podemos ver que os nomes gerados estão corretos. Mas digamos que queremos modificar get_formatted_name() para que também possa lidar com nomes do meio. Ao fazermos isso, queremos ter certeza de que não prejudicamos a forma como a função lida com nomes que têm somente um primeiro nome e um sobrenome. Poderíamos testar nosso código executando *names.py* e inserindo um nome como Janis Joplin sempre que modificássemos get_formatted_name(), mas isso ficaria cansativo. Felizmente, o pytest fornece uma maneira eficiente de automatizar o teste da saída de uma função. Se automatizarmos o teste de get_formatted_name(), podemos sempre estar seguros de que a função funcionará quando recebermos os tipos de nomes para os quais escrevemos os testes.

Testes unitários e casos de teste

Existe uma ampla variedade de abordagens para testar um software. Um dos tipos mais simples de teste é um teste unitário. Um *teste unitário* averigua se um aspecto específico do comportamento de uma função está correto. Um *caso de teste* é uma coleção de testes unitários que, juntos, provam que uma função se comporta como deveria, dentro de toda a gama de situações que esperamos que a função resolva.

Um bom caso de teste considera todos os tipos possíveis de entrada que uma função pode receber e abrange testes para representar cada uma dessas situações. Um caso de teste com *cobertura completa* abrange uma gama completa de testes unitários, compreendendo todas as formas possíveis de se usar uma função. Alcançar cobertura total em um grande projeto pode ser um desafio e tanto. Não raro, escrever testes para os comportamentos críticos do código já é o bastante. Recomenda-se cobertura completa apenas se o projeto começar a ser amplamente usado.

Um teste que passa

Com o pytest, podemos facilmente escrever nosso primeiro teste. Vamos escrever uma única função de teste. A função de teste chamará a função que estamos testando, e faremos uma asserção sobre o valor retornado. Se nossa asserção estiver correta, o teste passa; caso contrário, o teste falha.

Vejamos o primeiro teste da função get_formatted_name():

test_name_function.py

```
from name_function import get_formatted_name

1 def test_first_last_name():
    """Nomes como 'Janis Joplin' funcionam?"""
2   formatted_name = get_formatted_name('janis', 'joplin')
3   assert formatted_name == 'Janis Joplin'
```

Antes de executarmos o teste, analisaremos mais detalhadamente essa função. O nome de um arquivo de teste é importante; deve começar com *test_*. Quando solicitamos que execute os testes que escrevemos, o pytest procurará por qualquer arquivo que comece com *test_*, e executará todos os testes que encontrar nesse arquivo.

No arquivo de teste, primeiro, importamos a função que queremos testar: get_formatted_name(). Em seguida, definimos uma função de teste: nesse caso, test_first_last_name() 1. Temos um bom motivo para usar um nome de função mais extenso do que os que já usamos. Primeiro, as funções de teste precisam começar com a palavra *test*,

seguida por um underscore. Qualquer função que comece com test_ será *detectada* pelo pytest e executada como parte do processo de teste.

Além do mais, os nomes dos testes devem ser mais extensos e descritivos do que um típico nome de função. Nunca chamamos uma função sozinha; o pytest encontrará a função e a executará para você. Os nomes das funções de teste devem ser extensos o suficiente para que, se virmos o nome da função em um relatório de teste, possamos ter uma boa noção de qual comportamento estava sendo testado.

Em seguida, chamamos a função que estamos testando 2. Aqui, chamamos get_formatted_name() com os argumentos 'janis' e 'joplin', do mesmo jeito que fizemos quando executamos *names.py*. Atribuímos o valor de retorno dessa função a formatted_name.

Por último, fizemos uma asserção 3. Uma *asserção* é uma afirmação sobre uma condição. Aqui, estamos afirmando que o valor de formatted_name deve ser 'Janis Joplin'.

Executando um teste

Se executarmos o arquivo *test_name_function.py* diretamente, não receberemos nenhuma saída porque nunca chamamos a função de teste. Ao contrário, faremos com que o pytest execute o arquivo de teste para nós.

Para tal, abra uma janela de terminal e navegue até a pasta que contém o arquivo de teste. Caso esteja usando o VS Code, pode abrir a pasta que contém o arquivo de teste e usar o terminal integrado na janela do editor. Na janela do terminal, digite o comando **pytest**. Você deve ver isso:

```
  $ pytest
  ========================== test session starts
  ==========================
1 platform darwin -- Python 3.x.x, pytest-7.x.x, pluggy-1.x.x
2 rootdir: /.../python_work/chapter_11
3 collected 1 item
```

```
4 test_name_function.py .                                [100%]
============================ 1 passed in 0.00s
============================
```

Vamos tentar compreender essa saída. Antes de mais nada, vemos algumas informações sobre o sistema em que o teste está sendo executado 1. Estou executando o teste em um sistema macOS. Ou seja, você pode ver algumas saídas diferentes aqui. Mais importante ainda, podemos ver quais versões do Python, do pytest e de outros pacotes estão sendo usados para executar o teste.

Em seguida, vemos o diretório onde o teste está sendo executado 2: no meu caso, *python_work/chapter_11*. É possível ver que o pytest encontrou um teste para executar 3, e é possível ver o arquivo de teste que está sendo executado 4. O único ponto após o nome do arquivo nos informa que um único teste passou e o 100% mostra claramente que todos os testes foram executados. Um projeto grande pode ter centenas ou milhares de testes, e os pontos e o indicador de porcentagem concluída podem ajudar a monitorar o progresso geral da execução do teste

A última linha nos informa que um teste passou, e levou menos de 0,01 segundo para executar o teste.

Essa saída indica que a função get_formatted_name() sempre funcionará para nomes com um primeiro nome e sobrenome, a menos que modifiquemos a função. Ao modificar get_formatted_name(), podemos executar o este teste novamente. Se o teste passar, sabemos que a função ainda funcionará para nomes como Janis Joplin.

NOTA *Se não tiver certeza de como navegar para o local certo no terminal, confira "Executando programas Python em um terminal" na página 43. Além do mais, caso veja uma mensagem de que o comando* pytest *não foi encontrado, use o comando* python -m pytest*.*

Um teste que falha

Como é um teste que falha? Modificaremos get_formatted_name() para que possa lidar com nomes do meio, mas faremos isso de um jeito que a função lance um erro para nomes com somente um primeiro nome e sobrenome, como Janis Joplin.

Vejamos uma nova versão de get_formatted_name() que requer um argumento para nome do meio:

name_function.py

```
def get_formatted_name(first, middle, last):
    """Gera um nome completo, elegantemente formatado"""
    full_name = f"{first} {middle} {last}"
    return full_name.title()
```

Essa versão deve funcionar para pessoas com nomes do meio, mas quando a testamos, verificamos que quebramos a função para pessoas com apenas um primeiro nome e sobrenome.

Dessa vez, a execução do **pytest** fornece a seguinte saída:

```
$ pytest
========================== test session starts
==========================
-- trecho de código omitido --
 1 test_name_function.py F                                [100%]
 2 ================================ FAILURES
================================
 3 _____________________ test_first_last_name _____________________
     def test_first_last_name():
         """Nomes como 'Janis Joplin' funcionam?"""
 4 >       formatted_name = get_formatted_name('janis', 'joplin')
 5 E       TypeError: get_formatted_name() missing 1 required positional
             argument: 'last'

test_name_function.py:5: TypeError
======================== short test summary info
========================
FAILED test_name_function.py::test_first_last_name - TypeError:
    get_formatted_name() missing 1 required positional argument: 'last'
========================== 1 failed in 0.04s
==========================
```

Aqui, temos muitas informações, porque precisamos saber muitas

coisas quando um teste falha. O primeiro item da saída é um único F 1, que nos informa que um teste falhou. Em seguida, vemos uma seção que foca FAILURES 2, porque os testes com falha são geralmente o foco mais importante em uma execução de teste. Depois, vemos que test_first_last_name() foi a função de teste que falhou 3. Um colchete angular 4 indica a linha de código que causou a falha do teste. O E na próxima linha 5 mostra o erro real que causou a falha: um TypeError devido a um argumento posicional obrigatório ausente, last. Podemos ver que no final, as informações mais importantes são repetidas em um resumo menor. Assim, quando estivermos executando muitos testes, podemos ter uma noção rápida de quais testes falharam e por quê.

Respondendo a um teste que falhou

O que fazer quando um teste falha? Partindo do princípio que você esteja verificando as condições corretas, um teste que passou significa que a função está se comportando devidamente, e um teste que falhou significa que há um erro no novo código escrito. Assim, quando um teste falhar, não altere o teste. Se fizer isso, seus testes podem até passar, mas qualquer código que chame sua função como o teste, de repente deixará de funcionar. Em vez disso, corrija o código que está causando a falha do teste. Examine as alterações que acabou de fazer na função e entenda como prejudicaram o comportamento desejado.

Nesse caso, get_formatted_name() costumava exigir somente dois parâmetros: um primeiro nome e um sobrenome. Agora é necessário um primeiro nome, um nome do meio e um sobrenome. A inclusão do parâmetro obrigatório do nome do meio prejudicou o comportamento original de get_formatted_name(). Aqui, a melhor opção é tornar o nome do meio opcional. Ao fazer isso, nosso teste para nomes como Janis Joplin deve passar novamente, e devemos ser capazes de aceitar nomes do meio também. Mudaremos get_formatted_name() para que os nomes do meio sejam opcionais e, em

seguida, executaremos o caso de teste mais uma vez. Se for aprovado, passaremos a garantir que a função lide adequadamente com os nomes do meio.

Para tornar os nomes do meio opcionais, movemos o parâmetro middle para o final da lista de parâmetros na definição da função e lhe atribuímos um valor default vazio. Adicionamos também um teste if que cria corretamente o nome completo, dependendo se um nome do meio é fornecido:

name_function.py

```
def get_formatted_name(first, last, middle=''):
    """Gera um nome completo, elegantemente formatado"""
    if middle:
        full_name = f"{first} {middle} {last}"
    else:
        full_name = f"{first} {last}"
    return full_name.title()
```

Nessa nova versão de get_formatted_name(), o nome do meio é opcional. Se um nome do meio for passado para a função, o nome completo conterá um primeiro nome, nome do meio e sobrenome. Caso contrário, o nome completo será composto somente do primeiro nome e sobrenome. Agora, a função deve funcionar para ambos os tipos de nomes. Para descobrir se a função ainda funciona para nomes como Janis Joplin, executaremos o teste novamente:

```
$ pytest
========================== test session starts
==========================
-- trecho de código omitido --
test_name_function.py .                          [100%]
========================== 1 passed in 0.00s
==========================
```

Agora o teste passou. É o ideal; significa que a função funciona mais uma vez para nomes como Janis Joplin, sem que tenhamos que testá-la manualmente. Corrigir nossa função foi mais fácil, pois o teste que falhou nos ajudou a identificar como o novo código afetou o comportamento existente.

Adicionando testes novos

Agora que sabemos que get_formatted_name() funciona para nomes simples novamente, criaremos um segundo teste para pessoas com um nome do meio. Isso é possível adicionando outra função de teste ao arquivo *test_name_function.py*:

test_name_function.py

```
from name_function import get_formatted_name

def test_first_last_name():
    -- trecho de código omitido --

def test_first_last_middle_name():
    """Nomes como 'Wolfgang Amadeus Mozart' funcionam?"""
1   formatted_name = get_formatted_name(
        'wolfgang', 'mozart', 'amadeus')
2   assert formatted_name == 'Wolfgang Amadeus Mozart'
```

Nomeamos essa função nova como test_first_last_middle_name(). O nome da função deve começar com test_ para que seja executada automaticamente quando executamos o **pytest**. Nomeamos a função para que fique claro qual comportamento de get_formatted_name() estamos testando. Como resultado, se o teste falhar, saberemos imediatamente que tipos de nomes são afetados.

Para testar a função, chamamos get_formatted_name() com um primeiro nome, nome do meio e sobrenome 1 e, depois, fazemos uma asserção 2 de que o nome completo retornado corresponde ao nome completo (primeiro nome, sobrenome e nome do meio) que esperamos. Quando executamos o **pytest** mais uma vez, ambos os testes passam:

```
$ pytest
========================== test session starts
==========================
-- trecho de código omitido --
collected 2 items

1 test_name_function.py ..                                [100%]
========================== 2 passed in 0.01s
==========================
```

Os dois pontos `1` sinalizam que dois testes passaram, o que também fica claro na última linha de saída. Maravilha! Agora sabemos que a função ainda funciona para nomes como Janis Joplin, e podemos ficar seguros de que funcionará também para nomes como Wolfgang Amadeus Mozart.

FAÇA VOCÊ MESMO

11.1 Cidade, país: Crie uma função que aceite dois parâmetros: um nome de cidade e um nome de país. A função deve retornar uma única string com o formato *Cidade*, *País*, como Santiago, Chile. Armazene a função em um módulo chamado *city_functions.py* e salve esse arquivo em uma pasta nova para que o pytest não tente executar os testes que já escrevemos.

Crie um arquivo chamado *test_cities.py* que teste a função que você acabou de escrever. Escreva uma função chamada test_city_country() para verificar se chamar sua função com valores como 'santiago' e 'chile' resulta na string correta. Execute o teste e garanta que test_city_country() passe.

11.2 População: Altere sua função para que exija um terceiro parâmetro, population. Agora, a função deve retornar uma única string no formato *Cidade*, *País – população xxx*, como Santiago, Chile – population 5000000. Execute o teste mais uma vez e garanta que test_city_country() falhe desta vez.

Modifique a função para que o parâmetro population seja opcional. Execute o teste e garanta que test_city_country() passe mais uma vez.

Escreva um segundo teste chamado test_city_country_population() que verifique se você pode chamar sua função com os valores 'santiago', 'chile' e 'population=5000000'. Execute os testes mais uma vez e garanta que esse teste novo passe.

Testando uma classe

Na primeira parte deste capítulo, criamos testes para uma única função. Agora, criaremos testes para uma classe. Usaremos as classes em muitos programas, pois isso ajuda a evidenciar que nossas classes funcionam corretamente. Caso os testes que fizer para uma classe passarem, pode ter certeza de que as melhorias feitas nessa classe não prejudicarão acidentalmente seu comportamento atual.

Variedade de asserções

Até agora vimos somente um tipo de asserção: uma afirmação de

que uma string tem um valor específico. Ao escrever um teste, é possível fazer qualquer afirmação que possa ser expressa como uma instrução condicional. Se a condição for True como esperado, sua asserção sobre como essa parte do programa se comporta será constatada; pode ficar tranquilo de que erros não existem. Se a condição presumida como True for na realidade False, o teste falhará e você saberá que existe um problema para resolver. A Tabela 11.1 demonstra alguns dos tipos mais úteis de asserções usadas que podemos incluir em seus testes iniciais.

Tabela 11.1: Instruções de asserção comumente usadas em testes

Asserção	Afirmação
assert a == b	Afirma que dois valores são iguais.
assert a != b	Afirma que dois valores não são iguais.
assert a	Afirma que a avalia como True
assert not a	Afirma que a avalia como False.
assert *elemento* in *lista*	Afirma que um elemento está em uma lista.
assert *elemento* not in *lista*	Afirma que um elemento não está em uma lista.

Trata-se de apenas alguns exemplos; qualquer coisa que possa ser expressa como uma instrução condicional pode ser incluída em um teste.

Uma classe para testar

Testar uma classe é parecido com testar uma função, já que grande parte do trabalho envolve testar o comportamento dos métodos na classe. No entanto, como existem algumas diferenças, escreveremos uma classe para testar. Imagine uma classe que ajuda a administrar pesquisas anônimas:

survey.py

```
class AnonymousSurvey:
    """Coleta respostas anônimas para uma pergunta da pesquisa"""

1   def __init__(self, question):
        """Armazena uma pergunta e prepare para armazenar respostas"""
        self.question = question
```

```
        self.responses = []

2   def show_question(self):
        """Mostra a pergunta da pesquisa"""
        print(self.question)

3   def store_response(self, new_response):
        """Armazena uma única resposta à pesquisa"""
        self.responses.append(new_response)

4   def show_results(self):
        """Mostra todas as respostas fornecidas"""
        print("Survey results:")
        for response in self.responses:
            print(f"- {response}")
```

Essa classe começa com uma pergunta de pesquisa que você forneceu, 1 e inclui uma lista vazia para armazenar respostas. A classe tem métodos para exibir a pergunta da pesquisa 2, adicionar uma resposta nova à lista de respostas 3 e exibir todas as respostas armazenadas na lista 4. Para criar uma instância a partir dessa classe, tudo o que precisamos fornecer é uma pergunta. Após ter uma instância representando uma pesquisa específica, exiba a pergunta da pesquisa com show_question(), armazene uma resposta com store_response() e mostra os resultados com show_results().

A fim de exemplificar que a classe AnonymousSurvey funciona, escreveremos um programa que use a classe:

language_survey.py

```
from survey import AnonymousSurvey

# Define uma pergunta e faz uma pesquisa
question = "What language did you first learn to speak?"
language_survey = AnonymousSurvey(question)

# Mostra a pergunta e armazena respostas à pergunta
language_survey.show_question()
print("Enter 'q' at any time to quit.\n")
while True:
    response = input("Language: ")
    if response == 'q':
        break
```

```
    language_survey.store_response(response)

# Mostra os resultados da pesquisa
print("\nThank you to everyone who participated in the survey!")
language_survey.show_results()
```

Esse programa define uma pergunta (`"What language did you first learn to speak"`) e cria um objeto `AnonymousSurvey` com essa pergunta. O programa chama `show_question()` para exibir a pergunta e, em seguida, solicita respostas. Cada resposta é armazenada à medida que é recebida. Quando todas as respostas foram inseridas (o usuário insere `q` para sair), `show_results()` exibe os resultados da pesquisa:

```
What language did you first learn to speak?
Enter 'q' at any time to quit.

Language: English
Language: Spanish
Language: English
Language: Mandarin
Language: q

Thank you to everyone who participated in the survey!
Survey results:
- English
- Spanish
- English
- Mandarin
```

Essa classe funciona para uma pesquisa anônima simples, mas digamos que queremos melhorar `AnonymousSurvey` e o módulo em que está, `survey`. Poderíamos permitir que cada usuário fornecesse mais de uma resposta, criar um módulo para enumerar somente as respostas sozinhas de cada um e relatar a quantidade de vezes de cada resposta fornecida ou poderíamos até escrever outra classe para gerenciar pesquisas não anônimas.

O comportamento atual da classe `AnonymousSurvey` correria o risco de ser impactado, caso implementássemos essas mudanças. Por exemplo, é possível que, ao tentar permitir que cada usuário forneça múltiplas respostas, mudássemos sem querer a forma como as respostas individuais são tratadas. Para assegurar que não

prejudicaremos o comportamento existente conforme desenvolvemos esse módulo, podemos criar testes para essa classe.

Testando a classe AnonymousSurvey

Criaremos um teste que averigua um aspecto de como a classe AnonymousSurvey se comporta. Escreveremos um teste a fim de constatar se uma única resposta à pergunta da pesquisa está devidamente armazenada:

test_survey.py

```
from survey import AnonymousSurvey

1 def test_store_single_response():
    """Testa se uma única resposta está devidamente armazenada"""
    question = "What language did you first learn to speak?"
2   language_survey = AnonymousSurvey(question)
    language_survey.store_response('English')
3   assert 'English' in language_survey.responses
```

Começamos importando a classe que queremos testar, AnonymousSurvey. A primeira função de teste averiguará se, quando armazenamos uma resposta à pergunta da pesquisa, a resposta acabará na lista de respostas da pesquisa. Um bom nome descritivo para essa função é test_store_single_response() 1. Se esse teste falhar, saberemos pelo nome da função, no resumo do teste, que houve um problema ao armazenar uma única resposta à pesquisa.

Para testar o comportamento de uma classe, é necessário criar uma instância da classe. Criamos uma instância chamada language_survey 2 com a pergunta "What language did you first learn to speak?". Armazenamos uma única resposta, English, com o método store_response(). Em seguida, verificamos que a resposta foi devidamente armazenada, pela asserção de que English está na lista language_survey.responses 3.

Por padrão, executar o comando **pytest** sem argumentos executará todos os testes que o pytest identificar no diretório atual. Para concentrar os testes em um arquivo, passe o nome do arquivo de teste que quiser executar. Aqui, executaremos apenas um teste que

escrevemos para AnonymousSurvey:

```
$ pytest test_survey.py
========================== test session starts
==========================
-- trecho de código omitido --
test_survey.py .                                    [100%]
=========================== 1 passed in 0.01s
===========================
```

É um bom ponto de partida, mas uma pesquisa só serve de alguma coisa se gerar mais de uma resposta. Vamos averiguar se três respostas podem ser devidamente armazenadas. Para isso, adicionamos outro método a TestAnonymousSurvey:

```
from survey import AnonymousSurvey

def test_store_single_response():
    -- trecho de código omitido --

def test_store_three_responses():
    """Testa se três respostas individuais estão devidamente armazenadas"""
    question = "What language did you first learn to speak?"
    language_survey = AnonymousSurvey(question)
1   responses = ['English', 'Spanish', 'Mandarin']
    for response in responses:
        language_survey.store_response(response)

2   for response in responses:
        assert response in language_survey.responses
```

Nomeamos a função nova como test_store_three_responses(). Criamos um objeto de pesquisa como fizemos em test_store_single_response(). Definimos uma lista com três respostas diferentes 1 e, em seguida, chamamos store_response() para cada uma dessas respostas. Uma vez que as respostas foram armazenadas, escrevemos outro loop e fizemos uma asserção de que cada resposta está agora em language_survey.responses 2.

Ao executarmos o arquivo de teste mais uma vez, ambos os testes (para uma única resposta e para três respostas) passam:

```
$ pytest test_survey.py
========================= test session starts
=========================
-- trecho de código omitido --
test_survey.py ..                                    [100%]
========================== 2 passed in 0.01s
==========================
```

Funcionou perfeitamente. No entanto, como esses testes são um pouco repetitivos, usaremos outro recurso do pytest para torná-los mais eficientes.

Usando fixtures

Em *test_survey.py*, criamos uma instância nova de AnonymousSurvey em cada função de teste. Funciona sem problemas no breve exemplo com o qual estamos trabalhando, mas em um projeto real, com dezenas ou centenas de testes, seria problemático.

Nos testes, uma *fixture* ajuda a definir um ambiente de teste. Em geral, isso significa criar um recurso que é usado por mais de um teste. Criamos uma fixture no pytest escrevendo uma função com o decorator @pytest.fixture. Um *decorator* é uma diretiva inserida pouco antes de uma definição de função; o Python aplica essa diretiva à função antes de ser executada a fim de alterar como o código da função se comporta. Não se preocupe se isso parecer complicado; é possível começar a usar decorators de pacotes de terceiros antes de você mesmo aprender a escrevê-los.

Vamos usar uma fixture para criar uma única instância de pesquisa que pode ser usada em ambas as funções de teste em *test_survey.py*:

```
import pytest
from survey import AnonymousSurvey

@pytest.fixture
def language_survey():
    """Uma pesquisa que estará disponível para todas as funções de teste"""
    question = "What language did you first learn to speak?"
    language_survey = AnonymousSurvey(question)
```

```
    return language_survey

3 def test_store_single_response(language_survey):
    """Testa se uma única resposta está devidamente armazenada"""
4    language_survey.store_response('English')
    assert 'English' in language_survey.responses

5 def test_store_three_responses(language_survey):
    """Testa se três respostas individuais estão devidamente armazenadas"""
    responses = ['English', 'Spanish', 'Mandarin']
    for response in responses:
6        language_survey.store_response(response)

    for response in responses:
        assert response in language_survey.responses
```

Precisamos importar o pytest agora, porque estamos usando um decorator definido no pytest. Aplicamos o decorator @pytest.fixture 1 à função nova language_survey() 2. Essa função cria um objeto AnonymousSurvey e retorna à nova pesquisa.

Observe que as definições de ambas as funções de teste mudaram 3 5; cada função de teste agora tem um parâmetro chamado language_survey. Quando um parâmetro em uma função de teste corresponde ao nome de uma função com o decorator @pytest.fixture, a fixture será executada automaticamente, e o valor de retorno será passado à função de teste. Nesse exemplo, a função language_survey() fornece test_store_single_response() e test_store_three_responses() com uma instância de language_survey.

Não temos nenhum código novo em nenhuma das funções de teste, mas perceba que duas linhas foram removidas de cada função 4 6: a linha que definiu uma pergunta e a linha que criou um objeto AnonymousSurvey.

Ao executarmos o arquivo de teste novamente, ambos os testes ainda passam. Esses testes ajudariam bastante se tentássemos incrementar AnonymousSurvey para lidar com múltiplas respostas para cada pessoa. Após modificar o código para aceitar múltiplas respostas, podemos executar esses testes e garantir que não

afetaram a capacidade de armazenar uma única resposta ou uma série de respostas individuais.

A estrutura acima muito provavelmente parecerá complicada, pois contém alguns dos códigos mais abstratos que já vimos até agora. Não precisa usar as fixtures logo de cara; é melhor escrever testes que tenham muitos códigos repetitivos do que não escrever nenhum teste. Apenas saiba que, quando você escrever testes suficientes para que a repetição atrapalhe, já existe uma maneira consagrada de lidar com a repetição. Além disso, fixtures em exemplos simples como esse não encurtam nem simplificam o entendimento do código. No entanto, em projetos com muitos testes, ou em situações em que são necessárias muitas linhas para criar um recurso usado em múltiplos testes, as fixtures podem melhorar radicalmente o código de teste.

Se quiser escrever uma fixture, escreva uma função que gere o recurso usado por múltiplas funções de teste. Adicione o decorator `@pytest.fixture` à função nova e adicione o nome dessa função como um parâmetro a cada função de teste que usa esse recurso. Assim, seus futuros testes serão mais curtos, fáceis de escrever e manter.

FAÇA VOCÊ MESMO

11.3 Funcionário: Crie uma classe chamada Employee. O método `__init__()` deve aceitar um primeiro nome, um sobrenome e um salário anual e armazenar cada um deles como atributos. Crie um método chamado give_raise() que adiciona US$5.000 ao salário anual por padrão, mas também aceita um valor de aumento diferente.

Escreva um arquivo de teste para Employee com duas funções de teste, test_give_default_raise() e test_give_custom_raise(). Escreva seus testes uma vez sem usar uma fixture e garanta que ambos passem. Em seguida, escreva uma fixture para que você não precise criar uma instância nova de Employee em cada função de teste. Execute os testes mais uma vez e garanta que ambos ainda passem.

Recapitulando

Neste capítulo, aprendemos a escrever testes para funções e classes usando ferramentas no módulo `pytest`. Vimos como escrever funções de teste que averiguam comportamentos específicos que suas

funções e classes devem exibir e como as fixtures podem ser utilizadas para criar eficientemente recursos que podem ser usados em múltiplas funções de teste em um arquivo de teste.

Apesar do teste ser um tópico fundamental para se aprender a programar, muitos programadores iniciantes não são expostos a ele. Como programador iniciante, você não precisa escrever testes para todos os projetos simples que cria. Mas assim que começar a trabalhar em projetos com empenho significativo de desenvolvimento, será necessário testar os comportamentos críticos de suas funções e classes. Desse modo, você ficará mais seguro de que as tarefas novas em seus projetos não comprometerão as partes que já funcionam e terá mais liberdade para melhorar seu código. Caso quebre sem querer uma funcionalidade existente, você saberá no mesmo instante e conseguirá corrigi-la com facilidade. Responder a um teste executado com falha é bem mais fácil do que responder a um bug reportado por um usuário descontente.

Você será mais respeitado por outros programadores se incluir no código alguns testes iniciais. Eles se sentirão mais à vontade de testar seu código e estarão mais dispostos a trabalhar com você em projetos. Caso queira contribuir com um projeto em que outros programadores estão trabalhando, espera-se que você mostre que seu código passa nos testes existentes e, via de regra, espera-se que você escreva testes para qualquer comportamento novo que introduzir no projeto.

Brinque com testes em seu código para se familiarizar com os processos de teste. Escreva testes para os comportamentos mais críticos de suas funções e classes, mas não tenha como objetivo a cobertura completa em projetos iniciais, a menos que tenha uma razão específica.

PARTE II

Projetos

Parabéns! Agora você conhece o bastante sobre o Python para começar a desenvolver projetos interativos e significativos. Criar os próprios projetos possibilitará que você desenvolva novas habilidades e consolidará seu entendimento dos conceitos apresentados na Parte I.

A Parte II engloba três tipos de projetos, e você pode escolher qualquer um, ou todos os projetos, na ordem que bem entender. Vejamos uma breve descrição de cada projeto para ajudá-lo a decidir em qual deles você mergulhará de cabeça.

Invasão Alienígena: desenvolvendo um jogo em Python

No projeto Invasão Alienígena (**Capítulos 12**, **13** e **14**), utilizaremos o pacote Pygame para desenvolver um jogo 2D. O intuito do jogo é abater uma frota de alienígenas conforme espaçonaves descem pela tela, em níveis progressivos de velocidade e dificuldade. Ao final do projeto, você terá aprendido habilidades que lhe possibilitarão desenvolver os próprios jogos 2D com o Pygame.

Visualização de dados

Os projetos de Visualização de Dados começam no **Capítulo 15**: você aprenderá a gerar dados e criar uma série de visualizações funcionais e valiosas usando o Matplotlib e o Plotly. No **Capítulo 16**,

você aprenderá a acessar dados de fontes online e a fornecê-los a um pacote de visualização a fim de gerar gráficos de dados meteorológicos e um mapa da atividade global de terremotos. Por último, no **Capítulo 17**, você aprenderá como escrever um programa para fazer download e visualizar dados de modo automático. Aprender como gerar visualizações lhe possibilita explorar o campo da ciência de dados, uma das áreas de programação com maior demanda atualmente.

Aplicações web

No projeto de Aplicações web (**Capítulos 18**, **19** e **20**), você usará o pacote Django para desenvolver uma aplicação simples web, que possibilita aos usuários manter um registro sobre os diferentes tópicos que estão aprendendo. Os usuários criarão uma conta com nome de usuário e senha, digitarão um tópico e, em seguida, imputarão os dados sobre o que estão aprendendo. Implantaremos também nossa aplicação em um servidor remoto para que qualquer pessoa em qualquer lugar do mundo consiga acessá-la.

Após concluir esse projeto, você será capaz de começar a desenvolver suas aplicações simples web e estará pronto para se dedicar a conteúdos mais aprofundados sobre como desenvolver aplicações com o Django.

CAPÍTULO 12

Uma espaçonave que dispara balas

Criaremos um jogo chamado *Invasão Alienígena*! Usaremos o Pygame, coleção de módulos Python divertidos e robustos que lida com gráficos, animações e até sons, facilitando o desenvolvimento de jogos sofisticados. Já que o Pygame se encarrega da maioria das tarefas, como desenhar imagens na tela, podemos focar a lógica de alto nível da dinâmica do jogo.

Neste capítulo, faremos a configuração do Pygame e, em seguida, criaremos uma espaçonave que se movimenta da direita e para a esquerda, abrindo fogo em resposta à entrada do jogador. Nos próximos dois capítulos criaremos uma frota de alienígenas para destruir e, depois, prosseguiremos refinando o jogo, definindo limites para a quantidade de espaçonaves utilizadas e adicionando um scoreboard.

À medida que desenvolve esse jogo, você também aprenderá a lidar com grandes projetos contendo inúmeros arquivos. Vamos refatorar um boa dose de código e gerenciar conteúdos de arquivo para organizar o projeto e tornar o código eficiente.

Desenvolver jogos é a melhor forma de se divertir ao mesmo tempo que aprendemos uma linguagem de programação. É uma grande satisfação jogar um jogo que você mesmo desenvolveu. Ao criar um jogo simples, você aprende como os profissionais da área desenvolvem jogos. À medida que avança na leitura deste capítulo, insira e execute o código a fim de assimilar como cada bloco de

código contribui para a jogabilidade geral. Teste diferentes valores e configurações para entender melhor como refinar as interações em seus jogos.

NOTA Invasão Alienígena *terá inúmeros arquivos diferentes, então crie uma nova pasta chamada* alien_invasion*. Lembre-se de salvar todos os arquivos do projeto nesta pasta, assim suas instruções* import *executarão devidamente.*

Além do mais, caso se sinta à vontade com o controle de versão, talvez queira usá-lo neste projeto. Caso nunca tenha usado o controle de versão antes, confira o Apêndice D para obter uma visão geral.

Planejando seu projeto

Quando criamos um grande projeto, é essencial fazer um planejamento antes mesmo de começar a escrever o código. O planejamento o ajudará a se manter focado, aumentando as probabilidade de você concluir o projeto.

Vamos escrever uma descrição da jogabilidade geral. Apesar de não englobar todos os detalhes do jogo *Invasão Alienígena*, a descrição a seguir fornece uma noção clara de como começar a desenvolver o jogo:

> No jogo *Invasão Alienígena*, o jogador controla uma espaçonave que aparece no centro inferior da tela. O jogador pode manobrar a espaçonave para a direita e para a esquerda, usando as setas do teclado e atirar usando a barra de espaço. Quando o jogo começa, uma frota de alienígenas toma conta do céu, deslocando-se para cima, para baixo e para os lados da tela. O jogador abre fogo e destrói os alienígenas. Se o jogador destruir todos os alienígenas, aparece uma nova frota que se desloca mais rápido que a frota anterior. Se algum alienígena abater a espaçonave do jogador ou a parte inferior da tela, o jogador perde a espaçonave. Se o jogador perder três espaçonaves, o jogo

termina.

Na primeira fase de desenvolvimento, criaremos uma espaçonave que pode se movimentar para a direita e para a esquerda, conforme o jogador pressiona as setas do teclado, e abrir fogo quando o jogador pressionar a barra de espaço. Após definirmos esse comportamento, podemos criar os alienígenas e refinar a jogabilidade.

Instalando o Pygame

Antes de começar a programar, instale o Pygame. Faremos a instalação da mesma forma que fizemos com o pytest no Capítulo 11: com o pip. Caso tenha pulado o Capítulo 11 ou precise refrescar a memória sobre o que é pip, veja a seção "Instalando o pytest com pip" na página 263.

Para instalar o Pygame, digite o seguinte comando no prompt do terminal:

```
$ python -m pip install --user pygame
```

Se você usar um comando diferente de **python** para executar programas ou iniciar uma sessão de terminal, como **python3**, não se esqueça de usar esse comando.

Iniciando o projeto do jogo

Para desenvolver o jogo, começaremos criando uma janela vazia no Pygame. Mais tarde, desenharemos os elementos do jogo, como a espaçonave e os alienígenas, nesta janela. Faremos também nosso jogo responder à entrada do usuário, definiremos a cor do background e faremos o upload da imagem de uma espaçonave.

Criando uma janela Pygame e respondendo à entrada do usuário

Criaremos uma janela vazia Pygame por meio de uma classe para

representar o jogo. Em seu editor de texto, crie um novo arquivo e salve-o como *alien_invasion.py*; depois digite o seguinte:

alien_invasion.py

```
import sys

import pygame

class AlienInvasion:
    """Classe geral para gerenciar ativos e comportamento do jogo"""

    def __init__(self):
        """Inicializa o jogo e crie recursos do jogo"""
1       pygame.init()

2       self.screen = pygame.display.set_mode((1200, 800))
        pygame.display.set_caption("Alien Invasion")

    def run_game(self):
        """Inicia o loop principal do jogo"""
3       while True:
            # Observa eventos de teclado e mouse
4           for event in pygame.event.get():
5               if event.type == pygame.QUIT:
                    sys.exit()

            # Deixa a tela desenhada mais recente visível
6           pygame.display.flip()

if __name__ == '__main__':
    # Cria uma instância do jogo e execute o jogo
    ai = AlienInvasion()
    ai.run_game()
```

Primeiro, importamos os módulos `sys` e `pygame`. O módulo `pygame` tem a funcionalidade necessária para desenvolvermos um jogo. Usaremos ferramentas no módulo `sys` para encerrar jogo quando o jogador desistir de jogar.

O jogo *Invasão Alienígena* começa como uma classe chamada `AlienInvasion`. No método `__init__()`, a função `pygame.init()` inicializa as configurações de background de que o Pygame precisa para funcionar adequadamente `1`. Depois, chamamos

pygame.display.set_mode() para criar uma janela de exibição 2, na qual desenharemos todos os elementos gráficos do jogo. O argumento (1200, 800) é uma tupla que define as dimensões da janela do jogo, que terá 1.200 pixels de largura por 800 pixels de altura. (Podemos ajustar esses valores dependendo do tamanho da tela.) Atribuímos esta janela de exibição ao atributo self.screen, de modo que esteja disponível em todos os métodos da classe.

O objeto que atribuímos a self.screen é chamado de superfície. No Pygame, uma *superfície* é a parte da tela em que um elemento do jogo pode ser exibido. Cada elemento do jogo, como um alienígena ou uma espaçonave, é a própria superfície. A superfície retornada pelo display.set_mode() representa toda a janela do jogo. Ao habilitarmos o loop de animação do jogo, essa superfície será redesenhada a cada passagem pelo loop, de modo que possa ser atualizada com quaisquer mudanças desencadeadas pela entrada do usuário.

O jogo é controlado pelo método run_game(). Este método contém um loop while 3, executado continuamente. O loop while contém um loop de eventos e um código que gerencia as atualizações de tela. Um *evento* é uma ação que o usuário executa durante o jogo, como pressionar uma tecla ou mover o mouse. Para que nosso programa responda a eventos, escrevemos um *loop de eventos* que possa '*ouvir*' eventos e realizar tarefas apropriadas dependendo dos tipos de eventos ocorridos. O loop for 4 aninhado dentro do loop while é um loop de eventos.

Para acessar os eventos que o Pygame detecta, usaremos a função pygame.event.get(). Essa função retorna uma lista de eventos ocorridos desde a última vez que foi chamada. Qualquer evento de teclado ou mouse fará com que este loop for seja executado. Dentro do loop, escreveremos uma série de instruções if para detectar e responder a eventos específicos. Por exemplo, quando o jogador clica no botão fechar da janela do jogo, um evento pygame.QUIT é detectado e

chamamos o sys.exit() para sair do jogo 5.

A chamada para pygame.display.flip() 6 informa ao Pygame para deixar a tela desenhada mais recente visível. Nesse caso, o Pygame simplesmente desenha uma tela vazia em cada passagem pelo loop while, deletando a tela antiga, de modo que somente a nova tela fique visível. Quando movemos os elementos do jogo, o pygame.display.flip() atualiza continuamente a exibição, mostrando as novas posições dos elementos do jogo e ocultando as antigas, criando a ilusão de movimento suave.

No final do arquivo, criamos uma instância do jogo e chamamos o run_game(). Colocamos o run_game() em um bloco if que só executa se o arquivo for chamado diretamente. Ao executar o arquivo *alien_invasion.py*, você deverá ver uma janela vazia do Pygame.

Controlando a taxa de frame

De preferência, os jogos devem executar na mesma velocidade, ou na mesma *taxa de frames*, em todos os sistemas. Controlar a taxa de frames de um jogo que pode ser executado em diversos sistemas é uma questão complexa, mas o Pygame viabiliza um modo relativamente simples de fazer isso. Criaremos um relógio e garantiremos que funcione uma vez em cada passagem pelo loop principal. Sempre que o loop for processado mais rápido do que a taxa que definimos, o Pygame calculará o período correto de tempo para pausa, de modo que o jogo seja executado em uma taxa consistente.

Vamos definir o relógio no método __init__():

alien_invasion.py

```
    def __init__(self):
        """ Inicializa o jogo e cria recursos de jogo"""
        pygame.init()
        self.clock = pygame.time.Clock()
        -- trecho de código omitido --
```

Após inicializar o pygame, criamos uma instância da classe Clock, a

partir do módulo pygame.time. Depois, vamos fazer o relógio funcionar no final do loop while em run_game():

```
def run_game(self):
    """ Inicia o loop principal do jogo"""
    while True:
        -- trecho de código omitido --
        pygame.display.flip()
        self.clock.tick(60)
```

O método tick() recebe um argumento: a taxa de frames do jogo. Aqui, estou usando um valor de 60, logo, o Pygame fará o possível para que o loop seja executado exatamente 60 vezes por segundo.

NOTA *O relógio do Pygame deve ajudar o jogo a ser executado de forma consistente na maioria dos sistemas. Caso o jogo execute de forma menos consistente em seu sistema, tente inserir valores diferentes para a taxa de frames. Se não conseguir encontrar uma boa taxa de frames em seu sistema, é possível omitir o relógio e ajustar as configurações do jogo para que funcione bem em seu sistema.*

Definindo a cor do background

O Pygame cria uma tela preta por padrão, e isso é a maior chatice. Definiremos uma cor de blackground diferente. Faremos isso no final do método __init__().

alien_invasion.py

```
def __init__(self):
    -- trecho de código omitido --
    pygame.display.set_caption("Alien Invasion")

    # Define a cor do background.
1   self.bg_color = (230, 230, 230)

def run_game(self):
    -- trecho de código omitido --
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
```

```
            # Redesenha a tela durante cada passagem pelo loop
2           self.screen.fill(self.bg_color)

            # Deixa a tela desenhada mais recente visível
            pygame.display.flip()
            self.clock.tick(60)
```

No Pygame, as cores são especificadas como cores RGB: uma combinação de vermelho, verde e azul. Cada valor de cor pode variar de 0 a 255. O valor da cor (255, 0, 0) é vermelho, (0, 255, 0) é verde e (0, 0, 255) é azul. É possível combinar diferentes valores RGB para criar até 16 milhões de cores. O valor da cor (230, 230, 230) combina quantidades iguais de vermelho, azul e verde, gerando uma cor de background cinza claro. Atribuímos esta cor a self.bg_color 1.

Preenchemos a tela com a cor de background usando o método fill() 2, que funciona em uma superfície e recebe somente um argumento: uma cor.

Criando uma classe Settings

Sempre que introduzimos novas funcionalidades no jogo, em geral, definimos também algumas configurações novas. Em vez de definir configurações em todo o código, escrevemos um módulo chamado settings, contendo uma classe chamada Settings para armazenar todos esses valores em um lugar só. Essa abordagem nos possibilita trabalhar com somente um objeto settings, sempre que precisarmos acessar uma configuração individual. Facilita também modificar a aparência e o comportamento do jogo conforme crescimento do projeto. Para modificar o jogo, mudaremos os valores relevantes em *settings.py,* que criaremos a seguir, em vez de procurar configurações diferentes ao longo do projeto.

Crie um novo arquivo chamado *settings.py* dentro de sua pasta *alien_invasion* e adicione a classe inicial Settings:

settings.py

```
class Settings:
    """Classe para armazenar as configurações do Jogo Invasão Alienígena"""
```

```
    def __init__(self):
        """Inicializa as configurações do jogo"""
        # Configurações da tela
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)
```

Para criar uma instância de Settings no projeto e usá-la a fim de acessar nossas configurações, precisamos modificar o *alien_invasion.py* desse jeito:

alien_invasion.py

```
-- trecho de código omitido --
import pygame

from settings import Settings

class AlienInvasion:
    """Classe geral para gerenciar ativos e comportamento do jogo"""

    def __init__(self):
        """Inicializa o jogo e crie recursos de jogo"""
        pygame.init()
        self.clock = pygame.time.Clock()
1       self.settings = Settings()

2       self.screen = pygame.display.set_mode(
            (self.settings.screen_width, self.settings.screen_height))
        pygame.display.set_caption("Alien Invasion")

    def run_game(self):
            -- trecho de código omitido --
            # Redesenha a tela durante cada passagem pelo loop.
3           self.screen.fill(self.settings.bg_color)

            # Deixa a tela desenhada mais recente visível
            pygame.display.flip()
            self.clock.tick(60)
-- trecho de código omitido --
```

Importamos Settings para o arquivo principal do programa. Em seguida, criamos uma instância Settings e a atribuímos a self.settings 1, após chamar o pygame.init(). Ao criarmos uma tela 2, usamos os

atributos screen_width e screen_height de self.settings. Depois usamos self.settings para acessar a cor de background ao também preencher a tela 3.

Agora, quando executar o *alien_invasion.py*, você ainda não verá nenhuma mudança, pois tudo o que fizemos foi mover as configurações que já estávamos usando para outro lugar. Já estamos prontos para começar a adicionar novos elementos à tela.

Adicionando a imagem da espaçonave

Vamos adicionar a imagem de uma espaçonave ao nosso jogo. Para desenhar a espaçonave do jogador na tela, faremos o upload de uma imagem e usaremos o método blit() do Pygame para desenhar a imagem.

Ao escolher imagens para seus jogos, preste atenção ao direitos de imagem. A forma mais segura e barata de começar é recorrer a gráficos disponibilizados gratuitamente que podemos usar e modificar, de um site como o https://opengameart.org.

Podemos utilizar quase qualquer tipo de arquivo de imagem em nosso jogo. No entanto, é mais fácil usar o formato bitmap (*.bmp*), já que o Pygame faz o upload de imagens bitmaps por padrão. Ainda que possamos configurar o Pygame para usar outros tipos de arquivo, alguns tipos de arquivo dependem de determinadas bibliotecas de imagens que devem ser instaladas em seu computador. A maioria das imagens que podemos encontrar estão nos formatos *.jpg* ou *.png*, mas podemos convertê-las em bitmaps usando ferramentas como Photoshop, GIMP e Paint.

Preste atenção especial à cor de background na imagem escolhida. Tente encontrar um arquivo com background transparente ou sólido que você consiga substituir por qualquer cor, usando um editor de imagens. O visual do jogo será melhor se a cor de background da imagem corresponder à cor de background do jogo. Outra alternativa é combinar o background do seu jogo com o background

da imagem.

Para o jogo *Invasão Alienígena* é possível usar o arquivo *ship.bmp* (Figura 12.1), disponível nos recursos deste livro em https://ehmatthes.github.io/pcc_3e. A cor do background do arquivo corresponde às configurações que estamos usando neste projeto. Crie uma pasta chamada *images* dentro da pasta principal do projeto *alien_invasion*. Salve o arquivo *ship.bmp* na pasta *images*.

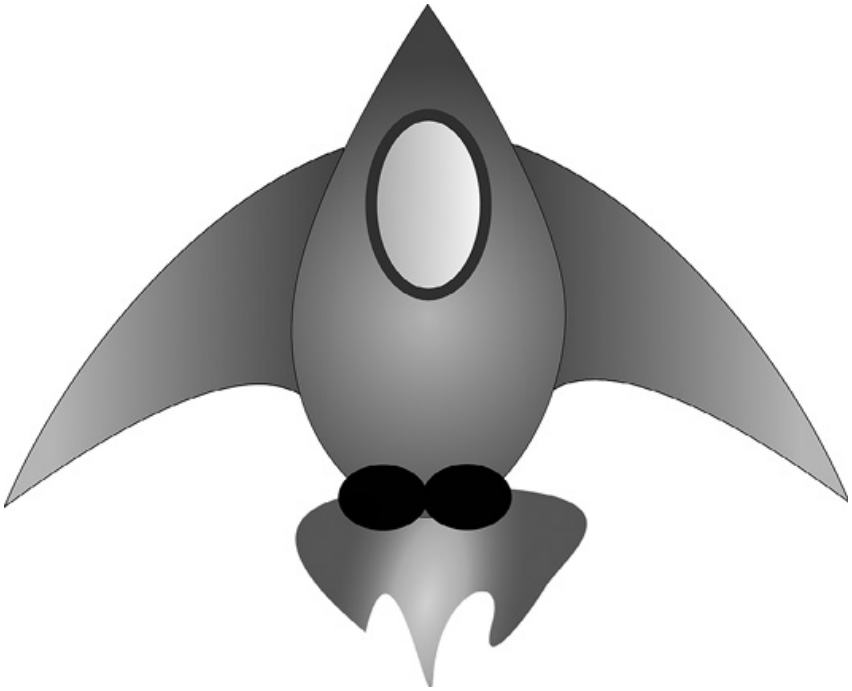


Figura 12.1: A espaçonave para o Jogo Invasão Alienígena.

Criando a classe Ship

Após escolher a imagem para a espaçonave, precisamos exibi-la na tela. A fim de usar nossa espaçonave, vamos criar um módulo novo `ship`, contendo a classe `Ship`. Esta classe será responsável por boa parte do comportamento da espaçonave do jogador:

ship.py

```
import pygame

class Ship:
    """Classe para cuidar da espaçonave"""
```

```
    def __init__(self, ai_game):
        """Inicializa a espaçonave e defina sua posição inicial"""
1       self.screen = ai_game.screen
2       self.screen_rect = ai_game.screen.get_rect()

        # Sobe a imagem da espaçonave e obtém seu rect
3       self.image = pygame.image.load('images/ship.bmp')
        self.rect = self.image.get_rect()

        # Começa cada espaçonave nova no centro inferior da tela
4       self.rect.midbottom = self.screen_rect.midbottom

5   def blitme(self):
        """Desenha a espaçonave em sua localização atual"""
        self.screen.blit(self.image, self.rect)
```

O Pygame é eficiente porque possibilita tratar todos os elementos do jogo como *retângulos (rects*), mesmo que não tenham exatamente o formato de retângulos. Tratar um elemento como um retângulo é eficiente, pois os retângulos são formas geométricas simples. Por exemplo, caso seja necessário descobrir se dois elementos do jogo podem colidir, o Pygame pode fazer isso mais rápido se tratar cada objeto como um retângulo. Essa abordagem normalmente funciona bem o suficiente para que ninguém que jogue o jogo perceba que não estamos trabalhando com a forma exata de cada elemento do jogo. Nesta classe, trataremos a espaçonave e a tela como retângulos.

Importamos o módulo `pygame` antes de definir a classe. O método `__init__()` de `Ship` recebe dois parâmetros: o `self` para referência e uma referência à instância atual da classe `AlienInvasion`. Isso fornecerá à `Ship` acesso a todos os recursos do jogo definidos em `AlienInvasion`. Depois, atribuímos a tela a um atributo de `Ship` `1`, para que possamos acessá-lo facilmente em todos os métodos desta classe. Acessamos o atributo `rect` da tela usando o método `get_rect()` e o atribuímos a `self.screen_rect` `2`. Isso nos possibilita colocar a espaçonave no local correto na tela.

Para subir a imagem, chamamos o `pygame.image.load()` `3` e fornecemos a

localização da imagem da espaçonave. Esta função retorna uma superfície representando a espaçonave, que atribuímos a self.image. Quando a imagem é carregada, chamamos o get_rect() para acessar o atributo rect da superfície da espaçonave, de modo que possamos usá-lo posteriormente para posicionar a espaçonave.

Ao trabalhar com um objeto rect, é possível usar as coordenadas *x* e *y* das bordas superior, inferior, esquerda e direita do retângulo, bem como o centro, para posicionar o objeto. Podemos definir qualquer um desses valores a fim de determinar a posição atual do rect. Ao centralizar um elemento do jogo, trabalhe com os atributos center, centerx ou centery de um rect. Quando estiver trabalhando em uma borda da tela, trabalhe com os atributos top, bottom, left ou right. Há também atributos que combinam essas propriedades, como midbottom, midtop, midleft e midright. Ao ajustar o posicionamento horizontal ou vertical do rect, é possível utilizar somente os atributos x e y, as coordenadas *x* e *y* do canto superior esquerdo. Por causa desses atributos, não precisamos fazer cálculos. Antes, os desenvolvedores tinham que fazer cálculos manualmente. Você usará esses atributos com frequência.

NOTA *No Pygame, a origem (0, 0) está no canto superior esquerdo da tela e as coordenadas aumentam à medida que você desce e se desloca à direita. Em uma tela de 1200×800, a origem está no canto superior esquerdo e o canto inferior direito tem as coordenadas (1200, 800). Essas coordenadas se referem à janela do jogo, não à tela física.*

Vamos posicionar a espaçonave no centro inferior da tela. Para tal, faça com que o valor de self.rect.midbottom corresponda ao atributo midbottom do rect 4 da tela. O Pygame usa esses atributos rect para posicionar a imagem da espaçonave de forma centralizada e horizontal, alinhada à parte inferior da tela.

Por último, definimos o método blitme() 5, que desenha a imagem na tela na posição especificada pelo self.rect.

Desenhando a espaçonave na tela

Agora, vamos atualizar *alien_invasion.py* para criar uma espaçonave e chamar o método blitme() da espaçonave:

alien_invasion.py

```
  -- trecho de código omitido --
  from settings import Settings
  from ship import Ship

  class AlienInvasion:
      """Classe geral para gerenciar ativos e comportamento do jogo"""

      def __init__(self):
          -- trecho de código omitido --
          pygame.display.set_caption("Alien Invasion")

1         self.ship = Ship(self)

      def run_game(self):
              trecho de código omitido -
              # Redesenha a tela durante cada passagem pelo loop
              self.screen.fill(self.settings.bg_color)
2             self.ship.blitme()

              # Deixa a tela desenhada mais recente visível

              pygame.display.flip()
              self.clock.tick(60)
  -- trecho de código omitido --
```

Importamos Ship e criamos uma instância de Ship após a criação da tela 1. Chamar Ship() exige um argumento: uma instância de AlienInvasion. Aqui, o argumento self se refere à instância atual de AlienInvasion. Esse é o parâmetro que dá acesso à Ship aos recursos do jogo, como o objeto screen. Atribuímos esta instância de Ship à self.ship.

Após preencher o background, desenhamos a espaçonave na tela chamando o ship.blitme(), para que a espaçonave apareça em cima do background 2.

Ao executar *alien_invasion.py*, você deverá ver uma tela de jogo vazia com a espaçonave no centro inferior, conforme mostrado na

Figura 12.2.

Figura ١٢.2: Jogo Invasão Alienígena com a espaçonave no centro inferior da tela.

Refatoração: métodos _check_events() e _update_screen()

Em projetos grandes, refatoraremos com frequência o código que escrevemos antes de adicionar mais código. A refatoração simplifica a estrutura do código escrito, facilitando seu desenvolvimento. Nesta seção, vamos dividir o método run_game(), que está ficando extenso, em dois métodos auxiliares. Um método auxiliar funciona dentro de uma classe, mas não deve ser utilizado pelo código fora da classe. No Python, um único underscore à esquerda sinaliza um método auxiliar.

Método _check_events()

Moveremos o código que gerencia eventos para um método separado chamado _check_events(). Isso simplificará o run_game() e

isolará o loop de gerenciamento de eventos. Isolar o loop de eventos possibilita gerenciar eventos separadamente de outros aspectos do jogo, como atualizar a tela.

Vejamos a classe AlienInvasion com o novo método _check_events(), que afeta somente o código em run_game():

alien_invasion.py

```
    def run_game(self):
        """Inicia o loop principal do jogo"""
        while True:
1           self._check_events()

            # Redesenha a tela durante cada passagem pelo loop
            -- trecho de código omitido --

2   def _check_events(self):
        """Responde as teclas pressionadas e a eventos de mouse"""
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
```

Criamos um novo método _check_events() 2 e movemos as linhas que verificam se o jogador clicou para fechar a janela neste novo método.

Para chamar um método de dentro de uma classe, use a notação de ponto com a variável self e o nome do método 1. Chamamos o método de dentro do loop while em run_game().

Método _update_screen()

A fim de simplificar ainda mais o run_game(), moveremos o código que atualiza a tela para um método separado chamado _update_screen():

alien_invasion.py

```
    def run_game(self):
        """Inicia o loop principal do jogo"""
        while True:
            self._check_events()
            self._update_screen()
            self.clock.tick(60)
```

```
    def _check_events(self):
        -- trecho de código omitido --

    def _update_screen(self):
        """Atualiza as imagens na tela e mude para a nova tela"""
        self.screen.fill(self.settings.bg_color)
        self.ship.blitme()

        pygame.display.flip()
```

Movemos o código que desenha o background e a espaçonave e vira a tela para `_update_screen()`. Agora o corpo do loop principal em `run_game()` é mais simples. É fácil ver que estamos procurando novos eventos, atualizando a tela e marcando o relógio em cada passagem pelo loop.

Caso já tenha desenvolvido diversos jogos, você provavelmente começará dividindo seu código em métodos como esses. Mas, caso nunca tenha se deparado com um projeto como esse, é bem provável que você não saiba como estruturar seu código a princípio. A abordagem usada lhe fornece uma noção de um processo de desenvolvimento na prática: você começa escrevendo o código da forma mais simples possível e, em seguida, refatora à medida que seu projeto fica mais complexo.

Agora que reestruturamos o código para facilitar a adição de mais códigos, podemos trabalhar nos aspectos dinâmicos do jogo!

FAÇA VOCÊ MESMO
12.1 Céu azul: Crie uma janela do Pygame com um background azul.
12.2 Personagem do jogo: Encontre uma imagem de bitmap de um personagem do jogo que você goste ou converta uma imagem em um bitmap. Crie uma classe que desenhe o personagem no centro da tela, depois combine a cor de background da imagem com a cor de backgroun da tela ou vice-versa.

Pilotando a espaçonave

Agora forneceremos ao jogador a capacidade de deslocar a espaçonave para a direita e para a esquerda. Vamos escrever um

código que responde quando o jogador pressiona a tecla de seta para a direita ou para a esquerda. Focaremos o primeiro movimento para a direita e, depois, empregaremos os mesmos princípios para controlar o movimento para a esquerda. À medida que escreve mais código, você aprenderá a controlar o movimento das imagens na tela e a responder à entrada do usuário.

Respondendo às teclas pressionadas

Sempre que o jogador pressiona uma tecla, essa tecla pressionada é registrada no Pygame como um evento. Cada evento é selecionado pelo método pygame.event.get(). É necessário especificar em nosso método _check_events() quais tipos de eventos queremos que o jogo verifique. Cada tecla pressionada é registrada como um evento KEYDOWN.

Quando o Pygame detecta um evento KEYDOWN, precisamos verificar se a tecla pressionada é aquela que desencadeia determinada ação. Por exemplo, se o jogador pressionar a tecla de seta para a direita, é necessário aumentar o valor rect.x da espaçonave a fim de deslocar a espaçonave para a direita:

alien_invasion.py

```
    def _check_events(self):
        """Responde as teclas pressionadas e a eventos de mouse"""
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
1           elif event.type == pygame.KEYDOWN:
2               if event.key == pygame.K_RIGHT:
                    # Move a espaçonave para a direita
3                   self.ship.rect.x += 1
```

Dentro do _check_events() adicionamos um bloco elif ao loop de eventos para responder quando Pygame detecta um evento KEYDOWN 1. Verificamos se a tecla pressionada, event.key, é a tecla de seta para a direita 2. A tecla de seta para a direita é representada por pygame.K_RIGHT. Se a tecla de seta para a direita for pressionada,

movemos a espaçonave para a direita aumentando o valor de self.ship.rect.x por 1 3.

Agora, quando executar *alien_invasion.py*, a espaçonave deve se mover um pixel para a direita sempre que você pressionar a tecla de seta para a direita. Já é alguma coisa, mas não a forma ideal de controlar a espaçonave. Vamos incrementar esse controle recorrendo ao movimento contínuo.

Movimento contínuo

Quando o jogador permanece com a tecla de seta para a direita pressionada, queremos que a espaçonave continue se movendo para a direita até o jogador soltar a tecla. Faremos com que o jogo detecte um evento pygame.KEYUP para sabermos quando a tecla de seta para a direita não estiver pressionada. Assim, usaremos os eventos KEYDOWN e KEYUP com uma flag chamada moving_right para implementar o movimento contínuo.

Quando a flag moving_right for False, a espaçonave ficará imóvel. Quando o jogador pressionar a tecla de seta para a direita, definiremos a flag como True e, quando o jogador soltar a tecla, definiremos mais uma vez a flag como False.

Como a classe Ship controla todos os atributos da espaçonave, forneceremos a ela um atributo chamado moving_right e um método update() para verificar o status da flag moving_right. O método update() mudará a posição da espaçonave se a flag estiver definida como True. Chamaremos esse método uma vez em cada passagem pelo loop while a fim de atualizar a posição da espaçonave.

Vejamos as mudanças em Ship:

ship.py

```
class Ship:
    """Classe para gerenciar a espaçonave"""

    def __init__(self, ai_game):
        -- trecho de código omitido --
```

```
        # Posiciona cada espaçonave nova na parte inferior central da tela
        self.rect.midbottom = self.screen_rect.midbottom

        # Flag de movimento; começa com uma espaçonave que não está se movendo
1       self.moving_right = False

2   def update(self):
        """Atualiza a posição da espaçonave com base na flag de movimento"""
        if self.moving_right:
            self.rect.x += 1

    def blitme(self):
        -- trecho de código omitido --
```

Adicionamos um atributo self.moving_right ao método __init__() e, a princípio, definimos como False 1. Em seguida, adicionamos update(), que desloca a espaçonave para a direita se a flag for True 2. O método update() será chamado fora da classe, logo não é considerado um método auxiliar.

Agora precisamos modificar o _check_events() para que o moving_right seja definido como True quando a tecla de seta para a direita for pressionada e False quando a tecla for solta:

alien_invasion.py

```
    def _check_events(self):
        """Responde as teclas pressionadas e a eventos de mouse"""
        for event in pygame.event.get():
            -- trecho de código omitido --
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RIGHT:
1                   self.ship.moving_right = True
2           elif event.type == pygame.KEYUP:
                if event.key == pygame.K_RIGHT:
                    self.ship.moving_right = False
```

Aqui, modificamos a forma como o jogo responde quando o jogador pressiona a tecla de seta para a direita: em vez de mudar a posição da espaçonave diretamente, apenas definimos moving_right como True 1. Em seguida, adicionamos um novo bloco elif, que responde aos eventos KEYUP 2. Quando o jogador solta a tecla de seta para a direita (K_RIGHT), definimos moving_right como False.

Depois, modificamos o loop while em run_game() para que chame o método update() da espaçonave em cada passagem pelo loop:

alien_invasion.py

```
def run_game(self):
    """Inicia o loop principal do jogo"""
    while True:
        self._check_events()
        self.ship.update()
        self._update_screen()
        self.clock.tick(60)
```

A posição da espaçonave será atualizada depois de verificarmos os eventos do teclado e antes de atualizarmos a tela. Isso possibilita que a posição da espaçonave seja atualizada em resposta à entrada do jogador e assegura que a posição atualizada seja usada ao desenhar a espaçonave na tela.

Quando você executa *alien_invasion.py* e mantém pressionada a tecla de seta para a direita, a espaçonave deve se mover continuamente para a direita até que você solte a tecla.

Movimentos à esquerda e à direita

Agora que a espaçonave pode se movimentar continuamente para a direita, adicionar o movimento à esquerda é simples. Novamente, modificaremos a classe Ship e o método _check_events(). Vejamos as mudanças relevantes do __init__() e do update() em Ship:

ship.py

```
def __init__(self, ai_game):
    -- trecho de código omitido --
    # Flags de movimento; comece com uma espaçonave que não está se movendo
    self.moving_right = False
    self.moving_left = False

def update(self):
    """Atualiza a posição do espaçonave com base nas flags de movimento"""
    if self.moving_right:
        self.rect.x += 1
```

```
        if self.moving_left:
            self.rect.x -= 1
```

Em `__init__()` adicionamos uma flag `self.moving_left`. Em `update()` utilizamos dois blocos `if` separados, em vez de um `elif`, viabilizando que o valor `rect.x` da espaçonave seja aumentado e depois reduzido quando ambas as teclas de seta são pressionadas. O resultado: a espaçonave permanece imóvel. Se usássemos `elif` para o movimento à esquerda, a tecla de seta para a direita sempre teria prioridade. Usar dois blocos `if` torna os movimentos mais precisos, já que o jogador pode pressionar momentaneamente ambas as teclas ao mudar de direção.

Temos que fazer duas adições ao`_check_events()`:

alien_invasion.py

```
    def _check_events(self):
        """Responde as teclas pressionadas e a eventos de mouse"""
        for event in pygame.event.get():
            -- trecho de código omitido --
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RIGHT:
                    self.ship.moving_right = True
                elif event.key == pygame.K_LEFT:
                    self.ship.moving_left = True

            elif event.type == pygame.KEYUP:
                if event.key == pygame.K_RIGHT:
                    self.ship.moving_right = False
                elif event.key == pygame.K_LEFT:
                    self.ship.moving_left = False
```

Se ocorrer um evento `KEYDOWN` para a tecla `K_LEFT`, definimos `moving_left` como `True`. Se ocorrer um evento `KEYUP` para a tecla `K_LEFT`, definimos `moving_left` como `False`. Aqui, podemos utilizar blocos `elif`, pois cada evento está conectado a apenas uma tecla. Se o jogador pressionar as duas teclas ao mesmo tempo, dois eventos separados serão detectados.

Agora, ao executar *alien_invasion.py*, é possível mover a espaçonave continuamente à direita e à esquerda. Caso mantenha as duas teclas

pressionadas, a espaçonave deve parar de se mover.

Em seguida, refinaremos ainda mais o movimento do espaçonave. Ajustaremos a velocidade da espaçonave e restringiremos a distância da espaçonave para que não desapareça nas laterais da tela.

Ajustando a velocidade da espaçonave

Até agora, a espaçonave descola um pixel por ciclo e por meio do loop `while`, mas podemos controlar melhor a velocidade da espaçonave adicionando um atributo `ship_speed` à classe `Settings`. Usaremos esse atributo para determinar a distância percorrida da espaçonave em cada passagem pelo loop. Vamos conferir o novo atributo em *settings.py*:

settings.py

```
class Settings:
    """Classe para armazenar as configurações do jogo Invasão Alienígena"""

    def __init__(self):
        -- trecho de código omitido --

        # Configurações da espaçonave
        self.ship_speed = 1.5
```

Definimos o valor inicial de `ship_speed` como `1.5`. Quando se desloca, a posição da espaçonave é ajustada em 1.5 pixels (em vez de 1 pixel) em cada passagem pelo loop.

Estamos usando um float para a configuração de velocidade a fim de termos um controle mais preciso da velocidade do espaçonave quando posteriormente aumentarmos o ritmo do jogo. No entanto, atributos `rect` como `x` armazenam somente valores inteiros. Ou seja, precisamos fazer algumas modificações em `Ship`:

ship.py

```
class Ship:
    """Classe para gerenciar a espaçonave"""
```

```
    def __init__(self, ai_game):
        """Inicializa a espaçonave e defina sua posição inicial"""
        self.screen = ai_game.screen
1       self.settings = ai_game.settings
        -- trecho de código omitido --

        # Coloca cada espaçonave nova na parte inferior central da tela
        self.rect.midbottom = self.screen_rect.midbottom

        # Armazena um float para a posição horizontal exata da espaçonave
2       self.x = float(self.rect.x)

        # Flags de movimento; começa com uma espaçonave que não está se movendo
        self.moving_right = False
        self.moving_left = False

    def update(self):
        """Atualiza a posição da espaçonave com base nas flags de movimento"""
        # Atualiza o valor x da espaçoave, não o rect
        if self.moving_right:
3           self.x += self.settings.ship_speed
        if self.moving_left:
            self.x -= self.settings.ship_speed

        # Atualiza o objeto rect de self.x
4       self.rect.x = self.x

    def blitme(self):
        -- trecho de código omitido --
```

Criamos um atributo setting para Ship, de modo que possamos usá-lo em update() 1. Como estamos ajustando a posição da espaçonave por frações de pixel, é necessário atribuir a posição a uma variável que possa ter um float atribuído. É possível usar float para definir um atributo de um rect, só que o rect manterá apenas a parte inteira desse valor. A fim de rastrear a posição da espaçonave com precisão, definimos um novo self.x 2. Usamos a função float() a fim de converter o valor de self.rect.x em um float e atribuir esse valor a self.x.

Agora, quando mudamos a posição da espaçonave em update(), o valor de self.x é ajustado pela quantidade armazenada em settings.ship_speed 3. Após self.x ser atualizado, utilizamos o novo valor

para atualizar self.rect.x, que controla a posição da espaçonave 4. Apenas a parte inteira de self.x será atribuída a self.rect.x, o que é bom para exibir a espaçonave.

Em seguida, podemos mudar o valor de ship_speed, e qualquer valor maior que 1 fará com que a espaçonave se mova mais rápido. Isso ajudará com que a espaçonave responda rápido o suficiente para abater alienígenas, e nos permitirá mudar o ritmo do jogo à medida que o jogador progride na jogabilidade.

Restringindo o alcance da espaçonave

Até agora, se pressionarmos a tecla de seta por tempo suficiente, a espaçonave desaparece das bordas da tela. Vamos corrigir isso para que a espaçonave pare de se mover quando alcança a borda da tela. Para isso, modificamos o método update() em Ship:

ship.py

```
    def update(self):
        """Atualiza a posição da espaçonave com base nas flags de movimento"""
        # Atualiza o valor x da espaçonave, não o rect
1       if self.moving_right and self.rect.right < self.screen_rect.right:
            self.x += self.settings.ship_speed
2       if self.moving_left and self.rect.left > 0:
            self.x -= self.settings.ship_speed

        # Atualiza o objeto react a partir de self.x
        self.rect.x = self.x
```

Esse código verifica a posição da espaçonave antes de alterar o valor de self.x. O código self.rect.right retorna a coordenada x da borda direita do rect da espaçonave. Se este valor for menor que o valor retornado por self.screen_rect.right, a espaçonave não alcançou a borda direita da tela 1. O mesmo vale para a borda esquerda: se o valor do lado esquerdo do rect for maior que 0, a espaçonave não alcançou a borda esquerda da tela 2. Isso garante que a espaçonave fique dentro desses limites antes de ajustarmos o valor de self.x.

Agora, ao executar *alien_invasion.py*, a espaçonave deve parar de se mover em qualquer borda da tela. Isso é bem legal; tudo o que

fizemos foi adicionar um teste condicional em uma instrução if, mas parece que a espaçonave colide contra uma parede ou contra campo de força em qualquer borda da tela!

Refatorando o _check_events()

O método _check_events() ficará cada vez maior conforme desenvolvemos o jogo. Por isso, vamos dividi-lo em dois métodos separados: um que lida com eventos KEYDOWN e outro que lida com eventos KEYUP:

alien_invasion.py

```
    def _check_events(self):
        """Responde as teclas pressionadas e a eventos de mouse"""
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
            elif event.type == pygame.KEYDOWN:
                self._check_keydown_events(event)
            elif event.type == pygame.KEYUP:
                self._check_keyup_events(event)

    def _check_keydown_events(self, event):
        """Responde a teclas pressionadas"""
        if event.key == pygame.K_RIGHT:
            self.ship.moving_right = True
        elif event.key == pygame.K_LEFT:
            self.ship.moving_left = True

    def _check_keyup_events(self, event):
        """Responde a teclas soltas"""
        if event.key == pygame.K_RIGHT:
            self.ship.moving_right = False
        elif event.key == pygame.K_LEFT:
            self.ship.moving_left = False
```

Criamos dois novos métodos auxiliares: _check_keydown_events() e _check_keyup_events(). Cada um precisa de um parâmetro self e um parâmetro event. Os corpos desses dois métodos são copiados de _check_events() e substituímos o código anterior por chamadas de novos métodos. O método _check_events() é mais simples com essa

estrutura de código mais limpa, o que facilitará o desenvolvimento de respostas adicionais à entrada do jogador.

Pressionando Q para encerrar o jogo

Agora que estamos respondendo as teclas pressionadas com eficiência, podemos adicionar outra forma de sair do jogo. É chato demais clicar no X na parte superior da janela do jogo para encerrá-lo sempre que testamos uma nova funcionalidade, então adicionaremos um atalho de teclado para encerrar o jogo quando o jogador pressionar Q:

alien_invasion.py

```
def _check_keydown_events(self, event):
    -- trecho de código omitido --
    elif event.key == pygame.K_LEFT:
        self.ship.moving_left = True
    elif event.key == pygame.K_q:
        sys.exit()
```

Em `_check_keydown_events()` adicionamos um novo bloco que encerra o jogo quando o jogador pressiona Q. Agora, ao testar, podemos pressionar Q para encerrar o jogo em vez de usar o cursor do mouse para fechar a janela.

Executando o jogo no modo de tela cheia

O Pygame tem um modo de tela cheia que talvez você goste mais do que uma janela normal. Alguns jogos ficam melhores no modo de tela cheia e, em alguns sistemas, o jogo pode ter um desempenho geral melhor.

Para executar o jogo no modo de tela cheia, faça as seguintes mudanças em `__init__()`:

alien_invasion.py

```
def __init__(self):
    """Inicializa o jogo e cria recursos de jogo"""
    pygame.init()
    self.settings = Settings()
```

```
1       self.screen = pygame.display.set_mode((0, 0), pygame.FULLSCREEN)
2       self.settings.screen_width = self.screen.get_rect().width
        self.settings.screen_height = self.screen.get_rect().height
        pygame.display.set_caption("Alien Invasion")
```

Ao criar a superfície da tela, passamos o tamanho `(0, 0)` e o parâmetro `pygame.FULLSCREEN` `1`. Isso informa ao Pygame para detectar um tamanho de janela que preencherá a tela. Como não sabemos de antemão a largura e a altura da tela, atualizamos essas configurações após a criação da tela `2`. Usamos os atributos `width` e `height` da tela do `rect` para atualizar o objeto `settings`.

Caso goste de como o jogo fica ou se comporta no modo de tela cheia, mantenha essas configurações. Se gostou mais do jogo em sua janela normal, você pode retomar a abordagem original, em que definimos um tamanho de tela específico para o jogo.

NOTA *Tenha certeza que pode encerrar o jogo pressionando Q antes de executá-lo em modo de tela cheia; o Pygame não oferece uma maneira padrão de encerrar um jogo no modo de tela cheia.*

Breve recapitulação

Na próxima seção, adicionaremos a capacidade de atirar, em um novo arquivo chamado *bullet.py* e faremos algumas mudanças em alguns dos arquivos que já estamos usando. No momento, temos três arquivos contendo diversas classes e métodos. Para esclarecer como o projeto está organizado, revisaremos cada um desses arquivos antes de adicionarmos outras funcionalidades.

alien_invasion.py

O arquivo principal, *alien_invasion.py*, contém a classe `AlienInvasion`. Essa classe cria inúmeros atributos importantes usados em todo o jogo: as configurações são atribuídas à `settings`, a superfície de exibição principal é atribuída à `screen` e uma instância da espaçonave

também é criada nesse arquivo. O loop principal do jogo, um loop while, também é armazenado neste módulo. O loop while chama _check_events(), ship.update() e _update_screen(). Marca também o relógio em cada passagem pelo loop.

O método _check_events() detecta eventos relevantes, como teclas pressionadas e soltas, e processa cada um desses tipos de eventos com os métodos _check_keydown_events() e _check_keyup_events(). Por ora, esses métodos gerenciam o movimento da espaçonave. A classe AlienInvasion também contém _update_screen(), que redesenha a tela em cada passagem pelo loop principal.

O arquivo *alien_invasion.py* é o único arquivo que você precisa executar quando quiser jogar o jogo *Invasão Alienígena*. Os outros arquivos, *settings.py* e *ship.py*, contêm código importado para este arquivo.

settings.py

O arquivo *settings.py* contém a classe Settings. Essa classe tem apenas um método __init__(), que inicializa os atributos controlando a aparência do jogo e a velocidade da espaçonave.

ship.py

O arquivo *ship.py* contém a classe Ship. A classe Ship tem um método __init__(), um método update() para gerenciar a posição da espaçonave e um método blitme() para desenhar a espaçonave na tela. A imagem da espaçonave é armazenada em *ship.bmp*, que fica na pasta de images.

FAÇA VOCÊ MESMO

12.3 Documentação do Pygame: Já estamos familiarizados com o jogo para que você queira espiar a documentação do Pygame. É possível acessá-la em https://pygame.org, e a página inicial da documentação em https://pygame.org/docs. Por enquanto, uma rápida olhada é o bastante. Você não precisará da documentação para concluir esse projeto, mas ajudará caso queira modificar o Jogo *Invasão Alienígena* ou criar o próprio jogo depois.

12.4 Espaçonave: Desenvolva um jogo que comece com uma espaçonave no centro da

tela. Permita que o jogador mova a espaçonave para cima, para baixo, à esquerda ou à direita usando as quatro teclas de seta. Garanta que a espaçonave nunca se mova além das bordas da tela.

12.5 Teclas: Crie um arquivo Pygame que tenha uma tela vazia. No loop de eventos, exiba o atributo event.key sempre que um evento pygame.KEYDOWN for detectado. Execute o programa e pressione diversas teclas para conferir como o Pygame responde.

Disparando balas

Agora, vamos adicionar a capacidade de atirar. Escrevermos um código que abre fogo, representado por um pequeno retângulo, quando o jogador pressionar a barra de espaço. Desse modo, os projéteis se deslocaram verticalmente na tela até desaparecerem na parte superior da tela.

Adicionando os projéteis

No final do método `__init__()`, atualizaremos *settings.py* para incluir os valores necessários para nova classe `Bullet`:

settings.py

```
def __init__(self):
    -- trecho de código omitido --
    # Configurações do projétil
    self.bullet_speed = 2.0
    self.bullet_width = 3
    self.bullet_height = 15
    self.bullet_color = (60, 60, 60)
```

Essas configurações criam projéteis cinza escuro com largura de 3 pixels e altura de 15 pixels. Os projéteis se deslocarão um pouco mais rápido do que as espaçonaves.

Criando a classe Bullet

Agora, crie um arquivo *bullet.py* para armazenar nossa classe `Bullet`. Vejamos a primeira parte do *bullet.py*:

bullet.py

```
import pygame
from pygame.sprite import Sprite

class Bullet(Sprite):
    """Classe para gerenciar os projéteis disparados da espaçonave"""

    def __init__(self, ai_game):
        """Cria um objeto bullet na posição atual da espaçonave"""
        super().__init__()
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        self.color = self.settings.bullet_color

        # Cria um bullet rect em (0, 0) e, em seguida, define a posição correta
1       self.rect = pygame.Rect(0, 0, self.settings.bullet_width,
            self.settings.bullet_height)
2       self.rect.midtop = ai_game.ship.rect.midtop

        # Armazena a posição do projétil como um float
3       self.y = float(self.rect.y)
```

A classe `Bullet` é herdada de `Sprite`, importado do módulo `pygame.sprite`. Ao usar sprites, é possível agrupar elementos relacionados em seu jogo e iterar em todos os elementos agrupados de uma só vez. Para criar uma instância de projétil, o `__init__()` precisa da instância atual de `AlienInvasion` e chamamos `super()` para herdar apropriadamente de `Sprite`. Definimos também atributos para os objetos de tela e de configurações e para a cor do projétil.

Em seguida, criamos o atributo `rect` do projétil `1`. O projétil não é baseado em uma imagem, logo temos que construir um `rect` do zero usando a classe `pygame.Rect()`. Essa classe exige as coordenadas x e y do canto superior esquerdo do `rect`, e a largura e altura do `rect`. Inicializamos o `rect` em (0, 0), mas vamos movê-lo para o local correto na próxima linha, pois a posição do projétil depende da posição da espaçonave. Obtemos a largura e a altura do projétil a partir dos valores armazenados em `self.settings`.

Definimos o atributo `midtop` da projétil para corresponder ao atributo `midtop` da espaçonave `2`. Isso fará com que o projétil surja da frente da espaçonave, parecendo que foi disparado da espaçonave.

Usamos um float para a coordenada y do projétil, de modo que possamos fazer ajustes finos na velocidade do projétil 3.

Vejamos a segunda parte de *bullet.py*, update() e draw_bullet():

bullet.py

```
    def update(self):
        """Desloca o projétil verticalmente pela tela
        # Atualiza a posição exata da projétil
1       self.y -= self.settings.bullet_speed
        # Atualiza a posição do react
2       self.rect.y = self.y

    def draw_bullet(self):
        """Desenha o projétil na tela"""
3       pygame.draw.rect(self.screen, self.color, self.rect)
```

O método update() gerencia a posição do projétil. Quando disparado, um projétil se move verticalmente pela tela, o que corresponde a um valor decrescente da coordenada y. Para atualizar a posição, subtraímos a quantidade armazenada em settings.bullet_speed de self.y 1. Depois, usamos o valor de self.y para definir o valor de self.rect.y 2.

A configuração bullet_speed nos possibilita aumentar a velocidade dos projéteis à medida que o jogo avança ou conforme necessário para refinar o comportamento do jogo. Uma vez que um projétil é disparado, nunca mudamos o valor de sua coordenada x, logo o projétil se deslocará verticalmente em linha reta, mesmo que a espaçonave se mova.

Quando queremos desenhar um projétil, chamamos draw_bullet(). A função draw.rect() preenche a parte da tela definida pelo rect do projétil com a cor armazenada em self.color 3.

Armazenando projéteis em um grupo

Agora que temos uma classe Bullet e as configurações necessárias definidas, podemos escrever um código para disparar um projeto sempre que o jogador pressionar a barra de espaço. Criaremos um grupo em AlienInvasion para armazenar todos os projéteis ativos, assim

podemos gerenciar as projéteis já disparados. Esse grupo será uma instância da classe pygame.sprite.Group, que se comporta como uma lista com algumas funcionalidades extras, úteis no desenvolvimento de jogos. Usaremos esse grupo para desenhar projéteis na tela em cada passagem pelo loop principal e atualizar a posição de cada projétil.

Primeiro, importaremos a nova classe Bullet:

alien_invasion.py

```
-- trecho de código omitido --
from ship import Ship
from bullet import Bullet
```

Em seguida, criaremos o grupo que armazena os projéteis em __init__():

alien_invasion.py

```
def __init__(self):
    -- trecho de código omitido --
    self.ship = Ship(self)
    self.bullets = pygame.sprite.Group()
```

Depois, precisamos atualizar a posição dos projéteis em cada passagem pelo loop while:

alien_invasion.py

```
def run_game(self):
    """Inicia o loop principal do jogo"""
    while True:
        self._check_events()
        self.ship.update()
        self.bullets.update()
        self._update_screen()
        self.clock.tick(60)
```

Ao chamarmos o update() em um grupo, o grupo automaticamente chama o update() para cada sprite no grupo. A linha self.bullets.update() chama o bullet.update() para cada projétil que inserimos no grupo bullets.

Disparando projéteis

Em AlienInvasion é necessário mudar _check_keydown_events() para disparar um projétil quando o jogador pressionar a barra de espaço.

Não é necessário alterar _check_keyup_events(), pois nada acontece quando a barra de espaço não é mais pressionada. É necessário também modificar _update_screen() a fim de garantir que cada projétil seja desenhado na tela antes de chamarmos flip().

É necessário nos esforçamos um pouco mais para fazer um projétil disparar. Logo, escreveremos um novo método, _fire_bullet().

alien_invasion.py

```
    def _check_keydown_events(self, event):
        -- trecho de código omitido --
        elif event.key == pygame.K_q:
            sys.exit()
1       elif event.key == pygame.K_SPACE:
            self._fire_bullet()

    def _check_keyup_events(self, event):
        -- trecho de código omitido --

    def _fire_bullet(self):
        """Cria um novo projétil e o adiciona ao grupo projéteis"""
2       new_bullet = Bullet(self)
3       self.bullets.add(new_bullet)

    def _update_screen(self):
        """Atualiza as imagens na tela e muda para a nova tela"""
        self.screen.fill(self.settings.bg_color)
4       for bullet in self.bullets.sprites():
            bullet.draw_bullet()
        self.ship.blitme()

        pygame.display.flip()
-- trecho de código omitido --
```

Chamamos _fire_bullet() quando a barra de espaço é pressionada 1. Em _fire_bullet() criamos uma instância de Bullet e a chamamos de new_bullet 2. Depois, a adicionamos ao grupo bullets usando o método add() 3. O método add() é semelhante ao append(), mas é escrito

especificamente para grupos Pygame.

O método bullets.sprites() retorna uma lista de todos os sprites no bullets. Para desenhar todos os projéteis disparados na tela, percorremos os sprites em bullets por meio de um loop e chamamos draw_bullet() em cada um deles 4. Inserimos esse loop antes da linha que desenha a espaçonave, para que os projéteis não comecem na frente da espaçonave.

Agora, quando executar o *alien_invasion.py*, você poderá deslocar a espaçonave para a direita e para a esquerda e disparar quantos projéteis quiser. As projéteis passam na tela e desaparecem quando alcançam a parte superior, conforme mostrado na Figura 12.3. É possível alterar o tamanho, a cor e a velocidade dos projéteis em *settings.py*.

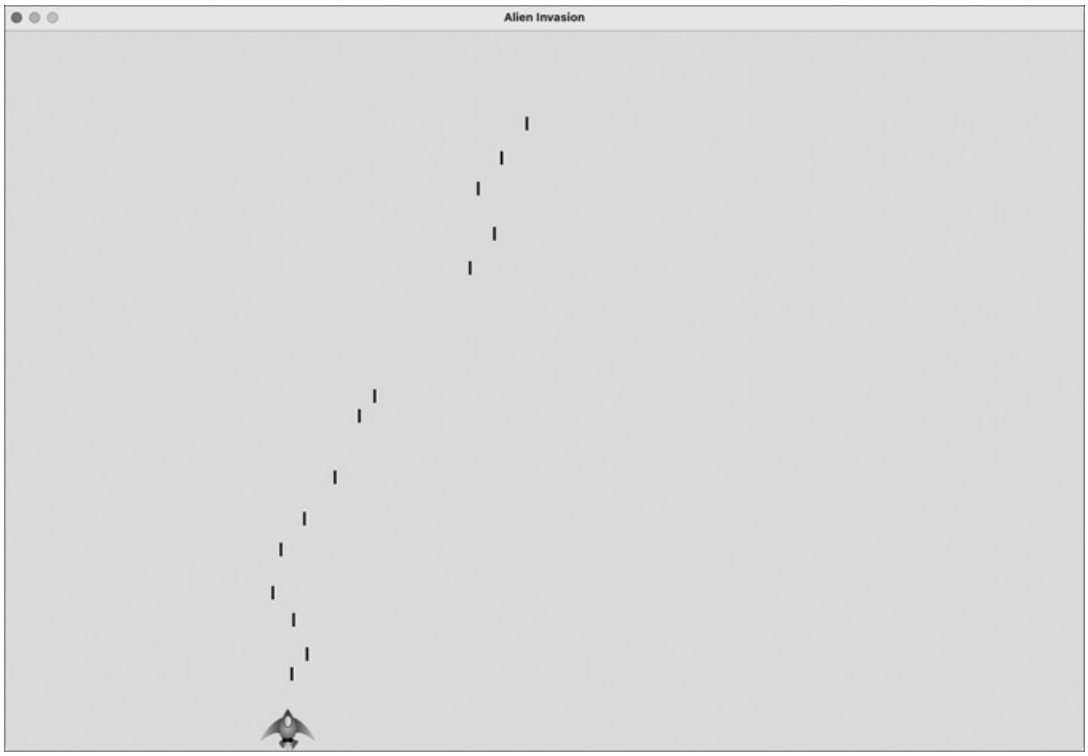


Figura ١٢.3: Espaçonave após disparar uma série de projéteis.

Deletando projéteis antigos

Até agora, os projéteis desaparecem quando alcançam a parte

superior da tela, mas só porque o Pygame não consegue desenhá-los acima da parte superior da tela. Na realidade, os projéteis ainda existem, seus valores da coordenada y ficam cada vez mais negativos. Isso é um problema, visto que continuam consumindo memória e processamento computacional.

Precisamos descartar esses projéteis antigos, ou o jogo ficará mais lento executando tanta coisa desnecessária. Para tal, precisamos detectar quando o valor bottom do rect de um projétil tem um valor 0, sinalizando que o projétil passou pela parte superior da tela:

alien_invasion.py

```
    def run_game(self):
        """Inicia o loop principal do jogo"""
        while True:
            self._check_events()
            self.ship.update()
            self.bullets.update()

            # Descarta os projéteis que desaparecem
1           for bullet in self.bullets.copy():
2               if bullet.rect.bottom <= 0:
3                   self.bullets.remove(bullet)
4           print(len(self.bullets))

            self._update_screen()
            self.clock.tick(60)
```

Ao utilizarmos um loop for com uma lista (ou um grupo no Pygame), o Python espera que a lista permaneça com o mesmo tamanho, contanto que o loop esteja em execução. Ou seja, não podemos remover itens de uma lista ou grupo dentro de um loop for. Assim, temos que fazer uma cópia do grupo no loop. Usamos o método copy() para configurar o loop for 1, assim podemos modificar o grupo original bullets dentro do loop. Verificamos se cada projétil desapareceu da parte superior da tela 2. Se sim, os removemos do bullets 3. Inserimos um print() para exibir a quantidade de projéteis atuais no jogo e verificamos se estão sendo deletados quando alcançam a parte superior da tela 4.

Se esse código executar sem problemas, é possível ver na saída no terminal, conforme os projéteis são disparados, que o número de projéteis reduz a zero após desaparecerem na parte superior da tela. Após verificar se os projéteis estão sendo deletados corretamente, remova o `print()`. Se optar por não removê-lo, o jogo ficará bem lento, pois leva mais tempo para gravar a saída no terminal do que para desenhar gráficos na janela do jogo.

Restringindo o número de projéteis

Muitos jogos de atirar restringem o número de projéteis que um jogador pode ter na tela ao mesmo tempo; isso incentiva os jogadores a atirarem com precisão. Faremos o mesmo com o Jogo *Invasão Alienígena*.

Primeiro, armazene o número de projéteis permitidos em *settings.py*:

settings.py

```
# Configurações dos projéteis
-- trecho de código omitido --
self.bullet_color = (60, 60, 60)
self.bullets_allowed = 3
```

Isso limita o jogador a três projéteis por vez. Usaremos essa configuração em `AlienInvasion` para verificar quantos projéteis existem antes de criar um novo projétil em `_fire_bullet()`:

alien_invasion.py

```
def _fire_bullet(self):
    """Cria um novo projétil e o adiciona ao grupo de projéteis"""
    if len(self.bullets) < self.settings.bullets_allowed:
        new_bullet = Bullet(self)
        self.bullets.add(new_bullet)
```

Quando o jogador pressiona a barra de espaço, verificamos o comprimento de `bullets`. Se `len(self.bullets)` for menor que três, criamos um novo projétil. Mas se tivermos três projeto ativos, nada acontece quando a barra de espaço é pressionada. Agora, quando executar o jogo, você só poderá disparar projéteis em grupos de três.

Criando o método _update_bullets()

Queremos manter a classe AlienInvasion razoavelmente bem organizada. Ou seja, agora que escrevemos e verificamos o código de gerenciamento de projéteis, podemos movê-lo para um método separado. Vamos criar um novo método chamado _update_bullets() e adicioná-lo logo antes de _update_bullets():

alien_invasion.py

```
def _update_bullets(self):
    """Atualiza a posição dos projéteis e descarta os projéteis antigos"""
    # Atualiza as posições dos projéteis
    self.bullets.update()

    # Descarta os projéteis que desapareceram
    for bullet in self.bullets.copy():
        if bullet.rect.bottom <= 0:
            self.bullets.remove(bullet)
```

O código para _update_bullets() é recortado e colado de run_game(); tudo o que fizemos aqui foi deixar as coisas claras nos comentários.

O loop while em run_game() parece simples mais uma vez:

alien_invasion.py

```
while True:
    self._check_events()
    self.ship.update()
    self._update_bullets()
    self._update_screen()
    self.clock.tick(60)
```

Agora nosso loop principal contém apenas o mínimo de código, a fim de que possamos ler rapidamente os nomes dos métodos e entender o que está acontecendo no jogo. O loop principal verifica a entrada do jogador e, em seguida, atualiza a posição da espaçonave e quaisquer projéteis disparados. Em seguida, usamos as posições atualizadas para desenhar uma nova tela e marcar o relógio no final de cada passagem pelo loop.

Execute *alien_invasion.py* mais uma vez e confita se ainda pode disparar projéteis sem erros.

FAÇA VOCÊ MESMO

12.6 Disparos laterais: Crie um jogo que posicione um espaçonave no lado esquerdo da tela e possibilite que o jogador a manobre para cima e para baixo. Faça a espaçonave disparar um projétil que atravesse a tela quando o jogador pressionar a barra de espaço. Garanta que os projéteis sejam deletados assim que desaparecerem da tela.

Recapitulando

Neste capítulo, aprendemos a fazer um planejamento para um jogo e a estrutura básica de um jogo desenvolvido com o Pygame. Aprendemos a definir a cor de um background e armazenar as configurações em uma classe separada, em que é possível ajustá-las com mais facilidade. Vimos como desenhar uma imagem na tela e fornecer ao jogador controle sobre o movimento dos elementos do jogo. Desenvolvemos elementos que se movem por conta própria, como projéteis voando pela tela, e deletamos objetos que não são mais necessários. Aprendemos a refatorar código reiteradamente em um projeto para facilitar o desenvolvimento contínuo.

No Capítulo 13, adicionaremos alienígenas ao jogo Invasão Alienígena. No final do capítulo, você poderá abrir fogo contra os alienígenas e, quem sabe, antes de atingirem sua espaçonave!

CAPÍTULO 13
Alienígenas!

Neste capítulo, adicionaremos alienígenas ao jogo *Invasão Alienígena*. Vamos inserir um alienígena perto da parte superior da tela e, em seguida, vamos gerar uma frota inteira deles. Faremos a frota se deslocar lateralmente e para baixo, e descartaremos qualquer alienígena alvejado por um projétil. Por último, limitaremos o número de espaçonaves de um jogador e encerraremos o jogo quando o jogador ficar sem espaçonaves.

Ao longo deste capítulo, aprenderemos mais sobre o Pygame e sobre como gerenciar um projeto grande. Aprenderemos também a detectar colisões entre os objetos do jogo, como projéteis e alienígenas. Detectar colisões ajuda a estabelecer interações entre os elementos do jogo. Por exemplo, é possível confinar um personagem dentro das paredes de um labirinto ou passar uma bola entre dois personagens. Continuaremos a trabalhar com um planejamento, que revisitaremos ocasionalmente para manter o foco de nossas sessões de escrita de código.

Antes de começarmos a desenvolver um código novo para adicionar uma frota de alienígenas à tela, examinaremos o projeto e atualizaremos nosso planejamento.

Revisando o projeto

Ao começar uma fase nova de desenvolvimento em um projeto grande, é sempre uma boa ideia rever o planejamento e explicitar o que quer realizar com o código que você está prestes a escrever. Neste capítulo, faremos o seguinte:

- Vamos adicionar um único alienígena ao canto superior esquerdo da tela, com distanciamento apropriado ao redor.
- Vamos preencher a parte superior da tela com o máximo de alienígenas que conseguirmos inserir horizontalmente. Depois, criaremos fileiras adicionais de alienígenas até que tenhamos uma frota completa.
- Faremos a frota se deslocar lateralmente e para baixo até que toda a frota seja abatida, um alienígena abata uma espaçonave ou um alienígena se choque contra o solo. Se toda a frota for abatida, criaremos uma frota nova. Se um alienígena abater uma espaçonave ou se chocar contra o solo, destruiremos a espaçonave e criaremos uma frota nova.
- Vamos restringir o número de espaçonaves que o jogador pode usar e encerrar o jogo quando o jogador tiver utilizado o número atribuído de espaçonaves.

Vamos refinar esse planejamento à medida que implementarmos funcionalidades, mas isso é específico o suficiente para começarmos a escrever código.

Em um projeto, quando começar a trabalhar em uma nova série de funcionalidades, revise também seu código existente. Em geral, como cada fase nova contribui sucessivamente com a complexidade de um projeto, é melhor limpar qualquer código desorganizado ou ineficiente. Como estamos refatorando à medida que avançamos, não precisamos refatorar nenhum código neste momento.

Criando o primeiro alienígena

Posicionar um alienígena na tela é como posicionar uma espaçonave. O comportamento de cada alienígena é controlado por uma classe chamada `Alien`, que estruturaremos como fizemos com a classe `Ship`. Para simplificar as coisas, continuaremos usando imagens bitmap. É possível encontrar uma imagem para seu alienígena ou usar aquela mostrada na Figura 13.1, disponível nos recursos do livro em https://ehmatthes.github.io/pcc_3e.

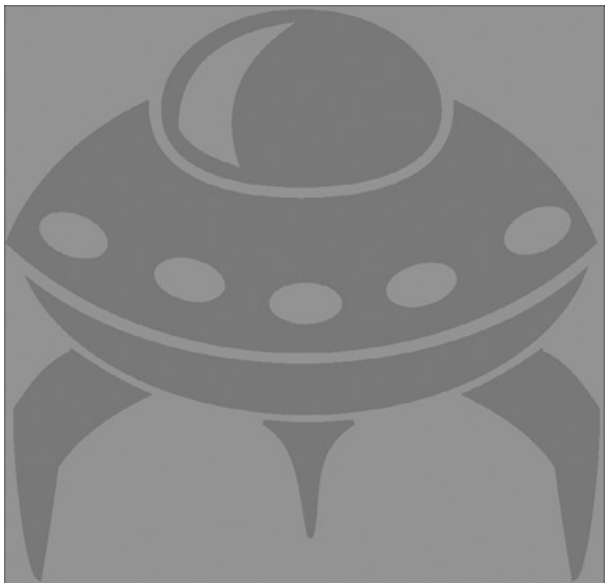


Figura 13.1: O alienígena que usaremos a fim de criar a frota.

Essa imagem tem um background cinza, que corresponde à cor do background da tela. Não se esqueça de salvar o arquivo de imagem que escolher na pasta *images*.

Criando a classe Alien

Agora, escreveremos a classe Alien e vamos salvá-la como *alien.py*:

alien.py

```
import pygame
from pygame.sprite import Sprite

class Alien(Sprite):
    """Classe para representar um único alienígena na frota"""

    def __init__(self, ai_game):
        """Inicializa o alienígena e define sua posição inicial"""
        super().__init__()
        self.screen = ai_game.screen

        #.Carrega a imagem do alienígena e define seu atributo rect
        self.image = pygame.image.load('images/alien.bmp')
        self.rect = self.image.get_rect()

        # Inicia cada alienígena novo perto do canto superior esquerdo da tela
1       self.rect.x = self.rect.width
```

```
        self.rect.y = self.rect.height

        # Armazena a posição horizontal exata do alienígena
2       self.x = float(self.rect.x)
```

Boa parte dessa classe é parecida com a classe Ship, exceto pela posição do alienígena na tela. De início, posicionamos cada alienígena perto do canto superior esquerdo da tela; adicionamos um espaço à esquerda, igual à largura do alienígena e um espaço acima dele, igual à sua altura 1, assim podemos facilmente vê-lo. Como estamos basicamente preocupados com a velocidade horizontal dos alienígenas, rastrearemos a posição horizontal de cada alienígena de modo preciso 2.

A classe Alien não precisa de um método para desenhá-la na tela; em vez disso, usaremos um método do Pygame que desenha automaticamente todos os elementos de um grupo em uma tela.

Criando uma instância do alienígena

Queremos criar uma instância de Alien para que possamos ver o primeiro alienígena na tela. Como faz parte do nosso trabalho de configuração, adicionaremos o código para essa instância no final do método __init__() em AlienInvasion. Mais tarde, criaremos uma frota inteira de alienígenas, o que dará um pouco de trabalho, por isso, criaremos um novo método auxiliar chamado _create_fleet().

A ordem dos métodos em uma classe não importa, desde que haja alguma consistência em como são posicionados. Vou inserir _create_fleet() logo antes do método _update_screen(), mas em qualquer lugar de AlienInvasion funcionará. Primeiro, importaremos a classe Alien.

Vejamos as instruções atualizadas de import para *alien_invasion.py*:

alien_invasion.py

```
-- trecho de código omitido --
from bullet import Bullet
from alien import Alien
```

O método __init__() atualizado:

alien_invasion.py

```
def __init__(self):
    -- trecho de código omitido --
    self.ship = Ship(self)
    self.bullets = pygame.sprite.Group()
    self.aliens = pygame.sprite.Group()

    self._create_fleet()
```

Criamos um grupo para armazenar a frota de alienígenas, e chamamos `_create_fleet()`, que estamos prestes a escrever.

Veja o método `new_create_fleet()`:

alien_invasion.py

```
def _create_fleet(self):
    """Cria a frota de alienígenas"""
    # Cria um alienígena
    alien = Alien(self)
    self.aliens.add(alien)
```

Nesse método, estamos criando uma instância de `Alien` e depois a adicionando ao grupo que armazenará a frota. O alienígena será posicionado na área default superior esquerda da tela.

Para fazer com que o alienígena apareça, precisamos chamar o método `draw()` do grupo em `_update_screen()`:

alien_invasion.py

```
def _update_screen(self):
    -- trecho de código omitido --
    self.ship.blitme()
    self.aliens.draw(self.screen)

    pygame.display.flip()
```

Ao chamar `draw()` em um grupo, o Pygame desenha cada elemento no grupo na posição definida por seu atributo `rect`. O método `draw()` exige um argumento: uma superfície na qual desenhar os elementos do grupo. A Figura 13.2 mostra o primeiro alienígena na tela.

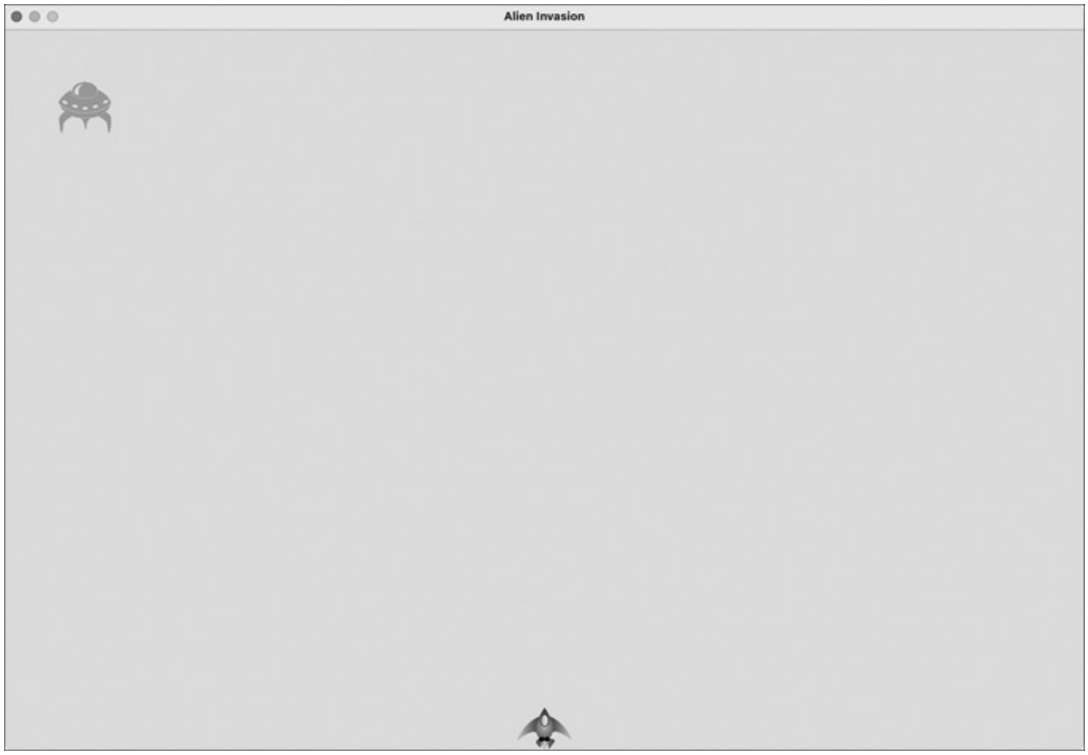


Figura 13.2: O primeiro alienígena aparece.

Agora que o primeiro alienígena aparece devidamente, vamos escrever o código para desenhar uma frota inteira.

Criando a frota alienígena

Para desenhar uma frota, é necessário identificar como preencher a parte superior da tela com alienígenas, sem superlotar a janela do jogo. Podemos fazer isso de várias formas. Recorreremos à abordagem de adicionar alienígenas na parte superior da tela, até que não haja mais espaço para um alienígena novo. Em seguida, repetiremos esse processo, contanto que tenhamos espaço vertical suficiente para adicionar uma fileira nova.

Criando uma fileira de alienígenas

Agora, estamos prontos para gerar uma fileira completa de alienígenas. Para fazer uma fileira completa, primeiro criaremos um único alienígena para que tenhamos acesso à sua largura.

Posicionaremos um alienígena no lado esquerdo da tela e depois continuaremos adicionando alienígenas até ficarmos sem espaço.

alien_invasion.py

```
    def _create_fleet(self):
        """Cria a frota de alienígenas"""
        # Cria um alienígena e continua adicionando alienígenas
        # até que não haja mais espaço
        # O distanciamento entre alienígenas é a largura de um alienígena
        alien = Alien(self)
        alien_width = alien.rect.width

1       current_x = alien_width
2       while current_x < (self.settings.screen_width - 2 * alien_width):
3           new_alien = Alien(self)
4           new_alien.x = current_x
            new_alien.rect.x = current_x
            self.aliens.add(new_alien)
5           current_x += 2 * alien_width
```

Obtemos a largura do alienígena do primeiro alienígena que criamos e definimos uma variável chamada current_x 1, que se refere à posição horizontal do próximo alienígena que pretendemos posicionar na tela. A princípio, ajustamos para uma largura alienígena, deslocando o primeiro alienígena na frota da borda esquerda da tela.

Em seguida, começamos o loop while 2; continuaremos adicionando alienígenas *enquanto* há espaço suficiente para inserir um. A fim de determinar se há espaço para posicionar outro alienígena, faremos uma comparação de current_x com algum valor máximo. Vejamos uma primeira tentativa de definir esse loop:

```
while current_x < self.settings.screen_width:
```

Talvez funcione, mas isso posicionaria o último alienígena na fileira da extremidade direita da tela. Desse modo, adicionamos uma pequena margem no lado direito da tela. Contanto que haja, pelo menos, duas larguras de alienígenas de espaço na borda direita da tela, entramos no loop e adicionamos outro alienígena à frota.

Sempre que houver espaço horizontal suficiente para continuar o loop, queremos fazer duas coisas: criar um alienígena na posição

correta e definir a posição horizontal do próximo alienígena na fileira. Criamos um alienígena e o atribuímos a new_alien 3. Em seguida, definimos a posição horizontal precisa para o valor atual de current_x 4. Posicionamos também o rect do alienígena nesse mesmo valor *x*, e adicionamos o alienígena novo ao grupo self.aliens.

Por último, incrementamos o valor de current_x 5. Adicionamos duas larguras de alienígenas à posição horizontal, para passar pelo alienígena que acabamos de adicionar e também permitir algum espaço entre os alienígenas. O Python reavaliará a condição no início do loop while e decidirá se há espaço para outro alienígena. Quando não houver mais espaço, o loop terminará e teremos uma fileira completa de alienígenas.

Ao executarmos *Invasão Alienígena* agora, devemos ver a primeira fileira de alienígenas aparecer, como na Figura 13.3.

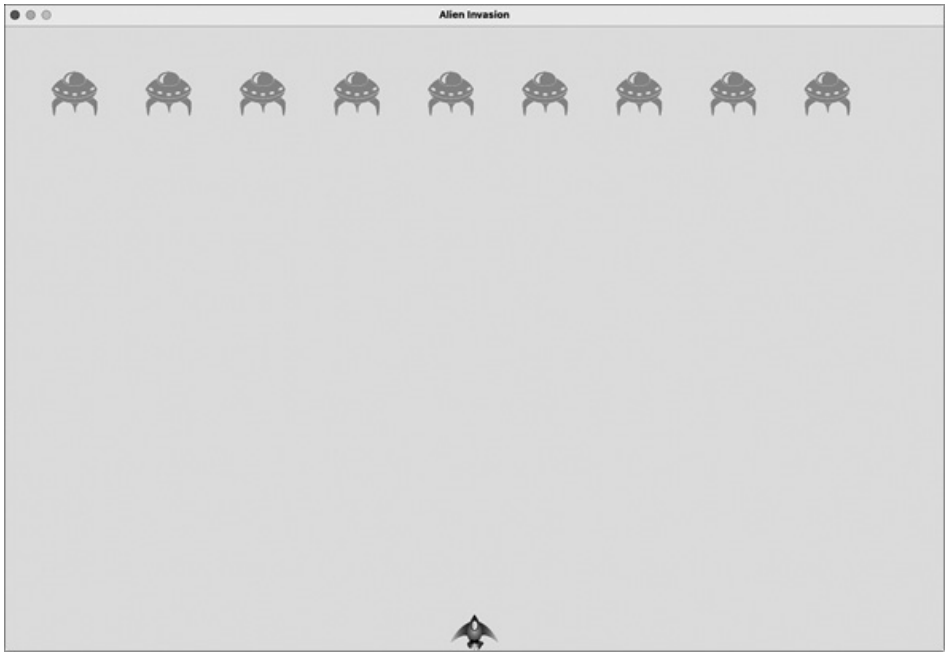


Figura 13.3: A primeira fileira de alienígenas.

NOTA *Nem sempre fica exatamente claro como criar um loop como o mostrado nesta seção. Uma das coisas boas da programação é que suas ideias iniciais de como abordar um problema como esse não precisam estar corretas. É*

perfeitamente aceitável iniciar um loop como esse com os alienígenas posicionados um tanto longe à direita e, em seguida, modificar o loop até que tenhamos uma quantidade adequada de espaço na tela.

Refatorando _create_fleet()

Se o código que desenvolvemos até agora fosse o necessário para criar uma frota, provavelmente deixaríamos _create_fleet() como está. No entanto, temos mais trabalho a fazer. Assim, limparemos um pouco o método. Adicionaremos um novo método auxiliar, _create_alien(), e o chamaremos a partir de _create_fleet():

alien_invasion.py

```
    def _create_fleet(self):
        -- trecho de código omitido --
        while current_x < (self.settings.screen_width - 2 * alien_width):
            self._create_alien(current_x)
            current_x += 2 * alien_width

1   def _create_alien(self, x_position):
        """Cria um alienígena e o posiciona na fileira"""
        new_alien = Alien(self)
        new_alien.x = x_position
        new_alien.rect.x = x_position
        self.aliens.add(new_alien)
```

O método _create_alien() exige um parâmetro além do self: o valor de *x* que especifica onde o alienígena deve ser posicionado 1. O código no corpo de _create_alien() é o mesmo código que estava em_create_fleet(), exceto que usamos o nome do parâmetro x_position no lugar de current_x. Essa refatoração facilitará a adição de fileiras novas e a criação de uma frota inteira.

Adicionando fileiras

Para terminar a frota, continuaremos adicionando mais fileiras até ficarmos sem espaço. Usaremos um loop aninhado – faremos um wrapper de outro loop while em torno do atual. O loop interno posicionará horizontalmente os alienígenas em uma fileira,

concentrando-se nos valores *x* dos alienígenas. O loop externo posicionará verticalmente os alienígenas, concentrando-se nos valores *y.* Pararemos de adicionar fileiras quando chegarmos perto da parte inferior da tela, deixando espaço suficiente para a espaçonave e um pouco de espaço para começar a disparar contra os alienígenas.

Vejamos como aninhar os dois loops while em_create_fleet():

```
    def _create_fleet(self):
        """Cria a frota de alienígenas"""
        # Cria um alienígena e continua adicionando alienígenas
        # até que não haja mais espaço
        # O distanciamento entre os alienígenas é de uma largura
        # de alienígena e uma altura de alienígena
        alien = Alien(self)
1       alien_width, alien_height = alien.rect.size

2       current_x, current_y = alien_width, alien_height
3       while current_y < (self.settings.screen_height - 3 * alien_height):
            while current_x < (self.settings.screen_width - 2 * alien_width):
4               self._create_alien(current_x, current_y)
                current_x += 2 * alien_width

5           # Termina uma fileira; redefine o valor de x, e incrementa o valor de y
            current_x = alien_width
            current_y += 2 * alien_height
```

Como é necessário saber a altura do alienígena para inserir fileiras, pegamos a largura e a altura do alienígena usando o atributo size do rect de um alienígena 1. O atributo size de um rect é uma tupla contendo sua largura e altura.

Em seguida, definimos os valores iniciais de *x* e de *y* para posicionar o primeiro alienígena na frota 2. Inserimos essa largura de alienígena à esquerda e a altura de alienígena de cima para baixo. Depois, definimos o loop while que controla quantas fileiras são posicionadas na tela 3. Desde que o valor de *y* para a próxima fileira seja menor que a altura da tela, menos três alturas de alienígenas, continuaremos adicionando fileiras. (Se isso não permitir a quantidade adequada de espaço, podemos ajustá-lo mais tarde.)

Chamamos _create_alien(), e passamos o valor de *y*, bem como sua posição *x* 4. Modificaremos _create_alien() daqui a pouco.

Repare na indentação das duas últimas linhas do código 5. Essas linhas estão dentro do loop while externo, mas fora do loop while interno. Esse bloco é executado após o loop interno ser concluído; é executado uma vez após cada fileira ser criada. Após cada fileira ser adicionada, redefinimos o valor de current_x para que o primeiro alienígena na próxima fileira seja inserido na mesma posição que o primeiro alienígena das fileiras anteriores. Em seguida, adicionamos duas alturas de alienígenas ao valor atual de current_y, assim a próxima fileira será posicionada mais abaixo na tela. Aqui, a indentação é fundamental; se não vir a frota correta ao executar *alien_invasion.py* no final desta seção, verifique a indentação de todas as linhas nesses loops aninhados.

É necessário modificar _create_alien() para definir corretamente a posição vertical do alienígena:

```
    def _create_alien(self, x_position, y_position):
        """Cria um alienígena e o posiciona na frota"""
        new_alien = Alien(self)
        new_alien.x = x_position
        new_alien.rect.x = x_position
        new_alien.rect.y = y_position
        self.aliens.add(new_alien)
```

Modificamos a definição do método a fim de aceitar o valor de *y* para o alienígena novo, e definimos a posição vertical do rect no corpo do método.

Agora, quando executar o jogo, você deve ver uma frota completa de alienígenas, como mostrado na Figura 13.4.

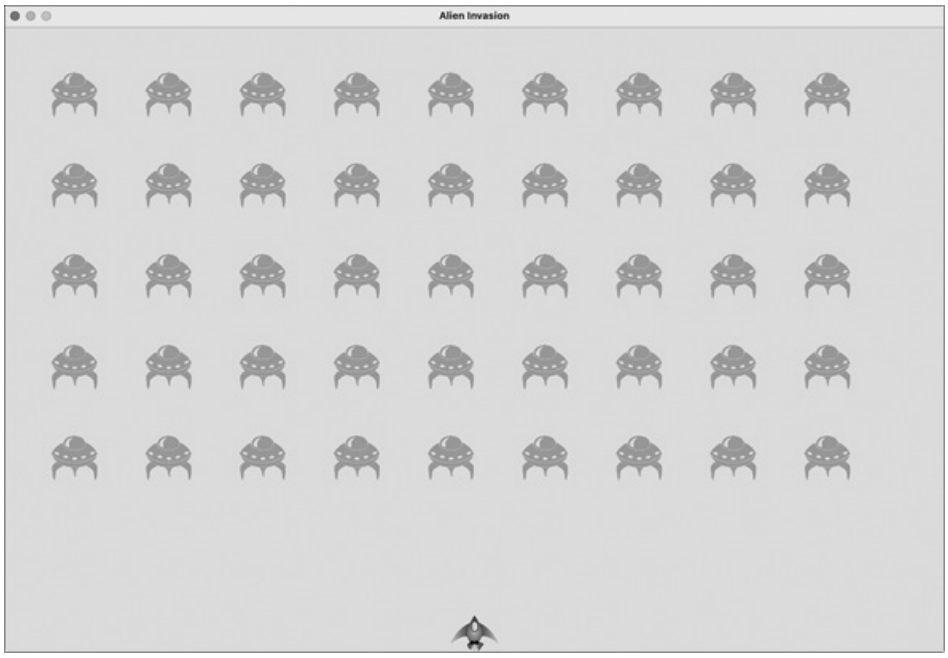


Figura 13.4: Aparece a frota completa.

Na próxima seção, faremos a frota se mover!

FAÇA VOCÊ MESMO

13.1 Estrelas: Encontre a imagem de uma estrela. Faça uma malha de estrelas aparecer na tela.

13.2 Estrelas melhores: É possível criar um padrão de estrela mais realista introduzindo aleatoriedade quando você posiciona uma delas. Lembre-se do Capítulo 9, onde obtivemos um número aleatório como este:

```
from random import randint
random_number = randint(-10, 10)
```

Esse código retorna um número inteiro aleatório entre -10 e 10. Usando seu código do Exercício 13.1, ajuste a posição de cada estrela com um valor aleatório.

Movendo a frota

Agora, vamos fazer a frota de alienígenas se mover para a direita por toda a tela até que alcance a borda e, em seguida, fazê-la descer uma determinada distância e mover-se na outra direção. Continuaremos esse movimento até que todos os alienígenas sejam abatidos ou até que algum deles colida com a espaçonave ou chegue à parte inferior da tela. Começaremos fazendo a frota se

mover para a direita.

Movendo os alienígenas para a direita

Para deslocar os alienígenas, recorreremos ao método update() em *alien.py*. Chamaremos esse método para cada alienígena no grupo de alienígenas. Primeiro, adicionamos uma configuração para controlar a velocidade de cada alienígena:

settings.py

```
    def __init__(self):
        -- trecho de código omitido --
        # Configurações do alienígena
        self.alien_speed = 1.0
```

Em seguida, usamos essa configuração para implementar update() em *alien.py*:

alien.py

```
    def __init__(self, ai_game):
        """Inicializa o alienígena e define sua posição inicial"""
        super().__init__()
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        -- trecho de código omitido --

    def update(self):
        """Move o alienígena para a direita"""
1       self.x += self.settings.alien_speed
2       self.rect.x = self.x
```

Criamos um parâmetro settings em __init__() para que possamos acessar a velocidade do alienígena em update(). Sempre que atualizarmos a posição de um alienígena, vamos movê-lo para a direita conforme a distância armazenada em alien_speed. Rastreamos a posição exata do alienígena com o atributo self.x, que pode armazenar valores floats 1. Depois, usamos o valor de self.x para atualizar a posição do rect 2 do alienígena.

No loop while principal, já temos chamadas para atualizar as posições da espaçonave e dos projéteis. Agora, adicionaremos uma chamada

para atualizar também a posição de cada alienígena:

alien_invasion.py

```
while True:
    self._check_events()
    self.ship.update()
    self._update_bullets()
    self._update_aliens()
    self._update_screen()
    self.clock.tick(60)
```

Como estamos prestes a escrever um pouco de código para gerenciar o movimento da frota, criamos um método novo chamado `_update_aliens()`. Atualizamos as posições dos alienígenas após os projéteis serem atualizados, pois em breve verificaremos se algum projétil abateu algum alienígena.

O lugar em que inserimos esse método no módulo não é crítico. No entanto, para manter o código organizado, vou inseri-lo logo após `_update_bullets()` para coincidir com a ordem das chamadas de método no loop `while`. Vejamos a primeira versão de `_update_aliens()`:

alien_invasion.py

```
def _update_aliens(self):
    """Atualiza as posições de todos os alienígenas na frota"""
    self.aliens.update()
```

Utilizamos o método `update()` no grupo `aliens`, que chama o método `update()` de cada alienígena. Agora, quando executar *Invasão Alienígena*, você deve ver a frota se mover para a direita e desaparecer na lateral da tela.

Criando configurações para a direção da frota

Agora, vamos criar as configurações que farão com que a frota se mova para baixo na tela e para a esquerda quando alcançar a borda direita da tela. Vejamos como implementar esse comportamento:

settings.py

```
# Configurações do alienígena
self.alien_speed = 1.0
```

```
        self.fleet_drop_speed = 10
        # fleet_direction de 1 representa a direita; -1 representa a esquerda
        self.fleet_direction = 1
```

A configuração `fleet_drop_speed` controla a rapidez com que a frota desce na tela sempre que um alienígena alcança uma das bordas. Ajuda a separar essa velocidade da velocidade horizontal dos alienígenas para que possamos ajustar as duas velocidades de forma independente.

Para implementar a configuração `fleet_direction`, poderíamos utilizar um valor de texto como `'left'` ou `'right'`, mas acabaríamos com instruções `if-elif` testando a direção da frota. Em vez disso, como só temos duas direções, utilizaremos os valores 1 e -1 e alternaremos entre eles sempre que a frota mudar de direção. (Utilizar números também faz sentido, já que se deslocar para a direita envolve somar um valor à coordenada *x* de cada alienígena, e se deslocar para a esquerda envolve subtrair um valor da coordenada *x* de cada alienígena.)

Verificando se um alienígena alcançou a borda

Precisamos de um método para verificar se um alienígena está em uma das bordas, e precisamos modificar `update()` a fim de possibilitar que cada alienígena se mova na direção adequada. O seguinte código faz parte da classe `Alien`:

alien.py

```
    def check_edges(self):
        """Retorna True se o alienígena estiver na borda da tela"""
        screen_rect = self.screen.get_rect()
1       return (self.rect.right >= screen_rect.right) or (self.rect.left <= 0)

    def update(self):
        """Desloca o alienígena para a direita ou para a esquerda"""
2       self.x += self.settings.alien_speed * self.settings.fleet_direction
        self.rect.x = self.x
```

Podemos chamar o método novo `check_edges()` em qualquer alienígena a fim de verificarmos se ele está na borda esquerda ou direita. Se o atributo `right` de seu `rect` for maior ou igual ao atributo `right` do `rect` da

tela, o alienígena ficará na borda direita. Agora, se seu valor left for menor ou igual a 0, o alienígena ficará na borda esquerda. 1. Em vez de inserir esse teste condicional em um bloco if, o inserimos diretamente na instrução return. Esse método retornará True se o alienígena estiver na borda direita ou esquerda, e False se não estiver em nenhuma das bordas.

Modificamos o método upadte() para possibilitar o movimento à esquerda ou à direita, multiplicando a velocidade do alienígena pelo valor de fleet_direction 2. Se fleet_direction for 1, o valor de alien_speed será somado à posição atual do alienígena, deslocando-o para a direita; se fleet_direction for -1, o valor será subtraído da posição do alienígena, deslocando-o para a esquerda.

Fazendo a frota descer e mudar de direção

Quando um alienígena alcança a borda, é necessário que toda a frota desça pela tela e mude de direção. Por isso, precisamos adicionar um pouco de código ao AlienInvasion, pois é aí que verificaremos se algum alienígena está na borda esquerda ou direita. Faremos isso escrevendo os métodos _check_fleet_edges() e _change_fleet_direction() e, em seguida, modificando _update_aliens(). Vou inserir esses métodos novos após _create_alien(), no entanto, repito, onde inserir esses métodos na classe não é crítico.

alien_invasion.py

```
    def _check_fleet_edges(self):
        """Responde apropriadamente se algum alienígena alcançou uma borda"""
1       for alien in self.aliens.sprites():
            if alien.check_edges():
2               self._change_fleet_direction()
                break

    def _change_fleet_direction(self):
        """Faz toda a frota descer e mudar de direção"""
        for alien in self.aliens.sprites():
3           alien.rect.y += self.settings.fleet_drop_speed
        self.settings.fleet_direction *= -1
```

Em _check_fleet_edges() percorremos a frota com um loop e chamamos check_edges() em cada alienígena 1. Se check_edges() retornar True, sabemos que um alienígena está em uma borda e toda a frota precisa mudar de direção; assim, chamamos _change_fleet_direction() e saímos do loop 2. Em _change_fleet_direction() percorremos todos os alienígenas com um loop e fizemos cada um descer pela tela com a configuração fleet_drop_speed 3; em seguida, alteramos o valor de fleet_direction multiplicando seu valor atual por -1. A linha que muda a direção da frota não faz parte do loop for. Apesar de querermos mudar a posição vertical de cada alienígena, só queremos mudar a direção da frota uma vez.

Vejamos as alterações de _update_aliens():

alien_invasion.py

```
def _update_aliens(self):
    """Verifica se a frota está na borda e, em seguida, atualiza as posições"""
    self._check_fleet_edges()
    self.aliens.update()
```

Modificamos o método chamando _check_fleet_edges() antes de atualizar a posição de cada alienígena.

Agora, quando executar o jogo, a frota deve se mover para frente e para trás entre as bordas da tela e descer sempre que alcançar uma borda. Podemos então começar a abater alienígenas e ficarmos atentos em qualquer alienígena que acerte a espaçonave ou alcance a parte inferior da tela.

FAÇA VOCÊ MESMO

13.3 Pingos de chuva: Encontre uma imagem de pingos de chuva e crie uma rede deles. Faça com que caiam em direção à parte inferior da tela até que desapareçam.

13.4 Chuva constante: Modifique o código do Exercício 13.3 para que, quando uma fileira de pingos de chuva desaparecer da parte inferior da tela, uma fileira nova apareça na parte superior da tela e comece a cair.

Disparando contra os alienígenas

Por mais que tenhamos criado nossa espaçonave e uma frota de

alienígenas, quando alcançam os alienígenas, os projéteis simplesmente os atravessam, pois ainda não temos colisões. Na programação de jogos, as *colisões* acontecem quando os elementos do jogo se sobrepõem. Para fazer os projéteis dispararem contra alienígenas, usaremos a função sprite.groupcollide() para detectar colisões entre membros de dois grupos.

Detectando colisões de projéteis

Queremos saber o quanto antes quando um projétil atinge um alienígena, pois, desse modo, podemos fazer um alienígena desaparecer assim que for abatido. Para isso, vamos detectar colisões logo após atualizar a posição de todos os projéteis.

A função sprite.groupcollide() compara os rects de cada elemento em um grupo com os rects de cada elemento em outro grupo. Nesse caso, a função compara o rect de cada projétil com o rect de cada alienígena e retorna um dicionário contendo os projéteis e alienígenas que colidiram. Cada chave do dicionário será um projétil, e o valor correspondente será o alienígena abatido. (Usaremos também esse dicionário quando implementarmos um sistema de pontuação no Capítulo 14.)

Adicione o seguinte código ao final de _update_bullets() para verificar se há colisões entre projéteis e alienígenas:

alien_invasion.py

```
    def _update_bullets(self):
        """Atualiza a posição dos projéteis e descarta os antigos"""
        -- trecho de código omitido --

        # Verifica se algum projétil atingiu um alienígena
        # Se sim, descarta o projétil e o alienígena
        collisions = pygame.sprite.groupcollide(
                self.bullets, self.aliens, True, True)
```

O código novo que adicionamos compara as posições de todos os projéteis em self.bullets e de todos os alienígenas em self.aliens, identificando qualquer sobreposição. Sempre que os rects de um

projétil e de um alienígena se sobrepõem, groupcollide() adiciona um par chave-valor ao dicionário que retorna. Os dois argumentos True solicitam que o Pygame delete os projéteis e os alienígenas que colidiram. (Para criar um projétil com alto poder de fogo, que possa se deslocar até a parte superior da tela e que destrua todos os alienígenas que cruzarem seu caminho, podemos definir o primeiro argumento booleano como False e manter o segundo argumento booleano como True. Os alienígenas abatidos desapareceriam, mas todos os projéteis permaneceriam ativos até desaparecerem na parte superior da tela.)

Agora, quando executar *Invasão Alienígenas*, os alienígenas abatidos devem desaparecer. A Figura 13.5 mostra uma frota parcialmente abatida.

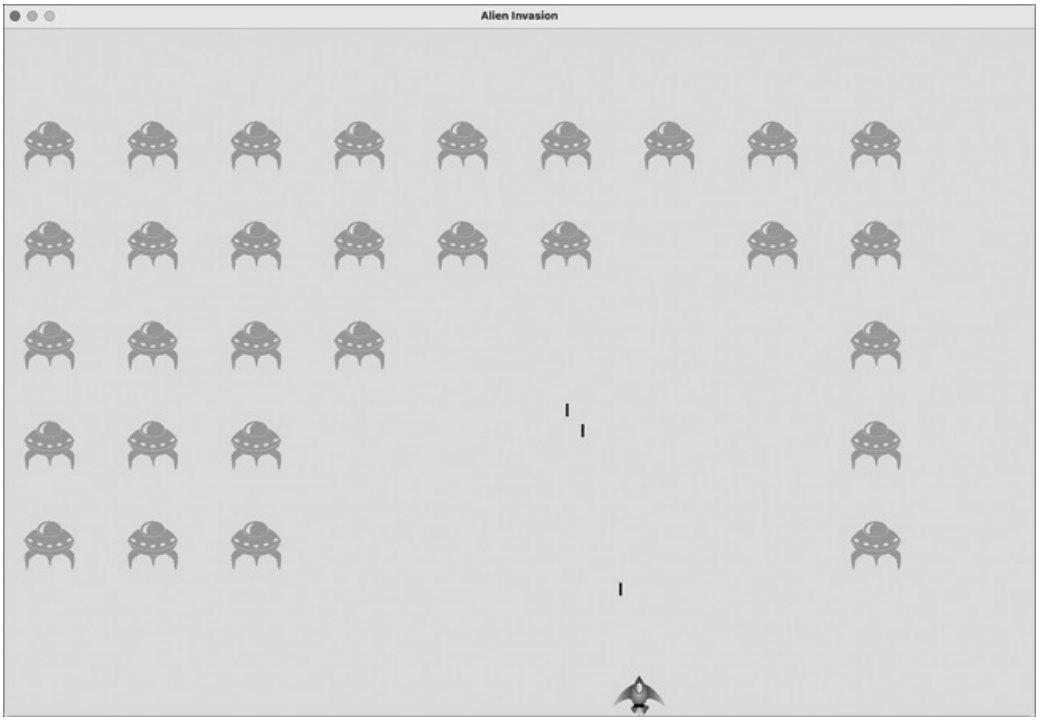


Figura 13.5: Podemos disparar contra os alienígenas!

Criando projéteis maiores para testes

É possível testar muitas funcionalidades do *Invasão Alienígena*, basta executá-lo. No entanto, é cansativo testar algumas delas na versão

normal do jogo. Por exemplo, é muito trabalhoso disparar inúmeras vezes contra todos os alienígenas na tela para testar se o código responde corretamente a uma frota vazia.

Para testar determinadas funcionalidades, podemos alterar determinadas configurações do jogo para que foquem uma área específica. Por exemplo, é possível reduzir o tamanho da tela a fim de que se tenha menos alienígenas contra os quais disparar ou aumentar a velocidade dos projéteis, concedendo uma boa quantidade deles ao jogador de uma vez só.

Minha mudança favorita para testar *Invasão Alienígena* é usar projéteis extremamente grandes, que permaneçam ativos mesmo após abaterem um alienígena (veja a Figura 13.6). Tente definir `bullet_width` para 300, ou mesmo 3.000, para verificar com que rapidez você abate uma frota!

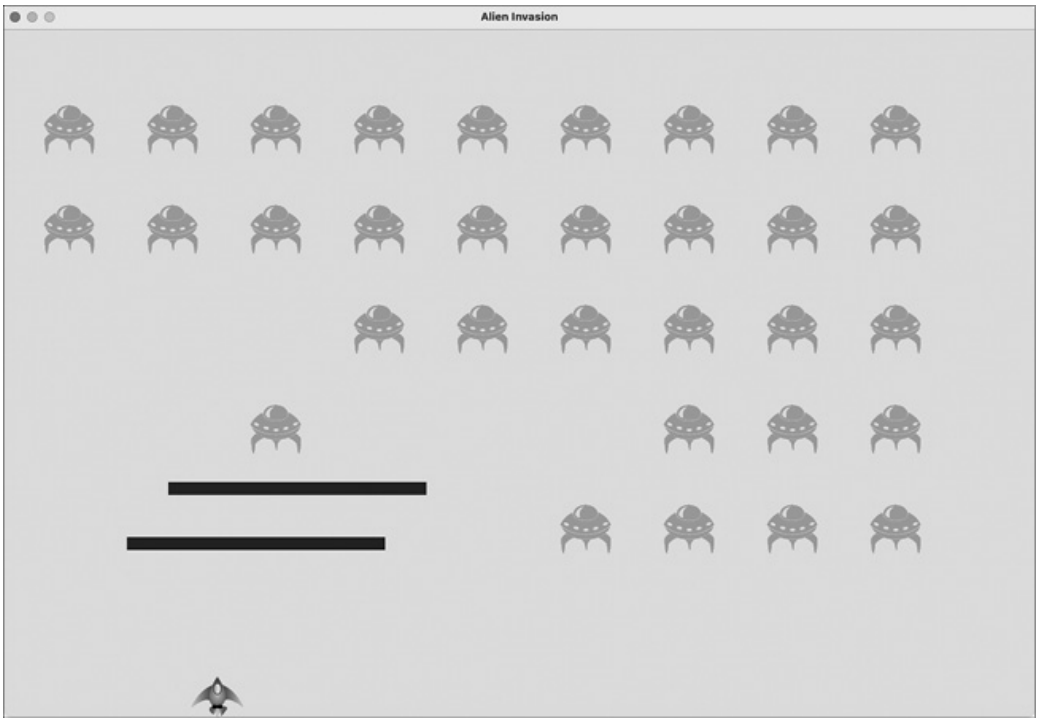


Figura 13.6: Projéteis com grande poder de fogo facilitam o teste de alguns aspectos do jogo.

Mudanças desse tipo o ajudarão a testar o jogo com mais eficiência e, possivelmente, podem estimular ideias para dar aos jogadores

bônus de poderes. Lembre-se de restaurar as configurações para o normal quando terminar de testar uma funcionalidade.

Repopulando a frota

A principal funcionalidade do *Invasão Alienígena* é que os alienígenas são implacáveis: sempre que a frota é destruída, uma frota nova deve aparecer.

Para fazer com que uma frota nova de alienígenas apareça após uma frota ser destruída, primeiro verificamos se o grupo `aliens` está vazio. Se estiver, chamamos `_create_fleet()`. Realizaremos essa verificação no final de `_update_bullets()`, porque é onde os alienígenas são destruídos individualmente.

alien_invasion.py

```
    def _update_bullets(self):
        -- trecho de código omitido --
1       if not self.aliens:
            # Destrói os projéteis existentes e cria uma frota nova
2           self.bullets.empty()
            self._create_fleet()
```

Verificamos se o grupo `aliens` está vazio `1`. Um grupo vazio é avaliado como `False`. Trata-se de uma forma simples de verificar se o grupo está vazio. Se sim, descartamos todos os projéteis existentes usando o método `empty()`, que remove todos os sprites restantes de um grupo `2`. Chamamos também `_create_fleet()`, que preenche a tela com alienígenas novamente.

Agora, uma frota nova aparece assim que a frota atual é destruída.

Criando projéteis mais rápidos

No estado atual do jogo, caso tente disparar contra os alienígenas, talvez você se dê conta de que os projéteis não estão nada rápidos. Os projéteis poder ficam lentos demais ou rápidos demais. Nesse momento, podemos modificar as configurações para tornar a jogabilidade mais interessante. Não se esqueça de que o jogo ficará

gradativamente mais rápido, por isso, não aumente a velocidade logo de início.

Modificamos a velocidade dos projéteis ajustando o valor de bullet_speed em *settings.py*. No meu sistema, ajustarei o valor de bullet_speed para 2.5, assim os projéteis ficam um pouco mais velozes:

settings.py

```
# Configurações dos projéteis
self.bullet_speed = 2.5
self.bullet_width = 3
-- trecho de código omitido --
```

O melhor valor para essa configuração depende de sua experiência no jogo, então encontre um valor que funcione para você. É possível também ajustar outras configurações.

Refatorando _update_bullets()

Vamos refatorar _update_bullets() para que não realize tantas tarefas diferentes. Vamos transferir o código que lida com colisões alienígenas e projéteis para um método separado:

alien_invasion.py

```
def _update_bullets(self):
    -- trecho de código omitido --
    # Descarta os projéteis que desapareceram
    for bullet in self.bullets.copy():
        if bullet.rect.bottom <= 0:
            self.bullets.remove(bullet)

    self._check_bullet_alien_collisions()

def _check_bullet_alien_collisions(self):
    """Responde à colisões alienígenas"""
    # Remove todos os projéteis e os alienígenas que tenham colidido
    collisions = pygame.sprite.groupcollide(
            self.bullets, self.aliens, True, True)

    if not self.aliens:
        # Destrói os projéteis existentes e cria uma frota nova
```

```
            self.bullets.empty()
            self._create_fleet()
```

Criamos um método novo, `_check_bullet_alien_collisions()`, para detectar colisões entre projéteis e alienígenas e para responder adequadamente à destruição de toda a frota. Isso impede que `_update_bullets()` aumente muito e simplifica desenvolvimentos posteriores.

FAÇA VOCÊ MESMO

13.5 Disparos laterais parte 2: Percorremos um longo caminho desde o Exercício 12.6, *Disparos laterais*. Neste exercício, tente incrementar *Disparos laterais* como fizemos com *Invasão Alienígena*. Adicione uma frota de alienígenas, e a faça se deslocar lateralmente em direção à espaçonave. Ou, desenvolva um código que posicione alienígenas em posições aleatórias ao longo do lado direito da tela e, em seguida, os envie à espaçonave. Além disso, escreva um código que faça os alienígenas desaparecerem quando forem abatidos.

Encerrando o jogo

Qual é a graça e o desafio de um jogo se não perdermos? Se o jogador não disparar contra a frota rápido o bastante, faremos com que os alienígenas destruam a espaçonave quando fizerem contato. Ao mesmo tempo, restringiremos a quantidade de espaçonaves que um jogador pode usar, e destruiremos a espaçonave quando um alienígena chegar à parte inferior da tela. O jogo encerrará quando o jogador tiver usado todas as suas espaçonaves.

Detectando colisões entre espaçonaves e alienígenas

Começaremos verificando se há colisões entre alienígenas e a espaçonave. Desse modo, podemos responder adequadamente quando um alienígena abatê-la. Verificaremos as colisões entre alienígenas e a espaçonave logo após atualizar a posição de cada alienígena em `AlienInvasion`:

alien_invasion.py

```
    def _update_aliens(self):
        -- trecho de código omitido --
```

```
        self.aliens.update()

        # Detecta colisões entre alienígenas e espaçonaves
1       if pygame.sprite.spritecollideany(self.ship, self.aliens):
2           print("Ship hit!!!")
```

A função spritecollideany() tem dois argumentos: um sprite e um grupo. A função analisa qualquer membro do grupo que colidiu com o sprite e para de percorrer o grupo com um loop assim que encontra um membro que colidiu com o sprite. Aqui, a função percorre o grupo aliens com um loop e retorna o primeiro alienígena que encontrou e que colidiu com ship.

Se não ocorrerem colisões, spritecollideany() retorna None e o bloco if não executará 1. Se detectar um alienígena que colidiu com a espaçonave, a função retorna esse alienígena e o bloco if executa: exibe Ship hit!! 2. Quando um alienígena abater uma espaçonave, é necessário realizar uma série de tarefas: deletar todos os alienígenas e projéteis restantes, recentralizar a espaçonave e criar uma frota nova. Antes de desenvolvermos o código para fazer tudo isso, queremos saber se nossa abordagem para detectar colisões entre espaçonaves e alienígenas funciona corretamente. Escrever um print() é um forma simples de garantir que estamos detectando corretamente essas colisões.

Agora, quando executarmos *Invasão Alienígena,* a mensagem *Ship hit!!!* deve aparecer no terminal sempre que um alienígena derrubar uma espaçonave. Ao testar essa funcionalidade, defina fleet_drop_speed com um valor maior, como 50 ou 100, para que os alienígenas alcancem sua espaçonave com mais rapidez.

Respondendo à colisões entre espaçonaves e alienígenas

Agora, precisamos identificar exatamente o que acontecerá quando um alienígena colidir contra uma espaçonave. Em vez de destruir a instância de ship e criar uma nova, contabilizaremos quantas vezes a espaçonave foi abatida recorrendo às estatísticas de rastreamento

do jogo. As estatísticas de rastreamento também ajudarão com a pontuação.

Escreveremos uma classe nova, GameStats, para rastrear as estatísticas do jogo e a salvaremos como *game_stats.py*:

game_stats.py

```
class GameStats:
    """Rastreia as estatísticas de Invasão Alienígena"""

    def __init__(self, ai_game):
        """Inicializa as estatísticas"""
        self.settings = ai_game.settings
1       self.reset_stats()

    def reset_stats(self):
        """Inicializa as estatísticas que podem mudar durante o jogo"""
        self.ships_left = self.settings.ship_limit
```

Criaremos uma instância de GameStats durante todo o tempo em que *Invasão Alienígena* estiver em execução, mas será necessário redefinir algumas estatísticas sempre que o jogador iniciar um jogo novo. Para fazer isso, inicializaremos a maioria das estatísticas no método reset_stats(), e não diretamente em __init__(). Vamos chamar esse método a partir de __init__() para que as estatísticas sejam adequadamente definidas quando a instância de GameStats for criada 1. No entanto, também podemos chamar reset_stats() sempre que o jogador iniciar um jogo novo. Por ora, temos somente um dado estatístico, ships_left, cujo valor mudará durante o jogo.

O número de espaçonaves com os quais o jogador começa a jogar deve ser armazenado em *settings.py* como ship_limit:

settings.py

```
        # Configurações da espaçonave
        self.ship_speed = 1.5
        self.ship_limit = 3
```

Precisamos também fazer algumas alterações em *alien_invasion.py* para criar uma instância de GameStats. Primeiro, atualizaremos as instruções import na parte superior do arquivo:

alien_invasion.py

```
import sys
from time import sleep

import pygame

from settings import Settings
from game_stats import GameStats
from ship import Ship
-- trecho de código omitido --
```

Importamos a função sleep() do módulo time da biblioteca padrão do Python. Assim, podemos pausar o jogo por um momento quando a espaçonave for abatida. Importamos também GameStats.

Criaremos uma instância de GameStats em __init__():

alien_invasion.py

```
    def __init__(self):
        -- trecho de código omitido --
        self.screen = pygame.display.set_mode(
            (self.settings.screen_width, self.settings.screen_height))
        pygame.display.set_caption("Alien Invasion")

        # Cria uma instância para armazenar estatísticas do jogo
        self.stats = GameStats(self)

        self.ship = Ship(self)
        -- trecho de código omitido --
```

Criamos a instância após criar a janela do jogo, mas antes de definir outros elementos, como a espaçonave.

Quando um alienígena abater uma espaçonave, subtrairemos 1 do número de espaçonaves restantes, destruiremos todos os alienígenas e projéteis existentes, criaremos uma frota nova e reposicionaremos a espaçonave no meio da tela. Pausaremos também o jogo por um momento para que o jogador possa ver a colisão e reagrupar antes que uma frota nova apareça.

Vamos inserir parte desse código em um método novo chamado _ship_hit(). Chamaremos esse método a partir de _update_alienens() quando um alienígena abater uma espaçonave:

alien_invasion.py

```
    def _ship_hit(self):
        """Responde à espaçonave sendo abatida por um alienígena"""
        # Decrementa ships_left
1       self.stats.ships_left -= 1

        # Descarta quaisquer projéteis e alienígenas restantes
2       self.bullets.empty()
        self.aliens.empty()

        # Cria uma frota nova e centraliza a espaçonave
3       self._create_fleet()
        self.ship.center_ship()

        # Pausa
4       sleep(0.5)
```

O método novo _ship_hit() coordena a resposta quando um alienígena abate uma espaçonave. Dentro de _ship_hit(), o número de espaçonaves restantes é reduzido em 1. Depois, esvaziamos os grupos bullets e aliens 2.

Em seguida, criamos uma frota nova e centralizamos a espaçonave 3. (Vamos adicionar o método center_ship() à Ship daqui a pouco.) Em seguida, após as atualizações terem sido feitas, adicionamos uma pausa em todos os elementos do jogo, mas antes que quaisquer alterações sejam desenhadas na tela, assim, o jogador consegue ver que sua espaçonave foi abatida 4. A chamada sleep() pausa a execução do programa por meio segundo, tempo suficiente para que o jogador veja que o alienígena abateu sua espaçonave. Quando a função sleep() termina, a execução do código passa para o método _update_screen(), que desenha a frota nova na tela.

Em _update_aliens(), substituímos o print() por uma chamada para _ship_hit() quando um alienígena abate uma espaçonave:

alien_invasion.py

```
    def _update_aliens(self):
        -- trecho de código omitido --
```

```
        if pygame.sprite.spritecollideany(self.ship, self.aliens):
            self._ship_hit()
```

Vejamos o método novo center_ship(), que pertence a *ship.py*:

ship.py

```
    def center_ship(self):
        """Centraliza a espaçonave na tela"""
        self.rect.midbottom = self.screen_rect.midbottom
        self.x = float(self.rect.x)
```

Centralizamos a espaçonave da mesma forma que em __init__(). Depois de centralizá-la, redefinimos o atributo self.x, que nos possibilita rastrear a posição exata da espaçonave.

NOTA *Perceba que nunca criamos mais de uma espaçonave; criamos somente uma instância da espaçonave para todo o jogo e a recentralizamos sempre que a espaçonave é abatida. A estatística* ships_left *nos relatará quando o jogador ficar sem espaçonaves.*

Execute o jogo, dispare contra alguns alienígenas, e deixe um alienígena abater a espaçonave. O jogo deve pausar e uma frota nova deve aparecer com a espaçonave centralizada na parte inferior da tela novamente.

Alienígenas que alcançam a parte inferior da tela

Caso um alienígena alcance a parte inferior da tela, faremos o jogo responder como quando um alienígena abate uma espaçonave. Para verificar quando isso acontece, adicione um método novo em *alien_invasion.py*:

alien_invasion.py

```
    def _check_aliens_bottom(self):
        """Verifica se algum alienígena chegou à parte inferior da tela"""
        for alien in self.aliens.sprites():
1           if alien.rect.bottom >= self.settings.screen_height:
                # Trata isso como se a espaçonave tivesse sido abatida
                self._ship_hit()
                break
```

O método _check_aliens_bottom() verifica se algum alienígena alcançou a parte inferior da tela. Um alienígena chega à parte inferior quando seu valor rect.bottom é maior ou igual à altura da tela 1. Caso um alienígena alcance a parte inferior da tela, chamamos _ship_hit(). Se um alienígena alcança a parte inferior, não é necessário verificar o resto, então saímos do loop após chamar _ship_hit().

Chamaremos esse método a partir de _update_aliens():

alien_invasion.py

```
def _update_aliens(self):
    -- trecho de código omitido --
    # Detecta colisões entre alienígenas e espaçonaves
    if pygame.sprite.spritecollideany(self.ship, self.aliens):
        self._ship_hit()

    # Procura por alienígenas se chocando contra a parte inferior da tela
    self._check_aliens_bottom()
```

Chamamos _check_alienens_bottom() após atualizar as posições de todos os alienígenas e depois de detectar colisões entre espaçonaves e alienígenas. Agora, uma frota nova aparecerá sempre que a espaçonave for abatida por um alienígena ou um alienígena chegar à parte inferior da tela.

Game Over!

Por mais que *Invasão Alienígena* pareça mais completo agora, um jogo nunca acaba. O valor de ships_left fica cada vez mais negativo. Adicionaremos a flag game_active para que possamos encerrar o jogo quando o jogador ficar sem espaçonaves. Definiremos essa flag no final do método __init__() em AlienInvasion:

alien_invasion.py

```
def __init__(self):
    -- trecho de código omitido --
    # Inicializa Invasão Alienígena em um estado ativo
    self.game_active = True
```

Agora adicionamos código a _ship_hit() que define game_active como False

quando o jogador tiver usado todas as suas espaçonaves:

alien_invasion.py

```
def _ship_hit(self):
    """ Responde à espaçonave sendo abatida por um alienígena"""
    if self.stats.ships_left > 0:
        # Decrementa ships_left
        self.stats.ships_left -= 1
        -- trecho de código omitido --
        # Pausa
        sleep(0.5)
    else:
        self.game_active = False
```

A maior parte de `_ship_hit()` permanece inalterada. Movemos todo o código existente para um bloco `if`. Desse modo, testamos e garantimos que o jogador tenha, pelo menos, uma espaçonave restante. Se tiver, criamos uma frota nova, fazemos uma pausa e seguimos em frente. Se o jogador não tiver mais espaçonaves, definimos `game_active` como `False`.

Identificando quais partes do jogo devem ser executadas

Precisamos identificar as partes do jogo que devem sempre ser executadas e as partes que devem ser executadas somente quando o jogo estiver ativo:

alien_invasion.py

```
def run_game(self):
    """Inicia o loop principal do jogo"""
    while True:
        self._check_events()

        if self.game_active:
            self.ship.update()
            self._update_bullets()
            self._update_aliens()

        self._update_screen()
        self.clock.tick(60)
```

No loop principal, sempre precisamos chamar _check_events(), mesmo que o jogo esteja inativo. Por exemplo, ainda é necessário saber se o usuário pressiona Q para sair do jogo ou clica no botão para fechar a janela. Continuamos também atualizando a tela para que possamos fazer mudanças nela enquanto esperamos se o jogador inicia um jogo novo ou não. As chamadas restantes de função precisam acontecer apenas quando o jogo está ativo, já que se o jogo estiver inativo, não precisamos atualizar as posições dos elementos do jogo.

Agora, ao jogar *Invasão Alienígena*, o jogo deve congelar quando você tiver usado todas as suas espaçonaves.

FAÇA VOCÊ MESMO

13.6 Game Over: Em *disparos laterais*, mantenha o registro da quantidade de vezes em que uma espaçonave é abatida e do número de vezes em que um alienígena é abatido pela espaçonave. Estabeleça uma condição adequada para encerrar o jogo, e pare o jogo quando essa situação ocorrer.

Recapitulando

Neste capítulo, aprendemos a adicionar um grande número de elementos idênticos a um jogo criando uma frota de alienígenas. Usamos loops aninhados para criar uma grade de elementos e para criar um conjunto grande de elementos que viabilizou movimentos para o jogo ao chamarmos o método update() de cada elemento. Vimos como controlar a direção dos objetos na tela e a responder a situações específicas, como quando a frota alcança a parte inferior da tela. Detectamos e respondemos a colisões quando projéteis abateram alienígenas e alienígenas abateram espaçonaves. Aprendemos também a rastrear estatísticas em um jogo e usar a flag game_active para determinar quando o jogo termina.

No próximo e último capítulo deste projeto, adicionaremos um botão Play a fim de que o jogador possa escolher quando iniciar o jogo e se jogará mais uma vez quando o jogo acabar. Faremos com que o

jogo fique mais rápido sempre que o jogador abater toda a frota e adicionaremos um sistema de pontuação. O resultado final será um jogo totalmente pronto para jogar!

CAPÍTULO 14
Pontuação

Neste capítulo, concluiremos o jogo *Invasão Alienígena*. Vamos adicionar um botão Play para iniciar o jogo sempre que o jogador quiser, e reiniciá-lo assim que encerrar. Além disso, alteraremos o jogo para que fique mais rápido quando o jogador passar de nível e implementaremos um sistema de pontuação. No final do capítulo, você saberá o bastante para começar a desenvolver jogos que aumentem em dificuldade conforme o jogador progride e que apresentem sistemas completos de pontuação.

Adicionando o botão Play

Nesta seção, adicionaremos um botão Play que aparecerá antes do jogo começar e reaparecerá quando o jogo terminar. Assim, o jogador consegue jogar novamente.

Agora, o jogo começa assim que executarmos *alien_invasion.py*. Iniciaremos o jogo em um estado inativo e, em seguida, pediremos ao jogador que clique em um botão Play para começar. Para tal, modifique o método `__init__()` de `AlienInvasion`:

alien_invasion.py

```
    def __init__(self):
        """Inicializa o jogo e sobe suas funcionalidades"""
        pygame.init()
        -- trecho de código omitido --

        # Inicia Invasão Alienígena em um estado inativo
        self.game_active = False
```

A partir deste momento, o jogo deve começar em um estado inativo,

e o jogador não consegue iniciá-lo até criarmos o botão Play.

Criando uma classe Button

Como o Pygame não tem um método built-in para criar botões, vamos escrever uma classe Button para criar um retângulo preenchido com um rótulo (label). Use esse código para criar qualquer botão de um jogo. Vejamos a primeira parte da classe Button; salve-a como *button.py*:

button.py

```
import pygame.font

class Button:
    """Classe para criar botões para o jogo"""

1   def __init__(self, ai_game, msg):
        """Inicializa os atributos do botão"""
        self.screen = ai_game.screen
        self.screen_rect = self.screen.get_rect()

        # Define as dimensões e propriedades do botão
2       self.width, self.height = 200, 50
        self.button_color = (0, 135, 0)
        self.text_color = (255, 255, 255)
3       self.font = pygame.font.SysFont(None, 48)

        # Cria o objeto rect do botão e o centraliza
4       self.rect = pygame.Rect(0, 0, self.width, self.height)
        self.rect.center = self.screen_rect.center

        # A mensagem do botão precisa ser preparada apenas uma vez
5       self._prep_msg(msg)
```

Inicialmente, importamos o módulo pygame.font, que possibilita que o Pygame renderize um texto na tela. O método __init__() aceita os parâmetros self, o objeto ai_game e msg, que contém o texto do botão 1. Definimos as dimensões do botão 2, definimos button_color para atribuir a cor verde-escuro ao objeto rect do botão e definimos text_color para renderizar o texto em branco.

Depois, preparamos um atributo font para renderizar o texto 3. O

argumento None informa ao Pygame que use a fonte default, e 48 especifica o tamanho do texto. Para centralizar o botão na tela, criamos um rect para o botão 4 e definimos seu atributo como center para corresponder ao da tela.

O Pygame funciona com texto ao renderizar a string que queremos exibir como uma imagem. Por último, chamamos_prep_msg() para lidar com essa renderização 5.

Vejamos o código de_prep_msg():

button.py

```
    def _prep_msg(self, msg):
        """Transforma msg em uma imagem renderizada e centraliza texto no botão"""
1       self.msg_image = self.font.render(msg, True, self.text_color,
                self.button_color)
2       self.msg_image_rect = self.msg_image.get_rect()
        self.msg_image_rect.center = self.rect.center
```

O método _prep_msg() precisa de um parâmetro self e do texto que deve ser renderizado como uma imagem (msg). A chamada para font.render() transforma o texto armazenado em msg em uma imagem, que armazenamos em self.msg_image 1. O método font.render() também utiliza um valor booleano para ativar ou desativar o antialiasing (o *antialiasing* ajuda a suavizar as bordas do texto). Os argumentos remanescentes são a cor da fonte especificada e a cor do background. Definimos o antialiasing como True e definimos o background do texto como a mesma cor do botão. (Se não incluir uma cor de background, o Pygame tentará renderizar a fonte com um background transparente.)

Centralizamos a imagem do texto no botão, criando um rect a partir da imagem e definindo seu atributo como center para corresponder ao do botão 2.

Por último, criamos um método draw_button() que podemos chamar para exibir o botão na tela:

button.py

```
    def draw_button(self):
        """Desenha o botão em branco e depois desenhe a mensagem"""
        self.screen.fill(self.button_color, self.rect)
        self.screen.blit(self.msg_image, self.msg_image_rect)
```

Chamamos screen.fill() para desenhar a parte retangular do botão. Em seguida, chamamos screen.blit() para desenhar a imagem de texto na tela, passando uma imagem e o objeto rect associado à imagem. Concluímos a classe Button.

Desenhando o botão na tela

Usaremos a classe Button para criar um botão Play em AlienInvasion. Primeiro, atualizaremos as instruções import:

alien_invasion.py

```
-- trecho de código omitido --
from game_stats import GameStats
from button import Button
```

Como precisamos de somente um botão Play, criaremos o botão no método __init__() de AlienInvasion. Podemos inserir esse código no final de __init__():

alien_invasion.py

```
    def __init__(self):
        -- trecho de código omitido --
        self.game_active = False

        # Cria o botão Play
        self.play_button = Button(self, "Play")
```

Esse código cria uma instância de Button com o rótulo Play, mas não desenha o botão na tela. Para isso, chamaremos o método draw_button() do botão em _update_screen():

alien_invasion.py

```
    def _update_screen(self):
        -- trecho de código omitido --
        self.aliens.draw(self.screen)

        # Desenha o botão Play se o jogo estiver inativo
        if not self.game_active:
```

```
        self.play_button.draw_button()

        pygame.display.flip()
```

Para que o botão Play fique mais visível do que todos os outros elementos na tela, o desenhamos após todos os outros elementos serem desenhados, mas antes de alternarmos para uma tela nova. Incluímos o botão em um bloco if, assim o botão só aparece quando o jogo está inativo.

Agora, quando executarmos *Invasão Alienígena*, devemos ver um botão Play no centro da tela, como mostrado na Figura 14.1.

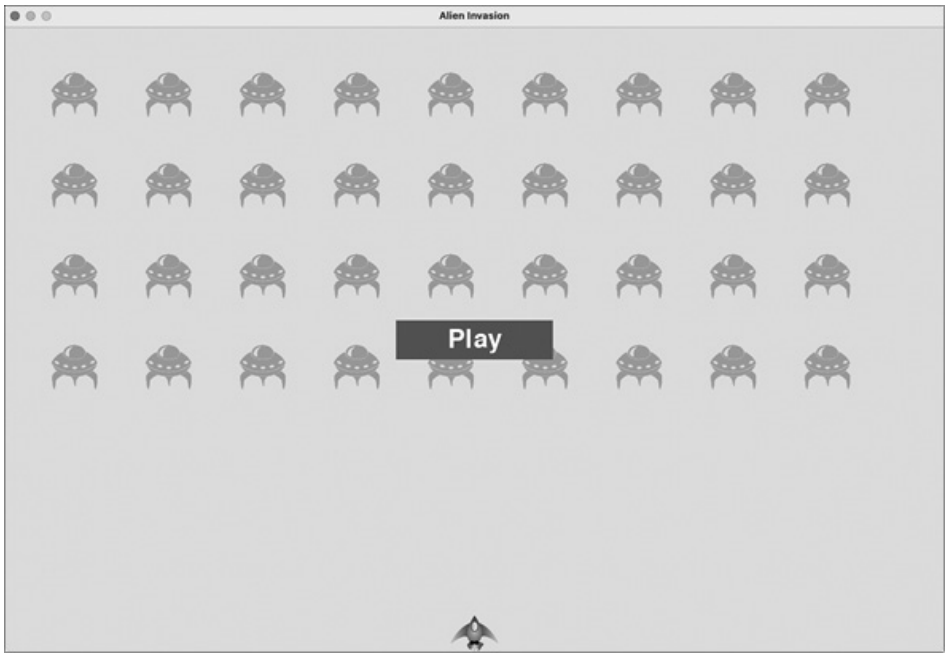


Figura 14.1: Um botão Play aparece quando o jogo está inativo.

Iniciando o jogo

Para iniciar um novo jogo quando o jogador clicar em Play, adicionamos o seguinte bloco elif ao final de _check_events() a fim de monitorarmos os eventos do mouse sobre o botão:

alien_invasion.py

```
    def _check_events(self):
        """Responde às teclas pressionadas e aos eventos do mouse"""
        for event in pygame.event.get():
```

```
            if event.type == pygame.QUIT:
                -- trecho de código omitido --
1           elif event.type == pygame.MOUSEBUTTONDOWN:
2               mouse_pos = pygame.mouse.get_pos()
3               self._check_play_button(mouse_pos)
```

Apesar de o Pygame detectar um evento MOUSEBUTTONDOWN quando o jogador clica em qualquer lugar da tela 1, queremos restringir nosso jogo para responder aos cliques do mouse apenas sobre o botão Play. Desse modo, usamos pygame.mouse.get_pos() que retorna uma tupla contendo as coordenadas *x* e *y* do cursor do mouse quando o botão do mouse é clicado 2. Enviamos esses valores para o método novo _check_play_button() 3.

Vejamos o _check_play_button(), optei por inseri-lo após _check_events():

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Inicia um jogo novo quando o jogador clica em Play"""
1       if self.play_button.rect.collidepoint(mouse_pos):
            self.game_active = True
```

Utilizamos o rect do método collidepoint() para verificar se o ponto do clique do mouse se sobrepõe à região definida pelo rect do botão Play 1. Se sim, definimos game_active como True e o jogo começa!

Neste momento, você deve conseguir iniciar o jogo e jogá-lo por completo. Quando o jogo terminar, o valor de game_active deve se tornar False e o botão Play deve reaparecer.

Reiniciando o jogo

O código do botão que acabamos de desenvolver funciona na primeira vez que o jogador clicar em Play. No entanto, não funciona mais depois que o primeiro jogo termina, já que as condições que encerram o jogo não foram redefinidas.

Para reiniciar o jogo sempre que o jogador clicar em Play, precisamos reiniciar as estatísticas do jogo, limpar os alienígenas e projéteis anteriores, criar uma frota nova e centralizar a espaçonave, como podemos ver:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Inicia um jogo novo quando o jogador clica em Play"""
        if self.play_button.rect.collidepoint(mouse_pos):
            # Redefine as estatísticas do jogo
1           self.stats.reset_stats()
            self.game_active = True

            # Descarta quaisquer projéteis e alienígenas restantes
2           self.bullets.empty()
            self.aliens.empty()

            # Cria uma frota nova e centraliza a espaçonave
3           self._create_fleet()
            self.ship.center_ship()
```

Redefinimos as estatísticas do jogo 1 e, como resultado, o jogador tem três espaçonaves novas. Depois, definimos game_active como True para que o jogo comece assim que o código nesta função acabar de ser executado. Esvaziamos os grupos aliens e bullets 2 e, em seguida, criamos uma frota nova e centralizamos a espaçonave 3.

Agora, o jogo será reiniciado adequadamente sempre que clicarmos em Play, possibilitando jogar quantas vezes quisermos!

Desativando o botão Play

O problema com nosso botão Play é que a região do botão na tela continuará respondendo aos cliques, mesmo quando o botão Play não estiver visível. Se clicarmos na área do botão Play sem querer após o início do jogo, o jogo será reiniciado!

Para corrigir isso, defina o jogo para começar apenas quando game_active for False:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Inicia um jogo novo quando o jogador clica em Play"""
1       button_clicked = self.play_button.rect.collidepoint(mouse_pos)
2       if button_clicked and not self.game_active:
            # Redefine as estatísticas do jogo
```

```
            self.stats.reset_stats()
            -- trecho de código omitido --
```

A flag button_clicked armazena um valor True ou False 1, e o jogo será reiniciado somente se Play for clicado *e* se o jogo não estiver ativo no momento 2. Para testar esse comportamento, inicie um jogo novo e clique repetidamente onde o botão Play deve estar. Se tudo funcionar como esperado, clicar na área do botão Play não deve afetar a jogabilidade.

Ocultando o cursor do mouse

Queremos que o cursor do mouse fique visível quando o jogo estiver inativo, mas assim que o jogo começar, o botão atrapalha. Para corrigir isso, vamos torná-lo invisível quando o jogo se tornar ativo. É possível fazer isso no final do bloco if em _check_play_button():

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Inicia um jogo novo quando o jogador clica em Play"""
        button_clicked = self.play_button.rect.collidepoint(mouse_pos)
        if button_clicked and not self.game_active:
            -- trecho de código omitido --
            # Oculta o cursor do mouse
            pygame.mouse.set_visible(False)
```

Passar False para set_visible() informa ao Pygame para ocultar o cursor quando o mouse estiver sobre a janela do jogo.

Faremos o cursor reaparecer assim que o jogo terminar. Desse modo, o jogador consegue clicar em Play novamente para começar um jogo novo. Vejamos o código que resolve isso:

alien_invasion.py

```
    def _ship_hit(self):
        """Responde à espaçonave sendo abatida por um alienígena"""
        if self.stats.ships_left > 0:
            -- trecho de código omitido --
        else:
            self.game_active = False
            pygame.mouse.set_visible(True)
```

Deixamos o cursor visível mais uma vez assim que o jogo fica inativo, o que acontece em _ship_hit(). A atenção a detalhes desse tipo faz com que o jogo fique mais profissional e possibilita que o jogador se concentre em jogar, em vez de tentar compreender como a interface do usuário funciona.

FAÇA VOCÊ MESMO

14.1 Pressione P para Jogar: Já que o Jogo *Invasão Alienígena* usa entradas do teclado para controlar a espaçonave, seria útil iniciar o jogo quando uma tecla é pressionada. Adicione um código que possibilite ao jogador pressionar P para iniciar. Talvez passar algum código de _check_play_button() para um método _start_game(), que pode ser chamado de _check_play_button() *e de* _check_keydown_events(), ajude.

14.2 Tiro ao alvo: Crie um retângulo na borda direita da tela que se mova para cima e para baixo a um ritmo constante. Depois, ao lado esquerdo da tela, crie uma espaçonave que o jogador possa mover para cima e para baixo enquanto dispara projéteis contra o alvo retangular. Adicione um botão Play que inicia o jogo e, quando o jogador errar o alvo três vezes, encerre o jogo e faça o botão Play reaparecer. Permita que o jogador reinicie o jogo com o botão Play.

Passando de nível

Em nosso jogo atual, uma vez que um jogador derruba toda a frota alienígena, o jogador passa para um nível novo, mas a dificuldade do jogo não. Vamos animar um pouco as coisas, possibilitando que o jogo fique mais desafiador, aumentando sua velocidade sempre que um jogador limpar a tela.

Modificando as configurações de velocidade

Primeiro, reorganizaremos a classe Settings para agrupar as configurações do jogo em estáticas e dinâmicas. Garantiremos também que quaisquer configurações que mudem durante o jogo sejam redefinidas quando iniciarmos um novo jogo. Veja a seguir o método __init__() para *settings.py*:

settings.py

```
    def __init__(self):
        """Inicializa as configurações estáticas do jogo"""
        # Configurações de tela
```

```
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)

        # Configurações da espaçonave
        self.ship_limit = 3

        # Configurações dos projéteis
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
        self.bullets_allowed = 3

        # Configurações do alienígena
        self.fleet_drop_speed = 10

        # A rapidez com que o jogo acelera
1       self.speedup_scale = 1.1

2       self.initialize_dynamic_settings()
```

Continuamos a inicializar as configurações que permanecem constantes no método `__init__()`. Adicionamos a configuração `speedup_scale` `1` para controlar a rapidez com que o jogo acelera: um valor de 2 dobrará a velocidade do jogo sempre que o jogador passar para um nível novo; um valor de 1 manterá a velocidade constante. Um valor como `1.1` deve aumentar a velocidade o suficiente para que o jogo fique desafiador, mas não impossível de jogar. Por último, chamamos o método `initialize_dynamic_settings()` para inicializar os valores dos atributos que precisam mudar ao longo do jogo `2`.

Vejamos o código de `initialize_dynamic_settings()`:

settings.py

```
    def initialize_dynamic_settings(self):
        """Inicializa as configurações que mudam ao longo do jogo"""
        self.ship_speed = 1.5
        self.bullet_speed = 2.5
        self.alien_speed = 1.0

        # fleet_direction de 1 representa a direita; -1 representa a esquerda
        self.fleet_direction = 1
```

Esse método define os valores iniciais para as velocidades da espaçonave, dos projéteis e dos alienígenas. Vamos aumentar essas velocidades conforme o jogador progride no jogo e vamos redefini-las sempre que o jogador iniciar um jogo novo. Incluímos fleet_direction nesse método para que os alienígenas sempre se desloquem à direita no início de um jogo novo. Não é necessário aumentar o valor de fleet_drop_speed, porque quando se movem mais rápido pela tela, os alienígenas também descem pela tela mais rápido.

A fim de aumentar as velocidades da espaçonave, dos projéteis e dos alienígenas sempre que o jogador passa para o próximo nível, escreveremos um método novo chamado increase_speed():

settings.py

```
def increase_speed(self):
    """Aumenta as configurações de velocidade"""
    self.ship_speed *= self.speedup_scale
    self.bullet_speed *= self.speedup_scale
    self.alien_speed *= self.speedup_scale
```

Para aumentar a velocidade desses elementos do jogo, multiplicamos cada configuração de velocidade pelo valor de speedup_scale.

Aumentamos o ritmo do jogo chamando increase_speed() em _check_bullet_alien_collisions() quando o último alienígena de uma frota for abatido:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    -- trecho de código omitido --
    if not self.aliens:
        # Destrói os projéteis existentes e cria uma frota nova.
        self.bullets.empty()
        self._create_fleet()
        self.settings.increase_speed()
```

Alterar os valores das configurações de velocidade de ship_speed, alien_speed e bullet_speed é o bastante para acelerar a velocidade de todo o jogo!

Redefinindo a velocidade

Agora, precisamos restaurar quaisquer configurações alteradas aos valores iniciais sempre que o jogador inicia um jogo novo; do contrário, cada jogo novo começaria com as configurações de velocidade aumentadas do jogo anterior:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Inicia um jogo novo quando o jogador clica em Play"""
        button_clicked = self.play_button.rect.collidepoint(mouse_pos)
        if button_clicked and not self.game_active:
            # Redefine as configurações do jogo
            self.settings.initialize_dynamic_settings()
            -- trecho de código omitido --
```

Agora, jogar *Invasão Alienígena* deve ser mais divertido e desafiador. Sempre que limpar a tela, o jogo deve ficar mais rápido e um pouco mais difícil. Se o jogo ficar excessivamente difícil e rápido, diminua o valor de `settings.speedup_scale`. Ou, se o jogo não ficar desafiador o suficiente, aumente um pouco o valor. Encontre um meio-termo intensificando a dificuldade em um período razoável de tempo. Os primeiros níveis devem ser fáceis, os próximos devem ser desafiadores, mas possíveis, e os níveis posteriores devem ser quase impossíveis de tão difíceis.

FAÇA VOCÊ MESMO

14.3 Tiro ao alvo desafiador: Comece com seu código do Exercício 14.2 (página *353*). Faça com que o alvo se mova mais rápido à medida que o jogo avança e reinicie o alvo para a velocidade original quando o jogador clicar em Play.

14.4 Níveis de dificuldade: Desenvolva um conjunto de botões para *Invasão Alienígena* que permite ao jogador selecionar um nível de dificuldade inicial adequado para o jogo. Cada botão deve atribuir os valores apropriados aos atributos necessários em Settings para criar diferentes níveis de dificuldade.

Pontuação

Implementaremos um sistema de pontuação para acompanhar a pontuação do jogo em tempo real e exibir a pontuação máxima, o

nível e a quantidade de espaçonaves restantes.

Como a pontuação é uma estatística de jogo, vamos adicionar um atributo score à GameStats.

game_stats.py

```
class GameStats:
    -- trecho de código omitido --
    def reset_stats(self):
        """Inicializa as estatísticas que podem mudar durante o jogo"""
        self.ships_left = self.ai_settings.ship_limit
        self.score = 0
```

Para redefinir a pontuação sempre que um jogo novo começar, inicializamos score em reset_stats() em vez de __init__().

Exibindo a pontuação

Para exibir a pontuação na tela, primeiro criamos uma classe nova, Scoreboard. Por ora, essa classe apenas exibirá a pontuação atual. Futuramente, vamos usá-la também para informar a pontuação máxima, o nível e a quantidade de espaçonaves restantes. Vejamos a primeira parte da classe; salve-a como *scoreboard.py*:

scoreboard.py

```
import pygame.font

class Scoreboard:
    """Classe para exibir informações de pontuação"""

1   def __init__(self, ai_game):
        """Inicializa os atributos de pontuação"""
        self.screen = ai_game.screen
        self.screen_rect = self.screen.get_rect()
        self.settings = ai_game.settings
        self.stats = ai_game.stats

        # Configurações de fonte para informações de pontuação
2       self.text_color = (30, 30, 30)
3       self.font = pygame.font.SysFont(None, 48)

        # Prepara a imagem inicial da pontuação
4       self.prep_score()
```

Como Scoreboard escreve texto na tela, começamos importando o módulo pygame.font. Em seguida, fornecemos a __init__() o parâmetro ai_game para que possa acessar os objetos settings, screen e stats, necessários para informar os valores que estamos rastreando 1. Em seguida, definimos uma cor de texto 2 e instanciamos um objeto de fonte 3.

Para transformar o texto a ser exibido em uma imagem, chamamos prep_score() 4, que definiremos a seguir:

scoreboard.py

```
    def prep_score(self):
        """Transforma a pontuação em uma imagem renderizada"""
1       score_str = str(self.stats.score)
2       self.score_image = self.font.render(score_str, True,
                self.text_color, self.settings.bg_color)

        # Exibe a pontuação no canto superior direito da tela
3       self.score_rect = self.score_image.get_rect()
4       self.score_rect.right = self.screen_rect.right - 20
5       self.score_rect.top = 20
```

Em prep_score(), transformamos o valor numérico stats.score em uma string 1 e, depois, passamos essa string para render(), que cria a imagem 2. Para exibir a pontuação nitidamente na tela, passamos a cor do background da tela e a cor do texto para render().

Vamos fixar a pontuação no canto superior direito da tela e expandi-la para a esquerda à medida que aumenta e o comprimento do número cresce. Para assegurar que a pontuação esteja sempre alinhada com o lado direito da tela, criamos um rect chamado score_rect 3 e definimos sua borda direita a 20 pixels da borda direita da tela 4. Em seguida, inserimos a borda superior 20 pixels abaixo da parte superior da tela 5.

Depois, criamos um método show_score() para exibir a imagem de pontuação renderizada:

scoreboard.py

```
    def show_score(self):
        """Desenha a pontuação na tela"""
        self.screen.blit(self.score_image, self.score_rect)
```

Esse método desenha a imagem da pontuação na tela e no local determinado por score_rect.

Criando um scoreboard

A fim de exibir a pontuação, criaremos uma instância de Scoreboard em AlienInvasion. Primeiro, atualizaremos as instruções import:

alien_invasion.py

```
-- trecho de código omitido --
from game_stats import GameStats
from scoreboard import Scoreboard
-- trecho de código omitido --
```

Em seguida, criamos uma instância de Scoreboard em __init__():

alien_invasion.py

```
    def __init__(self):
        -- trecho de código omitido --
        pygame.display.set_caption("Alien Invasion")

        # Cria uma instância para armazenar estatísticas do jogo,
        #   e cria um scoreboard
        self.stats = GameStats(self)
        self.sb = Scoreboard(self)
        -- trecho de código omitido --
```

Depois, desenhamos o scoreboard na tela em _update_screen():

alien_invasion.py

```
    def _update_screen(self):
        -- trecho de código omitido --
        self.aliens.draw(self.screen)

        # Desenha as informações da pontuação
        self.sb.show_score()

        # Desenha o botão Play se o jogo estiver inativo
        -- trecho de código omitido --
```

Chamamos show_score() pouco antes de desenharmos o botão Play.

Agora, quando executar *Invasão Alienígena*, um 0 deve aparecer no canto superior direito da tela. (Aqui, queremos apenas ter certeza de que a pontuação aparece no lugar adequado antes de desenvolver ainda mais o sistema de pontuação.) A Figura 14.2 mostra a pontuação antes do início do jogo.

A seguir, atribuiremos valores de pontos a cada alienígena.

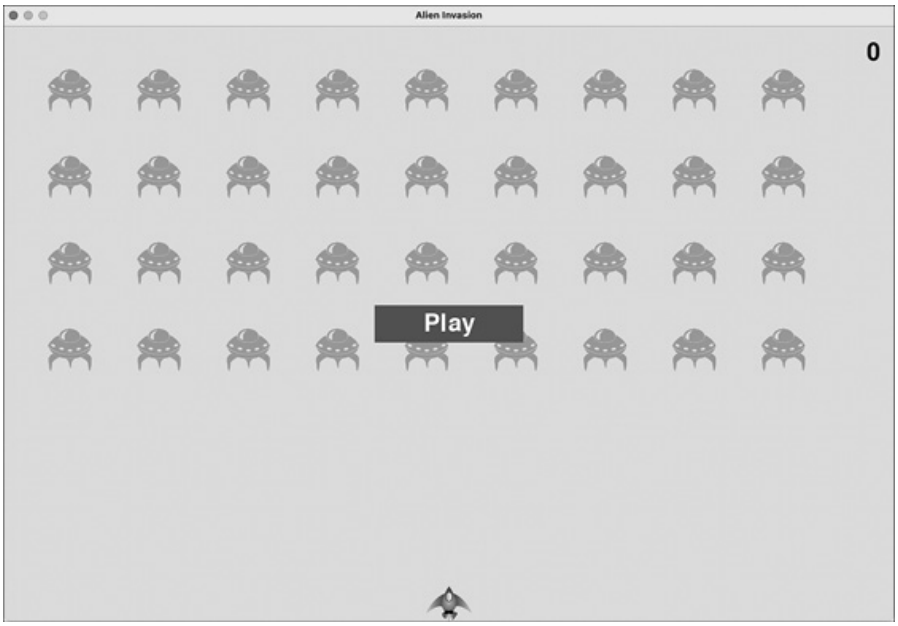


Figura 14.2: A pontuação aparece no canto superior direito da tela.

Atualizando a pontuação conforme os alienígenas são abatidos

Para escrever a pontuação em tempo real na tela, atualizamos o valor de stats.score sempre que um alienígena é abatido e, em seguida, chamamos prep_score() para atualizar a imagem da pontuação. Mas, antes, definiremos quantos pontos um jogador ganha sempre que abater um alienígena:

settings.py

```
def initialize_dynamic_settings(self):
    -- trecho de código omitido --

    # Configurações de pontuação
    self.alien_points = 50
```

Vamos aumentar o valor dos pontos de cada alienígena conforme o jogo avança. Para garantir que esse valor de ponto seja redefinido sempre que um jogo novo for iniciado, definimos o valor em initialize_dynamic_settings().

Vamos atualizar a pontuação em _check_bullet_alien_collisions() sempre que um alienígena é abatido:

alien_invasion.py

```
    def _check_bullet_alien_collisions(self):
        """Responde às colisões alienígenas"""
        # Remove todos os projéteis e os alienígenas que tenham colidido
        collisions = pygame.sprite.groupcollide(
                self.bullets, self.aliens, True, True)

        if collisions:
            self.stats.score += self.settings.alien_points
            self.sb.prep_score()
        -- trecho de código omitido --
```

Quando um projétil atinge um alienígena, o Pygame retorna um dicionário collisions. Verificamos se o dicionário existe e, se existir, o valor do alienígena é somado à pontuação. Depois, chamamos prep_score() a fim de criar uma imagem nova para a pontuação atualizada.

Agora, quando jogar *Invasão Alienígena*, você deve ser capaz de acumular pontos!

Redefinindo a pontuação

No momento, estamos somente preparando uma nova pontuação *após* um alienígena ser abatido, o que funciona na maior parte do jogo. No entanto, ao iniciarmos um jogo novo, ainda veremos nossa pontuação do jogo anterior até que o primeiro alienígena seja abatido.

Podemos corrigir isso preparando a pontuação ao iniciar um jogo novo:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    -- trecho de código omitido --
    if button_clicked and not self.game_active:
        -- trecho de código omitido --
        # Redefine as estatísticas do jogo
        self.stats.reset_stats()
        self.sb.prep_score()
        -- trecho de código omitido --
```

Chamamos `prep_score()` após redefinir as estatísticas do jogo ao iniciar um jogo novo. Isso prepara o scoreboard com a pontuação de 0.

Assegurando que todos os ataques sejam pontuados

Do modo que está escrito, nosso código pode não calcular os pontos de alguns alienígenas. Por exemplo, caso dois projéteis colidam contra alienígenas durante a mesma passagem pelo loop ou se criarmos um projétil extremamente grande para abater diversos alienígenas, o jogador só ganha pontos por um dos alienígenas abatidos. Para corrigir isso, refinaremos a detecção das colisões entre alienígenas e projéteis.

Em `_check_bullet_alien_collisions()`, qualquer projétil que colida contra um alienígena se torna uma chave no dicionário `collisions`. O valor associado a cada projétil é uma lista de alienígenas contra os quais o projétil colidiu. Percorremos com um loop os valores no dicionário `collisions` para garantir que atribuímos pontos para cada ataque alienígena.

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    -- trecho de código omitido --
    if collisions:
        for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
-- trecho de código omitido --
```

Se o dicionário `collisions` tiver sido definido, percorreremos com um loop todos os valores no dicionário. Não se esqueça de que cada valor é uma lista de alienígenas abatidos por um único projétil.

Multiplicamos o valor de cada alienígena pelo número de alienígenas em cada lista e somamos esse valor à pontuação atual. Caso queira fazer um teste, altere a largura de um projétil para 300 pixels e verifique se recebe pontos para cada alienígena abatido com projéteis demasiadamente grandes; em seguida, redefina a largura do projétil ao seu valor normal.

Aumentando os valores dos pontos

Como o jogo fica mais difícil sempre que um jogador passa para um nível novo, alienígenas dos próximos níveis devem valer mais pontos. Vamos implementar essa funcionalidade adicionando um código para aumentar o valor do ponto quando a velocidade do jogo aumentar:

settings.py

```
class Settings:
    """Classe para armazenar todas as configurações para Invasão Alienígena"""


    def __init__(self):
        -- trecho de código omitido --
        # A rapidez com que o jogo acelera
        self.speedup_scale = 1.1
        # Com que rapidez os valores dos pontos alienígenas aumentam
1       self.score_scale = 1.5

        self.initialize_dynamic_settings()

    def initialize_dynamic_settings(self):
        -- trecho de código omitido --

    def increase_speed(self):
        """Aumenta configurações de velocidade e valores dos pontos alienígenas"""
        self.ship_speed *= self.speedup_scale
        self.bullet_speed *= self.speedup_scale
        self.alien_speed *= self.speedup_scale

2       self.alien_points = int(self.alien_points * self.score_scale)
```

Definimos uma taxa em que os pontos aumentam, que chamamos

de score_scale 1. Um singelo aumento na velocidade (1.1) faz com que o jogo fique rapidamente mais desafiador. No entanto, para vermos uma diferença acentuada na pontuação, é necessário alterar o valor do ponto alienígena para um valor maior (1.5). Agora, quando aumentamos a velocidade do jogo, aumentamos também o valor do ponto de cada ataque 2. Utilizamos a função int() para aumentar o valor do ponto com números inteiros.

Para ver o valor de cada alienígena, adicione um print() ao método increase_speed() em Settings:

settings.py

```
def increase_speed(self):
    -- trecho de código omitido --
    self.alien_points = int(self.alien_points * self.score_scale)
    print(self.alien_points)
```

O novo valor de ponto deve aparecer no terminal sempre que passarmos para o próximo nível.

NOTA *Não deixe de remover o* print() *após verificar se o valor do ponto está aumentando, ou isso pode afetar o desempenho do jogo e distrair o jogador.*

Arredondando a pontuação

Já que maioria dos jogos de tiro no estilo arcade apresenta as pontuações como múltiplos de 10, faremos a mesma coisa com a nossa. Além disso, formataremos a pontuação para incluir vírgulas como separadores de números grandes. Faremos essa mudança em Scoreboard:

scoreboard.py

```
def prep_score(self):
    """Transforma a pontuação em uma imagem renderizada"""
    rounded_score = round(self.stats.score, -1)
    score_str = f"{rounded_score:,}"
    self.score_image = self.font.render(score_str, True,
            self.text_color, self.settings.bg_color)
    -- trecho de código omitido --
```

A função round() normalmente arredonda um float para um número definido de casas decimais fornecido como o segundo argumento. Apesar disso, quando passamos um número negativo como segundo argumento, round() arredondará o valor para o 10, 100, 1.000 mais próximo, e assim por diante. Esse código instruí o Python a arredondar o valor de stats.score para o múltiplo mais próximo e atribuí-lo a rounded_score.

Em seguida, usamos um especificador de formato na f-string para a pontuação. Um *especificador de formato* é uma sequência especial de caracteres que modifica a forma como o valor de uma variável é apresentado. Aqui, a sequência:, instruí o Python a inserir vírgulas em locais adequados no valor numérico fornecido. Isso resulta em strings como 1,000,000 em vez de 1000000.

Agora, quando executamos o jogo, devemos ver uma pontuação elegantemente formatada e arredondada, mesmo quando acumular muitos pontos, conforme mostrado na Figura 14.3.

Figura 14.3: Uma pontuação arredondada com vírgulas como separadores.

Pontuações máximas

Como todo jogador quer bater a pontuação máxima de um jogo, vamos rastrear e informar as pontuações máximas para que os jogadores se sintam desafiados. Armazenaremos as pontuações máximas em GameStats:

game_stats.py

```
    def __init__(self, ai_game):
        -- trecho de código omitido --
        # A pontuação máxima nunca deve ser redefinida
        self.high_score = 0
```

Como a pontuação máxima nunca deve ser redefinida, inicializamos high_score em __init__() em vez de reset_stats().

A seguir, modificaremos Scoreboard para exibir a pontuação máxima. Começaremos com o método __init__():

scoreboard.py

```
    def __init__(self, ai_game):
        -- trecho de código omitido --
        # Prepare as imagens da pontuação inicial
        self.prep_score()
1       self.prep_high_score()
```

A pontuação máxima será exibida separada da pontuação. Ou seja, precisamos de um método novo, prep_high_score(), para preparar a imagem de pontuação máxima 1.

Vejamos método prep_high_score():

scoreboard.py

```
    def prep_high_score(self):
        """Transforma a pontuação em uma imagem renderizada"""
1       high_score = round(self.stats.high_score, -1)
        high_score_str = f"{high_score:,}"
2       self.high_score_image = self.font.render(high_score_str, True,
                self.text_color, self.settings.bg_color)

        # Centraliza a pontuação máxima na parte superior da tela
        self.high_score_rect = self.high_score_image.get_rect()
```

```
3        self.high_score_rect.centerx = self.screen_rect.centerx
4        self.high_score_rect.top = self.score_rect.top
```

Arredondamos a pontuação máxima para o múltiplo mais próximo e a formatamos com vírgulas 1. Em seguida, geramos uma imagem a partir da pontuação máxima 2, centralizamos horizontalmente o rect da pontuação máxima 3 e definimos seu atributo top para que fique igual à imagem da pontuação 4.

Agora, o método show_score() desenha a pontuação atual no canto superior direito e a pontuação máxima no centro superior da tela:

scoreboard.py

```
def show_score(self):
    """Desenha a pontuação na tela"""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
```

Para verificar as pontuações máximas, escreveremos um método novo, check_high_score(), em Scoreboard:

scoreboard.py

```
def check_high_score(self):
  """ Verifica se há uma nova pontuação máxima"""
    if self.stats.score > self.stats.high_score:
        self.stats.high_score = self.stats.score
        self.prep_high_score()
```

O método check_high_score() verifica a pontuação atual em relação à pontuação máxima. Se a pontuação atual for maior, atualizamos o valor de high_score e chamamos prep_high_score() para atualizar a imagem da pontuação máxima.

É necessário chamar check_high_score() sempre que um alienígena é abatido após atualizar a pontuação em _check_bullet_alien_collisions():

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    -- trecho de código omitido --
    if collisions:
        for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
```

```
            self.sb.check_high_score()
        -- trecho de código omitido --
```

Chamamos check_high_score() quando o dicionário collisions está presente, e apenas fazemos isso após atualizar a pontuação de todos os alienígenas abatidos.

A primeira vez que jogar *Invasão Alienígena*, sua pontuação será a pontuação máxima e será exibida como pontuação atual e pontuação mais máxima. No entanto, ao iniciar um segundo jogo, sua pontuação máxima deve aparecer no meio na tela e sua pontuação atual deve aparecer à direita, como mostrado na Figura 14.4.

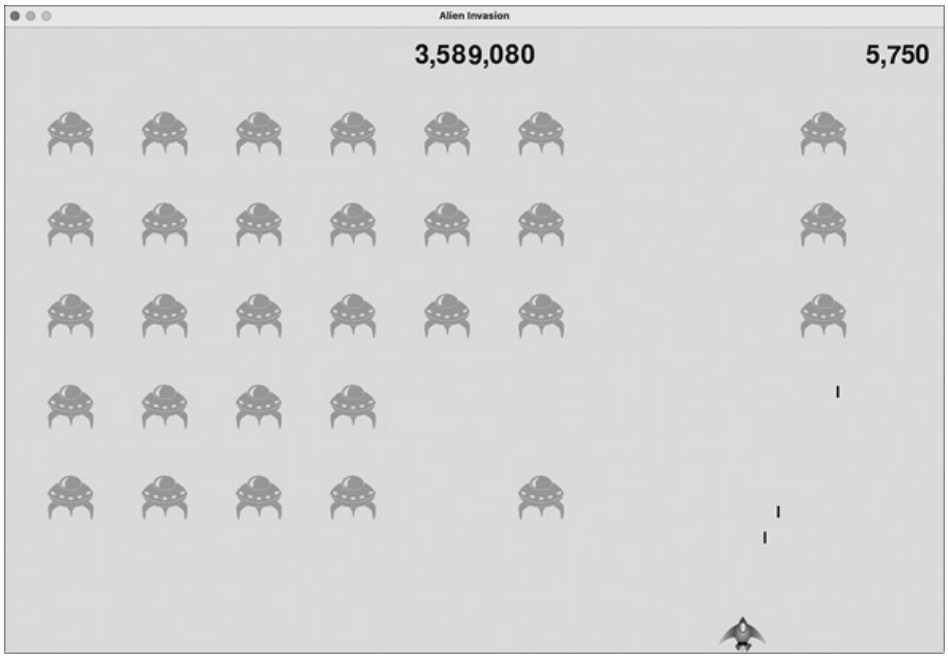


Figura 14.4: A pontuação máxima é mostrada no centro superior da tela.

Exibindo o nível

Para exibir o nível do jogador no jogo, é necessário primeiro um atributo em GameStats representando o nível atual. Para redefinir o nível no início de cada jogo novo, inicialize-o em reset_stats():

game_stats.py

```
    def reset_stats(self):
        """Inicializa as estatísticas que podem mudar durante o jogo"""
        self.ships_left = self.settings.ship_limit
        self.score = 0
        self.level = 1
```

Para que Scoreboard exiba o nível atual, chamamos um método novo, prep_level(), a partir de __init__():

scoreboard.py

```
    def __init__(self, ai_game):
        -- trecho de código omitido --
        self.prep_high_score()
        self.prep_level()
```

Vejamos o prep_level():

scoreboard.py

```
    def prep_level(self):
        """Transforma o nível em uma imagem renderizada"""
        level_str = str(self.stats.level)
1       self.level_image = self.font.render(level_str, True,
                self.text_color, self.settings.bg_color)

        # Posiciona o nível abaixo da pontuação
        self.level_rect = self.level_image.get_rect()
2       self.level_rect.right = self.score_rect.right
3       self.level_rect.top = self.score_rect.bottom + 10
```

O método prep_level() cria uma imagem a partir do valor armazenado em stats.level 1, definindo o atributo right da imagem para que corresponda ao atributo right da pontuação 2. Em seguida, define o atributo top 10 pixels abaixo da parte inferior da imagem de pontuação, permitindo espaço entre a pontuação e o nível 3.

Além disso, é necessário atualizar o show_score():

scoreboard.py

```
    def show_score(self):
        """Desenha as pontuações e o nível na tela"""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
        self.screen.blit(self.level_image, self.level_rect)
```

Essa linha nova desenha a imagem do nível na tela.

Vamos incrementar stats.level e atualizar a imagem do nível em _check_bullet_alien_collisions():

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    -- trecho de código omitido --
    if not self.aliens:
        # Destrói os projéteis existentes e cria uma frota nova
        self.bullets.empty()
        self._create_fleet()
        self.settings.increase_speed()

        # Aumenta o nível
        self.stats.level += 1
        self.sb.prep_level()
```

Se uma frota for destruída, incrementamos o valor de stats.level e chamamos prep_level() para garantir que o nível novo seja adequadamente exibido.

A fim de garantir que a imagem do nível seja devidamente atualizada no início de um jogo novo, chamamos também prep_level() quando o jogador clica no botão Play:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    -- trecho de código omitido --
    if button_clicked and not self.game_active:
        -- trecho de código omitido --
        self.sb.prep_score()
        self.sb.prep_level()
        -- trecho de código omitido --
```

Chamamos prep_level() logo após chamar prep_score().

Agora, é possível ver quantos níveis foram completados, conforme mostrado na Figura 14.5.

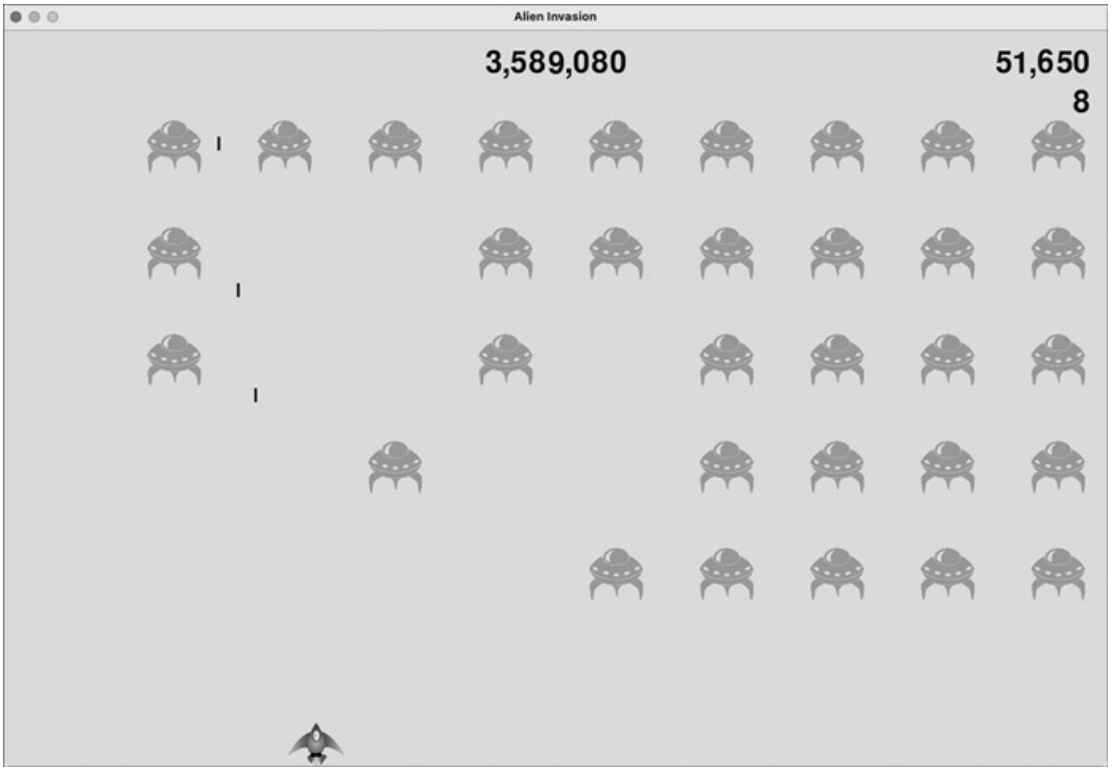


Figura 14.5: O nível atual aparece logo abaixo da pontuação atual.

NOTA *Em alguns jogos clássicos, as pontuações têm rótulos, como Pontuação, Pontuação Máxima e Nível. Omitimos os rótulos, pois o significado de cada número fica claro depois que jogamos. Para incluir esses rótulos, adicione-os às strings de pontuação logo antes das chamadas para* font.render() *em* Scoreboard*.*

Exibindo a quantidade de espaçonaves

Por último, exibiremos a quantidade restante de espaçonaves do jogador, mas, desta vez, utilizaremos ícones. Para tal, desenharemos espaçonaves no canto superior esquerdo da tela a fim de representar a quantidade restante delas, como em muitos jogos clássicos de arcade.

De início, é necessário fazer com que Ship herde de Sprite. Assim, podemos criar um grupo de espaçonaves:

ship.py

```
import pygame
from pygame.sprite import Sprite

1 class Ship(Sprite):
    """Classe para gerenciar a espaçonave"""

    def __init__(self, ai_game):
        """Inicializa a espaçonave e define sua posição inicial"""
2       super().__init__()
        -- trecho de código omitido --
```

Aqui, importamos Sprite, asseguramos que Ship herde de Sprite 1 e chamamos super() no início de __init__() 2.

Depois, precisamos modificar Scoreboard para criar um grupo de espaçonaves que possamos exibir. Vejamos as instruções import para Scoreboard:

scoreboard.py

```
import pygame.font
from pygame.sprite import Group

from ship import Ship
```

Já que estamos criando um grupo de espaçonaves, importamos as classes Group e Ship.

Vamos conferir o __init__():

scoreboard.py

```
    def __init__(self, ai_game):
        """Inicializa os atributos de pontuação"""
        self.ai_game = ai_game
        self.screen = ai_game.screen
        -- trecho de código omitido --
        self.prep_level()
        self.prep_ships()
```

Atribuímos a instância do jogo a um atributo, pois precisaremos dele para criar algumas espaçonaves. Chamamos prep_ships() após a chamada para prep_level().

Vamos conferir prep_ships():

scoreboard.py

```
    def prep_ships(self):
        """Mostra as espaçonaves restantes"""
1       self.ships = Group()
2       for ship_number in range(self.stats.ships_left):
            ship = Ship(self.ai_game)
3           ship.rect.x = 10 + ship_number * ship.rect.width
4           ship.rect.y = 10
5           self.ships.add(ship)
```

O método prep_ships() cria um grupo vazio, self.ships, a fim de armazenar as instâncias da espaçonave 1. Para preencher esse grupo, um loop é executado uma vez em cada espaçonave restante do jogador 2. Dentro do loop, criamos uma espaçonave nova e definimos o valor da coordenada *x* de cada espaçonave para que apareçam uma ao lado da outra, com uma margem de 10 pixels ao lado esquerdo do grupo de espaçonaves 3. Definimos o valor da coordenada *y* 10 pixels abaixo da parte superior da tela a fim de que as espaçonaves apareçam no canto superior esquerdo da tela 4. Em seguida, adicionamos cada espaçonave nova ao grupo ships 5.

Agora, é necessário desenhar as espaçonaves na tela.

scoreboard.py

```
    def show_score(self):
        """Desenha as pontuações, o nível e as espaçonaves na tela"""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
        self.screen.blit(self.level_image, self.level_rect)
        self.ships.draw(self.screen)
```

Para exibir as espaçonaves na tela, chamamos draw() no grupo, e o Pygame desenha cada uma delas.

Para mostrar quantas espaçonaves o jogador têm de início, chamamos prep_ships() quando um jogo novo começa. Fazemos isso em _check_play_button() em AlienInvasion:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        -- trecho de código omitido --
        if button_clicked and not self.game_active:
            -- trecho de código omitido --
```

```
        self.sb.prep_level()
        self.sb.prep_ships()
        -- trecho de código omitido --
```

Chamamos também prep_ships() quando uma espaçonave é abatida, assim, atualizamos a exibição de imagens da espaçonave quando o jogador perde uma delas:

alien_invasion.py

```
    def _ship_hit(self):
        """Responde à espaçonave sendo abatida por um alienígena"""
        if self.stats.ships_left > 0:
            # Decrementa ships_left e atualiza scoreboard
            self.stats.ships_left -= 1
            self.sb.prep_ships()
            -- trecho de código omitido --
```

Chamamos prep_ships() após decrementar o valor de ships_left. Desse modo, a quantidade correta de espaçonaves restantes é exibida sempre que uma delas é destruída.

A Figura 14.6 mostra o sistema de pontuação completo, com as espaçonaves restantes exibidas no canto superior esquerdo da tela.

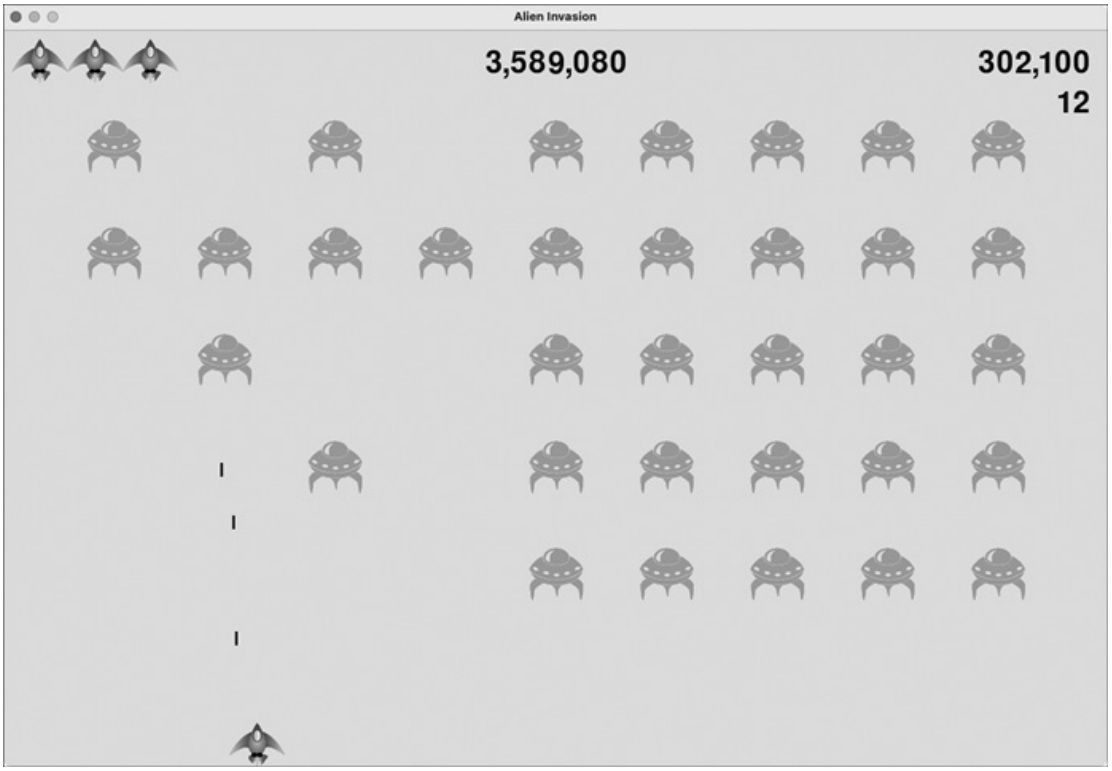


Figura 14.6: O sistema completo de pontuação do Jogo *Invasão Alienígena*.

FAÇA VOCÊ MESMO

14.5 Recorde de pontuação máxima: a pontuação máxima é redefinida sempre que um jogador fecha e reinicia o jogo *Invasão Alienígena*. Corrija isso escrevendo a pontuação máxima em um arquivo antes de chamar sys.exit() e lendo-a ao inicializar seu valor em GameStats.

14.6 Refatoração: Procure métodos que estão fazendo mais de uma tarefa e refatore-os a fim de organizar seu código e torná-lo eficiente. Por exemplo, transfira uma parte do código em _check_bullet_alien_collisions(), que inicia um nível novo quando a frota de alienígenas é destruída, para uma função chamada start_new_level(). Transfira também as quatro chamadas de método separadas no método __init__() em Scoreboard para um método chamado prep_images() para reduzir __init__(). O método prep_images() também pode ajudar a simplificar_check_play_button() ou start_game() se você já tiver refatorado _check_play_button().

NOTA Antes de tentar refatorar o projeto, confira o Apêndice D para saber como restaurá-lo ao estado anterior caso introduza bugs durante a refatoração.

14.7 Desenvolvendo mais o jogo: Pense em uma forma de expandir *Invasão Alienígena*. Por exemplo, é possível programar os alienígenas para que disparem contra sua espaçonave. É possível também adicionar escudos para que sua espaçonave seja protegida e que possam ser destruídos pelos projéteis de ambos os lados. Ou você pode usar o módulo pygame.mixer para adicionar efeitos sonoros, como explosões e sons de tiro.

14.8 Disparos laterais, versão final: Continue *desenvolvendo Disparos* Laterais, usando tudo o que fizemos neste projeto. Adicione um botão Play, acelere a velocidade do jogo adequadamente e desenvolva um sistema de pontuação. Faça questão de refatorar seu código conforme o programa e procure oportunidades para personalizar o jogo além do que foi mostrado neste capítulo.

Recapitulando

Neste capítulo, aprendemos a implementar um botão Play para iniciar um jogo novo. Vimos também como detectar eventos do mouse e ocultar o cursor em jogos ativos. É possível usar o que aprendemos para criar outros botões, como um botão Help, para exibir instruções sobre jogos. Além disso, aprendemos como modificar a velocidade de um jogo à medida que avança, como implementar um sistema de pontuação gradativa e como exibir informações de formas textuais e não textuais.

CAPÍTULO 15
Gerando dados

A *visualização de dados* é o uso de representações visuais para explorar e para evidenciar padrões em conjuntos de dados. Como está estreitamente associada à *análise de dados*, usa o código para explorar os padrões e as relações em um conjunto de dados. Um conjunto de dados pode ser uma pequena lista de números que se acomoda em uma única linha de código ou pode ser terabytes de dados com diversos tipos diferentes de informações.

Criar visualizações efetivas de dados envolve mais do que apresentar informações de maneira satisfatória. Quando a representação de um conjunto de dados é simples e visualmente agradável, todos conseguem entender claramente o que veem. As pessoas identificarão padrões e significância em seus conjuntos de dados que nem sabiam que existiam.

Felizmente, não precisamos de um supercomputador para visualizar dados complexos. O Python é tão eficiente que, basta um notebook, e podemos rapidamente explorar conjuntos de dados com milhões de pontos de dados individuais. Não necessariamente os pontos de dados são números; com as noções básicas que aprendeu na primeira parte deste livro, é possível também analisar dados não numéricos.

O Python é utilizado em áreas que exigem uso intensivo de dados (data-intensive) como genética, pesquisa climática, análise política e econômica e muito mais. Os cientistas de dados desenvolveram uma variedade extraordinária de ferramentas de visualização e análise em Python, muitas das quais estão disponíveis para quem quiser usá-la.

Uma das ferramentas mais populares é a Matplotlib, biblioteca para representação matemática de gráficos. Neste capítulo, usaremos a Matplotlib para plotar gráficos simples, como gráficos de linha e de dispersão. Em seguida, criaremos um conjunto de dados mais interessante com base no conceito de passeio aleatório (o famoso randow walk) – visualização gerada a partir de uma série de decisões aleatórias.

Usaremos também um pacote chamado Plotly, que cria visualizações que funcionam bem em dispositivos digitais, para analisar os resultados do lançamento de um dado. O Plotly gera visualizações que são redimensionadas automaticamente para caber em uma variedade de dispositivos de imagens. Essas visualizações também podem incluir uma série de características interativas, como enfatizar aspectos específicos do conjunto de dados quando os usuários passam o mouse sobre diferentes partes da visualização. Aprender a usar a Matplotlib e o Plotly o ajudará a começar a visualizar os tipos de dados em que você está mais interessado.

Instalando a Matplotlib

Para utilizarmos a Matplotlib em nosso conjunto inicial de visualizações, precisaremos instalá-la usando o pip, assim como fizemos com o pytest no Capítulo 11 (confira "Instalando o pytest com pip" na página *263*).

Para instalar a Matplotlib, digite o seguinte comando em um prompt de terminal:

```
$ python -m pip install --user matplotlib
```

Caso utilize um comando diferente de **python** para executar programas ou iniciar uma sessão de terminal, como **python3**, o comando será assim:

```
$ python3 -m pip install --user matplotlib
```

Para conferir os tipos de visualizações possíveis com a Matplotlib, visite a página inicial da Matplotlib em https://matplotlib.org e clique em

Plot types. Ao clicar em uma visualização na galeria, é possível ver o código usado para gerar o gráfico.

Plotando um simples gráfico de linhas

Vamos plotar um simples gráfico de linhas com a Matplotlib e vamos personalizá-lo para criar uma visualização de dados mais informativa. Utilizaremos a sequência de números elevados ao quadrado 1, 4, 9, 16 e 25 como dados para o gráfico.

Para criar um simples gráfico de linhas, especifique os números com os quais quer trabalhar que a Matplotlib se encarrega do restante:

mpl_squares.py

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]

1 fig, ax = plt.subplots()
ax.plot(squares)

plt.show()
```

Primeiro, importamos o módulo `pyplot` usando o alias `plt` para não precisarmos digitar `pyplot` repetidas vezes. (Como você verá essa convenção com frequência em exemplos online, vamos usá-la aqui.) O módulo `pyplot` contém uma série de funções que ajudam a gerar diagramas e gráficos.

Criamos uma lista chamada `squares` para armazenar os dados que plotaremos. Depois, adotamos outra convenção comum da Matplotlib chamando a função `subplots()` `1`. Essa função pode gerar um ou mais gráficos na mesma figura. A variável `fig` representa toda a *figura*, que é a coleção de gráficos gerados. A variável `ax` representa um único gráfico na figura; usaremos essa variável na maioria das vezes ao definir e personalizar um único gráfico.

Em seguida, usamos o método `plot()`, que tenta plotar os dados fornecidos de maneira relevante. A função `plt.show()` abre o

visualizador da Matplotlib e exibe o gráfico, como mostrado na Figura 15.1. O visualizador possibilita ampliar e navegar pelo gráfico, sendo possível salvar todas as imagens do gráfico que quisermos clicando no ícone do disquete.

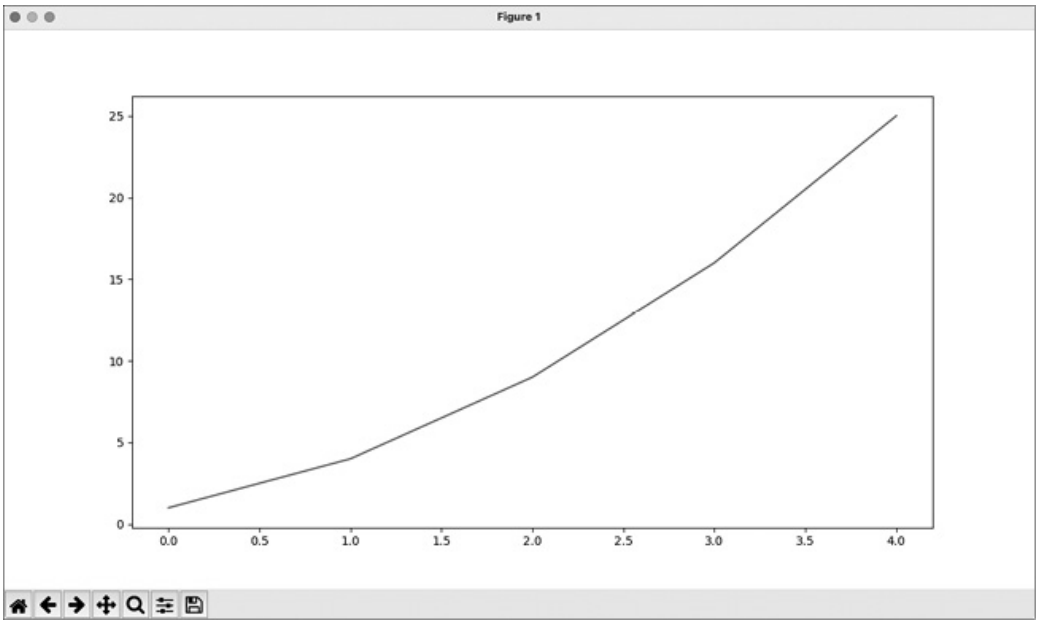


Figura 15.1: Um dos gráficos mais simples que podemos gerar com a Matplotlib.

Mudando o tipo de rótulo e a espessura da linha

Mesmo que o gráfico na Figura 15.1 mostre que os números estão aumentando, o tipo de rótulo é muito pequeno e a linha é um pouco fina para ser lida com facilidade. Felizmente, a Matplotlib possibilita ajustar todas as características de uma visualização.

Vamos utilizar algumas das personalizações disponíveis para melhorar a legibilidade deste gráfico. Começaremos adicionando um título e rotulando os eixos:

mpl_squares.py

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]

fig, ax = plt.subplots()
```

```
1 ax.plot(squares, linewidth=3)

  # Define o título do gráfico e os eixos do rótulo
2 ax.set_title("Square Numbers", fontsize=24)
3 ax.set_xlabel("Value", fontsize=14)
  ax.set_ylabel("Square of Value", fontsize=14)

  # Define o tamanho dos rótulos de marcação de escala
4 ax.tick_params(labelsize=14)

  plt.show()
```

O parâmetro linewidth controla a espessura da linha que plot() gera 1. Uma vez que um gráfico foi gerado, existem muitos métodos disponíveis para modificá-los antes de apresentá-lo. O método set_title() define um título geral para o gráfico 2. Os parâmetros fontsize, que aparecem repetidas vezes pelo código, controlam o tamanho do texto em diversos elementos do gráfico.

Os métodos set_xlabel() e set_ylabel() possibilitam definir um título para cada um dos eixos 3, e o método tick_params() estiliza as marcações 4. Aqui, tick_params() define o tamanho da fonte dos rótulos de marcação de escala para 14 em ambos os eixos.

Como podemos ver na Figura 15.2, o gráfico resultante é mais fácil de ler. O tipo de rótulo é maior, e as linhas do gráfico são mais espessas. Não raro, vale a pena testar esses valores para ver o que funciona melhor no gráfico resultante.

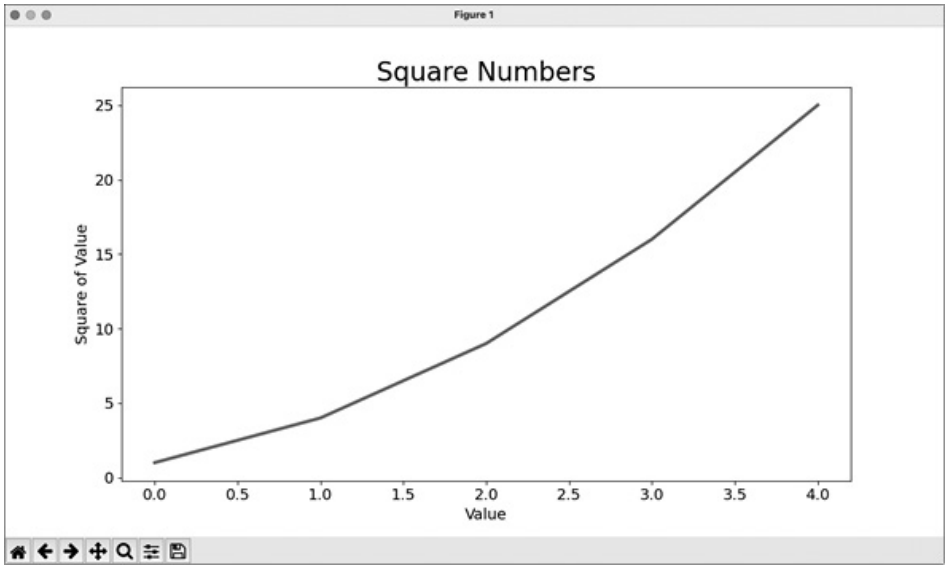


Figura 15.2: O gráfico fica bem mais fácil de ler agora.

Corrigindo o gráfico

Agora que conseguimos ler melhor o gráfico, podemos ver que os dados não estão devidamente plotados. Veja no final do gráfico que o quadrado de 4,0 é mostrado como 25! Vamos arrumar isso.

Ao fornecermos uma única sequência de números, o `plot()` faz a suposição de que o primeiro ponto de dados corresponde a um valor *x* de 0, só que nosso primeiro ponto corresponde a um valor *x* de 1. É possível sobrescrever o comportamento default fornecendo para `plot()` os valores de entrada e de saída usados para calcular os quadrados:

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

fig, ax = plt.subplots()
ax.plot(input_values, squares, linewidth=3)

# Define o título do gráfico e os eixos do rótulo
-- trecho de código omitido --
```

Agora plot() não precisa fazer nenhuma suposição sobre como os números de saída gerados. O gráfico resultante, mostrado na Figura 15.3, está correto.

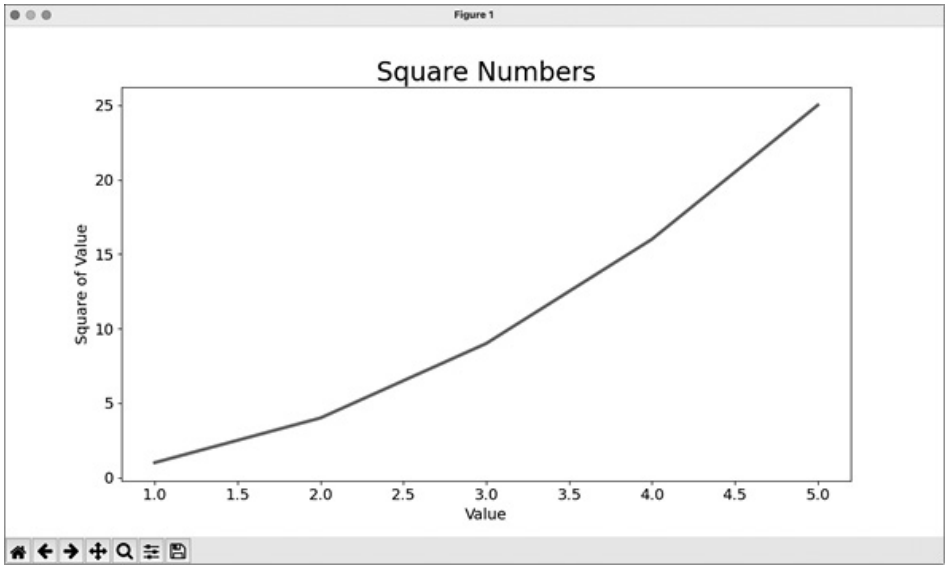


Figura 15.3: Os dados agora são devidamente plotados.

Podemos especificar determinado número de argumentos ao chamar plot() e usar inúmeros métodos para personalizar os gráficos depois de gerá-los. Vamos continuar explorando essas abordagens de personalização ao longo deste capítulo à medida que trabalhamos com conjuntos de dados mais interessantes.

Usando estilos built-in

A Matplotlib disponibiliza vários estilos predefinidos. Esses estilos têm uma variedade de configurações default para cores de background, linhas da grade, espessuras de linha, fontes, tamanhos de fonte e muito mais. Além disso, podem fazer com que as visualizações fiquem elegantes, sem exigir muita personalização. Para conferir a lista completa de estilos disponíveis, execute as seguintes linhas em uma sessão de terminal:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.available
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery',
-- trecho de código omitido --
```

Para usar qualquer um desses estilos, adicione uma linha de código antes de chamar subplots():

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
-- trecho de código omitido --
```

Esse código gera o gráfico mostrado na Figura 15.4. Já que uma grande variedade de estilos está disponível; brinque com esses estilos para encontrar alguns de que goste.

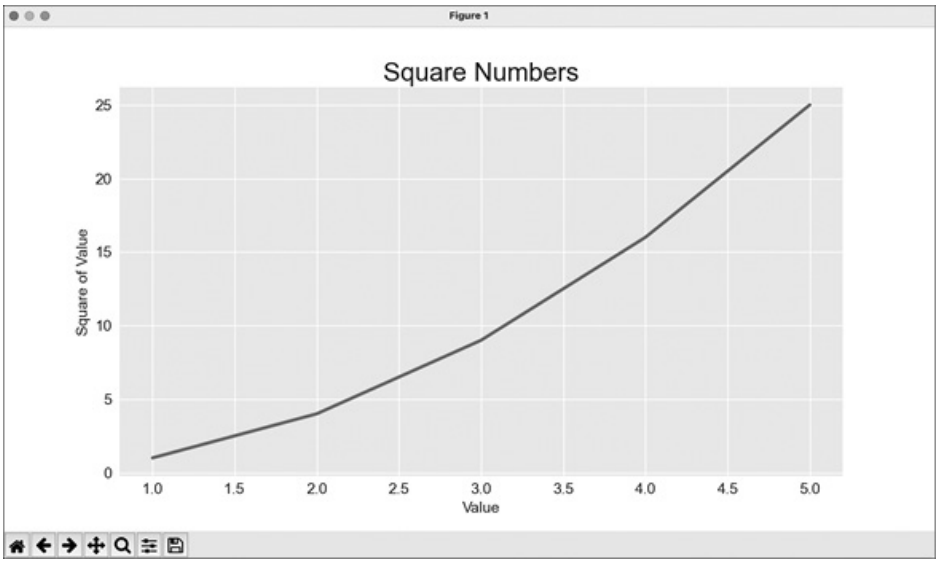


Figura 15.4: Estilo built-in seaborn.

Plotando e estilizando pontos individuais com o scatter()

Às vezes é útil plotar e estilizar pontos individuais com base em determinadas particularidades. Por exemplo, podemos plotar valores pequenos com uma cor e valores maiores com uma cor diferente. Podemos também plotar um grande conjunto de dados com um conjunto de opções de estilo e, em seguida, frisar os pontos

individuais, replotando-os com diferentes opções.

Para plotar um único ponto, passe os valores *x* e *y* do ponto para scatter():

scatter_squares.py

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(2, 4)

plt.show()
```

Vamos estilizar a saída para que fique mais relevante. Vamos adicionar um título, rotular os eixos e garantir que todo o texto seja grande o suficiente para ler:

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(2, 4, s=200)

# Define o título do gráfico e os eixos do rótulo
ax.set_title("Square Numbers", fontsize=24)
ax.set_xlabel("Value", fontsize=14)
ax.set_ylabel("Square of Value", fontsize=14)

# Define o tamanho dos rótulos de marcação de escala
ax.tick_params(labelsize=14)

plt.show()
```

Chamamos scatter() e usamos o argumentos para definir o tamanho dos pontos usados para desenhar o gráfico 1. Agora, quando executamos *scatter_squares.py*, veremos um único ponto no meio do gráfico, conforme mostrado na Figura 15.5.

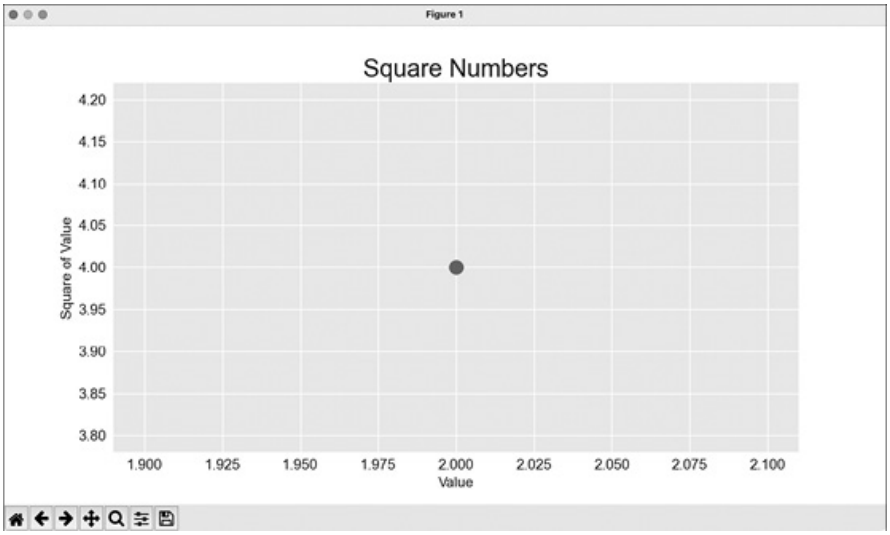


Figura 15.5: Plotando um único ponto.

Plotando uma série de pontos com scatter()

Para plotar uma série de pontos, podemos passar ao `scatter()` listas separadas de valores *x* e *y*, assim:

scatter_squares.py

```
import matplotlib.pyplot as plt

x_values = [1, 2, 3, 4, 5]
y_values = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, y_values, s=100)

# Define o título do gráfico e os eixos do rótulo
-- trecho de código omitido --
```

A lista `x_values` contém os números a serem elevados ao quadrado e `y_values` contém o quadrado de cada número. Quando essas listas são passadas para `scatter()`, a Matplotlib lê um valor de cada lista ao mesmo tempo em que plota cada ponto. Os pontos a serem plotados são (1, 1), (2, 4), (3, 9), (4, 16) e (5, 25); a Figura 15.6 mostra o resultado.

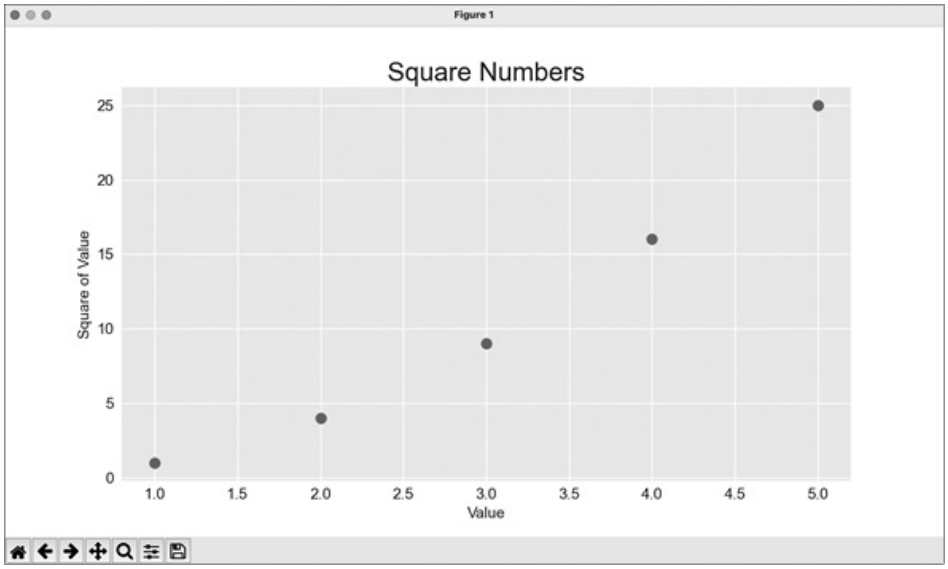


Figura 15.6: Um gráfico de dispersão com diversos pontos.

Calculando automaticamente os dados

Escrever listas manualmente pode ser ineficaz, ainda mais quando temos muitos pontos. Em vez de escrever cada valor, usaremos um loop para fazer os cálculos para nós.

Vejamos como seria um código com 1.000 pontos:

scatter_squares.py

```
  import matplotlib.pyplot as plt

1 x_values = range(1, 1001)
  y_values = [x**2 for x in x_values]

  plt.style.use('seaborn')
  fig, ax = plt.subplots()
2 ax.scatter(x_values, y_values, s=10)

  # Define o título do gráfico e os eixos do rótulo
  -- trecho de código omitido --

  # Define o intervalo para cada eixo
3 ax.axis([0, 1100, 0, 1_100_000])

  plt.show()
```

Começamos com um intervalo de valores *x* contendo os números de

1 a 1.000 1. Em seguida, uma list comprehension gera os valores *y* percorrendo os valores *x* com um loop (for x in x_values), elevando ao quadrado cada número (x**2) e atribuindo os resultados a y_values. Depois, passamos as listas de entrada e de saída para scatter() 2. Como se trata de um conjunto grande de dados, usamos um tamanho de ponto menor.

Antes de mostrar o gráfico, usamos o método axis() para especificar o intervalo de cada eixo 3. O método axis() exige quatro valores: os valores mínimo e máximo para o eixo *x* e para o eixo *y.* Aqui, executamos o eixo *x* de 0 a 1.100 e o eixo *y* de 0 a 1.100.000. A Figura 15.7 mostra o resultado.

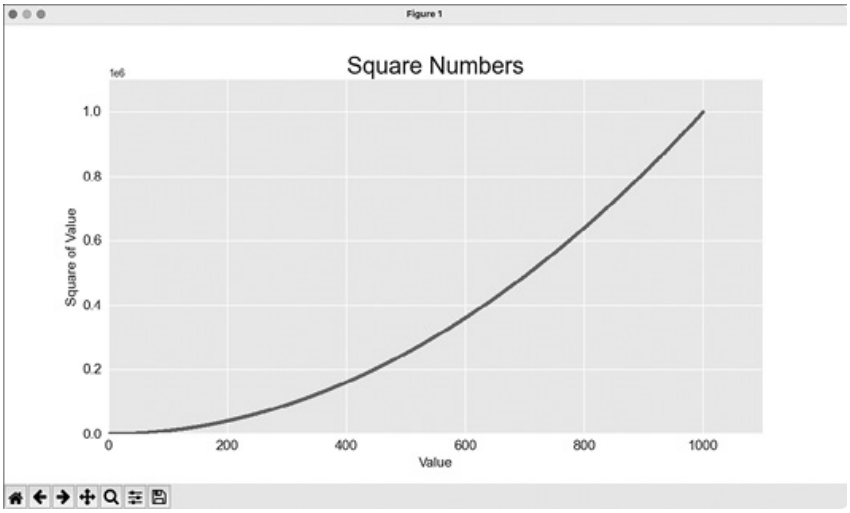


Figura 15.7: O Python pode plotar 1.000 pontos com a mesma facilidade com que plota 5 pontos.

Personalizando a marcação de escala dos rótulos

Quando os números em um eixo ficam grandes o bastante, o padrão da Matplotlib é a notação científica para escala dos rótulos. Em geral, é uma coisa boa, porque números maiores em notação simples ocupam muito espaço desnecessário em uma visualização.

Como quase todos os elementos de um gráfico são personalizáveis, podemos solicitar que a Matplotlib continue usando notação simples, se assim preferirmos.

```
-- trecho de código omitido --
# Define o intervalo para cada eixo
ax.axis([0, 1100, 0, 1_100_000])
ax.ticklabel_format(style='plain')

plt.show()
```

O método ticklabel_format() possibilita sobrescrever o estilo default de marcação de escala dos rótulos para qualquer gráfico.

Definindo cores personalizadas

Para alterar a cor dos pontos, passe o argumento color para scatter() com o nome da cor a ser usada entre aspas, como mostrado aqui:

```
ax.scatter(x_values, y_values, color='red', s=10)
```

É possível também definir cores personalizadas usando o modelo de cores RGB. Para definir uma cor, passe o argumento color para uma tupla contendo três valores floats (um para vermelho, verde e azul, nessa ordem), usando valores entre 0 e 1. Por exemplo, a linha a seguir cria um gráfico com pontos verde-claros:

```
ax.scatter(x_values, y_values, color=(0, 0.8, 0), s=10)
```

Valores mais próximos de 0 geram cores mais escuras e valores mais próximos de 1 geram cores mais claras.

Usando um colormap

Um *colormap* é uma sequência de cores em um gradiente que oscila de uma cor inicial para uma cor final. Nas visualizações, os colormaps são usados para destacar padrões nos dados. Por exemplo, podemos transformar valores baixos em uma cor clara e valores altos em uma cor mais escura. O uso de um colormap garante que todos os pontos na visualização variem de forma suave e com acurácia ao longo de uma escala de cores bem estruturada.

O módulo pyplot inclui um conjunto buit-in de colormaps. Para utilizar um desses colormaps, precisamos especificar como o pyplot deve atribuir uma cor a cada ponto no conjunto de dados. Veja como atribuir uma cor a cada ponto, com base em seu valor *y*:

scatter_squares.py

```
-- trecho de código omitido --
plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, y_values, c=y_values, cmap=plt.cm.Blues, s=10)

# Define o título do gráfico e os eixos do rótulo
-- trecho de código omitido --
```

O argumento `c` é semelhante a `color`, mas é usado para associar uma sequência de valores a um colormap. Passamos a lista de valores *y* para `c` e, em seguida, instruímos o `pyplot` qual colormap usar com o argumento `cmap`. Esse código colore os pontos de valores menores de *y* com azul claro e os pontos de valores maiores de *y* com azul escuro. A Figura 15.8 mostra o gráfico resultante.

NOTA *Confira todos os colormaps disponíveis de* `pyplot` *em* https://matplotlib.org. *Acesse Tutorials, role para baixo até Colors e clique em* **Choosing Colormaps in Matplotlib***.*

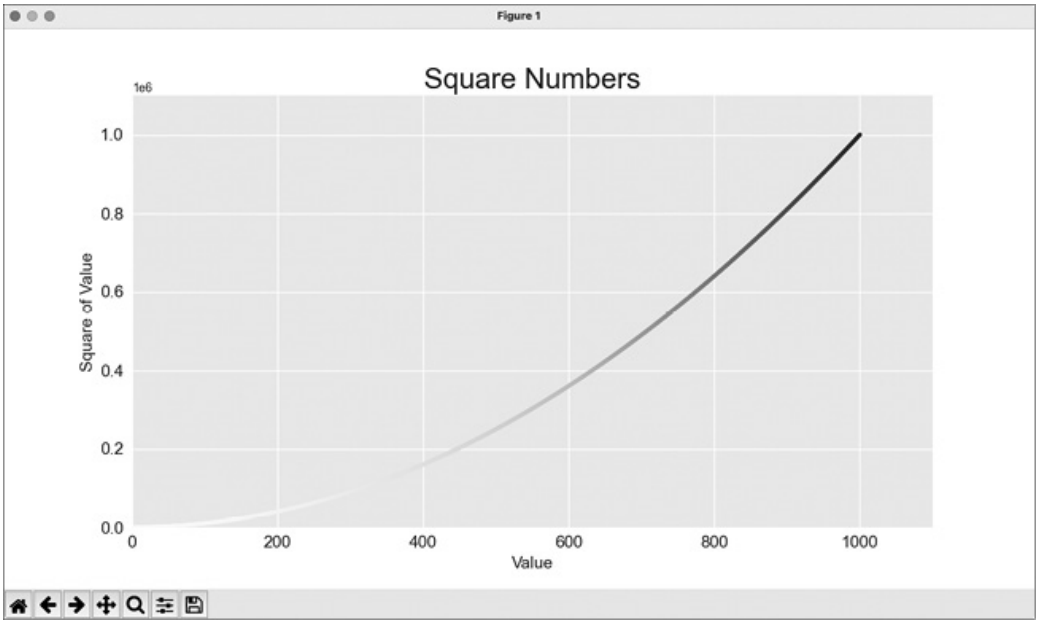


Figura 15.8: Um gráfico usando o colormap Blues.

Salvando automaticamente seus gráficos

Caso queira salvar o gráfico em um arquivo em vez de mostrá-lo no

visualizador da Matplotlib, é possível usar plt.savefig() em vez de plt.show():

```
plt.savefig('squares_plot.png', bbox_inches='tight')
```

O primeiro argumento é um nome de arquivo para a imagem do gráfico, que será salva no mesmo diretório que *scatter_squares.py*. O segundo argumento remove o espaço em branco extra do gráfico. Caso opte pelo espaço em branco extra em torno do gráfico, é possível omitir esse argumento. É possível também chamar savefig() com um objeto Path, e escrever o arquivo de saída em qualquer lugar que queira em seu sistema.

FAÇA VOCÊ MESMO

15.1 Cubos: Um número elevado à terceira potência se chama *cubo*. Plote os primeiros cinco números cúbicos e, em seguida, plote os primeiros 5.000 números cúbicos.

15.2 Cubos coloridos: Use um colormap ao seu gráfico de cubos.

Passeios aleatórios

Nesta seção, utilizaremos o Python para gerar dados a partir de um passeio aleatório e, em seguida, usaremos a Matplotlib para criar uma representação visualmente elegante desses dados. Um *passeio aleatório* é um caminho determinado por uma série de decisões simples, e cada uma delas é deixada inteiramente ao acaso. Imagine um passeio aleatório como o caminho que uma formiga confusa tomaria se desse cada passo em uma direção aleatória.

Passeios aleatórios apresentam usos práticos na natureza, na física, biologia, química e economia. Por exemplo, um grão de pólen que flutua sobre uma gota de água vai de um lado para o outro pela superfície aquosa porque é constantemente empurrado por moléculas de água. Já que movimento molecular em uma gota de água é aleatório, o caminho que um grão de pólen traça na superfície é um passeio aleatório. O código que desenvolveremos a seguir modela muitas situações do mundo cotidiano.

Criando a classe RandomWalk

Para criarmos um passeio aleatório, primeiro criaremos uma classe RandomWalk, que tomará decisões aleatórias sobre qual direção o passeio deve tomar. A classe deve receber três atributos: uma variável para rastrear o número de pontos no passeio e duas listas para armazenar as coordenadas *x* e *y* de cada ponto no passeio.

Precisaremos somente de dois métodos para a classe RandomWalk: o método __init__() e fill_walk(), que calculará os pontos no passeio. Começaremos com o método __init__():

random_walk.py

```
 1 from random import choice

   class RandomWalk:
       """Classe para gerar passeios aleatórios"""

 2     def __init__(self, num_points=5000):
           """Inicializa atributos de um passeio"""
           self.num_points = num_points

           # Todos os passeios começam em (0, 0)
 3         self.x_values = [0]
           self.y_values = [0]
```

A fim de tomar decisões aleatórias, vamos armazenar possíveis movimentos em uma lista e usar a função choice() (do módulo random) para decidir qual movimento fazer sempre que um passo é dado 1. Definimos o número default de pontos em um passeio como 5000, valor grande o bastante para gerar alguns padrões interessantes, mas pequeno o suficiente para gerar rapidamente os passeios 2. Depois, criamos duas listas para armazenar os valores de *x* e de *y*, e começamos cada passeio no ponto (0, 0) 3.

Escolhendo direções

Vamos utilizar o método fill_walk() para determinar a sequência completa de pontos no passeio. Adicione esse método a *random_walk.py*:

random_walk.py

```
    def fill_walk(self):
        """Calcula todos os pontos do passeio"""

        # Continua dando passos até que o passeio atinja o comprimento desejado
1       while len(self.x_values) < self.num_points:

            # Decide qual direção tomar, e até onde ir
2           x_direction = choice([1, -1])
            x_distance = choice([0, 1, 2, 3, 4])
3           x_step = x_direction * x_distance

            y_direction = choice([1, -1])
            y_distance = choice([0, 1, 2, 3, 4])
4           y_step = y_direction * y_distance

            # Rejeita movimentos que não vão a lugar algum
5           if x_step == 0 and y_step == 0:
                continue

            # Calcula a nova posição
6           x = self.x_values[-1] + x_step
            y = self.y_values[-1] + y_step

            self.x_values.append(x)
            self.y_values.append(y)
```

Primeiro, definimos um loop que é executado até que o passeio seja preenchido com o número correto de pontos 1. A parte principal de fill_walk() orienta o Python como simular quatro decisões aleatórias: Será que o passeio vai para a direita ou para a esquerda? Qual é a distância que o passeio percorrerá nessa direção? O passeio se deslocará para cima ou para baixo? Qual é a distância que o passeio percorrerá nessa direção.

Usamos choice([1, -1]) a fim de escolher um valor para x_direction, que retorna 1 para movimento à direita ou -1 para movimento à esquerda 2. Depois, choice([0, 1, 2, 3, 4]) seleciona aleatoriamente uma distância a ser percorrida em determinada direção. Atribuímos esse valor a x_distance. Incluir um 0 possibilita passos com movimento ao longo de apenas um eixo.

Determinamos a extensão de cada passo nas direções *x* e *y* multiplicando a direção do movimento pela distância escolhida 3 4. Um resultado positivo para x_step significa movimento à direita, um resultado negativo significa movimento à esquerda e 0 significa movimento vertical. Um resultado positivo para y_step significa movimento para cima, negativo significa movimento para baixo e 0 significa movimento horizontal. Se os valores de x_step e y_step forem 0, o passeio não se movimenta; quando isso acontece, continuamos interando com o loop 5.

A fim de obtermos o próximo valor *x* para o passeio, somamos o valor em x_step ao último valor armazenado em x_values 6 e fazemos o mesmo com os valores *y.* Ao termos as coordenadas do novo ponto, as anexamos a x_values e a y_values.

Plotando o passeio aleatório

Vejamos o código para plotar todos os pontos do passeio:

rw_visual.py

```
  import matplotlib.pyplot as plt

  from random_walk import RandomWalk

  # Cria um random walk
1 rw = RandomWalk()
  rw.fill_walk()

  # Plota os pontos no passeio
  plt.style.use('classic')
  fig, ax = plt.subplots()
2 ax.scatter(rw.x_values, rw.y_values, s=15)
3 ax.set_aspect('equal')
  plt.show()
```

Começamos importando Pyplot e RandomWalk. Em seguida, criamos um passeio aleatório e o atribuímos a rw 1, não esquecendo de chamar fill_walk(). Para visualizar o passeio, fornecemos os valores *x* e *y* dele para scatter() e escolhemos um tamanho adequado de ponto 2. Por padrão, a Matplotlib dimensiona cada eixo de forma independente.

Mas essa abordagem alongaria vertical ou horizontalmente a maioria dos passeios. Aqui, utilizamos o método set_aspect() para especificar que ambos os eixos devem ter espaçamento igual entre as marcas de escala 3.

A Figura 15.9 mostra o gráfico resultante com 5.000 pontos. Nesta seção, as imagens omitem o visualizador da Matplotlib, mas você conseguirá vê-lo quando executar *rw_visual.py*.

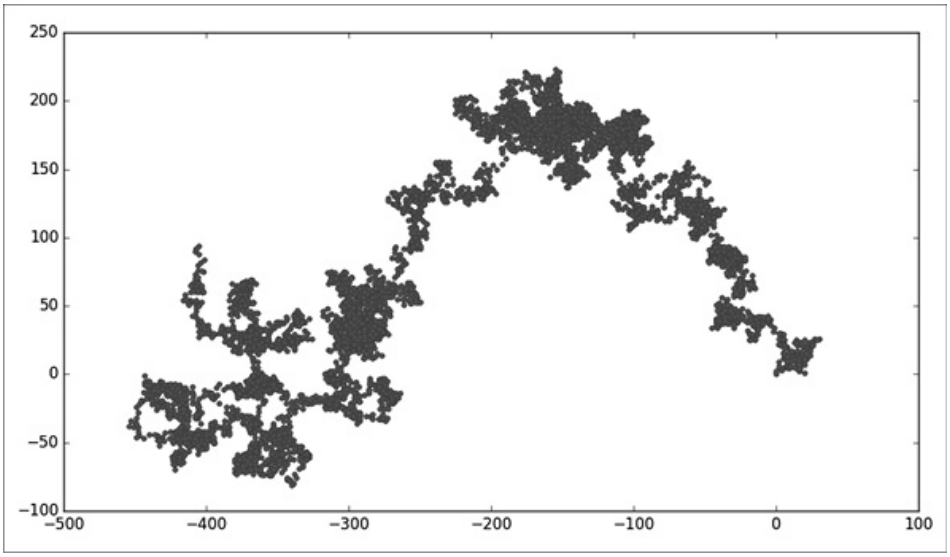


Figura 15.9: Passeio aleatório com 5.000 pontos.

Gerando múltiplos passeios aleatórios

Já que todo passeio aleatório é diferente, é divertido explorar os diversos padrões que podem ser gerados. Uma forma de usar o código anterior para criar múltiplos passeios aleatórios sem precisarmos executar o programa repetidas vezes é envolvê-lo em um loop while, assim:

rw_visual.py

```
import matplotlib.pyplot as plt

from random_walk import RandomWalk

# Continua criando passeios novos, desde que o programa esteja ativo
while True:
    # Cria um random walk
```

```
-- trecho de código omitido --
plt.show()

keep_running = input("Make another walk? (y/n): ")
if keep_running == 'n':
    break
```

Além de gerar um passeio aleatório, esse código o exibe no visualizador da Matplotlib e faz uma pausa com o visualizador aberto. Ao fechar o visualizador, seremos questionados se queremos gerar outro passeio. Caso gere alguns passeios, verá que alguns deles ficam perto do ponto inicial, outros passeiam majoritariamente em uma direção, alguns têm seções estreitas conectando grupos maiores de pontos e muitos outros tipos de passeios. Se quisermos encerrar o programa, basta pressionarmos N.

Estilizando o passeio

Nesta seção, vamos personalizar nossos gráficos a fim de destacar as particularidades importantes de cada passeio e reduzir o destaque dos elementos que podem ocasionar distração. Para tanto, identificamos as particularidades que queremos enfatizar, como onde o passeio começou e terminou, e o caminho percorrido. Depois, identificamos as particularidades a serem enfatizadas, como marcações de escala e rótulos. O resultado deve ser uma representação visual simples que transmita claramente o caminho percorrido em cada passeio aleatório.

Colorindo os pontos

Vamos utilizar um colormap para mostrar a ordem dos pontos no passeio e vamos remover o contorno preto de cada ponto para que a cor dos pontos fique mais nítida. Para colorir os pontos conforme sua posição no passeio, passamos ao argumento `c` uma lista contendo a posição de cada ponto. Como os pontos são plotados em ordem, a lista contém apenas os números de 0 a 4.999:

rw_visual.py

```
-- trecho de código omitido --
while True:
    # Cria um random walk
    rw = RandomWalk()
    rw.fill_walk()

    # Plota os pontos no passeio
    plt.style.use('classic')
    fig, ax = plt.subplots()
1   point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolors='none', s=15)
    ax.set_aspect('equal')
    plt.show()
    -- trecho de código omitido --
```

Usamos range() para gerar uma lista de números igual ao número de pontos no passeio 1. Atribuímos essa lista a point_numbers, que usaremos para definir a cor de cada ponto no passeio. Passamos point_numbers para o argumento c, utilizamos o colormap Blues e, em seguida, passamos edgecolors ='none' para descartarmos o contorno preto em torno de cada ponto. O resultado é um gráfico que varia do azul claro ao azul escuro, mostrando exatamente como o passeio se desloca do seu ponto inicial ao seu ponto final. A Figura 15.10 mostra o passeio.

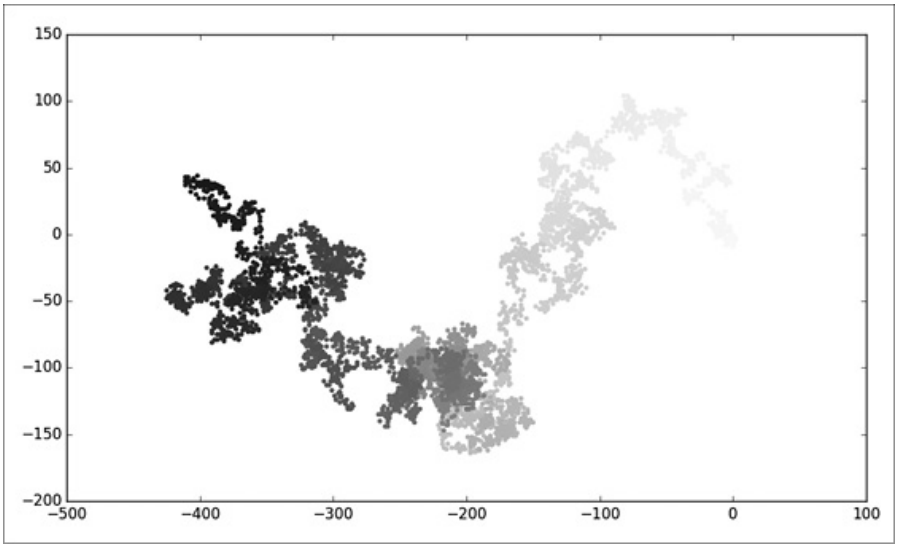


Figura 15.10: Um passeio aleatório colorido com o colormap Blues.

Plotando os pontos iniciais e finais

Além de colorir os pontos para evidenciar sua posição ao longo do passeio, ajudaria bastante ver exatamente onde cada passeio começa e termina. Para isso, é possível plotar o primeiro e o último pontos individualmente, após a série principal ser plotada. Vamos aumentar os pontos finais e vamos colori-los de forma diferente para que se destaquem:

rw_visual.py

```
-- trecho de código omitido --
while True:
    -- trecho de código omitido --
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolors='none', s=15)
    ax.set_aspect('equal')

    # Destaca o primeiro e o último ponto
    ax.scatter(0, 0, c='green', edgecolors='none', s=100)
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
        s=100)

    plt.show()
    -- trecho de código omitido --
```

A fim de mostrar o ponto inicial, plotamos o ponto (0, 0) em verde e em um tamanho maior (`s=100`) do que o restante dos pontos. Para destacar o ponto final, plotamos os últimos valores *x* e *y* em vermelho com um tamanho de 100 também. Não se esqueça de inserir esse código logo antes da chamada para `plt.show()`, pois, assim, os pontos inicial e final são desenhados em cima de todos os outros pontos.

Ao executarmos esse código, conseguimos identificar exatamente onde cada passeio começa e termina. Se os pontos finais não se destacarem de modo evidente, ajuste a cor e o tamanho até se destacarem.

Limpando os eixos

Removeremos os eixos desse gráfico para que não causem distração

do caminho de cada passeio. Vejamos como ocultar os eixos:

rw_visual.py

```
-- trecho de código omitido --
while True:
    -- trecho de código omitido --
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
        s=100)

    # Remove os eixos
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    plt.show()
    -- trecho de código omitido --
```

Para modificar os eixos, usamos os métodos ax.get_xaxis() e ax.get_yaxis() a fim de obter cada eixo e, em seguida, encadeamos o método set_visible() para que cada eixo ficasse invisível. À medida que trabalha cada vez mais com visualizações, você verá muito esse tipo de encadeamento de métodos para personalizar diferentes aspectos de uma visualização.

Agora, execute *rw_visual.py*; você deve ver uma série de gráficos sem eixos.

Adicionando pontos ao gráfico

Aumentaremos o número de pontos a fim de termos mais dados para trabalhar. Para isso, aumentamos o valor de num_points quando criamos uma instância RandomWalk e ajustamos o tamanho de cada ponto ao desenhar o gráfico:

rw_visual.py

```
-- trecho de código omitido --
while True:
    # Cria um random walk
    rw = RandomWalk(50_000)
    rw.fill_walk()
    # Plota os pontos no passeio
    plt.style.use('classic')
    fig, ax = plt.subplots()
```

```
point_numbers = range(rw.num_points)
ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
    edgecolors='none', s=1)
-- trecho de código omitido --
```

Nesse exemplo, criamos um passeio aleatório com 50.000 pontos e plotamos cada ponto com o tamanho `s=1`. O passeio resultante se parece com fumaça ou com uma nuvem, como mostrado na Figura 15.11. Desenhamos uma obra de arte a partir de um simples gráfico de dispersão!

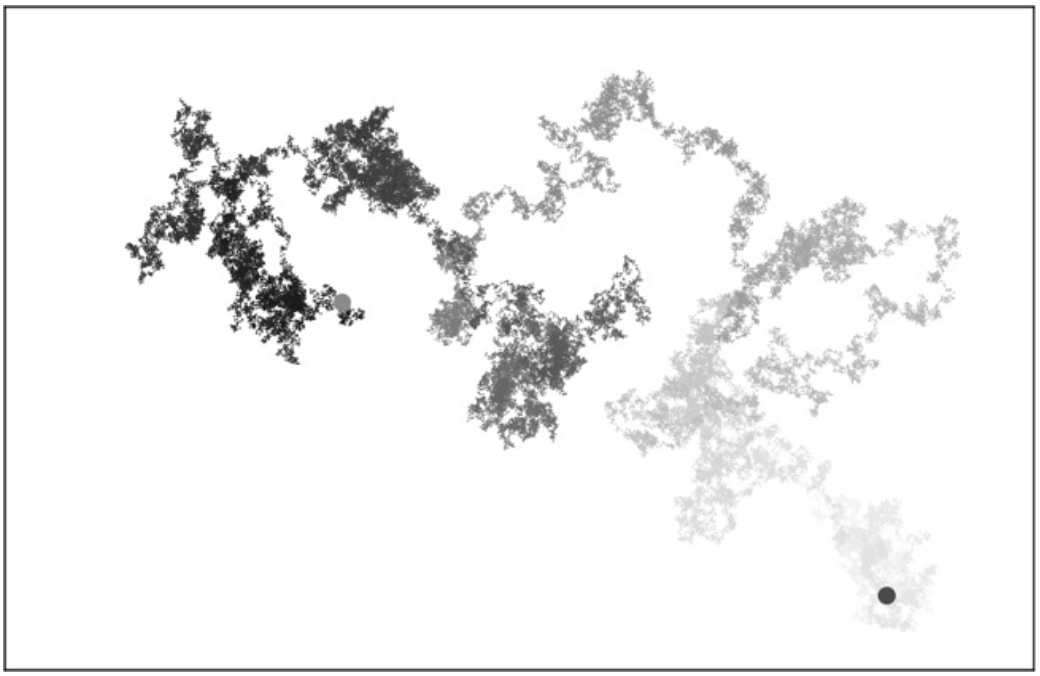


Figura 15.11: Um passeio com 50.000 pontos.

Teste esse código para ver o quanto consegue aumentar o número de pontos em um passeio antes que seu sistema comece a desacelerar significativamente ou o gráfico perca a elegância visual.

Alterando o tamanho para preencher a tela

Uma visualização transmite efetivamente os padrões dos dados se preencher a tela de forma satisfatória. Para fazer com que a janela de plotagem se ajuste melhor à tela, podemos ajustar o tamanho da saída da Matplotlib. Isso é feito na chamada `subplots()`:

```
fig, ax = plt.subplots(figsize=(15, 9))
```

Ao criar um gráfico, podemos passar subplots() a um argumento figsize, que define o tamanho da figura. O parâmetro figsize recebe uma tupla que informa à Matplotlib as dimensões da janela de plotagem em polegadas.

A Matplotlib faz a suposição de que a resolução de tela é de 100 pixels por polegada; caso esse código não forneça um tamanho de gráfico acurado, ajuste os números conforme necessário. Ou, caso saiba a resolução do seu sistema, passe subplots() à resolução usando o parâmetro dpi:

```
fig, ax = plt.subplots(figsize=(10, 6), dpi=128)
```

Isso deve ajudar a usar de forma mais eficiente do espaço disponível em sua tela.

FAÇA VOCÊ MESMO

15.3 Movimento molecular: Modifique *rw_visual.py* substituindo ax.scatter() por ax.plot(). Para simular o caminho de um grão de pólen na superfície de uma gota de água, passe rw.x_values e rw.y_values e inclua um argumento linewidth. Use 5.000 em vez de 50.000 pontos, assim o gráfico não fica muito saturado.

15.4 Passeios aleatórios modificados: Na classe RandomWalk, x_step e y_step são gerados a partir do mesmo conjunto de condições. A direção é escolhida aleatoriamente a partir da lista [1, -1] e a distância a partir da lista [0, 1, 2, 3, 4]. Modifique os valores dessas listas para ver o que acontece com o formato geral de seu passeio. Teste uma lista mais extensa de opções para a distância, como 0 a 8, ou remova o -1 da lista de direção *x* ou *y.*

15.5 Refatoração: O método fill_walk() é muito longo. Crie um método novo chamado get_step() a fim de determinar a direção e a distância de cada passo e, em seguida, calcule o passo. Você deve terminar com duas chamadas para get_step() em fill_walk():

```
x_step = self.get_step()
y_step = self.get_step()
```

Essa refatoração deve reduzir o tamanho de fill_walk() e facilitar a leitura e o entendimento do método.

Lançando dados com o Plotly

Nesta seção, usaremos o Plotly para gerar visualizações interativas. O Plotly é bastante útil quando estamos criando visualizações que serão exibidas em um navegador, já que as visualizações serão

dimensionadas automaticamente para caber na tela do visualizador. Além disso, essas visualizações são interativas; quando o usuário passa o mouse sobre determinados elementos na tela, as informações sobre esses elementos são destacadas. Criamos nossa visualização inicial com somente algumas linhas de código usando o *Plotly Express*, um subconjunto do Plotly cujo foco é gerar gráficos com o mínimo de código possível. Assim que soubermos que nosso gráfico está correto, personalizaremos a saída como fizemos com a Matplotlib.

Neste projeto, vamos analisar os resultados do lançamento de dados. Se lançarmos um dado normal com seis lados, temos a chance igual (probabilidade igual) de obter qualquer um dos números de 1 a 6. Apesar disso, quando usamos dois dados, é mais provável que possamos tirar determinados números do que outros. Vamos tentar determinar quais números têm a maior probabilidade de ocorrer gerando um conjunto de dados que represente o lançamento de dados. Em seguida, plotaremos os resultados de um grande número de lançamentos a fim de determinar quais resultados são mais prováveis do que outros.

Essa tarefa ajuda a modelar jogos envolvendo lançamento de dados, mas a ideia principal também pode ser usada em jogos que envolvem probabilidades de qualquer tipo, como jogos de cartas. Relaciona-se também com muitas situações cotidianas em que a aleatoriedade desempenha fator significativo.

Instalando o Plotly

Instale Plotly usando pip, assim como fizemos com a Matplotlib:

```
$ python -m pip install --user plotly
$ python -m pip install --user pandas
```

O Plotly Express depende do *pandas*, uma biblioteca para trabalhar de forma eficiente com dados. Ou seja, é necessário instalar o pandas também. Se usou **python3** ou outra coisa ao instalar a Matplotlib, lembre-se de usar o mesmo comando aqui.

Para conferir quais tipos de visualizações são possíveis com o Plotly, visite a galeria de tipos de gráficos em https://plotly.com/python. Cada exemplo tem um código-fonte, assim você pode ver como Plotly gera as visualizações.

Criando a classe Die

Criaremos a seguinte classe Die para simular o lançamento de um dado:

die.py

```
from random import randint

class Die:
    """Classe que representa um único dado"""

1   def __init__(self, num_sides=6):
        """ Faz a suposição de que um dado tem seis lados"""
        self.num_sides = num_sides

    def roll(self):
        """Retorna um valor aleatório entre 1 e o número de lados"""
2       return randint(1, self.num_sides)
```

O método __init__() recebe um argumento opcional 1. Com a classe Die, quando uma instância de nosso dado é criada, o número de lados será seis, se não incluirmos nenhum argumento. Se incluirmos *um* argumento, esse valor definirá o número de lados no dado. (Os dados são nomeados pelo número de lados: um dado de seis lados é um D6, um dado de oito lados é um D8, e assim por diante.)

O método roll() usa a função randint() para retornar um número aleatório entre 1 e o número de lados 2. Essa função pode retornar o valor inicial (1), o valor final (num_sides) ou qualquer número inteiro entre os dois.

Lançando o dado

Antes de criarmos uma visualização com base na classe Die, lançaremos um D6, exibiremos os resultados e verificaremos se os

resultados parecem razoáveis:

die_visual.py

```
from die import Die

# Cria um D6
die = Die()

# Realiza alguns testes e armazena os resultados em uma lista
results = []
for roll_num in range(100):
    result = die.roll()
    results.append(result)

print(results)
```

Criamos uma instância de `Die` com os seis lados como default `1`. Em seguida, lançamos o dado `100` vezes `2` e armazenamos o resultado de cada lançamento na lista `results`. Vejamos um exemplo do conjunto de resultados:

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

Ao verificarmos rapidamente esses resultados, podemos ver que a classe `Die` parece estar funcionando. Como estamos vendo os valores 1 e 6, sabemos que os possíveis valores menores e maiores estão sendo retornados, e como não vemos 0 ou 7, sabemos que todos os resultados estão no intervalo adequado. Vemos também cada número de 1 a 6, o que sinaliza que todos os resultados possíveis estão representados. Determinaremos exatamente quantas vezes cada número aparece.

Analisando os resultados

Vamos analisar os resultados do lançamento de um D6 contando quantas vezes tiramos cada número:

die_visual.py

```
-- trecho de código omitido --
# Realiza alguns testes e armazena os resultados em uma lista
results = []
for roll_num in range(1000):
    result = die.roll()
    results.append(result)

# Analisa os resultados
frequencies = []
poss_results = range(1, die.num_sides+1)
for value in poss_results:
    frequency = results.count(value)
    frequencies.append(frequency)

print(frequencies)
```

Como não estamos mais exibindo os resultados, é possível aumentar o número de lançamentos simulados para `1000` `1`. A fim de analisarmos os lançamentos, criamos uma lista vazia `frequencies` para armazenar o número de vezes que cada valor é tirado. Em seguida, geramos todos os resultados possíveis que poderíamos obter; nesse exemplo, são todos os números a partir de `1` até quantos lados `die` tiver `2`. Percorremos com um loop os valores possíveis, contamos quantas vezes cada número aparece em `results` `3` e, em seguida, anexamos esse valor à `frequencies` `4`. Vamos exibir essa lista antes de criarmos uma visualização:

```
[155, 167, 168, 170, 159, 181]
```

Ao que tudo indica, os resultados são razoáveis: vemos seis frequências, uma para cada número possível quando lançamos um D6. Vemos também que nenhuma frequência é significativamente maior do que qualquer outra. Agora, visualizaremos esses resultados.

Criando um histograma

Agora que temos os dados que queremos, podemos gerar uma visualização com apenas algumas linhas de código usando o Plotly Express:

die_visual.py

```
import plotly.express as px

from die import Die
-- trecho de código omitido --

for value in poss_results:
    frequency = results.count(value)
    frequencies.append(frequency)

# Visualiza os resultados
fig = px.bar(x=poss_results, y=frequencies)
fig.show()
```

Primeiro, importamos o módulo `plotly.express`, usando o alias convencional `px`. Depois, usamos a função `px.bar()` para criar um gráfico de barras. No uso mais simples dessa função, precisamos somente passar um conjunto de valores *x* e um conjunto de valores *y.* Aqui, os valores *x* são os resultados possíveis do lançamento de um único dado, e os valores *y* são as frequências para cada resultado possível.

A linha final chama `fig.show()`, que informa ao Plotly para renderizar o gráfico resultante como um arquivo HTML e abrir esse arquivo em uma nova guia do navegador. O resultado é mostrado na Figura 15.12.

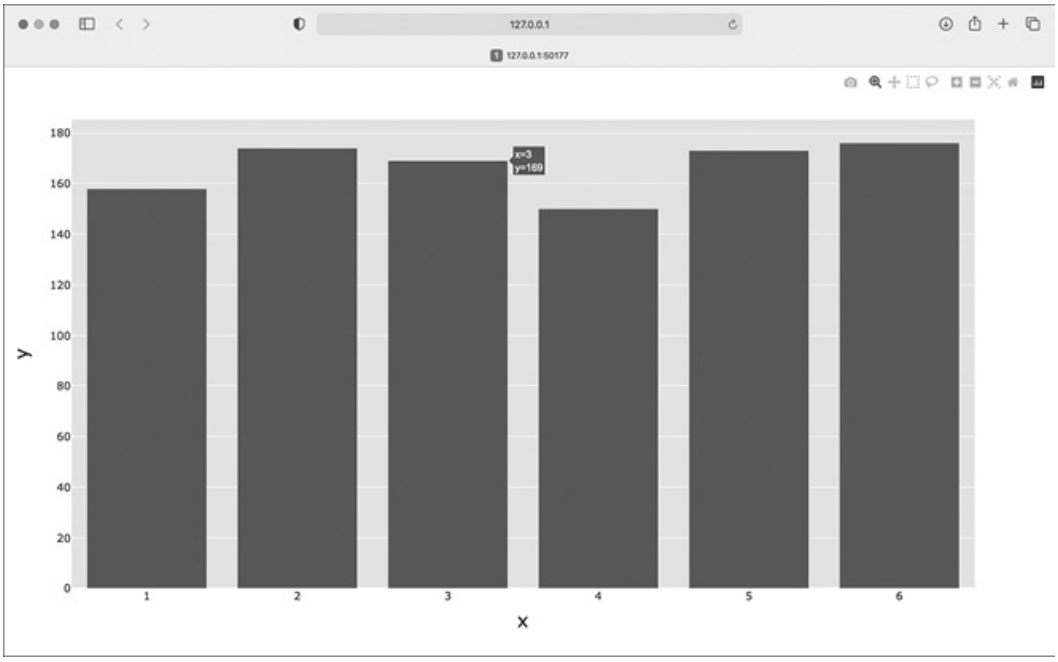


Figura 15.12: Gráfico inicia gerado pelo Plotly Express.

É um gráfico muito simples e, com certeza, não está completo. Mas é exatamente assim que o Plotly Express deve ser usado; basta escrever algumas linhas de código, verificar o visual do gráfico e garantir que represente os dados da maneira que queremos. Se gostar do que está vendo, pode passar para a personalização de elementos do gráfico, como rótulos e estilos. Mas se quiser explorar outros tipos possíveis de gráfico, pode seguir em frente agora, sem passar tempo demais com a personalização. Sinta-se à vontade para testar as coisas, alterando px.bar() para algo como px.scatter() ou px.line(). É possível encontrar uma lista completa dos tipos de gráficos disponíveis em https://plotly.com/python/plotly-express.

Esse gráfico é dinâmico e interativo. Se alterarmos o tamanho da janela do navegador, o gráfico será redimensionado para corresponder ao espaço disponível. Se passarmos o mouse sobre qualquer uma das barras, veremos um pop-up destacando os dados específicos relacionados a essa barra.

Personalizando o o gráfico

Agora que sabemos que temos o tipo correto de gráfico e nossos

dados estão sendo representados com acurácia, podemos nos concentrar em adicionar os rótulos e os estilos adequados para o gráfico.

A primeira forma de personalizar um gráfico com o Plotly é usar alguns parâmetros opcionais na chamada inicial que gera o gráfico, nesse caso, `px.bar()`. Vejamos como adicionar um título geral e um rótulo para cada eixo:

die_visual.py

```
  -- trecho de código omitido --
  # Visualiza os resultados
1 title = "Results of Rolling One D6 1,000 Times"
2 labels = {'x': 'Result', 'y': 'Frequency of Result'}
  fig = px.bar(x=poss_results, y=frequencies, title=title, labels=labels)
  fig.show()
```

De início, definimos o título que queremos, aqui atribuído a `title` 1. Para definir os rótulos dos eixos, escrevemos um dicionário 2. As chaves do dicionário se referem aos rótulos que queremos personalizar, e os valores são os rótulos personalizados que queremos usar. Aqui, fornecemos ao eixo *x* o rótulo `Result` e ao eixo *y* o rótulo `Frequency of Result`. Agora, a chamada para `px.bar()` inclui os argumentos opcionais `title` e `labels`.

Então, quando é gerado, o gráfico inclui um título e um rótulo adequados para cada eixo, como mostrado na Figura 15.13.

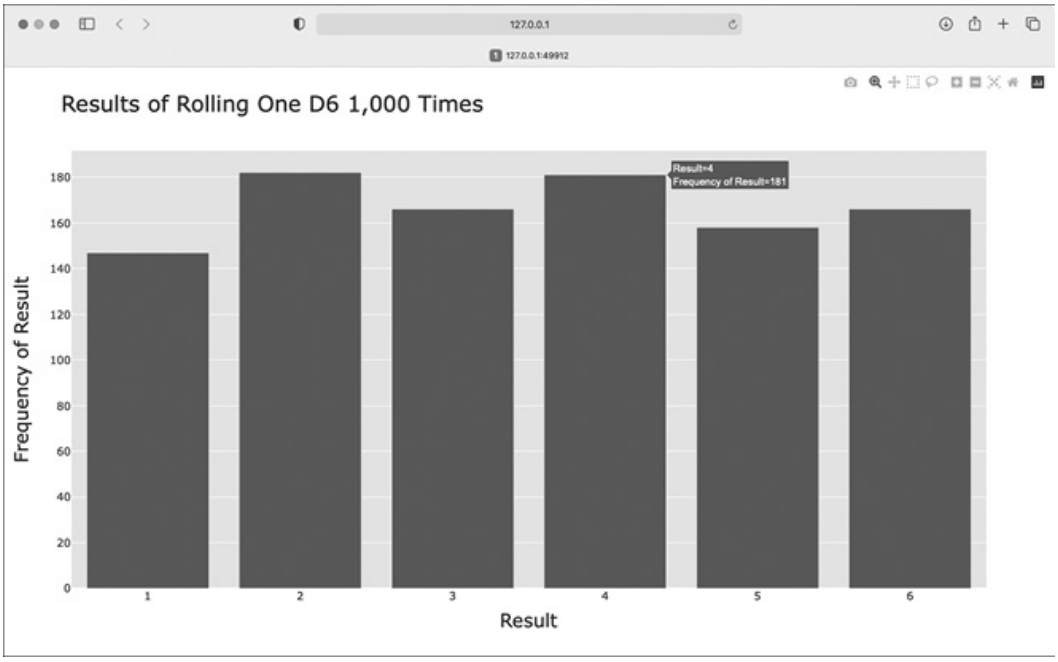


Figura 15.13: Um simples gráfico de barras criado com Plotly.

Lançando dois dados

O lançamento de dois dados resulta em números maiores e uma distribuição diferente de resultados. Modificaremos nosso código para criar dois dados D6, a fim de simular a maneira como lançamos dois dados. Sempre que lançarmos os dois dados, somamos os dois números (um de cada dado) e armazenamos a soma em `results`. Salve uma cópia de *die_visual.py* como *dice_visual.py* e faça o seguinte:

dice_visual.py

```
import plotly.express as px

from die import Die

# Cria dois dados D6
die_1 = Die()
die_2 = Die()

# Realiza alguns testes e armazena os resultados em uma lista
results = []
for roll_num in range(1000):
    result = die_1.roll() + die_2.roll()
    results.append(result)
```

```
    # Analisa os resultados
    frequencies = []
2   max_result = die_1.num_sides + die_2.num_sides
3   poss_results = range(2, max_result+1)
    for value in poss_results:
        frequency = results.count(value)
        frequencies.append(frequency)

    # Visualiza os resultados
    title = "Results of Rolling Two D6 Dice 1,000 Times"
    labels = {'x': 'Result', 'y': 'Frequency of Result'}
    fig = px.bar(x=poss_results, y=frequencies, title=title, labels=labels)
    fig.show()
```

Após criarmos duas instâncias de `Die`, lançamos os dados e calculamos a soma dos dois para cada lançamento `1`. O menor resultado possível (2) é a soma do menor número em cada dado. O maior resultado possível (12) é a soma do maior número em cada dado, que atribuímos a `max_result` `2`. A variável `max_result` facilita mais a leitura do código para gerar `poss_results` `3`. Poderíamos ter escrito `range(2, 13)`, mas funcionaria apenas para dois dados D6. Ao modelar situações reais, é melhor escrever um código que possa facilmente modelar uma variedade de situações. Esse código nos possibilita simular o lançamento de um par de dados com qualquer número de lados.

Após executarmos esse código, devemos ver um gráfico que se parece com o da Figura 15.14.

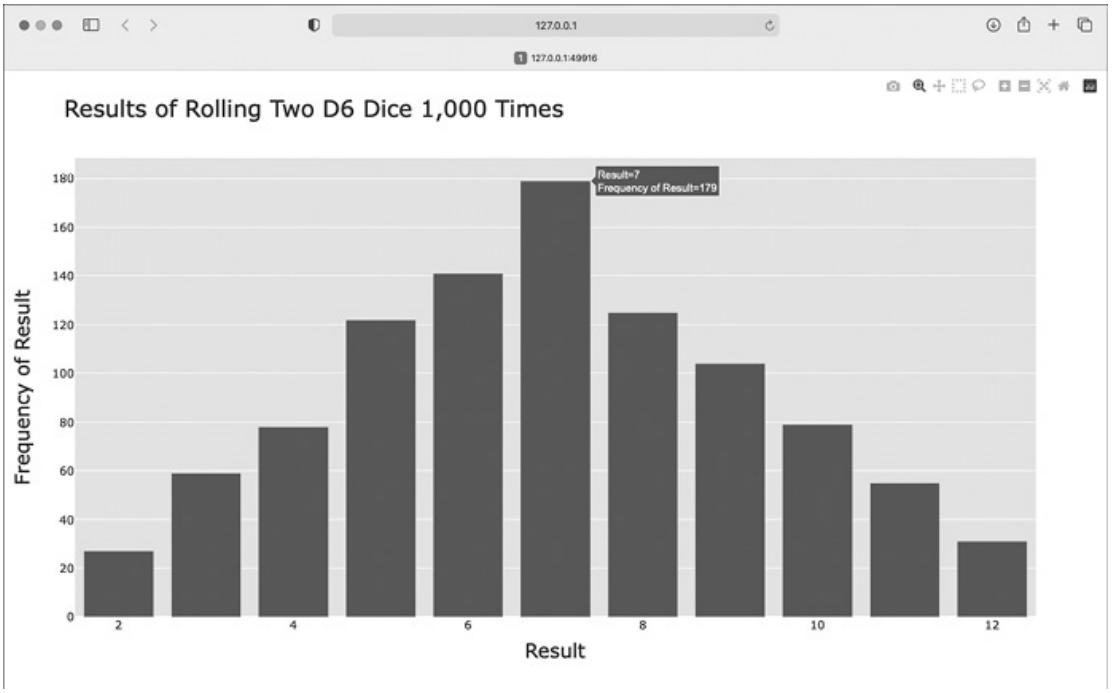


Figura 15.14: Resultados simulados do lançamento de dois dados com seis lados 1.000 vezes.

Esse gráfico mostra a distribuição aproximada dos resultados que provavelmente obteremos quando lançarmos um par de dados D6. Conforme podemos ver, é menos provável que tiremos um 2 ou um 12 e mais provável que tiremos um 7. Isso acontece porque existem seis maneiras de tirar um 7: 1 e 6, 2 e 5, 3 e 4, 4 e 3, 5 e 2, e 6 e 1.

Mais personalizações

Há um problema que devemos solucionar com o gráfico que acabamos de gerar. Como agora existem 11 barras, as configurações default de layout para o eixo *x* deixam algumas das barras sem rótulo. Apesar de as configurações default funcionarem bem para a maioria das visualizações, esse gráfico ficaria melhor visualmente com todas as barras rotuladas.

O Plotly tem um método update_layout() que pode ser usado para fazer uma ampla variedade de atualizações em uma figura após ser criada. Veja como solicitar ao Plotly para fornecer a cada barra o

próprio rótulo:

dice_visual.py

```
-- trecho de código omitido --
fig = px.bar(x=poss_results, y=frequencies, title=title, labels=labels)

# Personaliza ainda mais o gráfico
fig.update_layout(xaxis_dtick=1)

fig.show()
```

O método update_layout() atua no objeto fig, que representa o gráfico geral. Aqui, utilizamos o argumento xaxis_dtick, que especifica a distância entre as marcações de escala no eixo *x.* Como definimos esse espaçamento em 1, cada barra é rotulada. Quando executarmos *dice_visual.py* mais uma vez, deveremos ver um rótulo em cada barra.

Lançando dados com tamanhos diferentes

Criaremos um dado com seis lados e um dado com dez lados, e veremos o que acontece quando os lançamos 50.000 vezes:

dice_visual_d6d10.py

```
import plotly.express as px

from die import Die

# Cria um D6 e um D10
die_1 = Die()
die_2 = Die(10)

# Faz alguns lançamentos e armazena os resultados em uma lista
results = []
for roll_num in range(50_000):
    result = die_1.roll() + die_2.roll()
    results.append(result)

# Analisa os resultados
-- trecho de código omitido --

# Visualiza os resultados
```

```
2 title = "Results of Rolling a D6 and a D10 50,000 Times"
  labels = {'x': 'Result', 'y': 'Frequency of Result'}
  -- trecho de código omitido --
```

Para criar um D10, passamos o argumento `10` ao criar a segunda instância de `Die` `1` e alteramos o primeiro loop para simular 50.000 lançamentos em vez de 1.000. Alteramos também o título do gráfico `2`.

A Figura 15.15 mostra o gráfico resultante. Em vez de ter um resultado mais provável, temos cinco deles. Isso ocorre porque existe somente uma forma de tirar o menor valor (1 e 1) e o maior valor (6 e 10), mas o dado menor limita o número de formas por meio das quais podemos gerar os números intermediários. Existem seis formas de tirar um 7, 8, 9, 10 ou 11, pois como são os resultados mais comuns, temos probabilidade igual de tiramos um deles.

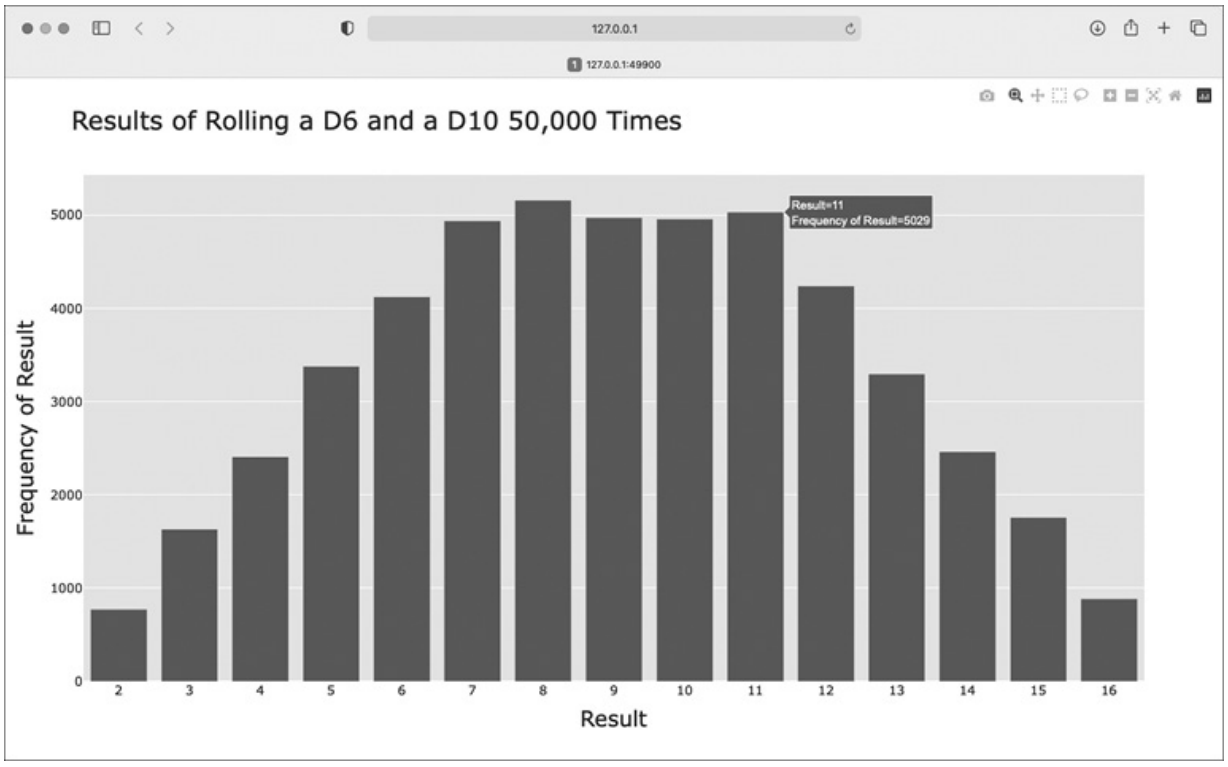


Figura 15.15: Os resultados do lançamento de um dado com seis lados e de um dado com dez lados 50.000 vezes.

Nossa habilidade de usar o Plotly para modelar o lançamento de

dados nos dá autonomia considerável para investigar esse fenômeno. Basta alguns minutos para simularmos um grande número de lançamentos usando uma ampla variedade de dados.

Salvando os gráficos

Quando gostamos de um gráfico, sempre podemos salvá-lo como um arquivo HTML em nosso navegador. Mas também podemos fazer isso de maneira programada. Para salvar seu gráfico como arquivo HTML, substitua a chamada `fig.show()` por uma para `fig.write_html()`:

```
fig.write_html('dice_visual_d6d10.html')
```

O método `write_html()` exige um argumento: o nome do arquivo para escrever. Se fornecermos apenas um nome de arquivo, esse arquivo será salvo no mesmo diretório que o arquivo *.py*. É possível também chamar `savefig()` com um objeto `Path`, e escrever o arquivo de saída em qualquer lugar que queira em seu sistema.

FAÇA VOCÊ MESMO

15.6 Dois D8s: Crie uma simulação mostrando o que acontece quando lançamos dois dados com oito lados 1.000 vezes. Tente imaginar como acha que a visualização será antes de executar a simulação e, em seguida, veja se sua intuição acertou. Aumente gradativamente o número de lançamentos até começar a estressar os limites dos recursos do seu sistema.

15.7 Três dados: Quando lançamos três dados D6, o menor número que podemos tirar é 3 e o maior número é 18. Crie uma visualização que mostre o que acontece quando lançamos três dados D6.

15.8 Multiplicação: Ao lançar dois dados, normalmente somamos os dois números para obter o resultado. Crie uma visualização que mostre o que acontece se multiplicarmos esses números uns pelos outros.

15.9 Die Comprehensions: Para que fique claro, as listagens nesta seção usam a forma longa de loops `for`. Caso se sinta à vontade para usar list comprehensions, tente escrever uma comprehensions para um ou ambos os loops em cada um desses programas.

15.10 Praticando com as duas bibliotecas: Tente usar a Matplotlib para criar a visualização do lançamento de um dado, e use o Plotly para criar a visualização de um passeio aleatório. (Será necessário consultar a documentação de cada biblioteca para concluir este exercício.)

Recapitulando

Neste capítulo, aprendemos a gerar conjuntos de dados e criar visualizações desses dados. Criamos gráficos simples com a biblioteca Matplotlib e usamos um gráfico de dispersão para explorar passeios aleatórios. Vimos também como criar um histograma com o Plotly, e o usamos para explorar os resultados do lançamento de dados de diferentes tamanhos.

Quando geramos nossos próprios conjuntos de dados com código, aprendemos um jeito interessante e efetivo de modelarmos e explorarmos uma ampla variedade de situações cotidianas. À medida que continua trabalhando nos projetos de visualização de dados a seguir, preste atenção à situações que podem ser modeladas com código. Fique atento às visualizações nas mídias de notícias e veja se consegue identificar as que foram geradas usando métodos semelhantes aos que você está aprendendo com esses projetos.

No Capítulo 16, faremos o download de dados em fontes online e continuaremos usando a Matplotlib e o Plotly para explorá-los.

CAPÍTULO 16

Fazendo download dos dados

Neste capítulo, faremos o download de conjuntos de dados em fontes online e criaremos visualizações práticas desses dados. É possível encontrar uma variedade extraordinária de dados online, e muitos deles sequer foram minuciosamente analisados. Com a habilidade de analisar esses dados, é possível identificar padrões e relações que ninguém mais detectou.

Acessaremos e visualizaremos dados armazenados em dois formatos comuns: CSV e JSON. Utilizaremos o módulo `csv` do Python a fim de processar dados meteorológicos armazenados no formato CSV e para analisar as temperaturas mínimas e máximas ao longo do tempo em dois locais diferentes. Em seguida, usaremos a Matplotlib para gerar um gráfico com base em nossos dados baixados a fim de apresentar variações de temperatura em dois ambientes antagônicos: Sitka, Alasca e Vale da Morte, Califórnia. Mais adiante neste capítulo, recorreremos ao módulo `json` para acessar dados de terremotos armazenados no formato GeoJSON e usaremos o Plotly para desenhar um mapa-múndi que mostre os locais e as magnitudes de terremotos recentes.

No final deste capítulo, você estará preparado para trabalhar com diversos tipos de conjuntos de dados em diferentes formatos e terá um conhecimento mais aprofundado de como criar visualizações complexas. Ser capaz de acessar e visualizar dados online é primordial para trabalhar com uma ampla variedade de conjuntos de dados do mundo real.

Formato de arquivo CSV

Uma forma simples de armazenar dados em um arquivo de texto é escrevê-los como uma série de valores separados por vírgulas, chamados de *valores separados por vírgulas*. Os arquivos resultantes são arquivos *CSV*. Por exemplo, vejamos um fragmento de um conjunto de dados meteorológicos no formato CSV:

```
"USW00025333","SITKA AIRPORT, AK US","2021-01-01",,"44","40"
```

Trata-se de um trecho dos dados meteorológicos a partir de 1º de janeiro de 2021, da cidade de Sitka, Alasca. Inclui as temperaturas máximas e mínimas do dia, bem como uma série de outras medições diárias. Para nós, seres humanos, pode ser cansativo ler arquivos CSV, apesar disso, os programas conseguem processar e extrair informações deles com rapidez e acurácia.

Vamos começar com um pequeno conjunto de dados meteorológicos em formato CSV registrados na cidade de Sitka; está disponível nos recursos deste livro em https://ehmatthes.github.io/pcc_3e. Crie uma pasta chamada *weather_data* dentro da pasta em que está salvando os programas deste capítulo. Copie o arquivo *sitka_weather_07-2021_simple.csv* para essa pasta nova. (Após fazer o download dos recursos deste livro, você terá todos os arquivos necessários para este projeto.)

NOTA *O download dos dados meteorológicos deste projeto foi originalmente feito de:* https://ncdc.noaa.gov/cdo-web.

Parseamento dos cabeçalhos dos arquivos CSV

O módulo `csv` da biblioteca padrão do Python faz o parse das linhas em um arquivo CSV e nos possibilita extrair de modo rápido os valores em que estamos interessados. Começaremos examinando a primeira linha do arquivo, que contém uma série de cabeçalhos para os dados. Esses cabeçalhos nos informam qual tipo de informação os dados armazenam:

sitka_highs.py

```
from pathlib import Path
import csv

1 path = Path('weather_data/sitka_weather_07-2021_simple.csv')
  lines = path.read_text().splitlines()

2 reader = csv.reader(lines)
3 header_row = next(reader)
  print(header_row)
```

Primeiro, importamos o Path e o módulo csv. Depois, criamos um objeto Path que analisa a pasta *weather_data* e aponta para o arquivo de dados meteorológicos específico com o qual queremos trabalhar 1. Lemos o arquivo e encadeamos o método splitlines() para obter uma lista de todas as linhas no arquivo, que atribuímos à lines.

Em seguida, criamos um objeto reader 2. Trata-se de um objeto que pode ser usado para fazer o parse de cada linha no arquivo. Para criar um objeto reader, chame a função csv.reader() e passe a ele a lista de linhas do arquivo CSV.

Ao receber um objeto reader, a função next() retorna a próxima linha do arquivo, começando no início dele. Como aqui chamamos next() somente uma vez, temos a primeira linha do arquivo, que contém os cabeçalhos do arquivo 3. Atribuímos os dados retornados a header_row. Como podemos ver, header_row contém cabeçalhos significativos, relacionados ao clima, que nos transmite quais informações cada linha de dados armazena:

```
['STATION', 'NAME', 'DATE', 'TAVG', 'TMAX', 'TMIN']
```

O objeto reader processa a primeira linha de valores separados por vírgulas no arquivo e armazena cada valor como um item em uma lista. O cabeçalho STATION representa o código para a estação meteorológica que registrou esses dados. A posição deste cabeçalho nos informa que o primeiro valor em cada linha será o código da estação meteorológica. O cabeçalho NAME indica que o segundo valor em cada linha é o nome da estação meteorológica que efetuou o

registro. O restante dos cabeçalhos especifica quais tipos de informações foram registradas em cada leitura. Por ora, os dados em que estamos mais interessados são a data (DATE), a temperatura máxima (TMAX) e a temperatura mínima (TMIN). Esse é um simples conjunto de dados que contém apenas dados relacionados à temperatura. Ao fazer o download de seus dados meteorológicos, é possível optar por incluir uma série de outras medições relacionadas à velocidade e à direção do vento e a dados de precipitação.

Exibindo os cabeçalhos e suas posições

Para facilitar a compreensão dos dados do cabeçalho do arquivo, exibiremos cada cabeçalho e sua posição na lista:

sitka_highs.py

```
-- trecho de código omitido --
reader = csv.reader(lines)
header_row = next(reader)

for index, column_header in enumerate(header_row):
    print(index, column_header)
```

A função enumerate() retorna o índice de cada item e o valor de cada item à medida que percorremos uma lista com um loop. (Repare que removemos a linha print(header_row) em prol desta versão mais detalhada.)

Vejamos a saída mostrando o índice de cada cabeçalho:

```
0 STATION
1 NAME
2 DATE
3 TAVG
4 TMAX
5 TMIN
```

Podemos ver que as datas e suas temperaturas máximas estão armazenadas nas colunas 2 e 4. Para explorar esses dados, vamos processar cada linha de dados em *sitka_weather_07-2021_simple.csv* e vamos extrair os valores com os índices 2 e 4.

Extraindo e lendo os dados

Agora que sabemos de quais colunas de dados precisamos, leremos alguns desses dados. Inicialmente, leremos a temperatura máxima de cada dia:

sitka_highs.py

```
-- trecho de código omitido --
reader = csv.reader(lines)
header_row = next(reader)

# Extrai as temperaturas máximas
highs = []
for row in reader:
    high = int(row[4])
    highs.append(high)

print(highs)
```

Criamos uma lista vazia chamada highs 1 e, em seguida, percorremos com um loop as linhas restantes no arquivo 2. O objeto reader continua de onde parou no arquivo CSV e retorna automaticamente cada linha seguida de sua posição atual. Como já lemos a linha de cabeçalho, o loop começará na segunda linha em que os próprios dados começam. Em cada passagem pelo loop, extraímos os dados do índice 4, correspondente ao cabeçalho TMAX, e os atribuímos à variável high 3. Recorremos à função int() para converter os dados, armazenados como uma string, em um formato numérico a fim de que possamos usá-los. Depois, anexamos esse valor à highs.

A listagem a seguir mostra os dados agora armazenados em highs:

```
[61, 60, 66, 60, 65, 59, 58, 58, 57, 60, 60, 60, 57, 58, 60, 61, 63, 63, 70,
64, 59, 63, 61, 58, 59, 64, 62, 70, 70, 73, 66]
```

Extraímos a temperatura máxima para cada data e armazenamos cada valor em uma lista. Agora, criaremos a visualização desses dados.

Plotando dados em um gráfico de temperatura

A fim de visualizarmos os dados de temperatura que temos, primeiro

criaremos um gráfico simples com as temperaturas máximas diárias usando a Matplotlib, como mostrado a seguir.

sitka_highs.py

```
from pathlib import Path
import csv

import matplotlib.pyplot as plt

path = Path('weather_data/sitka_weather_07-2021_simple.csv')
lines = path.read_text().splitlines()
    -- trecho de código omitido --

# Plota as temperaturas máximas
plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.plot(highs, color='red')

# Formata o gráfico
ax.set_title("Daily High Temperatures, July 2021", fontsize=24)
ax.set_xlabel('', fontsize=16)
ax.set_ylabel("Temperature (F)", fontsize=16)
ax.tick_params(labelsize=16)

plt.show()
```

Passamos a lista de temperaturas máximas para `plot()` e `color='red'` a fim de plotar os pontos com a cor vermelha 1. (Plotaremos as temperaturas máximas na cor vermelha e as temperaturas mínimas na cor azul.) Em seguida, especificamos alguns outros detalhes de formatação, como o título, o tamanho da fonte e os rótulos 2, assim como fizemos no Capítulo 15. Como ainda não adicionamos as datas, não rotularemos o eixo *x*, mas `ax.set_xlabel()` modifica o tamanho da fonte para que os rótulos default fiquem mais legíveis 3. A Figura 16.1 mostra o gráfico resultante: um simples gráfico de linhas das temperaturas máximas de julho de 2021, em Sitka, Alasca.

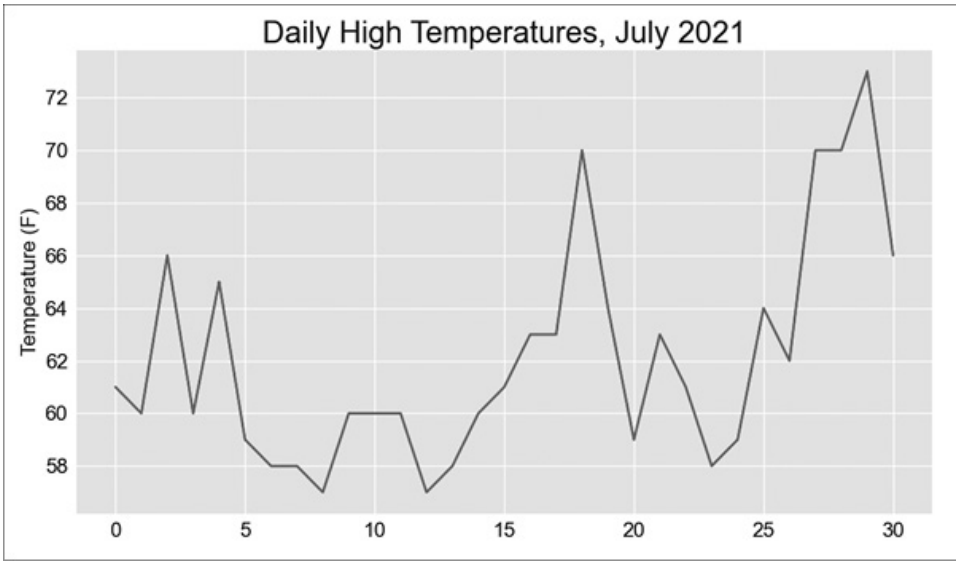


Figura 16.1: Um gráfico de linhas mostrando as temperaturas máximas diárias de julho de 2021, em Sitka, Alasca.

Módulo datetime

Adicionaremos datas ao nosso gráfico para que fique mais útil. A primeira data do arquivo de dados meteorológicos está na segunda linha:

```
"USW00025333","SITKA AIRPORT, AK US","2021-07-01",,"61","53"
```

Os dados serão lidos como uma string, logo é necessário uma maneira de converter a string `"2021-07-01"` em um objeto que represente essa data. É possível criar um objeto que represente 1º de julho de 2021, usando o método `strptime()` do módulo `datetime`. Confira como o `strptime()` funciona em uma sessão de terminal:

```
>>> from datetime import datetime
>>> first_date = datetime.strptime('2021-07-01', '%Y-%m-%d')
>>> print(first_date)
2021-07-01 00:00:00
```

Primeiro, importamos a classe `datetime` do módulo `datetime`. Em seguida, chamamos o método `strptime()` com a string contendo a data que queremos processar como primeiro argumento. O segundo argumento informa ao Python como a data está formatada. Nesse exemplo, `'%Y-'` instruí ao Python que procure um ano de quatro

dígitos antes do primeiro traço; '%m-' indica um mês de dois dígitos antes do segundo traço; e %d" significa que a última parte da string é o dia do mês, de 1 a 31.

O método strptime() pode receber uma variedade de argumentos para determinar como interpretar a data. A Tabela 16.1 mostra alguns desses argumentos.

Tabela 16.1: Argumentos para formatação de data e de hora do módulo datetime

Argumento	Significado
%A	Nome do dia da semana, como Monday
%B	Nome do mês, como January
%m	Mês, como um número (de 01 a 12)
%d	Dia do mês, como número (de 01 a 31)
%Y	Ano com quatro dígitos, como 2019
%y	Ano com dois dígitos, como 19
%H	Hora, em formato 24 horas (de 00 a 23)
%I	Hora em formato 12 horas (de 01 a 12)
%p	AM ou PM
%M	Minutos (de 00 a 59)
%S	Segundos (de 00 a 61)

Plotando datas

É possível melhorar nosso gráfico extraindo as datas a partir das leituras diárias de temperatura máxima e usar essas datas no eixo *x*:

sitka_highs.py

```
from pathlib import Path
import csv
from datetime import datetime

import matplotlib.pyplot as plt

path = Path('weather_data/sitka_weather_07-2021_simple.csv')
lines = path.read_text().splitlines()

reader = csv.reader(lines)
header_row = next(reader)
```

```
    # Extrai datas e temperaturas máximas
1   dates, highs = [], []
    for row in reader:
2       current_date = datetime.strptime(row[2], '%Y-%m-%d')
        high = int(row[4])
        dates.append(current_date)
        highs.append(high)

    # Plota as temperaturas máximas
    plt.style.use('seaborn')
    fig, ax = plt.subplots()
3   ax.plot(dates, highs, color='red')

    # Formata o gráfico
    ax.set_title("Daily High Temperatures, July 2021", fontsize=24)
    ax.set_xlabel('', fontsize=16)
4   fig.autofmt_xdate()
    ax.set_ylabel("Temperature (F)", fontsize=16)
    ax.tick_params(labelsize=16)

    plt.show()
```

Criamos duas listas vazias a fim de armazenar as datas e as temperaturas máximas do arquivo `1`. Em seguida, convertemos os dados que contêm as informações de data (`row[2]`) em um objeto `datetime` `2` e o anexamos à `date`. Passamos as datas e os valores de temperaturas máximas para `plot()` `3`. A chamada para `fig.autofmt_xdate()` `4` desenha os rótulos de data na diagonal a fim de evitar que se sobreponham. A Figura 16.2 mostra o gráfico melhorado.

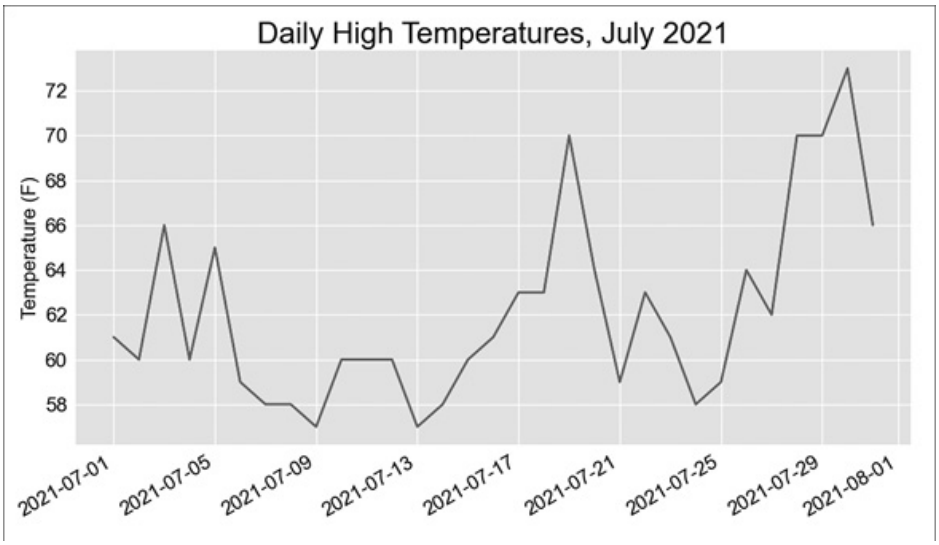


Figura 16.2: O gráfico está mais pertinente, agora que tem datas no eixo x.

Plotando um período de tempo maior

Com nosso gráfico definido, incluiremos dados adicionais para entender melhor as condições climáticas na cidade de Sitka. Copie o arquivo *sitka_weather_2021_simple.csv*, que contém dados meteorológicos de um ano inteiro da cidade de Sitka, para a pasta em que está armazenando os dados dos programas deste capítulo.

Agora, podemos gerar um gráfico das condições climáticas durante o período de um ano:

sitka_highs.py

```
-- trecho de código omitido --
path = Path('weather_data/sitka_weather_2021_simple.csv')
lines = path.read_text().splitlines()
-- trecho de código omitido --
# Formata o gráfico
ax.set_title("Daily High Temperatures, 2021", fontsize=24)
ax.set_xlabel('', fontsize=16)
-- trecho de código omitido --
```

Modificamos o nome do arquivo para usar o arquivo novo de dados *sitka_weather_2021_simple.csv*, e atualizamos o título do nosso gráfico para espelhar a mudança de seu conteúdo. A Figura 16.3 mostra o gráfico resultante.

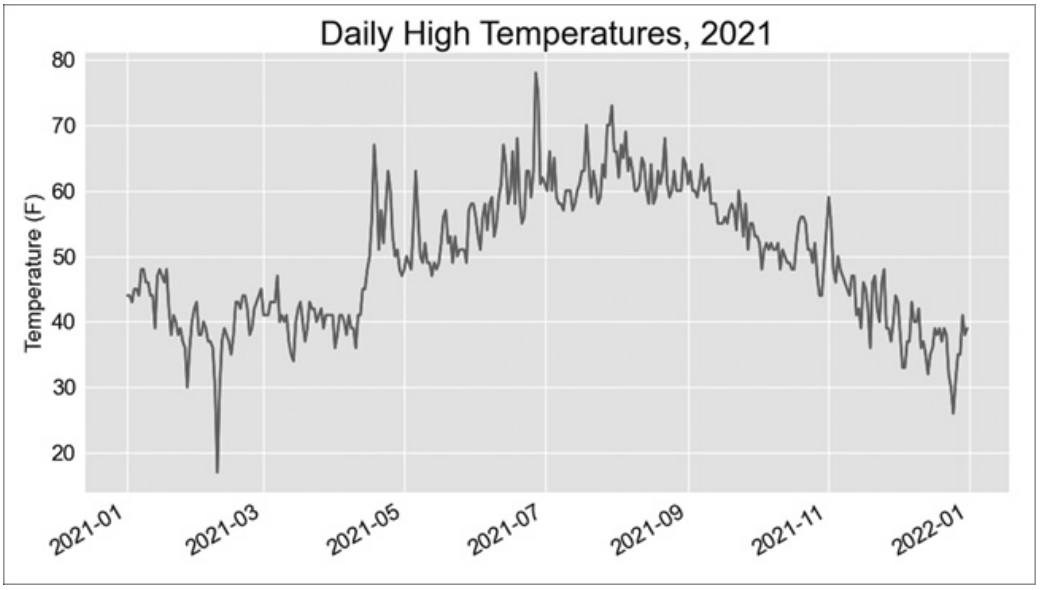


Figura 16.3: Dados de um ano.

Plotando uma segunda série de dados

Podemos incrementar ainda mais nosso gráfico, incluindo as temperaturas mínimas. É necessário extrair as temperaturas mínimas do arquivo de dados e adicioná-las ao nosso gráfico, como mostrado aqui:

sitka_highs_lows.py

```
    -- trecho de código omitido --
    reader = csv.reader(lines)
    header_row = next(reader)

    # Extrai datas, temperaturas máximas e mínimas
1   dates, highs, lows = [], [], []
    for row in reader:
        current_date = datetime.strptime(row[2], '%Y-%m-%d')
        high = int(row[4])
2       low = int(row[5])
        dates.append(current_date)
        highs.append(high)
        lows.append(low)

    # Plota as temperaturas máximas e mínimas
    plt.style.use('seaborn')
    fig, ax = plt.subplots()
    ax.plot(dates, highs, color='red')
```

```
3 ax.plot(dates, lows, color='blue')

  # Formata o gráfico
4 ax.set_title("Daily High and Low Temperatures, 2021", fontsize=24)
  -- trecho de código omitido --
```

Adicionamos a lista vazia `lows` para armazenar as temperaturas mínimas `1` e, em seguida, extraímos e armazenamos a temperatura mínima para cada data a partir da sexta posição em cada linha (`row[5]`) `2`. Adicionamos uma chamada a `plot()` para as temperaturas mínimas e colorimos esses valores de azul `3`. Por último, atualizamos o título `4`. A Figura 16.4 mostra o gráfico resultante.

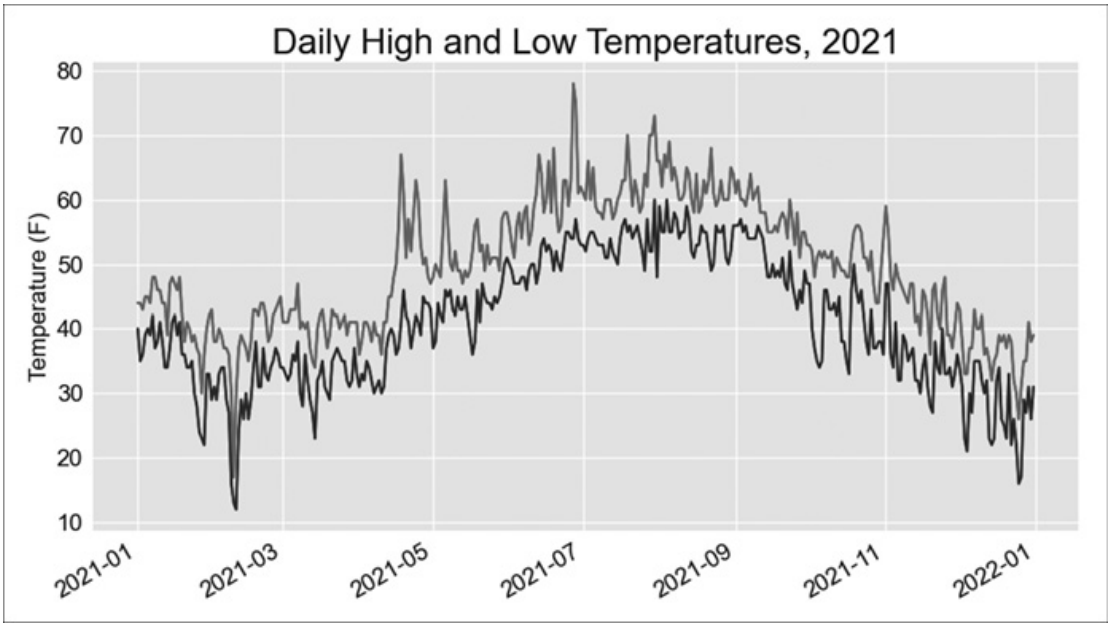


Figura 16.4: Duas séries de dados no mesmo gráfico.

Sombreando uma área no gráfico

Após adicionarmos duas séries de dados, agora podemos examinar a variação de temperaturas para cada dia. Daremos um toque final ao gráfico usando sombreamento para evidenciar o intervalo entre as temperaturas máximas e mínimas de cada dia. Para isso, utilizaremos o método `fill_between()`, que pega uma série de valores *x* e duas séries de valores *y* e preenche o espaço entre as duas séries de valores *y*:

sitka_highs_lows.py

```
    -- trecho de código omitido --
    # Plota as temperaturas máximas e mínimas
    plt.style.use('seaborn')
    fig, ax = plt.subplots()
1   ax.plot(dates, highs, color='red', alpha=0.5)
    ax.plot(dates, lows, color='blue', alpha=0.5)
2   ax.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)
    -- trecho de código omitido --
```

O argumento alpha controla a transparência de uma cor 1. Um valor alpha de 0 é completamente transparente, e um valor de 1 (o default) é completamente opaco. Ao definir alpha como 0,5, fazemos com que as linhas vermelhas e azuis do gráfico fiquem mais claras.

Passamos para fill_between() a lista dates para os valores *x* e, em seguida, passamos as duas séries de valores *y* highs e lows 2. O argumento facecolor determina a cor da região sombreada; nós lhe fornecemos um valor alpha baixo de 0,1. Assim, a região preenchida conecta as duas séries de dados sem desviar a atenção das informações que representam. A Figura 16.5 mostra o gráfico com a região sombreada entre as temperaturas máximas e mínimas.

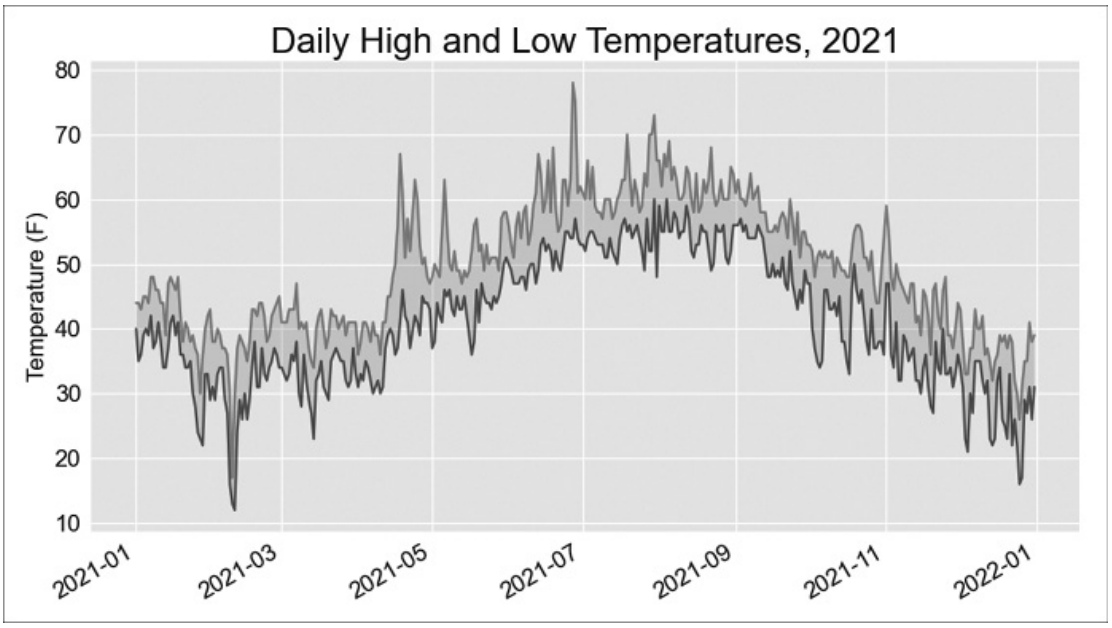


Figura 16.5: A região entre os dois conjuntos de dados fica sombreada.

O sombreamento ajuda a evidenciar imediatamente o intervalo entre os dois conjuntos de dados.

Verificação de erros

Devemos conseguir executar o código *sitka_highs_lows.py* usando dados de qualquer local. No entanto, algumas estações meteorológicas coletam dados diferentes de outras, e algumas delas, ocasionalmente, falham em coletar os dados e não conseguem coletar alguns dos dados que deveriam. Dados ausentes podem resultar em exceções que acarretam falhas em nossos programas, a menos que os manipulemos de modo adequado.

Por exemplo, vejamos o que acontece quando tentamos gerar um gráfico de temperatura para o Vale da Morte, Califórnia. Copie o arquivo *death_valley_2021_simple.csv* para a pasta em que está armazenando os dados dos programas deste capítulo.

Primeiro, executaremos o código para ver os cabeçalhos incluídos nesse arquivo de dados:

death_valley_highs_lows.py

```
from pathlib import Path
import csv

path = Path('weather_data/death_valley_2021_simple.csv')
lines = path.read_text().splitlines()

reader = csv.reader(lines)
header_row = next(reader)

for index, column_header in enumerate(header_row):
    print(index, column_header)
```

Aqui está a saída:

```
0 STATION
1 NAME
2 DATE
3 TMAX
4 TMIN
5 TOBS
```

A data está na mesma posição, no índice 2. Mas como as temperaturas máximas e mínimas estão nos índices 3 e 4, precisaremos alterar os índices em nosso código para espelhar essas posições novas. Em vez de incluir uma leitura de temperatura média diária, esta estação inclui TOBS, uma leitura para um tempo de observação específico.

Altere *sitka_highs_lows.py* a fim de gerar um gráfico para o Vale da Morte usando os índices que acabamos de observar e veja o que acontece:

death_valley_highs_lows.py

```
-- trecho de código omitido --
path = Path('weather_data/death_valley_2021_simple.csv')
lines = path.read_text().splitlines()
    -- trecho de código omitido --
# Extrai datas, temperaturas máximas e mínimas
dates, highs, lows = [], [], []
for row in reader:
    current_date = datetime.strptime(row[2], '%Y-%m-%d')
    high = int(row[3])
    low = int(row[4])
    dates.append(current_date)
-- trecho de código omitido --
```

Atualizamos o programa para ler o arquivo de dados do Vale da Morte e alteramos os índices para corresponder às posições TMAX e TMIN desse arquivo.

Quando executamos o programa, recebemos um erro:

```
Traceback (most recent call last):
  File "death_valley_highs_lows.py", line 17, in <module>
    high = int(row[3])
1 ValueError: invalid literal for int() with base 10: ''
```

O traceback nos informa que Python não consegue processar a temperatura máxima para uma das datas porque não pode transformar uma string vazia (") em um inteiro 1. Em vez de examinar os dados para descobrir qual leitura está faltando, vamos apenas lidar diretamente com o caso de dados ausentes.

Executaremos o código de verificação de erros quando os valores estiverem sendo lidos do arquivo CSV para lidar com exceções que possam ser lançadas. Veja o método usado:

death_valley_highs_lows.py

```
    -- trecho de código omitido --
    for row in reader:
        current_date = datetime.strptime(row[2], '%Y-%m-%d')
1       try:
            high = int(row[3])
            low = int(row[4])
        except ValueError:
2           print(f"Missing data for {current_date}")
3       else:
            dates.append(current_date)
            highs.append(high)
            lows.append(low)

    # Plota as temperaturas máximas e mínimas
    -- trecho de código omitido --

    # Formata o gráfico
4   title = "Daily High and Low Temperatures, 2021\nDeath Valley, CA"
    ax.set_title(title, fontsize=20)
    ax.set_xlabel('', fontsize=16)
    -- trecho de código omitido --
```

Sempre que examinamos uma linha, tentamos extrair a data e as temperaturas máximas e mínimas 1. Se algum dado estiver ausente, o Python lançará um `ValueError` e nós o manipularemos exibindo uma mensagem de erro que inclua a data dos dados ausentes 2. Após exibir o erro, o loop continuará processando a próxima linha. Caso todos os dados de uma data sejam acessados sem erro, o bloco `else` será executado e os dados serão anexados às listas adequadas 3. Como estamos plotando informações para um local novo, atualizamos o título para incluir o local no gráfico e utilizamos um tamanho de fonte menor para acomodar o título mais extenso 4.

Agora, quando executarmos *death_valley_highs_lows.py*, veremos que somente uma data tinha dados ausentes:

```
Missing data for 2021-05-04 00:00:00
```

Como o erro é tratado de forma apropriada, nosso código é capaz de gerar um gráfico, que ignora os dados ausentes. A Figura 16.3 mostra o gráfico resultante.

Ao comparar esse gráfico com o gráfico de Sitka, é possível constatar que o Vale da Morte é mais quente do que o sudeste do Alasca, como esperado. Além do mais, a variação de temperaturas a cada dia é maior no deserto. A altura da região sombreada confirma isso.

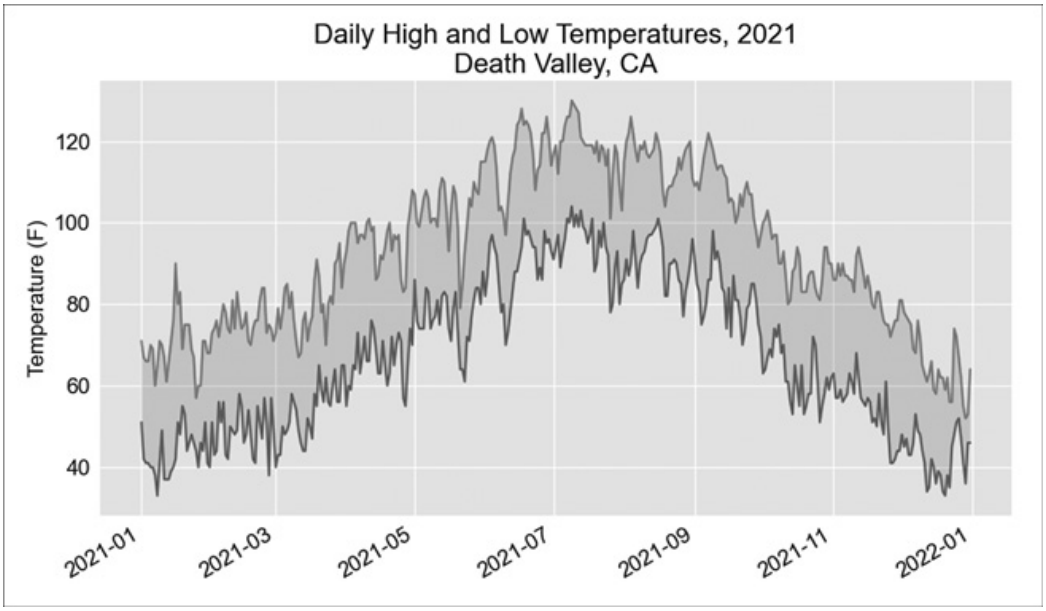


Figura 16.6: Temperaturas máximas e mínimas diárias para o Vale da Morte.

Você trabalhará com muitos conjuntos de dados com dados ausentes, formatados erroneamente ou incorretos. É possível usar as ferramentas que aprendeu na primeira metade deste livro para lidar com essas situações. Aqui usamos um bloco `try-except-else` para lidar com dados ausentes. Às vezes, usaremos `continue` para ignorar alguns dados, `remove()` ou `del` a fim de eliminar alguns dados depois de extraídos. Recorra a qualquer abordagem que funcione, desde que o resultado seja uma visualização significativa e com acurácia.

Fazendo o download de seus próprios dados

Para fazer o download de seus próprios dados meteorológicos, siga as seguintes etapas:

1. Visite o site NOAA Climate Data Online em https://www.ncdc.noaa.gov/cdo-web. Na seção Discover Data By, clique em **Search Tool**. Na caixa Select a Dataset box, selecione **Daily Summaries**.
2. Selecione um intervalo de datas e, na seção Search For, escolha **ZIP Codes**. Digite o código postal em que está interessado e clique em **Search**.
3. Na próxima página, você verá um mapa e algumas informações sobre a área que quer. Abaixo do nome do local, clique em **View Full Details** ou clique no mapa e, depois, em **Full Details**.
4. Role para baixo e clique em **Station List** para ver as estações meteorológicas disponíveis nessa área. Clique em um dos nomes das estações e clique em **Add to Cart**. Esses dados são gratuitos, ainda que o site use um ícone de carrinho de compras. No canto superior direito, clique no carrinho.
5. Em Select the Output Format, selecione **Custom GHCN-Daily CSV**. Verifique se o intervalo de datas está correto e clique em **Continue**.
6. Na próxima página, é possível selecionar os tipos de dados desejados. É possível fazer o download de um tipo de dados (por exemplo, com foco na temperatura do ar) ou fazer o download de todos os dados disponíveis nesta estação. Faça suas escolhas e clique em **Continue**.
7. Na última página, você verá um resumo do seu pedido. Digite seu endereço de e-mail e clique em **Submit Order**. Você receberá uma confirmação de que o seu pedido foi recebido e, dentro de alguns minutos, deverá receber outro e-mail com um link para fazer o download de seus dados.

Os dados baixados devem ser estruturados da mesma forma que os dados com os quais trabalhamos nesta seção. Talvez tenham cabeçalhos diferentes dos que vimos nesta seção, mas se você

seguir as mesmas etapas que seguimos aqui, poderá gerar visualizações dos dados nos quais está interessado.

FAÇA VOCÊ MESMO

16.1 Precipitação na cidade de Sitka: Sitka está localizada em uma floresta tropical temperada, por isso recebe uma boa quantidade de chuva. No arquivo de dados sitka_weather_2021_full.csv *há* um cabeçalho chamado PRCP, que representa as quantidades diárias de chuva. Crie uma visualização com foco nos dados dessa coluna. Caso esteja curioso, é possível repetir o exercício para o Vale da Morte a fim de saber como o índice pluviométrico é baixo no deserto.

16.2 Comparação entre a cidade de Sitka e o Vale da Morte: As escalas de temperatura nos gráficos Sitka e Death Valley retratam os diferentes intervalos de dados. Para comparar com acurácia as variações de temperatura em Sitka com a do Vale da Morte, precisaremos de escalas idênticas no eixo *y.* Altere as configurações do eixo *y* em um ou em ambos os gráficos das figuras 16.5 e 16.6. Em seguida, faça uma comparação direta entre as variações de temperatura em Sitka e no Vale da Morte (ou quaisquer dois lugares que queira comparar).

16.3 São Francisco: As temperaturas em São Francisco são mais parecidas com as temperaturas em Sitka ou com as temperaturas no Vale da Morte? Faça o download de alguns dados de São Francisco e gere um gráfico com as temperaturas máximas e mínimas para fazer uma comparação.

16.4 Índices automáticos: Nesta seção, programamos de modo fixo os índices correspondentes às colunas TMIN e TMAX. Use a linha de cabeçalho para determinar os índices para esses valores, assim seu programa passa a funcionar com Sitka ou com o Vale da Morte. Além disso, utilize o nome da estação para gerar automaticamente um título adequado para o seu gráfico.

16.5 Explore: Gere mais algumas visualizações que examinem qualquer outro aspecto meteorológico que estiver interessado para qualquer local que lhe desperte curiosidade.

Mapeando conjuntos de dados globais: Formato GeoJSON

Nesta seção, faremos o download de um conjunto de dados que representa todos os terremotos ocorridos no mundo durante o mês anterior. Em seguida, criaremos um mapa mostrando a localização desses terremotos e a magnitude de cada um deles. Como os dados são armazenados no formato GeoJSON, trabalharemos com esses dados usando o módulo `json`. Com o gráfico `scatter_geo()` do Plotly, criaremos visualizações que mostram claramente a distribuição global de terremotos.

Fazendo o download dos dados de terremotos

Crie uma pasta chamada *eq_data* dentro da pasta em que está salvando os programas deste capítulo. Copie o arquivo *eq_1_day_m1.geojson* para essa pasta nova. Os terremotos são categorizados por sua magnitude na escala Richter. Esse arquivo inclui dados para todos os terremotos com magnitude M1 ou maior, ocorridos nas últimas 24 horas (no momento em que eu escrevia este livro). Esses dados vêm de um dos feeds de dados de terremotos do United States Geological Survey, em https://earthquake.usgs.gov/earthquakes/feed.

Examinando Dados GeoJSON

Ao abrir eq_1_day_m1.geojson, veremos que se trata de um arquivo muito denso e difícil de ler:

```
{"type":"FeatureCollection","metadata":{"generated":1649052296000,...
{"type":"Feature","properties":{"mag":1.6,"place":"63 km SE of Ped...
{"type":"Feature","properties":{"mag":2.2,"place":"27 km SSE of Ca...
{"type":"Feature","properties":{"mag":3.7,"place":"102 km SSE of S...
{"type":"Feature","properties":{"mag":2.92000008,"place":"49 km SE...
{"type":"Feature","properties":{"mag":1.4,"place":"44 km NE of Sus...
-- trecho de código omitido --
```

A formatação desse arquivo é compatível para leitura de máquinas, não para humanos. No entanto, é possível constatar que o arquivo contém alguns dicionários, bem como informações que nos interessam, como magnitudes e locais de terremotos.

O módulo `json` fornece uma variedade de ferramentas para explorar e trabalhar com dados JSON. Algumas dessas ferramentas nos ajudarão a reformatar o arquivo para que possamos analisar os dados brutos com mais facilidade, antes de trabalhar programaticamente com o arquivo.

Vamos começar fazendo o download dos dados e exibindo-os em um formato mais fácil de ler. Como se trata de um arquivo extenso de dados, em vez de exibi-lo, reescreveremos os dados em um arquivo novo. Em seguida, podemos abrir esse arquivo para analisar os

dados de todas as formas possíveis com mais facilidade:

eq_explore_data.py

```
  from pathlib import Path
  import json

  # Lê os dados como uma string e os converte em um objeto Python
  path = Path('eq_data/eq_data_1_day_m1.geojson')
  contents = path.read_text()
1 all_eq_data = json.loads(contents)

  # Cria uma versão mais legível do arquivo de dados
2 path = Path('eq_data/readable_eq_data.geojson')
3 readable_contents = json.dumps(all_eq_data, indent=4)
  path.write_text(readable_contents)
```

Lemos o arquivo de dados como uma string e utilizamos `json.loads()` para converter a representação da string do arquivo em um objeto Python 1. É a mesma abordagem que usamos no Capítulo 10. Nesse caso, todo o conjunto de dados é convertido em um único dicionário, que atribuímos a `all_eq_data`. Depois, definimos um `path` novo em podemos escrever esses mesmos dados em um formato mais legível 2. A função `json.dumps()` que vimos no Capítulo 10 pode usar um argumento opcional `indent` 3, que informa o quanto indentar elementos aninhados na estrutura de dados.

Ao olharmos em nosso diretório *eq_data* e abrirmos o arquivo *readable_eq_data.json*, veremos o seguinte:

readable_eq_data.json

```
  {
     "type": "FeatureCollection",
1    "metadata": {
        "generated": 1649052296000,
        "url": "https://earthquake.usgs.gov/earthquakes/.../1.0_day.geojson",
        "title": "USGS Magnitude 1.0+ Earthquakes, Past Day",
        "status": 200,
        "api": "1.10.3",
        "count": 160
     },
2    "features": [
     -- trecho de código omitido --
```

A primeira parte do arquivo inclui uma seção com a chave "metadata"1. Isso nos informa quando o arquivo de dados foi gerado e onde podemos encontrar os dados online. O arquivo também nos fornece um título legível por humanos e o número de terremotos incluídos nele. Em um período de 24 horas, 160 terremotos foram registrados.

Esse arquivo GeoJSON tem uma estrutura útil para dados com localização. As informações são armazenadas em uma lista associada à chave "features" 2. Como esse arquivo contém dados de terremotos, os dados estão em forma de lista, em que cada item na lista corresponde a um único terremoto. Mesmo parecendo um tanto confusa, essa estrutura é bastante eficiente. Possibilita que os geólogos armazenem o máximo de informações de que precisarem sobre cada terremoto em um dicionário e, em seguida, inserirem todos esses dicionários em uma lista grande.

Vejamos um dicionário que representa um único terremoto:

readable_eq_data.json

```
    -- trecho de código omitido --
      {
        "type": "Feature",
1         "properties": {
          "mag": 1.6,
          -- trecho de código omitido --
2           "title": "M 1.6 - 27 km NNW of Susitna, Alaska"
        },
3         "geometry": {
          "type": "Point",
          "coordinates": [
4               -150.7585,
5               61.7591,
              56.3
          ]
        },
        "id": "ak0224bju1jx"
      },
```

A chave "properties" contém muitas informações sobre cada terremoto 1. Estamos interessados principalmente na magnitude de cada terremoto, associado à chave "mag". Estamos também

interessados no "title" de cada evento, que fornece um bom resumo de sua magnitude e localização 2.

A chave "geometry" nos ajuda a entender em que local ocorreu o terremoto 3. Precisaremos dessas informações para mapear cada evento. Podemos encontrar a longitude 4 e a latitude 5 para cada terremoto em uma lista associada à chave "coordinates".

Como esse arquivo tem mais aninhamento do que usaríamos no código que escrevemos, se parecer confuso, não se preocupe: o Python lidará com a maior parte da complexidade. Vamos trabalhar com um ou dois níveis de aninhamento por vez. Começaremos extraindo um dicionário para cada terremoto registrado no período de 24 horas.

NOTA *Quando se trata de locais, muitas vezes dizemos a latitude do local primeiro, seguido por sua longitude. Essa convenção possivelmente se originou devido ao fato de que os humanos descobriram primeiro a latitude antes de desenvolverem o conceito de longitude. No entanto, muitas abordagens geoespaciais elencam a longitude primeiro e depois a latitude, porque corresponde à convenção (*x, y*) que usamos em representações matemáticas. O formato GeoJSON segue a convenção (longitude, latitude). Caso use uma abordagem diferente, é importante aprender qual convenção adotar.*

Criando uma lista de todos os terremotos

Primeiro, criaremos uma lista com todas as informações sobre cada terremoto ocorrido.

eq_explore_data.py

```
from pathlib import Path
import json

# Lê os dados como uma string e os converte em um objeto Python
path = Path('eq_data/eq_data_1_day_m1.geojson')
contents = path.read_text()
all_eq_data = json.loads(contents)
```

```
# Examina todos os terremotos no conjunto de dados
all_eq_dicts = all_eq_data['features']
print(len(all_eq_dicts))
```

Pegamos os dados associados à chave 'features' no dicionário all_eq_data e os atribuímos a all_eq_dicts. Sabemos que esse arquivo contém registros de 160 terremotos, e a saída comprova que temos todos os terremotos no arquivo:

```
160
```

Repare como o código é pequeno. O arquivo elegantemente formatado *readable_eq_data.json* tem mais de 6.000 linhas. Mas com somente algumas linhas, é possível ler todos esses dados e armazená-los em uma lista Python. Em seguida, extrairemos as magnitudes de cada terremoto.

Extraindo magnitudes

Podemos percorrer a lista contendo os dados sobre cada terremoto com um loop e extrair qualquer informação que quisermos. Vamos extrair a magnitude de cada terremoto:

eq_explore_data.py

```
-- trecho de código omitido --
all_eq_dicts = all_eq_data['features']

mags = []
for eq_dict in all_eq_dicts:
    mag = eq_dict['properties']['mag']
    mags.append(mag)

print(mags[:10])
```

Criamos uma lista vazia para armazenar as magnitudes e, depois, percorremos com um loop a lista all_eq_dicts 1. Dentro deste loop, cada terremoto é representado pelo dicionário eq_dict. A magnitude de cada terremoto é armazenada na seção 'properties' desse dicionário, com a chave 'mag' 2. Armazenamos cada magnitude na variável mag e depois a anexamos à lista mags.

Exibimos as primeiras 10 magnitudes, assim podemos verificar se estamos obtendo os dados corretos:

```
[1.6, 1.6, 2.2, 3.7, 2.92000008, 1.4, 4.6, 4.5, 1.9, 1.8]
```

Em seguida, vamos extrair os dados da localização de cada terremoto para criarmos um mapa dos terremotos.

Extraindo os dados de localização

Os dados de localização para cada terremoto são armazenados com a chave "geometry". Dentro do dicionário da chave geometry tem uma chave "coordinates", e os dois primeiros valores dessa lista são a longitude e a latitude. Vejamos como extrair esses dados:

eq_explore_data.py

```
-- trecho de código omitido --
all_eq_dicts = all_eq_data['features']

mags, lons, lats = [], [], []
for eq_dict in all_eq_dicts:
    mag = eq_dict['properties']['mag']
1   lon = eq_dict['geometry']['coordinates'][0]
    lat = eq_dict['geometry']['coordinates'][1]
    mags.append(mag)
    lons.append(lon)
    lats.append(lat)

print(mags[:10])
print(lons[:5])
print(lats[:5])
```

Criamos listas vazias para as longitudes e latitudes. O código eq_dict['geometry'] acessa o dicionário que representa o elemento de geometria do terremoto 1. A segunda chave, 'coordinates', extrai a lista de valores associados à 'coordinates'. Por último, o índice 0 solicita o primeiro valor na lista de coordenadas, que corresponde à longitude de um terremoto.

Ao exibirmos as primeiras 5 longitudes e latitudes, a saída mostra que estamos extraindo os dados corretos:

```
[1.6, 1.6, 2.2, 3.7, 2.92000008, 1.4, 4.6, 4.5, 1.9, 1.8]
[-150.7585, -153.4716, -148.7531, -159.6267, -155.248336791992]
[61.7591, 59.3152, 63.1633, 54.5612, 18.7551670074463]
```

Com esses dados, podemos passar para o mapeamento de cada terremoto.

Criando um mapa-múndi

Usando as informações que extraímos até agora, é possível criar um mapa-múndi simples. Embora ainda não seja apresentável, queremos assegurar que as informações sejam devidamente exibidas antes de focarmos os problemas de estilo e apresentação. Vejamos o mapa inicial:

eq_world_map.py

```
from pathlib import Path
import json

import plotly.express as px

-- trecho de código omitido --
for eq_dict in all_eq_dicts:
    -- trecho de código omitido --

title = 'Global Earthquakes'
fig = px.scatter_geo(lat=lats, lon=lons, title=title)
fig.show()
```

Importamos `plotly.express` com o alias `px`, como fizemos no Capítulo 15. A função `scatter_geo()` 1 possibilita sobrepormos um gráfico de dispersão de dados geográficos em um mapa. No uso mais simples desse tipo de gráfico, basta fornecermos uma lista de latitudes e uma lista de longitudes. Passamos a lista `lats` para o argumento `lat`, e `lons` para o argumento `lon`.

Ao executarmos esse arquivo, você deve ver um mapa que se parece com o da Figura 16.7. Mais uma vez, podemos constatar o poder da biblioteca Plotly Express; com apenas três linhas de código, temos um mapa de atividades sísmicas globais.

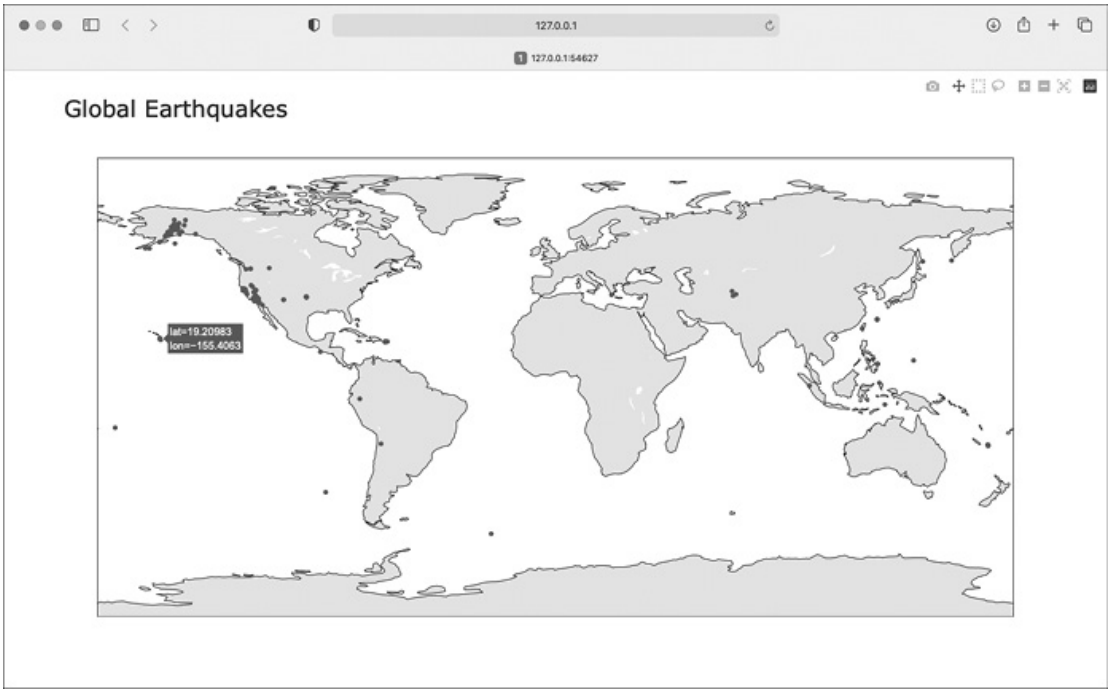


Figura 16.7: Um mapa simples mostrando a localização dos terremotos ocorridos nas últimas 24 horas.

Agora que sabemos que as informações em nosso conjunto de dados estão sendo devidamente plotadas, é possível fazer algumas alterações para que o mapa fique mais relevante e fácil de ler.

Representando magnitudes

Um mapa de atividades sísmicas deve mostrar a magnitude de cada terremoto. Podemos também incluir mais dados, agora que sabemos que os dados estão sendo corretamente plotados.

```
-- trecho de código omitido --
# Lê os dados como uma string e os converte em um objeto Python
path = Path('eq_data/eq_data_30_day_m1.geojson')
contents = path.read_text()
-- trecho de código omitido --

title = 'Global Earthquakes'
fig = px.scatter_geo(lat=lats, lon=lons, size=mags, title=title)
fig.show()
```

Carregamos o arquivo *eq_data_30_day_m1.geojson* a fim de incluir 30 dias completos de atividade sísmica. Além disso, usamos o

argumento size na chamada px.scatter_geo(), que especifica como os pontos no mapa serão dimensionados. Passamos a lista mags para size, assim os terremotos com maior magnitude aparecerão como pontos maiores no mapa.

O mapa resultante é mostrado na Figura 16.8. Em geral, os terremotos ocorrem perto dos limites das placas tectônicas, e o período mais longo de atividade do terremoto incluído nesse mapa revela as localizações exatas desses limites.

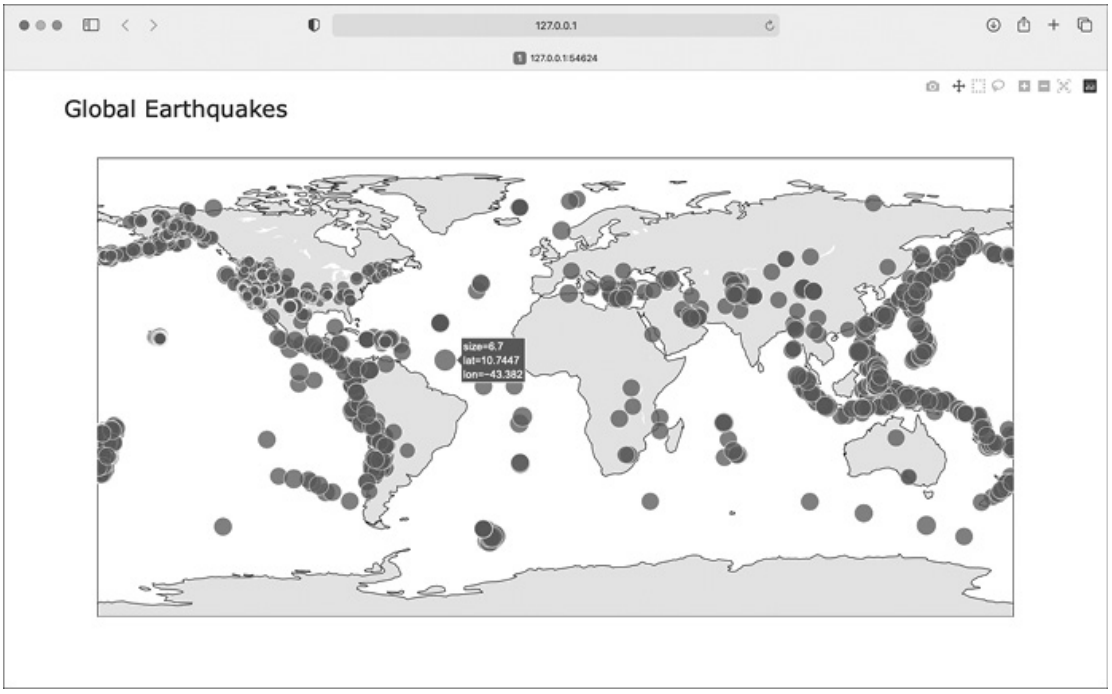


Figura 16.8: Agora, o mapa mostra a magnitude de todos os terremotos nos últimos 30 dias.

Apesar desse mapa ser melhor apresentável, ainda é difícil identificar quais pontos representam os terremotos mais significativos. É possível melhorá-lo mais usando cores para representar também as magnitudes.

Personalizando as cores das marcações

Podemos utilizar as escalas de cores do Plotly para personalizar a cor de cada marcação, conforme a gravidade do terremoto correspondente. Utilizaremos também uma projeção diferente para o

mapa base.

eq_world_map.py

```
  -- trecho de código omitido --
fig = px.scatter_geo(lat=lats, lon=lons, size=mags, title=title,
1       color=mags,
2       color_continuous_scale='Viridis',
3       labels={'color':'Magnitude'},
4       projection='natural earth',
    )
fig.show()
```

Aqui, todas as mudanças significativas ocorrem na chamada de função `px.scatter_geo()`. O argumento `color` informa ao Plotly quais valores usar para determinar a correspondência de cada marcação com a escala de cores `1`. Usamos a lista `mags` a fim de determinar a cor de cada ponto, assim como fizemos com o argumento `size`.

O argumento `color_continuous_scale` informa ao Plotly qual escala de cor usar `2`. *Viridis* é uma escala de cores que varia do azul escuro ao amarelo vivo, e funciona bem com esse conjunto de dados. Por padrão, a escala de cores à direita do mapa é rotulada como *color*, não representando o que as cores realmente significam. O argumento `labels`, mostrado no Capítulo 15, aceita um dicionário como valor `3`. Nesse gráfico, é necessário apenas definir um rótulo personalizado, assegurando que a escala de cores esteja rotulada como *Magnitude* em vez de *color*.

Adicionamos mais um argumento a fim de modificar o mapa base sobre o qual os terremotos são plotados. O argumento `projection` aceita uma série de projeções de mapa comuns `4`. Aqui, usamos a projeção `'natural earth'`, que arredonda as extremidades do mapa. Além do mais, observe a vírgula final após o último argumento. Quando uma chamada de função tem uma lista extensa de argumentos abrangendo diversas linhas como essa, é prática comum adicionar uma vírgula à direita para que posteriormente possamos adicionar outro argumento na próxima linha.

Agora, ao executarmos o programa, veremos um mapa mais

elegante. Na Figura 16.9, a escala de cores mostra a gravidade dos terremotos individuais; os terremotos mais graves se destacam como pontos amarelo-claros, contrastando com muitos pontos mais escuros. Podemos também identificar quais regiões do mundo têm atividade sísmica mais significativa.

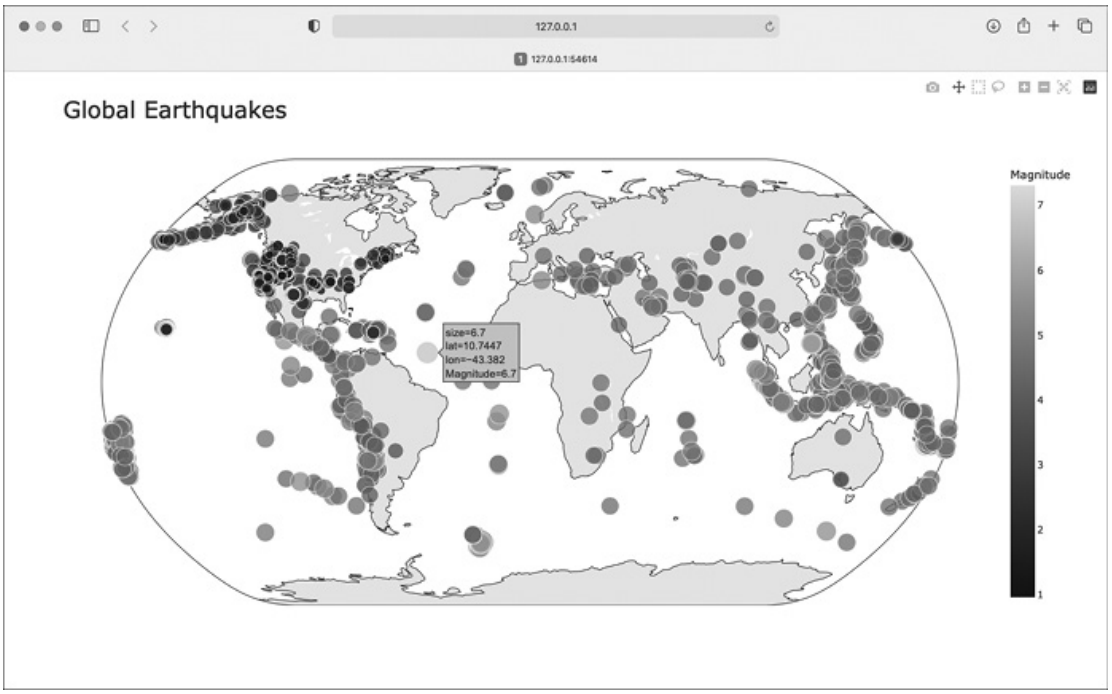


Figura 16.9: Em 30 dias de terremotos, cor e tamanho são usados para representar a magnitude de cada um deles.

Outras escalas de cores

Podemos escolher entre uma série de outras escalas de cores. Para ver as escalas de cores disponíveis, insira as duas linhas a seguir em uma sessão de terminal Python:

```
>>> import plotly.express as px
>>> px.colors.named_colorscales()
['aggrnyl', 'agsunset', 'blackbody', ..., 'mygbm']
```

Sinta-se à vontade para testar todas essas escalas de cores com o mapa do terremoto ou com qualquer conjunto de dados em que cores continuamente variadas possam ajudar a destacar padrões nos dados.

Adicionando texto flutuante

A fim de concluir esse mapa, adicionaremos um pouco de texto informativo que aparecerá quando passarmos o mouse sobre a marcação que representa um terremoto. Além de mostrar a longitude e a latitude, que aparecem por padrão, mostraremos a magnitude e forneceremos também uma descrição da localização aproximada.

Para fazer essa alteração, é necessário extrair um pouco mais de dados do arquivo:

eq_world_map.py

```
    -- trecho de código omitido --
1 mags, lons, lats, eq_titles = [], [], [], []
    mag = eq_dict['properties']['mag']
    lon = eq_dict['geometry']['coordinates'][0]
    lat = eq_dict['geometry']['coordinates'][1]
2   eq_title = eq_dict['properties']['title']
    mags.append(mag)
    lons.append(lon)
    lats.append(lat)
    eq_titles.append(eq_title)

  title = 'Global Earthquakes'
  fig = px.scatter_geo(lat=lats, lon=lons, size=mags, title=title,
        -- trecho de código omitido --
        projection='natural earth',
3       hover_name=eq_titles,
    )
  fig.show()
```

Primeiro, criamos uma lista chamada `eq_titles` para armazenar o título de cada terremoto `1`. A seção `'title'` dos dados contém um nome descritivo da magnitude e localização de cada terremoto, além de sua longitude e sua latitude. Extraímos essa informação e a atribuímos à variável `eq_title` `2` e, em seguida, a adicionamos à lista `eq_titles`.

Na chamada `px.scatter_geo()`, passamos `eq_titles` para o argumento `hover_name` `3`. Agora o Plotly adicionará as informações do título de

cada terremoto ao texto flutuante em cada ponto. Ao executarmos esse programa, devemos conseguir passar o mouse sobre cada marcação, ver uma descrição onde esse terremoto ocorreu e ler sua magnitude exata. Um exemplo dessas informações é mostrado na Figura 16.10.

Isso é incrível! Com menos de 30 linhas de código, criamos um mapa visualmente agradável e significativo das atividades sísmicas globais que também ilustra a estrutura geológica do planeta. O Plotly disponibiliza uma ampla variedade de formas para personalizar a aparência e o comportamento de visualizações. Com as muitas opções do Plotly, é possível criar gráficos e mapas que mostram exatamente o que queremos.

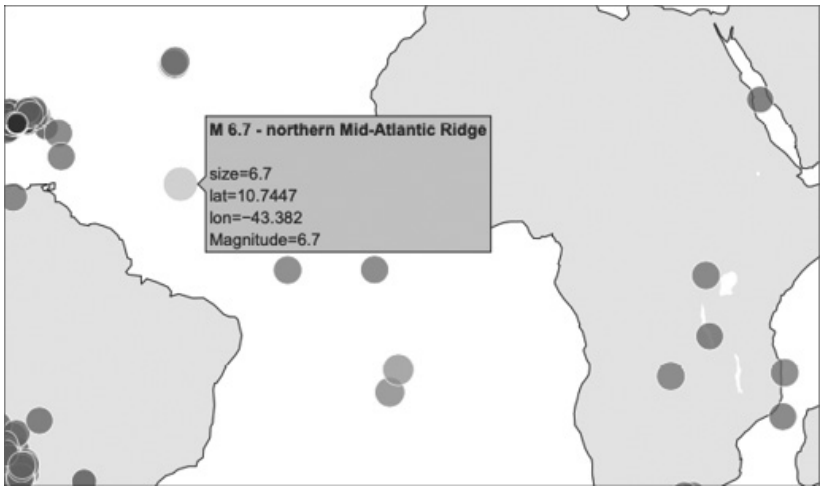


Figura 16.10: Agora, o texto flutuante inclui um resumo de cada terremoto.

FAÇA VOCÊ MESMO

16.6 Refatoração: O loop que extrai dados de all_eq_dicts usa variáveis para a magnitude, longitude, latitude e título de cada terremoto antes de adicionar esses valores às listas adequadas. Escolhi essa abordagem para explicar como extrair dados de um arquivo GeoJSON, não sendo necessária em seu código. Em vez de usar essas variáveis temporárias, extraia cada valor de eq_dict e os adicione à lista apropriada em uma linha. Isso deve reduzir o corpo desse loop em somente quatro linhas.

16.7 Título automatizado: Nesta seção, utilizamos o título genérico *Global Earthquakes*. Em vez disso, use o título para o conjunto de dados na parte de metadados do arquivo GeoJSON. Extraia esse valor e o atribua à variável title.

16.8 Terremotos recentes: É possível encontrar arquivos de dados online contendo

informações sobre os terremotos mais recentes em períodos de 1 hora, 1 dia, 7 dias e 30 dias. Acesse https://earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php e veja uma lista de links com conjuntos de dados de vários períodos de tempo, com foco em terremotos de diferentes magnitudes. Faça o dowload de um desses conjuntos de dados e crie uma visualização das atividades sísmicas mais recentes.

16.9 Incêndios pelo mundo: Nos recursos deste capítulo, você encontrará um arquivo chamado *world_fires_1_day.csv*. Esse arquivo contém informações sobre incêndios em diferentes locais ao redor do globo, incluindo a latitude, a longitude e a intensidade de cada incêndio. Recorra ao processamento de dados da primeira parte deste capítulo e ao mapeamento desta seção para criar um mapa que mostre quais partes do mundo são afetadas por incêndios.

Você pode fazer o download das versões mais recentes desses dados em https://earthdata.nasa.gov/earth-observation-data/near-real-time/firms/active-fire-data. É possível encontrar links para os dados no formato CSV na seção *SHP, KML, and TXT Files*.

Recapitulando

Neste capítulo, aprendemos a trabalhar com conjuntos de dados do mundo real. Processamos arquivos CSV e GeoJSON e extraímos os dados desejados. Com dados meteorológicos históricos, aprendemos mais sobre como trabalhar com a Matplotlib, bem como usar o módulo datetime e como plotar diversas séries de dados em um gráfico. Plotamos dados geográficos em um mapa-múndi com o Plotly e aprendemos a personalizar o estilo do mapa.

À medida que adquire experiência trabalhando com arquivos CSV e JSON, você será capaz de processar quase todos os dados que queira analisar. É possível fazer o download da maioria dos conjuntos de dados online em um ou ambos esses formatos. Ao trabalhar com esses formatos, é possível também aprender a trabalhar com outros formatos de dados com mais facilidade.

No próximo capítulo, desenvolveremos programas que coletam automaticamente os próprios dados de fontes online e, em seguida, criaremos visualizações desses dados. Caso queira programar como hobby, essas habilidades são divertidas. Agora, caso esteja interessado em se tornar um programador profissional, essas habilidades são essenciais.

CAPÍTULO 17

Trabalhando com APIs

Neste capítulo, vamos aprender a escrever um programa independente que gera uma visualização com base nos dados que acessa. Nosso programa usará uma *Interface de Programação de Aplicações (API)* para requisitar automaticamente informações específicas de um site e, em seguida, usar essas informações para gerar uma visualização. Como os programas escritos dessa maneira sempre usam dados atuais para gerar visualizações, ainda que esses dados mudem em um ritmo acelerado, essas visualizações sempre serão atualizadas.

Usando uma API

Uma API é a parte de um site desenvolvido para interagir com programas. Por sua vez, esses programas usam URLs bastante específicos para requisitar determinadas informações. Esse tipo de requisição é denominado como *API*. Os dados solicitados serão retornados em um formato de fácil processamento, como JSON ou CSV. A maioria das aplicações que utilizam fontes externas de dados, como apps que se integram a sites de mídia social, depende de chamadas de API.

Git e GitHub

Basearemos nossa visualização em informações do GitHub (https://github.com), site que possibilita aos programadores colaborarem com projetos de programação. Vamos utilizar a API do GitHub para requisitar informações sobre projetos Python do site e, em seguida,

vamos gerar uma visualização interativa da popularidade relativa desses projetos usando o Plotly.

O GitHub deve seu nome ao Git, sistema de controle de versão/versionamento distribuído. Como o Git ajuda as pessoas a gerenciarem as próprias tarefas em um projeto, as mudanças efetuadas por uma pessoa não interferem nas alterações que outras pessoas estão fazendo. Quando implementamos uma funcionalidade nova em um projeto, o Git rastreia as alterações feitas em cada arquivo. Assim que o código novo estiver funcionando, fazemos o *commit* das mudanças feitas, e o Git registra o novo estado do projeto. Se cometer um erro e quiser reverter suas mudanças, é possível retornar facilmente a qualquer estado anterior da tarefa. (Para saber mais sobre o controle de versão usando o Git, confira o Apêndice D.) No GitHub, os projetos são armazenados em *repositórios*, que contêm tudo associado ao projeto: o código, informações sobre os colaboradores, quaisquer problemas ou relatórios de bugs, e assim por diante.

Quando os usuários do GitHub gostam de um projeto, podem marcá-lo com uma "estrela (starred)" para mostrar apoio e acompanhar os projetos que talvez queiram usar. Neste capítulo, desenvolveremos um programa para fazer o download automático de informações sobre os projetos Python com mais estrelas no GitHub e, depois, criaremos uma visualização informativa desses projetos.

Requisitando dados com uma chamada de API

A API do GitHub possibilita que solicitemos uma ampla gama de informações por meio de chamadas de API. Para vermos como é uma chamada de API, digite o seguinte na barra de endereço do navegador e pressione a tecla ENTER:

https://api.github.com/search/repositories?q=language:python+sort:stars

Essa chamada retorna o número de projetos Python atualmente

hospedados no GitHub, bem como informações sobre os repositórios Python mais populares. Vamos examinar essa chamada. A primeira parte, https://api.github.com/, direciona a requisição para a parte do GitHub que responde às chamadas da API. A próxima parte, search/repositories, instrui a API a realizar uma pesquisa em todos os repositórios no GitHub.

O ponto de interrogação após repositories sinaliza que estamos prestes a passar um argumento. O q significa *query*, e o sinal de igual (=) nos possibilita começar especificando uma query (q=). Quando utilizamos language:python, indicamos que queremos informações apenas sobre repositórios que tenham o Python como a linguagem principal. A parte final, +sort:stars, classifica os projetos pelo número de estrelas recebidas.

O trecho a seguir mostra as primeiras linhas da resposta:

```
  {
1   "total_count": 8961993,
2   "incomplete_results": true,
3   "items": [
      {
        "id": 54346799,
        "node_id": "MDEwOlJlcG9zaXRvcnk1NDM0Njc5OQ==",
        "name": "public-apis",
        "full_name": "public-apis/public-apis",
        -- trecho de código omitido --
```

Podemos observar pela resposta que esse URL não foi criado para ser inserido por humanos, pois está em um formato que deve ser processado por um programa. O GitHub encontrou pouco menos de nove milhões de projetos Python a partir dessa linha 1. Como o valor para "incomplete_results" é true, significa que GitHub não processou totalmente a query 2. O GitHub limita o tempo de execução de cada query a fim de que a API seja responsiva para todos os usuários. Nesse caso, o Github encontrou alguns dos repositórios Python mais populares, mas não teve tempo de encontrar todos; daqui a pouco vamos corrigir isso. Os "items" retornados são exibidos na lista a seguir, que contém detalhes sobre os projetos Python mais

populares no GitHub 3.

Instalando o pacote Requests

O pacote *Requests* possibilita que um programa Python requisite facilmente informações de um site e examine a resposta. Use o pip para instalar o pacote Requests:

```
$ python -m pip install --user requests
```

Caso utilize um comando diferente de **python** para executar programas ou iniciar uma sessão de terminal, como **python3**, o comando será assim:

```
$ python3 -m pip install --user requests
```

Processando uma resposta de API

Agora, vamos escrever um programa para executar automaticamente uma chamada de API e processar os resultados:

python_repos.py

```
  import requests

  # Cria uma chamada de API e verifica a resposta
1 url = "https://api.github.com/search/repositories"
  url += "?q=language:python+sort:stars+stars:>10000"

2 headers = {"Accept": "application/vnd.github.v3+json"}
3 r = requests.get(url, headers=headers)
4 print(f"Status code: {r.status_code}")

  # Converte o objeto de resposta em um dicionário
5 response_dict = r.json()

  # Processa os resultados
  print(response_dict.keys())
```

Primeiro, importamos o módulo requests. Depois, atribuímos o URL da chamada de API à variável url 1. Como se trata de um URL extenso, o dividimos em duas linhas. A primeira linha é a parte principal do URL, e a segunda linha é a string da query. Incluímos mais uma condição na string da query original: stars:>10000, que informa ao

GitHub para procurar somente repositórios Python que tenham mais de 10.000 estrelas. Isso deve possibilitar que o GitHub retorne um conjunto completo e consistente de resultados.

Atualmente, o GitHub está na terceira versão de sua API, por isso, definimos cabeçalhos para a chamada de API que solicitam explicitamente para usar essa versão da API e retornamos os resultados no formato JSON 2. Em seguida, utilizamos requests a fim de criar a chamada para a API 3. Chamamos get() e passamos o URL e o cabeçalho que definimos, e atribuímos o objeto de resposta à variável r.

O objeto de resposta tem um atributo chamado status_code, que nos informa se a requisição foi bem-sucedida. (Um status code de 200 sinaliza uma resposta bem-sucedida.) Exibimos o valor de status_code para garantir que a chamada foi realizada com sucesso 4. Solicitamos à API para retornar as informações no formato JSON, depois usamos o método json() para converter as informações em um dicionário Python 5. Atribuímos o dicionário resultante a response_dict.

Por último, exibimos as chaves de response_dict e vemos a seguinte saída:

```
Status code: 200
dict_keys(['total_count', 'incomplete_results', 'items'])
```

Como o status code é 200, sabemos que a requisição foi bem-sucedida. O dicionário de respostas contém apenas três chaves: 'total_count', 'incomplete_results' e 'items'. Vamos conferir esse dicionário de respostas.

Trabalhando com o dicionário de resposta

Com as informações da chamada de API representada como um dicionário, podemos trabalhar com os dados armazenados dentro dele. Geraremos uma saída que resuma as informações. É uma boa forma de garantir que recebemos as informações que esperávamos e começar a examinar aquelas em que estamos interessados:

python_repos.py

```
import requests

# Cria uma chamada de API e armazena a resposta
-- trecho de código omitido --

# Converte o objeto de resposta em um dicionário
response_dict = r.json()
print(f"Total repositories: {response_dict['total_count']}")
print(f"Complete results: {not response_dict['incomplete_results']}")

# Explora informações sobre os repositórios
repo_dicts = response_dict['items']
print(f"Repositories returned: {len(repo_dicts)}")

# Examina o primeiro repositório
repo_dict = repo_dicts[0]
print(f"\nKeys: {len(repo_dict)}")
for key in sorted(repo_dict.keys()):
    print(key)
```

Começamos explorando o dicionário de respostas ao exibir o valor associado a `'total_count'`, que representa o número total de repositórios Python retornados por essa chamada de API `1`. Utilizamos também o valor associado a `'incomplete_results'`. Desse modo, saberemos se o GitHub conseguiu processar a query por completo. Em vez de exibir esse valor diretamente, exibimos seu oposto: um valor de `True` indicará que recebemos um conjunto completo de resultados.

O valor associado a `'items'` é uma lista contendo muitos dicionários, sendo que cada um deles contém dados sobre um repositório individual Python. Atribuímos essa lista de dicionários a `repo_dicts` `2`. Depois, exibimos o tamanho de `repo_dicts` para ver de quantos repositórios temos informações.

A fim de analisar com mais detalhe as informações retornadas sobre cada repositório, extraímos o primeiro item de `repo_dicts` e atribuímos a `repo_dict` `3`. Em seguida, exibimos o número de chaves do dicionário para verificar quanta informação temos `4`. Por último, exibimos todas as chaves do dicionário para vermos quais tipos de informação estão incluídas `5`.

Os resultados nos fornecem uma ideia mais precisa sobre os dados reais:

```
  Status code: 200
1 Total repositories: 248
2 Complete results: True
  Repositories returned: 30

3 Keys: 78
  allow_forking
  archive_url
  archived
  -- trecho de código omitido --
  url
  visiblity
  watchers
  watchers_count
```

No momento em que eu escrevia este livro, existiam apenas 248 repositórios Python com mais de 10.000 estrelas 1. Podemos ver que o GitHub foi capaz de processar totalmente a chamada de API 2. Nessa resposta, o GitHub retornou informações sobre os primeiros 30 repositórios que correspondem às condições de nossa query. Se quisermos mais repositórios, é possível requisitar páginas adicionais de dados.

A API do GitHub retorna muitas informações sobre cada repositório: existem 78 chaves no repo_dict 3. Ao examinarmos essas chaves, temos uma noção do tipo de informação que podemos extrair sobre um projeto. (A única forma de saber quais informações estão disponíveis por meio de uma API é ler a documentação ou examinar as informações por meio de código, como estamos fazendo aqui.)

Extrairemos os valores para algumas das chaves do repo_dict:

python_repos.py

```
  -- trecho de código omitido --
  # Examina o primeiro repositório
  repo_dict = repo_dicts[0]

  print("\nSelected information about first repository:")
1 print(f"Name: {repo_dict['name']}")
```

```
print(f"Owner: {repo_dict['owner']['login']}")
print(f"Stars: {repo_dict['stargazers_count']}")
print(f"Repository: {repo_dict['html_url']}")
print(f"Created: {repo_dict['created_at']}")
print(f"Updated: {repo_dict['updated_at']}")
print(f"Description: {repo_dict['description']}")
```

Aqui, exibimos os valores para um número de chaves do primeiro dicionário do repositório. Começamos com o nome do projeto `1`. Um dicionário inteiro representa o proprietário do projeto, logo usamos a chave `owner` para acessar o dicionário que representa o proprietário e, em seguida, usamos a chave `login` para obter o nome de login do proprietário `2`. Em seguida, exibimos quantas estrelas o projeto ganhou `3` e o URL do repositório GitHub do projeto. Então, mostramos quando foi criado `4` e quando foi atualizado pela última vez `5`. Por último, exibimos a descrição do repositório.

A saída deve ser mais ou menos assim:

```
Status code: 200
Total repositories: 248
Complete results: True
Repositories returned: 30

Selected information about first repository:
Name: public-apis
Owner: public-apis
Stars: 191493
Repository: https://github.com/public-apis/public-apis
Created: 2016-03-20T23:49:42Z
Updated: 2022-05-12T06:37:11Z
Description: A collective list of free APIs
```

É possível constatar que o projeto Python com mais estrelas no GitHub (quando escrevi este livro) é o *public-apis*. O proprietário é uma organização com o mesmo nome, e recebeu estrelas de 200.000 usuários do GitHub. Podemos ver o URL do repositório do projeto, a data de sua criação, março de 2016, e que foi atualizado recentemente. Além disso, a descrição nos reporta que *public-apis* contém uma lista de APIs gratuitas que podem interessar aos programadores.

Resumindo os repositórios mais importantes

Ao criarmos uma visualização para esses dados, queremos incluir mais de um repositório. Vamos escrever um loop a fim de exibir as informações selecionadas sobre cada repositório que a chamada de API retorna, para que, assim, possamos incluir todos na visualização:

python_repos.py

```
-- trecho de código omitido --
# Explora informações sobre os repositórios
repo_dicts = response_dict['items']
print(f"Repositories returned: {len(repo_dicts)}")

print("\nSelected information about each repository:")
for repo_dict in repo_dicts:
    print(f"\nName: {repo_dict['name']}")
    print(f"Owner: {repo_dict['owner']['login']}")
    print(f"Stars: {repo_dict['stargazers_count']}")
    print(f"Repository: {repo_dict['html_url']}")
    print(f"Description: {repo_dict['description']}")
```

Primeiro, exibimos uma mensagem introdutória 1. Depois, percorremos com um loop todos os dicionários em repo_dicts 2. Dentro do loop, exibimos o nome de cada projeto, seu proprietário, quantas estrelas tem, seu URL no GitHub e a descrição do projeto:

```
Status code: 200
Total repositories: 248
Complete results: True
Repositories returned: 30

Selected information about each repository:

Name: public-apis
Owner: public-apis
Stars: 191494
Repository: https://github.com/public-apis/public-apis
Description: A collective list of free APIs

Name: system-design-primer
Owner: donnemartin
Stars: 179952
```

```
Repository: https://github.com/donnemartin/system-design-primer
Description: Learn how to design large-scale systems. Prep for the system
  design interview.  Includes Anki flashcards.
-- trecho de código omitido --

Name: PayloadsAllTheThings
Owner: swisskyrepo
Stars: 37227
Repository: https://github.com/swisskyrepo/PayloadsAllTheThings
Description: A list of useful payloads and bypass for Web Application Security
  and Pentest/CTF
```

Como projetos interessantes aparecem nesses resultados, talvez valha a pena conferir alguns deles. Mas não fique muito tempo fazendo isso, pois estamos prestes a criar uma visualização que facilitará e muito a legibilidade dos resultados.

Monitoramento os limites da taxa de requisições da API

A maioria das APIs tem *limites de taxa*, ou seja: tem um limite de quantas requisições podemos fazer em um determinado período de tempo. Para verificar se estamos nos aproximando dos limites do GitHub, digite https://api.github.com/rate_limit em um navegador web. Você deve ver uma resposta que começa assim:

```
{
  "resources": {
    -- trecho de código omitido --
    "search": {
      "limit": 10,
      "remaining": 9,
      "reset": 1652338832,
      "used": 1,
      "resource": "search"
    },
    -- trecho de código omitido --
```

As informações que nos interessam são o limite de taxa para a search API 1. Vemos que o limite é de 10 requisições por minuto 2 e que temos 9 requisições restantes para o minuto atual 3. O valor associado à chave "reset" representa o tempo do *Unix* ou tempo da

época (o número de segundos desde a meia-noite de 1º de janeiro de 1970) quando nossa cota será redefinida [4]. Caso atinja o limite de sua cota, você receberá uma resposta curta informando que atingiu o limite da API. Se atingir o limite, espere até que sua cota seja redefinida.

NOTA *Muitas APIs exigem o registro e obtenção de uma chave de API ou token de acesso para fazer chamadas de API. No momento em que eu escrevia este livro, o GitHub ainda não tinha essa exigência, mas se obtiver um token de acesso, os limites serão maiores.*

Visualizando repositórios com o Plotly

Criaremos uma visualização usando os dados que coletamos para mostrar a popularidade relativa dos projetos Python no GitHub. Desenvolveremos um gráfico de barras interativo: a altura de cada barra representará o número de estrelas que o projeto ganhou e poderemos clicar no rótulo da barra para acessar a página inicial desse projeto no GitHub.

Salve uma cópia do programa em que estamos trabalhando como *python_repos_visual.py* e o modifique da seguinte forma:

python_repos_visual.py

```
import requests
import plotly.express as px

# Cria uma chamada de API e verifica a resposta
url = "https://api.github.com/search/repositories"
url += "?q=language:python+sort:stars+stars:>10000"

headers = {"Accept": "application/vnd.github.v3+json"}
r = requests.get(url, headers=headers)
print(f"Status code: {r.status_code}")

# Processa os resultados gerais
response_dict = r.json()
print(f"Complete results: {not response_dict['incomplete_results']}")
```

```
    # Processa as informações do repositório
    repo_dicts = response_dict['items']
3   repo_names, stars = [], []
    for repo_dict in repo_dicts:
        repo_names.append(repo_dict['name'])
        stars.append(repo_dict['stargazers_count'])

    # Cria a visualização
4   fig = px.bar(x=repo_names, y=stars)
    fig.show()
```

Importamos o Plotly Express e, em seguida, criamos a chamada de API como temos feito. Continuamos exibindo o status da resposta de chamada da API para sabermos se existe um problema 1. Ao processarmos os resultados gerais, continuamos exibindo a mensagem confirmando que recebemos um conjunto completo de resultados 2. Removemos o resto das chamadas print(), já que não estamos mais na fase exploratória; sabemos que temos os dados que queremos.

Em seguida, criamos duas listas vazias 3 para armazenar os dados que incluiremos no gráfico inicial. Precisaremos do nome de cada projeto para rotular as barras (repo_names) e o número de estrelas a fim de determinar a altura das barras (stars). No loop, anexamos o nome de cada projeto e o número de estrelas que cada um tem nessas listas.

Criamos a visualização inicial com apenas duas linhas de código 4. Isso é compatível com a filosofia do Plotly Express de que devemos conseguir ver a visualização o mais rápido possível antes de refinar sua aparência. Aqui, usamos a função px.bar() para criar um gráfico de barras. Passamos a lista repo_names como argumento x e stars como argumento y.

A Figura 17.1 mostra o gráfico resultante. É possível constatar que os primeiros projetos são significativamente mais populares do que os outros, apesar de todos serem projetos importantes no ecossistema Python.

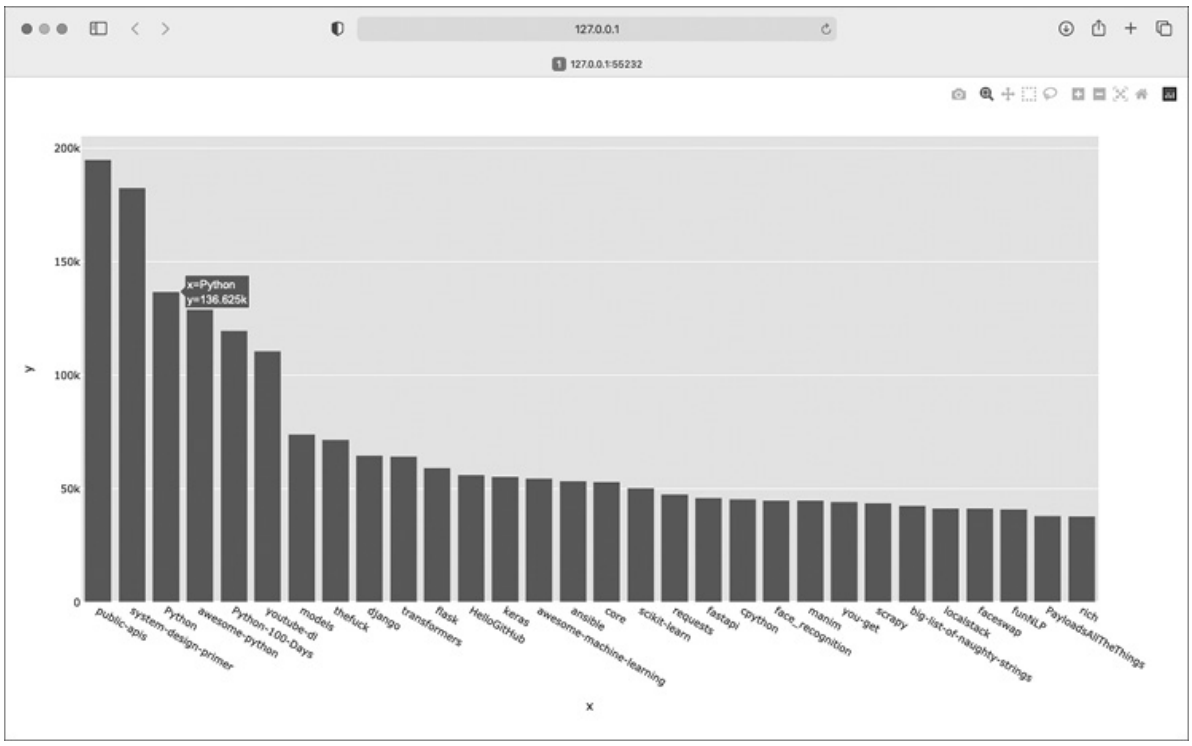


Figura 17.1: Os projetos Python com mais estrelas no GitHub.

Estilizando o gráfico

Uma vez que sabemos que as informações no gráfico estão corretas, o Plotly suporta diversas maneiras de estilizar e personalizar os gráficos. Faremos algumas alterações na chamada `px.bar()` inicial e, em seguida, faremos alguns ajustes adicionais no objeto `fig` depois de criá-lo.

Começaremos a estilizar o gráfico adicionando um título e rótulos para cada eixo:

python_repos_visual.py

```
-- trecho de código omitido --
# Cria a visualização
title = "Most-Starred Python Projects on GitHub"
labels = {'x': 'Repository', 'y': 'Stars'}
fig = px.bar(x=repo_names, y=stars, title=title, labels=labels)

fig.update_layout(title_font_size=28, xaxis_title_font_size=20,
    yaxis_title_font_size=20)
```

```
fig.show()
```

Primeiro, adicionamos um título e rótulos para cada eixo, como fizemos nos capítulos 15 e 16. Em seguida, usamos o método `fig.update_layout()` a fim de modificar elementos específicos do gráfico `1`. O Plotly adota a convenção em que os aspectos de um elemento gráfico são conectados por underscores. À medida que se familiarizar com a documentação do Plotly, você começará a identificar padrões consistentes em como os diferentes elementos de um gráfico são nomeados e modificados. Nesse caso, definimos o tamanho da fonte do título como `28` e o tamanho da fonte para cada título do eixo como `20`. O resultado é mostrado na Figura 17.2.

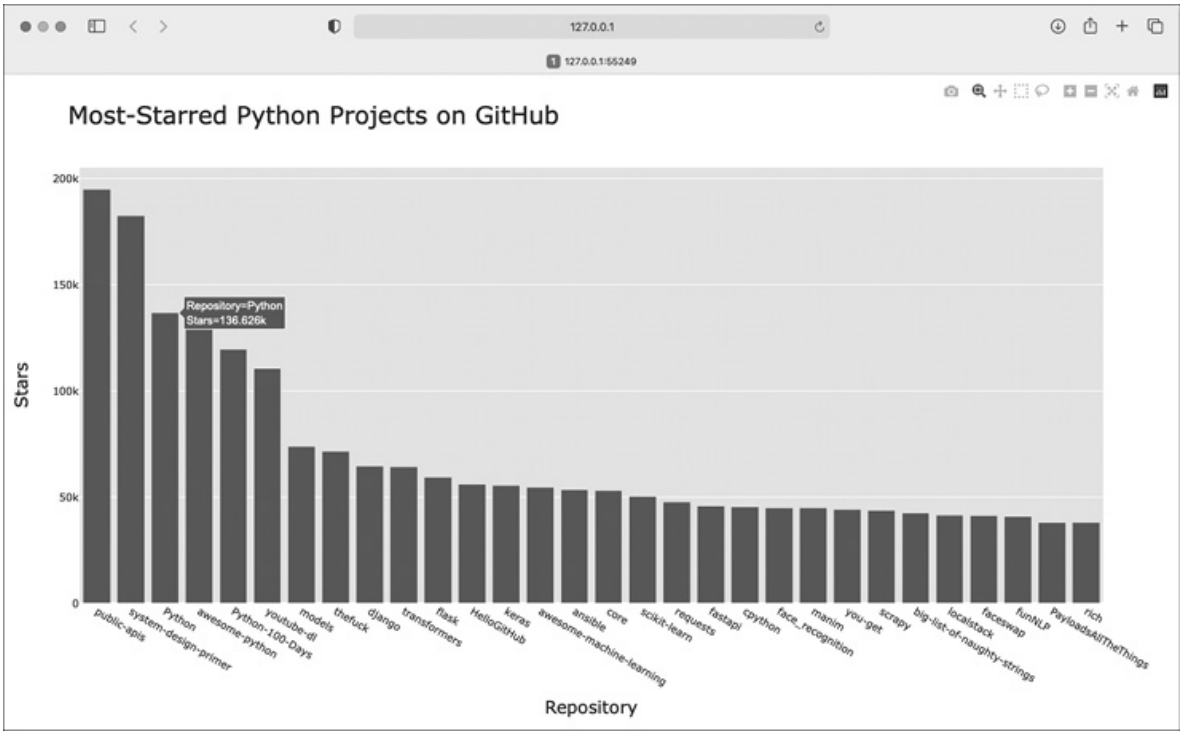


Figura 17.2: Um título foi adicionado ao gráfico principal e a cada eixo também.

Adicionando tooltips personalizadas

No Plotly, podemos passar o cursor sobre uma barra individual para mostrar as informações que a barra representa. Em geral, chamamos isso de *tooltip* (dica de contexto) e, aqui, as tooltips

mostram o número de estrelas que um projeto tem. Vamos criar uma tooltip personalizada para mostrar a descrição e o proprietário de cada projeto.

Será necessário extrair alguns dados adicionais para gerá-las:

python_repos_visual.py

```
  -- trecho de código omitido --
  # Processa as informações do repositório
  repo_dicts = response_dict['items']
1 repo_names, stars, hover_texts = [], [], []
  for repo_dict in repo_dicts:
      repo_names.append(repo_dict['name'])
      stars.append(repo_dict['stargazers_count'])

      # Cria textos flutuantes
2     owner = repo_dict['owner']['login']
      description = repo_dict['description']
3     hover_text = f"{owner}<br />{description}"
      hover_texts.append(hover_text)

  # Cria a visualização
  title = "Most-Starred Python Projects on GitHub"
  labels = {'x': 'Repository', 'y': 'Stars'}
4 fig = px.bar(x=repo_names, y=stars, title=title, labels=labels,
          hover_name=hover_texts)

  fig.update_layout(title_font_size=28, xaxis_title_font_size=20,
          yaxis_title_font_size=20)

  fig.show()
```

De início, definimos uma nova lista vazia, `hover_texts`, para armazenar o texto que queremos exibir em cada projeto `1`. No loop em que processamos os dados, extraímos o proprietário e a descrição para cada projeto `2`. Já que o Plotly possibilita que usemos código HTML dentro dos elementos de texto, geramos uma string para o rótulo com uma quebra de linha (`<br />`) entre o nome de usuário do proprietário do projeto e a descrição `3`. Em seguida, anexamos esse rótulo à lista `hover_texts`.

Na chamada `px.bar()`, adicionamos o argumento `hover_name` e o

passamos para hover_texts 4. É a mesma abordagem que usamos para personalizar o rótulo para cada ponto no mapa de atividades sísmicas globais. Como cria cada barra, o Plotly extraí os rótulos dessa lista e apenas os exibe quando o usuário passa o mouse sobre as barras. A Figura 17.3 mostra uma tooltip personalizada.

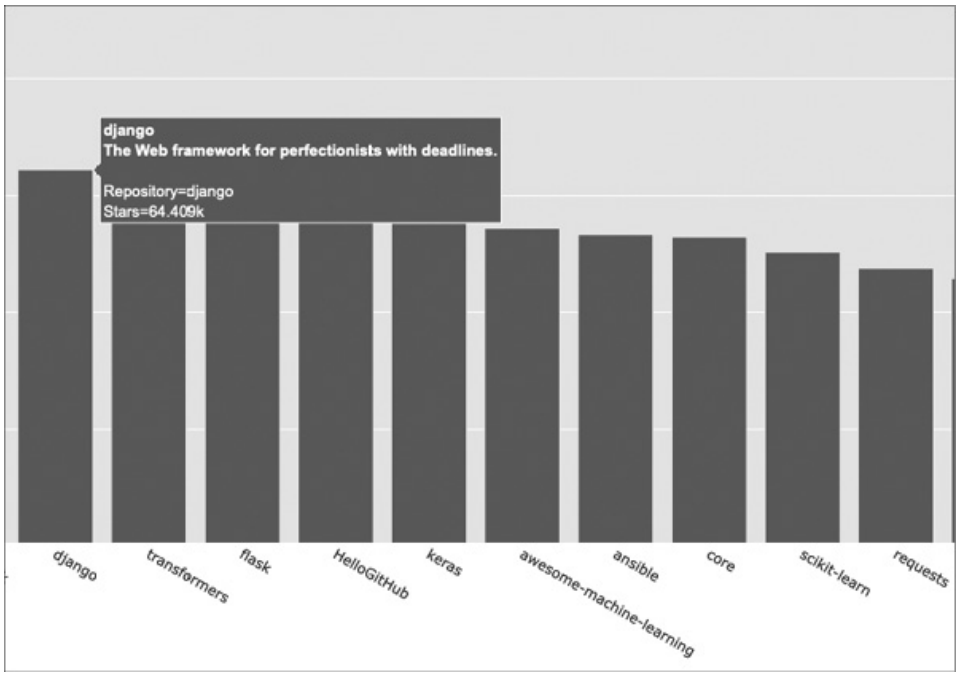


Figura 17.3: Passar o mouse sobre uma barra mostra o proprietário e a descrição do projeto.

Adicionando links clicáveis

Como o Plotly possibilita que usemos HTML em elementos de texto, podemos facilmente adicionar links a um gráfico. Usaremos os rótulos do eixo *x* como forma de possibilitar que o usuário visite a página inicial de qualquer projeto no GitHub. Precisamos extrair os URLs dos dados e usá-los quando gerarmos os rótulos do eixo *x*:

python_repos_visual.py

```
    -- trecho de código omitido --
    # Processa as informações do repositório
    repo_dicts = response_dict['items']
1   repo_links, stars, hover_texts = [], [], []
    for repo_dict in repo_dicts:
```

```
        # Transforma os nomes dos repositórios em links ativos
        repo_name = repo_dict['name']
2       repo_url = repo_dict['html_url']
3       repo_link = f"<a href='{repo_url}'>{repo_name}</a>"
        repo_links.append(repo_link)

        stars.append(repo_dict['stargazers_count'])
        -- trecho de código omitido --

    # Cria a visualização
    title = "Most-Starred Python Projects on GitHub"
    labels = {'x': 'Repository', 'y': 'Stars'}
    fig = px.bar(x=repo_links, y=stars, title=title, labels=labels,
            hover_name=hover_texts)

    fig.update_layout(title_font_size=28, xaxis_title_font_size=20,
            yaxis_title_font_size=20)

    fig.show()
```

Atualizamos o nome da lista que estamos criando de `repo_names` para `repo_links`. Assim, reportamos com mais acurácia o tipo de informação que estamos consolidando para o gráfico `1`. Depois, extraímos o URL do projeto de `repo_dict` e o atribuímos à variável temporária `repo_url` `2`. Em seguida, geramos um link para o projeto `3`. Usamos a tag âncora HTML, que tem o formato `<a href='URL'>link text</a>` para gerar o link. Em seguida, adicionamos esse link a `repo_links`.

Quando chamamos `px.bar()`, usamos `repo_links` para os valores *x* no gráfico. Apesar de o resultado parecer o mesmo de antes, agora o usuário pode clicar em qualquer um dos nomes do projeto na parte inferior do gráfico para visitar a página inicial do projeto no GitHub. Temos uma visualização interativa e informativa dos dados acessados por meio de uma API!

Personalizando as cores das marcações

Após ser criado, quase todos os aspectos do gráfico podem ser personalizado por meio de um método de atualização. Já usamos o método `update_layout()` antes. Outro método, `update_traces()`, pode ser utilizado para personalizar os dados que são representados em um

gráfico.

Modificaremos as barras para um azul mais escuro e adicionaremos um pouco de transparência:

```
-- trecho de código omitido --
fig.update_layout(title_font_size=28, xaxis_title_font_size=20,
        yaxis_title_font_size=20)

fig.update_traces(marker_color='SteelBlue', marker_opacity=0.6)

fig.show()
```

No Plotly, um *trace* se refere a uma coleção de dados em um gráfico. O método update_traces() pode receber diversos argumentos diferentes; qualquer argumento que comece com marker_ afeta as marcações no gráfico. Aqui, definimos a cor de cada marcação como 'SteelBlue'; qualquer cor CSS nomeada funcionará. Definimos também a opacidade de cada marcação como 0,6. Uma opacidade de 1,0 será totalmente opaca, e uma opacidade de 0 será completamente invisível.

Mais sobre o Plotly e a API do GitHub

Embora a documentação do Plotly seja abrangente e bem-organizada, talvez seja difícil saber por onde começar a lê-la. Um bom ponto de partida é o artigo "Plotly Express in Python", em https://plotly.com/python/plotly-express. Trata-se de uma visão geral de todos os gráficos que podemos criar com o Plotly Express, além de encontrarmos links de artigos mais detalhados sobre cada tipo de gráfico individual.

Caso se interesse em entender melhor como personalizar gráficos com o Plotly, o artigo "Styling Plotly Express Figures in Python" ampliará os conceitos vistos nos capítulos 15 a 17. É possível encontrar esse artigo em https://plotly.com/python/styling-plotly-express.

Para obter mais informações sobre a API do GitHub, confira a documentação em https://docs.github.com/en/rest. Aqui, você aprenderá a extrair uma grande variedade de informações do GitHub. Para ver

ainda mais coisas, procure a seção Search na barra lateral. Se tiver uma conta do GitHub, poderá trabalhar com os próprios dados e também com os dados publicamente disponíveis dos repositórios de outros usuários.

API do Hacker News

Para explorar como usar chamadas de API em outros sites, vamos conferir o Hacker News (https://news.ycombinator.com). No Hacker News, as pessoas compartilham artigos sobre programação e tecnologia e engajam-se em discussões estimulantes sobre esses artigos. A API do Hacker News fornece acesso a dados sobre todas as contribuições de artigos e sobre os comentários no site, e é possível usá-la sem ter que se registrar para obter uma chave.

A chamada a seguir retorna informações sobre o artigo principal atual (no momento em que eu escrevia esta obra):

```
https://hacker-news.firebaseio.com/v0/item/31353677.json
```

Ao inserir esse URL em um navegador, veremos que o texto na página está entre chaves, ou seja, é um dicionário. No entanto, é difícil examinar a resposta sem uma formatação melhor. Executaremos esse URL com o método `json.dumps()`, como fizemos no projeto do terremoto no Capítulo 16. Desse modo, podemos analisar o tipo de informação retornada sobre um artigo:

hn_article.py

```
import requests
import json

# Cria uma chamada de API e armazena a resposta
url = "https://hacker-news.firebaseio.com/v0/item/31353677.json"
r = requests.get(url)
print(f"Status code: {r.status_code}")

# Explora a estrutura dos dados
response_dict = r.json()
response_string = json.dumps(response_dict, indent=4)
print(response_string)
```

Este programa deve lhe parecer familiar, pois tudo que escrevemos já foi usado nos dois capítulos anteriores. Aqui, a principal diferença é que podemos exibir a string de resposta formatada 1 em vez de gravá-la em um arquivo, pois a saída não é extensa.

A saída é um dicionário de informações sobre o artigo com o ID 31353677.

```
  {
     "by": "sohkamyung",
1    "descendants": 302,
     "id": 31353677,
2    "kids": [
        31354987,
        31354235,
        -- trecho de código omitido --
     ],
     "score": 785,
     "time": 1652361401,
3    "title": "Astronomers reveal first image of the black hole
        at the heart of our galaxy",
     "type": "story",
4    "url": "https://public.nrao.edu/news/.../"
  }
```

O dicionário contém uma série de chaves com as quais podemos trabalhar. A chave "descendants" nos informa o número de comentários que o artigo recebeu 1. A chave "kids" fornece os IDs de todos os comentários feitos diretamente em resposta a essa contribuição 2. Cada um desses comentários também pode originar os próprios comentários, de modo que o número de "descendants" que uma contribuição tem é geralmente maior do que o número de "kids". Podemos ver o título do artigo em discussão 3 e também um URL para o artigo discutido 4.

O URL a seguir retorna uma lista simples de todos os IDs dos principais artigos atuais no Hacker News:

```
https://hacker-news.firebaseio.com/v0/topstories.json
```

É possível usar essa chamada para descobrir quais artigos estão na página inicial agora e, em seguida, gerar uma série de chamadas de

API semelhantes às que acabamos de examinar. Com essa abordagem, podemos exibir um resumo de todos os artigos na primeira página do Hacker News:

hn_submissions.py

```
  from operator import itemgetter

  import requests

  # Cria uma chamada de API e verifica a resposta
1 url = "https://hacker-news.firebaseio.com/v0/topstories.json"
  r = requests.get(url)
  print(f"Status code: {r.status_code}")

  # Processa as informações sobre cada contribuição de artigo
2 submission_ids = r.json()
3 submission_dicts = []
  for submission_id in submission_ids[:5]:
      # Cria uma nova chamada de API para cada contribuição de artigo
4     url = f"https://hacker-news.firebaseio.com/v0/item/{submission_id}.json"
      r = requests.get(url)
      print(f"id: {submission_id}\tstatus: {r.status_code}")
      response_dict = r.json()

      # Cria um dicionário para cada artigo
5     submission_dict = {
          'title': response_dict['title'],
          'hn_link': f"https://news.ycombinator.com/item?id={submission_id}",
          'comments': response_dict['descendants'],
      }
6     submission_dicts.append(submission_dict)

7 submission_dicts = sorted(submission_dicts, key=itemgetter('comments'),
                            reverse=True)

8 for submission_dict in submission_dicts:
      print(f"\nTitle: {submission_dict['title']}")
      print(f"Discussion link: {submission_dict['hn_link']}")
      print(f"Comments: {submission_dict['comments']}")
```

Primeiro, criamos uma chamada de API e exibimos o status da resposta 1. Essa chamada de API retorna uma lista contendo os IDs de até 500 dos artigos mais populares no Hacker News, no momento

que a chamada é executada. Em seguida, convertemos o objeto de resposta em uma lista Python 2, que atribuímos a submission_ids. Utilizaremos esses IDs para criar um conjunto de dicionários, cada um contendo informações sobre uma das contribuições atuais.

Definimos uma lista vazia chamada submission_dicts para armazenar esses dicionários 3. Depois, analisamos os IDs das 30 principais contribuições de artigo. Criamos uma nova chamada de API para cada contribuição gerando um URL que com o valor atual de submission_id 4. Exibimos o status de cada requisição com seu ID, para que possamos verificar se foi bem-sucedida.

Em seguida, criamos um dicionário para a contribuição que está sendo atualmente processada 5. Armazenamos o título da contribuição, um link para a página de discussão desse item e o número de comentários que o artigo recebeu até agora. Depois, anexamos cada submission_dict à lista submission_dicts 6.

No Hacker News, cada contribuição de artigo é ranqueada de acordo com uma pontuação geral e com base em vários fatores, incluindo quantas vezes foi votada, quantos comentários recebeu e a data mais recente. Queremos classificar a lista de dicionários pelo número de comentários. Para fazer isso, usamos uma função chamada itemgetter() 7, oriunda do módulo operator. Passamos a essa função a chave 'comments' para extrair o valor associado a essa chave em cada dicionário na lista. A função sortted() então usa esse valor como base para classificar a lista. Classificamos a lista em ordem inversa, assim as histórias mais comentadas são ordenadas em primeiro lugar.

Após a lista ser classificada, a percorremos com um loop 8 e exibimos três informações sobre cada uma das principais contribuições: o título, um link para a página de discussão e o número de comentários que a contribuição atualmente recebeu:

```
Status code: 200
id: 31390506    status: 200
id: 31389893    status: 200
id: 31390742    status: 200
```

```
-- trecho de código omitido --

Title: Fly.io: The reclaimer of Heroku's magic
Discussion link: https://news.ycombinator.com/item?id=31390506
Comments: 134

Title: The weird Hewlett Packard FreeDOS option
Discussion link: https://news.ycombinator.com/item?id=31389893
Comments: 64

Title: Modern JavaScript Tutorial
Discussion link: https://news.ycombinator.com/item?id=31390742
Comments: 20
-- trecho de código omitido --
```

Você recorreria a um processo semelhante para acessar e para analisar informações com qualquer API. Com esses dados, é possível criar uma visualização mostrando quais contribuições de artigos inspiraram as discussões recentes mais ativas. É também a base para aplicações que fornecem uma experiência de leitura personalizada para sites como o Hacker News. Para saber mais sobre quais tipos de informações é possível acessar por meio da API do Hacker News, visite a página da documentação em https://github.com/HackerNews/API.

NOTA *Às vezes, o Hacker News possibilita que as empresas que apoia façam postagens de contratação especiais, e os comentários são desativados nessas postagens. Se executarmos esse programa com uma dessas postagens, receberemos um* KeyError*. Se isso ocasionar um problema, é possível envolver o código que cria* submission_dict *em um bloco* try-except *para desconsiderar essas postagens.*

FAÇA VOCÊ MESMO

17.1 Outras linguagens: Modifique a chamada de API em *python_repos.py* para que gere um gráfico mostrando os projetos mais populares em outras linguagens. Teste linguagens como *JavaScript*, *Ruby*, *C*, *Java*, *Perl*, Haskell e *Go*.

17.2 Discussões ativas: Usando os dados do *hn_submissions.py*, crie um gráfico de barras mostrando as discussões mais ativas e atuais no Hacker News. A altura de cada barra deve corresponder ao número de comentários que cada contribuição recebeu. O rótulo para cada barra deve incluir o título da contribuição de artigo e um link para a

página de discussão dessa contribuição. Se receber um KeyError ao criar um gráfico, use um bloco try-except para ignorar as postagens promocionais

17.3 Testando python_repos.py: No *python_repos.py*, exibimos o valor de status_code para garantir que a chamada da API foi bem-sucedida. Desenvolva um programa chamado test_python_repos.py que usa o pytest para fazer uma asserção de que o valor de status_code é 200. Identifique algumas outras asserções possíveis: por exemplo, se o número de itens retornados é esperado e se o número total de repositórios é maior do que determinada quantidade.

17.4 Exploração adicional: Visite a documentação do Plotly e a API do GitHub ou a API do Hacker News. Use algumas das informações que encontrar lá para personalizar o estilo das plotagens que já fizemos ou para extrair algumas informações diferentes e criar as próprias visualizações. Caso esteja curioso para explorar outras APIs, confira as APIs mencionadas no repositório do GitHub em https://github.com/public-apis.

Recapitulando

Neste capítulo, aprendemos a usar APIs para escrever programas independentes que coletam automaticamente os dados de que precisam e usam esses dados para criar uma visualização. Utilizamos a API do GitHub para explorar os projetos Python com mais estrelas, e também vimos brevemente a API do Hacker News. Vimo como usar o pacote Requests para executar automaticamente uma chamada de API e como processar os resultados dessa chamada. Tivemos também um contato rápido com algumas configurações do Plotly que personalizam ainda mais a aparência dos gráficos que geramos.

No próximo capítulo, usaremos o Django para criar uma aplicação web como projeto final.

CAPÍTULO 18

Primeiros passos com o Django

À medida que a internet evoluiu, a linha entre sites e aplicativos móveis ficou tênue. Os sites e aplicativos ajudam os usuários a interagir com os dados de variadas maneiras. Felizmente, podemos recorrer ao Django para criar um único projeto que atenda a um site dinâmico e a um conjunto de aplicativos móveis. *O Django* é o *framework web* mais popular do Python, um conjunto de ferramentas arquitetadas para o desenvolvimento de aplicações web interativas. Neste capítulo, aprenderemos a usar o Django para desenvolver um projeto chamado Registro de Aprendizagem (no código, usaremos Log Learning), um sistema de registro online que possibilita acompanhar as informações aprendidas sobre diferentes tópicos.

Vamos escrever uma especificação para esse projeto e, logo depois, vamos definir modelos para os dados com os quais a aplicação trabalhará. Utilizaremos o sistema de administração do Django para fornecer alguns dados iniciais e, em seguida, escreveremos views e templates para que o Django possa criar as páginas de um site.

O Django é capaz de responder a requisições de página e facilitar a leitura e a gravação em um banco de dados, gerenciar usuários e muito mais. Nos capítulos 19 e 20, refinaremos o projeto Registro de Aprendizagem e faremos seu deploy em um live server (servidor ao vivo) para que você (e todos as pessoas ao redor do mundo) possam usá-lo.

Criando um projeto

Ao começarmos a trabalhar em algo tão importante como uma aplicação web, primeiro é necessário especificar os objetivos do projeto em um documento chamado especificação (*spec*). Após ter um conjunto claro de objetivos, podemos começar a identificar tarefas gerenciáveis para atingir esses objetivos.

Nesta seção, escreveremos uma especificação para o Registro de Aprendizagem e começaremos a trabalhar na primeira fase do projeto. Vamos configurar um ambiente virtual e vamos definir os aspectos iniciais de um projeto Django.

Escrevendo uma especificação

Uma especificação completa detalha os objetivos do projeto e suas funcionalidades, além de abordar seus aspectos e a interface do usuário. Como qualquer bom projeto ou plano de negócios, uma especificação deve mantê-lo focado e ajudar a manter seu projeto nos trilhos. Aqui, não vamos elaborar uma especificação completa de projeto, mas vamos definir alguns objetivos claros para manter o foco no processo de desenvolvimento. Vejamos a especificação que usaremos:

> Desenvolveremos uma aplicação web chamada Registro de Aprendizagem (no código e no site Learning Log) que viabiliza com que os usuários registrem os tópicos em que estão interessados e forneçam entradas sobre sua jornada de aprendizagem à medida que aprendem sobre cada tópico. A página inicial do Registro de Aprendizagem descreverá o site e convidará os usuários a se registrarem ou fazerem login. Uma vez logado, um usuário pode criar tópicos novos, adicionar entradas novas, ler e editar entradas existentes.

Ao pesquisar um tópico novo, o usuário consegue registrar um diário sobre o que aprendeu para ajudá-lo a acompanhar as novas informações e as informações já encontradas. Isso é particularmente

válido quando estudamos assuntos técnicos. Uma boa aplicação, como a que desenvolveremos, pode ajudar a contribuir efetivamente com esse processo.

Criando um ambiente virtual

Para trabalhar com o Django, primeiro precisamos configurar um ambiente virtual. Um *ambiente virtual* é um local de nosso sistema em que podemos instalar pacotes e isolá-los de todos os outros pacotes Python. Separar as bibliotecas de um projeto de outros projetos é conveniente, e será necessário quando fizermos o deploy do Registro de Aprendizagem em um servidor no Capítulo 20.

Crie um diretório novo para seu projeto chamado *learning_log*, acesse esse diretório em um terminal e insira o seguinte código para criar um ambiente virtual:

```
learning_log$ python -m venv ll_env
learning_log$
```

Aqui, estamos executando o módulo de ambiente virtual venv para criar um ambiente chamado *ll_env* (perceba que o nome começa com dois *Ls* minúsculos, não maiúsculos). Caso utilize um comando como o python3 ao executar programas ou instalar pacotes, não se esqueça de usar esse comando aqui.

Ativando o ambiente virtual

Agora precisamos ativar o ambiente virtual, com o seguinte comando:

```
learning_log$ source ll_env/bin/activate
(ll_env)learning_log$
```

Esse comando executa o script *activate* em *ll_env/bin/*. Quando o ambiente estiver ativo, veremos o nome do ambiente entre parênteses. Isso sinaliza que podemos instalar pacotes novos no ambiente e usar pacotes que já instalados. Os pacotes instalados em *ll_env* ficarão indisponíveis quando o ambiente estiver inativo.

NOTA *Se estiver usando o Windows, use o comando* **ll_env\Scripts\activate** *(sem a palavra* **source***) para ativar o ambiente virtual. Se estiver usando o PowerShell, talvez seja necessário inserir a primeira letra de* **Activate** *em maiúsculo.*

Desative um ambiente virtual digitando **deactivate**:

```
(ll_env)learning_log$ deactivate
learning_log$
```

O ambiente também ficará inativo quando fecharmos o terminal em que está sendo executado.

Instalando o Django

Com o ambiente virtual ativado, digite o seguinte para atualizar o pip e instalar o Django:

```
(ll_env)learning_log$ pip install --upgrade pip
(ll_env)learning_log$ pip install django
Collecting django
-- trecho de código omitido --
Installing collected packages: sqlparse, asgiref, django
Successfully installed asgiref-3.5.2 django-4.1 sqlparse-0.4.2
(ll_env)learning_log$
```

Como faz o download de recursos de variadas fontes, o pip é atualizado com bastante frequência. É boa ideia atualizar o pip sempre que criarmos um ambiente virtual novo.

Como agora estamos trabalhando em um ambiente virtual, o comando para instalar o Django é o mesmo em todos os sistemas. Não há necessidade de usar comandos extensos, como python -m pip install *nome_package*, ou incluir a flag --user. Lembre-se de que o Django estará disponível somente quando o ambiente *ll_env* estiver ativo.

NOTA *O Django lança uma versão nova a cada oito meses. Ou seja, talvez você veja uma versão mais recente quando instalá-lo. Provavelmente, esse projeto funcionará como está aqui, mesmo com versões mais recentes do Django. Caso queira usar a mesma versão do Django que estamos usando aqui, digite o comando* pip install django==4.1.*. *Isso instalará a*

versão mais recente do Django 4.1. Se tiver algum problema relacionado à versão que está usando, confira os recursos online deste livro em https://ehmatthes.github.io/pcc_3e*.*

Criando um projeto no Django

Sem sair do ambiente virtual ativo (lembre-se de procurar *ll_env* entre parênteses no prompt do terminal), insira os seguintes comandos para criar um projeto novo:

```
1 (ll_env)learning_log$ django-admin startproject ll_project.
2 (ll_env)learning_log$ ls
  ll_env ll_project manage.py
3 (ll_env)learning_log$ ls ll_project
  __init__.py asgi.py settings.py urls.py wsgi.py
```

O comando startproject 1 informa ao Django para definir um projeto novo chamado *ll_project*. O ponto (.) no final do comando cria o projeto novo com uma estrutura de diretório que facilitará a implantação da aplicação em um servidor quando terminarmos de desenvolvê-lo.

NOTA *Não se esqueça deste ponto, ou você pode ter alguns problemas de configuração ao implantar a aplicação. Caso esqueça do ponto, exclua os arquivos e pastas que foram criados (exceto ll_env) e execute novamente o comando.*

Executar o comando ls (dir no Windows) 2 mostra que o Django criou um diretório novo chamado *ll_project*. Um arquivo *manage.py* também foi criado, um programa curto que recebe comandos e os fornece para a parte relevante do Django. Utilizaremos esses comandos para gerenciar tarefas, como trabalhar com bancos de dados e subir servidores.

O diretório *ll_project* contém quatro arquivos 3; os mais importantes são *settings.py*, *urls.py* e *wsgi.py*. O arquivo *settings.py* controla como o Django interage com seu sistema e gerencia seu projeto. Vamos modificar algumas dessas configurações e adicionar configurações próprias conforme o projeto evolui. O arquivo *urls.py*

informa ao Django quais páginas criar em resposta às requisições do navegador. O arquivo *wsgi.py* ajuda o Django a fornecer os arquivos que cria. O nome do arquivo é um acrônimo para "web server gateway interface (interface de porta de entrada de servidor web)".

Criando o banco de dados

O Django armazena a maioria das informações de um projeto em um banco de dados. Por isso, é necessário criar um com o qual o framework possa trabalhar. Digite o seguinte comando (ainda em um ambiente ativo):

```
(ll_env)learning_log$ python manage.py migrate
1 Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  -- trecho de código omitido --
  Applying sessions.0001_initial... OK
2 (ll_env)learning_log$ ls
db.sqlite3 ll_env ll_project manage.py
```

Sempre que alteramos um banco de dados, falamos que estamos *migrando* o banco de dados. Executar o comando `migrate` pela primeira vez faz com que o Django garanta que o banco de dados corresponda ao estado atual do projeto. A primeira vez que executarmos esse comando com o SQLite (daqui a pouco falaremos mais sobre o SQLite) em um projeto novo, o Django criará um banco de dados novo para nós. Aqui, o Django relata que preparará o banco de dados a fim de armazenar as informações necessárias para lidar com tarefas referentes a administração de banco e de autenticação `1`.

Executar o comando `ls` mostra que o Django criou outro arquivo chamado *db.sqlite3* `2`. O *SQLite* é um banco de dados que executa um único arquivo; é ideal para desenvolver aplicações simples, pois não precisamos nos atentar muito ao gerenciamento do banco de dados.

NOTA *Em um ambiente virtual ativo, use o comando* python *para executar os comandos* manage.py*, ainda que você use algo diferente, como* python3*, para executar outros programas. Em um ambiente virtual, o comando* python *se refere à versão do Python usada para criar o ambiente virtual.*

Visualizando o projeto

Vamos assegurar que o Django configurou devidamente o projeto. Digite o comando **runserver** para visualizar o projeto em seu estado atual:

```
(ll_env)learning_log$ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
May 19, 2022 - 21:52:35
Django version 4.1, using settings 'll_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

O Django deve iniciar um servidor chamado *development server* (servidor de desenvolvimento). Assim, podemos visualizar o projeto em nosso sistema e ver se está funcionando ou não. Ao requisitarmos uma página inserindo um URL em um navegador, o servidor Django responde a essa requisição criando a página adequada e a enviando ao navegador.

Primeiro, o Django verifica se o projeto está corretamente configurado 1; em seguida, informa a própria versão e o nome do arquivo settings usado 2. Por último, informa o URL em que o projeto está sendo disponibilizado 3. O URL *http://127.0.0.1:8000/* indica que o projeto está ouvindo requisições na porta 8000 do nosso computador, chamada de localhost. O termo *localhost* se refere a um servidor que processa apenas requisições em seu sistema, não permitindo que ninguém veja as páginas que estamos desenvolvendo.

Abra um navegador e digite o URL *http://localhost:8000/*, ou

http://127.0.0.1:8000/ caso o primeiro não funcione. Devemos ver algo como a Figura 18.1: uma página que o Django cria para que saibamos que tudo está funcionando devidamente até agora. Por enquanto, mantenha o servidor funcionando, mas caso queira desativá-lo, pressione CTRL+C no terminal em que o comando runserver foi executado.

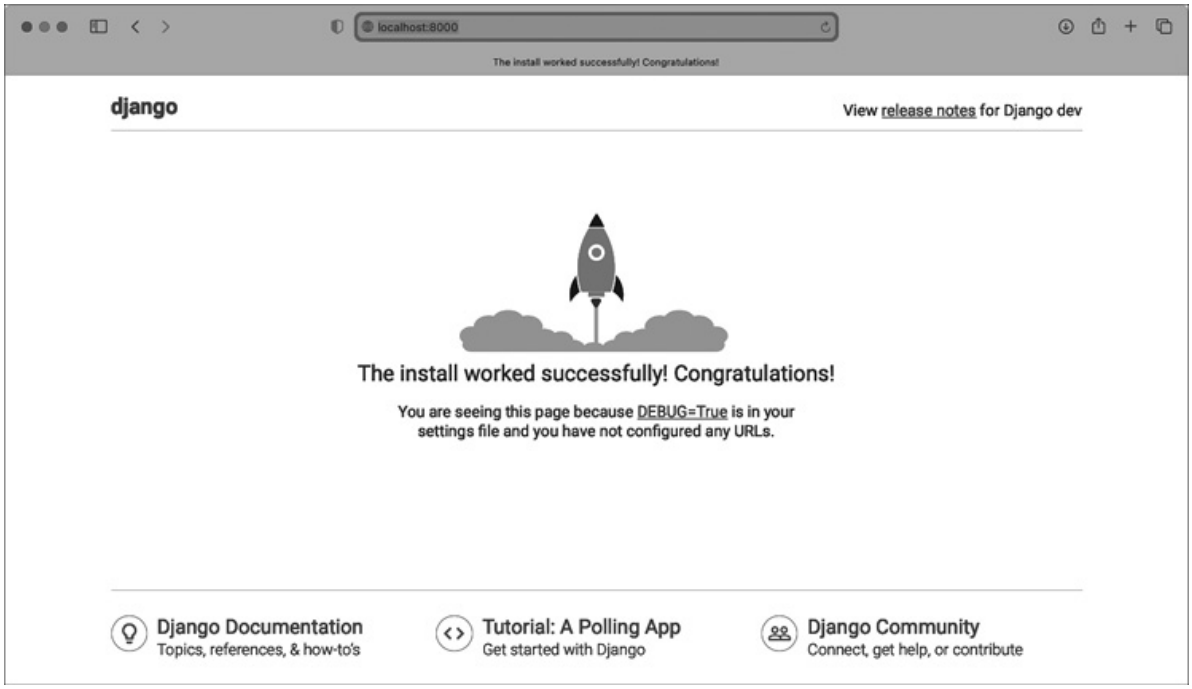


Figura 18.1: Até o momento, tudo está funcionando.

NOTA *Se receber a mensagem de erro "That port is already in use", instrua o Django a usar uma porta diferente inserindo* python manage.py runserver 8001 *e, em seguida, passe pelos números maiores até encontrar uma porta disponível.*

FAÇA VOCÊ MESMO

18.1 Projetos novos: Para ter uma noção melhor de como o framework funciona, crie alguns projetos vazios e veja o que o Django cria. Crie uma pasta nova com um nome simples, como *tik_gram* ou *insta_tok* (fora do seu diretório *learning_log*), navegue até essa pasta em um terminal e crie um ambiente virtual. Instale o Django e execute o comando django-admin.py startproject tg_project. (não se esqueça do ponto no final do comando).

Verifique os arquivos e pastas que esse comando cria e os compare com o Learning Log. Faça isso algumas vezes, até se familiarizar com o que Django cria ao iniciar um

projeto novo. Em seguida, exclua os diretórios do projeto, caso queira.

Iniciando uma aplicação

Um *projeto* Django é estruturado como um grupo de *aplicações* individuais que trabalham juntas para que o projeto funcione como um todo. Por ora, vamos criar uma aplicação que se encarregue da maior parte do trabalho do nosso projeto. Adicionaremos outra aplicação no Capítulo 19 para gerenciar contas de usuário.

É necessário que o development server esteja rodando na janela de terminal que abrimos anteriormente. Abra uma janela nova de terminal (ou guia) e navegue até o diretório com o *manage.py*. Ative o ambiente virtual e execute o comando **startapp**:

```
  learning_log$ source ll_env/bin/activate
  (ll_env)learning_log$ python manage.py startapp learning_logs
1 (ll_env)learning_log$ ls
  db.sqlite3 learning_logs ll_env ll_project manage.py
2 (ll_env)learning_log$ ls learning_logs/
  __init__.py admin.py apps.py migrations models.py tests.py views.py
```

O comando startapp *appname* solicita que o Django crie a infraestrutura necessária para desenvolver a aplicação. Agora, quando olhamos no diretório do projeto, veremos uma pasta nova chamada *learning_logs* 1. Use o comando **ls** para conferir o que Django criou 2. Os arquivos mais importantes são *models.py*, *admin.py* e *views.py*. Usaremos *models.py* para definir os dados que queremos gerenciar em nossa aplicação. Veremos *admin.py* e *views.py* um pouco mais tarde.

Definindo modelos

Vamos pensar em nossos dados por um momento. É necessário que cada usuário crie uma série de tópicos em seu registro de aprendizagem. Cada entrada inserida será vinculada a um tópico, e essas entradas serão exibidas como texto. Além disso, será necessário armazenarmos o timestamp de cada entrada, assim conseguimos mostrar aos usuários quando criaram cada uma delas.

Vamos abrir o arquivo *models.py* e ver o conteúdo existente:

models.py

```
from django.db import models

# Create your models here
```

Um módulo chamado models está sendo importado, e estamos sendo convidados a criar nossos modelos. Um *modelo* informa ao Django como trabalhar com os dados que serão armazenados na aplicação. Um modelo é uma classe; tem atributos e métodos, assim como todas as classes que estudamos. Vejamos o modelo para os tópicos que os usuários armazenarão:

```
from django.db import models

class Topic(models.Model):
    """Um tópico que o usuário está aprendendo"""
1   text = models.CharField(max_length=200)
2   date_added = models.DateTimeField(auto_now_add=True)

3   def __str__(self):
        """ Retorna uma representação de string do modelo"""
        return self.text
```

Criamos uma classe chamada Topic, que herda de Model – uma classe-pai incluída no Django que define a funcionalidade básica de um modelo. Adicionamos dois atributos à classe Topic: text e date_added.

O atributo text é um CharField, um dado composto por caracteres ou texto 1. Recorremos ao CharField quando queremos armazenar uma pequena quantidade de texto, como um nome, um título ou uma cidade. Ao definirmos um atributo CharField, temos que informar ao Django quanto espaço deve reservar no banco de dados. Aqui, atribuímos um max_length de 200 caracteres, que deve ser suficiente para armazenar a maioria dos nomes de tópicos.

O atributo date_added é um DateTimeField, um dado que registrará uma data e hora 2. Passamos o argumento auto_now_add=True, que informa ao Django para definir automaticamente esse atributo com a data e hora atuais sempre que o usuário criar um tópico novo.

É bom informar ao Django como queremos que o framework represente uma instância de um modelo. Se um modelo tem um método __str__(), o Django chama esse método sempre que for necessário gerar uma saída referente a uma instância desse modelo. Aqui, escrevemos um método __str__() que retorna o valor atribuído ao atributo text 3.

Para ver os diferentes tipos de campos que podemos usar em um modelo, confira a página "Model Field Reference" em https://docs.djangoproject.com/en/4.1/ref/models/fields. Não precisa saber de tudo nesse exato momento, mas isso será extremamente útil quando você estiver desenvolvendo seus projetos Django.

Ativando modelos

Para utilizar nossos modelos, é necessário dizer ao Django para incluir nossa aplicação no projeto geral. Basta abrir *settings.py* (no diretório *ll_project*); veremos uma seção que informa ao Django quais aplicações estão instalados no projeto:

settings.py

```
    -- trecho de código omitido --
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
    -- trecho de código omitido --
```

Adicione nossa aplicação a essa lista modificando INSTALLED_APPS para que fique assim:

```
    -- trecho de código omitido --
INSTALLED_APPS = [
    # Minhas aplicações
    'learning_logs',

    # Aplicações default do Django
```

```
    'django.contrib.admin',
    -- trecho de código omitido--
]
-- trecho de código omitido --
```

Agrupar aplicações em um projeto ajuda a acompanhá-las à medida que o projeto aumenta e passar a ter mais aplicações. Em nosso caso, começamos uma seção chamada Minhas aplicações, que, por ora, inclui somente 'learning_logs'. É importante posicionar suas aplicações antes das aplicações default, caso precise substituir qualquer comportamento das aplicações default por seu comportamento personalizado.

Em seguida, precisamos solicitar que o Django modifique o banco de dados para que consiga armazenar informações relacionadas ao modelo Topic. No terminal, execute o seguinte comando:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
  learning_logs/migrations/0001_initial.py
    - Create model Topic
(ll_env)learning_log$
```

O comando makemigrations solicita que Django identifique como modificar o banco de dados para que possa armazenar os dados associados a quaisquer modelos novos que definimos. A saída mostra que o Django criou um arquivo de migração chamado *0001_initial.py*. Essa migração criará uma tabela para o modelo Topic no banco de dados.

Agora, usaremos essa migração e o Django modificará o banco de dados para nós:

```
(ll_env)learning_log$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
  Applying learning_logs.0001_initial... OK
```

Boa parte da saída desse comando é idêntica à saída que obtivemos quando executamos migrate pela primeira vez. Precisamos verificar a última linha dessa saída, em que Django valida que a migração para

learning_logs funcionou OK.

Sempre que quisermos modificar os dados que o learning_logs gerencia, seguiremos esses três passos: modificar *models.py,* chamar makemigrations em learning_logs e dizer ao Django para executar migrate no projeto.

Site admin do Django

O Django facilita trabalhar como seus modelos por meio de seu site admin. O *site admin* do Django deve ser usado apenas pelos administradores do site; não se destina a usuários regulares. Nesta seção, vamos configurar o site admin e vamos usá-lo para adicionar alguns tópicos por meio do modelo Topic.

Configurando um superusuário

O Django possibilita que criemos um *superusuário*, um usuário que tem todos os privilégios disponíveis no site. Os *privilégios* de um usuário controlam as ações que ele pode tomar. As configurações mais restritivas de privilégio possibilitam que um usuário leia somente informações públicas no site. No site, usuários registrados normalmente têm o privilégio de ler os próprios dados privados e algumas informações selecionadas disponíveis apenas para os membros. Para administrar efetivamente um projeto, em geral, o proprietário do site precisa ter acesso a todas as informações armazenadas nele. Um bom administrador é meticuloso com as informações confidenciais de seus usuários, já que eles confiam muito nas aplicações que acessam.

Para criar um superusuário no Django, digite o seguinte comando e responda aos prompts:

```
  (ll_env)learning_log$ python manage.py createsuperuser
1 Username (leave blank to use 'eric'): ll_admin
2 Email address:
3 Password:
  Password (again):
```

```
Superuser created successfully.
(ll_env)learning_log$
```

Ao executar o comando createuperuser, o Django nos pede para fornecer um nome de usuário para o superusuário 1. Aqui, estou usando ll_admin, mas é possível inserir qualquer nome de usuário que quiser. Podemos inserir um endereço de e-mail ou apenas deixar esse campo em branco 2. Será necessário digitar a senha duas vezes 3.

NOTA *Algumas informações confidenciais podem ser ocultadas dos administradores de um site. Por exemplo, o Django não armazena a senha que você digita; ao contrário, armazena uma string derivada da senha, chamada de* hash*. Sempre que digitar sua senha, o Django faz um hash de sua entrada e a compara com o hash armazenado. Se os dois hashes corresponderem, você é autenticado. Ao exigir que os hashes correspondam, o Django garante que, se um hacker obtiver acesso ao banco de dados de um site, ele conseguirá ler os hashes armazenados, não as senhas. Quando um site é adequadamente desenvolvido, é quase impossível obter as senhas originais dos hashes*.

Registrando um modelo com o site admin

Embora o Django inclua automaticamente alguns modelos no site admin, como User e Group, precisamos adicionar manualmente os modelos que criamos.

Quando iniciamos o aplicação learning_logs, o Django criou um arquivo *admin.py* no mesmo diretório que o *models.py*. Abra o arquivo *admin.py*:

admin.py

```
from django.contrib import admin

# Register your models here.
```

Para registrar Topic no site admin, digite o seguinte:

```
from django.contrib import admin

from .models import Topic

admin.site.register(Topic)
```

Esse código inicialmente importa o modelo que queremos registrar, `Topic`. O ponto na frente de `models` instrui que o Django procure *models.py* no mesmo diretório que *admin.py*. O código `admin.site.register()` informa ao Django para gerenciar nosso modelo por meio do site admin.

Agora, use a conta de superusuário para acessar o site admin. Acesse *http://localhost:8000/admin/* e insira o nome de usuário e a senha do superusuário que acabamos de criar. Deveremos ver uma tela semelhante à mostrada na Figura 18.2. Essa página possibilita adicionar usuários e grupos novos e mudar os existentes. É possível também trabalhar com dados relacionados ao modelo `Topic` que acabamos de definir.

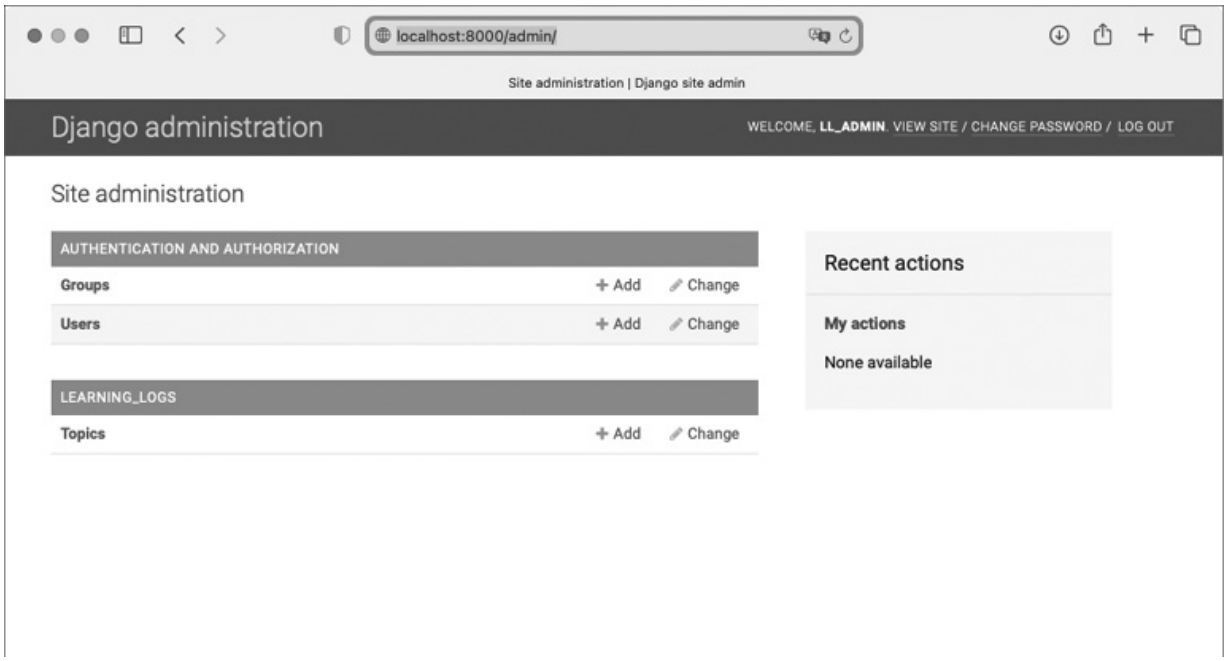


Figura 18.2: O site admin com Topic incluído.

NOTA *Caso veja uma mensagem em seu navegador de que a página da web não está disponível, verifique se o servidor*

Django ainda está rodando em uma janela de terminal. Caso contrário, ative um ambiente virtual e execute novamente o comando python manage.py runserver*. Se estiver tendo dificuldades para visualizar seu projeto em qualquer momento do processo de desenvolvimento, fechar qualquer terminal aberto e executar mais uma vez o comando* runserver *é um bom primeiro passo para solucionar o problema.*

Adicionando tópicos

Agora que Topic foi registrado no site admin, adicionaremos nosso primeiro tópico. Clique em **Topics** para ir para a página Topics, que está praticamente vazia, pois ainda não temos tópicos para gerenciar. Clique em **Add Topic**, e um formulário para adicionar um tópico novo será exibido. Digite **Chess** na primeira caixa e clique em **Save**. Seremos direcionados de volta para a página admin Topics, e podemos ver o tópico que acabamos de criar.

Criaremos um segundo tópico para termos mais dados com os quais trabalhar. Clique **Add Topic** novamente e insira **Rock Climbing**. Clique em **Save** e, mais uma vez, seremos enviados à página principal Topic. Agora, podemos ver Chess (Xadrez) e Rock Climbing (Escalada em Rocha).

Definindo o modelo Entry

Para que um usuário registre o que está aprendendo sobre xadrez e escalada em rocha, é necessário definir um modelo para os tipos de entradas que os usuários podem criar em seus registros de aprendizagem. Cada entrada precisa ser associada a um tópico específico. Chamamos esse relacionamento de *muitos-para-um*. Ou seja, muitas entradas podem ser associadas a um tópico.

Vejamos o código para o modelo Entry Insira-o no seu arquivo *models.py*:

models.py

```
from django.db import models

class Topic(models.Model):
    -- trecho de código omitido --

1 class Entry(models.Model):
    """Algo específico aprendido sobre um tópico"""
2   topic = models.ForeignKey(Topic, on_delete=models.CASCADE)
3   text = models.TextField()
    date_added = models.DateTimeField(auto_now_add=True)

4   class Meta:
        verbose_name_plural = 'entries'

    def __str__(self):
        """ Retorna uma string simples representando a entrada"""
5       return f"{self.text[:50]}..."
```

A classe `Entry` herda da classe `Model` base do Django, assim como `Topic` fez `1`. O primeiro atributo, `topic`, é uma instância de `ForeignKey` `2`. Uma *chave estrangeira* (foreign key) é um termo de banco de dados; trata-se de uma referência a outro registro no banco de dados. É o código que relaciona cada entrada a um tópico específico. Cada tópico recebe uma *chave*, ou ID, quando é criado. Ao precisar estabelecer uma conexão entre dois tópicos, o Django usa as chaves associadas a cada informação. Daqui a pouco, usaremos essas conexões a fim de acessar todas as entradas associadas a um determinado tópico. O argumento `on_delete= models.CASCADE` informa ao Django que quando um tópico é excluído, todas as entradas associadas a esse tópico também devem ser excluídas. Isso é conhecido como *exclusão em cascata ou deletar em cascata*.

O próximo é um atributo chamado `text`, uma instância de `TextField` `3`. Não é necessário limitar o tamanho desse tipo de campo, pois não queremos restringir o tamanho das entradas individuais. O atributo `date_added` nos possibilita apresentar entradas na ordem em que foram criadas e inserir um timestamp ao lado de cada entrada.

A classe `Meta` está aninhada dentro da classe `Entry` `4`. A classe `Meta` armazena informações extras para gerenciar um modelo; aqui, a

classe nos permite definir um atributo especial informando ao Django para usar Entries quando for necessário referenciar mais de uma entrada. Sem isso, o Django referenciaria múltiplas entradas como Entrys.

O método __str__() informa ao Django qual informação mostrar quando referenciar a entradas individuais. Como uma entrada pode ser um corpo extenso de texto, __str__() retorna somente os primeiros 50 caracteres de text 5. Além disso, adicionamos reticências para elucidar que nem sempre estamos exibindo toda a entrada.

Migrando o modelo Entry

Como adicionamos um modelo novo, precisamos migrar mais uma vez o banco de dados. Esse processo se tornará muito familiar: modificamos *models.py*, executamos o comando **python manage.py makemigrations *app_name*** e, em seguida, executamos o comando **python manage.py migrate.**

Migre o banco de dados e verifique a saída digitando os seguintes comandos:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
1  learning_logs/migrations/0002_entry.py
    - Create model Entry
(ll_env)learning_log$ python manage.py migrate
Operations to perform:
  -- trecho de código omitido --
2  Applying learning_logs.0002_entry... OK
```

Gera-se uma migração nova chamada *0002_entry.py*, que informa ao Django como modificar o banco de dados para armazenar informações relacionadas ao modelo Entry 1. Ao executarmos o comando migration, vemos que o Django fez essa migração e tudo funcionou de modo correto 2.

Registrando o modelo Entry no site admin

Precisamos também registrar o modelo Entry. Veja como deve ficar o

admin.py agora:

admin.py

```
from django.contrib import admin

from .models import Topic, Entry

admin.site.register(Topic)
admin.site.register(Entry)
```

Basta acessar mais uma vez *http://localhost/admin/*, que veremos as entradas listadas em *Learning_Logs*. Clique no link **Add** de **Entries** ou clique em **Entries** e escolha **Add entry**. Deveremos ver uma lista suspensa para selecionar o tópico para o qual estamos criando uma entrada e uma caixa de texto para adicionar uma entrada. Selecione **Chess** na lista suspensa e adicione uma entrada. Vejamos a primeira entrada que fiz:

> A abertura é a primeira parte do jogo, mais ou menos os primeiros dez movimentos. Na abertura, é uma boa ideia fazer três coisas – avançar seus bispos e cavalos, tentar controlar o centro do tabuleiro e mover seu rei com um roque.
> Obviamente, não passam de orientações. Será importante aprender quando seguir essas orientações e quando desconsiderar essas sugestões.

Ao clicar em **Save**, retornaremos à página admin principal de entradas. Nesse momento, você perceberá a vantagem de usar `text[:50]` como representação de string para cada entrada; fica mais fácil trabalhar com múltiplas entradas na interface de administração se virmos somente a primeira parte de uma entrada, em vez de todo o texto de cada entrada.

Crie uma segunda entrada para Chess e uma entrada para Rock Climbing para termos alguns dados iniciais. Vejamos a segunda entrada para Chess:

> Na fase de abertura do jogo, é importante avançar com seus bispos e seus cavalos. Trata-se de peças poderosas e manobráveis o suficiente para desempenhar um papel

significativo nos movimentos iniciais de um jogo.

Vejamos a primeira entrada para Rock Climbing:

> Na escalada, um dos conceitos primordiais é equilibrar o peso nos pés o máximo possível. Existe um mito de que os alpinistas conseguem ficar pendurados com os braços o dia inteiro. Na realidade, bons alpinistas praticam maneiras específicas de equilibrar seu peso sobre os pés sempre que possível.

Essas três entradas nos fornecerão algo com o qual trabalhar à medida que continuamos a desenvolver o Registro de Aprendizagem.

Shell do Django

Agora que inserimos alguns dados, podemos programaticamente examiná-los por meio de uma sessão interativa de terminal. Trata-se de um ambiente interativo chamado de *shell* do Django, um ótimo ambiente para testar e solucionar problemas do projeto. Vejamos um exemplo de uma sessão interativa do shell:

```
(ll_env)learning_log$ python manage.py shell
>>> from learning_logs.models import Topic
>>> Topic.objects.all()
<QuerySet [<Topic: Chess>, <Topic: Rock Climbing>]>
```

O comando `python manage.py shell`, executado em um ambiente virtual ativo, inicia o interpretador Python que podemos utilizar para explorar os dados armazenados no banco de dados do projeto. Aqui, importamos o modelo `Topic` do módulo `learning_logs.models 1`. Depois, usamos o método `Topic.objects.all()` para obter todas as instâncias do modelo `Topic`; a lista retornada é chamada de *queryset*.

É possível percorrer uma queryset com um loop do mesmo jeito que fazemos com uma lista. Vejamos o ID que foi atribuído a cada objeto de tópico:

```
>>> topics = Topic.objects.all()
>>> for topic in topics:
...     print(topic.id, topic)
...
```

```
1 Chess
2 Rock Climbing
```

Atribuímos queryset a topics e, em seguida, exibimos o atributo id e a representação da string de cada tópico. Podemos ver que Chess tem um ID de 1 e Rock Climbing tem um ID de 2.

Caso saiba o ID de um objeto específico, é possível recorrer ao método Topic.objects.get() para acessar esse objeto e examinar qualquer atributo que o objeto tenha. Vejamos os valores text e date_added para Chess:

```
>>> t = Topic.objects.get(id=1)
>>> t.text
'Chess'
>>> t.date_added
datetime.datetime(2022, 5, 20, 3, 33, 36, 928759,
    tzinfo=datetime.timezone.utc)
```

É possível ver também as entradas relacionadas a um determinado tópico. Anteriormente, definimos o atributo topic para o modelo Entry. Tratava-se de uma ForeignKey, uma conexão entre cada entrada e um tópico. O Django pode utilizar essa conexão para obter todas as entradas relacionadas a um determinado tópico, assim.

```
1 >>> t.entry_set.all()
  <QuerySet [<Entry: The opening is the first part of the game, roughly...>, <Entry:
  In the opening phase of the game, it's important t...>]>
```

Para obter dados por meio de um relacionamento de chave estrangeira, escreva o nome do modelo relacionado com letras minúsculas, seguido por um underscore e pela palavra set 1. Por exemplo, digamos que você tenha os modelos Pizza e Topping, e Topping está relacionado à Pizza por meio de uma chave estrangeira. Caso seu objeto se chame my_pizza, representando uma única pizza, é possível obter todos os ingredientes da pizza com o código my_pizza.topping_set.all().

Vamos recorrer a essa sintaxe quando começarmos a desenvolver as páginas que os usuários podem solicitar. O shell é bastante útil para garantir que o código acesse os dados que queremos. Caso seu código funcione no shell como o esperado, também deve funcionar

adequadamente nos arquivos do seu projeto. Se o código gerar erros ou não acessar os dados esperados, fica mais fácil solucionar problemas em um simples ambiente shell do que nos arquivos que geram páginas web. Apesar de não usarmos muito o shell aqui, você deve usá-lo para praticar e trabalhar com a sintaxe do Django e a fim de acessar os dados armazenados no projeto.

Sempre que modificar seus modelos, será necessário reiniciar o shell para conferir os efeitos dessas mudanças. Para sair de uma sessão do shell, pressione as teclas CTRL+D; no Windows, pressione as teclas Ctrl+Z e, em seguida, pressione a tecla ENTER.

FAÇA VOCÊ MESMO

18.2 Entradas menores: Atualmente, o método __str__() no modelo Entry anexa uma reticência a cada instância de Entry quando o Django a mostra no site admin ou no shell. Adicione uma instrução if ao método __str__() para inserir reticências somente se a entrada tiver mais de 50 caracteres. Use o site admin para adicionar uma entrada com menos de 50 caracteres e verifique se essa entrada não tem reticência quando visualizada.

18.3 A API do Django: Ao escrevermos código para acessar os dados em nosso projeto, estamos escrevendo uma *query*. Leia rapidamente a documentação sobre query e dados em https://docs.djangoproject.com/en/4.1/topics/db/queries. Você verá muita coisa nova, que será bastante útil quando começar a trabalhar em seus projetos.

18.4 Pizzaria: Comece um projeto novo chamado pizzeria_project com uma aplicação chamada pizzas. Defina um modelo Pizza com um campo chamado name para armazenar valores de nome, como Hawaiian e Meat Lovers. Defina um modelo chamado Topping com campos chamados pizza e name. O campo pizza deve ser uma chave estrangeira para Pizza, e o campo name deve ser capaz de armazenar valores como pineapple, Canadian bacon e sausage.

Registre os dois modelos com o site admin e use o site para inserir alguns nomes e ingredientes de pizza. Use o shell para explorar os dados inseridos.

Criando páginas: a página inicial do Registro de Aprendizagem

A criação de páginas com o Django tem três etapas: definir URLs, escrever views e escrever templates. É possível fazer isso independente da ordem. No entanto, com esse projeto, sempre começaremos definindo o padrão de URL. Um *padrão de URL*

descreve a forma como o URL é apresentado, e também instrui o Django o que procurar ao combinar uma requisição do navegador com um URL do site. Assim, o framework sabe qual página retornar.

Desse modo, cada URL é mapeado para uma view específica. A função *view* acessa e processa os dados necessários para determinada página. Via de regra, a função view renderiza a página usando um *template*, que contém a estrutura geral da página. Para vermos como isso funciona, criaremos a página inicial do Registro de Aprendizado. Vamos definir o URL para a página inicial, vamos escrever sua função view e vamos criar um template simples.

Como apenas queremos garantir que o Registro de Aprendizagem funcione como deveria, criaremos uma página simples por enquanto. É divertido estilizar uma aplicação web funcional, quando está completa; uma aplicação, supostamente bacana, mas que não funciona, não serve de nada. Por ora, a página inicial exibirá somente um título e uma breve descrição.

Mapeando um URL

Como os usuários acessam páginas inserindo URLs em um navegador e clicando em links, precisamos decidir quais URLs são necessários. O URL da página inicial é o primeiro: é o URL base que as pessoas usam para acessar o projeto. No momento, o URL base, *http://localhost:8000/*, retorna o site default do Django que nos informa que o projeto foi devidamente configurado. Vamos alterar isso mapeando o URL base para a página inicial do Registro de Aprendizagem.

Na pasta principal *ll_project*, abra o arquivo *urls.py*. Devemos ver o seguinte código:

ll_project/urls.py

```
1 from django.contrib import admin
  from django.urls import path

2 urlpatterns = [
```

```
3     path('admin/', admin.site.urls),
  ]
```

As duas primeiras linhas importam o módulo admin e uma função para criar paths de URL 1. O corpo do arquivo define a variável urlpatterns 2. No arquivo *urls.py*, que define URLs para o projeto geral, a variável urlpatterns inclui conjuntos de URLs das aplicações do projeto. A lista inclui o módulo admin.site.urls, que define todos os URLs que podem ser requisitados a partir do site admin 3.

Como precisamos incluir os URLs para learning_logs, adicione o seguinte:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('learning_logs.urls')),
]
```

Importamos a função include() e também adicionamos uma linha para o módulo learning_logs.urls.

O *urls.py* default se encontra na pasta *ll_project*; agora precisamos criar um segundo arquivo *urls.py* na pasta *learning_logs*. Crie um arquivo novo Python, salve-o como *urls.py* em *learning_logs* e insira o seguinte código nele:

learning_logs/urls.py

```
1 """Define padrões de URL para learning_logs"""

2 from django.urls import path

3 from . import views

4 app_name = 'learning_logs'
5 urlpatterns = [
      # Página inicial
6     path('', views.index, name='index'),
  ]
```

Para explicitar em qual *urls.py* estamos trabalhando, adicionamos uma docstring no início do arquivo 1. Em seguida, importamos a

função path, necessária ao mapear URLs para as views 2. Importamos também o módulo views 3; o ponto instrui o Python a importar o módulo *views.py* do mesmo diretório que o módulo *urls.py* atual. A variável app_name ajuda o Django a diferenciar esse arquivo *urls.py* de arquivos com o mesmo nome em outras aplicações do projeto 4. Nesse módulo, a variável urlpatterns é uma lista de páginas individuais que podem ser requisitadas a partir da aplicação learning_logs 5.

O padrão efetivo de URL é uma chamada para a função path(), que recebe três argumentos 6. O primeiro argumento é uma string que ajuda o Django a encaminhar adequadamente a requisição atual. O Django recebe o URL solicitado e tenta encaminhar a requisição para uma view. O framework faz isso pesquisando todos os padrões de URL que definimos para encontrar um que corresponda à requisição atual. O Django ignora o URL base para o projeto (*http://localhost:8000/*), assim a string vazia (") corresponde ao URL base. Qualquer outro URL não corresponderá a esse padrão, e o Django retornará uma página de erro se o URL solicitado não corresponder a nenhum padrão de URL existente.

O segundo argumento em path() 6 especifica qual função chamar em *views.py*. Quando um URL requisitado corresponde ao padrão que estamos definindo, o Django chama a função index() em *views.py*. (Na próxima seção, escreveremos essa função view.) Como o terceiro argumento fornece o nome *index* para esse padrão de URL, podemos referenciá-lo com mais facilidade em outros arquivos durante todo o projeto. Sempre que quisermos fornecer um link para a página inicial, usaremos esse nome em vez de escrever um URL.

Escrevendo uma view

Uma função view recebe informações de uma requisição, prepara os dados necessários para gerar uma página e envia os dados de volta para o navegador. Normalmente, usando um template que define como a página será.

O arquivo *views.py* em *learning_logs* foi gerado automaticamente quando executamos o comando python manage.py startapp. Vamos conferir o que está em *views.py* agora:

views.py

```
from django.shortcuts import render

# Create your views here
```

Atualmente, esse arquivo apenas importa a função render(), que renderiza a resposta com base nos dados fornecidos pelas views. Abra *views.py* e adicione o seguinte código para a página inicial:

```
from django.shortcuts import render

def index(request):
    """A página inicial para o Registro de Aprendizagem"""
    return render(request, 'learning_logs/index.html')
```

Quando uma requisição de URL corresponde ao padrão que acabamos de definir, o Django procura uma função chamada index() no arquivo *views.py*. Depois, o Django passa o objeto request para essa função view. Nesse caso, não é necessário processar nenhum dado para a página, logo o único código na função é uma chamada para render(). Aqui, a função render() passa dois argumentos: o objeto request original e um template que pode usar para criar a página. Agora, vamos escrever esse template.

Escrevendo um template

O template define a aparência da página, e o Django fornece os dados relevantes sempre que a página é solicitada. Um template possibilita acessar todos os dados fornecidos pela view. Já que nossa view para a página inicial não fornece dados, esse template é relativamente simples.

Dentro da pasta *learning_logs*, crie uma pasta nova chamada *templates*. Dentro da pasta *templates*, crie outra pasta chamada *learning_logs*. Por mais que pareça um tanto redundante (temos uma pasta chamada *learning_logs* dentro de uma pasta chamada

templates, que está dentro de uma pasta chamada *learning_logs*), isso define uma estrutura que o Django pode interpretar inequivocamente, mesmo no contexto de um projeto enorme com muitas aplicações individuais. Dentro da pasta interna *learning_logs*, crie um arquivo novo chamado *index.html*. O path para o arquivo será *ll_project/learning_logs/templates/learning_logs/index.html*. Insira o seguinte código nesse arquivo:

index.html

```
<p>Learning Log</p>

<p>Learning Log helps you keep track of your learning, for any topic you're
interested in.</p>
```

Trata-se de um arquivo muito simples. Caso não conheça HTML, as tags `<p></p>` significam parágrafos. A tag `<p>` abre um parágrafo e a tag `</p>` fecha um parágrafo. Temos dois parágrafos: o primeiro se comporta como título, e o segundo descreve o que os usuários podem fazer com o Registro de Aprendizagem.

Agora, quando acessarmos o URL base do projeto, *http://localhost:8000/*, devemos ver a página que acabamos de criar em vez da página default do Django. O Django verificará se o URL solicitado corresponde ao padrão `''`; em seguida, chamará a função `views.index()`, que renderizará a página usando o template contido em *index.html*. A Figura 18.3 mostra a página resultante.

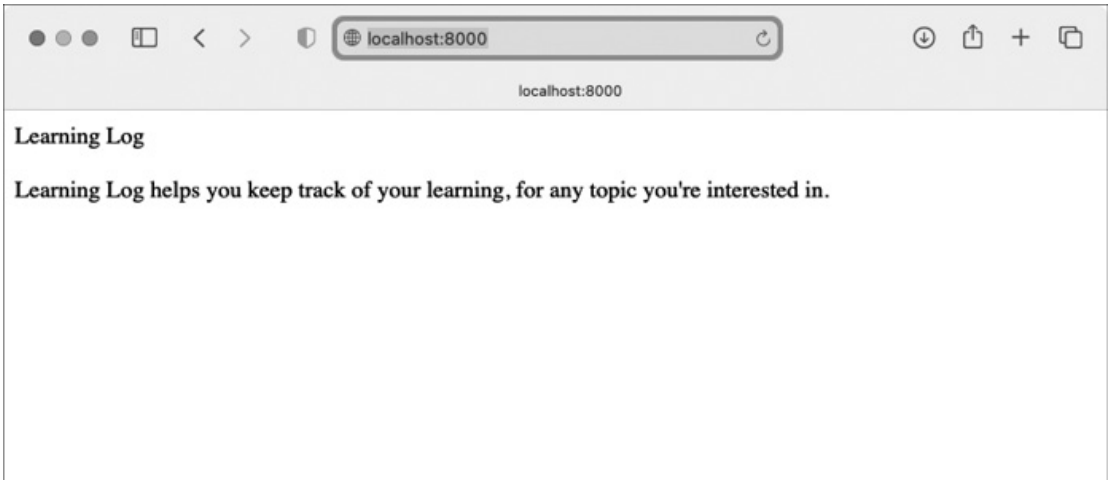


Figura 18.3: A página inicial do Registro de Aprendizagem.

Mesmo que, aparentemente, o processo de criar uma página seja complicado, a separação entre URLs, views e templates funciona até que bem. Possibilita que pensemos sobre cada aspecto de um projeto de forma separada. Em projetos maiores possibilita que as pessoas que trabalham juntas foquem as áreas em que são mais experientes. Por exemplo, um especialista em banco de dados pode focar os modelos, um programador pode se encarregar do código da view e um especialista em front-end se concentrar nos templates.

NOTA *Talvez você se depare com a seguinte mensagem de erro:*

```
ModuleNotFoundError: No module named 'learning_logs.urls'
```

Em caso afirmativo, pare o development server pressionando CTRL+C na janela de terminal em que executou comando `runserver`*. Em seguida, execute novamente o comando* `python manage.py runserver`*. Você deve ver a página inicial. Sempre que se deparar com um erro como esse, tente parar e reiniciar o servidor.*

FAÇA VOCÊ MESMO

18.5 Planejador de cardápios: Imagine um aplicativo que ajude as pessoas a planejar as refeições ao longo da semana. Crie uma pasta nova chamada *meal_planner* e inicie um projeto novo Django dentro dessa pasta. Em seguida, crie uma aplicação nova chamada meal_plans. Crie uma página inicial simples para esse projeto.

18.6 Página inicial da Pizzaria: Adicione uma página inicial ao projeto Pizzaria que iniciou no Exercício 18.4 (página *478*).

Criando páginas adicionais

Agora que estabelecemos uma rotina para criação de uma página, podemos começar a desenvolver o projeto Registro de Aprendizagem. Vamos criar duas páginas que exibem dados: uma página que enumera todos os tópicos e uma página que mostra todas as entradas para um tópico específico. Para cada página, especificaremos um padrão de URL, escreveremos uma função view e um template. Mas, antes de tudo, criaremos um template base

que todos os templates do projeto podem herdar.

Herança de template

Ao desenvolver um site, alguns elementos precisarão ser repetidos em cada página. Em vez de escrever esses elementos diretamente em cada página, é possível criar um template base com os elementos repetidos e, em seguida, fazer com que cada página herde esse template. Essa abordagem viabiliza focar o desenvolvimento dos aspectos específicos de cada página e facilita muito alterar a aparência geral do projeto.

Template-pai

Criaremos um template chamado *base.html* no mesmo diretório que *index.html*. Esse arquivo conterá os elementos comuns a todas as páginas; todos os outros templates herdarão de *base.html*. O único elemento que queremos repetir em cada página agora é o título na parte superior. Já que incluiremos esse template em todas as páginas, faremos com que o título tenha um link para a página inicial:

base.html

```
  <p>
1   <a href="{% url 'learning_logs:index' %}">Learning Log</a>
  </p>

2 {% block content %}{% endblock content %}
```

A primeira parte desse arquivo cria um parágrafo com o nome do projeto, que também se comporta como um link para a página inicial. Para gerar um link, utilizamos uma *template tag*, sinalizada por chaves e sinais de porcentagem (`{% %}`). Uma template tag gera informações para serem exibidas em uma página. A template tag `{% url 'learning_logs:index' %}` mostrada aqui gera um URL correspondente ao padrão de URL definido em *learning_logs/urls.py* com o nome `'index'` 1. Nesse exemplo, `learning_logs` é o *namespace* e `index` é um padrão de URL nomeado exclusivamente nesse namespace. O

namespace se origina do valor que atribuímos a app_name no arquivo *learning_logs/urls.py*.

Em uma página HTML simples, um link é contornado pela *tag âncora* <a>:

```
<a href="link_url">link text</a>
```

Quando fazemos com que a template tag gera o URL para nós, fica bem mais fácil manter nossos links atualizados. Basta alterar o padrão de URL em *urls.py*, e o Django inserirá automaticamente o URL atualizado na próxima vez que a página for solicitada. Já que todas as páginas do nosso projeto serão herdadas de *base.html*, a partir de agora, todas terão um link que retorne à página inicial.

Na última linha, inserimos um par de tags block 2. Esse bloco, chamado content, é um placeholder; o template-filho definirá o tipo de informação será inserida no bloco content.

Um template-filho não precisa definir todos os blocos de seu pai. Desse modo, podemos reservar espaço nos templates-pai para quantos blocos quisermos; o template-filho usa somente os blocos que precisar.

NOTA *No código Python, quase sempre usamos quatro espaços na indentação. Os arquivos de template costumam ter mais níveis de aninhamento do que os arquivos Python, então é comum utilizar apenas dois espaços para cada nível de indentação.*

Template-filho

Agora, é necessário reescrevermos *index.html* para herdar de *base.html*. Adicione o seguinte código ao *index.html*:

index.html

```
1 {% extends 'learning_logs/base.html' %}

2 {% block content %}
    <p>Learning Log helps you keep track of your learning, for any topic you're
```

```
    interested in.</p>
3 {% endblock content %}
```

Se compararmos esse arquivo com o *index.html* original, veremos que substituímos o título Learning Log pelo código que herda de um template-pai 1. Um template-filho deve ter uma tag {% extends %} na primeira linha para informar ao Django de qual template-pai herdar. Como o *arquivo* base.html faz parte de learning_logs, incluímos *learning_logs* no path para o template-pai. Essa linha extrai tudo o que está no *template* base.html e possibilita que *index.html* defina o que será inserido no espaço reservado pelo bloco content.

Definimos o bloco de conteúdo inserindo uma tag {% block %} com o nome content 2. Tudo o que não estamos herdando do template-pai será inserido dentro do bloco content. Aqui, esse é o parágrafo que descreve o projeto Registro de Aprendizagem. Indicamos que terminamos de definir o conteúdo usando uma tag {% endblock content %} 3. A tag {% endblock %} não exige um nome, mas se um template aumentar para acomodar múltiplos blocos, ajuda saber exatamente o final de cada bloco.

Talvez você esteja começando a perceber as vantagens da herança de template: no template-filho, basta incluir conteúdo específico dessa página. Isso não apenas simplifica cada template, como também facilita e muito a alterar o site. Para modificar um elemento comum a muitas páginas, basta modificar o template-pai. Assim, as mudanças são transferidas para todas as páginas que herdam desse template. Em um projeto com dezenas ou centenas de páginas, uma estrutura como essa facilita e agiliza bastante melhorar nosso site.

Em um projeto grande, é comum ter um template-pai chamado *base.html* para todo o site e templates-pai para cada seção principal do site. Todos os templates de seção herdam de *base.html* e cada página no site herda de um template de seção. Dessa forma, conseguimos modificar facilmente a aparência geral do site, assim como qualquer seção ou qualquer página individual. Essa configuração viabiliza um modo mais eficiente de se trabalhar e nos

incentiva a atualizar regularmente nosso projeto a longo prazo.

Página de tópicos

Agora que temos uma abordagem eficaz para criar páginas, podemos focar nossas próximas duas páginas: a página de tópicos gerais e a página para exibir entradas para um único tópico. A página de tópicos mostrará todos os tópicos que os usuários criaram, sendo a primeira página em que trabalharemos com a utilização de dados.

Padrão de URL dos tópicos

Primeiro, definimos o URL para a página de tópicos. É comum escolher um trecho simples de URL que retrate o tipo de informação apresentado na página. Usaremos a palavra *topics*. Logo o URL *http://localhost:8000/topics/* retornará essa página. Veja como modificamos *learning_logs/urls.py*:

learning_logs/urls.py

```
"""Define padrões de URL para learning_logs"""
-- trecho de código omitido --
urlpatterns = [
    # Página inicial
    path('', views.index, name='index'),
    # Página que mostra todos os tópicos
    path('topics/', views.topics, name='topics'),
]
```

O novo padrão de URL é a palavra *topics*, seguida por uma barra. Quando o Django examina um URL solicitado, esse padrão corresponderá a qualquer URL que tenha o URL base seguido por *topics*. É possível incluir ou omitir uma barra no final, mas não pode haver mais nada após a palavra *topics*, ou o padrão não corresponderá. Qualquer requisição com um URL que corresponda a esse padrão será passada para a função `topics()` em *views.py*.

View de tópicos

A função topics() precisa acessar alguns dados do banco de dados e enviá-los para o template. Adicione o seguinte a *views.py*:

views.py

```
from django.shortcuts import render

1 from .models import Topic

def index(request):
    -- trecho de código omitido --

2 def topics(request):
    """Mostra todos os tópicos"""
3   topics = Topic.objects.order_by('date_added')
4   context = {'topics': topics}
5   return render(request, 'learning_logs/topics.html', context)
```

Primeiro, importamos o template associado aos dados de que precisamos 1. A função topics() precisa de um parâmetro: o objeto request do Django recebido do servidor 2. Consultamos o banco de dados solicitando os objetos Topic, ordenados pelo atributo date_added 3. Atribuímos o queryset resultante a topics.

Em seguida, definimos um contexto que enviaremos para o template 4. Um *contexto* é um dicionário no qual as chaves são nomes que utilizaremos no template a fim de acessar os dados que queremos, e os valores são os dados que precisamos enviar para o template. Aqui, há um par chave-valor, que contém o conjunto de tópicos que serão exibidos na página. Ao criar uma página que usa dados, chamamos render() com o objeto request, o template que queremos usar e o dicionário context 5.

Template de tópicos

O template para a página de tópicos recebe o dicionário context, assim consegue usar os dados que topics() fornecem. Crie um arquivo chamado *topics.html* no mesmo diretório que *index.html*. Veja como podemos exibir os tópicos no template:

topics.html

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <p>Topics</p>

  <ul>
    {% for topic in topics %}
      <li>{{ topic.text }}</li>
    {% empty %}
      <li>No topics have been added yet.</li>
    {% endfor %}
  </ul>

{% endblock content %}
```

Utilizamos a tag `{% extends %}` para herdar de *base.html*, assim como fizemos na página inicial e, depois, abrimos um bloco `content`. O corpo dessa página contém uma lista com marcadores dos tópicos inseridos. No HTML padrão, uma lista com marcadores se chama *lista não ordenada* e é indicada pelas tags `<ul></ul>`. A tag de abertura `<ul>` inicia a lista de tópicos com os marcadores `1`.

Em seguida, usamos uma template tag, equivalente a um loop `for`, que percorre a lista do `topics` dicionário `context` `2`. Nos templates, o código usado difere bastante do código Python. O Python usa indentação para sinalizar quais linhas de uma instrução `for` fazem parte de um loop. Em um template, cada loop `for` precisa de uma tag explícita `{% endfor %}` sinalizando onde o final do loop ocorre. Por isso, em um template, veremos loops escritos assim:

```
{% for item in lista %}
  faz algo com cada item
{% endfor %}
```

Dentro do loop, queremos transformar cada tópico em um item na lista com marcadores. Para exibir uma variável em um template, insira o nome da variável em chaves duplas. As chaves duplas não aparecerão na página; apenas indicam ao Django que estamos usando uma variável de template. Assim, o código `{{ topic.text }}` `3` será

substituído pelo valor do atributo text do tópico atual em cada passagem pelo loop. A tag HTML indica um *item de lista*. Qualquer coisa entre essas tags, dentro de um par de tags , aparecerá como um item com marcadores na lista.

Além disso, utilizamos a template tag {% empty %} 4, que informa ao Django o que fazer se não houver itens na lista. Nesse caso, exibimos uma mensagem informando ao usuário que nenhum tópico foi adicionado ainda. As duas últimas linhas fecham o loop for 5 e, em seguida, encerram a lista com marcadores 6.

Agora, é necessário modificar o template base a fim de incluir um link para a página de tópicos. Adicione o seguinte código à *base.html*:

base.html

```
  <p>
1   <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
2   <a href="{% url 'learning_logs:topics' %}">Topics</a>
  </p>

  {% block content %}{% endblock content %}
```

Adicionamos um traço após o link para a página inicial 1 e, depois, adicionamos um link para a página de tópicos usando a template tag {% url %} novamente 2. Essa linha informa ao Django para gerar um link que corresponda ao padrão de URL com o nome 'topics' em *learning_logs/urls.py*.

Agora, ao atualizar a página inicial do navegador, veremos um link Topics. Ao clicar no link, vermos uma página semelhante à página mostrada na Figura 18.4.

Figura 18.4: Página de tópicos.

Páginas com tópicos individuais

Depois, é necessário criar uma página que tenha como foco um único tópico, mostrando o nome desse tópico e todas as entradas para ele. Vamos definir um novo padrão de URL, escrever uma view e criar um template. Vamos alterar também a página de tópicos para que cada item da lista com marcadores seja vinculado à página de tópicos correspondente.

Padrão de URL do tópico

O padrão de URL para a página do tópico é um pouco diferente dos padrões de URL anteriores, pois esse utilizará o atributo `id` do tópico a fim de sinalizar qual tópico foi solicitado. Por exemplo, se o usuário quiser ver a página de detalhes para o tópico Chess (em que o `id` é 1), o URL será *http://localhost:8000/topics/1/*. Vejamos um padrão que corresponda com esse URL, e que precisamos inserir em *learning_logs/urls.py*:

learning_logs/urls.py

```
    -- trecho de código omitido --
urlpatterns = [
    -- trecho de código omitido --
    # Página de detalhes para um único tópico
```

```
    path('topics/<int:topic_id>/', views.topic, name='topic'),
]
```

Examinaremos a string `'topics/<int:topic_id>/'` nesse padrão de URL. A primeira parte da string solicita que o Django procure URLs que tenham a palavra *topics* após a URL base. A segunda parte da string, `/<int:topic_id>/`, corresponde a um número inteiro entre duas barras e atribui o valor inteiro a um argumento chamado `topic_id`.

Ao encontrar um URL que corresponda a esse padrão, o Django chama a função view `topic()` com o valor atribuído a `topic_id` como argumento. Usaremos o valor de `topic_id` para obter o tópico correto dentro da função.

View do tópico

A função `topic()` precisa obter o tópico e todas as entradas associadas do banco de dados, assim como fizemos anteriormente no shell do Django:

views.py

```
  -- trecho de código omitido --
1 def topic(request, topic_id):
    """Mostra um único tópico e todas as suas entradas"""
2   topic = Topic.objects.get(id=topic_id)
3   entries = topic.entry_set.order_by('-date_added')
4   context = {'topic': topic, 'entries': entries}
5   return render(request, 'learning_logs/topic.html', context)
```

É a primeira função view que requer um parâmetro diferente do objeto `request`. A função aceita o valor coletado pela expressão `/<int:topic_id>/` e o atribui a `topic_id` `1`. Em seguida, usamos `get()` para acessar o tópico, assim como fizemos no shell do Django `2`. Depois, obtemos todas as entradas associadas a esse tópico e as ordenamos de acordo com `date_added` `3`. O sinal de menos na frente de `date_added` ordena os resultados em ordem inversa, que exibirá as entradas mais recentes primeiro. Armazenamos o tópico e as entradas no dicionário `contexto` `4` e chamamos `render()` com o objeto `request`, com o *modelo* topic.html e com o dicionário `context` `5`.

NOTA *As frases de código em* 2 *e* 3 *são chamadas de* queries*, pois consultam o banco de dados por meio de queries para obter informações específicas. Quando escrevemos queries como esse em nossos projetos, primeiro, recomenda-se testá-las no shell do Django. No shell, teremos um feedback mais rápido do que teríamos se criássemos uma view e um template para verificar os resultados em um navegador.*

Template do tópico

O template precisa exibir o nome do tópico e as entradas. É necessário também informar o usuário se ainda não foram inseridas entradas para esse tópico.

topic.html

```
{% extends 'learning_logs/base.html' %}

{% block content %}

1   <p>Topic: {{ topic.text }}</p>

    <p>Entries:</p>
2   <ul>
3     {% for entry in entries %}
        <li>
4         <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
5         <p>{{ entry.text|linebreaks }}</p>
        </li>
6     {% empty %}
        <li>There are no entries for this topic yet.</li>
      {% endfor %}
    </ul>

{% endblock content %}
```

Incrementamos *base.html*, como faremos em todas as páginas do projeto. Depois, mostramos o atributo text do tópico solicitado 1. A variável topic está disponível porque está incluída no dicionário context. Assim, iniciamos uma lista com marcadores 2 a fim de mostrar cada uma das entradas e percorrê-las com um loop 3, como fizemos com os tópicos anteriores.

Cada marcador enumera duas informações: o timestamp o texto completo de cada entrada. Para o timestamp 4, exibimos o valor do atributo date_added. Nos templates Django, uma linha vertical (|) representa um *filtro* de template – uma função que modifica o valor em uma variável de template durante o processo de renderização. O filtro date:'M d, Y H:i' exibe timestamps no formato *January 1, 2022 23:00*. A próxima linha exibe o valor do atributo text da entrada atual. O filtro linebreaks 5 assegura que entradas extensas de texto incluam quebras de linha em um formato compreendido pelos navegadores, em vez de mostrar um bloco ininterrupto de texto. Mais uma vez, recorremos à template tag {% empty %} 6 a fim de exibir uma mensagem informando ao usuário que nenhuma entrada foi inserida.

Links da página de tópicos

Antes de conferirmos a página do tópico em um navegador, é necessário modificar o template de tópicos para que cada tópico seja vinculado à página adequada. Vejamos a alteração que precisamos fazer em *topics.html*:

topics.html

```
-- trecho de código omitido --
    {% for topic in topics %}
      <li>
        <a href="{% url 'learning_logs:topic' topic.id %}">
          {{ topic.text }}</a></li>
      </li>
    {% empty %}
-- trecho de código omitido --
```

Usamos a template tag do URL para gerar o link adequado, com base no padrão de URL em learning_logs com o nome 'topic'. Esse padrão de URL exige um argumento topic_id, então adicionamos o atributo topic.id à template tag do URL. Agora, cada tópico na lista de tópicos é um link para uma página de tópico, como *http://localhost:8000/topics/1/*.

Ao atualizamos a página de tópicos e clicarmos em um tópico, devemos ver uma página que se parece com a da Figura 18.5.

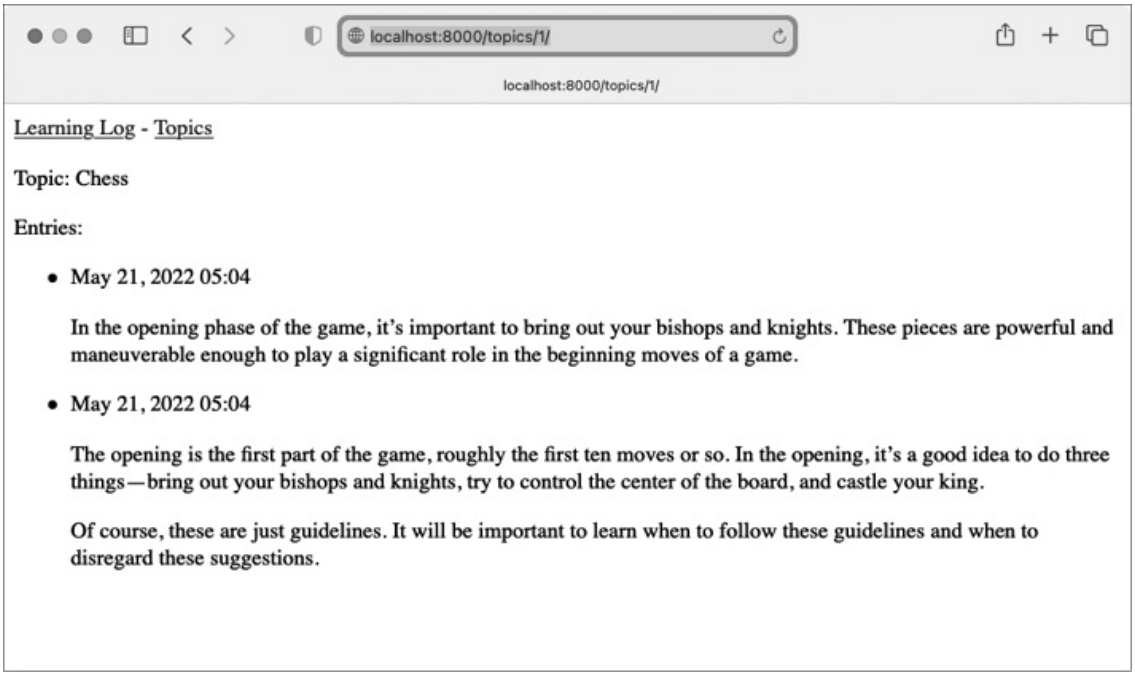


Figura 18.5: A página de detalhes de um único tópico, mostrando todas as entradas para um tópico.

NOTA *Existe uma diferença sutil, porém importante, entre* topic.id *e* topic_id*. A expressão* topic.id *examina um tópico e acessa o valor do ID correspondente. No código, a variável* topic_id *é uma referência a esse ID. Caso se depare com erros ao manipular IDs, lembre-se de usar essas expressões de forma adequada.*

FAÇA VOCÊ MESMO

18.7 Documentação do template: Confira rapidamente a documentação dos templates Django em https://docs.djangoproject.com/en/4.1/ref/templates. Você pode consultá-la quando estiver desenvolvendo os próprios projetos.

18.8 Páginas da pizzaria: Adicione uma página ao projeto da pizzaria do Exercício 18.6 (página *483*) que mostre o nome das pizzas disponíveis. Em seguida, vincule cada nome de pizza a uma página que exiba os ingredientes de cada pizza. Não se esqueça de recorrer à herança do template para criar suas páginas de forma eficiente.

Recapitulando

Neste capítulo, aprendemos como começar a desenvolver aplicações web simples com o framework Django. Vimos uma breve especificação do projeto, instalamos o Django em um ambiente virtual, configuramos um projeto e verificamos se o projeto foi devidamente configurado. Configuramos uma aplicação e definimos modelos para representar os dados dessa aplicação. Vimos como os bancos de dados e como o Django nos ajuda a migrar banco de dados após alterarmos modelos. Criamos um superusuário para o site admin e usamos esse site para fornecer alguns dados iniciais.

Além do mais, exploramos o shell do Django, que possibilita trabalhar com os dados do projeto em uma sessão de terminal. Aprendemos a definir URLs, criar funções view e escrever templates para criar páginas de um site. Recorremos também à herança de templates a fim de simplificar a estrutura de templates individuais e facilitar a modificação do site à medida que o projeto evolui.

No Capítulo 19, desenvolveremos páginas intuitivas e fáceis de usar que possibilitam aos usuários adicionar entradas e tópicos novos e editar entradas existentes sem passar pelo site admin. Veremos também um sistema de registro de usuário, que possibilita aos usuários criarem uma conta e seu próprio registro de aprendizagem. Trata-se do cerne de uma aplicação web – a capacidade de desenvolver uma aplicação com o qual qualquer número de usuários consegue interagir.

CAPÍTULO 19
Contas de usuário

No cerne de uma aplicação web reside a capacidade de qualquer usuário, em qualquer lugar do mundo, de registrar uma conta em sua aplicação e começar a usá-la. Neste capítulo, desenvolveremos formulários para que os usuários possam adicionar os próprios tópicos e entradas e editar as entradas existentes. Aprenderemos também como o Django se protege de ataques habituais contra páginas baseadas em formulários. Desse modo, não precisamos ficar pensando muito tempo na segurança de nossas aplicações.

Além disso, implementaremos um sistema de autenticação de usuário. Vamos desenvolver uma página de cadastro para que os usuários criem contas e, depois, vamos restringir o acesso de determinadas páginas somente aos usuários logados. Em seguida, modificaremos algumas funções view para que os usuários possam visualizar apenas os próprios dados. Aprenderemos a manter os dados de nossos usuários seguros e protegidos.

Permitindo que os usuários forneçam dados

Primeiro, antes de arquitetarmos um sistema de autenticação para criar contas, adicionaremos algumas páginas que permitem aos usuários fornecer os próprios dados. Vamos conceder aos usuários a capacidade de adicionar entradas e tópicos novos e editar as entradas anteriores.

Atualmente, apenas um superusuário consegue inserir dados por

meio do site admin. Como não queremos que os usuários interajam com o site admin, recorreremos às ferramentas de criação de formulários do Django para criar páginas que possibilitem aos usuários fornecer dados.

Adicionando novos tópicos

Começaremos permitindo que os usuários adicionem um tópico novo. Adicionar uma página baseada em formulário funciona do mesmo jeito que adicionar as páginas que já criamos: definimos um URL, escrevemos uma função view e um template. A diferença significativa é a adição de um módulo novo chamado *forms.py*, que armazenará os formulários.

ModelForm para tópicos

Qualquer página que permita a um usuário inserir e fornecer informações em uma página web envolve um elemento HTML chamado *formulário*. Quando os usuários fornecem informações, é necessário *validarmos* se essas informações fornecidas representam o tipo adequado de dados e não dados maliciosos, como um código desenvolvido para inativar nosso servidor. Em seguida, precisamos processar e salvar as informações válidas no local adequado no banco de dados. O Django automatiza boa parte desses processos.

No Django, o jeito mais simples de criar um formulário é com o ModelForm, que usa as informações dos modelos que definimos no Capítulo 18 para criar automaticamente um formulário. Vamos escrever nosso primeiro formulário no arquivo *forms.py*, que deve ser criado no mesmo diretório que *models.py*:

forms.py

```
from django import forms

from .models import Topic

class TopicForm(forms.ModelForm):
    class Meta:
```

```
2       model = Topic
3       fields = ['text']
4       labels = {'text': ''}
```

Primeiro, importamos o módulo forms e o modelo com o qual trabalharemos, Topic. Em seguida, definimos uma classe chamada TopicForm, que herda de forms.ModelForm 1.

A versão mais simples de ModelForm consiste em uma classe Meta aninhada que informa ao Django em qual modelo basear o formulário e quais campos incluir no formulário. Aqui, especificamos que o formulário deve ser baseado no modelo Topic 2, e que deve incluir apenas o campo text 3. A string vazia no dicionário de rótulos instrui o Django a não gerar um rótulo para o campo text 4.

URL new_topic

O URL para uma página nova deve ser breve e descritivo. Quando quiser adicionar um tópico novo, direcionaremos o usuário para *http://localhost:8000/new_topic/*. Vejamos o padrão de URL para a página new_topic; adicione esse código a *learning_logs/urls.py*:

learning_logs/urls.py

```
-- trecho de código omitido --
urlpatterns = [
    -- trecho de código omitido --
    # Página para adicionar um tópico novo
    path('new_topic/', views.new_topic, name='new_topic'),
]
```

Esse padrão de URL envia requisições para a função view new_topic(), que escreveremos a seguir.

Função view new_topic()

A função new_topic() precisa lidar com duas situações diferentes: requisições iniciais para a página new_topic, caso em que deve mostrar um formulário em branco; e o processamento de quaisquer dados fornecidos no formulário. Após os dados de um formulário enviado serem processados, é necessário redirecionar o usuário à página

topics:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic
from .forms import TopicForm

-- trecho de código omitido --
def new_topic(request):
    """Adiciona um tópico novo"""
    if request.method != 'POST':
        # Nenhum dado enviado; cria um formulário em branco
        form = TopicForm()
    else:
        # Dados POST enviados; processa os dados
        form = TopicForm(data=request.POST)
        if form.is_valid():
            form.save()
            return redirect('learning_logs:topics')

    # Exibe um formulário em branco ou inválido
    context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
```

Importamos a função `redirect`, que usaremos para redirecionar o usuário à página `topics` depois que ele inserir seu tópico. Importamos também o formulário que acabamos de escrever, `TopicForm`.

Requisições GET e POST

Ao desenvolvermos aplicações, usaremos os dois tipos principais de requisição: GET e POST. Usamos requisições *GET* para páginas que leem apenas os dados do servidor. Em geral, usamos requisições *POST* quando o usuário precisa fornecer informações por meio de um formulário. Vamos especificar o método POST para processar todos os nossos formulários. (Existem alguns outros tipos de requisições, mas não as usaremos neste projeto.)

A função `new_topic()` recebe o objeto `request` como parâmetro. A princípio, quando o usuário solicita essa página, o navegador dele enviará uma requisição GET. Após o usuário preencher e enviar o

formulário, o navegador dele enviará uma requisição POST. Dependendo da requisição, saberemos se o usuário está solicitando um formulário em branco (GET) ou nos pedindo para processar um formulário preenchido (POST).

Usamos um teste if para constatar se o método de requisição é GET ou POST 1. Caso o método de requisição não seja POST, a requisição provavelmente será GET. Desse modo, precisamos retornar um formulário em branco. (Se for outro tipo de requisição, ainda é seguro retornar um formulário em branco.) Criamos uma instância de TopicForm 2, atribuímos à variável form e enviamos o formulário para o template no dicionário context 7. Já que não incluímos argumentos ao instanciar TopicForm, o Django cria um formulário em branco que o usuário pode preencher.

Caso o método de requisição seja POST, o bloco else é executado e processa os dados enviados no formulário. Criamos uma instância de TopicForm 3 e passamos os dados fornecidos pelo usuário, que são atribuídos a request.POST. O objeto form retornado contém as informações enviadas pelo usuário.

Não podemos salvar as informações enviadas ao banco de dados até verificarmos se são válidas ou não 4. O método is_valid() verifica se todos os campos obrigatórios foram preenchidos (todos os campos em um formulário são obrigatórios por padrão) e se os dados fornecidos correspondem aos tipos de campo esperados – por exemplo, se o comprimento de text é inferior a 200 caracteres, conforme especificado em *models.py* no Capítulo 18. Essa validação automática faz grande parte do trabalho. Se tudo for válido, podemos chamar save() 5, que grava os dados do formulário no banco de dados.

Assim que salvarmos os dados, podemos sair dessa página. A função redirect() recebe o nome de uma view e redireciona o usuário para a página associada a essa view. Aqui, utilizamos redirect() a fim de redirecionar o navegador do usuário para a página topics 6, em que o

usuário deve ver o tópico que acabou de inserir na lista de tópicos.

A variável context é definida no final da função view e a página é renderizada usando o template *new_topic.html*, que criaremos a seguir. Esse código é inserido fora de qualquer bloco if; será executado se um formulário em branco foi criado, e será executado se um formulário enviado for considerado inválido. Um formulário inválido terá algumas mensagens de erro padrão para ajudar o usuário a enviar dados aceitáveis.

Template new_topic

Agora, criaremos um template novo chamado *new_topic.html* para exibir o formulário que acabamos de desenvolver:

new_topic.html

```
{% extends "learning_logs/base.html" %}

{% block content %}
  <p>Add a new topic:</p>

1  <form action="{% url 'learning_logs:new_topic' %}" method='post'>
2    {% csrf_token %}
3    {{ form.as_div }}
4    <button name="submit">Add topic</button>
   </form>

{% endblock content %}
```

Esse modelo estende *base.html*, por isso tem a mesma estrutura base que o restante das páginas no Registro de Aprendizagem. Usamos as tags <form></form> para definir um formulário HTML 1. O argumento action informa ao navegador para onde enviar os dados fornecidos no formulário; nesse caso, nós os reenviamos para a função view new_topic(). O argumento method instrui o navegador a enviar os dados como uma requisição POST.

O Django usa a template tag {% csrf_token %} 2 a fim de impedir que cibercriminosos usem o formulário para obter acesso não autorizado ao servidor. (Esse tipo de ataque é conhecido como *cross-site*

request forgery [falsificação de solicitação entre sites].) Em seguida, exibimos o formulário; aqui, é possível ver como o Django simplifica determinadas tarefas, como exibir um formulário. Basta incluir a variável de template {{ form.as_div }} a fim de que o Django crie todos os campos necessários para exibir automaticamente o formulário 3. O modificador as_div instrui o Django a renderizar todos os elementos do formulário como elementos HTML ; uma maneira simples de exibir elegantemente o formulário.

Como o Django não cria um botão de envio para formulários, definimos um antes de fechar o formulário 4.

Link para a página new_topic

Vamos incluir um link para a página new_topic na página topics:

topics.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p>Topics</p>

  <ul>
    -- trecho de código omitido --
  </ul>

  <a href="{% url 'learning_logs:new_topic' %}">Add a new topic</a>

{% endblock content %}
```

Insira o link após a lista de tópicos existentes. A Figura 19.1 mostra o formulário resultante; tente usar o formulário para adicionar alguns tópicos novos.

Figura 19.1: A página para adicionar um tópico novo.

Adicionando novas entradas

Como agora o usuário consegue adicionar um tópico novo, ele também vai querer adicionar entradas novas. Mais uma vez, definiremos um URL, escreveremos uma função view e um template e incluiremos o link para a página. No entanto, adicionaremos primeiro outra classe ao *forms.py*.

ModelForm para entradas

É necessário criar um formulário associado ao modelo `Entry`, mas, dessa vez, vamos personalizá-lo um pouco mais do que o `TopicForm`:

forms.py

```
from django import forms

from .models import Topic, Entry

class TopicForm(forms.ModelForm):
    -- trecho de código omitido --

class EntryForm(forms.ModelForm):
    class Meta:
```

```
        model = Entry
        fields = ['text']
1       labels = {'text': ''}
2       widgets = {'text': forms.Textarea(attrs={'cols': 80})}
```

Atualizamos a instrução import para incluir Entry e Topic. Criamos uma classe nova chamada EntryForm que herda de forms.ModelForm. A classe EntryForm tem uma classe Meta aninhada listando o modelo em que se baseia e o campo a ser incluído no formulário. Mais uma vez, atribuímos ao campo "text" um rótulo em branco 1.

Incluímos o atributo widgets em EntryForm 2.Um *widget* é um elemento de formulário HTML, como uma caixa de texto de linha única, área de texto de várias linhas ou lista suspensa. Ao incluir o atributo widgets, podemos substituir as opções default de widget do Django. Aqui, estamos instruindo o Django a utilizar um elemento a forms.Textarea com uma largura de 80 colunas, em vez das 40 colunas default. Com isso, os usuários têm espaço suficiente para fornecer uma entrada significativa.

URL new_entry

Como entradas novas devem ser associadas a um tópico específico, precisamos incluir um argumento topic_id no URL para adicionar uma entrada nova. Vejamos o URL que adicionamos a *learning_logs/urls.py*:

learning_logs/urls.py

```
-- trecho de código omitido --
urlpatterns = [
    -- trecho de código omitido --
    # Página para adicionar uma entrada nova
    path('new_entry/<int:topic_id>/', views.new_entry, name='new_entry'),
]
```

Esse padrão de URL corresponde a qualquer URL com o formulário *http://localhost:8000/new_entry/id/*, em que *id* é um número correspondente ao ID do tópico. O código <int:topic_id> coleta um valor numérico e o atribui à variável topic_id. Quando um URL

correspondente a esse padrão é solicitado, o Django envia a requisição e o ID do tópico à função view new_entry().

Função view new_entry()

A função view para new_entry é bem parecida com a função que adiciona um tópico novo. Adicione o seguinte código ao arquivo *views.py*:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic
from .forms import TopicForm, EntryForm

-- trecho de código omitido --
def new_entry(request, topic_id):
    """Adiciona uma entrada nova para um tópico específico"""
1   topic = Topic.objects.get(id=topic_id)

2   if request.method != 'POST':
        # Nenhum dado enviado; cria um formulário em branco
3       form = EntryForm()
    else:
        # Dados POST enviados; processa os dados
4       form = EntryForm(data=request.POST)
        if form.is_valid():
5           new_entry = form.save(commit=False)
6           new_entry.topic = topic
            new_entry.save()
7           return redirect('learning_logs:topic', topic_id=topic_id)

    # Exibe um formulário em branco ou inválido
    context = {'topic': topic, 'form': form}
    return render(request, 'learning_logs/new_entry.html', context)
```

Atualizamos a instrução import para incluir EntryForm que acabamos de criar. A definição de new_entry() tem um parâmetro topic_id para armazenar o valor que recebe do URL. Como precisaremos do tópico para renderizar a página e processar os dados do formulário, usamos topic_id para obter o objeto de tópico adequado 1.

Depois, verificamos se o método de requisição é POST ou GET 2. O

bloco if é executado se for uma requisição GET, e criamos uma instância em branco a partir de EntryForm 3.

Caso o método de requisição seja POST, processaremos os dados criando uma instância a partir de EntryForm, preenchida com os dados POST do objeto request 4. Em seguida, verificamos se o formulário é válido. Em caso afirmativo, é necessário definir o atributo topic do objeto de entrada antes de salvá-lo no banco de dados. Ao chamarmos save(), incluímos o argumento commit=False 5 que instrui o Django a criar um objeto novo de entrada e atribuí-lo a new_entry, sem salvá-lo no banco de dados ainda. Definimos o atributo topic de new_entry para o tópico que extraímos do banco de dados no início da função 6. Depois, chamamos save() sem argumentos, salvando a entrada no banco de dados com o tópico associado correto.

A chamada redirect() exige dois argumentos: o nome da view para a qual queremos redirecionar e o argumento que a função view exige 7. Aqui, estamos redirecionando para topic(), que precisa do argumento topic_id. Essa view renderiza a página de tópico, na qual o usuário inseriu uma entrada, e ele deve ver sua entrada nova na lista de entradas.

No final da função, criamos um dicionário context e renderizamos a página com o template *new_entry.html*. Esse código será executado para um formulário em branco ou para um formulário que foi enviado, mas que se revela inválido.

Template para new_entry

Conforme podemos observar no código a seguir, o template para new_entry é semelhante ao template para new_topic:

new_entry.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

1  <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>
```

```
    <p>Add a new entry:</p>
2   <form action="{% url 'learning_logs:new_entry' topic.id %}" method='post'>
     {% csrf_token %}
     {{ form.as_div }}
     <button name='submit'>Add entry</button>
    </form>

  {% endblock content %}
```

Mostramos o tópico na parte superior da página 1. Assim, o usuário pode ver em qual tópico está adicionando uma entrada. O tópico também se comporta como um link que retorna à página principal desse tópico.

O argumento action do formulário inclui o valor topic.id no URL a fim de que a função view consiga associar a entrada nova ao tópico correto 2. Desconsiderando isso, esse template se parece com *new_topic.html*.

Link para a página New_entry

É necessário incluir um link para a página new_entry de cada página de tópico, no template de tópico:

topic.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p>Topic: {{ topic }}</p>

  <p>Entries:</p>
  <p>
    <a href="{% url 'learning_logs:new_entry' topic.id %}">Add new entry</a>
  </p>

  <ul>
  -- trecho de código omitido --
  </ul>

{% endblock content %}
```

Inserimos o link para adicionar entradas antes de mostrá-las, porque

adicionar uma entrada nova será a ação mais comum dessa página. A Figura 19.2 mostra a página new_entry. Agora, os usuários podem adicionar tópicos novos e quantas entradas quiserem para cada tópico. Teste a página new_entry: adicione algumas entradas a tópicos que você criou.

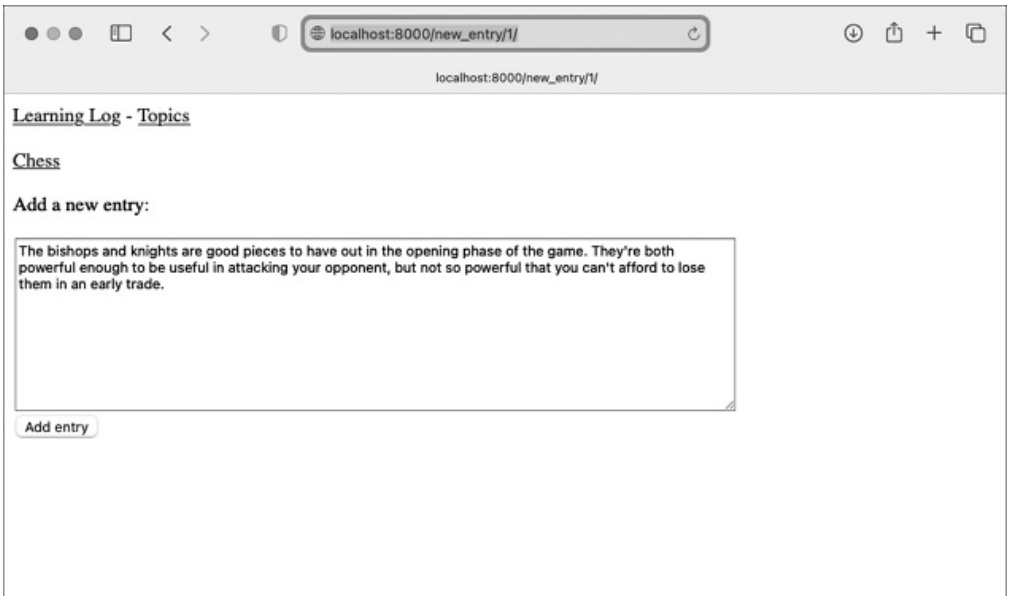


Figura 19.2: A página new_entry.

Editando as entradas

Agora, desenvolveremos uma página para que os usuários consigam editar as entradas que adicionaram.

URL edit_entry

É necessário que o URL da página passe o ID da entrada a ser editada. Vejamos *learning_logs/urls.py*:

urls.py

```
-- trecho de código omitido --
urlpatterns = [
    -- trecho de código omitido --
    # Página para editar uma entrada
    path('edit_entry/<int:entry_id>/', views.edit_entry, name='edit_entry'),
]
```

Esse padrão de URL corresponde aos URLs como *http://localhost:8000/edit_entry/id/*. Em nosso caso, o valor de *id* é atribuído ao parâmetro entry_id. O Django envia requisições que correspondem com esse formato à função view edit_entry().

Função view edit_entry()

Quando a página edit_entry recebe uma requisição GET, a função edit_entry() retorna um formulário para editar a entrada. Quando a página recebe uma requisição POST com o texto de entrada revisado, salva o texto modificado no banco de dados:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic, Entry
from .forms import TopicForm, EntryForm
-- trecho de código omitido --

def edit_entry(request, entry_id):
    """Edita uma entrada existente"""
1   entry = Entry.objects.get(id=entry_id)
    topic = entry.topic

    if request.method != 'POST':
        # Requisição inicial; pré-preenche formulário com a entrada atual
2       form = EntryForm(instance=entry)
    else:
        # Dados POST enviados; processa os dados
3       form = EntryForm(instance=entry, data=request.POST)
        if form.is_valid():
4           form.save()
5           return redirect('learning_logs:topic', topic_id=topic.id)

    context = {'entry': entry, 'topic': topic, 'form': form}
    return render(request, 'learning_logs/edit_entry.html', context)
```

Inicialmente, importamos o modelo Entry. Em seguida, obtemos o objeto de entrada que o usuário quer editar 1 e o tópico associado a essa entrada. No bloco if, executado para uma requisição GET, criamos uma instância a partir de EntryForm com o argumento

instance=entry 2. Esse argumento informa ao Django para criar o formulário, pré-preenchido com informações do objeto de entrada existente. O usuário verá seus dados existentes e poderá editá-los.

Ao processar uma requisição POST, passamos os argumentos instance=entry e data=request.POST 3. Esses argumentos instruem o Django a criar uma instância do formulário com base nas informações associadas ao objeto de entrada existente, atualizadas com quaisquer dados relevantes do request.POST. Depois, analisamos se o formulário é válido; se for, chamamos save() sem argumentos, pois a entrada já está associada ao tópico correto 4. Em seguida, redirecionamos para a página topic, em que o usuário deve ver a versão atualizada da entrada que editou 5.

Se estivermos mostrando um formulário inicial para editar a entrada ou se o formulário enviado for inválido, criamos o dicionário context e renderizamos a página com o template *edit_entry.html*.

Template para edit_entry

A seguir, criamos um template edit_entry.html semelhante ao *new_entry.html*:

edit_entry.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

  <p>Edit entry:</p>

1 <form action="{% url 'learning_logs:edit_entry' entry.id %}" method='post'>
   {% csrf_token %}
   {{ form.as_div }}
2   <button name="submit">Save changes</button>
  </form>

{% endblock content %}
```

O argumento action reenvia o formulário à função edit_entry() para

processamento 1. Já que incluímos entry.id como um argumento na tag {% url %}, a função view pode modificar o objeto correto de entrada. Rotulamos o botão enviar como Save changes. Desse modo, lembramos ao usuário que ele está salvando edições, não criando uma entrada nova 2.

Link para a página edit_entry

Agora, é necessário incluir um link para a página edit_entry para cada entrada na página do tópico:

topic.html

```
-- trecho de código omitido --
    {% for entry in entries %}
      <li>
        <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
        <p>{{ entry.text|linebreaks }}</p>
        <p>
          <a href="{% url 'learning_logs:edit_entry' entry.id %}">
           Edit entry</a></p>
      </li>
-- trecho de código omitido --
```

Incluímos o link de edição após a exibição da data e do texto de cada entrada. Utilizamos a template tag {% url %} a fim de determinar o URL para o padrão de URL nomeado edit_entry, com o atributo ID da entrada atual no loop (entry.id). O texto do link Edit entry aparece após cada entrada na página. A Figura 19.3 mostra como é a página do tópico com esses links.

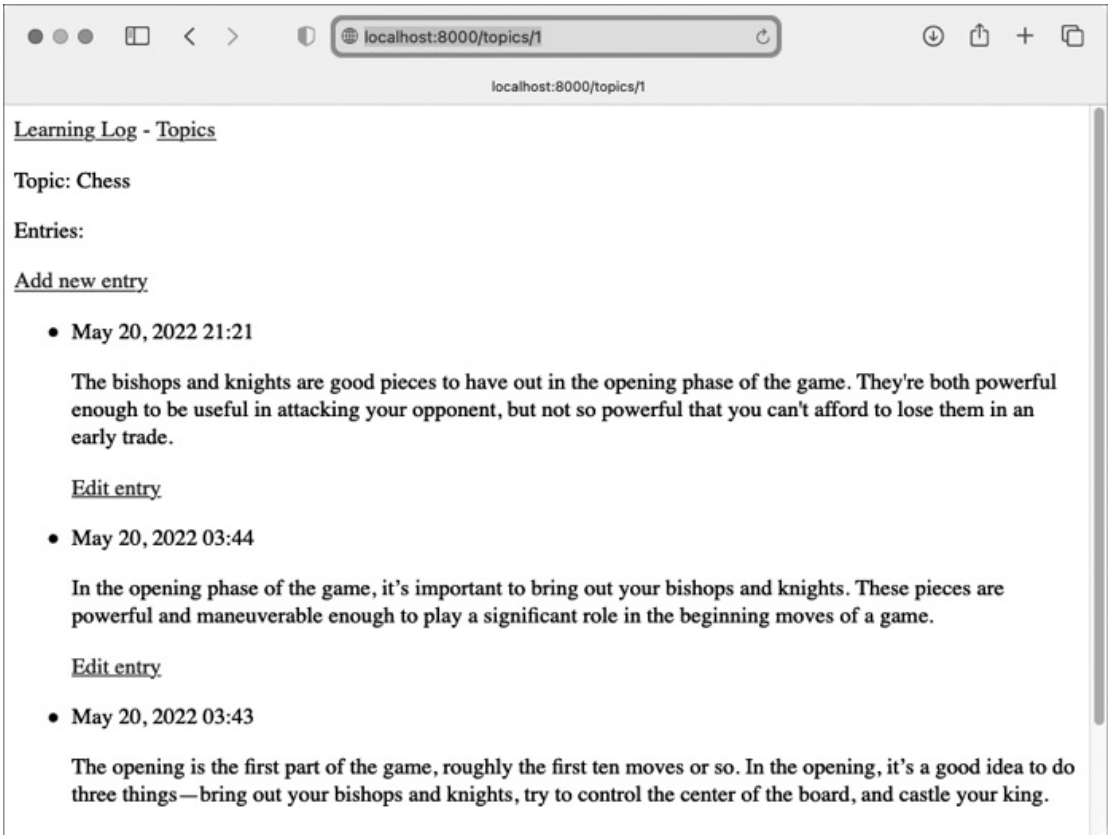


Figura 19.3: Agora, cada entrada tem um link para editá-la.

Agora, o Registro de Aprendizagem tem boa parte das funcionalidades necessárias. Os usuários conseguem adicionar tópicos e entradas, além de poderem ler qualquer conjunto de entradas que quiserem. Na próxima seção, implementaremos um sistema de registro de usuários a fim de que qualquer pessoa consiga criar uma conta no Registro de Aprendizagem e criar o próprio conjunto de tópicos e entradas.

FAÇA VOCÊ MESMO

19.1 Blog: Inicie um projeto Django novo chamado *Blog*. Desenvolva uma aplicação chamada *blogs*, com um modelo que represente um blog geral e um modelo que represente uma postagem individual do blog. Atribua a cada modelo um conjunto adequado de campos. Crie um superusuário para o projeto e use o site admin a fim de criar um blog e algumas breves postagens. Crie uma página inicial que mostre todas as postagens em ordem apropriada.

Desenvolva páginas para criar um blog, postagens novas e para editar as postagens existentes. Use as páginas que criou para garantir que realmente funcionam.

Configurando contas de usuário

Nesta seção, desenvolveremos um sistema de registro e autorização de usuários para que as pessoas possam criar uma conta, fazer login e logout. Desenvolveremos uma aplicação nova com todas as funcionalidades relacionadas às tarefas de usuários. Recorreremos ao sistema default de autenticação de usuário do Django para conseguirmos fazer o máximo possível. Além disso, modificaremos ligeiramente o modelo Topic para que cada tópico pertença a um determinado usuário.

Aplicação accounts

Começaremos desenvolvendo uma aplicação nova chamada accounts com o comando startapp:

```
(ll_env)learning_log$ python manage.py startapp accounts
(ll_env)learning_log$ ls
1 accounts db.sqlite3 learning_logs ll_env ll_project manage.py
(ll_env)learning_log$ ls accounts
2 __init__.py admin.py apps.py migrations models.py tests.py views.py
```

Como o sistema de autenticação default é criado com base no conceito de contas de usuário, usar o nome accounts facilita a integração com o sistema default. O comando startapp mostrado aqui cria um diretório novo chamado *accounts* 1 com uma estrutura idêntica à aplicação learning_logs 2.

Adicionando accounts a settings.py

É necessário adicionar nossa aplicação nova a INSTALLED_APPS em *settings.py*, da seguinte forma:

settings.py

```
-- trecho de código omitido ---
INSTALLED_APPS = [
    # Minhas aplicações
    'learning_logs',
    'accounts',

    # Aplicações default do Django
```

```
    -- trecho de código omitido --
]
-- trecho de código omitido --
```

Agora, o Django incluirá a aplicação accounts no projeto geral.

Incluindo os URLs da aplicação accounts

Em seguida, precisamos modificar o root de *urls.py* para que inclua os URLs que escreveremos para a aplicação accounts:

ll_project/urls.py

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('accounts.urls')),
    path('', include('learning_logs.urls')),
]
```

Adicionamos uma linha para incluir o arquivo *urls.py* de accounts. Essa linha corresponderá a qualquer URL que comece com a palavra *accounts*, como *http://localhost:8000/accounts/login/*.

Página de login

Primeiro, implementaremos uma página de login. Como vamos utilizar a view login default que o Django fornece, o padrão de URL para essa aplicação será um pouco diferente. Crie um arquivo novo *urls.py* no diretório *ll_project/accounts/*e adicione o seguinte código dentro dele:

accounts/urls.py

```
"""Define padrões de URL para contas"""

from django.urls import path, include

app_name = 'accounts'
urlpatterns = [
    # Inclui URLs de autenticação default
    path('', include('django.contrib.auth.urls')),
]
```

Importamos a função path e, em seguida, importamos a função include para que possamos incluir alguns URLs de autenticação default definidos pelo Django. Esses URLs default incluem padrões de URL nomeados, como 'login' e 'logout'. Definimos a variável app_name como 'accounts' a fim de que o Django consiga distinguir esses URLs dos URLs pertencentes a outras aplicações. Mesmo os URLs default fornecidos pelo Django, quando incluídos no arquivo *urls.py* da aplicação accounts, serão acessíveis por meio do namespace de accounts.

O padrão da página de login corresponde ao URL *http://localhost:8000/accounts/login/*. Quando o Django lê esse URL, a palavra *accounts* informa ao Django para procurar em *accounts/urls.py*, e *login* instrui a enviar requisições para a view login default do Django.

Template de login

Quando o usuário solicita a página de login, embora o Django use uma função view default, ainda precisamos fornecer um template para a página. Como as views default de autenticação procuram templates dentro de uma pasta chamada *registration*, é necessário criar essa pasta. Dentro do diretório *ll_project/accounts/*, crie um diretório chamado *templates*; dentro dele, crie outro diretório chamado *registration*. Vejamos o template *login.html*, que deve ser salvo em *ll_project/accounts/templates/registration*:

login.html

```
{% extends 'learning_logs/base.html' %}

{% block content %}

1  {% if form.errors %}
     <p>Your username and password didn't match. Please try again.</p>
   {% endif %}

2  <form action="{% url 'accounts:login' %}" method='post'>
     {% csrf_token %}
```

```
3     {{ form.as_div }}

4     <button name="submit">Log in</button>
    </form>

  {% endblock content %}
```

Esse template incrementa *base.html* para garantir que a página de login tenha a mesma aparência que o restante do site. Perceba que o template em uma aplicação pode herdar de um template de outra aplicação.

Caso o atributo errors do formulário esteja definido, exibimos uma mensagem de erro 1, informando que a combinação de nome e de usuário e de senha não corresponde a nenhuma informação armazenada no banco de dados.

Queremos que view de login processe o formulário, assim definimos o argumento action como URL da página de login 2. A view de login envia um objeto form para o template, e cabe a nós exibir o formulário 3 e adicionar um botão de envio 4.

Configurando LOGIN_REDIRECT_URL

Quando um usuário faz login com sucesso, o Django precisa saber para onde enviar esse usuário. Controlamos esse processo no arquivo settings.

Adicione o seguinte código ao final de *settings.py*:

settings.py

```
  -- trecho de código omitido --
  # Minhas configurações
  LOGIN_REDIRECT_URL = 'learning_logs:index'
```

Já que temos todas as configurações default em *settings.py*, é útil marcar a seção em que estamos adicionando configurações novas. A primeira configuração nova que adicionaremos é LOGIN_REDIRECT_URL, que informa ao Django para qual URL redirecionar após uma tentativa de login bem-sucedida.

Link da página de login

Adicionaremos o link de login à *base.html* para que apareça em todas as páginas. Como não queremos que o link seja exibido quando o usuário já estiver logado, vamos aninhá-lo dentro de uma tag `{% if %}`:

base.html

```
<p>
  <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
  <a href="{% url 'learning_logs:topics' %}">Topics</a> -
1 {% if user.is_authenticated %}
2   Hello, {{ user.username }}.
  {% else %}
3   <a href="{% url 'accounts:login' %}">Log in</a>
  {% endif %}
</p>

{% block content %}{% endblock content %}
```

No sistema de autenticação do Django, cada template tem um objeto `user` disponível que sempre tem um conjunto de atributos `is_authenticated`: o atributo é `True` se o usuário estiver logado e `False` se não estiver. Esse atributo possibilita exibir uma mensagem para usuários autenticados e outra para usuários não autenticados.

Aqui, exibimos uma mensagem aos usuários atualmente logados `1`. Usuários autenticados têm um conjunto adicional de atributos `username`, que utilizamos para personalizar a mensagem e lembrar ao usuário que está logado `2`. Para usuários que não foram autenticados, exibimos um link para a página de login `3`.

Usando a página de login

Como já configuramos uma conta de usuário, vamos logar para ver se a página funciona. Acesse *http://localhost:8000/admin/*. Se ainda estiver logado como administrador, procure um link de **logout** no cabeçalho e clique nele.

Ao fazer logout, acesse *http://localhost:8000/accounts/login/*. Você

deve ver uma página de login semelhante à mostrada na Figura 19.4. Digite o nome de usuário e a senha que configurou anteriormente, e você deve ser redirecionado à página inicial. Na página inicial, o cabeçalho deve exibir uma mensagem personalizada com seu nome de usuário.

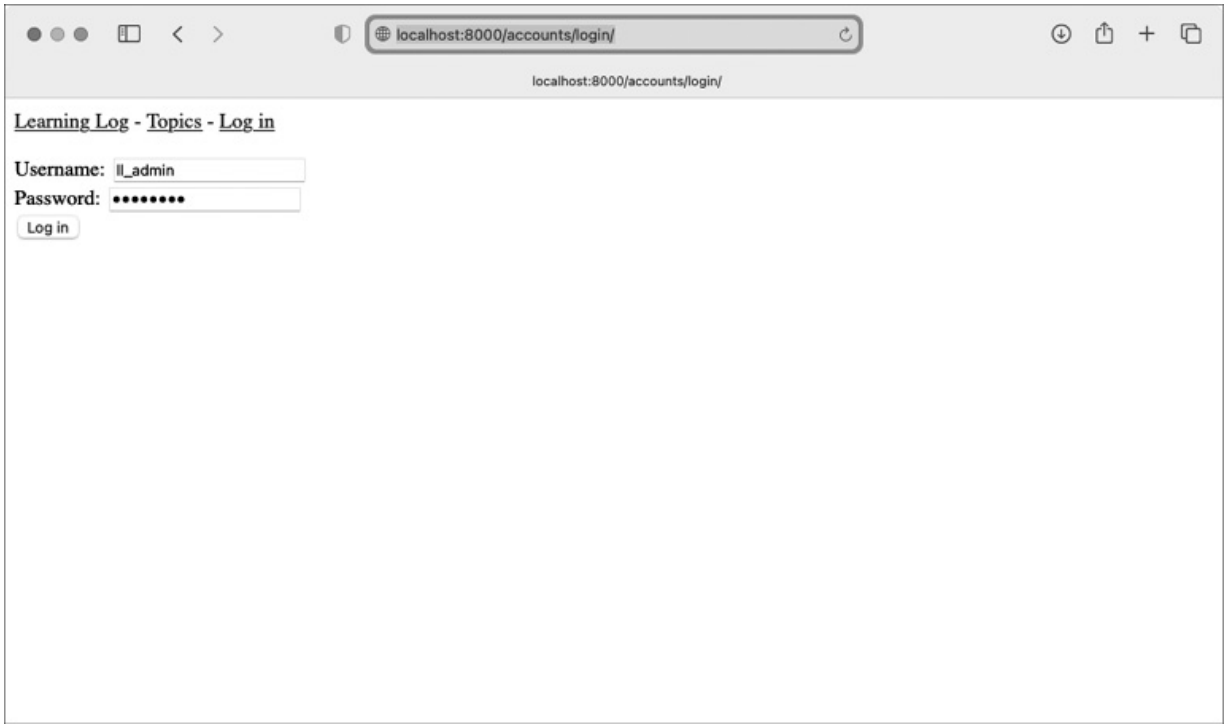


Figura 19.4: A página de login.

Fazendo logout

Agora, é necessário fornecer um modo de os usuários fazerem logout. As requisições de logout devem ser enviadas como requisições POST, logo adicionaremos um pequeno formulário de logout a *base.html*. Quando clicam no botão de logout, os usuários são direcionados para uma página, que confirma que fizeram logout.

Adicionando um formulário de logout à base.html

Adicionaremos o formulário de logout à *base.html* para que fique disponível em todas as páginas. Vamos inseri-lo em outro bloco `if`. Assim, apenas os usuários que já estão logados podem vê-lo:

base.html

```
-- trecho de código omitido --
{% block content %}{% endblock content %}

{% if user.is_authenticated %}
  <hr />
  <form action="{% url 'accounts:logout' %}" method='post'>
    {% csrf_token %}
    <button name='submit'>Log out</button>
  </form>
{% endif %}
```

O default de URL padrão para logout é 'accounts/logout/'. No entanto, a requisição deve ser enviada como uma requisição POST; caso contrário, hackers podem realizar ataques sobrecarregando facilmente as requisições de logout. Para fazer a requisição de logout usar POST, definimos um formulário simples.

Inserimos o formulário na parte inferior da página, abaixo de um elemento de régua horizontal (<hr />) 1. É um modo fácil de sempre manter o botão de logout em uma posição consistente, abaixo de qualquer outro conteúdo da página. O formulário propriamente dito tem o URL de logout como seu argumento action e 'post' como o método de requisição 2. No Django, cada formulário precisa incluir {% csrf_token %}, até mesmo um formulário simples como esse. Esse formulário está vazio, exceto pelo botão enviar.

Configurando LOGOUT_REDIRECT_URL

Quando o usuário clica no botão de logout, o Django precisa saber para onde enviá-los. Controlamos esse comportamento em *settings.py*:

settings.py

```
-- trecho de código omitido --
# Minhas configurações
LOGIN_REDIRECT_URL = 'learning_logs:index'
LOGOUT_REDIRECT_URL = 'learning_logs:index'
```

Aqui, a configuração LOGOUT_REDIRECT_URL instrui o Django a

redirecionar os usuários deslogados à página inicial. É um modo simples de confirmar que eles foram desconectados, já que não devem ver mais o nome de usuário após fazerem o logout.

Página de cadastro

A seguir, criaremos uma página para que usuários novos possam se cadastrar na aplicação. Utilizaremos o UserCreationForm default do Django, mas escreveremos nossa própria função view e template.

URL para register

O código a seguir fornece o padrão de URL para a página de registro, que deve ser inserido em *accounts/urls.py*:

accounts/urls.py

```
"""Define padrões de URL para contas

from django.urls import path, include

from . import views

app_name = accounts
urlpatterns = [
    # Inclui URLs de autenticação default
    path('', include('django.contrib.auth.urls')),
    # Página de cadastro
    path('register/', views.register, name='register'),
]
```

Importamos o módulo views de accounts, isso é necessário porque estamos escrevendo nossa própria view para a página de cadastro. O padrão para a página de cadastro corresponde ao URL *http://localhost:8000/accounts/register/*e envia requisições para a função register() que estamos prestes a escrever.

Função view register()

A função view register() precisa exibir um formulário de cadastro em branco quando a página de cadastro é solicitada pela primeira vez e, em seguida, processar os formulários concluídos quando são

enviados. Se um usuário se registrar e o cadastro for bem-sucedido, a função também precisa fazer login do usuário novo. Adicione o seguinte código a *accounts/views.py*:

accounts/views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from django.contrib.auth.forms import UserCreationForm

def register(request):
    """Cadastra um usuário novo"""
    if request.method != 'POST':
        # Exibe o formulário em branco de cadastro
1       form = UserCreationForm()
    else:
        # Processa o formulário preenchido
2       form = UserCreationForm(data=request.POST)

3       if form.is_valid():
4           new_user = form.save()
            # Faz o login do usuário e o redireciona para a página inicial
5           login(request, new_user)
6           return redirect('learning_logs:index')

    # Exibe um formulário em branco ou inválido
    context = {'form': form}
    return render(request, 'registration/register.html', context)
```

Importamos as funções `render()` e `redirect()` e, em seguida, importamos a função `login()` para logar o usuário, caso as informações de seu cadastro estiverem corretas. Importamos também o `UserCreationForm` default. Na função `register()`, verificamos se estamos respondendo a uma requisição POST. Caso contrário, criamos uma instância a partir de `UserCreationForm` sem os dados iniciais `1`.

Se estivermos respondendo a requisição POST, criamos uma instância a partir de `UserCreationForm` com base nos dados fornecidos `2`. Verificamos então se os dados são válidos `3` – nesse caso: se o nome de usuário apresenta os caracteres adequados, se as senhas correspondem e se o usuário não está tentando nada malicioso no envio.

Caso os dados sejam válidos, chamamos o método save() do formulário a fim de salvar o nome de usuário e o hash da senha no banco de dados 4. O método save() retorna o objeto de usuário recém-criado, que atribuímos a new_user. A partir do momento em que as informações do usuário são salvas, efetuamos o login chamando a função login() com os objetos request e new_user 5, que criam uma sessão válida para o usuário novo. Por último, redirecionamos o usuário à página inicial 6, em que uma mensagem personalizada no cabeçalho informa que o cadastro foi bem-sucedido.

No final da função, renderizamos a página, que será um formulário em branco ou um formulário enviado que é inválido.

Template do cadastro

Agora, crie um template para a página de cadastro de usuários, que será semelhante à página de login. Não se esqueça de salvá-lo no mesmo diretório que *login.html*:

register.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <form action="{% url 'accounts:register' %}" method='post'>
    {% csrf_token %}
    {{ form.as_div }}

    <button name="submit">Register</button>
  </form>

{% endblock content %}
```

Esse template deve ser parecido com os outros templates baseados em formulários que estamos desenvolvendo. Mais uma vez, utilizamos o método as_div para que o Django exiba todos os campos devidamente no formulário, incluindo quaisquer mensagens de erro se o formulário não for preenchido de forma correta.

Link para a página de cadastro

Vamos adicionar o código para mostrar o link da página de cadastro a qualquer usuário que não esteja logado no momento:

base.html

```
-- trecho de código omitido --
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
  {% else %}
    <a href="{% url 'accounts:register' %}">Register</a> -
    <a href="{% url 'accounts:login' %}">Log in</a>
  {% endif %}
-- trecho de código omitido --
```

Agora, os usuários logados veem uma mensagem personalizada e um botão de logout. Os usuários que não estão logados veem um link de cadastro e um link de login. Teste a página de cadastro criando múltiplas contas de usuário com nomes de usuário diferentes.

Na próxima seção, restringiremos algumas das páginas para que fiquem disponíveis apenas aos usuários registrados e garantiremos que cada tópico pertença a um usuário específico.

NOTA *O sistema de cadastro que configuramos possibilita que qualquer pessoa crie diversas contas na aplicação Registro de Aprendizagem. Alguns sistemas exigem que os usuários validem sua identidade enviando um e-mail de confirmação que os usuários devem responder. Com isso, o sistema gera menos contas de spam do que o sistema simples que estamos usando aqui. Contudo, como você está aprendendo a desenvolver aplicações, é perfeitamente adequado praticar com um sistema simples de cadastro de usuários como o que estamos usando.*

FAÇA VOCÊ MESMO

19.2 Contas do blog: Adicione um sistema de autenticação e cadastro de usuários ao projeto do Blog iniciado no Exercício 19.1 (página *509*). Lembre-se de que os usuários logados devem ver seu nome de usuário em algum lugar na tela e que os usuários não

cadastrados devem ver um link para a página de registro.

Permitindo que os usuários sejam proprietários de seus dados

Como os usuários devem ser capazes de fornecer dados privados em seus registros de aprendizagem, desenvolveremos um sistema para identificar quais dados pertencem a qual usuário. Em seguida, restringiremos o acesso a determinadas páginas para que os usuários tenham acesso somente aos próprios dados.

Vamos modificar o modelo `Topic` para que cada tópico pertença a um usuário específico. Isso também se aplicará às entradas, porque cada entrada pertence a um tópico específico. Começaremos restringindo o acesso a páginas específicas.

Restringindo acesso com @login_required

O Django facilita a restrição de acesso a determinadas páginas por meio do decorator `@login_required`. Lembre-se do Capítulo 11: um *decorator* é uma diretiva inserida logo antes de uma definição de função, que modifica o comportamento dessa função. Vejamos um exemplo.

Restringindo o acesso à página de tópicos

Cada tópico será de propriedade de um usuário, assim, apenas usuários cadastrados podem solicitar a página de tópicos. Adicione o seguinte código a *learning_logs/views.py*:

learning_logs/views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required

from .models import Topic, Entry
-- trecho de código omitido --

@login_required
def topics(request):
```

```
    """Mostra todos os tópicos"""
    -- trecho de código omitido --
```

Primeiro, importamos a função login_required(). Usamos login_required() como um decorator para a função view topics(), anexando login_required com o símbolo @. Com isso, o Python sabe como executar o código em login_required() antes do código em topics().

Em login_required(), o código verifica se um usuário está logado e, em topics(), o Django executa o código apenas se um usuário estiver logado. Se não estiver logado, o usuário será redirecionado para a página de login.

Para que esse redirecionamento funcione, é necessário modificar *settings.py*, assim o Django sabe onde encontrar a página de login. Adicione o seguinte código ao final de *settings.py*:

settings.py

```
-- trecho de código omitido --
# Minhas configurações
LOGIN_REDIRECT_URL = 'learning_logs:index'
LOGOUT_REDIRECT_URL = 'learning_logs:index'
LOGIN_URL = 'accounts:login'
```

Agora, quando um usuário não autenticado solicita uma página protegida pelo decorator @login_required, o Django enviará o usuário para o URL definido por LOGIN_URL em *settings.py*.

É possível testar essa configuração fazendo logout de qualquer conta de usuário e indo para a página inicial. Clique no link **Topics**, que deve redirecioná-lo para a página de login. Depois, faça login em qualquer uma de suas contas e, na página inicial, clique mais uma vez no link **Topic**. Você deve ser capaz de acessar a página de tópicos.

Restringindo o acesso em Registro de Aprendizagem

O Django facilita a restrição de acesso às páginas, mas precisamos decidir quais páginas proteger. Primeiro, é melhor pensar em quais páginas devem ficar sem restrições e, depois, aplicar a restrição em

outras páginas do projeto. É possível corrigir com facilidade o acesso com restrição excessiva, já que é menos perigoso do que não restringir páginas confidenciais.

No Registro de Aprendizado, manteremos a página inicial e a página de registro sem restrições. Restringiremos o acesso às demais páginas.

Vejamos o *learning_logs/views.py com* os decorators @login_required aplicados a todas as views, exceto em index():

learning_logs/views.py

```
-- trecho de código omitido --
@login_required
def topics(request):
    -- trecho de código omitido --

@login_required
def topic(request, topic_id):
    -- trecho de código omitido --

@login_required
def new_topic(request):
    -- trecho de código omitido --

@login_required
def new_entry(request, topic_id):
    -- trecho de código omitido --

@login_required
def edit_entry(request, entry_id):
    -- trecho de código omitido --
```

Tente acessar cada uma dessas páginas quando estiver deslogado; você deve ser redirecionado à página de login. Não será possível clicar em links para páginas como new_topic. Mas, caso insira o URL *http://localhost:8000/new_topic/*, você será redirecionado para a página de login. É necessário restringir o acesso a qualquer URL que seja publicamente acessível, relacionado a dados privados do usuário.

Vinculando dados a determinados usuários

Logo depois, precisamos vincular os dados ao usuário que os enviaram. Basta vincular os dados de nível mais alto da hierarquia a um usuário. Assim, os dados de nível mais baixo também serão vinculados. No Registro de Aprendizagem, os tópicos estão no nível mais alto de dados da aplicação e todas as entradas estão associadas a um tópico. Desde que cada tópico pertença a um usuário específico, podemos rastrear a propriedade de cada entrada no banco de dados.

Modificaremos o modelo `Topic` adicionando uma relação de chave estrangeira a um usuário. Será necessário migrar o banco de dados. Por último, modificaremos algumas das views para que mostrem somente os dados associados ao usuário atualmente conectado.

Modificando o modelo Topic

Modificamos somente duas linhas de *models.py*:

models.py

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """Um tópico que o usuário está aprendendo"""
    Text = models.CharField(max_length=200)
    date_added = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(User, on_delete=models.CASCADE)

    def __str__(self):
        """Retorna uma string representando o tópico"""
        Return self.text

class Entry(models.Model):
    -- trecho de código omitido --
```

Importamos o modelo `user` de `django.contrib.auth`. Depois, adicionamos um campo `owner` a `Topic`, que estabelece uma relação de chave estrangeira com o modelo `User`. Se um usuário for excluído, todos os tópicos associados a ele também serão excluídos.

Identificando usuários existentes

Quando migrarmos o banco de dados, o Django modificará o banco de dados a fim de que possa armazenar uma conexão entre cada tópico e um usuário. Para realizar a migração, o Django precisa saber qual usuário associar a cada tópico existente. A abordagem mais simples é começar atribuindo todos os tópicos existentes a um usuário – por exemplo, ao superusuário. Mas, antes, precisamos saber o ID do usuário.

Vejamos os IDs de todos os usuários criados até agora. Inicie uma sessão shell do Django e execute os seguintes comandos:

```
  (ll_env)learning_log$ python manage.py shell
1 >>> from django.contrib.auth.models import User
2 >>> User.objects.all()
  <QuerySet [<User: ll_admin>, <User: eric>, <User: willie>]>
3 >>> for user in User.objects.all():
  ...     print(user.username, user.id)
  ...
  ll_admin 1
  eric 2
  willie 3
  >>>
```

Primeiro, importamos o modelo `User` para a sessão do shell 1. Em seguida, examinamos todos os usuários criados até agora 2. A saída mostra três usuários para a minha versão do projeto: `ll_admin`, `eric` e `willie`.

Depois, percorremos a lista de usuários com um loop e exibimos o nome de usuário e ID 3 de cada usuário. Quando o Django perguntar a qual usuário associar os tópicos existentes, utilizaremos um desses valores de ID.

Migrando o banco de dados

Agora que sabemos os IDs, podemos migrar o banco de dados. Quando fizermos isso, o Python solicitará que associemos o modelo `Topic` a um proprietário específico temporariamente ou que adicionemos um modelo default ao nosso arquivo *models.py* para

que esse arquivo saiba o que fazer. Escolha a opção **1**:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
It is impossible to add a non-nullable field 'owner' to topic without
 specifying a default. This is because...
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a
    null value for this column)
 2) Quit and manually define a default value in models.py.
Select an option: 1
Please enter the default value now, as valid Python
 The datetime and django.utils.timezone modules are available...
 Type 'exit' to exit this prompt
>>> 1
 Migrations for 'learning_logs':
  learning_logs/migrations/0003_topic_owner.py
 - Add field owner to topic
 (ll_env)learning_log$
```

Começamos executando o comando makemigrations 1. Na saída, o Django indica que estamos tentando adicionar um campo obrigatório (*non-nullable*) a um modelo existente (topic) sem nenhum valor default especificado 2. O Django nos oferece duas opções: podemos fornecer um default agora ou podemos sair e adicionar um valor default em *models.py* 3. Aqui, escolhi a primeira opção 4. O Django então solicita a inserção do valor default 5.

Para associar todos os tópicos existentes com o usuário admin original, ll_admin, inseri o ID de usuário de 1 6. É possível usar o ID de qualquer usuário que tenhamos criado; não precisa ser um superusuário. Em seguida, o Django migra o banco de dados usando esse valor e gera o arquivo de migração *0003_topic_owner.py*, que adiciona o campo owner ao modelo Topic.

Agora, podemos migrar o banco. Digite o seguinte em um ambiente virtual ativo:

```
(ll_env)learning_log$ python manage.py migrate
Operations to perform:
 Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
1  Applying learning_logs.0003_topic_owner... OK
 (ll_env)learning_log$
```

O Django aplica a migração nova, e o resultado é OK 1.

Podemos constatar que a migração funcionou como esperado em uma sessão do shell, assim:

```
>>> from learning_logs.models import Topic
>>> for topic in Topic.objects.all():
...     print(topic, topic.owner)
...
Chess ll_admin
Rock Climbing ll_admin
>>>
```

Importamos Topic de learning_logs.models e, depois, percorremos todos os tópicos existentes com um loop, exibindo cada tópico e o usuário ao qual pertence. Agora, é possível ver que cada tópico pertence ao usuário ll_admin. (Caso receba um erro ao executar esse código, tente sair do shell e iniciar uma janela nova do shell.)

NOTA *Podemos simplesmente reinicializar o banco de dados em vez de migrá-lo, mas todos os dados existentes seriam perdidos. É boa prática aprender a migrar um banco de dados enquanto se assegura a integridade dos dados dos usuários. Caso queira começar com um banco de dados novo, execute o comando* python manage.py flush *para recriar a estrutura do banco de dados. Será necessário criar um superusuário novo e todos os seus dados desaparecerão.*

Restringindo o acesso de tópicos a usuários adequados

Atualmente, se estiver conectado, você poderá ver todos os tópicos, independentemente do usuário com o qual estiver logado. Vamos alterar isso mostrando aos usuários apenas os tópicos que pertencem a eles.

Faça a seguinte alteração na função topics() em *views.py*:

learning_logs/views.py

```
-- trecho de código omitido --
@login_required
```

```
def topics(request):
    """Mostra todos os tópicos"""
    topics = Topic.objects.filter(owner=request.user).order_by('date_added')
    context = {'topics': topics}
    return render(request, 'learning_logs/topics.html', context)
-- trecho de código omitido --
```

Quando um usuário está logado, o objeto request tem um conjunto de atributos request.user, que contém informações sobre o usuário. A query Topic.objects.filter(owner= request.user) instrui o Django a recuperar somente os objetos Topic do banco de dados cujo atributo owner corresponde ao usuário atual. Como não estamos alterando a forma como os tópicos são exibidos, não precisamos alterar o template da página de tópicos.

Para ver se isso funciona, faça login como o usuário ao qual vinculou todos os tópicos existentes e acesse a página de tópicos. Você deve ver todos os tópicos. Agora saia e faça login mais uma vez como um usuário diferente. Você deve ver a mensagem "No topics have been added yet."

Protegendo os tópicos de um usuário

Como ainda não restringimos o acesso às páginas de tópicos, qualquer usuário cadastrado pode tentar acessar diversos URLs (como *http://localhost:8000/topics/1/*) e as páginas de tópicos que coincidam.

Faça o teste. Enquanto estiver logado com o usuário que tem acesso a todos os tópicos, copie o URL ou anote o ID no URL de um tópico e, em seguida, saia e faça login novamente com um usuário diferente. Digite o URL do tópico. Você deve ser capaz de ler as entradas, mesmo que esteja logado com um usuário diferente.

Vamos corrigir isso executando uma verificação antes de acessar as entradas solicitadas na função view topic():

learning_logs/views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
```

```
1 from django.http import Http404

  -- trecho de código omitido --
  @login_required
  def topic(request, topic_id):
      """Mostra um único tópico e todas as suas entradas"""
      topic = Topic.objects.get(id=topic_id)
      # Verifica se o tópico pertence ao usuário atual
2     if topic.owner != request.user:
          raise Http404

      entries = topic.entry_set.order_by('-date_added')
      context = {'topic': topic, 'entries': entries}
      return render(request, 'learning_logs/topic.html', context)
  -- trecho de código omitido --
```

A resposta 404 é uma resposta de erro padrão, retornada quando um recurso solicitado não existe em um servidor. Aqui, importamos a exceção `Http404` `1`, que será lançada se o usuário solicitar um tópico ao qual não deve ter acesso. Após receber a requisição de tópico, garantimos que o usuário do tópico corresponda ao usuário atualmente logado antes de renderizar a página. Se o proprietário do tópico requisitado não for o mesmo que o usuário atual, lançamos a exceção `2` `Http404` e o Django retorna uma página 404-error.

Agora, se tentarmos visualizar as entradas de tópico de outro usuário, vermos a mensagem "Page Not Found" do Django. No Capítulo 20, vamos configurar o projeto para que os usuários vejam uma página de erro adequada em vez de uma página de depuração.

Protegendo a página edit_entry

As páginas `edit_entry` têm URLs do formulário *http://localhost:8000/edit_entry/entry_id/*, em que o *entry_id* é um número. Protegeremos essa página para que ninguém consiga usar o URL a fim de obter acesso às entradas de outra pessoa:

learning_logs/views.py

```
  -- trecho de código omitido --
  @login_required
  def edit_entry(request, entry_id):
```

```
    """Edita uma entrada existente"""
    entry = Entry.objects.get(id=entry_id)
    topic = entry.topic
    if topic.owner != request.user:
        raise Http404

    if request.method != 'POST':
        -- trecho de código omitido --
```

Acessamos a entrada e o tópico associado a essa entrada. Em seguida, verificamos se o proprietário do tópico corresponde ao usuário atualmente logado; se não corresponderem, criamos uma exceção Http404.

Associando tópicos novos ao usuário atual

Até agora, a página para adicionar tópicos novos está fora do ar porque não associa tópicos novos a nenhum usuário específico. Caso tente adicionar um tópico novo, verá a mensagem IntegrityError com a restrição NOT NULL constraint failed: learning_logs_topic.owner_id. O Django está informando que não podemos criar um tópico novo sem especificar um valor para o campo owner do tópico.

É possível corrigir esse problema facilmente, porque temos acesso ao usuário atual por meio do objeto request. Adicione o seguinte código, que associa o tópico novo ao usuário atual:

learning_logs/views.py

```
-- trecho de código omitido --
@login_required
def new_topic(request):
    -- trecho de código omitido --
    else:
        # Dados POST enviados; processa os dados
        form = TopicForm(data=request.POST)
        if form.is_valid():
1           new_topic = form.save(commit=False)
2           new_topic.owner = request.user
3           new_topic.save()
            return redirect('learning_logs:topics')

    # Exibe um formulário em branco ou inválido
```

```
    context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
-- trecho de código omitido --
```

Ao chamarmos `form.save()` pela primeira vez, passamos o argumento `commit=False`, já que precisamos modificar o tópico novo antes de salvá-lo no banco de dados 1. Depois, definimos o atributo `owner` do tópico novo para o usuário atual 2. Por último, chamamos `save()` na instância do tópico que acabamos de definir 3. Agora, o tópico tem todos os dados necessários e será salvo com sucesso.

Devemos conseguir adicionar quantos tópicos novos quisermos para quantos usuários diferentes desejarmos. Cada usuário terá acesso aos próprios dados, quer esteja visualizando dados, inserindo dados novos ou modificando dados antigos.

FAÇA VOCÊ MESMO

19.3 Refatoração: Existem dois locais em *views.py* em que asseguramos que o usuário associado a um tópico corresponde ao usuário atualmente logado. Insira o código para essa verificação em uma função chamada check_topic_owner(), e chame essa função nos locais adequados.

19.4 Protegendo new_entry: Atualmente, um usuário pode adicionar uma entrada nova ao registro de aprendizagem de outro usuário inserindo um URL com o ID de um tópico pertencente a outro usuário. Evite esse ataque verificando se o usuário atual tem o tópico da entrada antes de salvar a entrada nova.

19.5 Blog protegido: Em seu projeto Blog, assegure que cada postagem do blog esteja logada a um usuário específico. Garanta que todas as postagens sejam acessíveis ao público e que apenas os usuários cadastrados possam adicionar postagens e editar postagens existentes. Na view que possibilita aos usuários editar suas postagens, verifique se o usuário está editando a própria postagem antes de processar o formulário.

Recapitulando

Neste capítulo, aprendemos como os formulários possibilitam que os usuários adicionem entradas e tópicos novos e editem entradas existentes. Em seguida, aprendemos como implementar contas de usuário. Viabilizamos aos usuários existentes a capacidade de fazer login e logout, e recorremos ao `UserCreationForm` default do Django para permitir que pessoas criassem contas novas.

Após desenvolvermos um sistema simples de autenticação e registro de usuários, restringimos o acesso a usuários conectados a determinadas páginas com o decorator `@login_required`. Em seguida, atribuímos dados a usuários específicos por meio de um relacionamento de chave estrangeira. Além disso, aprendemos a migrar o banco de dados quando a migração exige que especifiquemos alguns dados padrão.

Por último, aprendemos como garantir que um usuário só possa ver os dados que lhe pertencem modificando as funções view. Acessamos os dados adequados com o método `filter()` e comparamos o proprietário dos dados solicitados ao usuário atualmente logado.

Nem sempre fica claro quais dados devemos disponibilizar e quais dados devemos proteger, pois essa habilidade é adquirida com muita prática. Neste capítulo, as decisões que tomamos para proteger os dados de nossos usuários também exemplificam por que trabalhar com outras pessoas é uma boa ideia ao criar um projeto: quando outra pessoa analisa seu projeto, a probabilidade de você identificar áreas vulneráveis é maior.

Agora, temos um projeto que funciona perfeitamente quando executado em nossa máquina local. No capítulo final, vamos estilizar o Registro de Aprendizagem para que fique visualmente elegante e vamos fazer o deploy do projeto em um servidor. Assim, qualquer pessoa com acesso à internet pode se cadastrar no site e criar uma conta.

CAPÍTULO 20

Estilizando e fazendo o deploy de uma aplicação

Agora, o Projeto Registro de Aprendizagem está totalmente funcionando, embora não esteja estilizado e seja apenas executado em sua máquina local. Neste capítulo, vamos estilizar o projeto de maneira simples, ainda que profissional, e vamos fazer o deploy em um servidor ativo para que qualquer pessoa no mundo consiga criar uma conta e usá-lo.

Para estilizar nosso projeto, recorreremos à biblioteca *Bootstrap*, uma coleção de ferramentas para estilizar aplicações web a fim de que pareçam profissionais em todos os dispositivos modernos, de um simples smartphone até um grande monitor de desktop. Para isso, utilizaremos a aplicação django-bootstrap5, assim também adquirimos a prática de usar aplicações desenvolvidas por outros desenvolvedores Django.

Faremos o deploy do Registro de Aprendizagem usando o *Platform.sh*, site que possibilita carregar um projeto para um de seus servidores, disponibilizando-o para qualquer pessoa com conexão à internet. Começaremos também a usar um sistema de controle de versão chamado Git para rastrear as mudanças no projeto.

Quando concluir o Registro de Aprendizagem, você será capaz de desenvolver aplicações web simples, com aparência profissional, e fazer o deploy delas para um servidor ativo. Além do mais, será capaz de usar recursos de aprendizagem mais avançados à medida que suas habilidades progridem.

Estilizando o Registro de Aprendizagem

Até agora, ignoramos intencionalmente a estilização com o intuito de focarmos primeiro as funcionalidades do Registro de Aprendizagem. Trata-se de uma boa abordagem de desenvolvimento, já que uma aplicação só tem serventia se funcionar. Uma vez que a aplicação esteja funcionando, é fundamental estilizarmos sua aparência para que as pessoas queiram usá-la.

Nesta seção, instalaremos a aplicação django-bootstrap5 e vamos adicioná-la ao projeto. Em seguida, vamos usá-la para estilizar as páginas individuais do projeto, assim todas terão uma aparência consistente.

Aplicação django-bootstrap5

Recorreremos à aplicação django-bootstrap5 para integrar a Bootstrap ao nosso projeto. Essa aplicação faz o download dos arquivos necessários da biblioteca Bootstrap, os insere em um local adequado em seu projeto e disponibiliza as diretivas de estilo nos templates do projeto.

Para instalar o django-bootstrap5, digite o seguinte comando em um ambiente virtual ativo:

```
(ll_env)learning_log$ pip install django-bootstrap5
-- trecho de código omitido --
Successfully installed beautifulsoup4-4.11.1 django-bootstrap5-21.3
    soupsieve-2.3.2.post1
```

Em seguida, precisamos adicionar django-bootstrap5 a INSTALLED_APPS em *settings.py*:

settings.py

```
-- trecho de código omitido --
INSTALLED_APPS = [
    # Minhas aplicações
    'learning_logs',
    'accounts',

    # Aplicações de terceiros
```

```
    'django_bootstrap5',

    # Aplicações default do Django
    'django.contrib.admin',
    -- trecho de código omitido --
```

Inicie uma seção nova chamada Aplicações de terceiros, para aplicações desenvolvidas por outros desenvolvedores, e adicione 'django_bootstrap5' a essa seção. Não se esqueça de inserir essa seção após Minhas aplicações, antes da seção Aplicações default do Django.

Bootstrap para estilizar Registro de Aprendizagem

A Bootstrap é uma coleção enorme de ferramentas de estilo. Disponibiliza também uma série de templates que podemos usar em nossos projetos para criar um estilo geral. É mais fácil usar esses templates do que usar ferramentas individuais de estilização. Para conferir os modelos que a Bootstrap oferece, acesse https://getbootstrap.com e clique em **Examples**. Vamos utilizar o template *Navbar static*, que fornece uma barra de navegação superior simples e um contêiner para o conteúdo da página.

A Figura 20.1 mostra como será a página inicial após aplicarmos o template da Bootstrap à *base.html* e modificarmos um bocado o *index.html*.

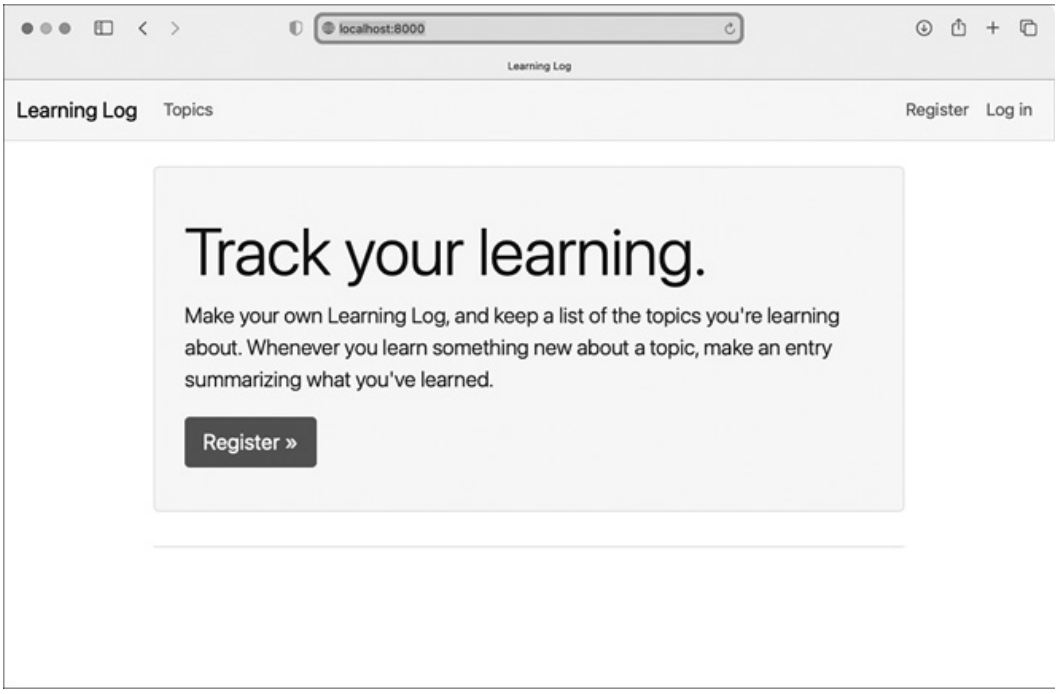


Figura 20.1: A página inicial do Registro de Aprendizagem usando a Bootstrap. No código, como Learning Log.

Modificando base.html

É necessário reescrever *base.html* usando o template da Bootstrap. Incrementaremos o novo *base.html* em seções. Como se trata de um arquivo grande, talvez você queira copiá-lo a partir dos recursos online, disponíveis em https://ehmatthes.github.io/pcc_3e. Se copiar o arquivo, é essencial ler a seção a seguir para entender as mudanças feitas.

Definindo os cabeçalhos HTML

A primeira mudança que faremos em *base.html* define os cabeçalhos HTML no arquivo. Além disso, adicionaremos alguns requisitos para usar a Bootstrap em nossos templates e forneceremos um título à página. Exclua todo o conteúdo em *base.html* e o substitua pelo seguinte código:

base.html

```
1 <!doctype html>
2 <html lang="en">
3 <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
4   <title>Learning Log</title>

5   {% load django_bootstrap5 %}
    {% bootstrap_css %}
    {% bootstrap_javascript %}

  </head>
```

Primeiro, declaramos esse arquivo como um documento HTML 1 escrito em inglês 2. Um arquivo HTML é dividido em duas partes principais: o *cabeçalho* e o *corpo*. O cabeçalho do arquivo começa com a tag de abertura <head> 3. O cabeçalho de um arquivo HTML não armazena nenhum conteúdo da página; apenas informa ao navegador o necessário para que exiba corretamente a página. Incluímos um elemento <title> para a página, que será exibido na barra de título do navegador sempre que o Registro de Aprendizagem estiver aberto 4.

Antes de fecharmos a seção do cabeçalho, carregamos a coleção de template tags disponíveis no django-bootstrap5 5. A template tag {% bootstrap_css %} é uma tag personalizada do django-bootstrap5; carrega todos os arquivos CSS necessários para implementar os estilos Bootstrap. A tag a seguir habilita todo o comportamento interativo que podemos usar em uma página, como barras de navegação recolhíveis. A tag de fechamento </head> aparece na última linha.

Agora, todas as opções de estilo da Bootstrap estão disponíveis em qualquer template herdado de *base.html*. Caso queira usar as template tags personalizadas do django-bootstrap5, cada template precisará incluir a tag {% load django_bootstrap5 %}.

Definindo a barra de navegação

O código que define a barra de navegação na parte superior da

página é bastante extenso, visto que precisa funcionar igualmente bem em telas estreitas de celulares e em monitores grandes de desktop. Trabalharemos na barra de navegação em seções.

Vejamos a primeira parte da barra de navegação:

base.html

```
  -- trecho de código omitido --
  </head>
  <body>

1   <nav class="navbar navbar-expand-md navbar-light bg-light mb-4 border">
      <div class="container-fluid">
2       <a class="navbar-brand" href="{% url 'learning_logs:index' %}">
          Learning Log</a>

3       <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
         data-bs-target="#navbarCollapse" aria-controls="navbarCollapse"
         aria-expanded="false" aria-label="Toggle navigation">
         <span class="navbar-toggler-icon"></span>
        </button>

4       <div class="collapse navbar-collapse" id="navbarCollapse">
5         <ul class="navbar-nav me-auto mb-2 mb-md-0">
6           <li class="nav-item">
7             <a class="nav-link" href="{% url 'learning_logs:topics' %}">
                Topics</a></li>
          </ul> <!-- Fim dos links no lado esquerdo da barra de navegação -->
        </div> <!-- Fecha as partes recolhíveis da barra de navegação -->

      </div> <!-- Fecha o contêiner da barra de navegação -->
    </nav> <!-- Fim da barra de navegação -->

8 {% block content %}{% endblock content %}

  </body>
  </html>
```

O primeiro elemento novo é a tag de abertura `<body>`. O *corpo* de um arquivo HTML tem o conteúdo que os usuários verão em uma página. Em seguida, temos um elemento `<nav>`, que abre o código para a barra de navegação na parte superior da página 1. Tudo dentro desse elemento é estilizado segundo as regras de estilo da

Bootstrap, definidas pelos seletores navbar, navbar-expand-md e o restante que podemos ver aqui. Um *seletor* estabelece a quais elementos de uma página uma determinada regra de estilo se aplica. Os seletores navbar-light e bg-light estilizam a barra de navegação com background de tema claro. O mb em mb-4 é abreviação de *margin-bottom*; esse seletor garante que um pouco de espaço apareça entre a barra de navegação e o resto da página. O seletor border fornece uma borda fina ao redor do background claro para diferenciá-lo um pouco do restante da página.

A tag na próxima linha abre um contêiner redimensionável que armazenará a barra de navegação geral. O termo *div* é abreviação de *division*; desenvolvemos uma página web dividindo-a em seções e definindo regras de estilo e comportamento que se aplicam a essa seção. Quaisquer regras de estilo ou comportamento definidas em uma tag de abertura afetam tudo o que vemos até sua tag de fechamento correspondente, escrita como .

Depois, definimos o nome do projeto, Learning Log, para aparecer como primeiro elemento na barra de navegação 2. Isso também servirá como link para a página inicial, assim como fizemos na versão minimamente estilizada do projeto que desenvolvemos nos dois capítulos anteriores. O seletor navbar-brand estiliza esse link para que se destaque do resto dos links e ajuda a adicionar uma marcar ao site.

O template da Bootstrap define um botão que aparece se a janela do navegador for muito estreita, a fim de exibir horizontalmente a barra de navegação por completo 3. Quando o usuário clicar no botão, os elementos de navegação aparecem em uma lista suspensa. A referência collapse faz com que a barra de navegação seja recolhida quando o usuário reduz a janela do navegador ou quando o site é exibido em dispositivos com telas pequenas.

Em seguida, abrimos uma seção nova ( ) da barra de navegação 4. Essa é a parte da barra de navegação que pode ser

recolhida dependendo do tamanho da janela do navegador.

A Bootstrap define elementos de navegação como itens em uma lista não ordenada 5, com regras de estilo que fazem com que essa lista não se pareça nada com uma. Cada link ou elemento necessário na barra pode ser incluído como um item em uma lista não ordenada 6. Aqui, o único item na lista é o nosso link para a página de tópicos 7. Repare na tag de fechamento no final do link; cada tag de abertura precisa de uma tag de fechamento correspondente.

O restante das linhas mostradas aqui encerra todas as tags abertas. Em HTML, escrevemos um comentário assim:

```
<!-- Um comentário em HTML. -->
```

Via de regra, as tags de fechamento não têm comentários, mas caso seja novato em HTML, rotular algumas de suas tags de fechamento pode ajudar bastante. Uma única tag ausente ou uma tag extra pode prejudicar o layout de uma página inteira. Incluímos o bloco content 8 e também as tags de fechamento </body> e as tags </html>.

Ainda não concluímos a barra de navegação, mas agora, temos um documento HTML completo. Se runserver estiver ativo no momento, desative o servidor atual e o reinicie. Acesse a página inicial do projeto e você verá uma barra de navegação com alguns dos elementos mostrados na Figura 20.1. Agora, adicionaremos o restante dos elementos à barra de navegação.

Adicionando links de conta de usuário

Ainda é necessário adicionar os links associados às contas de usuário. Começaremos adicionando todos os links relacionados à conta, exceto o formulário de logout.

Faça as seguintes mudanças em *base.html*:

base.html

```
-- trecho de código omitido --
</ul> <!-- Fim dos links no lado esquerdo da barra de navegação -->

<!-- Links relacionados à conta -->
```

```
1       <ul class="navbar-nav ms-auto mb-2 mb-md-0">

2         {% if user.is_authenticated %}
           <li class="nav-item">
3             <span class="navbar-text me-2">Hello, {{ user.username }}.
               </span></li>
4         {% else %}
           <li class="nav-item">
             <a class="nav-link" href="{% url 'accounts:register' %}">
                Register</a></li>
           <li class="nav-item">
             <a class="nav-link" href="{% url 'accounts:login' %}">
                Log in</a></li>
         {% endif %}

        </ul> <!-- Fim dos links relacionados à conta -->

      </div> <!-- Fecha as partes recolhíveis da barra de navegação -->
      -- trecho de código omitido --
```

Começamos um conjunto novo de links usando outra tag de abertura `<ul>` 1. É possível termos quantos grupos de links precisarmos em uma página. O seletor `ms-auto` é abreviação de *margin-start-automatic*: esse seletor examina os outros elementos na barra de navegação e define uma margem esquerda (inicial) que posiciona esse grupo de links ao lado direito da janela do navegador.

O bloco `if` é o mesmo bloco condicional que usamos antes para exibir mensagens adequada aos usuários, se eles estiverem logados 2. Agora, esse bloco é um pouco maior, pois existem algumas regras de estilo dentro das tags condicionais. A mensagem para usuários autenticados é envolvida em um elemento `<span>` 3. Um *elemento span* estiliza pedaços de texto ou elementos de uma página que fazem parte de uma linha mais extensa. Enquanto os elementos div criam as divisões em uma página, os elementos span são contínuos dentro de uma seção maior. À primeira vista, pode parecer confuso, já que muitas páginas têm elementos div profundamente aninhados. Aqui, estamos usando o elemento span para estilizar o texto informativo na barra de navegação: nesse caso, o nome do usuário logado.

No bloco else, executado para usuários não autenticados, incluímos os links para registrar uma conta nova e fazer login 4. Esses devem se parecer com o link para a página de tópicos.

Caso queira adicionar mais links à barra de navegação, adicione outro item <li> a um dos grupos <ul> que definimos, usando diretivas de estilo como as que você viu aqui.

Agora, adicionaremos o formulário de logout à barra de navegação.

Adicionando o formulário de logout à barra de navegação

Quando criamos o formulário de logout pela primeira vez, o adicionamos à parte inferior de *base.html*. Agora, vamos inseri-lo em um lugar melhor, na barra de navegação:

base.html

```
    -- trecho de código omitido --
    </ul> <!-- Fim dos links relacionados à conta -->

    {% if user.is_authenticated %}
      <form action="{% url 'accounts:logout' %}" method='post'>
        {% csrf_token %}
1         <button name='submit' class='btn btn-outline-secondary btn-sm'>
            Log out</button>
      </form>
    {% endif %}

  </div> <!-- Fecha as partes recolhíveis da barra de navegação -->
  -- trecho de código omitido --
```

Devemos inserir o formulário de logout após o conjunto de links relacionados à conta, mas dentro da seção recolhível da barra de navegação. No formulário, a única mudança é a adição de diversas classes de estilização da Bootstrap no elemento <button>, que aplicam elementos de estilo da Bootstrap ao botão de logout 1.

Recarregue a página inicial, e será possível fazer login e logout usando qualquer uma das contas criadas.

Ainda precisamos adicionar mais um pouco de coisas ao *base.html*.

É necessário definir dois blocos que as páginas individuais podem utilizar a fim de posicionar o conteúdo específico dessas páginas.

Definindo a parte principal da página

O restante de *base.html* contém a parte principal da página:

base.html

```
    -- trecho de código omitido --
    </nav> <!-- Fim da barra de navegação -->

1   <main class="container">
2     <div class="pb-2 mb-2 border-bottom">
        {% block page_header %}{% endblock page_header %}
      </div>
3     <div>
        {% block content %}{% endblock content %}
      </div>
    </main>

  </body>
  </html>
```

Primeiro, abrimos uma tag `<main>` 1. Usamos o elemento *main* para a parte mais significativa do corpo de uma página. Nesse caso, atribuímos o seletor bootstrap `container`, uma forma simples de agrupar elementos em uma página. Vamos inserir dois elementos div nesse contêiner.

O primeiro elemento div contém um bloco `page_header` 2. Vamos utilizar esse bloco para fornecer um título a maioria das páginas. Para destacar essa seção do restante da página, colocamos um padding abaixo do cabeçalho. *Padding* se refere ao espaço entre o conteúdo de um elemento e sua borda. O seletor `pb-2` é uma diretiva da bootstrap que fornece uma quantidade moderada de padding na parte inferior do elemento estilizado. Uma *margem* é o espaço entre a borda de um elemento e outros elementos na página. O seletor `mb-2` fornece uma quantidade moderada de margem na parte inferior desse div. Como queremos uma borda na parte inferior desse bloco, usamos o seletor `border-bottom`, que fornece uma borda fina na parte

inferior do bloco page_header.

Em seguida, definimos mais um elemento div que contém o bloco content 3. Não aplicamos nenhuma estilização específica a esse bloco. Desse modo, podemos estilizar o conteúdo de qualquer página conforme julgarmos adequado. O final do arquivo *base.html* tem tags de fechamento para os elementos main, body e html.

Ao carregar a página inicial do Registro de Aprendizagem em um navegador, devemos ver uma barra de navegação com aparência profissional que corresponda àquela mostrada na Figura 20.1. Tente redimensionar a janela para que fique bastante estreita; um botão deve substituir a barra de navegação. Clique no botão e todos os links devem aparecer em uma lista suspensa.

Estilizando a página inicial com um Jumbotron

A fim de atualizar a página inicial, usaremos um elemento da Bootstrap chamado *jumbotron*, uma caixa grande que se destaca do restante da página. Esse elemento normalmente é utilizado em páginas iniciais para armazenar uma breve descrição do projeto geral e uma chamada à ação que convida o usuário a clicar em alguma coisa.

Vejamos o arquivo *index.html* revisado:

index.html

```
{% extends "learning_logs/base.html" %}

{% block page_header %}
  <div class="p-3 mb-4 bg-light border rounded-3">
    <div class="container-fluid py-4">
      <h1 class="display-3">Track your learning.</h1>

      <p class="lead">Make your own Learning Log, and keep a list of the
      topics you're learning about. Whenever you learn something new
      about a topic, make an entry summarizing what you've learned.</p>

      <a class="btn btn-primary btn-lg mt-1"
       href="{% url 'accounts:register' %}">Register &raquo;</a>
```

```
    </div>
  </div>
{% endblock page_header %}
```

Inicialmente, informamos ao Django que estamos prestes a definir o que será inserido no bloco page_header 1. Um jumbotron é implementado como um par de elementos div com um conjunto de diretivas de estilo aplicadas 2. O div externo armazena configurações de padding e margem, uma cor clara de background e bordas arredondadas. O div interno é um contêiner que muda junto com o tamanho da janela e tem também um pouco de padding. O seletor py-4 adiciona o padding à parte superior e inferior do elemento div. Sinta-se à vontade para ajustar os números dessas configurações e ver como a página inicial fica.

Temos três elementos dentro do jumbotron. O primeiro é uma mensagem curta, Track your learning, que fornece aos novos visitantes uma noção de como o Registro de Aprendizagem funciona 3. O elemento <h1> é um cabeçalho de primeiro nível e o seletor display-3 faz com que esse cabeçalho específico tenha uma aparência mais tênue e fique maior. Incluímos também uma mensagem mais extensa com mais informações sobre o que o usuário pode fazer com seu registro de aprendizagem 4. Tudo é formatado como um parágrafo lead, que deve se destacar dos parágrafos regulares.

Em vez de apenas usar um link de texto, criamos um botão que convida os usuários a registrar uma conta no Registro de Aprendizagem 5. É o mesmo link do cabeçalho, mas o botão se destaca na página e mostra ao usuário o que ele precisa fazer para começar a usar o projeto. Os seletores que vemos aqui estilizam isso como um botão grande que representa uma chamada à ação. O código » é uma *entidade* HTML que se parece com dois colchetes angulares combinados (>>). Por último, fornecemos tags div de fechamento e encerramos o bloco page_header. Como esse arquivo tem apenas dois elementos div, não é muito útil rotular as tags div de fechamento. Como não estamos adicionando mais nada

a essa página, não precisamos definir o bloco content nesse template.

Agora, a página inicial se parece com a da Figura 20.1. É uma melhoria acentuada em relação à versão não estilizada do projeto!

Estilizando a página de login

Refinamos a aparência geral da página de login, mas o formulário de login ainda não tem nenhuma estilização. Vamos modificar *login.html* a fim de melhorar o formulário para que seja consistente com o restante da página:

login.html

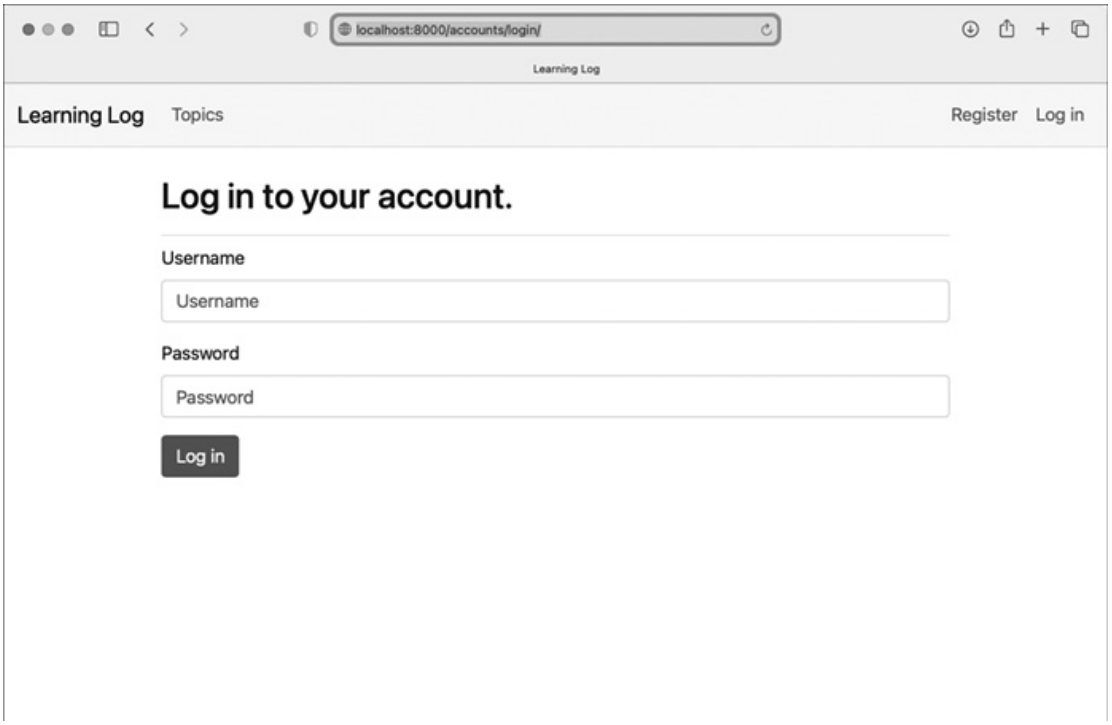
```
  {% extends 'learning_logs/base.html' %}
1 {% load django_bootstrap5 %}

2 {% block page_header %}
    <h2>Log in to your account.</h2>
  {% endblock page_header %}

  {% block content %}

    <form action="{% url 'accounts:login' %}" method='post'>
      {% csrf_token %}
3     {% bootstrap_form form %}
4     {% bootstrap_button button_type="submit" content="Log in" %}
    </form>

  {% endblock content %}
```

Primeiro, carregamos as template tags do bootstrap5 nesse template 1. Em seguida, definimos o bloco page_header, que informa ao usuário o propósito da página 2. Repare que removemos o bloco {% if form.errors %} do template; o django-bootstrap5 gerencia automaticamente os erros de formulário.

Para exibir o formulário, usamos a template tag {% bootstrap_form %} 3; que substitui o elemento {{ form.as_div }} que estávamos usando no Capítulo 19. A template tag {% booststrap_form %} insere regras de estilo da Bootstrap nos elementos individuais do formulário à medida que o formulário é renderizado. Para gerar o botão de enviar, usamos a

tag {% bootstrap_button %} com argumentos que a definem como um botão de enviar e lhe atribuímos o rótulo Log in4.

A Figura 20.2 mostra o formulário de login. Agora, a página está mais limpa, com estilização consistente e um propósito claro. Tente efetuar login com um nome de usuário ou senha incorretos; você verá que até mesmo as mensagens de erro são estilizadas de forma consistente e se integram perfeitamente ao site como um todo.



Figur 20.2: A página de login estilizada com Bootstrap.

Estilizando a página de tópicos

Agora, garantiremos que as páginas para visualização de informações também estejam devidamente estilizadas, começando com a página de tópicos:

topics.html

```
{% extends 'learning_logs/base.html' %}

{% block page_header %}
  <h1>Topics</h1>
{% endblock page_header %}
```

```
{% block content %}

<ul class="list-group border-bottom pb-2 mb-4">
  {% for topic in topics %}
    <li class="list-group-item border-0">
      <a href="{% url 'learning_logs:topic' topic.id %}">
        {{ topic.text }}</a>
    </li>
  {% empty %}
    <li class="list-group-item border-0">No topics have been added yet.</li>
  {% endfor %}
</ul>

<a href="{% url 'learning_logs:new_topic' %}">Add a new topic</a>

{% endblock content %}
```

Não precisamos da tag `{% load bootstrap5 %}`, já que não estamos usando nenhuma template tag personalizada da bootstrap5 nesse arquivo. Transferimos o cabeçalho `Topics` para o bloco `page_header` e o tornamos um elemento `<h1>` em vez de um parágrafo simples 1.

Como o conteúdo principal dessa página é uma lista de tópicos, utilizamos o componente *grupo de lista* da Bootstrap para renderizá-la. Isso aplica um conjunto simples de diretivas de estilização à lista geral e a cada item da lista. Ao abrirmos a tag `<ul>`, primeiro incluímos a classe `list-group` a fim de aplicar as diretivas de estilização default à lista 2. Personalizamos ainda mais a lista inserindo uma borda na parte inferior da lista, um pouco de padding abaixo da lista (`pb-2`) e uma margem abaixo da borda inferior (`mb-4`).

Na lista, cada item precisa da classe `list-group-item`, e personalizamos a estilização default removendo a borda ao redor dos itens individuais 3. A mensagem exibida quando a lista está vazia precisa dessas mesmas classes 4.

Agora, quando acessamos a página de tópicos, devemos ver uma página com estilo correspondente à página inicial.

Estilizando as entradas na página de tópico

Na página do tópico, usaremos o componente card da Bootstrap a fim de destacar cada entrada. Um *card* é um conjunto aninhável de divs com estilos flexíveis e predefinidos, perfeitos para exibir as entradas de um tópico:

topic.html

```
{% extends 'learning_logs/base.html' %}

{% block page_header %}
  <h1>{{ topic.text }}</h1>
{% endblock page_header %}

{% block content %}
  <p>
    <a href="{% url 'learning_logs:new_entry' topic.id %}">Add new entry</a>
  </p>

  {% for entry in entries %}
    <div class="card mb-3">
      <!-- Cabeçalho do card com timestamp e link de edição -->
      <h4 class="card-header">
        {{ entry.date_added|date:'M d, Y H:i' }}
        <small><a href="{% url 'learning_logs:edit_entry' entry.id %}">
          edit entry</a></small>
      </h4>
      <!-- Corpo do card com texto de entrada -->
      <div class="card-body">{{ entry.text|linebreaks }}</div>
    </div>
  {% empty %}
    <p>There are no entries for this topic yet.</p>
  {% endfor %}

{% endblock content %}
```

Primeiro, inserimos o tópico no bloco page_header 1. Em seguida, excluímos a estrutura de lista não ordenada utilizada anteriormente nesse template. Em vez de fazer de cada entrada um item de lista, abrimos um elemento div com seletor card 2. Esse card comporta dois elementos aninhados: um para armazenar o timestamp e o link para editar a entrada, e outro para armazenar o corpo da entrada. O

seletor card se encarrega da maior parte da estilização de que precisamos para esse div; personalizamos o card adicionando uma pequena margem na parte inferior de cada card (mb-3).

O primeiro elemento no card é um cabeçalho, que é um elemento <h4> com o seletor card-header 3. Esse cabeçalho contém a data em que a entrada foi feita e um link para editá-la. A tag <small> em torno do link edit_entry faz com que pareça um pouco menor que o timestamp 4. O segundo elemento é um div com o seletor card-body 5, que insere o texto da entrada em uma caixa simples no card. Perceba que o código Django para incluir as informações na página está inalterado; mudamos somente os elementos que afetam a aparência da página. Já que não temos mais uma lista não ordenada, substituímos as tags de item de lista ao redor da mensagem da lista vazia por tags de parágrafo simples 6.

A Figura 20.3 mostra a página do tópico com um visual novo. As funcionalidades do Registro de Aprendizagem não mudaram. No entanto, agora nosso site parece mais profissional e mais convidativo aos usuários.

Caso queira utilizar um template da Bootstrap diferente para um projeto, siga um processo semelhante ao que fizemos até agora neste capítulo. Copie o template que deseja usar em *base.html* e modifique os elementos que têm o conteúdo propriamente dito para que o template exiba as informações do projeto. Depois, use as ferramentas individuais de estilização da Bootstrap para estilizar o conteúdo em cada página.

NOTA *O projeto Bootstrap disponibiliza documentação excelente. Acesse a página inicial em* https://getbootstrap.com *e clique em* Docs *para saber mais sobre o que a Bootstrap oferece.*

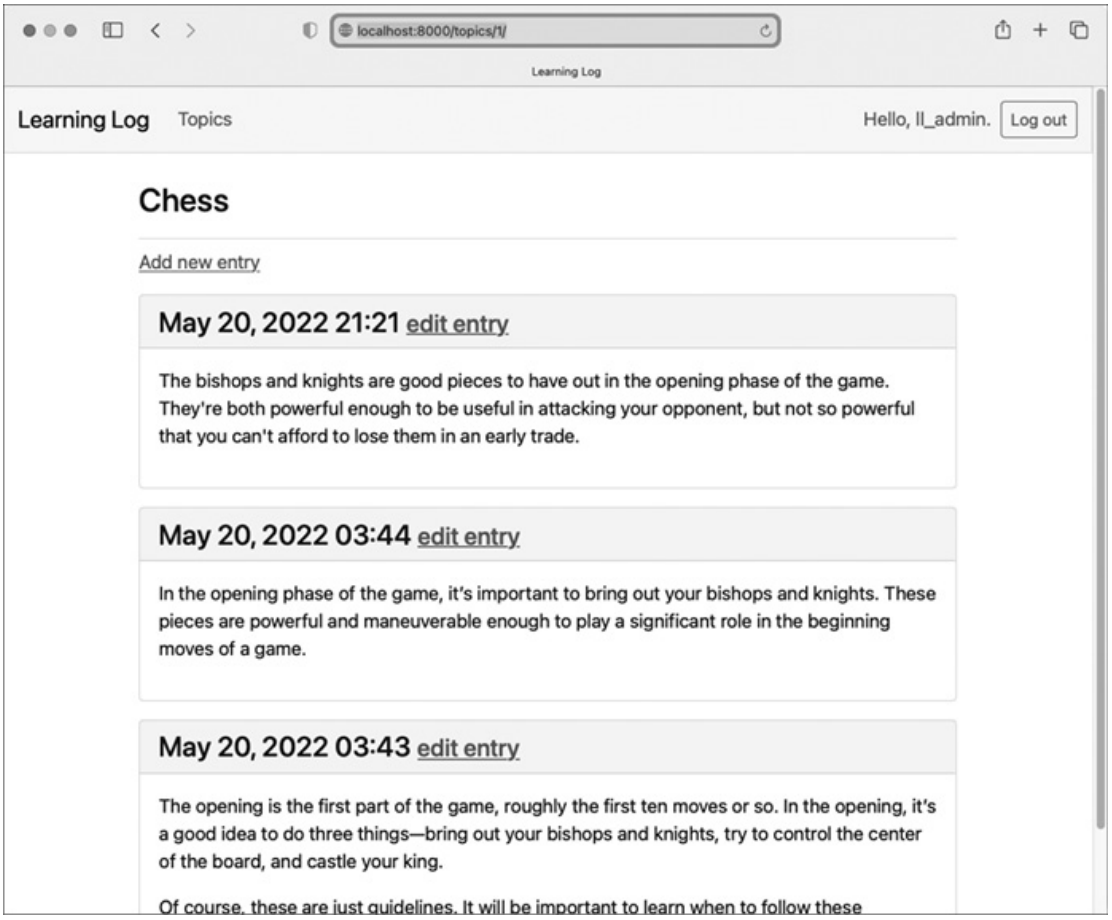


Figura 20.3: A página de tópico com estilização da Bootstrap.

FAÇA VOCÊ MESMO

20.1 Outros formulários: Aplicamos os estilos da Bootstrap à página login. Faça alterações parecidas no restante das páginas baseadas em formulários, incluindo new_topic, new_entry, edit_entry e register.

20.2 Blog estiloso: Use a Bootstrap para estilizar o projeto do Blog que você criou no Capítulo 19.

Fazendo o deploy do projeto Registro de Aprendizagem

Agora que temos um projeto com aparência profissional, faremos o deploy dele para um servidor ativo. Assim, qualquer pessoa com uma conexão à internet conseguirá usá-lo. Usaremos o Platform.sh, uma plataforma baseada na web que possibilita gerenciar a implantação de aplicações web. Vamos subir e executar o Registro

de Aprendizagem no Platform.sh.

Como criar uma conta no Platform.sh

Para criar uma conta, acesse https://platform.sh e clique no botão **Free Trial**. O Platform.sh tem um nível gratuito que, até o momento em que eu escrevia este livro, não exige cartão de crédito. O período de avaliação possibilita fazer o deploy de uma aplicação com o mínimo de recursos. Ou seja, podemos testar o deploy do projeto antes recorrermos a um plano de hospedagem paga.

> **NOTA** *Os limites específicos dos planos de teste costumam mudar periodicamente, visto que as plataformas de hospedagem combatem spam e o abuso de recursos. É possível conferir os limites atuais do período de avaliação gratuito em https://platform.sh/free-trial.*

Instalando a CLI do Platform.sh

Para implantar e gerenciar um projeto no Platform.sh, precisaremos das ferramentas disponíveis na Interface de Linha de Comando (CLI). Para instalar a versão mais recente da CLI, visite https://docs.platform.sh/development/cli.html e siga as instruções para o seu sistema operacional.

Na maioria dos sistemas, podemos instalar a CLI executando o seguinte comando em um terminal:

```
$ curl -fsS https://platform.sh/cli/installer | php
```

Depois que esse comando terminar de executar, é necessário abrir uma nova janela de terminal antes usar a CLI.

> **NOTA** *É bem provável que esse comando não execute em um terminal padrão no Windows. Você pode usar o Windows Subsystem for Linux (WSL) ou um terminal Git Bash. Caso seja necessário instalar o PHP, é possível usar o instalador do XAMPP em https://apachefriends.org. Se tiver alguma dificuldade para instalar o Platform.sh CLI, confira as instruções de*

instalação mais detalhadas no Apêndice E.

Instalando platformshconfig

Precisaremos também instalar um pacote adicional, o platformshconfig. Esse pacote ajuda a identificar se o projeto está sendo executado em um sistema local ou em um servidor do Platform.sh. Em um ambiente virtual ativo, digite o seguinte comando:

```
(ll_env)learning_log$ pip install platformshconfig
```

Utilizaremos esse pacote para modificar as configurações do projeto quando estiver sendo executado no servido ativo.

Criando um arquivo requirements.txt

Como o servidor remoto precisa saber de quais pacotes o Registro de Aprendizagem depende, usaremos o pip para gerar um arquivo que enumere cada um deles. De novo, em um ambiente virtual ativo, digite o seguinte comando:

```
(ll_env)learning_log$ pip freeze > requirements.txt
```

O comando freeze solicita que o pip escreva os nomes de todos os pacotes atualmente instalados do projeto no arquivo *requirements.txt*. Abra esse arquivo para ver os pacotes e o número de versões instalados em nosso projeto:

requirements.txt

```
asgiref==3.5.2
beautifulsoup4==4.11.1
Django==4.1
django-bootstrap5==21.3
platformshconfig==2.4.0
soupsieve==2.3.2.post1
sqlparse==0.4.2
```

Como podemos ver, o Registro de Aprendizagem depende das versões específicas de sete pacotes diferentes, ou seja, exige um ambiente compatível para ser executado corretamente em um servidor remoto. (Instalamos três desses pacotes manualmente, e quatro deles foram instalados automaticamente como dependências

desses pacotes.)

Quando fizermos o deploy do Projeto Registro de Aprendizagem, o Platform.sh instalará todos os pacotes enumerados no arquivo *requirements.txt*, criando um ambiente com os mesmos pacotes que estamos usando localmente. Por isso, podemos ter certeza de que o projeto implantado funcionará exatamente como funciona em nosso sistema local. Trata-se de uma abordagem fundamental para gerenciar um projeto, sobretudo quando começamos a desenvolver e a manter diversos projetos em um sistema.

NOTA *Se o número da versão de um pacote listado em seu sistema for diferente do que é mostrado aqui, mantenha a versão do seu sistema.*

Requisitos adicionais de deploy

O servidor ativo exige dois pacotes adicionais. São pacotes usados para atender o projeto em um ambiente de produção, em que muitos usuários podem fazer muitas requisições ao mesmo tempo.

No mesmo diretório em que *requirements.txt* é salvo, crie um arquivo novo chamado *requirements_remote.txt*. Adicione os dois pacotes a seguir:

requirements_remote.txt

```
# Requisitos para projeto
gunicorn
psycopg2
```

O pacote *gunicorn* responde às requisições à medida que chegam ao servidor remoto; isso substitui o servidor de desenvolvimento que estamos usando localmente. O pacote psycopg2 é necessário, pois possibilita que o Django gerencie o banco de dados Postgres que o Platform.sh usa. O *Postgres* é um banco de dados open source preparado para aplicações em produção.

Adicionando arquivos de configuração

Toda plataforma de hospedagem requer um pouco de configuração para que um projeto seja adequadamente executado em seus servidores. Nesta seção, adicionaremos três arquivos de configuração:

- *.platform.app.yaml* Principal arquivo de configuração do projeto. Informa ao Platform.sh que tipo de projeto estamos tentando implantar e quais tipos de recursos nosso projeto precisa, incluindo comandos para criá-lo no servidor.
- *.platform/routes.yaml* Arquivo que define as rotas para o nosso projeto. Quando o Platform.sh recebe uma requisição, essa é a configuração que ajuda a direcioná-la para nosso projeto específico.
- *.platform/services.yaml* Arquivo que define quaisquer serviços adicionais e necessários ao nosso projeto.

Todos esses arquivos são YAML (YAML Ain ‘t Markup Language, YAML Não É Uma Linguagem de Marcação). *YAML* é uma linguagem arquitetada para escrever arquivos de configuração; é desenvolvida para ser lida facilmente por humanos e computadores. É possível escrever ou modificar um típico arquivo YAML manualmente, ainda que um computador consiga ler e interpretar o arquivo sem equívocos.

Arquivos YAML são ótimos para a configuração de deploy, já que oferecem um bom controle sobre o que acontece durante o processo de implantação.

Fazendo os arquivos ocultos ficarem visíveis

A maioria dos sistemas operacionais ocultam arquivos e pastas que começam com um ponto, como *.platform*. Por padrão, quando abrimos um navegador de arquivos, não vemos esses tipos de arquivos e pastas. Mas como programador, é necessário vê-los. Vejamos como visualizar arquivos ocultos, dependendo do sistema

operacional:

- No Windows, abra o Explorador de Arquivos e, em seguida, abra uma pasta como *Este Computador*. Clique na guia **Exibir** e verifique se as **extensões de nome de arquivo** e os **itens ocultos** estão marcados.
- No macOS, pressione ⌘+SHIFT+. (ponto) em qualquer janela do Finder para ver pastas e arquivos ocultos.
- Em sistemas Linux, como o Ubuntu, pressione Ctrl+H em qualquer navegador de arquivos para exibir arquivos e pastas ocultos. Para que essa configuração seja permanente, abra um navegador de arquivos como Nautilus e clique na guia de opções (indicada por três linhas). Marque a caixa de seleção **Mostrar arquivos ocultos**.

Arquivo de configuração .platform.app.yaml

O primeiro arquivo de configuração é o mais extenso, porque controla o processo geral de deploy. Vamos mostrá-lo em partes; é possível inseri-lo manualmente em seu editor de texto ou fazer o download de uma cópia a partir dos recursos online em https://ehmatthes.github.io/pcc_3e.

Aqui está a primeira parte de *.platform.app.yaml*, que deve ser salvo no mesmo diretório que *manage.py*:

.platform.app.yaml

```
1 name: "ll_project"
  type: "python:3.10"

2 relationships:
    database: "db:postgresql"

  # A configuração da aplicação quando exposta na web
3 web:
    upstream:
      socket_family: unix
    commands:
4     start: "gunicorn -w 4 -b unix:$SOCKET ll_project.wsgi:application"
5   locations:
```

```
"/":
    passthru: true
"/static":
    root: "static"
    expires: 1h
    allow: true

  # O tamanho do disco permanente da aplicação (em MB)
6 disk: 512
```

Ao salvar esse arquivo, lembre-se de incluir o ponto no início do nome do arquivo. Se omitir o ponto, o Platform.sh não encontrará o arquivo e não conseguiremos fazer o deploy do projeto.

Por ora, não é necessário entender tudo que está escrito no arquivo *.platform.app.yaml*; vou ressaltar as partes mais importantes da configuração. O arquivo começa especificando o `name` do projeto, que estamos chamando de `'ll_project'` para que seja coerente com o nome que usamos ao iniciar o projeto `1`. É necessário também especificar a versão do Python que estamos utilizando (3.10, no momento em que eu redigia esta obra). Podemos encontrar uma lista de versões compatíveis em https://docs.platform.sh/languages/python.html.

A seguir, vemos uma seção denominada `relationships`, que define outros serviços necessários ao projeto `2`. Aqui, a única relação é com um banco de dados Postgres. Depois disso, temos a seção `web` `3`. A seção `commands:start` informa ao Platform.sh qual processo usar para atender às requisições recebidas. Nesse caso, estamos especificando que `gunicorn` se encarregará das requisições `4`. Esse comando substitui o comando `python manage.py runserver` que usamos localmente.

A seção `locations` informa ao Platform.sh para onde enviar as requisições recebidas `5`. A maioria das requisições deve ser passada para `gunicorn`; nossos arquivos *urls.py* informarão exatamente ao `gunicorn` como lidar com essas requisições. As requisições de arquivos estáticos serão tratadas separadamente e serão atualizadas uma vez por hora. A última linha mostra que estamos solicitando 512 MB de espaço em disco em um dos servidores do Platform.sh `6`.

Vejamos o restante do arquivo *.platform.app.yaml*:

```
  -- trecho de código omitido --
  disk: 512

  # Define mounts de leitura/gravação local
1 mounts:
    "logs":
      source: local
      source_path: logs

  # Os hooks executados em vários pontos do ciclo de vida da aplicação
2 hooks:
    build: |
3     pip install --upgrade pip
      pip install -r requirements.txt
      pip install -r requirements_remote.txt

      mkdir logs
4     python manage.py collectstatic
      rm -rf logs
5   deploy: |
      python manage.py migrate
```

A seção `mount` `1` nos permite definir diretórios em que podemos ler e gravar dados enquanto o projeto está em execução. Essa seção define um diretório*/logs* para o projeto implantado.

A seção `hooks` `2` define as ações que são executadas em diversos pontos durante o processo de deploy. Na seção `build`, instalamos todos os pacotes necessários para atender ao projeto no ambiente ativo `3`. Além disso, executamos `collectstatic` `4`, que coleta todos os arquivos estáticos necessários para o projeto em um só lugar para que possam ser atendidos com eficiência.

Por último, na seção `deploy` `5`, especificamos que as migrações devem ser executadas sempre que o projeto for implantado. Em um projeto simples, isso não terá efeito quando não houver mudanças.

Os outros dois arquivos de configuração são menores; vamos escrevê-los agora.

Arquivo de configuração routes.yaml

Uma *rota* é o caminho que uma requisição segue à medida que é processada pelo servidor. Quando uma requisição é recebida pelo Platform.sh, a plataforma precisa saber para onde enviá-la.

Crie uma pasta nova chamada *.platform*, no mesmo diretório que *manage.py*. Não se esqueça de incluir o ponto no início do nome. Dentro dessa pasta, crie um arquivo chamado *routes.yaml* e digite o seguinte:

.platform/routes.yaml

```
# Cada rota descreve como um URL de entrada será processado pelo Platform.sh.

"https://{default}/":
    type: upstream
    upstream: "ll_project:http"

"https://www.{default}/":
    type: redirect
    to: "https://{default}/"
```

Esse arquivo garante que requisições como *https://project_url.com* e *www.project_url.com* sejam encaminhadas para o mesmo lugar.

Arquivo de configuração services.yaml

Nosso último arquivo de configuração especifica os serviços que nosso projeto precisa para ser executado. Salve esse arquivo no diretório *.platform/*, ao lado de *routes.yaml*:

.platform/routes.yaml

```
# Cada serviço listado será implantado no próprio contêiner
# como parte de seu do projeto Platform.sh.

db:
    type: postgresql:12
    disk: 1024
```

O arquivo define um serviço, um banco de dados Postgres.

Modificando settings.py para Platform.sh

Agora, é necessário adicionarmos uma seção no final de *settings.py* para modificar algumas configurações do ambiente do Platform.sh. Adicione o seguinte código ao final de *settings.py*:

settings.py

```
    -- trecho de código omitido --
    # Configurações do Platform.sh.
1   from platformshconfig import Config

    config = Config()
2   if config.is_valid_platform():
3       ALLOWED_HOSTS.append('.platformsh.site')

4       if config.appDir:
            STATIC_ROOT = Path(config.appDir) / 'static'
5       if config.projectEntropy:
            SECRET_KEY = config.projectEntropy

        if not config.in_build():
6           db_settings = config.credentials('database')
            DATABASES = {
                'default': {
                    'ENGINE': 'django.db.backends.postgresql',
                    'NAME': db_settings['path'],
                    'USER': db_settings['username'],
                    'PASSWORD': db_settings['password'],
                    'HOST': db_settings['host'],
                    'PORT': db_settings['port'],
                },
            }
```

Normalmente, inserimos as instruções `import` no início de um módulo, porém, nesse caso, é bom manter todas as configurações específicas e remotas de controle em uma seção. Aqui, importamos `Config` de `platformshconfig` `1`, que ajuda a determinar as configurações no servidor remoto. Modificamos apenas as configurações se o método `config.is_valid_platform()` retornar `True` `2`, sinalizando que as configurações estão sendo usadas em um servidor Platform.sh.

Modificamos `ALLOWED_HOSTS` para possibilitar que o projeto seja

atendido por hosts que terminam em *.platformsh.site* 3. Todos os projetos implantados no nível gratuito serão atendidos usando esse mesmo host. Se as configurações estiverem sendo carregadas no diretório 4 da aplicação implantada, definimos STATIC_ROOT para que os arquivos estáticos sejam corretamente atendidos. Definimos também uma SECRET_KEY mais segura no servidor remoto 5.

Para concluir, configuramos o banco de dados de produção 6. O banco apenas é definido se o processo de build tiver terminado de ser executado e o projeto estiver sendo atendido. Tudo o que estamos vendo aqui é necessário, pois é assim que o Django se comunica com o servidor Postgres que o Platform.sh definiu para o projeto.

Usando o Git para rastrear os arquivos do projeto

Conforme vimos no Capítulo 17, o Git é um programa de controle de versão/versionamento que possibilita tirar um snapshot do código do projeto sempre que implementarmos uma funcionalidade nova com sucesso. Se algo sair errado, podemos retornar facilmente ao último snapshot da tarefa do projeto; por exemplo, se introduzirmos sem querer um bug enquanto trabalhamos em uma funcionalidade nova. Chamamos cada snapshot de *commit*.

Com o Git, é possível implementar funcionalidades novas sem se preocupar em prejudicar nada do projeto. Quando estamos fazendo o deploy em um servidor ativo, é necessário termos certeza de que estamos fazendo deploy da versão do projeto que está funcionando. Para ler mais sobre o Git e o controle de versão, confira o Apêndice D.

Instalando o Git

Talvez o Git já esteja instalado em seu sistema. Vamos descobrir: abra uma janela nova de terminal e execute o comando **git --version**:

```
(ll_env)learning_log$ git --version
git version 2.30.1 (Apple Git-130)
```

Caso receba uma mensagem de que o Git não está instalado, confira as instruções de instalação no Apêndice D.

Configurando o Git

O Git rastreia quem faz alterações em um projeto, mesmo quando somente uma pessoa está trabalhando no projeto. Para isso, o Git precisa saber seu nome de usuário e e-mail. Embora seja necessário um nome de usuário, você pode criar um e-mail apenas para a prática de seus projetos:

```
(ll_env)learning_log$ git config --global user.name "eric"
(ll_env)learning_log$ git config --global user.email "eric@example.com"
```

Caso esqueça dessas etapas, o Git solicitará essas informações quando você fizer o primeiro commit.

Ignorando arquivos

Como não precisamos do Git para rastrear todos os arquivos do projeto, solicitaremos que ignore alguns arquivos. Crie um arquivo chamado *.gitignore* na pasta que contém *manage.py*. Veja que o nome desse arquivo começa com um ponto e não tem extensão de arquivo. Vejamos o código a ser inserido em *.gitignore*:

.gitignore

```
ll_env/
__pycache__/
*.sqlite3
```

Solicitamos que Git ignore todo o diretório *ll_env*, porque podemos recriá-lo automaticamente a qualquer momento. Além do mais, não rastreamos o diretório *__pycache__*, que contém os arquivos *.pyc*, criados automaticamente quando os arquivos *.py* são executados. Não rastreamos as alterações no banco de dados local, porque não é uma prática recomendada: se estivermos usando o SQLite em um servidor, podemos acidentalmente sobrescrever o banco de dados ativo com nosso banco de dados de teste local ao enviarmos o projeto para o servidor. O asterisco em `*.sqlite3` instrui o Git a ignorar

qualquer arquivo que termine com a extensão *.sqlite3*.

NOTA *Caso esteja usando o macOS, adicione* .DS_Store *em seu arquivo* .gitignore*. É um arquivo que armazena informações sobre as configurações de pasta no macOS e não tem nada a ver com esse projeto.*

Fazendo commit do projeto

Precisamos inicializar um repositório Git para o Registro de Aprendizagem, adicionar todos os arquivos necessários ao repositório e fazer o commit do estado inicial do projeto. Vejamos como fazer isso:

```
1 (ll_env)learning_log$ git init
  Initialized empty Git repository in /Users/eric/.../learning_log/.git/
2 (ll_env)learning_log$ git add .
3 (ll_env)learning_log$ git commit -am "Ready for deployment to Platform.sh."
  [main (root-commit) c7ffaad] Ready for deployment to Platform.sh.
  42 files changed, 879 insertions(+)
  create mode 100644 .gitignore
  create mode 100644 .platform.app.yaml
  -- trecho de código omitido --
  create mode 100644 requirements_remote.txt
4 (ll_env)learning_log$ git status
  On branch main
  nothing to commit, working tree clean
  (ll_env)learning_log$
```

Executamos o comando git init para inicializar um repositório vazio no diretório que contém o Registro de Aprendizagem 1. Em seguida, usamos o comando git add, que adiciona todos os arquivos que não estão sendo ignorados no repositório 2. (Não se esqueça do ponto.) Depois, executamos o comando git commit -am "*mensagem de commit*": a flag -a instrui o Git a incluir todos os arquivos alterados nesse commit, e a flag -m solicita que o Git registre uma mensagem de log 3.

Executar o comando git status 4 indica que estamos no branch *main* e que nosso diretório de trabalho está *limpo*. É o status que desejamos ver sempre que fizermos o push de um projeto para um

servidor remoto.

Criando um projeto no Platform.sh

Até agora, o Projeto Registro de Aprendizagem ainda é executado em nosso sistema local e também é configurado para ser adequadamente executado em um servidor remoto. Usaremos a CLI do Platform.sh para criar um projeto novo no servidor e, em seguida, enviar nosso projeto para o servidor remoto.

Lembre-se de abrir um terminal. No diretório *learning_log/*, execute o seguinte comando:

```
(ll_env)learning_log$ platform login
Opened URL: http://127.0.0.1:5000
Please use the browser to log in.
-- trecho de código omitido --
1 Do you want to create an SSH configuration file automatically? [Y/n] Y
```

Esse comando abrirá uma guia do navegador onde podemos efetuar o login. Após o login, podemos fechar a guia do navegador e retornar ao terminal. Se for solicitado a criar um arquivo de configuração SSH 1, digite **Y** para que possa se conectar ao servidor remoto posteriormente.

Agora, criaremos um projeto. Como temos diversas saídas, analisaremos o processo de criação em seções. Comece executando o commando **create**:

```
(ll_env)learning_log$ platform create
* Project title (--title)
Default: Untitled Project
1 > ll_project

* Region (--region)
The region where the project will be hosted
  --trecho de código--
  [us-3.platform.sh] Moses Lake, United States (AZURE) [514 gC02eq/kWh]
2 > us-3.platform.sh
* Plan (--plan)
Default: development
Enter a number to choose:
  [0] development
```

--trecho de código--

3 > **0**

* Environments (--environments)
The number of environments
Default: 3

4 > **3**

* Storage (--storage)
The amount of storage per environment, in GiB
Default: 5

5 > **5**

O primeiro prompt solicita um nome para o projeto 1, logo utilizamos o nome **ll_project.** O próximo prompt pergunta em qual região gostaríamos que o servidor estivesse em 2. Escolha o servidor mais próximo de sua localização; para mim, é us-3.platform.sh. Para o resto dos prompts, podemos aceitar os padrões: um servidor desenvolvimento menor 3, três ambientes para o projeto 4 e 5 GB de armazenamento para o projeto geral 5.

É necessário respondermos a mais três prompts:

Default branch (--default-branch)
The default Git branch name for the project (the production environment)
Default: main

1 > **main**

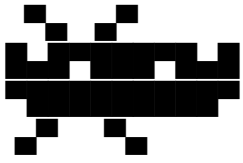
Git repository detected: /Users/eric/.../learning_log

2 Set the new project ll_project as the remote for this repository? [Y/n] **Y**

The estimated monthly cost of this project is: $10 USD

3 Are you sure you want to continue? [Y/n] **Y**

The Platform.sh Bot is activating your project



The project is now ready!

Um repositório Git pode ter diversos branchs; o Platform.sh está nos

perguntando se o branch padrão para o projeto deve ser o main 1. Em seguida, pergunta se queremos conectar o repositório do projeto local ao repositório remoto 2. Por último, somos informados de que esse projeto custará aproximadamente US$10 por mês se o mantivermos em execução além do período de avaliação gratuita 3. Se ainda não inseriu um cartão de crédito, não precisa se preocupar com esse custo. O Platform.sh simplesmente suspenderá seu projeto caso exceda os limites de avaliação gratuita sem adicionar um cartão de crédito.

Fazendo o push para o Platform.sh

O último passo antes de verificarmos a versão ativa do projeto é enviar nosso código para o servidor remoto. Para fazer isso, execute o seguinte comando:

```
(ll_env)learning_log$ platform push
1 Are you sure you want to push to the main (production) branch? [Y/n] Y
  -- trecho de código omitido --
The authenticity of host 'git.us-3.platform.sh (...)' can't be established.
RSA key fingerprint is SHA256:Tvn...7PM
2 Are you sure you want to continue connecting (yes/no/[fingerprint])? Y
Pushing HEAD to the existing environment main
  -- trecho de código omitido --
To git.us-3.platform.sh:3pp3mqcexhlvy.git
 * [new branch]      HEAD -> main
```

Ao executarmos o comando **platform push**, seremos perguntados mais uma vez se queremos confirmar o push do projeto 1. Podemos ver também uma mensagem sobre a autenticidade do Platform.sh, caso seja a primeira vez que você se conecta ao site 2. Digite **Y** para cada um desses prompts, e veremos diversas saídas passarem pela tela. A princípio, essas saídas parecem um tanto confusas. No entanto, se algo sair errado, essa saída pode ajudar muito na solução de problemas. Caso examine a saída, poderá ver onde o Platform.sh instala os pacotes necessários, coleta arquivos estáticos, aplica migrações e configura URLs para o projeto.

NOTA *Talvez você consiga identificar um erro com facilidade,*

como um erro de digitação em um dos arquivos de configuração. Se isso ocorrer, corrija o erro no seu editor de texto, salve o arquivo e execute novamente o comando git commit*. Em seguida, execute* platform push *mais uma vez.*

Visualizando o projeto ativo

Assim que o push for concluído, podemos abrir o projeto:

```
(ll_env)learning_log$ platform url
Enter a number to open a URL
  [0] https://main-bvxea6i-wmye2fx7wwqgu.us-3.platformsh.site/
  -- trecho de código omitido --
> 0
```

O comando platform url lista os URLs associados a um projeto implantado; é possível escolher diversos URLs válidos para o projeto. Escolha um e seu projeto deve abrir em uma nova guia do navegador! Mesmo sendo parecido com o projeto que estamos executando localmente, podemos compartilhar esse URL com qualquer pessoa no mundo para que possa acessá-lo e usá-lo.

NOTA *Caso faça deploy de um projeto usando uma conta de avaliação, não se surpreenda se às vezes demorar mais do que o normal para uma página ser carregada. Na maioria das plataformas de hospedagem, os recursos gratuitos ociosos geralmente são suspensos e reiniciados apenas quando requisições novas chegam. A maioria das plataformas responde melhor aos planos pagos de hospedagem.*

Refinando a implantação do Platform.sh

Agora refinaremos a implantação criando um superusuário, assim como fizemos localmente. Faremos também com que o projeto fique mais seguro alterando a configuração DEBUG para False. Dessa forma, as mensagens de erro não mostram aos usuários nenhuma informação extra que eles consigam usar para atacar o servidor.

Criando um superusuário no Platform.sh

Mesmo que o banco de dados para o projeto esteja configurado, está completamente vazio. Todos os usuários que criamos anteriormente existem apenas em nossa versão local do projeto.

Para criar um superusuário na versão ativa do projeto, iniciaremos uma sessão SSH (secure socket shell) em que podemos executar comandos de gerenciamento no servidor remoto:

```
(ll_env)learning_log$ platform environment:ssh


 ___ _      _    __                  _
| _ \ |__ _| |_ / _|___ _ _ _ __    __| |_
|  _/ / _` |  _|  _/ _ \ '_| '  \ _(_-< ' \
|_| |_\__,_|\__|_| \___/_| |_|_|_(_)__/_||_|

Welcome to Platform.sh.

1 web@ll_project.0:~$ ls
  accounts  learning_logs  ll_project  logs  manage.py  requirements.txt
     requirements_remote.txt  static
2 web@ll_project.0:~$ python manage.py createsuperuser
3 Username (leave blank to use 'web'): ll_admin_live
  Email address:
  Password:
  Password (again):
  Superuser created successfully.
4 web@ll_project.0:~$ exit
  logout
  Connection to ssh.us-3.platform.sh closed.
5 (ll_env)learning_log$
```

Ao executarmos pela primeira vez o comando **platform environment:ssh,** podemos obter outro prompt sobre a autenticidade desse host. Se vir essa mensagem, digite **Y** e você deve se conectar em uma sessão de terminal remoto.

Após executarmos o comando **ssh,** nosso terminal se comporta como um terminal no servidor remoto. Observe que o prompt foi alterado para indicar que estamos conectados a uma sessão web, associada ao projeto chamado ll_project 1. Se executarmos o comando **ls,** veremos os arquivos enviados para o servidor do Platform.sh.

Execute o mesmo comando **createuperuser** que usamos no Capítulo 18 2. Desta vez, forneci um nome de usuário de administrador, **ll_admin_live**, diferente daquele que usei localmente 3. Quando terminar de trabalhar na sessão de terminal remoto, digite o comando **exit** 4. O prompt indicará que você está usando novamente seu sistema local 5.

Agora, podemos adicionar */admin/* ao final do URL da aplicação e logarmos no site admin. Se outras pessoas estiverem usando seu projeto, fique ciente de que terá acesso a todos os dados delas! Leve essa responsabilidade a sério, e os usuários sempre confiarão os próprios dados a você.

NOTA *Os usuários do Windows usarão os mesmos comandos mostrados aqui (como* ls *em vez de* dir*), porque estamos executando um terminal Linux por meio de uma conexão remota.*

Protegendo um projeto ativo

Temos um problema flagrante de segurança no modo como nosso projeto está atualmente implantado: a configuração DEBUG = True em *settings.py*, que fornece mensagens de depuração quando ocorrem erros. As páginas de erro do Django fornecem informações indispensáveis de depuração quando estamos desenvolvendo um projeto; no entanto, se habilitadas em um servidor ativo, essas páginas também fornecem informações aos invasores.

Para ver o quanto isso é prejudicial, acesse a página inicial do seu projeto implantado. Faça login na conta de um usuário e adicione*/topics/999/*ao final do URL da página inicial. Partindo do princípio de que você não tenha criado milhares de tópicos, é possível ver uma página com a mensagem *DoesNotExist at /topics/999/*. Se deslizar o mouse na tela, verá muitas informações sobre o projeto e o servidor. Não queremos que usuários vejam isso, e certamente não queremos que essa informação fique disponível a

qualquer pessoa interessada em atacar o site.

É possível impedir que essas informações sejam exibidas no site definindo `DEBUG = False` na parte de *settings.py* que usa apenas a versão implantada do projeto. Dessa forma, ainda vemos as informações de depuração localmente, onde são úteis, mas não serão exibidas no site.

Abra *settings.py* no editor de texto e adicione uma linha de código à parte que modifica as configurações do Platform.sh:

settings.py

```
    -- trecho de código omitido --
    if config.is_valid_platform():
        ALLOWED_HOSTS.append('.platformsh.site')
        DEBUG = False
        -- trecho de código omitido --
```

Todo o trabalho para definir a configuração da versão implantada do projeto valeu a pena. Se quisermos ajustar a versão ativa do projeto, basta alterarmos a parte relevante da configuração que definimos anteriormente.

Commit e push de alterações

Agora, precisamos efetuar o commit das alterações feitas em *settings.py* e enviá-las para o Platform.sh. Vejamos uma sessão terminal mostrando a primeira parte desse processo:

```
1 (ll_env)learning_log$ git commit -am "Set DEBUG False on live site."
  [main d2ad0f7] Set DEBUG False on live site.
    1 file changed, 1 insertion(+)
2 (ll_env)learning_log$ git status
  On branch main
  nothing to commit, working tree clean
  (ll_env)learning_log$
```

Executamos `git commit` com uma breve mensagem de commit, porém descritiva `1`. Lembre-se de que a flag `-am` garante que o Git crie todos os arquivos que foram alterados e registre a mensagem de log. O Git reconhece que um arquivo foi alterado e efetua o commit dessa alteração no repositório.

Executar **git status** mostra que estamos trabalhando no branch main do repositório e que agora não há novas alterações para fazer o commit 2. É importante verificar o status antes efetuar o push para um servidor remoto. Caso não veja um status limpo, significa que algumas alterações não foram comitadas e que não serão enviadas para o servidor. Podemos tentar executar mais uma vez o comando commit; se não tiver certeza de como resolver o problema, leia o Apêndice D para entender melhor como trabalhar com o Git.

Agora, faremos o push do repositório atualizado para o Platform.sh:

```
(ll_env)learning_log$ platform push
Are you sure you want to push to the main (production) branch? [Y/n] Y
Pushing HEAD to the existing environment main
--trecho de código--
  To git.us-3.platform.sh:wmye2fx7wwqgu.git
     fce0206..d2ad0f7  HEAD -> main
(ll_env)learning_log$
```

O Platform.sh reconhece que o repositório foi atualizado e recria o projeto a fim de garantir que todas as alterações foram levadas em consideração. Como o banco de dados não é recriado, não perdemos nenhum dado.

Para garantir que essa alteração ocorra, visite o URL */topics/999/* novamente. Você deve ver apenas a mensagem *Server Error (500)*, sem nenhuma informação sensível sobre o projeto.

Criando páginas de erro personalizadas

No Capítulo 19, configuramos o Registro de Aprendizagem para retornar um erro 404, caso o usuário solicitasse um tópico ou entrada que não lhe pertencesse. Agora acabamos de ver também um 500 server error. Em geral, um erro 404 significa que seu código Django está correto, mas o objeto requisitado não existe. Um erro 500 normalmente significa que há um erro no código escrito, como um erro em uma função em *views.py*. Atualmente, o Django retorna a mesma página de erro genérica em ambas as situações, mas podemos desenvolver nossos próprios templates de página de

erro 404 e 500, que correspondam à aparência geral do Registro de Aprendizagem. Esses templates pertencem ao diretório root de templates.

Criando templates personalizados

Na pasta *learning_log*, crie uma pasta nova chamada *templates*. Em seguida, crie um arquivo novo chamado *404.html*; o path para esse arquivo deve ser *learning_log/templates/404.html*. Vejamos o código para esse arquivo:

404.html

```
{% extends "learning_logs/base.html" %}

{% block page_header %}
  <h2>The item you requested is not available. (404)</h2>
{% endblock page_header %}
```

Esse template simples fornece as informações genéricas da página de erro 404, mas é estilizado para ficar como o restante do site.

Crie outro arquivo chamado *500.html* usando o seguinte código:

500.html

```
{% extends "learning_logs/base.html" %}

{% block page_header %}
  <h2>There has been an internal error. (500)</h2>
{% endblock page_header %}
```

Esses arquivos novos exigem uma pequena alteração em *settings.py*.

settings.py

```
-- trecho de código omitido --
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates'],
        'APP_DIRS': True,
        -- trecho de código omitido --
    },
```

```
]
-- trecho de código omitido --
```

Essa alteração instrui o Django a procurar no diretório root os templates de página de erro e quaisquer outros que não estejam associados a uma aplicação específica.

Fazendo o push das alterações para o Platform.sh

Agora, precisamos efetuar o commit das alterações que acabamos de fazer e enviá-las para o Platform.sh:

```
1 (ll_env)learning_log$ git add .
2 (ll_env)learning_log$ git commit -am "Added custom 404 and 500 error pages."
  3 files changed, 11 insertions(+), 1 deletion(-)
  create mode 100644 templates/404.html
  create mode 100644 templates/500.html
3 (ll_env)learning_log$ platform push
  -- trecho de código omitido --
    To git.us-3.platform.sh:wmye2fx7wwqgu.git
       d2ad0f7..9f042ef  HEAD -> main
  (ll_env)learning_log$
```

Executamos o comando `git add .` `1` porque criamos alguns arquivos novos no projeto. Depois, fizemos o commit das alterações `2` e o push do projeto atualizado para o Platform.sh `3`.

Agora, quando aparece, uma página de erro deve ter o mesmo estilo que o restante do site, proporcionando uma experiência de usuário mais amigável quando surgirem erros.

Desenvolvimento contínuo

Talvez você queira desenvolver ainda mais o Registro de Aprendizagem após o envio inicial para o servidor ativo, ou talvez queira desenvolver os próprios projetos para deploy. Caso faça isso, há um processo bastante consistente para atualizar os projetos.

Em primeiro lugar, faça as alterações necessárias em seu projeto local. Se suas alterações resultarem em arquivos novos, adicione esses arquivos ao repositório Git com o comando `git add .` (não se esqueça de incluir o ponto no final). Qualquer alteração que exija

uma migração de banco de dados precisará desse comando, pois cada migração gera um arquivo novo.

Em segundo lugar, faça o commit das alterações em seu repositório usando `git commit -am "mensagem de commit"`. Em seguida, envie suas alterações para o Platform.sh, com o comando `platform push`. Visite seu projeto e verifique se as alterações esperadas surtiram efeito.

Como é fácil cometer erros durante esse processo, não se surpreenda quando algo der errado. Se o código não funcionar, revise o que fez e tente identificar o erro. Caso não consiga identificar o erro ou descobrir como desfazê-lo, confira as sugestões de ajuda no Apêndice C. Não tenha vergonha de pedir ajuda: todos os programadores aprenderem a desenvolver projetos perguntando as mesmas coisas que você provavelmente perguntará. Sempre tem alguém que ficará feliz em ajudá-lo. Resolver cada problema que surge o ajuda a desenvolver cada vez mais suas habilidades até que você esteja criando projetos significativos e confiáveis e respondendo às perguntas de outros programadores também.

Excluindo um projeto no Platform.sh

Trata-se de uma ótima prática executar o processo de implantação diversas vezes com o mesmo projeto ou com uma série de projetos pequenos a fim de dominar o processo de deploy. No entanto, é necessário saber como excluir um projeto que foi implantado. O Platform.sh também limita o número de projetos que podemos hospedar gratuitamente, e não queremos bagunçar uma conta com projetos que usamos para praticar.

Podemos excluir um projeto com a CLI:

```
(ll_env)learning_log$ platform project:delete
```

Você será solicitado a confirmar se deseja realizar essa ação destrutiva. Responda aos prompts e seu projeto será excluído.

O comando `platform create` também forneceu ao repositório Git local uma referência ao repositório remoto nos servidores Platform.sh.

Podemos remover também esse controle da linha de comando:

```
(ll_env)learning_log$ git remote
platform
(ll_env)learning_log$ git remote remove platform
```

O comando git remote enumera os nomes de todos os URLs remotos associados ao repositório atual. O comando git remote remove *nome_remoto* exclui esses URLs remotos do repositório local.

É possível também excluir os recursos de um projeto fazendo login no site Platform.sh e visitando seu dashboard em https://console.platform.sh. Essa página especifica todos os seus projetos ativos. Clique nos três pontos na caixa de um projeto e clique em **Edit Plan**. É uma página de preços para o projeto; na parte inferior da página, clique no botão **Delete Project** e você verá uma página de confirmação na qual poderá prosseguir com a exclusão. Ainda que tenha excluído seu projeto com a CLI, é bom se familiarizar com o dashboard de qualquer provedor de hospedagem que você usa para fazer deploy.

NOTA *Excluir um projeto no Platform.sh não afeta sua versão local do projeto. Se ninguém usou seu projeto implantado e você está apenas praticando o processo de deploy, é perfeitamente cabível excluir seu projeto no Platform.sh e reimplantá-lo. Esteja ciente de que, se as coisas pararem de funcionar, você pode ter extrapolado as limitações do nível gratuito do host.*

FAÇA VOCÊ MESMO

20.3 Blog ativo: Faça o deploy do projeto do Blog em que está trabalhando no Platform.sh. Lembre-se de definir DEBUG como False, para que os usuários não vejam as páginas de erro completas do Django quando algo der errado.

20.4 Registro de Aprendizagem incrementado: Adicione uma funcionalidade ao Registro de Aprendizagem e envie essa alteração para seu deploy ativo. Tente fazer mudanças simples, como fornecer mais informações sobre o projeto na página inicial. Em seguida, tente adicionar uma funcionalidade mais avançada, como fornecer aos usuários a opção de criar um tópico público. Isso exigirá um atributo public como parte do modelo Topic (que deve ser definido como False por padrão) e um elemento de formulário na página new_topic que possibilita o usuário alterar um tópico de privado para público. Depois, será necessário migrar o projeto e revisar *views.py* para que qualquer tópico que seja público também fique visível aos usuários não autenticados.

Recapitulando

Neste capítulo, aprendemos como conferir aos projetos uma aparência simples, mas profissional, com a biblioteca Bootstrap e a aplicação django-bootstrap5. Com a Bootstrap, podemos escolher estilos que funcionarão consistentemente em quase todos os dispositivos que as pessoas usam para acessar seu projeto.

Vimos os templates da Bootstrap e recorremos ao template *Navbar static* a fim de criar uma aparência simples para o Registro de Aprendizagem. Usamos um jumbotron para destacar a mensagem de uma página inicial e vimos como estilizar todas as páginas de um site de forma coerente.

Na parte final do projeto, aprendemos como fazer o deploy de um projeto para um servidor remoto a fim de que qualquer pessoa possa acessá-lo. Criamos uma conta no Platform.sh e instalamos algumas ferramentas que ajudam a gerenciar o processo de implantação. Usamos o Git para fazer o commit do projeto em um repositório e, em seguida, enviamos o repositório para um servidor remoto no Platform.sh. Por último, aprendemos a proteger uma aplicação definindo `DEBUG = False` no servidor ativo. Além disso, criamos páginas personalizadas de erro. Desse modo, quando erros inevitáveis aparecerem terão um aspecto melhor.

Agora que finalizou o Projeto Registro de Aprendizagem, você pode começar a desenvolver os próprios projetos. Comece simples e tenha certeza de que o projeto funciona antes de adicionar mais complexidade. Aproveite seu aprendizado contínuo e boa sorte com seus projetos!

APÊNDICE A

Instalação e solução de problemas

O Python tem muitas versões disponíveis e inúmeros modos de configurá-lo em cada sistema operacional. Se a abordagem no Capítulo 1 não funcionou, ou caso queira instalar uma versão diferente do Python do que a atualmente instalada, as instruções neste apêndice podem ajudá-lo.

Python no Windows

As instruções no Capítulo 1 mostram como instalar o Python com o instalador oficial em https://python.org. Se não conseguiu fazer com que o Python fosse executado após usar o instalador, as instruções de solução de problemas desta seção devem ajudá-lo a instalá-lo corretamente.

Usando py em vez de python

Caso execute um instalador Python recente e, em seguida, execute o comando **python** em um terminal, deverá ver o prompt Python para uma sessão de terminal (>>>). Quando não reconhece o comando python, o Windows acaba abrindo a Microsoft Store, pois assume que o Python não está instalado, ou você acaba recebendo uma mensagem como "Python was not found". Se a Microsoft Store abrir, feche-a; é melhor usar o instalador oficial do Python em https://python.org do que o disponibilizado pela Microsoft.

A solução mais simples, sem efetuar alterações no seu sistema, é

tentar o comando **py**. É um utilitário do Windows que encontra a versão mais recente do Python instalada em seu sistema, executando o interpretador. Se esse comando funcionar e você quiser usá-lo, basta usar **py** em qualquer lugar deste livro que vir o comando python ou python3.

Executando novamente o instalador

A razão mais comum pela qual o comando python não funciona é que as pessoas se esquecem de selecionar a opção Adicionar Python ao PATH, ao executar o instalador; é um erro fácil de cometer. A variável PATH é uma configuração do sistema que instrui o Python onde procurar programas usados com frequência. Nesse caso, o Windows não sabe como encontrar o interpretador Python.

Nessa situação, a solução mais simples é executar novamente o instalador. Se houver um instalador mais recente disponível em https://python.org, faça o download do novo instalador e o execute. Lembre-se de marcar a caixa **Adicionar Python ao PATH**.

Se já tiver o instalador mais recente, execute-o novamente e selecione a opção **Modificar**. Você verá uma lista de recursos opcionais; mantenha as opções padrão selecionadas nesta tela. Em seguida, clique em **Avançar** e marque a caixa **Adicionar Python às Variáveis de Ambiente**. Por último, clique em **Instalar**. O instalador reconhecerá que o Python já está instalado e adicionará a localização do interpretador Python à variável PATH. Não se esqueça de fechar todos os terminais abertos, porque ainda estarão usando a variável PATH antiga. Abra uma janela nova de terminal e execute o comando **python** mais uma vez; você deve ver um prompt Python (>>>).

Python no macOS

As instruções de instalação no Capítulo 1 usam o instalador oficial do Python em https://python.org. Embora o instalador oficial do Python

funcione há anos, algumas coisas podem sair errado. Esta seção ajudará se algo não estiver funcionando.

Instalando sem querer a versão da Apple do Python

Caso execute o comando `python3` e o Python ainda não estiver instalado em seu sistema, provavelmente verá uma mensagem de que the *command line developer tools* precisam ser instaladas. Neste momento, a melhor abordagem é fechar o pop-up que mostra a mensagem, fazer o download do instalador do Python em https://python.org, e executar o instalador.

Caso opte por instalar as ferramentas de desenvolvedor de linha de comando, o macOS instalará a versão da Apple do Python com as ferramentas de desenvolvedor. O único problema é que a versão da Apple do Python geralmente está um pouco atrasada em relação à versão oficial mais recente do Python. No entanto, ainda é possível fazer o download e executar o instalador oficial em `https://python`, e o `python3` apontará para a versão mais recente. Não se preocupe em ter as ferramentas de desenvolvimento instaladas; existem algumas ferramentas úteis lá, incluindo o sistema de controle de versão Git discutido no Apêndice D.

Python 2 em versões mais antigas do macOS

Em versões mais antigas do macOS, antes de Monterey (macOS 12), uma versão desatualizada do Python 2 era instalada por padrão. Nesses sistemas, o comando `python` aponta para o interpretador de sistema desatualizado. Se estiver usando uma versão do macOS com Python 2 instalado, lembre-se de utilizar o comando **`python3`**, assim, você sempre usará a versão do Python instalada.

Python no Linux

Quase todos os sistemas Linux vêm com o Python por padrão. No entanto, se a versão padrão no seu sistema for anterior ao

Python 3.9, é necessário instalar a versão mais recente. É possível também instalar a versão mais recente, caso queira os recursos mais recentes, como as mensagens refinadas de erro do Python. As instruções a seguir devem funcionar para a maioria dos sistemas baseados em APT.

Usando a instalação padrão do Python

Se quiser usar a versão do Python para a qual o python3 aponta, lembre-se de instalar esses três pacotes adicionais:

```
$ sudo apt install python3-dev python3-pip python3-venv
```

Esses pacotes incluem ferramentas úteis para desenvolvedores e ferramentas que possibilitam instalar pacotes de terceiros, como os usados na seção de projetos deste livro.

Instalando a versão mais recente do Python

Recorremos a um pacote chamado deadsnakes, que facilita a instalação de diversas versões do Python. Digite os seguintes comandos:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt update
$ sudo apt install python3.11
```

Esses comandos instalarão o Python 3.11 em seu sistema.

Digite o seguinte comando para iniciar uma sessão de terminal que executa Python 3.11:

```
$ python3.11
>>>
```

Neste livro, em qualquer lugar que vir o comando python, substitua-o por **python3.11**. Você também vai querer usar este comando quando executar programas no terminal.

Será necessário instalar mais dois pacotes para aproveitar ao máximo a instalação do Python:

```
$ sudo apt install python3.11-dev python3.11-venv
```

Esses pacotes incluem módulos necessários ao instalar e executar pacotes de terceiros, como os usados nos projetos na segunda

metade do livro.

NOTA *O pacote* deadsnakes *foi mantido ativamente por um longo tempo. Quando as versões mais recentes do Python forem lançadas, será possível usar esses mesmos comandos, substituindo* python3.11 *pela versão mais recente atualmente disponível.*

Verificando qual versão do Python você está usando

Se estiver tendo problemas para executar o Python ou instalar pacotes adicionais, pode ser útil saber exatamente qual versão do Python você está usando. Podemos ter diversas versões do Python instaladas e não termos certeza sobre qual versão estamos usando no momento.

Execute o seguinte comando em um terminal:

```
$ python --version
Python 3.11.0
```

Esse comando informa exatamente para qual versão o comando python está apontando. O comando mais curto python -V fornecerá a mesma saída.

Palavras reservadas e funções built-in do Python

O Python comporta o próprio conjunto de palavras reservadas e funções built-in. É fundamental conhecê-los, já que, no Python, sempre estamos nomeando variáveis: não podemos utilizar os mesmos nomes das palavras reservadas e não devemos usar os mesmos nomes das funções, caso contrário, podemos sobrescrevê-las.

Nesta seção, listaremos as palavras reservadas e os nomes de funções built-in do Python, para que você saiba quais nomes evitar.

Palavras reservadas do Python

Cada uma das palavras reservadas a seguir tem um significado específico. Caso tente utilizá-las como nome de uma variável, você receberá um erro.

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Funções built-on do Python

Você não receberá um erro se usar uma das seguintes funções built-in disponíveis como um nome de variável, mas substituirá o comportamento dessa função:

```
abs()            complex()     hash()          min()         slice()
aiter()          delattr()     help()          next()        sorted()
all()            dict()        hex()           object()      staticmethod()
any()            dir()         id()            oct()         str()
anext()          divmod()      input()         open()        sum()
ascii()          enumerate()   int()           ord()         super()
bin()            eval()        isinstance()    pow()         tuple()
bool()           exec()        issubclass()    print()       type()
breakpoint()     filter()      iter()          property()    vars()
bytearray()      float()       len()           range()       zip()
bytes()          format()      list()          repr()        __import__()
callable()       frozenset()   locals()        reversed()
chr()            getattr()     map()           round()
classmethod()    globals()     max()           set()
compile()        hasattr()     memoryview()    setattr()
```

APÊNDICE B

Editores de texto e IDEs

Os programadores passam um bom tempo escrevendo, lendo e editando código, e usar um editor de texto ou um IDE (ambiente de desenvolvimento integrado) para que nosso trabalho seja o mais eficiente possível é essencial. Um bom editor fará tarefas simples, como realçar a estrutura do código para que possamos identificar bugs comuns enquanto programamos, não ao ponto de distrai-lo de seus pensamentos. Além do mais, os editores tem funcionalidade úteis, como indentação automática, marcadores para mostrar o comprimento adequado da linha e atalhos de teclado para operações comuns.

Um *IDE* é um editor de texto com uma série de outras ferramentas incluídas, como depuradores interativos e introspecção de código. Um IDE examina o código à medida que você o escreve e tenta aprender sobre o projeto que está criando. Por exemplo, quando começamos a digitar o nome de uma função, um IDE pode mostrar todos os argumentos que a função aceita. Esse comportamento pode ajudar muito quando tudo funciona e entendemos o que estamos vendo. No entanto, pode ser assustador para um iniciante e de difícil resolução de problemas quando não temos certeza do porquê nosso código não funciona em um IDE.

Hoje em dia, as linhas entre editores de texto e IDEs são tênues. Os editores mais populares têm algumas funcionalidades que costumavam ser exclusivas dos IDEs. Da mesma forma, a maioria dos IDEs pode ser configurada para ser executada em um modo com mais desempenho, que nos distrai menos enquanto

programamos, mesmo possibilitando que usemos as funcionalidades mais avançadas quando precisarmos.

Se já tem um editor ou IDE instalado que goste, e caso já esteja configurado para funcionar com uma versão recente do Python em seu sistema, incentivo que fique com aquele que você já sabe usar. Por mais que seja divertido explorar editores diferentes, é uma forma de evitar o trabalhar de aprender uma linguagem nova de programação.

Caso não tenha um editor ou IDE instalado, recomendo o VS Code por vários motivos:

- É gratuito e tem licença open source.
- Pode ser instalado em todos os principais sistemas operacionais.
- Embora seja amigável para iniciantes, é eficiente o bastante ao ponto de muitos programadores profissionais o usarem como editor principal.
- O VS Code encontra as versões do Python instaladas e normalmente não exige nenhuma configuração para executar os primeiros programas.
- Disponibiliza um terminal integrado, ou seja, a saída aparece na mesma janela que o código.
- Disponibiliza uma extensão Python que torna o editor altamente eficiente para escrever e manter o código Python.
- Bastante personalizável, é possível ajustá-lo para combinar com o jeito como programamos.

Neste apêndice, aprenderemos como começar a configurar o VS Code para que possamos usá-lo. Aprenderemos também alguns atalhos que possibilitam programar com mais eficiência. Em programação, digitar rápido não é tão importante como algumas pessoas pensam, agora entender um editor de texto e saber como usá-lo efetivamente ajuda bastante.

Dito isso, não é todo mundo que consegue usar o VS Code. Por algum motivo, caso o VS Code não funcione bem em seu sistema ou

passe a distrai-lo enquanto programa, recorra a outros editores que possam ser interessantes. Este apêndice inclui uma breve descrição de alguns outros editores e IDEs que talvez possam lhe interessar.

Programando de maneira eficiente com o VS Code

No Capítulo 1, instalamos o VS Code e também adicionamos nele uma extensão Python. Esta seção mostrará algumas configurações adicionais que podemos fazer, além de atalhos para trabalhar efetivamente com o código.

Configurando o VS Code

É possível alterar as configurações padrão do VS Code de algumas formas. Podemos efetuar algumas alterações por meio da interface e outras têm que ser feitas nos arquivos de configuração. Às vezes, essas alterações afetarão tudo o que fizermos no VS Code, ao passo que outras afetarão apenas os arquivos dentro da pasta com o arquivo de configuração.

Por exemplo, se tiver um arquivo de configuração em sua pasta *python_work*, essas configurações afetarão somente os arquivos nessa pasta (e suas subpastas). É um bom recurso, porque significa que podemos ter configurações específicas do projeto que substituem nossas configurações globais.

Usando tabulação e espaços

Caso use uma mistura de tabulações e espaços em seu código, seus programas podem apresentar problemas difíceis de identificar. Ao trabalhar em um arquivo *.py* com a extensão Python instalada, o VS Code é configurado para inserir quatro espaços sempre que pressionarmos a tecla TAB. Se estiver somente escrevendo código e tiver a extensão Python instalada, provavelmente nunca terá problemas com tabulações e espaços.

No entanto, talvez a instalação de seu VS Code não esteja corretamente configurada. Além do mais, em algum momento, você pode acabar trabalhando em um arquivo que tem apenas tabulações ou uma mistura de tabulações e espaços. Caso suspeite de qualquer problema com tabulações e espaços, veja a barra de status na parte inferior da janela do VS Code e clique em **Espaços** ou **Alterar Tamanho de Exibição da Guia**. Será exibido um menu suspenso que permite alternar entre o uso de tabulações e o uso de espaços. Podemos alterar também o nível de indentação padrão e converter todas as indentações no arquivo em tabuações ou espaços.

Se estiver analisando algum código e não tiver certeza se a indentação consiste em tabulações ou espaços, realce várias linhas do código. Isso fará com que os caracteres de espaço em branco invisíveis fiquem visíveis. Cada espaço aparecerá como um ponto e cada tabulação aparecerá como uma seta.

NOTA *Em programação, os espaços são preferidos às tabulações porque podem ser interpretados sem equívocos por todas as ferramentas que trabalham com um arquivo de código. A largura das tabulações pode ser interpretada de forma diferente por ferramentas distintas, resultando em erros que podem ser extremamente difíceis de identificar.*

Alterando o tema de cores

O VS Code usa um tema escuro por padrão. Caso queira alterá-lo, clique em **Arquivo** (**Código** na barra de menu no macOS), clique em **Preferências** e escolha **Tema de Cores**. Uma lista suspensa aparecerá e permitirá que você escolha o tema que achar melhor.

Configurando o indicados de comprimento de linha

A maioria dos editores possibilita configurarmos uma sugestão visual, geralmente uma linha vertical, para mostrar onde as linhas devem terminar. Na comunidade Python, a convenção é restringir as linhas a 79 caracteres ou menos.

Para definir esse recurso, clique em **Arquivos**, depois, em **Preferências** e, em seguida, escolha **Configurações**. Na caixa de diálogo exibida, digite **rulers**. Você verá uma configuração para o Editor: Rulers; clique no link intitulado *Editar em settings.json*. No arquivo exibido, adicione o seguinte à configuração editor.rulers:

settings.json

```
"editor.rulers": [
80,
]
```

Esse código adicionará uma linha vertical na janela de edição na posição de 80 caracteres. É possível ter mais de uma linha vertical; por exemplo, se quiser uma linha adicional com 120 caracteres, o valor para sua configuração seria [80, 120]. Se não vir as linhas verticais, não se esqueça de salvar o arquivo settings; talvez seja necessário sair e reabrir o VS Code para que as alterações ocorram em alguns sistemas.

Simplificando a saída

Por padrão, o VS Code mostra a saída de seus programas em uma janela de terminal incorporada. Essa saída inclui os comandos que estão sendo usados para executar o arquivo. Em muitas situações, é o ideal, mas pode causar mais distrações do que você quer, sobretudo quando está aprendendo Python.

Para simplificar a saída, feche todas as guias abertas e saia do VS Code. Inicie o VS Code novamente e abra a pasta com os arquivos Python em que está trabalhando; pode ser apenas a pasta *python_work* onde *hello_world.py* é salvo.

Clique no ícone Executar/Depurar (parecido com um triângulo com um pequeno inseto) e clique em **crie um arquivo** *launch.json*. No menu suspenso, digite Python entre as opções exibidas. No arquivo *launch.json* que é aberto, faça a seguinte alteração:

launch.json

```
{
    -- trecho de código omitido --
    "configurations": [
        {
            -- trecho de código omitido --
            "console": "internalConsole",
            "justMyCode": true
        }
    ]
}
```

Aqui, estamos alterando a configuração do console IntegratedTerminal para internalConsole. Após salvar o arquivo settings, abra um arquivo *.py* como *hello_world.py* e execute-o pressionando CTRL+F5. No painel de saída do VS Code, clique em **Console de Depuração** se ainda não estiver selecionado. Você deve ver apenas a saída do programa, e a saída deve ser atualizada sempre que executar um programa.

NOTA *O Console de Depuração é somente leitura. Não funciona com arquivos que usam a função* input()*, utilizada no Capítulo 7. Quando precisar executar esses programas, é possível alterar a configuração de* console *de volta para* IntegratedTerminal *ou executar esses programas em uma janela de terminal separada, conforme descrito em "Executando programas Python em um terminal" na página* 43*.*

Explorando mais personalizações

Podemos personalizar o VS Code ainda mais e de diversas maneiras para programarmos com eficiência. Para começar a explorar as personalizações disponíveis, clique em Arquivos, depois em **Preferências** e, em seguida, **Configurações**. Você verá uma lista chamada Comumente Usado; clique em qualquer um dos subtítulos para ver algumas maneiras comuns de modificar sua instalação do VS Code. Reserve um tempo para explorar as melhores configurações do VS Code que possam ajudá-lo, mas não a ponto de postergar seu aprendizado Python!

Atalhos do VS Code

Todos os editores e IDEs viabilizam maneiras eficientes de executar tarefas comuns que todos precisam realizar ao escrever e manter o código. Por exemplo, é possível indentar facilmente uma única linha de código ou um bloco inteiro; podemos deslocar um bloco de linhas para cima ou para baixo em um arquivo.

Como o VS Code tem inúmeros atalhos, não será possível abordarmos todos aqui. Esta seção compartilhará somente alguns que você provavelmente achará úteis à medida que escreve os primeiros arquivos Python. Caso opte por um editor diferente do VS Code, faça questão de aprender essas mesmas tarefas de forma eficiente no editor que escolher.

Indentando e removendo a indentação de blocos de código

Para indentar um bloco inteiro de código, realce-o e pressione CTRL+], ou ⌘+] no macOS. Para remover a indentação de um bloco de código, realce-o e pressione CTRL+[, ou ⌘+[no macOS.

Comentando blocos de código

Para desativar temporariamente um bloco de código, é possível realça-lo e comentá-lo para que o Python o ignore. Realce a seção de código que deseja ignorar e pressione CTRL+/ ou ⌘+/ no macOS. As linhas selecionadas serão comentadas com uma cerquilha (#), indentada no mesmo nível da linha de código, sinalizando que não são comentários regulares. Quando quiser descomentar o bloco de código, realce o bloco e execute o mesmo comando.

Deslocando linhas para cima e para baixo

À medida que seus programas ficam mais complexos, talvez você queira deslocar um bloco de código para cima ou para baixo em um

arquivo. Para tal, realce o código que quer mover e pressione ALT+seta para cima ou Option-seta para cima no macOS. A mesma combinação de teclas com a seta para baixo deslocará o bloco para baixo no arquivo.

Caso esteja deslocando uma única linha para cima ou para baixo, você pode clicar em qualquer lugar nessa linha; não precisa realçar toda a linha para deslocá-la.

Ocultando o explorador de arquivos

O explorador de arquivos integrado do VS Code é bastante conveniente. No entanto, pode ser uma distração quando você está programando e pode ocupar um espaço valioso em uma tela menor. O comando CTRL+B, ou ⌘+B no macOS, alterna a visibilidade do painel do explorador de arquivos.

Encontrando atalhos adicionais

Trabalhar com eficiência em um ambiente de um editor de texto exige prática, e também vontade. Quando estiver aprendendo a programar, preste atenção às coisas que você faz repetidamente. Qualquer ação que executar em seu editor provavelmente tem um atalho; se estiver clicando em itens de menu para realizar tarefas de edição, procure os atalhos para essas ações. Se estiver alternando entre o teclado e o mouse com frequência, procure os atalhos de navegação a fim de não usar o mouse com tanta frequência.

É possível conferir todos os atalhos de teclado no VS Code clicando em **Arquivo**, depois em **Preferências** e, em seguida, em **Atalhos de Teclado**. É possível usar a barra de pesquisa para encontrar um atalho específico ou você pode percorrer a lista para encontrar atalhos que possam ajudá-lo a programar com mais eficiência.

Lembre-se, é melhor focar o código em que está trabalhando e evitar passar muito tempo com as ferramentas que está usando.

Outros editores de texto e IDEs

Você ouvirá e verá pessoas usando outros editores de texto. A maioria deles pode ser configurada e personalizada para ajudá-lo da mesma forma que o VS Code. A seguir, mostro uma breve seleção de editores de texto que talvez você ouça falar.

IDLE

O *IDLE* é um editor de texto incluído no Python. É um pouco menos intuitivo de programar do que outros editores mais modernos. No entanto, você verá o IDLE em outros minicursos para iniciantes e talvez queira testá-lo.

Geany

O *Geany* é um editor de texto simples que exibe a saída em uma janela de terminal separada, o que nos ajuda a nos sentir à vontade usando terminais. Embora o Geany tenha uma interface bastante minimalista, é poderosa o suficiente para que um número significativo de programadores experientes ainda o use.

Se achar que o VS Code tem muita distração e muitas funcionalidade, considere o Geany.

Sublime Text

O *Sublime Text* é outro editor minimalista que você deve considerar usar se achar o VS Code poluído visualmente. O Sublime Text tem uma interface mais limpa e é conhecido por funcionar perfeitamente mesmo com arquivos muito grandes. É um editor que não atrapalhará e possibilitará que você se concentre no código que está escrevendo.

O Sublime Text tem uma avaliação gratuita ilimitada, mas não é gratuito ou open source. Caso decida usá-lo e goste, e possa se dar ao luxo de comprar uma licença completa, compre-o. Você compra somente uma vez, não é uma assinatura de software.

Emacs and Vim

O *Emacs* e o *Vim* são dois editores populares, preferidos por muitos programadores experientes, já que são arquitetados para que possamos usá-los sem tirar as mãos do teclado. Uma vez que você aprende como esses editores funcionam, a escrita, a legibilidade e a modificação do código se tornam mais eficientes. Significa também que ambos editores têm uma curva de aprendizado bastante acentuada. O Vim está incluído na maiorias das máquinas com Linux ou macOS, e tanto o Emacs como o Vim podem ser executados em um terminal. Por essa razão, costumam ser usados para escrever código em servidores por meio de sessões de terminal remoto.

Em geral, os programadores experientes recomendam testá-los, porém muitos deles se esquecem da quantidade de conteúdo que os programadores iniciantes já estão tentando aprender. É bom conhecer esses editores, mas lembre-se: só aprenda a usá-los quando se sentir à vontade programando em um editor mais amigável, que possibilite que você foque aprender a programar e não aprender a usar um editor.

PyCharm

O *PyCharm* é um IDE popular entre os programadores Python porque foi desenvolvido especificamente para trabalhar com Python. A versão completa exige assinatura paga, mas uma versão gratuita chamada PyCharm Community Edition também está disponível, e muitos desenvolvedores a consideram útil.

Caso teste o PyCharm, fique sabendo que, por padrão, esse editor define um ambiente isolado para cada um dos projetos. Isso geralmente é bom, mas pode levar a um comportamento inesperado, caso você não entenda o que o editor está fazendo.

Jupyter Notebooks

O *Jupyter Notebook* é um tipo de ferramenta diferente dos editores

de texto ou IDEs tradicionais, pois é uma aplicação web desenvolvida principalmente por blocos; cada bloco é um bloco de código ou um bloco de texto. Como os blocos de texto são renderizados em Markdown, pode-se incluir formatação simples.

Os Jupyter Notebooks foram desenvolvidos para aplicações científicas com o Python, mas desde então é usado produtivamente em uma ampla variedade de situações. Em vez de apenas escrever comentários dentro de um arquivo *.py*, é possível escrever texto não criptografado com formatação simples, como cabeçalhos, listas com marcadores e hiperlinks entre as seções do código. Cada bloco de código pode ser executado de forma independente, possibilitando testar pequenos pedaços do programa ou podemos executar todos os blocos de código de uma só vez. Cada bloco de código tem a própria área de saída, sendo possível ativar ou desativar as áreas de saída conforme necessário.

Não raro, os Jupyter Notebooks podem ser confusos por causa das interações entre diferentes células. Se definir uma função em uma célula, essa função também estará disponível para outras células. Na maioria das vezes, isso é vantajoso, mas pode ficar confuso em notebooks mais extensos e caso você não entenda totalmente como o ambiente Notebook funciona.

Se estiver desenvolvendo alguma tarefa científica ou seu foco for dados com Python, certamente verá os Jupyter Notebooks em algum momento.

APÊNDICE C
Obtendo ajuda

Em algum momento, todo mundo fica perdido quando está aprendendo a programar. Dito isso, uma das habilidades fundamentais como programador é saber lidar eficientemente com situações problemáticas. Este apêndice apresenta diversas formas de ajudá-lo a recomeçar quando você se sentir perdido com seus códigos.

Primeiros passos

Quando ficar sem saber o que fazer, o primeiro passo é avaliar sua situação. Antes de pedir ajuda a qualquer outra pessoa, responda claramente às três perguntas:

- O que você está tentando fazer?
- O que já tentou fazer até agora?
- Quais resultados tem obtido?

Responda da forma mais específica possível. Para a primeira pergunta, respostas explícitas como "Estou tentando instalar a versão mais recente do Python no meu novo notebook Windows" tem detalhes suficientes para que outras pessoas da comunidade Python o ajudem. Respostas como "Estou tentando instalar o Python" não fornecem informações suficientes para que outras pessoas ofereçam ajuda.

A resposta à segunda pergunta deve fornecer detalhes suficientes para que você não seja aconselhado a repetir o que já tentou fazer: "Acessei https://python.org/downloads e cliquei no botão **Download** do

meu sistema. Depois, executei o instalador" é mais útil do que "Acessei ao site do Python e fiz o download de alguma coisa".

Para a terceira pergunta, ajuda saber as mensagens exatas de erro que recebeu. Assim, você consegue pesquisar uma solução ou fornecê-las quando pedir ajuda.

Às vezes, o fato de responder a essas três perguntas antes de pedir ajuda possibilita que você identifique algo que deixou passar sem precisar recorrer a outros recursos. Os programadores até têm um nome para isso: *debug com patinho de borracha*. A ideia é que, caso explique claramente sua situação para um patinho de borracha (ou para qualquer objeto inanimado) e faça uma pergunta específica, pode acontecer de você responder à própria pergunta. Algumas equipes de programação até têm um patinho de borracha para incentivar as pessoas a "falarem com o pato".

Tente outra vez

Recomeçar tudo do zero e tentar outra vez pode ser o bastante para solucionar muitos problemas. Digamos que você esteja tentando escrever um loop `for` com base em um exemplo deste livro. Talvez você tenha se esquecido de algo simples, como os dois-pontos no final da linha `for`. Recomeçar todos os passos pode ajudá-lo a evitar o mesmo tipo de erro de novo.

Faça uma pausa

Caso esteja martelando o mesmo problema há algum tempo, fazer uma pausa é uma das melhores táticas. Quando ficamos empacados na mesma tarefa por longos períodos de tempo, nosso cérebro começa a focar apenas em uma solução. Como perdemos o foco de nossas suposições, fazer uma pausa nos ajuda a enxergar o problema sob uma nova perspectiva. Não é necessário uma pausa longa: basta fazer algo que não seja raciocinar sobre o problema. Se está sentado há muito tempo, recorra à atividade física: caminhe ou saia um pouco, talvez beber um copo de água ou comer um lanche

saudável.

Caso esteja ficando frustrado, talvez seja melhor recomeçar no outro dia. Uma boa noite de sono quase sempre ajuda com a solução de problemas.

Consulte os recursos deste livro

Os recursos online deste livro, disponíveis em https://ehmatthes.github.io/pcc_3e, incluem várias seções úteis sobre como configurar seu sistema e estudar cada capítulo. Caso não tenha feito isso ainda, confira esses recursos e veja se há algo que o ajude a resolver sua situação.

Pesquisando online

Ao pesquisar online, a probabilidade de que outra pessoa tenha o mesmo problema que você e tenha pedido ajuda online é grande. Boas habilidades de pesquisa e perguntas específicas o ajudam a encontrar recursos existentes para resolver o problema que está enfrentando. Por exemplo, caso esteja tendo dificuldades para instalar a versão mais recente do Python em um novo sistema Windows, pesquisar *instalar o Python no Windows* e limitar os resultados aos recursos do ano passado pode direcioná-lo para uma resposta clara.

Pesquisar a mensagem de erro exata também pode ajudar bastante. Por exemplo, digamos que você obtenha o seguinte erro ao tentar executar um programa Python em um terminal de um novo sistema Windows:

```
> python hello_world.py
Python was not found; run without arguments to install from the Microsoft
  Store...
```

Pesquisar a frase completa, "Python was not found; run without arguments to install from the Microsoft Store", provavelmente renderá algumas orientações úteis.

Quando começamos a pesquisar tópicos relacionados à programação, alguns sites aparecerão repetidas vezes. Abordarei alguns desses sites brevemente, para que você saiba como podem ser úteis e ajudá-lo.

Stack Overflow

O *Stack Overflow* (https://stackoverflow.com) é um dos sites de perguntas e respostas mais populares para programadores e, não raro, aparece na primeira página de resultados em pesquisas relacionadas ao Python. Os participantes postam perguntas quando estão com problemas, e outros participantes tentam ajudar com respostas. Como os usuários podem votar nas respostas que consideram mais úteis, as melhores respostas normalmente são as primeiras que você verá.

No Stack Overflow, muitas perguntas básicas sobre Python apresentam respostas claras, já que foram refinadas pela comunidade ao longo do tempo. Os usuários também são incentivados a postar atualizações, ou seja, as respostas costumam ser relativamente atualizadas. No momento em que eu escrevia este livro, quase dois milhões de perguntas relacionadas ao Python foram respondidas no Stack Overflow.

No entanto, espera-se que você saiba algumas coisas antes de postar no Stack Overflow. As perguntas devem ser breves e incluir o tipo de problema que você está enfrentando. Se você postar as linhas do código (de 5 a 20 linhas) que geram o erro que está enfrentando e se recorrer às orientações da seção "*Primeiros passos*" deste apêndice, na página *583*, provavelmente alguém o ajudará. Agora, caso compartilhe um link para um projeto com diversos arquivos grandes, as pessoas provavelmente não ajudarão. Confira este ótimo guia para elaborar uma boa pergunta, acesse https://stackoverflow.com/help/how-to-ask. É possível usar as sugestões deste guia para obter ajuda em qualquer comunidade de programadores.

Documentação oficial do Python

A documentação oficial do Python (https://docs.python.org) é um pouco confusa para iniciantes, já que o objetivo é mais documentar a linguagem do que fornecer explicações. Na documentação oficial, os exemplos até funcionam, mas você pode não entender tudo o que é mostrado. Ainda assim, trata-se de um bom recurso para explorar quando aparecer em suas pesquisas. À medida que você compreende cada vez mais o Python, a documentação se tornará mais útil.

Documentação oficial da biblioteca

Se estiver usando uma biblioteca específica, como Pygame, Matplotlib ou Django, os links da documentação oficial geralmente aparecerão nas pesquisas. Por exemplo, https://docs.djangoproject.com é muito útil quando se trabalha com Django. Caso planeje trabalhar com qualquer uma dessas bibliotecas, é boa ideia se familiarizar com a documentação oficial.

r/learnpython

O Reddit é composto por vários subfóruns chamados *subreddits*. O subreddit *r/learnpython* (https://reddit.com/r/learnpython) é muito ativo e acolhedor. É possível ler as perguntas dos outros e postar as suas também. Não raro, você obterá diversas perspectivas sobre as questões que levantar, o que pode ajudar bastante a desenvolver um entendimento mais aprofundado do tópico com o qual está trabalhando.

Postagens em blogs

Muitos programadores têm blogs para compartilharem postagens sobre as linguagens com as quais estão trabalhando. Procure a data nas postagens do blog que encontrar, assim é possível ver como as informações podem ser aplicáveis à versão do Python que você está usando.

Discord

O *Discord* é um ambiente de bate-papo online com comunidades Python onde você pode pedir ajuda e seguir discussões relacionadas ao Python.

Para conferir, acesse https://pythondiscord.com e clique no link do **Discord** no canto superior direito. Se já tiver uma conta no Discord, poderá fazer login com sua conta existente. Se não tiver uma conta, digite um nome de usuário e siga as instruções para concluir seu registro no Discord.

Caso seja a primeira vez em que visita o Discord do Python, é necessário aceitar as regras da comunidade antes de participar totalmente. Depois disso, você pode participar de qualquer um dos canais que lhe interessam. Se estiver procurando por ajuda, lembre-se de postar em um dos canais de ajuda do Python.

Slack

O Slack é outro ambiente de bate-papo online. É frequentemente usado para comunicações corporativas internas, mas também há muitos grupos públicos que você pode participar. Se quiser conferir os grupos do Python no Slack, comece com https://pyslackers.com. Clique no link **Slack** na parte superior da página e digite seu endereço de e-mail para receber um convite.

Quando você estiver no workspace do Python Developers, verá uma lista de canais. Clique em **Canais** e escolha os tópicos que lhe interessam. Talvez você queira conferir os canais *#help* e *#django*.

APÊNDICE D

Usando o Git para controle de versões

O software de controle de versão possibilita tirar snapshots de um projeto sempre que ele estiver em um estado de atividade. Ao alterarmos um projeto – por exemplo, quando implementamos uma funcionalidade nova – podemos voltar a um estado de trabalho anterior se o estado atual do projeto não estiver funcionando bem.

Usar o software de controle de versão proporciona a liberdade de trabalhar em melhorias e cometer erros sem nos preocuparmos em prejudicar todo o projeto. Isso vale criticamente para projetos grandes, mas também pode ser útil em projetos menores, mesmo quando estamos trabalhando com programas em um único arquivo.

Neste apêndice, aprenderemos a instalar o Git e a usá-lo para controle de versão nos programas em que estamos trabalhando agora. Atualmente, o *Git* é o software de controle de versão mais popular em uso. Muitas de suas ferramentas sofisticadas ajudam as equipes a colaborar em grandes projetos, no entanto, seus recursos mais básicos também funcionam bem com um desenvolvedor só. O Git implementa o controle de versão rastreando as alterações feitas em todos os arquivos de um projeto; se cometer um erro, poderá retornar a um estado salvo anteriormente.

Instalando o Git

O Git funciona em todos os sistemas operacionais, mas existem diferentes abordagens para instalá-lo em cada um deles. As seções a

seguir fornecem instruções específicas para cada sistema operacional.

O Git é incluído em alguns sistemas por padrão e geralmente vem junto com outros pacotes que você já pode ter instalado. Antes de tentar instalá-lo, veja se já tem o Git no sistema. Abra uma janela nova de terminal e execute o comando **git --version**. Se vir a saída listando um número de versão específico, o Git será instalado em seu sistema. Se vir uma mensagem solicitando que instale ou atualize o Git, siga as instruções na tela.

Se não vir nenhuma instrução na tela e estiver usando o Windows ou o macOS, poderá fazer o download do instalador em https://git-scm.com. Caso seja usuário Linux com um sistema compatível APT, pode instalar o Git com o comando **sudo apt install git**.

Configurando o Git

O Git acompanha quem efetua alterações em um projeto, mesmo quando o projeto tem somente uma pessoa. Para fazer isso, o Git precisa saber seu nome de usuário e e-mail. É necessário fornecer um nome de usuário, mas você pode criar um endereço de e-mail falso:

```
$ git config --global user.name "username"
$ git config --global user.email "username@example.com"
```

Caso se esqueça essa etapa, o Git solicitará essas informações quando fizer seu primeiro commit.

Além do mais, é melhor definir o nome padrão para o branch principal em cada projeto. Um nome bom para esse branch é main:

```
$ git config --global init.defaultBranch main
```

Quando fazemos essa configuração, cada projeto novo, em que usamos o Git para gerenciar, começará com um único branch de commits chamado *main*.

Criando um projeto

Criaremos um projeto com o qual trabalhar. Crie uma pasta em algum lugar do seu sistema chamada *git_practice*. Dentro dela, crie um programa Python simples:

hello_git.py

```
print("Hello Git world!")
```

Usaremos esse programa para explorar as funcionalidades básicas do Git.

Ignorando arquivos

Os arquivos com a extensão *.pyc* são gerados automaticamente a partir de arquivos *.py*, ou seja, não é necessário usar o Git para rastreá-los. Esses arquivos são armazenados em um diretório chamado *__pycache__*. Para instruir o Git a ignorar esse diretório, crie um arquivo especial chamado *.gitignore* – com um ponto no início do nome do arquivo e sem extensão de arquivo – e adicione a seguinte linha:

.gitignore

```
__pycache__/
```

Esse arquivo instrui o Git a ignorar qualquer arquivo no diretório *__pycache__*. Usar um arquivo *.gitignore* fará com que seu projeto fique organizado e mais fácil de trabalhar.

Talvez seja necessário modificar as configurações do navegador de arquivos para que arquivos ocultos (arquivos cujos nomes começam com um ponto) sejam exibidos. No Explorador de Arquivos, clique no menu Exibir e, em seguida, marque a caixa **Itens ocultos**. No macOS, pressione ⌘+SHIFT+.(ponto). No Linux, procure uma configuração chamada Mostrar Arquivos Ocultos.

NOTA *Se estiver no macOS, adicione mais uma linha ao* .gitignore*. Adicione o nome* .DS_Store*; são arquivos ocultos que contêm informações sobre cada diretório no macOS, e desorganizam seu projeto caso não os adicionem ao* .gitignore*.*

Inicializando um repositório

Agora que temos um diretório com um arquivo Python e um arquivo *.gitignore*, podemos inicializar um repositório Git. Abra um terminal, navegue até a pasta *git_practice* e execute o seguinte comando:

```
git_practice$ git init
Initialized empty Git repository in git_practice/.git/
git_practice$
```

A saída mostra que o Git inicializou um repositório vazio no *git_practice*. Um *repositório* é o conjunto de arquivos em um programa que o Git está ativamente rastreando. Todos os arquivos que o Git usa para gerenciar o repositório estão localizados no diretório oculto *.git*, com o qual não precisaremos lidar. Basta não excluir esse diretório, ou você perderá o histórico do seu projeto.

Verificando o status

Antes de fazer qualquer outra coisa, verificaremos o status do projeto:

```
git_practice$ git status
1 On branch main
  No commits yet

2 Untracked files:
    (use "git add <file>..." to include in what will be committed)
        .gitignore
        hello_git.py

3 nothing added to commit but untracked files present (use "git add" to track)
  git_practice$
```

No Git, um *branch* é uma versão do projeto em que estamos trabalhando; aqui, pode ver que estamos em um branch chamado main 1. Sempre que verificarmos o status do projeto, devemos ver o branch main. Pois assim, podemos fazer o commit inicial. Um *commit* é um snapshot do projeto em um determinado momento.

O Git nos informa que arquivos não rastreados estão no projeto 2, pois ainda não informamos quais arquivos rastrear. Em seguida,

somos informados de que não há nada adicionado ao commit atual, mas há arquivos não rastreados que podemos querer adicionar ao repositório 3.

Adicionando arquivos ao repositório

Vamos adicionar dois arquivos ao repositório e verificar o status mais uma vez:

```
1 git_practice$ git add .
2 git_practice$ git status
  On branch main
  No commits yet

  Changes to be committed:
    (use "git rm --cached <file>..." to unstage)
3       new file:   .gitignore
        new file:   hello_git.py

  git_practice$
```

O comando `git add .` adiciona ao repositório todos os arquivos dentro de um projeto que ainda não estão sendo rastreados 1, desde que não estejam listados no *.gitignore*. Esse comando não faz o commit dos arquivos; apenas solicita que o Git comece a prestar atenção neles. Quando verificamos o status do projeto agora, podemos ver que o Git reconhece algumas mudanças que precisam de commit 2. O rótulo *new file* significa que esses arquivos foram adicionados recentemente ao repositório 3.

Fazendo um commit

Faremos o primeiro commit:

```
1 git_practice$ git commit -m "Started project."
2 [main (root-commit) cea13dd] Started project.
3 2 files changed, 5 insertions(+)
  create mode 100644 .gitignore
  create mode 100644 hello_git.py
4 git_practice$ git status
  On branch main
```

```
nothing to commit, working tree clean
git_practice$
```

Executamos o comando git commit -m "*mensagem*" 1 para criarmos um snapshot do projeto. A flag -m instrui o Git a gravar a mensagem que se segue (Started project.) no log do projeto. A saída mostra que estamos no branch main 2 e que dois arquivos foram alterados 3.

Quando verificamos o status agora, podemos ver que estamos no branch main e temos um diretório limpo de trabalho 4. É a mensagem que você deve ver sempre que efetuar o commit de um estado de trabalho do projeto. Se receber uma mensagem diferente, leia-a com atenção; é provável que você tenha esquecido de adicionar um arquivo antes de fazer um commit.

Verificando o log

O Git mantém um log de todos os commits realizados no projeto. Vamos verificar o log:

```
git_practice$ git log
commit cea13ddc51b885d05a410201a54faf20e0d2e246 (HEAD -> main)
Author: eric <eric@example.com>
Date:   Mon Jun 6 19:37:26 2022 -0800

    Started project.
git_practice$
```

Sempre que fazemos um commit, o Git gera um ID de referência exclusivo de 40 caracteres. Além disso, registra quem fez o commit, quando foi realizado e a mensagem registrada. Como nem sempre precisamos de todas essas informações, o Git oferece a opção para exibir uma versão mais simples das entradas de log:

```
git_practice$ git log --pretty=oneline
cea13ddc51b885d05a410201a54faf20e0d2e246 (HEAD -> main) Started project.
git_practice$
```

A flag --pretty=oneline fornece as duas informações mais importantes: o ID de referência do commit e a mensagem gravada para o commit.

Segundo commit

Para conferirmos como o controle de versão é efetivo, precisamos fazer uma mudança no projeto e comitar essa mudança. Aqui, vamos apenas adicionar outra linha ao *hello_git.py*:

hello_git.py

```
print("Hello Git world!")
print("Hello everyone.")
```

Quando verificamos o status do projeto, veremos que o Git identificou o arquivo alterado:

```
git_practice$ git status
1 On branch main
  Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)

2 modified:   hello_git.py

3 no changes added to commit (use "git add" and/or "git commit -a")
  git_practice$
```

Vemos o branch em que estamos trabalhando 1, o nome do arquivo modificado 2 e que nenhuma alteração foi comitada 3. Faremos o commit dessa alteração e verificaremos o status mais uma vez:

```
1 git_practice$ git commit -am "Extended greeting."
  [main 945fa13] Extended greeting.
  1 file changed, 1 insertion(+), 1 deletion(-)
2 git_practice$ git status
  On branch main
  nothing to commit, working tree clean
3 git_practice$ git log --pretty=oneline
  945fa13af128a266d0114eebb7a3276f7d58ecd2 (HEAD -> main) Extended greeting.
  cea13ddc51b885d05a410201a54faf20e0d2e246 Started project.
  git_practice$
```

Fizemos um commit novo, passando as flags `-am` quando usamos o comando `git commit` 1. A flag `-a` instrui o Git a adicionar todos os arquivos modificados no repositório ao commit atual. (Caso crie arquivos novos entre commits, execute novamente o comando `git add .` para inclui-los no repositório.) A flag `-m` instrui o Git a gravar

uma mensagem no log para esse commit.

Ao verificarmos o status do projeto, vemos que mais uma vez temos um diretório limpo de trabalho 2. Finalmente, vemos os dois commits no log 3.

Revertendo alterações

Agora, veremos como descartar uma alteração e retornar ao estado de trabalho anterior. Primeiro, adicione uma linha nova a *hello_git.py*:

hello_git.py

```
print("Hello Git world!")
print("Hello everyone.")

print("Oh no, I broke the project!")
```

Salve e execute esse arquivo.

Verificamos o status e vemos que o Git percebe essa alteração:

```
git_practice$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)

1     modified:   hello_git.py

no changes added to commit (use "git add" and/or "git commit -a")
git_practice$
```

O Git sabe que modificamos *hello_git.py* 1 e podemos efetuar o commit da alteração se quisermos. Mas desta vez, em vez de fazer o commit da alteração, voltaremos ao último commit quando nosso projeto estava funcionando. Não faremos nada em *hello_git.py*: não excluiremos a linha nem usaremos o recurso Undo no editor de texto. Em vez disso, insira os seguintes comandos na sessão do terminal:

```
git_practice$ git restore .
git_practice$ git status
```

```
On branch main
nothing to commit, working tree clean
git_practice$
```

O comando git restore *nome_arquivo* possibilita descartar todas as alterações desde o último commit em um arquivo específico. O comando git restore . descarta todas as alterações feitas em todos os arquivos desde o último commit; essa ação reverte o projeto para o último estado de commit.

Quando retornamos ao editor de texto, veremos que *hello_git.py* mudou:

```
print("Hello Git world!")
print("Hello everyone.")
```

Nesse projeto simples, embora retornar a um estado anterior pareça comum, se estivéssemos trabalhando em um projeto grande com dezenas de arquivos modificados, todos os arquivos alterados desde o último commit seriam restaurados. Esse recurso é extremamente útil: é possível fazer quantas alterações quisermos ao implementar uma funcionalidade nova e, caso não funcionarem, poderemos descartá-las sem afetar o projeto. Não é necessário se lembrar dessas alterações e desfazê-las manualmente. O Git cuida de tudo isso.

NOTA *Talvez seja necessário atualizar o arquivo no editor para verificar a versão restaurada.*

Check out de commits anteriores

Podemos rever qualquer commit em nosso log, usando o comando checkout, com os seis primeiros caracteres de um ID de referência. Após verificar e analisar um commit anterior, é possível retornar ao commit mais recente ou descartar um trabalho recente e retomar o desenvolvimento do commit anterior:

```
git_practice$ git log --pretty=oneline
945fa13af128a266d0114eebb7a3276f7d58ecd2 (HEAD -> main) Extended greeting.
cea13ddc51b885d05a410201a54faf20e0d2e246 Started project.
```

```
git_practice$ git checkout cea13d
Note: switching to 'cea13d'.

1 You are in 'detached HEAD' state. You can look around, make experimental
  changes and commit them, and you can discard any commits you make in this
  state without impacting any branches by switching back to a branch.

  If you want to create a new branch to retain commits you create, you may
  do so (now or later) by using -c with the switch command. Example:

    git switch -c <new-branch-name>

2 Or undo this operation with:

    git switch -

  Turn off this advice by setting config variable advice.detachedHead to false

  HEAD is now at cea13d Started project.
  git_practice$
```

Ao fazermos o check out de um commit anterior, saímos do branch main e entramos no local que o Git chama de estado *detached HEAD* 1. *HEAD* é o estado comitado atual do projeto; estamos *desanexados* porque saímos do branch (main, nesse caso).

Para voltar ao branch main, siga a sugestão 2 para desfazer a operação anterior:

```
git_practice$ git switch -
Previous HEAD position was cea13d Started project.
Switched to branch 'main'
git_practice$
```

Esse comando nos faz retornar ao branch main. A menos que você queira trabalhar com alguns recursos mais avançados do Git, é melhor não fazer nenhuma alteração em seu projeto quando fizer check out de um commit anterior. No entanto, caso seja a única pessoa no projeto e queira descartar todos os commits mais recentes e voltar a um estado anterior, poderá reinicializar o projeto em um commit anterior. A partir do branch main, insira o seguinte:

```
1 git_practice$ git status
  On branch main
```

```
  nothing to commit, working directory clean
2 git_practice$ git log --pretty=oneline
  945fa13af128a266d0114eebb7a3276f7d58ecd2 (HEAD -> main) Extended greeting.
  cea13ddc51b885d05a410201a54faf20e0d2e246 Started project.
3 git_practice$ git reset --hard cea13d
  HEAD is now at cea13dd Started project.
4 git_practice$ git status
  On branch main
  nothing to commit, working directory clean
5 git_practice$ git log --pretty=oneline
  cea13ddc51b885d05a410201a54faf20e0d2e246 (HEAD -> main) Started project.
  git_practice$
```

Primeiro, verificamos o status para ter certeza de que estamos no branch `main` `1`. Ao analisarmos o log, vemos ambos os commits `2`. Em seguida, executamos o comando `git reset --hard` com os seis primeiros caracteres do ID de referência do commit para o qual queremos voltar permanentemente `3`. Verificamos o status novamente e vemos que estamos no branch `main` sem nada para comitar `4`. Ao analisarmos o log mais uma vez, vemos que estamos no commit do qual queremos recomeçar `5`.

Excluindo o repositório

Às vezes, o histórico de nosso repositório fica desorganizado e não sabemos como recuperá-lo. Caso isso ocorra, antes de pedir ajuda, recorre às abordagens apresentadas no Apêndice C. Se não for bem-sucedido e estiver trabalhando sozinho em um projeto, é possível trabalhar com os arquivos, excluindo o diretório *.git* para descartar o histórico do projeto. Isso não impactará o estado atual de nenhum dos arquivos, mas excluirá todos os commits. Ou seja, não será possível fazer check out de nenhum outro estado do projeto.

Para isso, abra um navegador de arquivos e exclua o repositório *.git* ou o exclua a partir da linha de comando. Em seguida, precisaremos recomeçar com um repositório novo para rastrear as alterações novamente. Vejamos todo esse processo em uma sessão de terminal:

```
1 git_practice$ git status
  On branch main
  nothing to commit, working directory clean
2 git_practice$ rm -rf .git/
3 git_practice$ git status
  fatal: Not a git repository (or any of the parent directories): .git
4 git_practice$ git init
  Initialized empty Git repository in git_practice/.git/
5 git_practice$ git status
  On branch main
  No commits yet

  Untracked files:
    (use "git add <file>..." to include in what will be committed)
        .gitignore
        hello_git.py

  nothing added to commit but untracked files present (use "git add" to track)
6 git_practice$ git add .
  git_practice$ git commit -m "Starting over."
  [main (root-commit) 14ed9db] Starting over.
  2 files changed, 5 insertions(+)
  create mode 100644 .gitignore
  create mode 100644 hello_git.py
7 git_practice$ git status
  On branch main
  nothing to commit, working tree clean
  git_practice$
```

Primeiro, verificamos o status e vemos que temos um diretório de trabalho limpo 1. Em seguida, usamos o comando rm -rf .git/ para excluir o diretório *.git* (del .git no Windows) 2. Quando verificamos o status após excluir a pasta *.git*, somos informados de que esse não é um repositório Git 3. Todas as informações que o Git usa para rastrear um repositório são armazenadas na pasta *.git*, portanto, removê-la exclui todo o repositório.

Assim, podemos usar o git init para iniciar um repositório novo 4. Verificar o status mostra que retornamos ao estágio inicial, aguardando o primeiro commit 5. Adicionamos os arquivos e fazemos o primeiro commit 6. Agora, quando verificamos o status, vemos um novo branch main, sem nada para comitar 7.

Utilizar o controle de versão exige um pouco de prática, mas uma vez que começar a usá-lo, você nunca mais vai querer trabalhar sem ele.

APÊNDICE E

Solução de problemas para deploy

Conseguir fazer o deploy de uma aplicação é extremamente satisfatório, ainda mais se você nunca fez antes. No entanto, existem muitos obstáculos que podem surgir no processo de deploy e, infelizmente, alguns desses problemas podem ser difíceis de identificar e resolver. Este apêndice o ajudará a entender as abordagens modernas de deploy e fornecerá maneiras específicas de solucionar problemas do processo de implantação quando as coisas não funcionarem direito.

Se as informações adicionais deste apêndice não forem suficientes para ajudá-lo a efetuar o processo de deploy com sucesso, confira os recursos online em https://ehmatthes.github.io/pcc_3e; as atualizações quase certamente o ajudarão a realizar uma implantação bem-sucedida.

Entendendo os deploys

Ao tentarmos solucionar problemas de uma tentativa de deploy específico, é importante entender claramente como um típico deploy funciona. *Deploy* ou implantação é o processo de pegar um projeto que funciona em nosso sistema local e copiá-lo para um servidor remoto, de uma forma que possibilite responder à requisições de qualquer usuário na internet. O ambiente remoto difere de um típico sistema local em vários aspectos importantes: provavelmente não é o mesmo sistema operacional (SO) usado e é mais provável que seja

um dos muitos servidores virtuais em um único servidor físico.

Ao fazer a implantação de um projeto ou o *push* para o servidor remoto, é necessário executar as seguintes etapas:

- Criar um servidor virtual em uma máquina física em um datacenter.
- Estabelecer uma conexão entre o sistema local e o servidor remoto.
- Copiar o código do projeto para o servidor remoto.
- Identificar todas as dependências do projeto e instalá-las no servidor remoto.
- Configurar um banco de dados e executar todas as migrações existentes.
- Copiar arquivos estáticos (CSS, arquivos JavaScript e arquivos de mídia) para um local em que possam ser atendidos com eficiência.
- Inicializar um servidor para lidar com as requisições recebidas.
- Começar a rotear as requisições recebidas para o projeto, assim que estiver pronto para lidar com essas requisições.

Ao pensarmos em tudo necessário para um deploy, não é de se admirar que deploys regularmente falhem. Felizmente, uma vez que entendemos claramente todo o processo, teremos uma probabilidade maior de identificar o que saiu de errado. Se conseguir identificar o que deu errado, talvez consiga identificar uma correção que fará com que a próxima tentativa de implantação seja bem-sucedida.

É possível desenvolver localmente em um tipo de sistema operacional e enviar um projeto para um servidor com um sistema operacional diferente. É fundamental saber para qual tipo de sistema estamos enviando o projeto, pois, assim, é mais fácil solucionar problemas. No momento em que eu escrevia este apêndice, um servidor remoto básico no Platform.sh usava o Debian Linux; a maioria dos servidores remotos funcionam com sistemas Linux.

Solução de problemas básicos

Algumas etapas de solução de problemas são específicas de cada sistema operacional. Em breve, falaremos delas. Primeiro, vejamos as etapas que todos devem executar ao tentar solucionar problemas de deploy.

O melhor recurso é a saída gerada durante a tentativa de push. Talvez essa saída pareça assustadora; caso seja iniciante com deploys de aplicação, tudo pode parecer extremamente técnico e, de fato, há muita tecnicidade. Mas, a boa notícia é que não precisamos entender tudo o que a saída gera. Temos dois objetivos ao examinar a saída de log: identificar as etapas de deploy que funcionaram e as etapas que não funcionaram. Se conseguir fazer isso, poderá identificar o que mudar no projeto ou no processo de implantação para que o próximo push seja bem-sucedido.

Siga as sugestões na tela

Às vezes, a plataforma para a qual estamos enviando um projeto gera mensagens com sugestões claras de como solucionar problemas. Por exemplo, vejamos a mensagem gerada quando criamos um projeto no Platform.sh, antes de inicializarmos um repositório Git e, em seguida, tentarmos fazer o push dele:

```
  $ platform push
1 Enter a number to choose a project:
    [0] ll_project (votohz445ljyg)
  > 0

2   [RootNotFoundException]
    Project root not found. This can only be run from inside a project
      directory.

3  To set the project for this Git repository, run:
      platform project:set-remote [id]
```

Estamos tentando fazer o push de um projeto, mas o projeto local ainda não foi associado a um projeto remoto. Por isso, a CLI do Platform.sh pergunta para qual projeto remoto queremos enviar 1.

Digitamos **0**, para selecionar o único projeto listado. Mas a seguir, vemos uma RootNotFoundException 2. Isso acontece porque o Platform.sh procura um diretório *.git* quando inspeciona o projeto local, a fim de descobrir como conectar o projeto local ao projeto remoto. Nesse caso, como não havia diretório *.git* quando o projeto remoto foi criado, essa conexão nunca foi estabelecida. A CLI sugere uma correção 3; nos informa que podemos especificar o projeto remoto que deve ser associado a esse projeto local, usando o comando project:set-remote.

Vamos testar essa sugestão:

```
$ platform project:set-remote votohz445ljyg
Setting the remote project for this repository to: ll_project (votohz445ljyg)

The remote project for this repository is
    now set to: ll_project (votohz445ljyg)
```

Na saída anterior, a CLI mostrou o ID desse projeto remoto, votohz4451jyg. Assim, executamos o comando sugerido, usando esse ID, e a CLI é capaz de fazer a conexão entre o projeto local e o projeto remoto.

Agora, tentaremos fazer o push desse projeto mais uma vez:

```
$ platform push
Are you sure you want to push to the main (production) branch? [Y/n] y
Pushing HEAD to the existing environment main
-- trecho de código omitido --
```

É tentativa bem-sucedida; seguir a sugestão na tela funcionou.

Seja cuidadoso ao executar comandos que não entende completamente. No entanto, se tiver boas razões para acreditar que um comando pode causar pouco dano e se confiar na fonte da recomendação, é razoável tentar as sugestões oferecidas com as ferramentas que está usando.

NOTA *Lembre-se de que existem indivíduos que lhe dirão para executar comandos que limparão seu sistema ou o exporão à exploração remota. Seguir as sugestões de uma ferramenta fornecida por uma empresa ou organização em que confia é*

diferente de seguir as sugestões de pessoas aleatórias online. Sempre que estiver lidando com conexões remotas, proceda com muita cautela.

Leia a saída do log

Conforme mencionado antes, a saída de log que vemos quando executamos um comando como `platform push` pode ser informativa e um tanto assustadora. Leia o seguinte trecho da saída de log, reproduzido de uma tentativa diferente de usar `platform push`, e veja se consegue detectar o problema:

```
-- trecho de código omitido --
Collecting soupsieve==2.3.2.post1
  Using cached soupsieve-2.3.2.post1-py3-none-any.whl (37 kB)
Collecting sqlparse==0.4.2
  Using cached sqlparse-0.4.2-py3-none-any.whl (42 kB)
Installing collected packages: platformshconfig, sqlparse,...
Successfully installed Django-4.1 asgiref-3.5.2 beautifulsoup4-4.11.1...
W: ERROR: Could not find a version that satisfies the requirement gunicorrn
W: ERROR: No matching distribution found for gunicorrn

130 static files copied to '/app/static'.

Executing pre-flight checks...
-- trecho de código omitido --
```

Quando uma tentativa de deploy falha, uma boa estratégia é examinar a saída de log e verificar se é possível detectar algo que pareça avisos ou erros. Os avisos são bastante comuns; via de regra, são mensagens sobre as próximas mudanças nas dependências de um projeto, que ajudam os desenvolvedores a solucionar problemas antes que causem falhas.

Um push bem-sucedido pode ter avisos, mas não deve ter erros. Nesse caso, o Platform.sh não conseguiu encontrar uma maneira de instalar o requisito `gunicorrn`. Trata-se de um erro de digitação no arquivo *requirements_remote.txt*, que deveria incluir `gunicorn` (com um r). Nem sempre é fácil identificar o problema raiz na saída do log, ainda mais quando provoca uma série de erros e avisos

progressivos. Assim como ao ler um traceback em seu sistema local, é boa ideia examinar minuciosamente os primeiros erros listados e também os últimos. A maioria dos erros intermediários costumam ser pacotes internos acusando que algo deu errado e passando mensagens sobre o erro para outros pacotes internos. O erro que podemos corrigir geralmente é um dos primeiros ou últimos erros listados.

Às vezes, conseguiremos detectar o erro e, outras vezes, não teremos ideia do que a saída significa. Com certeza vale a pena tentar, e usar a saída de log para identificar com sucesso um erro traz uma satisfação muito grande. À medida que reserva um tempo analisando a saída de log, você ficará melhor em identificar as informações mais significativas.

Solução de problemas específicos de sistema operacional

É possível desenvolver em qualquer sistema operacional preferido e fazer o push para qualquer host. As ferramentas para fazer push são desenvolvidas o suficiente para modificar o projeto conforme necessário e para executá-lo corretamente no sistema remoto. No entanto, existem alguns problemas específicos de sistema operacional que podem surgir.

No processo de deploy do Platform.sh, uma das fontes mais prováveis de dificuldades é a instalação da CLI. Vejamos o comando para fazer isso:

```
$ curl -fsS https://platform.sh/cli/installer | php
```

O comando começa com `curl`, uma ferramenta que possibilita solicitar recursos remotos, acessados por meio de um URL, dentro de um terminal. Aqui está sendo usado para fazer o download do instalador da CLI de um servidor Platform.sh. A seção `-fsS` do comando é um conjunto de flags que modificam como o curl é executado. A flag `f` informa ao curl para suprimir a maioria das mensagens de erro,

assim o instalador da CLI consegue lidar com essas mensagens em vez de informá-las. A flag `s` instrui o curl a executar silenciosamente; permite que o instalador da CLI decida quais informações mostrar no terminal. A flag `S` informa ao curl para mostrar uma mensagem de erro se o comando geral falhar. O `| php` no final do comando instrui seu sistema a executar o arquivo do instalador baixado com um interpretador PHP, já que a CLI do Platform.sh é desenvolvida em PHP.

Ou seja, seu sistema precisa de curl e PHP para instalar a CLI do Platform.sh. Para usar a CLI, precisaremos também do Git e de um terminal que possa executar comandos Bash. O *Bash* é uma linguagem disponível na maioria dos ambientes de servidor. A maioria dos sistemas modernos tem muito espaço para a instalação de diversas ferramentas como essa.

As próximas seções o ajudarão a atender a esses requisitos para seu sistema operacional. Se ainda não tiver o Git instalado, confira as instruções para instalá-lo na página *588* do Apêndice D e confira a seção do sistema operacional que está usando.

NOTA *Uma excelente ferramenta para entender comandos de terminal como o mostrado aqui é* https://explainshell.com*. Digite o comando que está tentando entender e o site mostrará a documentação de todas as partes do comando. Veja o comando usado para instalar a CLI do Platform.sh.*

Deploy a partir do Windows

Nos últimos anos, o Windows ressurgiu em popularidade entre os programadores. O Windows integrou muitos elementos diferentes de outros sistemas operacionais, fornecendo aos usuários diversas opções de como fazer o trabalho de desenvolvimento local e interagir com sistemas remotos.

Uma das dificuldades mais significativas com um deploy a partir do Windows é que o núcleo do sistema operacional Windows não é o

mesmo usado por um servidor remoto Linux. Um sistema Windows básico tem um conjunto diferente de ferramentas e linguagens do que um sistema Linux básico. Ou seja, para fazer o deploy a partir do Windows, é necessário escolher como integrar conjuntos de ferramentas Linux em seu ambiente local.

Subsistema do Windows para Linux

Uma abordagem popular é usar o *Windows Subsystem for Linux* (*WSL*), ambiente que possibilita que o Linux seja executado diretamente no Windows. Se tiver o WSL configurado, usar a CLI do Platform.sh no Windows fica tão fácil quanto usá-la no Linux. A CLI não saberá que está sendo executada no Windows; apenas verá o ambiente Linux em que a estamos usando.

A configuração do WSL é um processo com duas etapas: primeiro, instale o WSL e, em seguida, escolha uma distribuição Linux para instalar no ambiente WSL. A configuração de ambiente WSL foge ao escopo deste livro; caso esteja interessado nessa abordagem e caso ainda não tenha configurado o ambiente, consulte a documentação em https://docs.microsoft.com/en-us/windows/wsl/about. Após configurarmos o WSL, podemos seguir as instruções na seção Linux deste apêndice para prosseguirmos com o deploy.

Git Bash

Podemos recorrer a outra abordagem para criar um ambiente local em que podemos fazer deploy: usar o *Git Bash*, ambiente de terminal compatível com o Bash, porém executado no Windows. O Git Bash é instalado com o Git quando usamos o instalador em https://git-scm.com. Essa abordagem pode funcionar, mas não é tão otimizada quanto o WSL. É necessário usar um terminal Windows para algumas etapas e um terminal Git Bash para outras.

Primeiro, precisamos instalar o PHP. É possível fazer essa instalação com o *XAMPP*, pacote que agrupa o PHP com algumas outras ferramentas cujo foco é desenvolvedor. Acesse https://apachefriends.org e

clique no botão para fazer o download do XAMPP para Windows. Abra o instalador e o execute; se vir um aviso sobre restrições de Controle de Conta de Usuário (UAC), clique em **OK**. Aceite todos os default do instalador.

Quando o instalador terminar de executar, precisaremos adicionar o PHP ao path do sistema; isso informará ao Windows onde procurar quando quisermos executar o PHP. No menu Iniciar, insira o **path** e clique em **Editar as Variáveis do Ambiente do Sistema**; clique no botão **Variáveis do Ambiente**. Devemos ver a variável Path destacada; clique em **Editar**. Clique em **Novo** para adicionar um novo path à lista atual de paths. Supondo que tenha mantido as configurações default ao executar o instalador do XAMPP, adicione **C:\xampp\php** na caixa exibida e clique em **OK**. Quando terminar, feche todas as caixas de diálogo do sistema que ainda estão abertas.

Com esses requisitos atendidos, podemos instalar a CLI do Platform.sh. Precisaremos usar um terminal Windows com privilégios de administrador; insira **command** no menu Iniciar e, no aplicativo Prompt de Comando, clique em **Executar como administrador**. No terminal que aparece, digite o seguinte comando:

```
> curl -fsS https://platform.sh/cli/installer | php
```

Isso instalará a CLI do Platform.sh, conforme descrito anteriormente.

Finalmente, usaremos o Git Bash. Para abrir um terminal Git Bash, acesse o menu Iniciar e procure por **git bash**. Clique no **aplicativo Git Bash** que aparece; veremos uma janela de terminal aberta. Nesse terminal, podemos usar comandos tradicionais Linux como ls e comandos Windows como dir. Para garantir que a instalação foi bem-sucedida, execute **platform list**. Você deve ver uma lista com todos os comandos da CLI do Platform.sh. Desse ponto em diante, faça todo o deploy usando a CLI do Platform.sh dentro de uma janela de terminal do Git Bash.

Deploy a partir do macOS

Apesar de o sistema operacional macOS não ser baseado em Linux, ambos foram desenvolvidos com princípios semelhantes. Na prática, significa que muitos dos comandos e fluxos de trabalho usados no macOS também funcionarão em um ambiente de servidor remoto. Talvez seja necessário instalar alguns recursos cujo foco seja o desenvolvedor para que todas as ferramentas sejam disponibilizadas no ambiente local do macOS. Se receber um prompt para instalar as *ferramentas de desenvolvedor de linha de comando* a qualquer momento, clique em **Instalar** para aprovar a instalação.

Ao instalar a CLI do Platform.sh, a maior dificuldade é garantir que o PHP seja instalado. Caso veja uma mensagem de que o comando `php` não foi encontrado, precisará instalar o PHP. Uma das maneiras mais fáceis de instalá-lo é recorrer ao gerenciador de pacotes *Homebrew*, que facilita a instalação de uma grande variedade de pacotes dos quais os programadores dependem. Se ainda não tem o Homebrew instalado, visite https://brew.sh e siga as instruções para instalá-lo.

Após o Homebrew ser instalado, use o seguinte comando para instalar o PHP:

```
$ brew install php
```

Isso levará um tempo para ser executado, mas, uma vez concluído, será possível instalar com sucesso a CLI do Platform.sh.

Deploy a partir do Linux

Como a maioria dos ambientes de servidor é em Linux, teremos pouca dificuldade para instalar e usar a CLI do Platform.sh. Se tentar instalar a CLI em um sistema com uma nova instalação do Ubuntu, você será informado exatamente de quais pacotes precisa:

```
$ curl -fsS https://platform.sh/cli/installer | php
Command 'curl' not found, but can be installed with:
sudo apt  install curl
Command 'php' not found, but can be installed with:
sudo apt install php-cli
```

A saída terá mais informações sobre alguns outros pacotes que funcionariam, além de algumas informações de versão. O seguinte comando instala curl e PHP:

```
$ sudo apt install curl php-cli
```

Após executar esse comando, o comando de instalação da CLI do Platform.sh deve ser executado com êxito. Como seu ambiente local é bastante semelhante à maioria dos ambientes de hospedagem em Linux, muito do que usamos para trabalhar em um terminal é válido também em um ambiente remoto.

Outras abordagens de deploy

Se o Platform.sh não funcionar, ou se quiser tentar uma abordagem diferente, há muitas plataformas de hospedagem para escolher. Algumas funcionam de modo semelhante ao processo descrito no Capítulo 20, e algumas têm uma abordagem bem diferente para realizar as etapas descritas no início deste apêndice:

- O Platform.sh possibilita utilizar um navegador para executar as etapas, nas quais usamos a CLI. Caso prefira interfaces baseadas em navegador do que fluxos de trabalho baseados em terminal, talvez queira usar essa abordagem.
- Existem outros provedores de hospedagem que oferecem abordagens baseadas em CLI e em navegador. Alguns desses provedores disponibilizam terminais em um navegador, assim não precisamos instalar nada em nosso sistema.
- Alguns provedores possibilitam fazer o push de um projeto para um site de hospedagem de código remoto, como o GitHub e, em seguida, conectam o repositório do GitHub ao site de hospedagem. O host então faz o push do código a partir do GitHub, em vez de exigir que enviemos o código do sistema local diretamente para o host. O Platform.sh também suporta esse tipo de fluxo de trabalho.
- Alguns provedores oferecem uma variedade de serviços que podemos escolher, a fim de montar uma infraestrutura que

funcione para o projeto. Normalmente, isso exige uma compreensão mais aprofundada do processo de deploy e do que um servidor remoto precisa para atender a um projeto. Por exemplo, o Amazon Web Services (AWS) e a plataforma Azure da Microsoft. E, talvez, seja mais difícil acompanhar os custos nesses tipos de plataformas, pois cada serviço pode acumular cobranças de forma independente.

- Muitas pessoas hospedam seus projetos em um servidor privado virtual (VPS). Nessa abordagem, alugamos um servidor virtual que se comporta como um computador remoto, faz login no servidor, instala o software necessário para executar o projeto, copia o código, define as conexões adequadas e possibilita que seu servidor comece a aceitar requisições.

Novas plataformas e abordagens de hospedagem surgem com frequência; encontre uma que seja amigável e invista tempo para aprender o processo de deploy desse provedor. Mantenha seu projeto por tempo suficiente a fim de saber o que funciona bem com a abordagem do provedor e o que não funciona. Nenhuma plataforma de hospedagem será perfeita; sempre precisaremos avaliar continuamente se o provedor atual que estamos usando é bom o bastante para nosso caso de uso.

Agora, farei uma última advertência sobre a escolha de uma plataforma e sobre a abordagem geral para deploy. Algumas pessoas entusiasmadas recomendarão abordagens e serviços de deploy bastante complexos, que tornam um projeto altamente confiável e são capazes de atender a milhões de usuários simultaneamente. Muitos programadores gastam bastante tempo, dinheiro e energia criando uma estratégia complexa de implantação, apenas para descobrirem que quase ninguém está usando seu projeto. A maioria dos projetos Django pode ser configurada com um pequeno plano de hospedagem e ajustada para atender a milhares de requisições por minuto. Caso seu projeto esteja recebendo um nível de tráfego menor, reserve um tempo para configurar sua implantação a fim de

que funcione bem em uma plataforma, antes de investir em infraestrutura destinada a alguns dos maiores sites do mundo.

Não raro, fazer deploy é um desafio e tanto, mas traz uma grande satisfação quando nosso projeto funciona conforme o esperado. Aproveite o desafio e procure ajuda quando necessário.


Entendendo
algoritmos
Um guia *ilustrado* para programadores e outros curiosos
Aditya Y. Bhargava
novatec
MANNING

Entendendo Algoritmos

Bhargava, Aditya Y.
9788575226629
264 páginas

Um guia ilustrado para programadores e outros curiosos. Um algoritmo nada mais é do que um procedimento passo a passo para a resolução de um problema. Os algoritmos que você mais utilizará como um programador já foram descobertos, testados e provados. Se você quer entendê-los, mas se recusa a estudar páginas e mais páginas de provas, este é o livro certo. Este guia cativante e completamente ilustrado torna simples aprender como utilizar os principais algoritmos nos seus programas. O livro Entendendo Algoritmos apresenta uma abordagem agradável para esse tópico essencial da ciência da computação. Nele, você aprenderá como aplicar algoritmos comuns nos problemas de programação enfrentados diariamente. Você começará com tarefas básicas como a ordenação e a pesquisa. Com a prática, você enfrentará problemas mais complexos, como a compressão de dados e a inteligência artificial. Cada exemplo é apresentado em detalhes e inclui diagramas e códigos

completos em Python. Ao final deste livro, você terá dominado algoritmos amplamente aplicáveis e saberá quando e onde utilizá-los. O que este livro inclui A abordagem de algoritmos de pesquisa, ordenação e algoritmos gráficos Mais de 400 imagens com descrições detalhadas Comparações de desempenho entre algoritmos Exemplos de código em Python Este livro de fácil leitura e repleto de imagens é destinado a programadores autodidatas, engenheiros ou pessoas que gostariam de recordar o assunto.


Loiane Groner
Estruturas de dados e algoritmos com JavaScript
2ª Edição
Escreva um código JavaScript complexo e eficaz usando a mais recente ECMAScript
novatec
Packt>

Estruturas de dados e algoritmos com JavaScript

Groner, Loiane
9788575227282
408 páginas

Uma estrutura de dados é uma maneira particular de organizar dados em um computador com o intuito de usar os recursos de modo eficaz. As estruturas de dados e os algoritmos são a base de todas as soluções para qualquer problema de programação. Com este livro, você aprenderá a escrever códigos complexos e eficazes usando os recursos mais recentes da ES 2017. O livro Estruturas de dados e algoritmos com JavaScript começa abordando o básico sobre JavaScript e apresenta a ECMAScript 2017, antes de passar gradualmente para as estruturas de dados mais importantes, como arrays, filas, pilhas e listas ligadas. Você adquirirá um conhecimento profundo sobre como as tabelas hash e as estruturas de dados para conjuntos funcionam, assim como de que modo as árvores e os mapas hash podem ser usados para buscar arquivos em um disco rígido ou para representar

um banco de dados. Este livro serve como um caminho para você mergulhar mais fundo no JavaScript. Você também terá uma melhor compreensão de como e por que os grafos – uma das estruturas de dados mais complexas que há – são amplamente usados em sistemas de navegação por GPS e em redes sociais. Próximo ao final do livro, você descobrirá como todas as teorias apresentadas podem ser aplicadas para solucionar problemas do mundo real, trabalhando com as próprias redes de computador e com pesquisas no Facebook. Você aprenderá a: • declarar, inicializar, adicionar e remover itens de arrays, pilhas e filas; • criar e usar listas ligadas, duplamente ligadas e ligadas circulares; • armazenar elementos únicos em tabelas hash, dicionários e conjuntos; • explorar o uso de árvores binárias e árvores binárias de busca; • ordenar estruturas de dados usando algoritmos como bubble sort, selection sort, insertion sort, merge sort e quick sort; • pesquisar elementos em estruturas de dados usando ordenação sequencial e busca binária


O'REILLY
Introdução à linguagem
SQL
ABORDAGEM PRÁTICA
PARA INICIANTES
novatec
Thomas Nield

Introdução à Linguagem SQL

Nield, Thomas
9788575227466
144 páginas

Atualmente as empresas estão coletando dados a taxas exponenciais e mesmo assim poucas pessoas sabem como acessá-los de maneira relevante. Se você trabalha em uma empresa ou é profissional de TI, este curto guia prático lhe ensinará como obter e transformar dados com o SQL de maneira significativa. Você dominará rapidamente os aspectos básicos do SQL e aprenderá como criar seus próprios bancos de dados. O autor Thomas Nield fornece exercícios no decorrer de todo o livro para ajudá-lo a praticar em casa suas recém descobertas aptidões no uso do SQL, sem precisar empregar um ambiente de servidor de banco de dados. Além de aprender a usar instruções-chave do SQL para encontrar e manipular seus dados, você descobrirá como projetar e gerenciar eficientemente bancos de dados que atendam às suas necessidades. Também veremos como: •Explorar bancos de dados relacionais, usando modelos leves e centralizados •Usar o SQLite e o SQLiteStudio para criar

bancos de dados leves em minutos •Consultar e transformar dados de maneira significativa usando SELECT, WHERE, GROUP BY e ORDER BY •Associar tabelas para obter uma visualização mais completa dos dados da empresa •Construir nossas próprias tabelas e bancos de dados centralizados usando princípios de design normalizado •Gerenciar dados aprendendo como inserir, excluir e atualizar registros


Fundamentos de
HTML5 e CSS3
CSS3
HTML5
novatec
Maurício Samy Silva
www.maujor.com

Fundamentos de HTML5 e CSS3

Silva, Maurício Samy
9788575227084
304 páginas

Fundamentos de HTML5 e CSS3 tem o objetivo de fornecer aos iniciantes e estudantes da área de desenvolvimento web conceitos básicos e fundamentos da marcação HTML e estilização CSS, para a criação de sites, interfaces gráficas e aplicações para a web. Maujor aborda as funcionalidades da HTML5 e das CSS3 de forma clara, em linguagem didática, mostrando vários exemplos práticos em funcionamento no site do livro. Mesmo sem conhecimento prévio, com este livro o leitor será capaz de: •Criar um código totalmente semântico empregando os elementos da linguagem HTML5. •Usar os atributos da linguagem HTML5 para criar elementos gráficos ricos no desenvolvimento de aplicações web. •Inserir mídia sem dependência de plugins de terceiros ou extensões proprietárias. •Desenvolver formulários altamente interativos com validação no lado do cliente utilizando atributos criados especialmente para essas finalidades. •Conhecer os mecanismos de aplicação de estilos, sua

sintaxe, suas propriedades básicas, esquemas de posicionamento, valores e unidades CSS3. •Usar as propriedades avançadas das CSS3 para aplicação de fundos, bordas, sombras, cores e opacidade. •Desenvolver layouts simples com uso das CSS3.

A Linguagem de Programação Go

Alan A. A. Donovan
Brian W. Kernighan

novatec

A Linguagem de Programação Go

Donovan, Alan A. A.

9788575226551

480 páginas

A linguagem de programação Go é a fonte mais confiável para qualquer programador que queira conhecer Go. O livro mostra como escrever código claro e idiomático em Go para resolver problemas do mundo real. Esta obra não pressupõe conhecimentos prévios de Go nem experiência com qualquer linguagem específica, portanto você a achará acessível, independentemente de se sentir mais à vontade com JavaScript, Ruby, Python, Java ou C++.

Compre agora e leia