

CORY ALTHOFF

# PROGRAMADOR *Autodidata*

Guia definitivo para  
programar profissionalmente

novatec

# **Programador Autodidata**

**Guia definitivo para  
programar profissionalmente**

**Cory Althoff**

Novatec



All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review.

**Todos os direitos reservados. Este livro ou qualquer parte dele não pode ser reproduzido ou usado de qualquer maneira sem a permissão expressa por escrito do editor, exceto para o uso de citações breves em uma resenha de livro.**

Copyright © 2022 Novatec Editora Ltda.

**Editor: Rubens Prates 20221011**

Tradução: Aldir Coelho Corrêa da Silva Revisão gramatical: Alexandra Resende

ISBN impresso: 978-85-7522-835-7

**ISBN ebook: 978-85-7522-836-4**

Histórico de impressões:

**Outubro/2022 Primeira edição**

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil Tel.: +55 11 2959-6529

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

Site: <https://novatec.com.br>

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

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

LinkedIn: [linkedin.com/in/novatec](https://linkedin.com/in/novatec) GRA20221011

*Dedico este livro aos meus pais, Abby e James Althoff,  
por sempre me apoiarem.*

# Sumário

## Sobre o autor

## Agradecimentos

## Parte I Introdução à programação

### Capítulo 1 Introdução

- Como o livro está estruturado
- O fim do jogo primeiro
- Você não está sozinho
- A vantagem de ser autodidata
- Porque você deve programar
- Não desista
- Como este livro está formatado
- Tecnologias usadas no livro
- Vocabulário
- Desafio

### Capítulo 2 Começando

- O que é programar
- O que é Python
- Instalação do Python
- Solução de problemas
- Shell interativo
- Salvando programas
- Execução de exemplos de programas
- Vocabulário
- Desafio

### Capítulo 3 Introdução à programação

- Exemplos
- Comentários
- Exibição
- Linhas
- Palavras-chave
- Espaçamento
- Tipos de dados
- Constantes e variáveis
- Sintaxe

[Erros e exceções](#)  
[Operadores aritméticos](#)  
[Operadores de comparação](#)  
[Operadores lógicos](#)  
[Instruções condicionais](#)  
[Instruções](#)  
[Vocabulário](#)  
[Desafios](#)

## **Capítulo 4 Funções**

[Representação de conceitos](#)  
[Funções](#)  
[Definição de funções](#)  
[Funções internas](#)  
[Reutilização de funções](#)  
[Parâmetros obrigatórios e opcionais](#)  
[Escopo](#)  
[Manipulação de exceções](#)  
[Docstrings](#)  
[Só use uma variável quando necessário](#)  
[Vocabulário](#)  
[Desafios](#)

## **Capítulo 5 Contêineres**

[Métodos](#)  
[Listas](#)  
[Tuplas](#)  
[Dicionários](#)  
[Contêineres em contêineres](#)  
[Vocabulário](#)  
[Desafios](#)

## **Capítulo 6 Manipulação de strings**

[Strings com aspas triplas](#)  
[Índices](#)  
[Strings são imutáveis](#)  
[Concatenação](#)  
[Multiplicação de strings](#)  
[Alteração de letras maiúsculas para minúsculas e vice-versa](#)  
[Método format](#)  
[Método split](#)  
[Método join](#)  
[Remoção de espaços](#)

[Método replace](#)  
[Busca de um índice](#)  
[Palavra-chave in](#)  
[Escape de strings](#)  
[Nova linha](#)  
[Fatiamento](#)  
[Vocabulário](#)  
[Desafios](#)

## **Capítulo 7 Loops**

[Loops for](#)  
[Função range](#)  
[Loops while](#)  
[Instrução break](#)  
[Instrução continue](#)  
[Loops aninhados](#)  
[Vocabulário](#)  
[Desafios](#)

## **Capítulo 8 Módulos**

[Módulos internos importantes](#)  
[Importação de outros módulos](#)  
[Vocabulário](#)  
[Desafios](#)

## **Capítulo 9 Arquivos**

[Gravação em arquivos](#)  
[Fechamento automático de arquivos](#)  
[Leitura em arquivos](#)  
[Arquivos CSV](#)  
[Vocabulário](#)  
[Desafios](#)

## **Capítulo 10 Juntando tudo**

[Força](#)  
[Desafio](#)

## **Capítulo 11 Prática**

[Leitura](#)  
[Outros recursos](#)  
[Como obter ajuda](#)

## **Parte II Introdução à programação orientada a objetos**



## **Capítulo 12 Paradigmas de programação**

[Estado](#)

[Programação procedural](#)

[Programação funcional](#)

[Programação orientada a objetos](#)

[Vocabulário](#)

[Desafios](#)

## **Capítulo 13 Os quatro pilares da programação orientada a objetos**

[Encapsulamento](#)

[Abstração](#)

[Polimorfismo](#)

[Herança](#)

[Composição](#)

[Vocabulário](#)

[Desafios](#)

## **Capítulo 14 Mais programação orientada a objetos**

[Variáveis de classe versus variáveis de instância](#)

[Métodos mágicos](#)

[Palavra-chave is](#)

[Vocabulário](#)

[Desafios](#)

## **Capítulo 15 Juntando tudo**

[Cartas](#)

[Baralho](#)

[Jogador](#)

[Jogo](#)

[Guerra](#)

## **Parte III Introdução às ferramentas de programação**

### **Capítulo 16 Bash**

[Acompanhamento](#)

[Busca do Bash](#)

[Comandos](#)

[Comandos recentes](#)

[Caminhos relativos versus absolutos](#)

[Navegação](#)

[Flags](#)

[Arquivos ocultos](#)

[Pipes](#)

[Variáveis de ambiente](#)

[Usuários](#)

[Saiba mais](#)

[Vocabulário](#)

[Desafios](#)

## **Capítulo 17 Expressões regulares**

[Preparação](#)

[Busca simples](#)

[Busca no começo e no fim](#)

[Busca de vários caracteres](#)

[Busca de dígitos](#)

[Repetição](#)

[Escape](#)

[Ferramenta de expressão regular](#)

[Vocabulário](#)

[Desafios](#)

## **Capítulo 18 Gerenciadores de pacotes**

[Pacotes](#)

[Gerenciador de pacotes pip](#)

[Ambientes virtuais](#)

[Vocabulário](#)

[Desafio](#)

## **Capítulo 19 Controle de versões**

[Repositórios](#)

[Como começar](#)

[Pushing e pulling](#)

[Exemplo de pushing](#)

[Exemplo de pulling](#)

[Reversão de versões](#)

[diff](#)

[Próximas etapas](#)

[Vocabulário](#)

[Desafios](#)

## **Capítulo 20 Juntando tudo**

[HTML](#)

[Scraping no Google Notícias](#)

[Vocabulário](#)

[Desafio](#)

## **Parte IV Introdução à ciência da computação**

## **Capítulo 21 Estruturas de dados**

[Estruturas de dados](#)

[Pilhas](#)

[Inversão de uma string com uma pilha](#)

[Filas](#)

[Fila para ingressos](#)

[Vocabulário](#)

[Desafios](#)

## **Capítulo 22 Algoritmos**

[FizzBuzz](#)

[Busca sequencial](#)

[Palíndromo](#)

[Anagrama](#)

[Contagem das ocorrências de caracteres](#)

[Recursão](#)

[Vocabulário](#)

[Desafio](#)

## **Parte V Conseguir um emprego**

### **Capítulo 23 Melhores práticas de programação**

[Escreva código como último recurso](#)

[DRY](#)

[Ortogonalidade](#)

[Cada dado deve ter apenas uma representação](#)

[As funções devem executar uma única ação](#)

[Se estiver demorando muito, provavelmente você cometeu um erro](#)

[Faça as coisas da melhor maneira na primeira vez](#)

[Siga as convenções](#)

[Use um IDE poderoso](#)

[Logging](#)

[Teste](#)

[Revisões de código](#)

[Segurança](#)

[Vocabulário](#)

### **Capítulo 24 Seu primeiro emprego em programação**

[Escolha um caminho](#)

[Ganhando a experiência inicial](#)

[Como conseguir uma entrevista](#)

[A entrevista](#)

[Atalhos para o sucesso na entrevista](#)

## **Capítulo 25 Trabalho em equipe**

[Domine o básico](#)

[Não pergunte sobre o que puder encontrar no Google](#)

[Alteração de código](#)

[Síndrome do impostor](#)

## **Capítulo 26 Leitura adicional**

[Os clássicos](#)

[Aulas online](#)

[Hacker News](#)

## **Capítulo 27 Próximas etapas**

[Encontre um mentor](#)

[Tente aprofundar-se](#)

[Outros conselhos](#)

# Sobre o autor

Cory Althoff é um programador, palestrante e autor, cujo trabalho inclui os livros *Programador Autodidata* e *Cientista da Computação Autodidata*. Depois de se formar em ciência política, Cory aprendeu sozinho a programar, tornando-se um engenheiro de software no eBay. Os livros de Cory foram traduzidos para vários idiomas e ele foi destaque em publicações, como a Forbes e a CNBC. Mais de 250 mil desenvolvedores fazem parte da comunidade de programadores autodidatas que ele criou por meio de seu popular grupo no Facebook, blog, curso e newsletter. Cory mora na Califórnia com sua esposa e filha.

# Agradecimentos

Gostaria de agradecer a todos que ajudaram a tornar este livro possível. Meus pais, Abby e James Althoff, deram um enorme apoio durante todo o processo. Meu pai examinou cada página do livro e forneceu ótimos feedbacks. Sem ele, essa publicação não teria se tornado realidade. Minha namorada, Lauren Wordell, foi muito paciente enquanto eu trabalhava no livro. Gostaria de agradecer ao incrivelmente talentoso ilustrador Blake Bowers; a meus editores Steve Bush, Madeline Luce, Pam Walatka e Lawrence Sanfilippo; e ao meu amigo Antoine Sindu – muitas de nossas discussões acabaram entrando no livro. Também quero agradecer a Randee Fenner, que deu apoio ao projeto na Kickstarter e me apresentou a Pam. Obrigado ao meu antigo chefe Anzar Afaq, que me ajudou muito quando ingressei em sua equipe no eBay. Gostaria de lhe agradecer por todos os leitores beta<sup>1</sup> que examinaram este livro antecipadamente e deram feedback. Para concluir, quero agradecer a todos os profissionais da Kickstarter que endossaram este projeto, principalmente Jin Chun, Sunny Lee e Leigh Forrest. Muito obrigado a todos!

---

<sup>1</sup> N.T.: Leitores beta são pessoas, profissionais ou não, que tem como função ler uma obra, completa ou inacabada, que ainda não foi publicada para, posteriormente, fazer uma análise crítica.

PARTE I

# Introdução à programação

# CAPÍTULO 1

## Introdução

*“A maioria dos bons programadores programa não porque espera ser paga ou adulada pelas pessoas, mas porque é divertido programar.”*

### – Linus Torvalds

Formei-me em ciência política na Universidade Clemson. Pensei em ciência da computação, e cheguei até mesmo a me inscrever em uma aula de Introdução à Programação no primeiro ano, mas desisti rapidamente. Era muito difícil. Enquanto vivia no Vale do Silício após a graduação, decidi que precisava aprender a programar. Um ano depois estava trabalhando como engenheiro de software II no eBay (acima do nível inicial do cargo de engenheiro de software, mas abaixo de um engenheiro de software sênior). Não quero dar a impressão de que foi fácil. Foi desafiador. Contudo, enquanto tentava ser bem-sucedido, também me diverti bastante.

Comecei minha jornada aprendendo a programar em Python, uma linguagem de programação popular. No entanto, não escrevi este livro apenas para ensiná-lo a programar em uma linguagem específica – embora ele faça isso. Ele também é sobre tudo o mais que os recursos padrão não ensinam. É sobre as coisas que tive de aprender por conta própria para me tornar um engenheiro de software. Este livro não é para alguém que esteja procurando uma introdução casual à programação para poder escrever código como um hobby. Ele foi escrito especificamente para quem deseja programar profissionalmente. Se o seu objetivo é tornar-se um engenheiro de software, um empreendedor, ou usar suas novas habilidades em programação em outra profissão, este livro foi escrito para você.

Aprender uma linguagem de programação é apenas parte da empreitada. Outras habilidades são necessárias para falarmos a linguagem dos cientistas da computação. Vou lhe ensinar tudo o que aprendi em minha jornada de programador iniciante a engenheiro de software profissional. Escrevi este livro para fornecer aos aspirantes a programador uma descrição do que eles precisam saber. Como programador autodidata, eu não sabia o que precisava aprender. Os livros de introdução à programação são todos iguais. Eles ensinam o básico de como programar em Python ou Ruby e nos deixam por conta própria. O feedback que ouvi de pessoas que terminaram de ler esses livros foi “O que faço agora? Ainda não sou um programador e não sei o que aprender em seguida”. Este livro é minha resposta a essa pergunta.

### Como o livro está estruturado

Muitos dos assuntos que são abordados em um único capítulo deste livro poderiam



ser – e são – abordados por livros inteiros. Meu objetivo não é abordar cada detalhe de cada assunto que você precisa conhecer. O que quero é lhe dar um mapa – uma descrição de todas as habilidades que você precisará desenvolver para programar profissionalmente. O livro foi dividido em cinco partes:

- **Parte I: Introdução à programação.** Você escreverá seu primeiro programa o mais rápido possível, espero que ainda hoje.
- **Parte II: Introdução à programação orientada a objetos.** Abordarei os diferentes paradigmas da programação – concentrando-me na programação orientada a objetos. Você criará um game que mostrará o poder de programar. Depois desta seção você ficará viciado em programar.
- **Parte III: Introdução às ferramentas de programação.** Você aprenderá a usar diferentes ferramentas para levar sua produtividade em programação para o próximo nível. A essa altura, estará viciado em programar e vai querer ficar cada vez melhor. Você aprenderá mais sobre seu sistema operacional, como usar expressões regulares para aumentar sua produtividade, como instalar e gerenciar programas de outras pessoas e como colaborar com outros engenheiros usando o controle de versões.
- **Parte IV: Introdução à ciência da computação.** Esta seção é uma breve introdução à ciência da computação. Abordarei dois tópicos importantes: algoritmos e estruturas de dados.
- **Parte V: Conseguir um emprego.** A última seção é sobre melhores práticas de programação, conseguir um emprego como engenheiro de software, trabalhar em equipe e ser um programador melhor. Fornecerei dicas de como passar em uma entrevista técnica e como trabalhar em equipe, e darei aconselhamento sobre como aperfeiçoar ainda mais suas habilidades.

Se você não tem nenhuma experiência em programação, deve praticar o máximo possível por conta própria entre cada seção. Não tente ler este livro rápido demais. Use-o como um guia e pratique o que aprendeu pelo tempo que precisar entre as seções.

Criei um curso online baseado neste livro (com conteúdo adicional) que você também pode achar útil. Ele está disponível em <https://www.udemy.com/course/selftaught-programmer/> (curso em inglês). Você também pode se inscrever para receber minha newsletter de programação semanal em [theselftaughtprogrammer.io](https://theselftaughtprogrammer.io) (também em inglês).

## O fim do jogo primeiro

A maneira como aprendi a programar é oposta a como geralmente a ciência da computação é ensinada, e estruturei o livro para que siga minha abordagem.

Tradicionalmente, passamos muito tempo aprendendo teoria – tanto tempo que muitos graduados em ciência da computação saem da faculdade sem saber programar. Em seu blog “Why Can’t Programmers.. Program?”<sup>1</sup>, Jeff Atwood escreveu “Como eu, o autor está tendo problemas com o fato de que 199 dos 200 candidatos a cada vaga de programação não consegue escrever códigos. Repito, eles não conseguem escrever código algum”. Essa revelação levou Atwood a criar o desafio de codificação **FizzBuzz**, um teste de programação usado em entrevistas para eliminar candidatos. A maioria das pessoas não consegue resolvê-lo e é por isso que você passará tanto tempo neste livro aprendendo as habilidades que usará na prática. Não se preocupe, você também aprenderá como passar no teste FizzBuzz.

Em *The Art of Learning*, Josh Waitzkin, que ficou conhecido pelo filme *Lances Inocentes*, descreve como ele aprendeu a jogar xadrez de maneira inversa. Em vez de estudar movimentos de abertura, primeiro começou a aprender o fim do jogo (quando existem apenas algumas peças no tabuleiro). Essa estratégia o ajudou a entender melhor o xadrez e ele acabou ganhando muitos campeonatos. Da mesma forma, acho mais eficiente aprender a programar primeiro, e depois aprender a teoria, quando você estiver ansioso para saber como tudo funciona. É por isso que vou esperar até a Parte IV do livro para apresentar a teoria da ciência da computação e vou simplificá-la ao máximo. Embora a teoria seja importante, ela terá muito mais valor quando você já tiver experiência em programação.

## **Você não está sozinho**

É cada vez mais comum aprender a programar fora da escola. Uma pesquisa de 2015 do Stack Overflow (uma comunidade de programadores online) descobriu que 48% dos entrevistados não tinha diploma em ciência da computação.<sup>2</sup>

## **A vantagem de ser autodidata**

Quando o eBay me contratou, entrei em uma equipe que incluía programadores com diplomas de ciência da computação das universidades de Stanford, da Califórnia-Berkeley, e da Universidade Duke, assim como dois doutores em física. Aos 25 anos fiquei assustado com o fato de meus colegas de equipe de 21 anos de idade saberem 10 vezes mais sobre programação e ciência da computação do que eu sabia.

Mesmo sendo assustador trabalhar com pessoas com diploma de graduação, de mestrado e de doutorado em ciência da computação, nunca se esqueça de que você tem o que gosto de chamar de a “vantagem de ser autodidata”. Você não está lendo este livro porque um professor mandou, você o está lendo porque tem desejo de aprender, e querer aprender é a maior vantagem que podemos ter. Além disso, não se

esqueça de que algumas das pessoas mais bem-sucedidas são programadores autodidatas. Steve Wozniak, fundador da Apple, é programador autodidata. Margaret Hamilton, que recebeu a Medalha Presidencial da Liberdade por seu trabalho nas missões Apollo da NASA de viagem à lua também é; David Karp, fundador do Tumblr, Jack Dorsey, fundador do Twitter, e Kevin Systrom, fundador do Instagram, também são.

## **Porque você deve programar**

A programação pode ajudá-lo em sua carreira, seja qual for sua profissão. Aprender a programar nos deixa mais capacitados. Gosto muito de inventar novas ideias e sempre tenho um novo projeto que quero iniciar. Após aprender a programar, pude sentar-me e desenvolver minhas ideias sem precisar encontrar alguém para fazer isso para mim.

Programar também o tornará melhor em tudo o que você fizer. Não existem muitas tarefas que não se beneficiem de boas habilidades de solução de problemas. Recentemente, precisei me ocupar da tediosa tarefa de procurar hospedagem no site Craigslist. Consegui escrever um programa para fazer o trabalho para mim e me enviar um email com os resultados. Aprendendo a programar você nunca mais terá de executar tarefas repetitivas.

Se você quiser tornar-se um engenheiro de software, há uma demanda crescente e não existe um número suficiente de pessoas qualificadas para preencher as vagas disponíveis. Em 2020, estima-se que 1 milhão de vagas de programação ficaram sem ser preenchidas.<sup>3</sup> Mesmo se o seu objetivo não for tornar-se um engenheiro de software, vagas em áreas como ciências e finanças estão começando a preferir candidatos com experiência em programação.

## **Não desista**

Se você não tiver nenhuma experiência em programação e estiver com medo de fazer essa jornada, quero que saiba que é capaz de fazê-la. Existem algumas considerações equivocadas sobre os programadores, como a de que eles são ótimos em matemática. Não são. Você não precisa ser bom em matemática para aprender a programar, mas é preciso se esforçar. Dito isso, grande parte do material abordado neste livro é mais fácil de aprender do que você pensa.

Para melhorar suas habilidades em programação você deve praticar todos os dias. A única coisa que pode atrapalhar é se você desistir, logo, veremos duas maneiras que o ajudarão a não fazer isso.

Quando eu estava começando, usava uma lista de verificação (checklist) para me assegurar de praticar todos os dias, e isso me ajudou a me manter focado.

Se você precisar de ajuda adicional, Tim Ferriss, um especialista em produtividade, recomenda a técnica a seguir para manter-se motivado: dê dinheiro para um amigo ou um membro da família com instruções para devolvê-lo após seu objetivo ser atingido dentro de um período específico ou doe-o para uma organização da qual não gosta, se falhar.

## Como este livro está formatado

Os capítulos do livro são interligados. Tentei evitar reexplicar conceitos, portanto, lembre-se disso. Os termos importantes aparecem em **negrito** na primeira vez que os apresento. Há uma seção de vocabulário no fim de cada capítulo onde cada palavra em **negrito** é definida. Também há desafios no fim de cada capítulo para ajudá-lo a desenvolver suas habilidades em programação, assim como links para as soluções.

## Tecnologias usadas no livro

Este livro ensina a usar certas tecnologias que lhe darão muita experiência prática. Tentei ser imparcial quanto às tecnologias, e em vez de me concentrar nelas, enfoco os conceitos.

Em alguns casos, tive de escolher entre várias tecnologias diferentes. No Capítulo 20, “Controle de versões” (para leitores não familiarizados com o controle de versões, explicarei posteriormente), percorro os aspectos básicos do uso do Git, um popular sistema de controle de versões. Escolhi o Git porque o considero o padrão do setor empresarial para o controle de versões. Usei Python na maioria dos exemplos de programação por se tratar de uma linguagem de programação popular e por ser muito fácil de ler, mesmo que você nunca a tenha usado. Também há grande procura por desenvolvedores Python em quase todas as áreas.

Para seguir os exemplos do livro você precisará de um computador. O computador tem um **sistema operacional** – um programa que é o intermediário entre os componentes físicos da máquina e você. O que vemos quando olhamos para a tela do computador chama-se **interface gráfica de usuário** ou GUI (graphical user interface), que faz parte do sistema operacional.

Existem três sistemas operacionais populares para computadores desktop e laptop: **Windows**, **Unix** e **Linux**. O Windows é o sistema operacional da Microsoft. O Unix é um sistema operacional criado nos anos 1970. O sistema operacional atual da Apple é baseado no Unix. De agora em diante, quando me referir ao Unix, estarei me referindo ao sistema operacional desktop da Apple. O Linux é um sistema operacional **open-source** usado pela maioria dos **servidores** mundiais. Um servidor é um computador ou um programa de computador que executa tarefas, como hospedar um site. Open-source significa que uma empresa ou pessoa não é o

proprietário do software e ele pode ser redistribuído e modificado. Tanto o Linux quanto o Unix são sistemas operacionais **baseados no padrão Unix (Unix-like)**, o que significa que eles são muito semelhantes. Este livro presume que você esteja usando um computador que execute o Windows, o Unix ou o Ubuntu (uma versão popular do Linux) como seu sistema operacional.

## Vocabulário

**FizzBuzz:** Teste de programação usado em entrevistas para eliminar candidatos.

**Interface gráfica de usuário (GUI):** Parte do sistema operacional que vemos quando olhamos para a tela do computador.

**Linux:** Sistema operacional open-source usado pela maioria dos servidores mundiais.

**Open-source:** Software que não é de propriedade de uma empresa ou de uma pessoa e em vez disso é mantido por um grupo de voluntários.

**Servidor:** Computador ou programa de computador que executa tarefas, como hospedar um site.

**Sistema operacional:** Programa que é o intermediário entre o usuário e os componentes físicos do computador.

**Sistemas operacionais baseados no padrão Unix:** Unix e Linux.

**Unix:** Sistema operacional criado nos anos 1970. O sistema operacional da Apple é baseado no Unix.

**Windows:** Sistema operacional da Microsoft.

## Desafio

1. Crie uma lista de verificação diária que inclua praticar programação.

---

1 <https://blog.codinghorror.com/why-cant-programmers-program/>

2 <https://www.infoworld.com/article/2908474/stack-overflow-survey-finds-nearly-half-have-no-degree-in-computer-science.html>

3 <https://www.wsj.com/articles/computer-programming-is-a-trade-lets-act-like-it-1407109947>

## CAPÍTULO 2

### Começando

*“Um bom programador é alguém que sempre olha para os dois lados antes de atravessar uma rua de mão única.”*

– Doug Linder

### O que é programar

Programar é escrever instruções para que um computador as execute. As instruções podem solicitar ao computador que exiba **Hello, World!**, colete dados na internet ou leia o conteúdo de um arquivo e o salve em um banco de dados. Essas instruções são chamadas de **código**. Os programadores escrevem código em muitas linguagens de programação diferentes. No passado era muito mais difícil programar, já que os programadores eram forçados a usar **linguagens de programação** enigmáticas e de baixo nível, como a **linguagem assembly**. Quando uma linguagem de programação é de baixo nível, sua aparência fica mais próxima à do sistema binário (0s e 1s) do que em uma **linguagem de programação de alto nível** (cuja leitura se parece mais com a do idioma inglês) e, portanto, é mais difícil de entender. Aqui está um programa simples escrito em uma linguagem assembly:

```
# http://tinyurl.com/z6facmk
```

```
global _start
section .text
_start:
mov rax, 1
mov rdi, 1
mov rsi, message
mov rdx, 13
syscall
; exit(0)
mov eax, 60
xor rdi, rdi
syscall
message:
db "Hello, World!", 10
```

A seguir temos o mesmo programa escrito em uma linguagem de programação moderna:

```
# http://tinyurl.com/zhj8ap6
```

```
print("Hello, World!")
```

Como você pode ver, atualmente a tarefa dos programadores ficou muito mais fácil. Você não precisará desperdiçar seu tempo aprendendo linguagens de programação enigmáticas e de baixo nível para programar. Em vez disso, aprenderá uma linguagem de programação fácil de ler chamada Python.

## O que é Python

**Python** é uma linguagem de programação open-source criada pelo programador holandês Guido van Rossum cujo nome vem do grupo de comediantes britânico Monty Python. Um dos principais insights de van Rossum foi o de que os programadores passam mais tempo lendo códigos do que escrevendo-os, logo, ele criou uma linguagem fácil de ler. O Python é uma das linguagens de programação mais populares e fáceis de aprender existentes no mundo. Ele é executado em todos os principais sistemas operacionais e computadores e é usado no desenvolvimento de servidores web, na criação de aplicações desktop e em tudo o mais que existe entre essas duas extremidades. Em razão de sua popularidade, há uma demanda significativa por programadores Python.

## Instalação do Python

Para seguir os exemplos deste livro, você precisa ter o Python versão 3 instalado. Você pode baixar o Python para Windows e Unix em <http://python.org/downloads>. Se estiver usando o Ubuntu, o Python 3 já estará instalado por padrão. Certifique-se de baixar o Python 3 e não o Python 2. Alguns dos exemplos do livro não funcionarão se você estiver usando o Python 2.

O Python está disponível para computadores de 32 e de 64 bits. Se você comprou seu computador depois de 2007, o mais provável é que ele seja de 64 bits. Se não tiver certeza, uma busca na internet deve ajudá-lo a descobrir.

Se estiver usando o Windows ou um Mac, baixe a versão do Python de 32 ou de 64 bits, abra o arquivo e siga as instruções. Você também pode visitar <http://theselftaughtprogrammer.io/installpython> para ver vídeos explicando como instalar o Python em cada sistema operacional.

## Solução de problemas

Desse ponto em diante, você precisa ter o Python instalado. Se estiver tendo problemas para instalar o Python, passe para o Capítulo 11 e veja a seção chamada “Como obter ajuda”.

## Shell interativo



O Python vem com um programa chamado IDLE, que é a abreviação de interactive development environment (e também é o sobrenome de Eric Idle, um dos membros da série Flying Circus do grupo Monty Python). É no IDLE que você digitará seu código Python. Logo depois de você baixar o Python, procure o IDLE no Explorer (PC), no Finder (Mac) ou no Nautilus (Ubuntu). Recomendo criar um atalho na área de trabalho para tornar fácil achá-lo.

Clique no ícone do IDLE e um programa com as linhas a seguir será aberto (isso pode mudar, logo, não se preocupe se a mensagem estiver ausente ou for diferente):

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 5 2015, 21:12:44) [GCC 4.2.1 (Apple Inc.
build 5666) (dot 3)] on darwin Type "copyright", "credits" or "license()" for
more information. >>>
```

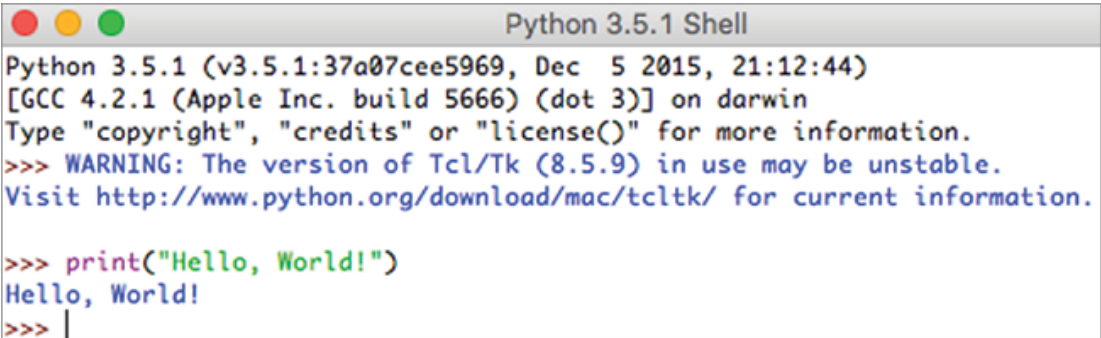
Esse programa chama-se shell interativo. Você pode digitar o código Python diretamente no shell interativo e ele exibirá os resultados. No prompt `>>>` digite:

```
print("Hello, World!")
```

Em seguida, pressione **Enter**.

O IDLE pode rejeitar o código copiado do Kindle, de outros ebooks ou de processadores de texto, como o Microsoft Word. Se você copiar e colar o código e for exibida uma mensagem de erro inexplicável, tente digitá-lo diretamente no shell. Você deve digitá-lo exatamente como escrito no exemplo, incluindo aspas, parênteses e qualquer outro símbolo de pontuação.

O shell interativo responderá exibindo **Hello, World!** (Alô Mundo).

A screenshot of a terminal window titled "Python 3.5.1 Shell". The window has a standard macOS title bar with red, yellow, and green buttons. The text inside the terminal shows the Python version and build information, followed by a warning about the Tcl/Tk version. Then, the user enters the command `>>> print("Hello, World!")` and the output `Hello, World!` is displayed. The prompt `>>>` is followed by a vertical bar, indicating the cursor is ready for the next command.

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be unstable.
Visit http://www.python.org/download/mac/tcltk/ for current information.
>>> print("Hello, World!")
Hello, World!
>>> |
```

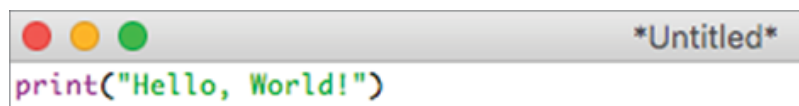
No mundo da programação, quando ensinamos a alguém uma nova linguagem de programação, é tradição que o primeiro programa ensinado seja como exibir **Hello, World!**. Portanto, parabéns! Você acabou de escrever seu primeiro programa.

## Salvando programas

O shell interativo é útil para tarefas computacionais breves, testar pequenos trechos de código e escrever programas curtos que você não pretenda usar novamente. Você também pode usar o IDLE para salvar um programa para reutilização. Inicie a aplicação IDLE, clique em **File** (na barra de menus no canto superior esquerdo do



editor do IDLE) e selecione **New File**. Selecionar essa opção abrirá um editor de texto que, geralmente, tem um plano de fundo vazio e branco. Você pode escrever o código nesse editor de texto e salvá-lo para execução posterior. Quando executar seu código, a saída aparecerá no shell interativo. Você terá de salvar as alterações feitas quando editar o código, antes de poder executá-lo novamente. Digite o programa **Hello, World!** no editor de texto:



Vá até **File** novamente e selecione **Save As**. Nomeie seu arquivo como **hello\_world.py** e salve-o. O nome de arquivos Python deve terminar com a extensão **.py**. Após salvar seu arquivo, clique em **Run** (na barra de menus no canto superior esquerdo do editor do IDLE) e selecione **Run Module**. Alternativamente, você pode pressionar a tecla de comando **F5**, o que seria equivalente a selecionar **Run Module** na barra de menus. **Hello, World!** será exibido no shell interativo, como se você tivesse digitado essa linha de código. No entanto, já que você salvou seu programa, agora poderá executá-lo quantas vezes quiser.

O programa que você criou é simplesmente um arquivo com extensão **.py**, localizado em seu computador onde quer que ele tenha sido salvo. O nome que escolhi para o arquivo – **hello\_world.py** – é totalmente arbitrário; você pode dar o nome que quiser a ele. Como nesse exemplo, escrever programas em Python só envolve digitar texto em arquivos e executá-los com o uso do shell interativo. Fácil, não é mesmo?

## Execução de exemplos de programas

No decorrer do livro, forneço exemplos de código e os resultados que serão exibidos quando você os executar. Sempre que eu fizer isso, você deve digitar o código e executá-lo por conta própria.

É melhor executar os exemplos curtos usando o shell, e o editor de texto será melhor para programas mais longos que você quiser salvar e editar. Se você cometer um erro em seu código no shell interativo – um erro de digitação, por exemplo – e o código não funcionar, será preciso digitar tudo novamente. Usar o editor de texto permitirá que você salve seu trabalho, logo, se cometer um erro terá apenas que corrigi-lo e reexecutar o programa.

Outra razão que torna essa distinção importante é que a saída de um programa executado a partir de um arquivo versus a partir do shell pode ser um pouco diferente. Se você digitar **100** no shell interativo e pressionar **Enter**, o shell interativo exibirá **100**. Se você digitar **100** em um arquivo **.py** e executá-lo, não haverá saída. Essa diferença pode causar confusão, logo, preste atenção no local de onde estiver

executando um programa se não obtiver a mesma saída do exemplo.

## Vocabulário

**Código:** As instruções que os programadores escrevem para um computador executar.

**Linguagem assembly:** Um tipo de linguagem de programação difícil de ler.

**Linguagem de programação de alto nível:** Linguagem de programação cuja leitura é mais aproximada à do idioma inglês do que na leitura de uma linguagem de programação de baixo nível.

**Linguagem de programação de baixo nível:** Linguagem de programação cuja escrita tem aparência mais próxima do sistema binário (0s e 1s) do que a de uma linguagem de programação de alto nível.

**Programação:** Escrever instruções para um computador executar.

**Python:** Linguagem de programação open-source fácil de ler que você aprenderá a usar neste livro. Foi criada por Guido van Rossum e recebeu seu nome em homenagem ao grupo de comediantes britânico Monty Python.

## Desafio

1. Tente exibir algo diferente de `Hello, World!`.

Solução: <http://tinyurl.com/noeujfu>.

## CAPÍTULO 3

### Introdução à programação

*“É o único trabalho que me vem à mente no qual posso ser ao mesmo tempo engenheiro e artista. Existe um incrível e rigoroso elemento técnico em sua execução, que é algo que me agrada porque é preciso ter um raciocínio muito preciso. Por outro lado, apresenta um lado criativo no qual os limites da imaginação são a única restrição.”*

#### – Andy Hertzfeld

Nosso primeiro programa exibiu **Hello, World!**. Vamos exibir essa frase 100 vezes. Digite o código a seguir no shell interativo (o comando `print` precisa ser indentado em exatamente quatro espaços): # <http://tinyurl.com/h79ob7s>

```
for i in range(100):  
    print("Hello, World!")
```

Seu shell deve exibir **Hello, World!** 100 vezes. Mesmo que provavelmente você nunca precise exibir **Hello, World!** 100 vezes, esse exemplo mostra como a programação é poderosa. Consegue pensar em outra coisa que você possa fazer 100 vezes tão facilmente? Eu não consigo. Esse é o poder da programação.

### Exemplos

De agora em diante, os exemplos de código terão esta aparência:

```
# http://tinyurl.com/h4qntgk
```

```
for i in range(100):  
    print("Hello, World!")
```

```
>> Hello, World!
```

```
>> Hello, World!
```

```
>> Hello, World!
```

```
...
```

A linha # <http://tinyurl.com/h4qntgk> contém uma URL que o levará a uma página onde está o código associado, assim você poderá copiá-lo e colá-lo facilmente no editor de texto do IDLE se estiver tendo problemas para fazê-lo ser executado. O texto que vem depois de `>>` é a saída do shell interativo. No decorrer do livro, você verá `>>` depois de cada exemplo de programação, representando a saída do programa (exibida no shell interativo). As reticências (...) significam “e assim por diante”.

Se não houver `>>` depois de um exemplo, isso significa que o programa não produz

saída ou que estou explicando um conceito e a saída não é importante.

Qualquer coisa que estiver escrita na fonte **Ubuntu Mono** será algum tipo de código, uma saída de um código ou um jargão de programação. Por exemplo, se eu me referir à palavra **for** do exemplo anterior, ela será escrita na fonte **Ubuntu Mono**.

**Ubuntu Mono** é uma fonte de largura fixa (não proporcional) que com frequência é usada para exibir texto de programação. Cada caractere tem a mesma largura, logo, a indentação e outras características de exibição para alinhamento de código são mais fáceis de observar.

Você pode executar os exemplos a partir do shell ou a partir de um arquivo **.py**. Lembre-se de que, como mencionei anteriormente, a saída a partir do shell é um pouco diferente, portanto, se você não estiver obtendo a mesma saída, essa é a razão. Se um exemplo exibir uma saída, mas não tiver a palavra **print** nele, você deve inserir o código no shell. Se a palavra **print** estiver presente em um exemplo, você deve executar o código a partir de um arquivo **.py**.

## Comentários

Um **comentário** é uma linha (ou parte de uma linha) de código escrita em inglês (ou em outro idioma), antecedida por um símbolo que solicitará à linguagem de programação que estiver sendo usada para ignorar essa linha (ou parte de uma linha) de código. Em Python, o símbolo de jogo da velha (**#**) é usado para criar comentários.

Um comentário explica o que uma linha de código faz. Os programadores usam comentários para facilitar o entendimento da linha de código para quem quer que a leia. Você pode escrever o que quiser em um comentário, contanto que ele só tenha uma linha: **# <http://tinyurl.com/hut6nwu>**

```
# Este é um comentário
```

```
print("Hello, World!")
```

```
>> Hello, World!
```

Só escreva um comentário se estiver fazendo algo incomum em seu código ou explicando algo que não seja óbvio no próprio código. Use os comentários moderadamente – não faça comentários em cada linha de código que você escrever – guarde-os para situações especiais. Aqui está um exemplo de um comentário desnecessário: **# <http://tinyurl.com/jpzlwqq>**

```
# exhibe Hello, World!
```

```
print("Hello, World!")
```

Ele é desnecessário porque já está claro o que a linha de código faz. A seguir temos

um exemplo de um comentário adequado: # <http://tinyurl.com/z52c8z8>

```
import math

# tamanho de uma diagonal
l = 4
w = 10
d = math.sqrt(l**2 + w**2)
```

Mesmo que você tenha entendido como esse código funciona, pode não saber como calcular o tamanho da diagonal de um retângulo, logo, o comentário é útil.

## Exibição

Você não está restrito a exibir **Hello, World!** em seus programas. Você pode exibir o que quiser, contanto que coloque entre aspas: # <http://tinyurl.com/zh5g2a3>

```
print("Python")
>> Python

# http://tinyurl.com/hhwqva2

print("Hola!")
>> Hola!
```

## Linhas

Os programas Python são compostos de linhas de código. Considere este programa:

```
# http://tinyurl.com/jq2w5ro

# linha1
# linha2
# linha3
```

Há três linhas de código. É útil podermos referenciar cada trecho de código pela linha em que ele se encontra. No IDLE podemos ir em **Edit** e selecionar **Go to Line** para saltar para uma linha específica de um programa. Só podemos inserir no shell uma linha de código de cada vez. Não é possível copiar e colar várias linhas.

Pode ocorrer de um trecho de código ser longo e ocupar mais de uma linha. Um código incluído entre três aspas, parênteses, colchetes e chaves pode se estender para uma nova linha: # <http://tinyurl.com/zcdx3yo>

```
print("""This is a really really
really really long line of
```

```
code."")
```

Você pode usar uma barra invertida (\) para estender o código para a próxima linha, algo que não poderia ser feito de outra forma: # <http://tinyurl.com/hjcf2sa>

```
print\  
("This is a really really  
really long line of code.")
```

Esse exemplo e o anterior têm a mesma saída. A barra me permitiu inserir ("This is a really really really long line of code.") e `print` em linhas separadas, o que de outra forma não seria permitido. Essa prática não é muito comum, mas a usarei no decorrer do livro para estreitar os exemplos de modo que caibam em leitores de ebook com telas pequenas.

## Palavras-chave

Linguagens de programação como Python têm palavras com significados especiais, que são chamadas de **palavras-chave**. `for`, uma palavra-chave que você já viu, é usada para a execução de um código várias vezes. Você aprenderá mais palavras-chave ao longo deste capítulo.

## Espaçamento

Vejamos novamente seu programa que exibe `Hello, World!` 100 vezes: # <http://tinyurl.com/glp9xq6>

```
for i in range(100):  
    print("Hello, World!")
```

Como mencionei anteriormente, `print` tem indentação de quatro espaços. Abordarei o porquê em breve, mas essa prática permite que o Python saiba quando os blocos de código começam e terminam. Enquanto isso, é preciso que você saiba que sempre que encontrar uma indentação em um exemplo, ela será de quatro espaços. Sem um espaçamento apropriado, seu programa não funcionará.

Outras linguagens de programação não usam o espaçamento; elas usam palavras-chave ou chaves. Aqui está o mesmo programa escrito em outra linguagem de programação chamada JavaScript: # <http://tinyurl.com/hwa2zae>

```
# Este é um programa JavaScript  
# Ele não funcionará
```

```
for (i = 0; i < 100; i++) {
```

```
console.log("Hello, World!");
```

```
}
```

Os proponentes do Python acreditam que o uso obrigatório do espaçamento apropriado torna a linguagem menos difícil de ler e escrever do que outras linguagens. Como no exemplo anterior, mesmo quando o espaço não faz parte da linguagem de programação, os programadores o incluem para tornar seu código mais fácil de ler.

## Tipos de dados

O Python agrupa os dados em diferentes categorias chamadas **tipos de dados**. Em Python, cada valor de dado, como `2` ou `"Hello, World!"`, é chamado de **objeto**. Você aprenderá mais sobre os objetos na Parte II, mas por enquanto considere um objeto em Python como um valor de dado com três propriedades: identidade, tipo de dado e valor. A identidade de um objeto é sua localização na memória do computador que nunca muda. O tipo de dado de um objeto é a categoria de dados à qual o objeto pertence, que determina as propriedades que o objeto tem e nunca muda. O valor de um objeto é o dado que ele representa – o número `2`, por exemplo, tem o valor `2`.

`"Hello, World!"` é um objeto com o tipo de dado **str**, que é a abreviação de **string**, e o valor `"Hello, World!"`. Quando referenciamos um objeto com o tipo de dado **str**, ele é chamado de string. Uma string é uma sequência de um ou mais caracteres incluídos entre aspas. Um **caractere** é um único símbolo como `a` ou `1`. Você pode usar aspas simples ou duplas, mas as aspas do começo e do fim de uma string devem ser iguais: # <http://tinyurl.com/hh5kjwp>

```
"Hello, World!"
```

```
>> 'Hello, World!'
```

```
# http://tinyurl.com/heaxhsh
```

```
'Hello, World!'
```

```
>> 'Hello, World!'
```

As strings são usadas para representar texto e têm propriedades únicas.

Os números que você usou para fazer cálculos na seção anterior também são objetos – mas não são strings. Os números inteiros (`1`, `2`, `3`, `4` etc.) têm o tipo de dado **int**, que é a abreviação de **integer**. Como as strings, os inteiros têm propriedades únicas. Por exemplo, você pode multiplicar dois inteiros, mas não pode multiplicar duas strings.



Os números decimais (números com casas decimais) têm um tipo de dado chamado **float**. **2.1**, **8.2** e **9.9999** são todos objetos com o tipo de dado **float**. Eles são chamados de **números de ponto flutuante**. Como todos os tipos de dados, os números de ponto flutuante têm propriedades únicas e se comportam de maneira específica, ou seja, de forma semelhante aos inteiros: # <http://tinyurl.com/guoc4gy>

```
2.2 + 2.2
```

```
>> 4.4
```

Objetos com um tipo de dado **bool** são chamados de **booleanos** e têm um valor **True** ou **False**: # <http://tinyurl.com/jyllj2k>

```
True
```

```
>> True
```

```
# http://tinyurl.com/jzgsxz4
```

```
False
```

```
>> False
```

Objetos com um tipo de dado **NoneType** sempre têm o valor **None**. Eles são usados para representar a ausência de valor: # <http://tinyurl.com/h8oqo5v>

```
None
```

Explicarei como usar os diferentes tipos de dados no decorrer deste capítulo.

## Constantes e variáveis

Você pode usar o Python para fazer cálculos, como faria em uma calculadora. Pode somar, subtrair, dividir, multiplicar, elevar um número a uma potência, e muito mais. Lembre-se de digitar todos os exemplos desta seção no shell.

```
# http://tinyurl.com/zs65dp8
```

```
2 + 2
```

```
>> 4
```

```
# http://tinyurl.com/gs9nwrw
```

```
2 - 2
```

```
>> 0
```

```
# http://tinyurl.com/hasegvj
```

```
4 / 2
```

```
>> 2.0
```



```
# http://tinyurl.com/z8ok4q3
```

```
2 * 2
```

```
>> 4
```

Uma **constante** é um valor que nunca muda. Cada um dos números do exemplo anterior é uma constante; o número dois sempre representará o valor 2. Uma **variável**, por outro lado, referencia um valor que pode mudar. A variável é composta de um nome com um ou mais caracteres. Esse nome recebe um valor com o uso do **operador de atribuição** (o sinal =).

Algumas linguagens de programação exigem que o programador inclua “declarações” de variáveis que informem à linguagem que tipo de dado a variável terá. Por exemplo, na linguagem de programação C, podemos criar uma variável desta forma: # Não execute

```
int a;
```

```
a = 144;
```

Em Python é mais fácil, pois podemos criar uma variável simplesmente atribuindo um valor a ela com o operador de atribuição: # <http://tinyurl.com/hw64mrr>

```
b = 100
```

```
b
```

```
>> 100
```

Veja como alterar o valor de uma variável:

```
# http://tinyurl.com/hw97que
```

```
x = 100
```

```
x
```

```
x = 200
```

```
x
```

```
>> 100
```

```
>> 200
```

Você também pode usar duas variáveis para executar operações aritméticas:

```
# http://tinyurl.com/z8hv5j5
```

```
x = 10
```

```
y = 10
```

```
z = x + y
```

```
z
```

```
a = x - y
```

```
a
```

```
>> 20
```

```
>> 0
```

Com frequência, quando estiver programando, você vai precisar **incrementar** (aumentar) ou **decrementar** (diminuir) o valor de uma variável. Já que essa é uma operação padrão, o Python tem uma sintaxe especial – um atalho – para o incremento e o decremento de variáveis. Para incrementar uma variável, atribua-a a ela própria e no outro lado do sinal de igualdade some-a ao número segundo o qual deseja incrementá-la: # <http://tinyurl.com/zvzf786>

```
x = 10
```

```
x = x + 1
```

```
x
```

```
>> 11
```

Para decrementar uma variável, faça o mesmo, mas subtraia o número ao qual deseja decrementá-la: # <http://tinyurl.com/gmuzdr9>

```
x = 10
```

```
x = x - 1
```

```
x
```

```
>> 9
```

Esses exemplos são perfeitamente válidos, mas existe um método mais curto que você deve usar: # <http://tinyurl.com/zdva5wq>

```
x = 10
```

```
x += 1
```

```
x
```

```
>> 11
```

```
# http://tinyurl.com/jqw4m5r
```

```
x = 10
```

```
x -= 1
```

```
x
```

```
>> 9
```

As variáveis não estão restritas ao armazenamento de valores inteiros. Elas podem referenciar qualquer tipo de dado: # <http://tinyurl.com/jsyggqcy>

```
hi = "Hello, World!"
```

```
# http://tinyurl.com/h47ty49
```

```
my_float = 2.2
```

```
# http://tinyurl.com/hx9xluq
```

```
my_boolean = True
```

Você pode nomear as variáveis como quiser, contanto que siga quatro regras:

1. Os nomes das variáveis não podem ter espaços. Se quiser usar duas palavras em uma variável, insira um caractere underscore ( `_` ) entre elas, como em `my_variable = "A string!"`
2. Os nomes de variáveis só podem conter letras, números e o caractere underscore ( `_` ).
3. Você não pode começar o nome de uma variável com um número. Embora seja possível começar uma variável com um caractere underscore, ele tem um significado especial que abordarei posteriormente, portanto, evite usá-lo até chegarmos lá.
4. Você não pode usar palavras-chave Python para nomes de variáveis. Uma lista das palavras-chave pode ser encontrada em <http://theselftaughtprogrammer.io/keywords>.

## Sintaxe

A **sintaxe** é o conjunto de regras, princípios e processos que controlam a estrutura das frases de determinada linguagem, especificamente a ordem das palavras<sup>1</sup>. Todos os idiomas (inglês, português etc.) têm sintaxe, assim como o Python.

Em Python, as strings são sempre incluídas em aspas. Esse é um exemplo da sintaxe do Python. A seguir temos um programa Python válido: `# http://tinyurl.com/j7c2npf`

```
print("Hello, World!")
```

Ele é válido porque seguimos a sintaxe do Python incluindo o texto em aspas ao definir a string. Se só usássemos aspas em um lado do texto, violaríamos a sintaxe e o código não funcionaria.

## Erros e exceções

Se você escrever um programa Python e desconsiderar a sintaxe da linguagem, verá um ou mais erros ao executar seu programa. O shell Python informará que seu código não funcionou e fornecerá informações sobre o erro. Veja o que acontecerá se você tentar definir uma string em Python com aspas apenas em um lado: `# http://tinyurl.com/hp2plhs`

```
# Este código tem um erro
```

```
my_string = "Hello World.
```

```
>> File "/Users/coryalthoff/PycharmProjects/se.py", line 1 my_string = 'd ^
SyntaxError: EOL while scanning string literal Essa mensagem está informando que
há um erro de sintaxe em seu programa. Os erros de sintaxe são fatais. Um programa
não pode ser executado com um erro de sintaxe. Quando você tentar executar um
programa com um erro de sintaxe, o Python lhe informará sobre isso no shell. A
mensagem descreverá em que arquivo o erro se encontra, em que linha ele ocorreu e
qual é o tipo de erro. Embora esse erro possa parecer assustador, ele acontece o
tempo todo.
```

Quando houver um erro em seu código, você deve ir até o número de linha correspondente ao local em que o problema ocorreu e tentar descobrir o que fez de errado. Nesse exemplo, você teria de acessar a primeira linha do código. Após examiná-la acabaria notando que há aspas apenas em um lado da string. Para corrigir o erro, adicione aspas ao final da string e reexecute o programa. De agora em diante, representarei a saída de um erro desta forma: `>> SyntaxError: EOL while scanning string literal` Para facilitar a leitura, exibirei apenas a última linha da mensagem de erro.

O Python tem dois tipos de erros: erros de sintaxe e exceções. Qualquer erro que não for um erro de sintaxe será uma **exceção**. Um `ZeroDivisionError` é uma exceção que ocorrerá se você tentar fazer uma divisão por zero.

Ao contrário dos erros de sintaxe, as exceções não são necessariamente fatais (existe uma maneira de fazer um programa ser executado mesmo se houver uma exceção, o que você aprenderá no próximo capítulo). Quando ocorre uma exceção, os programadores Python dizem “O Python (ou o programa) lançou uma exceção”. Aqui está um exemplo de uma exceção: # <http://tinyurl.com/jxpztcx>

```
# Este código tem um erro
```

```
10 / 0
```

```
>> ZeroDivisionError: division by zero Se você indentar seu código
incorretamente, verá um IndentationError: # http://tinyurl.com/gtp6amr
```

```
# Este código tem um erro
```

```
y = 2
```

```
x = 1
```

```
>> IndentationError: unexpected indent Já que você está aprendendo a programar,
verá com frequência erros de sintaxe e exceções (incluindo alguns que não
abordei), mas eles diminuirão com o tempo. Lembre-se, quando você encontrar um
erro de sintaxe ou uma exceção, vá até a linha na qual o problema ocorreu,
examine-a e ache a solução (procure o erro ou a exceção na internet se não souber
resolver).
```

## Operadores aritméticos

Anteriormente, usei o Python para fazer cálculos aritméticos simples, como  $4 / 2$ . Os símbolos que você usou nesses exemplos chamam-se **operadores**. O Python divide os operadores em várias categorias e os que você viu até agora chamam-se **operadores aritméticos**. A seguir temos alguns dos operadores aritméticos mais comuns em Python:

Operador	Significado	Exemplo	Resultado
**	Expoente	$2 ** 2$	4
%	Módulo/resto	$14 \% 4$	2
//	Divisão de inteiros/floored quotient <sup>2</sup>	$13 // 8$	1
/	Divisão	$13 / 8$	1.625
*	Multiplicação	$8 * 2$	16
-	Subtração	$7 - 1$	6
+	Adição	$2 + 2$	4

Quando dois números são divididos, há um quociente e um resto. O quociente é o resultado da divisão e o resto é o que sobra. O operador módulo (%) retorna o resto. Por exemplo, 13 dividido por 5 é 2 com 3 como resto: # <http://tinyurl.com/grdcl95>

```
13 // 5
```

```
>> 2
```

```
# http://tinyurl.com/zsqwukd
```

```
13 % 5
```

```
>> 3
```

Quando você usar o módulo com o número dois como divisor, se não houver resto

(o módulo retornar 0), o número é par. Se houver resto, o número é ímpar: # <http://tinyurl.com/jerpe6u>

```
# par
12 % 2
```

```
>> 0
```

```
# http://tinyurl.com/gkudhcr
```

```
# ímpar
11 % 2
```

```
>> 1
```

Dois operadores são usados para a divisão. O primeiro é `//`, que retorna o quociente: # <http://tinyurl.com/hh9fqzy>

```
14 // 3
>> 4
```

O segundo é `/`, que retorna o resultado do primeiro número dividido pelo segundo como um número de ponto flutuante: # <http://tinyurl.com/zlkjjdp>

```
14 / 3
>> 4.666666666666667
```

Você pode elevar um número a uma potência com o operador de exponenciação:

```
# http://tinyurl.com/h8vuwd4
```

```
2 ** 2
>> 4
```

Os valores (neste caso, números) existentes dos dois lados de um operador são chamados de **operandos**. Juntos, dois operandos e um operador formam uma **expressão**. Quando seu programa for executado, o Python avaliará cada expressão e retornará um único valor. Se você digitar a expressão `2+2` no shell, o Python a avaliará como 4.

A **ordem das operações** é um conjunto de regras usadas em cálculos matemáticos para a avaliação de uma expressão. Você se lembra de "Please Excuse My Dear Aunt Sally (*Por favor, desculpe minha querida tia Sally*)?" Trata-se de um acrônimo que ajuda a lembrar da ordem das operações em equações matemáticas: parênteses, expoentes, multiplicação, divisão, adição e subtração. Os parênteses vêm antes dos expoentes, que vêm antes da multiplicação e da divisão, e depois temos a adição e a subtração. Se houver um empate entre os operadores, como no caso de `15 / 3 * 2`, a avaliação ocorrerá da esquerda para a direita. Neste caso, a resposta é o resultado de 15 dividido por 3 vezes 2. O Python segue a mesma ordem de operações quando

avalia expressões matemáticas: # <http://tinyurl.com/hgjy.j7o>

```
2 + 2 * 2
```

```
>> 6
```

```
# http://tinyurl.com/hsq7rcz
```

```
(2 + 2) * 2
```

```
>> 8
```

No primeiro exemplo,  $2 * 2$  é avaliado primeiro porque a multiplicação tem precedência sobre a adição.

No segundo exemplo,  $(2+2)$  é avaliado primeiro porque o Python sempre avalia antes as expressões em parênteses.

## Operadores de comparação

Os **operadores de comparação** são outra categoria de operadores em Python. Como os operadores aritméticos, eles são usados em expressões com operandos dos dois lados. Ao contrário das expressões com operadores aritméticos, as expressões com operadores de comparação apresentam como resultado **True** ou **False**.

Operador	Significado	Exemplo	Resultado
>	Maior que	$100 > 10$	True
<	Menor que	$100 < 10$	False
>=	Maior ou igual a	$2 <= 2$	True
<=	Menor ou igual a	$1 <= 4$	True
==	Igual	$6 == 9$	False
!=	Diferente	$3 != 2$	True

Uma expressão com o operador > retornará o valor **True** se o número da esquerda for maior do que o número da direita; caso contrário, retornará **False**: # <http://tinyurl.com/jm7cxzp>

```
100 > 10
```

```
>> True
```

Uma expressão com o operador < retornará o valor **True** se o número da esquerda for menor do que o número da direita; caso contrário, retornará **False**: # <http://tinyurl.com/gsdhr8q>

```
100 < 10
```

```
>> False
```

Uma expressão com o operador `>=` retornará o valor **True** se o número da esquerda for maior ou igual ao número da direita. Caso contrário, a expressão retornará **False**: # <http://tinyurl.com/jy2oefs>

```
2 >= 2
```

```
>> True
```

Uma expressão com o operador `<=` retornará o valor **True** se o número da esquerda for menor ou igual ao número da direita. Caso contrário, a expressão retornará **False**: # <http://tinyurl.com/jk599re>

```
2 <= 2
```

```
>> True
```

Uma expressão com o operador `==` retornará o valor **True** se os dois operandos forem iguais; caso contrário, retornará **False**: # <http://tinyurl.com/j2tsz9u>

```
2 == 2
```

```
>> True
```

```
# http://tinyurl.com/j5mr2q2
```

```
1 == 2
```

```
>> False
```

Uma expressão com o operador `!=` retornará **True** se os dois operandos não forem iguais; caso contrário, retornará **False**: # <http://tinyurl.com/gsw3zoe>

```
1 != 2
```

```
>> True
```

```
# http://tinyurl.com/z7pffk3
```

```
2 != 2
```

```
>> False
```

Anteriormente, você atribuiu números a variáveis, como em `x = 100`, usando `=`. Você pode ficar tentado a ler essa operação mentalmente como “x é igual a 100”, mas não o faça. Como já vimos, `=` é usado para atribuir um valor a uma variável e não para indicar igualdade. Se você se deparar com `x = 100`, pense “x está recebendo o valor 100”. O operador de comparação `==` é que é usado para indicar igualdade, logo, só quando você se deparar com `x == 100` é que poderá pensar “x é igual a 100”.



## Operadores lógicos

Os **operadores lógicos** são mais uma categoria de operadores em Python. Como os operadores de comparação, os operadores lógicos também apresentam como resultado **True** ou **False**.

Operador	Significado	Exemplo	Resultado
<b>and</b>	e	True and True	True
<b>or</b>	ou	True or False	True
<b>not</b>	não	not True	False

A palavra-chave **and** do Python recebe duas expressões e retorna **True** quando ambas apresentam como resultado **True**. Se alguma das expressões for **False**, o operador retornará **False**: # <http://tinyurl.com/zdqghb2>

```
1 == 1 and 2 == 2
```

```
>> True
```

```
# http://tinyurl.com/zkp2jzy
```

```
1 == 2 and 2 == 2
```

```
>> False
```

```
# http://tinyurl.com/honkev6
```

```
1 == 2 and 2 == 1
```

```
>> False
```

```
# http://tinyurl.com/zjrxsrc
```

```
2 == 1 and 1 == 1
```

```
>> False
```

Você pode usar a palavra-chave **and** várias vezes em uma instrução: # <http://tinyurl.com/zpvk56u>

```
1 == 1 and 10 != 2 and 2 < 10
```

```
>> True
```

A palavra-chave **or** recebe duas ou mais expressões e é igual a **True** quando uma das expressões é **True**: # <http://tinyurl.com/hosuh7c>

```
1==1 or 1==2
```

```
>> True
```

```
# http://tinyurl.com/zj6q8h9
```

```
1==1 or 2==2
```

```
>> True
```

```
# http://tinyurl.com/j8ngufo
```

```
1==2 or 2==1
```

```
>> False
```

```
# http://tinyurl.com/z728zxz
```

```
2==1 or 1==2
```

```
>> False
```

Como com **and**, você pode usar várias palavras-chave **or** em uma instrução: # <http://tinyurl.com/ja9mech>

```
1==1 or 1==2 or 1==3
```

```
>> True
```

Essa expressão é avaliada como **True** porque **1==1** é igual a **True**, ainda que o restante da expressão seja avaliado como **False**.

Inserir a palavra-chave **not** na frente de uma expressão altera o resultado da avaliação para o oposto do que de outra forma ele teria sido. Se a expressão seria avaliada como **True**, seu resultado será **False** se ela for precedida por **not**: # <http://tinyurl.com/h45eq6v>

```
not 1 == 1
```

```
>> False
```

```
# http://tinyurl.com/gsqj6og
```

```
not 1 == 2
```

```
>> True
```

## Instruções condicionais

As palavras-chave **if**, **elif** e **else** são usadas em **instruções condicionais**. As instruções condicionais são um tipo de **estrutura de controle**: um bloco de código que toma decisões analisando os valores de variáveis. Uma instrução condicional é um código que pode executar outro código condicionalmente. Aqui está um exemplo em **pseudocódigo** (uma notação parecida com código, usada para ilustrar um exemplo) para mostrar como funciona: # Não execute

```
If (expressão) Then (code_area1)
```

```
Else
```

```
(code_area2)
```

Esse pseudocódigo mostra que você pode definir duas instruções condicionais para funcionarem em conjunto. Se a expressão definida na primeira instrução condicional for **True**, todo o código de **code\_area1** será executado. Se a expressão definida na primeira instrução condicional for **False**, o código de **code\_area2** é que será executado. A primeira parte do exemplo chama-se instrução **if** e a segunda chama-se instrução **else**. Juntas, elas formam uma instrução **if-else**: uma maneira de os programadores dizerem “se isso ocorrer, faça isso; caso contrário, faça aquilo”. Aqui está um exemplo de uma instrução **if-else** em Python: ❶ # <http://tinyurl.com/htvy6g3>

```
❷
❸ home = "America"
❹ if home == "America": ❺ print("Hello, America!") ❻ else:
❼ print("Hello, World!") >> Hello, America!
```

As linhas ❹ e ❺ formam uma instrução **if**. A instrução **if** é composta de uma linha de código que começa com a palavra-chave **if**, seguida de uma expressão, dois pontos, uma indentação e uma ou mais linhas de código para serem executadas se a expressão da primeira linha for avaliada como **True**. As linhas ❻ e ❼ formam uma instrução **else**. A instrução **else** começa com a palavra-chave **else**, seguida de dois pontos, uma indentação e uma ou mais linhas de código para serem executadas se a expressão da instrução **if** for avaliada como **False**.

Juntas, elas formam uma instrução **if-else**. Esse exemplo exhibe **Hello, America!**, porque a expressão da instrução **if** é avaliada como **True**. Se você alterar a variável **home** para **Canada**, a expressão da instrução **if** será avaliada como **False**, o código da instrução **else** será executado e o programa exibirá **Hello, World!**.

```
# http://tinyurl.com/jytyg5x
```

```
home = "Canada"
if home == "America":
    print("Hello, America!")
else:
    print("Hello, World!")
>> Hello, World!
```

Você pode usar uma instrução **if** autônoma: # <http://tinyurl.com/jyg7dd2>

```
home = "America"
if home == "America":
    print("Hello, America!")
>> Hello, America!
```

E pode ter várias instruções **if** em sequência: # <http://tinyurl.com/z24ckye>

```
x = 2
if x == 2:
    print("The number is 2.")
if x % 2 == 0:
    print("The number is even.")
if x % 2 != 0:
    print("The number is odd.")
```

```
>> The number is 2.
```

```
>> The number is even.
```

Cada instrução **if** executará seu código somente se sua expressão for avaliada como **True**. Nesse caso, as duas primeiras expressões são avaliadas como **True**, logo, seu código é executado, mas a terceira expressão é avaliada como **False** e seu código não é executado.

Se você quiser complicar pode até mesmo inserir uma instrução **if** dentro de outra instrução **if** (isso se chama aninhar): # <http://tinyurl.com/zrodgne>

```
x = 10
y = 11

if x == 10:
if y == 11:
    print(x + y)
```

```
>> 21
```

Nesse exemplo, o resultado de **x + y** só será exibido se as expressões das duas instruções **if** forem avaliadas como **True**. Não é possível usar uma instrução **else** de forma autônoma; elas só podem ser usadas no fim de uma instrução **if-else**.

Você pode usar a palavra-chave **elif** para criar instruções **elif**. **elif** é a abreviação de **else if** e as instruções **elif** podem ser adicionadas sem restrições a uma instrução **if-else** para permitir que ela tome mais decisões.

Se uma instrução **if-else** tiver instruções **elif**, a expressão da instrução **if** será avaliada primeiro. Se a expressão dessa instrução for avaliada como **True**, só seu código será executado. No entanto, se for avaliada como **False**, cada uma das instruções **elif** consecutivas será avaliada. Assim que uma expressão de uma instrução **elif** for avaliada como **True**, somente seu código será executado. Se nenhuma das instruções **elif** for avaliada como **True**, o código da instrução **else** será executado. Aqui está um exemplo de uma instrução **if-else** com instruções **elif**: # <http://tinyurl.com/jpr265j>

```
home = "Thailand"
if home == "Japan":
    print("Hello, Japan!")
```

```
elif home == "Thailand":  
    print("Hello, Thailand!")  
elif home == "India":  
    print("Hello, India!")  
elif home == "China":  
    print("Hello, China!")  
else:  
    print("Hello, World!")
```

```
>> Hello, Thailand!
```

A seguir temos um exemplo em que nenhuma das expressões das instruções **elif** são avaliadas como **True** e o código da instrução **else** é executado: # <http://tinyurl.com/zdvuhs>

```
home = "Mars"  
if home == "America":  
    print("Hello, America!")  
elif home == "Canada":  
    print("Hello, Canada!")  
elif home == "Thailand":  
    print("Hello, Thailand!")  
elif home == "Mexico":  
    print("Hello, Mexico!")  
else:  
    print("Hello, World!")
```

```
>> Hello, World!
```

Para concluir, você pode ter várias instruções **if** e **elif** em sequência: # <http://tinyurl.com/hzyxgf4>

```
x = 100  
if x == 10:  
    print("10!")  
elif x == 20:  
    print("20!")  
else:  
    print("I don't know!")
```

```
if x == 100:  
    print("x is 100!")
```

```
if x % 2 == 0:  
    print("x is even!")  
else:  
    print("x is odd!")
```

```
>> I don't know!  
>> x is 100!  
>> x is even!
```

## Instruções

**Instrução** é um termo técnico que descreve várias partes da linguagem Python. Podemos considerar uma instrução Python como um comando ou um cálculo. Nesta seção, examinaremos detalhadamente a sintaxe das instruções. Não se preocupe se inicialmente parecer confuso, pois começará a fazer mais sentido conforme você for praticando o uso da linguagem e o ajudará a entender muitos conceitos de programação.

O Python tem dois tipos de instruções: **simples** e **compostas**. As instruções simples podem ser expressas em uma única linha de código, enquanto geralmente as instruções compostas se estendem por várias linhas. Aqui estão alguns exemplos de instruções simples: # <http://tinyurl.com/jrowero>

```
print("Hello, World!")  
>> Hello, World!  
# http://tinyurl.com/h2y549y
```

```
2 + 2  
>> 4
```

As instruções **if**, as instruções **if-else** e o primeiro programa que você escreveu neste capítulo para exibir **Hello, World!** 100 vezes são exemplos de instruções compostas.

As instruções compostas contêm uma ou mais **cláusulas**. Uma cláusula é composta de duas ou mais linhas de código: um **cabeçalho** seguido de um bloco de instruções. O cabeçalho é uma linha de código em uma cláusula que contém uma palavra-chave, seguida de dois pontos (:) e uma sequência de uma ou mais linhas de código indentado. Depois da indentação, existem um ou mais blocos. Um bloco é apenas uma linha de código em uma cláusula. O cabeçalho controla os blocos. Seu programa que exibe **Hello, World!** 100 vezes contém uma única instrução composta: # <http://tinyurl.com/zfz3eel>

```
for i in range(100):  
    print("Hello, World!")  
>> Hello, World!  
>> Hello, World!  
>> Hello, World!
```

...

A primeira linha do programa é o cabeçalho. Ele é composto de uma palavra-chave **for** seguida de dois pontos (:). Após a indentação há um bloco de instruções – **print("Hello, World!")**. Nesse caso, o cabeçalho usa o bloco para exibir **Hello, World!** 100 vezes. O código desse exemplo chama-se **loop**, uma estrutura sobre a qual você aprenderá mais no Capítulo 7. Esse código só tem uma cláusula.

Uma instrução composta pode conter várias cláusulas. Você já viu isso nas instruções **if-else**. Sempre que uma instrução **if** é seguida de uma instrução **else**, o resultado é uma instrução composta com várias cláusulas. Quando uma instrução composta tem várias cláusulas, as cláusulas de cabeçalho operam em conjunto. No caso de uma instrução composta **if-else**, quando a instrução **if** é avaliada como **True**, os blocos dessa instrução é que são executados em vez dos contidos na instrução **else**. Quando a instrução **if** é avaliada como **False**, seus blocos de instruções não são executados e os blocos da instrução **else** é que o são. O último exemplo da seção anterior inclui três instruções compostas: # <http://tinyurl.com/hpwkdo4>

```
x = 100
if x == 10:
    print("10!")
elif x == 20:
    print("20!")
else:
    print("I don't know!")
```

```
if x == 100:
    print("x is 100!")
```

```
if x % 2 == 0:
    print("x is even!")
else:
    print("x is odd!")
```

```
>> I don't know!
>> x is 100!
>> x is even!
```

A primeira instrução composta tem três cláusulas, a segunda tem uma cláusula e a última tem duas cláusulas.

Uma última observação que devo fazer sobre as instruções é que pode haver espaços entre elas. Os espaços entre as instruções não afetam o código. Às vezes espaços são usados entre as instruções para tornar o código mais legível: #

<http://tinyurl.com/zlgcwoc>

```
print("Michael")
```

```
print("Jordan")
```

```
>> Michael
```

```
>> Jordan
```

## Vocabulário

**Bloco de instruções:** Uma linha de código em uma cláusula controlada por um cabeçalho.

**Bool:** Tipo de dado de objetos booleanos.

**Booleano:** Objeto com o tipo de dado **bool**. Seu valor é **True** ou **False**.

**Cabeçalho:** Linha de código em uma cláusula contendo uma palavra-chave, seguida de dois pontos e uma sequência de uma ou mais linhas de código indentado.

**Caractere:** Um único símbolo, como a ou 1.

**Cláusula:** Blocos que constituem as instruções compostas. Uma cláusula é composta de duas ou mais linhas de código: um cabeçalho seguido de um ou mais blocos de instruções.

**Comentário:** Linha (ou parte de uma linha) de código escrita em inglês (ou em outro idioma) precedida por um símbolo que informa à linguagem de programação usada que ela deve ignorar essa linha (ou parte dessa linha).

**Constante:** Um valor que nunca muda.

**Decrementar:** Diminuir o valor de uma variável.

**Erro de sintaxe:** Erro de programação fatal causado pela violação da sintaxe de uma linguagem de programação.

**Estrutura de controle:** Bloco de código que toma decisões analisando os valores das variáveis.

**Exceção:** Erro de programação não fatal.

**Expressão:** Código que tem um operador com um operando de cada lado.

**Incrementar:** Aumentar o valor de uma variável.

**Instrução composta:** Instrução que geralmente se estende por várias linhas de código.

**Instrução condicional:** Código que pode executar um código adicional



condicionalmente.

**Instrução `elif`:** Instruções que podem ser adicionados sem restrições a uma instrução `if-else` para permitir que ela tome mais decisões.

**Instrução `else`:** Segunda parte de uma instrução `if-else`.

**Instrução `if`:** Primeira parte de uma instrução `if-else`.

**Instrução `if-else`:** Uma maneira dos programadores dizerem “se isto ocorrer, faça isso; caso contrário, faça aquilo”.

**Instrução `simples`:** Instrução que pode ser expressa em uma única linha de código.

**`Int`:** Tipo de dado dos números inteiros.

**`Integer`:** Objeto com o tipo de dado `int`. Seu valor é um número inteiro.

**`Float`:** Tipo de dado dos números decimais.

**`None`:** Objeto com o tipo de dado `NoneType`. Seu valor é sempre `None`.

**`NoneType`:** Tipo de dado de objetos `None`.

**Número de ponto flutuante:** Objeto com o tipo de dado `float`. Seu valor é um número decimal.

**Objeto:** Valor de dados em Python com três propriedades: uma identidade, um tipo de dado e um valor.

**Operador:** Símbolo usado com operandos em uma expressão.

**Operador aritmético:** Categoria dos operadores usados em expressões aritméticas.

**Operador de atribuição:** O sinal `=` em Python.

**Operador de comparação:** Categoria dos operadores usados em uma expressão que é avaliada como `True` ou `False`.

**Operador lógico:** Categoria dos operadores que avaliam duas expressões e retornam `True` ou `False`.

**Operando:** O valor existente em cada um dos lados de um operador.

**Ordem das operações:** Conjunto de regras usadas em cálculos matemáticos para a avaliação de uma expressão.

**Palavra-chave:** Palavra com um significado especial em uma linguagem de programação. Você pode ver todas as palavras-chave do Python em <http://theselftaughtprogrammer.io/keywords>.

**Pseudocódigo:** Notação que parece um código e que é usada para ilustrar um exemplo.

**Sintaxe:** Conjunto de regras, princípios e processos que controlam a estrutura das frases de determinada linguagem, especificamente a ordem das palavras.<sup>3</sup>

**Str:** Tipo de dado de uma string.

**String:** Objeto com o tipo de dado **str**. Seu valor é uma sequência de um ou mais caracteres entre aspas.

**Tipo de dado:** Uma categoria de dados.

**Variável:** Nome ao qual é atribuído um valor com o uso do operador de atribuição.

## Desafios

1. Exiba três strings diferentes.
2. Escreva um programa que exiba uma mensagem se uma variável for menor do que 10 e outra mensagem se a variável for maior ou igual a 10.
3. Escreva um programa que exiba uma mensagem se uma variável for menor ou igual a 10, outra mensagem se a variável for maior do que 10, mas menor ou igual a 25, e ainda outra mensagem se a variável for maior do que 25.
4. Crie um programa que divida duas variáveis e exiba o resto.
5. Crie um programa que receba duas variáveis, as divida, e exiba o quociente.
6. Escreva um programa com uma variável **age** que receba um inteiro e exiba strings diferentes dependendo de que inteiro **age** receber.

Soluções: <http://tinyurl.com/zx7o2v9>.

---

<sup>1</sup> <https://en.wikipedia.org/wiki/Syntax>.

<sup>2</sup> N.T.: Na operação floored quotient, ou floor division, o operador divide o primeiro operando pelo segundo e havendo casas decimais o resultado é arredondado para baixo, para o número inteiro mais próximo.

<sup>3</sup> <https://en.wikipedia.org/wiki/Syntax>

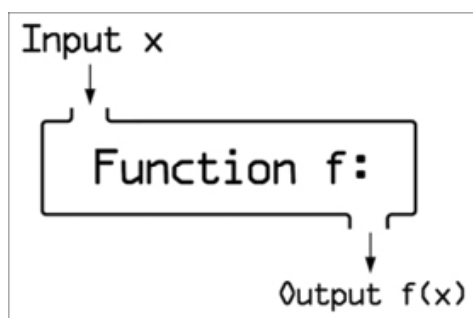
## CAPÍTULO 4

### Funções

*“As funções devem realizar uma tarefa e devem realizá-la bem. Elas só devem se ocupar de uma única tarefa.”*

#### – Robert C. Martin

Neste capítulo, você conhecerá as **funções**: instruções compostas que podem receber entradas, executar operações e retornar uma saída. As funções nos permitem definir e reutilizar funcionalidades em nossos programas.



### Representação de conceitos

De agora em diante, usarei uma nova **convenção** (uma maneira acordada de se fazer algo) para explicar os conceitos de programação. Aqui está um exemplo da convenção que usarei: `print("[o_que_será_impresso]")`, que ilustra como usar a função `print`.

Combinei o código Python com colchetes contendo uma descrição para ilustrar esse conceito. Quando eu fornecer um exemplo como esse, ele será composto de código Python válido, exceto pelos colchetes e pelo texto existente dentro deles, que precisará ser substituído por um código válido quando você seguir o exemplo. O texto dentro dos colchetes é uma dica do código que você deve usar para substituí-lo. O Python usa colchetes em sua sintaxe, logo, usarei colchetes duplos quando eles precisarem permanecer no código.

### Funções

**Chamar** uma função significa fornecer a entrada necessária para ela executar suas instruções e retornar uma saída. Cada entrada de uma função é um **parâmetro**. Quando fornecemos um parâmetro para uma função, isso se chama “passar” um parâmetro.

As funções em Python são semelhantes a funções matemáticas. Se você não se lembra das funções da álgebra, veja um exemplo: `# Não execute`

```
f(x) = x * 2
```

O lado esquerdo da instrução acima define uma função **f**, que recebe um parâmetro **x**. O lado direito da instrução é a definição da função que usa o parâmetro passado em (**x**) para fazer um cálculo e retornar o resultado (a saída). Nesse caso, o valor da função é definido como o valor do parâmetro multiplicado por dois.

Tanto em Python quanto na álgebra chamamos uma função com a sintaxe a seguir:

`[nome_função]([parâmetros_separados_por_vírgulas])` Ela é chamada com a inserção de parâmetros após seu nome. Os parâmetros são inseridos dentro dos parênteses e separados com vírgulas. Para uma função matemática **f**, definida como  $f(x) = 2 * x$ , o valor de **f(2)** é 4 e o valor de **f(10)** é 20.

## Definição de funções

Para criar uma função em Python, escolhemos um nome para a função, definimos seus parâmetros, estabelecemos o que ela faz e definimos que valor ela retorna. Aqui está a sintaxe para a definição de uma função: # Não execute

```
def [nome_função]([parâmetros]): [definição_função]
```

Sua função matemática  $f(x) = x * 2$  ficaria assim em Python: # <http://tinyurl.com/j9dctl>

```
def f(x):  
    return x * 2
```

A palavra-chave **def** informa ao Python que estamos definindo uma função. Depois de **def**, é preciso especificar o nome da função, que deve seguir as mesmas regras usadas para nomes de variáveis. Por convenção, você nunca deve usar letras maiúsculas em um nome de função e as palavras devem ser separadas por underscores: **desta forma**.

Uma vez que você tiver nomeado sua função, insira um par de parênteses depois dela. Dentro dos parênteses, defina o(s) parâmetro(s) que deseja que a função aceite.

Depois dos parênteses insira dois pontos e comece uma nova linha indentada em quatro espaços (como em qualquer instrução composta). Qualquer código indentado em quatro espaços depois dos dois pontos será a definição da função. Nesse caso, a definição tem apenas uma linha – **return x \* 2**. A palavra-chave **return** define o valor que uma função exhibe quando chamada, o que é denominado de valor que a função retorna.

Você pode usar a sintaxe `[nome_função]([parâmetros_separados_por_vírgulas])` para

chamar uma função em Python. Aqui está um exemplo da chamada da função `f` do exemplo anterior com `2` como parâmetro: # <http://tinyurl.com/zheas3d>

```
# Continuação do  
# último exemplo
```

```
f(2)
```

O console não exibiu nada. Você pode salvar a saída de sua função em uma variável e passá-la para a função `print`: # <http://tinyurl.com/gspjcgj>

```
# Continuação do  
# último exemplo
```

```
result = f(2)  
print(result)
```

```
>> 4
```

É possível salvar o resultado que a função retorna em uma variável sempre que você precisar usar o valor posteriormente em seu programa: # <http://tinyurl.com/znqp8fk>

```
def f(x):  
    return x + 1
```

```
z = f(4)
```

```
if z == 5:  
    print("z is 5")  
else:  
    print("z is not 5")
```

```
>> z is 5
```

Uma função pode ter um parâmetro, vários parâmetros ou nenhum parâmetro. Para definir uma função que não demande parâmetros, deixe o local dos parâmetros vazio ao defini-la: # <http://tinyurl.com/htk7tr6>

```
def f():  
    return 1 + 1
```

```
result = f()  
print(result)
```

```
>> 2
```

Se quiser que sua função aceite mais de um parâmetro, você deve separar os parâmetros dentro dos parênteses com vírgulas: # <http://tinyurl.com/ggmkft7>

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

```
result = f(1, 2, 3)  
print(result)
```

```
>> 6
```

Para concluir, uma função não precisa incluir uma instrução **return**. Se a função não tiver uma instrução **return**, ela retornará **None**: # <http://tinyurl.com/j8qygov>

```
def f():  
    z = 1 + 1
```

```
result = f()  
print(result)
```

```
>> None
```

## Funções internas

O Python vem com uma biblioteca de funções já incorporadas à linguagem de programação que são chamadas de **funções internas**. Elas executam vários tipos de cálculos e tarefas e estão prontas para ser usadas sem que seja necessário nenhum esforço de nossa parte. Você já viu um exemplo de uma função interna: o primeiro programa que escreveu usou a função **print** para exibir "Hello, World!".

**len** é outra função interna. Ela retorna o tamanho de um objeto – por exemplo, o tamanho de uma string (seu número de caracteres): # <http://tinyurl.com/zfkzqw6>

```
len("Monty")
```

```
>> 5
```

```
# http://tinyurl.com/h75c3cf
```

```
len("Python")
```

```
>> 6
```

A função interna **str** recebe um objeto e retorna um novo objeto com o tipo de dado **str**. Por exemplo, você pode usar **str** para converter um inteiro em uma string: # <http://tinyurl.com/juzxg2z>

```
str(100)
```

```
>> '100'
```

A função **int** recebe um objeto e retorna um novo objeto integer: # <http://tinyurl.com/j42qhkf>

```
int("1")
```

```
>> 1
```

A função **float** recebe um objeto como parâmetro e retorna um objeto de número de ponto flutuante: # <http://tinyurl.com/hnk8gh2>

```
float(100)
```

```
>> 100.0
```

O parâmetro que você passar para uma função **str**, **int** ou **float** tem de poder ser convertido para uma string, um inteiro ou um número de ponto flutuante. A função **str** pode aceitar a maioria dos objetos como parâmetro, mas a função **int** só pode aceitar uma string com um número ou um objeto de ponto flutuante. A função **float** só pode receber uma string com um número ou um objeto integer: # <http://tinyurl.com/jcchmlx>

```
int("110")
```

```
int(20.54)
```

```
float("16.4")
```

```
float(99)
```

```
>> 110
```

```
>> 20
```

```
>> 16.4
```

```
>> 99.0
```

O Python lançará uma exceção se você tentar passar para a função **int** ou **float** um parâmetro que ela não possa converter em um inteiro ou em um número de ponto flutuante: # <http://tinyurl.com/zseo2ls>

```
int("Prince")
```

```
>> ValueError: invalid literal for int() with base 10: 'Prince'
```

**input** é uma função interna que coleta informações da pessoa que está usando o programa: # <http://tinyurl.com/zynprpg>

```
age = input("Enter your age:")
```

```
int_age = int(age)
```

```
if int_age < 21:
print("You are young!")
else:
print("Wow, you are old!")
```

>> Enter your age: A função `input` recebe uma string como parâmetro e a exibe no shell para a pessoa que está usando o programa. O usuário poderá então digitar uma resposta no shell e você poderá salvá-la em uma variável - nesse caso, na variável `age`.

Em seguida, use a função `int` para alterar a variável `age` de uma string para um inteiro. A função `input` coleta os dados do usuário como uma `str`, mas você quer que sua variável seja um `int` para poder compará-lo com outros inteiros. Uma vez que você tiver um inteiro, sua instrução `if-else` determinará que mensagem será exibida para o usuário, dependendo do que ele tiver digitado no shell. Se o usuário digitar um número menor do que `21`, será exibido `You are young!`. Se ele digitar um número maior do que `21`, será exibido `Wow, you are old!`.

## Reutilização de funções

As funções não são usadas apenas para fazer cálculos e retornar valores. Elas podem encapsular as funcionalidades que você quiser reutilizar: # <http://tinyurl.com/zhy8y4m>

```
def even_odd(x):
if x % 2 == 0:
print("even")
else:
print("odd")
```

```
even_odd(2)
even_odd(3)
```

```
>> even
>> odd
```

Você não definiu um valor para sua função retornar, mas mesmo assim ela é útil. Ela verifica se `x % 2 == 0` e exibe se `x` é par (`even`) ou ímpar (`odd`).

As funções permitem escrever menos código porque você pode reutilizar funcionalidades. Veja um exemplo de um programa escrito sem funções: # <http://tinyurl.com/jk8lugl>

```
n = input("type a number:")
n = int(n)
```



```
if n % 2 == 0:
print("n is even.")
else:
print("n is odd.")
```

```
n = input("type a number:")
n = int(n)
if n % 2 == 0:
print("n is even.")
else:
print("n is odd.")
```

```
n = input("type a number:")
n = int(n)
if n % 2 == 0:
print("n is even.")
else:
print("n is odd.")
```

>> type a number: Esse programa solicita ao usuário que insira um número três vezes. Em seguida, uma instrução if-else verifica se o número é par. Se for, "n is even" é exibido; caso contrário, ele exibe "n is odd".

O problema do programa é que ele repete o mesmo código três vezes. Você pode fazer com que esse programa fique mais curto e fácil de ler inserindo sua funcionalidade em uma função e chamando-a três vezes: # <http://tinyurl.com/zzn22mz>

```
def even_odd():
n = input("type a number:")
n = int(n)
if n % 2 == 0:
print("n is even.")
else:
print("n is odd.")
```

```
even_odd()
even_odd()
even_odd()
```

>> type a number: Esse novo programa tem a mesma funcionalidade do programa anterior, mas já que você a inseriu em uma função que pode ser chamada onde quer que seja necessário, o programa ficou muito mais curto e fácil de ler.

## Parâmetros obrigatórios e opcionais

Existem dois tipos de parâmetros que uma função pode aceitar. Os parâmetros que

you've seen up to now are called **mandatory parameters**. When a user calls a function, he needs to pass all the mandatory parameters or Python will raise an exception.

Python has another type of parameter – the **optional parameters** – that allows the caller of the function to pass a parameter if necessary, but it's not mandatory. If an optional parameter is not passed, the function will use its default value. Optional parameters are defined with the following syntax: `[nome_função] ([nome_parâmetro]=[valor_parâmetro])`. Like the mandatory parameters, the optional parameters must be separated by commas. Below is an example of a function that receives an optional parameter: [#http://tinyurl.com/h3ych4h](http://tinyurl.com/h3ych4h)

```
def f(x=2):  
    return x**x
```

```
print(f())  
print(f(4))
```

```
>> 4
```

```
>> 256
```

First, you called the function without passing a parameter. Since the parameter is optional, `x` received 2 automatically and the function returned 4.

Next, you called the function and passed 4 as a parameter. The function ignored the default value, `x` received 4 and the function returned 256. You can define a function that has both mandatory and optional parameters, but you must define all the mandatory parameters before the optional ones: [#http://tinyurl.com/hm5svn9](http://tinyurl.com/hm5svn9)

```
def add_it(x, y=10):  
    return x + y
```

```
result = add_it(2)  
print(result)
```

```
>> 12
```

## Escopo

Variables have an important property called **scope**. When we define a variable, its scope refers to the part of the program that can read it and write to it. Reading a variable means knowing its value. Writing a variable means changing its value. The scope of a variable is determined by where, in the program, it was defined. If you define a variable outside a function (or outside a class, which you'll learn about in Part II), it will have **global scope**: it can be read or written to

partir de qualquer local de seu programa. Uma variável com escopo global é chamada de **variável global**. Se você definir uma variável dentro de uma função (ou de uma classe), ela terá **escopo local**: seu programa só poderá lê-la ou gravá-la na função (ou na classe) dentro da qual ela foi definida. As variáveis a seguir têm escopo global: # <http://tinyurl.com/zhmxnqt>

```
x = 1
y = 2
z = 3
```

Essas variáveis não foram definidas dentro de uma função (ou classe), portanto, têm escopo global. Isso significa que você poderá lê-las ou gravá-las a partir de qualquer local – inclusive dentro de uma função: # <http://tinyurl.com/hgvnj4p>

```
x = 1
y = 2
z = 3
```

```
def f():
    print(x)
    print(y)
    print(z)
```

```
f()
>> 1
>> 2
>> 3
```

Se você definir essas mesmas variáveis dentro de uma função, só poderá lê-las ou gravá-las dentro dessa função. Se tentar acessá-las fora da função na qual elas foram definidas, o Python lançará uma exceção: # <http://tinyurl.com/znka93k>

```
def f():
    x = 1
    y = 2
    z = 3
```

```
print(x)
print(y)
print(z)
```

```
>> NameError: name 'x' is not defined
Se você definir as variáveis dentro da
função, seu código funcionará:
```

```
# http://tinyurl.com/z2k3jds
```

```
def f():
```

```
x = 1
y = 2
z = 3
print(x)
print(y)
print(z)
```

```
f()
>> 1
>> 2
>> 3
```

Digamos que uma variável fosse definida dentro de uma função. Tentar usá-la fora da função será semelhante a usar uma variável que ainda não foi definida, o que fará o Python lançar a mesma exceção: # <http://tinyurl.com/zn8zjmr>

```
if x > 100:
    print("x is > 100")
```

```
>> NameError: name 'x' is not defined
```

Você pode atribuir um valor a uma variável global a partir de qualquer lugar, mas atribuir a uma variável global dentro de um escopo local demanda uma etapa adicional. Você precisa usar explicitamente a palavra-chave `global`, seguida da variável que deseja alterar. O Python nos faz executar essa etapa adicional para assegurar que se você definir a variável `x` dentro de uma função, o valor de alguma variável definida anteriormente não seja alterado acidentalmente fora da função. Aqui está um exemplo de atribuição a uma variável global de dentro de uma função: # <http://tinyurl.com/zclmda7>

```
x = 100
```

```
def f():
    global x
    x += 1
    print(x)
```

```
f()
>> 101
```

Sem escopo, você poderia acessar todas as variáveis em qualquer local de seu programa, o que poderia ser problemático. Se você tiver um programa grande, e escrever uma função que use a variável `x`, pode alterar acidentalmente o valor de uma variável também chamada `x` definida anteriormente em algum lugar do programa. Erros como esse podem alterar o comportamento do programa e causar erros ou resultados inesperados. Quanto maior ficar seu programa, e quanto maior for o número de variáveis que ele tiver, mais probabilidades haverá de isso ocorrer.

## Manipulação de exceções

Se você confiar na entrada do usuário fornecida pela função `input`, não estará controlando a entrada de seu programa – o usuário é que estará no controle e essa entrada pode causar um erro. Por exemplo, digamos que você escrevesse um programa que coletasse dois números com o usuário e exibisse o resultado do primeiro número dividido pelo segundo número: # <http://tinyurl.com/jcg5qwp>

```
a = input("type a number:")
b = input("type another:")
a = int(a)
b = int(b)
print(a / b)
```

```
>> type a number: >> 10
>> type another:
>> 5
>> 2
```

Seu programa parece funcionar. No entanto, você terá um problema se o usuário fornecer `0` como segundo número: # <http://tinyurl.com/ztpcjs4>

```
a = input("type a number:")
b = input("type another:")
a = int(a)
b = int(b)
print(a / b)
```

```
>> type a number: >> 10
>> type another:
>> 0
```

```
>> ZeroDivisionError: integer division or modulo by zero
Você não pode apenas esperar que alguém que estiver usando esse programa não insira 0 como segundo número. Uma maneira de resolver isso é usando a manipulação de exceções, que permite verificar condições de erro, “capturar” exceções se elas ocorrerem e decidir como proceder.
```

As palavras-chave `try` e `except` são usadas para a manipulação de exceções. Quando você alterar esse programa para que use a manipulação de exceções, se um usuário inserir `0` como segundo número, em vez de lançar uma exceção, o programa poderia exibir uma mensagem solicitando que ele não insira `0`.

Em Python, cada exceção é um objeto, portanto, você pode usá-las em seus programas. É possível ver uma lista das exceções internas aqui: [https://www.tutorialspoint.com/python/standard\\_exceptions.htm](https://www.tutorialspoint.com/python/standard_exceptions.htm). Se você estiver em uma situação na qual ache que seu código pode lançar uma exceção, use uma

instrução composta com as palavras-chave **try** e **except** para capturá-la.

A cláusula **try** contém o erro que poderia ocorrer. A cláusula **except** contém o código que só será executado se a exceção da cláusula **try** ocorrer. Aqui está um exemplo de como você pode usar a manipulação de exceções em seu programa para que, se um usuário inserir **0** como segundo número, o programa não quebrará: # <http://tinyurl.com/j2scn4f>

```
a = input("type a number:")
b = input("type another:")
a = int(a)
b = int(b)
try:
    print(a / b)
except ZeroDivisionError:
    print("b cannot be zero.")
```

```
>> type a number: >> 10
>> type another:
>> 0
>> b cannot be zero.
```

Se o usuário inserir algo diferente de **0** para **b**, o código do bloco **try** será executado e o bloco **except** não fará nada. Se o usuário inserir **0** para **b**, em vez de ser lançada uma exceção, o código do bloco **except** será executado e o programa exibirá "**b cannot be zero**" (b não pode ser zero).

Seu programa também quebrará se o usuário inserir uma string que o Python não possa converter para um inteiro: a = input("type a number:")

```
b = input("type another:")
a = int(a)
b = int(b)
try:
    print(a / b)
except ZeroDivisionError:
    print("b cannot be zero.")
```

```
>> type a number: >> Hundo
>> type another:
>> Million
>> ValueError: invalid literal for int() with base 10: 'Hundo'
```

Você pode corrigir isso movendo para dentro da instrução **try** a parte do programa que coleta entradas e fazendo a instrução **except** ficar atenta para a ocorrência de duas exceções: **ZeroDivisionError** e **ValueError**. **ValueError** ocorre quando fornecemos uma entrada errada para as funções internas **int**, **string** ou **float**. Você pode fazer com que sua instrução **except** capture duas exceções adicionando

parênteses a **except** e separando as exceções com uma vírgula: # <http://tinyurl.com/jlus42v>

```
try:
a = input("type a number:")
b = input("type another:")
a = int(a)
b = int(b)
print(a / b)
except(ZeroDivisionError,
ValueError):
print("Invalid input.")

>> type a number: >> Hundo
>> type another:
>> Million
>> Invalid input.
```

Variáveis definidas em uma instrução **try** não devem ser usadas em uma instrução **except**, porque uma exceção pode ocorrer antes de a variável ser definida e essa exceção será lançada dentro da instrução **except** quando você tentar usá-la: # <http://tinyurl.com/hockur5>

```
try:
10 / 0
c = "I will never get defined."
except ZeroDivisionError:
print(c)

>> NameError: name 'c' is not defined
```

## Docstrings

Ao definirmos uma função com parâmetros, pode ocorrer de eles precisarem ser de um tipo de dado específico para a função funcionar. Como informar isso para quem chamar a função? Na criação de uma função, é boa prática deixar um comentário chamado **docstring** no início informando que tipo de dado cada parâmetro precisa ter. As docstrings explicam o que a função faz e documentam de que tipos de parâmetros ela precisa: # <http://tinyurl.com/zhahdcg>

```
def add(x, y):
"""
Retorna x + y.
:param x: int.
:param y: int.
```

```
:return: int soma de x e y.  
"""  
return x + y
```

A primeira linha da docstring explica claramente o que sua função faz e, portanto, quando outros desenvolvedores reutilizarem sua função ou método, eles não precisarão percorrer todo o código para descobrir sua finalidade. As outras linhas da docstring listam os parâmetros da função, os tipos dos parâmetros e o que ela retorna. As docstrings o ajudarão a programar com mais rapidez porque você poderá ler uma docstring para saber o que uma função faz, em vez de precisar ler o código todo. Para manter concisos os exemplos deste livro, omiti as docstrings que normalmente incluiria. Quando escrevo código, geralmente incluo docstrings para tornar meu código fácil de entender para todos que precisarem dele no futuro.

## Só use uma variável quando necessário

Só salve dados em uma variável se for usá-la posteriormente. Por exemplo, não armazene um inteiro em uma variável apenas para exibi-lo: # <http://tinyurl.com/zptktex>

```
x = 100  
print(x)  
  
>> 100
```

Em vez disso, passe o inteiro diretamente para a função `print`: # <http://tinyurl.com/hmwr4kd>

```
print(100)  
  
>> 100
```

Virolei essa regra em muitos exemplos deste livro para tornar o que estava fazendo fácil para você entender. Você não precisa fazer o mesmo quando estiver escrevendo códigos.

## Vocabulário

**Convenção:** Maneira acordada de se fazer algo.

**Chamar:** Fornecer para a função a entrada da qual ela precisa para executar suas instruções e retornar uma saída.

**Docstring:** As docstrings explicam o que uma função faz e documenta que tipos de parâmetros ela recebe.

**Escopo:** Local onde uma variável pode ser lida ou gravada.

**Escopo global:** Escopo de uma variável que pode ser lida ou gravada a partir de



qualquer local em um programa.

**Escopo local:** Escopo de uma variável que só pode ser lida ou gravada a partir da função (ou classe) dentro da qual ela foi definida.

**Funções:** Instruções compostas que podem receber entradas, executar operações e retornar uma saída.

**Função interna:** Uma função que vem com o Python.

**Manipulação de exceções:** Conceito de programação que permite verificar condições de erro, “capturar” exceções se elas ocorrerem e decidir como proceder.

**Parâmetro:** Dado passado para uma função.

**Parâmetro obrigatório:** Um parâmetro não opcional.

**Parâmetro opcional:** Um parâmetro não obrigatório.

**Variável global:** Variável com escopo global.

## Desafios

1. Escreva uma função que receba um número como entrada e retorne esse número ao quadrado.
2. Crie uma função que receba uma string como parâmetro e a exiba.
3. Escreva uma função que receba três parâmetros obrigatórios e dois parâmetros opcionais.
4. Escreva um programa com duas funções. A primeira função deve receber um inteiro como parâmetro e retornar o resultado do inteiro dividido por 2. A segunda função deve receber um inteiro como parâmetro e retornar o resultado do inteiro multiplicado por 4. Chame a primeira função, salve o resultado como uma variável e passe-a como parâmetro para a segunda função.
5. Escreva uma função que converta uma string em um **float** e retorne o resultado. Use a manipulação de exceções para capturar a exceção que pode ocorrer.
6. Adicione uma docstring a todas as funções que escreveu nos desafios 1-5.

Soluções: <http://tinyurl.com/hkzggrv>.

# CAPÍTULO 5

## Contêineres

*“O tolo duvida, o sábio pergunta.”*

### – Benjamin Disraeli

No Capítulo 3, você aprendeu como armazenar objetos em variáveis. Neste capítulo, você verá como armazenar objetos em contêineres. Os contêineres são como fichários: eles mantêm os dados organizados. Você conhecerá três contêineres muito usados: as listas, as tuplas e os dicionários.

## Métodos

No Capítulo 4, você conheceu as funções. O Python tem um conceito semelhante chamado de **método**. Os métodos são funções associadas a um tipo de dado específico. Eles executam código e podem retornar um resultado como uma função. Ao contrário da função, o método é chamado em um objeto. Também podemos passar parâmetros para eles. Aqui está um exemplo da chamada dos métodos **upper** e **replace** em uma string: # <http://tinyurl.com/zdllght>

```
"Hello".upper()
```

```
>> 'HELLO'
```

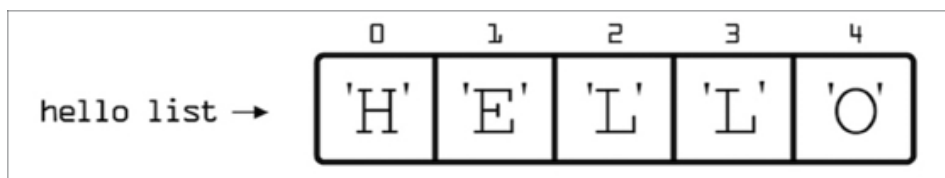
```
# http://tinyurl.com/hfgpst5
```

```
"Hello".replace("o", "@") >> 'Hell@'
```

Você aprenderá mais sobre os métodos na Parte II.

## Listas

Uma **lista** é um contêiner que armazena objetos em uma ordem específica.



As listas são representadas por colchetes. Existem duas sintaxes que você pode usar para criar uma lista. Você pode criar uma lista vazia com a função **list**: # <http://tinyurl.com/h4go6kg>

```
fruit = list()
```

```
fruit
```

```
>> []
```

Ou pode usar colchetes:

```
# http://tinyurl.com/jft8p7x
```

```
fruit = []
```

```
fruit
```

```
>> []
```

Você pode criar uma lista já contendo itens usando a segunda sintaxe [], e inserindo dentro dos colchetes cada item que deseja na lista, separando-os com vírgulas: # <http://tinyurl.com/h2y8nos>

```
fruit = ["Apple", "Orange", "Pear"]
```

```
fruit
```

```
>> ['Apple', 'Orange', 'Pear']
```

Existem três itens em sua lista: "Apple", "Orange" e "Pear". As listas armazenam os itens em ordem. A menos que você altere a ordem de sua lista, "Apple" será sempre o primeiro item, "Orange" será o segundo e "Pear", o terceiro. "Apple" se encontra no começo da lista e "Pear" está no final. Para adicionar um novo item a uma lista, use o método **append**: # <http://tinyurl.com/h9w3z2m>

```
fruit = ["Apple", "Orange", "Pear"]
```

```
fruit.append("Banana")
```

```
fruit.append("Peach")
```

```
fruit
```

```
>> ['Apple', 'Orange', 'Pear', 'Banana', 'Peach']
```

Agora cada objeto passado para o método **append** é um item em sua lista. **append** sempre adiciona um novo item ao fim da lista.

As listas não estão restritas ao armazenamento de strings – elas podem armazenar qualquer tipo de dado: # <http://tinyurl.com/zhpntsr>

```
random = []
```

```
random.append(True)
```

```
random.append(100)
```

```
random.append(1.1)
```

```
random.append("Hello")
```

```
random
```

```
>> [True, 100, 1.1, 'Hello']
```

As strings, as listas e as tuplas são **iteráveis**. Um objeto é iterável quando podemos acessar cada item usando um loop. Os objetos pelos quais podemos iterar são chamados de **iteráveis**. Cada item de um iterável tem um índice – um número que

representa a posição do item no iterável. O primeiro item de uma lista tem índice 0 e não 1.

No exemplo a seguir, "Apple" está no índice 0, "Orange" está no índice 1 e "Pear" está no índice 2: # <http://tinyurl.com/z8zzk8d>

```
fruit = ["Apple", "Orange", "Pear"]
```

Você pode recuperar um item com seu índice usando a sintaxe [*nome\_lista*][*índice*]: # <http://tinyurl.com/jqtlwvf>

```
fruit = ["Apple", "Orange", "Pear"]
```

```
fruit[0]
```

```
fruit[1]
```

```
fruit[2]
```

```
>> 'Apple'
```

```
>> 'Orange'
```

```
>> 'Pear'
```

Se você tentar acessar um índice que não existe, o Python lançará uma exceção: # <http://tinyurl.com/za3rv95>

```
colors = ["blue", "green", "yellow"]
```

```
colors[4]
```

```
>> IndexError: list index out of range
```

As listas são mutáveis. Um contêiner é mutável quando podemos adicionar ou remover objetos. Você pode alterar um item em uma lista atribuindo seu índice a um novo objeto: # <http://tinyurl.com/h4ahvf9>

```
colors = ["blue", "green", "yellow"]
```

```
colors
```

```
colors[2] = "red"
```

```
colors
```

```
>> ['blue', 'green', 'yellow']
```

```
>> ['blue', 'green', 'red']
```

Para remover o último item de uma lista, use o método `pop`: # <http://tinyurl.com/j52uvmq>

```
colors = ["blue", "green", "yellow"]
```

```
colors
```

```
item = colors.pop()
```

```
item
```

```
colors
```

```
>> ['blue', 'green', 'yellow']
```

```
>> 'yellow'
>> ['blue', 'green']
```

Não é possível usar **pop** em uma lista vazia. Se você tentar fazer isso, o Python lançará uma exceção.

Você pode combinar duas listas com o operador de adição (+): # <http://tinyurl.com/jjxnk4z>

```
colors1 = ["blue", "green", "yellow"]
colors2 = ["orange", "pink", "black"]
colors1 + colors2

>> ['blue', 'green', 'yellow', 'orange', 'pink', 'black']
```

Podemos verificar se um item existe em uma lista com a palavra-chave **in**: # <http://tinyurl.com/z4fnv39>

```
colors = ["blue", "green", "yellow"]
"green" in colors

>> True Use a palavra-chave not para verificar se um item não existe em uma
lista: # http://tinyurl.com/jqzk8pj
```

```
colors = ["blue", "green", "yellow"]
"black" not in colors

>> True Você pode saber o tamanho de uma lista (o número de itens que existem
nela) com a função len: # http://tinyurl.com/hhx6rx4
```

```
len(colors)

>> 3
```

Aqui está um exemplo de como você pode usar uma lista na prática:

```
# http://tinyurl.com/gq7yjr7
```

```
colors = ["purple",
"orange",
"green"]
```

```
guess = input("Guess a color:")
```

```
if guess in colors:
    print("You guessed correctly!")
else:
    print("Wrong! Try again.")
```

```
>> Guess a color: Sua lista colors contém várias strings que representam cores. 0
```

programa usa a função interna `input` para solicitar ao usuário que tente adivinhar uma cor, e a resposta é salva em uma variável. Se a resposta fizer parte da lista `colors`, o programa informará ao usuário que seu palpite estava correto. Caso contrário, ele solicitará outro palpite.

## Tuplas

Uma **tupla** é um contêiner que armazena objetos em uma ordem específica. Ao contrário das listas, as tuplas são **imutáveis**, o que significa que seu conteúdo não pode mudar. Uma vez que você criar uma tupla, não poderá modificar o valor de nenhum dos itens, adicionar novos itens ou remover itens. As tuplas são representadas com parênteses. Os itens da tupla precisam ser separados com vírgulas. Existem duas sintaxes para a criação de uma tupla: # <http://tinyurl.com/zo88eal>

```
my_tuple = tuple()
my_tuple
>> ()
```

E:

```
# http://tinyurl.com/zm3y26j.
```

```
my_tuple = ()
my_tuple
>> ()
```

Para adicionar objetos, crie uma tupla com a segunda sintaxe com cada item que deseja adicionar, separando-os com vírgulas: # <http://tinyurl.com/zlwwfe3>

```
rndm = ("M. Jackson", 1958, True)
rndm
```

```
>> ('M. Jackson', 1958, True) Mesmo se uma tupla só tiver um item, será preciso
inserir uma vírgula depois dele. Dessa forma, o Python poderá diferenciá-lo de um
número incluído entre parênteses para representar sua posição na ordem das
operações: # http://tinyurl.com/j8mca8o
```

```
# isto é uma tupla
("self_taught",)
```

```
# isto não é uma tupla
(9) + 1
```

```
>> ('self_taught',) >> 10
```

Você não poderá adicionar novos itens a uma tupla ou alterá-la após ela ter sido criada. Se tentar alterar um objeto em uma tupla após tê-la criado, o Python lançará

uma exceção: # <http://tinyurl.com/z3x34nk>

```
dys = ("1984",  
"Brave New World",  
"Fahrenheit 451")
```

```
dys[1] = "Handmaid's Tale"
```

>> TypeError: 'tuple' object does not support item assignment Você pode acessar os itens de uma tupla da mesma forma que faria em uma lista - referenciando o índice do item: # <http://tinyurl.com/z9dc6lo>

```
dys = ("1984",  
"Brave New World",  
"Fahrenheit 451")
```

```
dys[2]
```

```
>> 'Fahrenheit 451'
```

Podemos verificar se um item existe em uma tupla usando a palavra-chave `in`: # <http://tinyurl.com/j3bsel7>

```
dys = ("1984",  
"Brave New World",  
"Fahrenheit 451")
```

```
"1984" in dys
```

>> True Insira a palavra-chave `not` antes de `in` para verificar se um item não existe em uma tupla: # <http://tinyurl.com/jpdjjv9>

```
dys = ("1984",  
"Brave New World",  
"Fahrenheit 451")
```

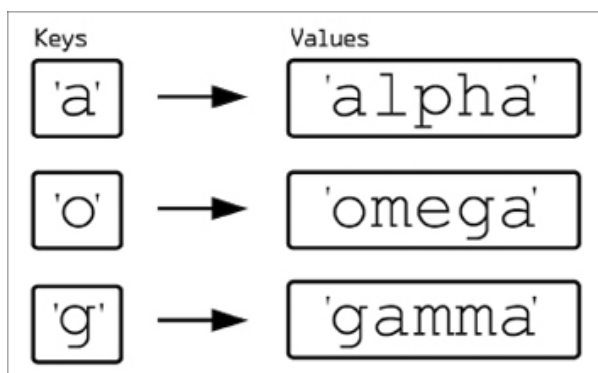
```
"Handmaid's Tale" not in dys
```

>> True Você deve estar se perguntando por que usaria uma estrutura de dados que parece ser uma lista menos flexível. As tuplas serão úteis quando você estiver lidando com valores que souber que nunca mudarão e quiser assegurar que outras partes de seu programa não os alterem. As coordenadas geográficas são um exemplo de dados cujo armazenamento em uma tupla seria útil. Você deve armazenar a longitude e a latitude de uma cidade em uma tupla porque esses valores nunca mudarão e armazenar os dados desta forma assegurará que outras partes do programa não consigam alterá-los acidentalmente. As tuplas - ao contrário das listas - não podem ser usadas como chaves em dicionários, algo sobre o que você aprenderá na próxima seção deste capítulo.

## Dicionários

Os **dicionários** são outro contêiner interno para o armazenamento de objetos. Eles são usados para vincular um objeto, chamado de **chave**, a outro objeto – chamado **valor**. A vinculação de um objeto a outro é chamada de **mapeamento**. O resultado é um **par chave-valor**. Você adicionará os pares chave-valor a um dicionário. Poderá então procurar uma chave no dicionário e obter seu valor. Não poderá, no entanto, usar um valor para procurar uma chave.

Os dicionários são mutáveis, logo, é possível adicionar novos pares chave-valor a eles. Ao contrário das listas e das tuplas, os dicionários não armazenam os objetos em uma ordem específica. Sua utilidade vem das associações formadas entre chaves e valores, e existem muitas situações nas quais podemos precisar armazenar dados em pares. Por exemplo, você poderia armazenar informações sobre alguém em um dicionário. Poderia mapear uma chave chamada altura a um valor representando a altura da pessoa, uma chave chamada cordosolhos a um valor representando a cor dos olhos dessa pessoa, e uma chave chamada nacionalidade a um valor representando a nacionalidade do indivíduo.



Os dicionários são representados por chaves. Existem duas sintaxes para a criação de dicionários: # <http://tinyurl.com/zfn6jmw>

```
my_dict = dict()
```

```
my_dict
```

```
>> {}
```

E:

```
# http://tinyurl.com/jfgemf2
```

```
my_dict = {}
```

```
my_dict
```

```
>> {}
```

Você poderá adicionar pares chave-valor a um dicionário ao criá-lo. A seguir, na primeira sintaxe, a chave foi separada do valor por um operador de atribuição e na segunda por dois pontos. Uma vírgula deve separar cada par chave-valor. Ao



contrário do que ocorre com a tupla, se você só tiver um par chave-valor, não precisará inserir uma vírgula depois dele. Veja como adicionar pares chave-valor a um dicionário quando você o criar: # <http://tinyurl.com/hplqc4u>

```
fruits = {"Apple":  
"Red",  
"Banana":  
"Yellow"}  
fruits  
>> {'Apple': 'Red', 'Banana': 'Yellow'}
```

A saída do seu shell pode listar os itens do dicionário em uma ordem diferente da vista nesse exemplo porque os dicionários não armazenam suas chaves em ordem e o Python exibe os itens em uma ordem arbitrária (isso se aplica a todos os exemplos desta seção).

Os dicionários são mutáveis. Uma vez que você criar um dicionário, poderá adicionar pares chave-valor com a sintaxe `[nome_dicionário][chave]=[valor]` e procurar um valor usando uma chave com a sintaxe `[nome_dicionário][chave]`: # <http://tinyurl.com/grc28lh>

```
facts = dict()  
  
# adiciona um valor  
facts["code"] = "fun"  
# pesquisa uma chave  
facts["code"]  
  
# adiciona um valor  
facts["Bill"] = "Gates"  
# pesquisa uma chave  
facts["Bill"]  
  
# adiciona um valor  
facts["founded"] = 1776  
# pesquisa uma chave  
facts["founded"]  
  
>> 'fun'  
>> Gates >> 1776
```

Qualquer objeto pode ser um valor do dicionário. No exemplo anterior, os dois primeiros valores são strings e o último, **1776**, é um inteiro.

Ao contrário do valor em um dicionário, a chave deve ser imutável. Uma string ou uma tupla podem ser uma chave em um dicionário, mas uma lista ou um dicionário

não podem.

Para verificar se uma chave existe em um dicionário, use a palavra-chave `in`. Você não pode usar a palavra-chave `in` para verificar se um valor existe em um dicionário: # <http://tinyurl.com/hgf9vmp>

```
bill = {"Bill Gates":  
"charitable"}
```

```
"Bill Gates" in bill
```

```
>> True Se você tentar acessar uma chave que não exista em um dicionário, o  
Python lançará uma exceção.
```

Adicione a palavra-chave `not` a `in` para verificar se uma chave não existe em um dicionário: # <http://tinyurl.com/he3g993>

```
bill = {"Bill Gates":  
"charitable"}
```

```
"Bill Doors" not in bill
```

```
>> True Você pode excluir um par chave-valor de um dicionário com a palavra-chave  
del: # http://tinyurl.com/htrd9lj
```

```
books = {"Dracula": "Stoker", "1984": "Orwell",  
"The Trial": "Kafka"}
```

```
del books["The Trial"]
```

```
books
```

```
>> {'Dracula': 'Stoker', '1984': 'Orwell'}
```

Aqui está um exemplo de programa que usa um dicionário:

```
# http://tinyurl.com/gnjvcp7
```

```
rhymes = {"1": "fun",  
"2": "blue",  
"3": "me",  
"4": "floor",  
"5": "live"
```

```
}
```

```
n = input("Type a number:")  
if n in rhymes:
```

```
rhyme = rhymes[n]
print(rhyme)
else:
print("Not found.")
```

Type a number: Seu dicionário (rhymes) tem seis nomes de canções (chaves) mapeados para seis músicos (valores). Você solicitou ao usuário que digite o nome de uma canção e salve sua resposta em uma variável. Antes de procurar a resposta dele em seu dicionário, verifique se a chave existe usando a palavra-chave in. Se a chave existir, procure o nome da canção no dicionário e exiba o nome do artista que a canta. Caso contrário, exiba uma mensagem para informar ao usuário que o nome da canção não está disponível.

## Contêineres em contêineres

Você pode armazenar contêineres em outros contêineres. Por exemplo, é possível armazenar listas em outra lista: # <http://tinyurl.com/gops9fz>

```
lists = []
rap = ["Kanye West",
"Jay Z",
"Eminem",
"Nas"]
```

```
rock = ["Bob Dylan",
"The Beatles",
"Led Zeppelin"]
```

```
djs = ["Zeds Dead",
"Tiesto"]
```

```
lists.append(rap)
lists.append(rock)
lists.append(djs)
```

```
print(lists)
```

```
>> [['Kanye West', 'Jay Z', 'Eminem', 'Nas'], ['Bob Dylan', 'The Beatles', 'Led Zeppelin'], ['Zeds Dead', 'Tiesto']]
```

Nesse exemplo, **lists** tem três índices. Cada índice é uma lista: o primeiro índice é uma lista de rappers, o segundo é uma lista de roqueiros e o terceiro é uma lista de DJs. Você pode acessar essas listas usando o índice correspondente: # <http://tinyurl.com/gu4mudk>

```
# Continuação do
```

```
# último exemplo
```

```
rap = lists[0]
```

```
print(rap)
```

```
>> ['Kanye West', 'Jay Z', 'Eminem', 'Nas']
```

Se você acrescentar um novo item à sua lista `rap`, verá a alteração quando exibir `lists`: # <http://tinyurl.com/hdtosm2>

```
# Continuação do
```

```
# último exemplo
```

```
rap = lists[0]
```

```
rap.append("Kendrick Lamar")
```

```
print(rap)
```

```
print(lists)
```

```
>> ['Kanye West', 'Jay Z', 'Eminem', 'Nas', 'Kendrick Lamar']
```

```
>> [['Kanye West', 'Jay Z', 'Eminem', 'Nas', 'Kendrick Lamar'], ['Bob Dylan',  
    'The Beatles', 'Led Zeppelin'], ['Zeds Dead', 'Tiesto']]
```

Você pode armazenar uma tupla dentro de uma lista, uma lista dentro de uma tupla e um dicionário dentro de uma lista ou tupla: # <http://tinyurl.com/z9dhema>

```
locations = []
```

```
la = (34.0522, 188.2437)
```

```
chicago = (41.8781, 87.6298)
```

```
locations.append(la)
```

```
locations.append(chicago)
```

```
print(locations)
```

```
>> [(34.0522, 188.2437), (41.8781, 87.6298)]
```

```
# http://tinyurl.com/ht7gpsd
```

```
eights = ["Edgar Allan Poe",
```

```
"Charles Dickens"]
```

```
nines = ["Hemingway",
```

```
"Fitzgerald",
```

```
"Orwell"]
```

```
authors = (eights, nines)
```

```
print(authors)
>> ([ 'Edgar Allan Poe', 'Charles Dickens'], [ 'Hemingway', 'Fitzgerald',
'Orwell']) # http://tinyurl.com/h8ck5er
```

```
bday = {"Hemingway":
"7.21.1899",
"Fitzgerald":
"9.24.1896"}
```

```
my_list = [bday]
print(my_list)
my_tuple = (bday,)
print(my_tuple)
>> [{'Hemingway': '7.21.1899', 'Fitzgerald': '9.24.1896'}]
>> ({'Hemingway': '7.21.1899', 'Fitzgerald': '9.24.1896'},) Uma lista, uma tupla
ou um dicionário podem ser um valor em um dicionário: #
http://tinyurl.com/zqupwx4
```

```
ny = {"location":
(40.7128,
74.0059),
```

```
"celebs":
["W. Allen",
"Jay Z",
"K. Bacon"],
```

```
"facts":
{"state":
"NY",
"country":
"America"}
```

```
}
```

Nesse exemplo, seu dicionário **ny** tem três chaves: "**location**", "**celebs**" e "**facts**". O valor da primeira chave é uma tupla porque coordenadas geográficas nunca mudam. O valor da segunda chave é uma lista de celebridades que vivem em Nova York; uma lista foi usada porque podem ocorrer mudanças. O valor da terceira chave é um dicionário porque os pares chave-valor são a melhor maneira de apresentar fatos sobre Nova York.

## Vocabulário

**Dicionário:** Contêiner interno de armazenamento de objetos usado para mapear um objeto – chamado chave – para outro objeto – chamado valor.

**Iterável:** Um objeto é iterável quando podemos acessar cada item usando um loop.

**Iteráveis:** Objetos pelos quais podemos iterar como as strings, as listas e as tuplas.

**Lista:** Contêiner que armazena objetos em uma ordem específica.

**Método:** Função associada a um tipo de dado específico.

**Mutável:** Quando um contêiner é mutável, seu conteúdo pode mudar.

**Par chave-valor:** Uma chave mapeada para um valor em um dicionário.

**Valor:** Um valor mapeado para uma chave em um dicionário.

## Desafios

1. Crie uma lista de seus músicos favoritos.
2. Crie uma lista de tuplas, com cada tupla, contendo a longitude e a latitude de algum lugar onde você morou ou que visitou.
3. Crie um dicionário contendo diferentes atributos sobre você: altura, cor favorita, autor favorito *etc.*
4. Escreva um programa que permita que o usuário pergunte sua altura, cor favorita ou autor favorito e retorne o resultado a partir do dicionário criado no desafio anterior.
5. Crie um dicionário que mapeie seus músicos favoritos para uma lista das canções que você mais gosta deles.
6. As listas, as tuplas e os dicionários são apenas alguns dos contêineres internos do Python. Faça uma pesquisa sobre os conjuntos (um tipo de contêiner) do Python. Quando você usaria um conjunto?

Soluções: <http://tinyurl.com/z54w9cb>.

## CAPÍTULO 6

### Manipulação de strings

*“Em teoria, não existe diferença entre a teoria e a prática. No entanto, na prática ela existe.”*

#### – Jan L. A. van de Snepscheut

O Python tem funcionalidades internas para a manipulação de strings, como dividir uma string em duas partes na posição de um caractere específico ou alterar as letras de uma string de maiúsculas para minúsculas, e vice-versa. Por exemplo, se você tiver uma string TODA EM LETRAS MAIÚSCULAS e quiser que ela passe para minúsculas, pode alterar isso usando o Python. Neste capítulo, você conhecerá as strings e examinará algumas das ferramentas mais úteis do Python para a sua manipulação.

### Strings com aspas triplas

Se uma string se estender por mais de uma linha, você terá de inseri-la em aspas triplas: # <http://tinyurl.com/h59ygda>

```
""" linha um  
linha dois  
linha três  
"""
```

Se você tentar definir uma string que se estenda por mais de uma linha com aspas simples ou duplas, verá uma mensagem de erro de sintaxe.

### Índices

As strings, como as listas e as tuplas, são iteráveis. Você pode procurar cada caractere de uma string com um índice. Como em outros iteráveis, o primeiro caractere de uma string fica no índice 0 e cada índice subsequente é incrementado em 1: # <http://tinyurl.com/zqgc2jw>

```
author = "Kafka"  
author[0]  
author[1]  
author[2]  
author[3]  
author[4]  
  
>> 'K'  
>> 'a'
```

```
>> 'f'
>> 'k'
>> 'a'
```

Nesse exemplo, você usou os índices **0**, **1**, **2**, **3** e **4** para buscar cada um dos caracteres da string **"Kafka"**. Se tentar procurar um caractere após o último índice de sua string, o Python lançará uma exceção: # <http://tinyurl.com/zk52tef>

```
author = "Kafka"
author[5]
```

```
>> IndexError: string index out of range
O Python também permite procurar cada item de uma lista com um índice negativo: um índice (que deve ser um número negativo) que você poderá usar para procurar itens em um iterável da direita para a esquerda, em vez de da esquerda para a direita. Você poderia usar o índice negativo -1 para buscar o último item de um iterável: # http://tinyurl.com/hyju2t5
```

```
author = "Kafka"
author[-1]
```

```
>> a
```

O índice negativo **-2** procura o penúltimo item, o índice negativo **-3** procura o antepenúltimo item, e assim por diante: # <http://tinyurl.com/jtpx7sr>

```
author = "Kafka"
author[-2]
author[-3]
```

```
>> k
>> f
```

## Strings são imutáveis

As strings, como as tuplas, são imutáveis. Você não pode alterar os caracteres de uma string. Se quiser alterá-los, terá de criar uma nova string: # <http://tinyurl.com/hsr83lv>

```
ff = "F. Fitzgerald"
ff = "F. Scott Fitzgerald"
ff
>> 'F. Scott Fitzgerald'
```

O Python tem vários métodos para a criação de novas strings a partir de strings existentes, que você aprenderá a usar neste capítulo.



## Concatenação

Você pode combinar duas (ou mais) strings usando o operador de adição. O resultado será uma string composta dos caracteres da primeira string, seguidos pelos caracteres da(s) próxima(s) string(s). A combinação de strings chama-se concatenação: # <http://tinyurl.com/h4z5mlg>

```
"cat" + "in" + "hat"
```

```
>> 'catinhat'
```

```
# http://tinyurl.com/gsrjle
```

```
"cat" + " in" + " the" + " hat"
```

```
>> 'cat in the hat'
```

## Multiplicação de strings

É possível multiplicar uma string por um número com o operador de multiplicação: # <http://tinyurl.com/zvm9gng>

```
"Sawyer" * 3
```

```
>> SawyerSawyerSawyer
```

## Alteração de letras maiúsculas para minúsculas e vice-versa

Você pode alterar todos os caracteres de uma string para letras maiúsculas chamando o método `upper`: # <http://tinyurl.com/hhancz6>

```
"We hold these truths...".upper()
```

```
>> 'WE HOLD THESE TRUTHS...'
```

Da mesma forma, pode alterar todas as letras de uma string para minúsculas chamando o método `lower`: # <http://tinyurl.com/zkz48u5>

```
"SO IT GOES.".lower()
```

```
>> 'so it goes.'
```

Você pode fazer com que a primeira letra de uma frase seja maiúscula chamando o método `capitalize` em uma string: # <http://tinyurl.com/jp5hexn>

```
"four score and...".capitalize()
```

```
>> 'Four score and...'
```

## Método format

Podemos criar uma nova string usando o método **format**, que procura ocorrências de chaves ({} ) na string e as substitui pelos parâmetros passados: # <http://tinyurl.com/juvguy8>

```
"William {}".format("Faulkner")  
>> 'William Faulkner'
```

Também podemos passar uma variável como parâmetro:

```
# http://tinyurl.com/zcpt9se
```

```
last = "Faulkner"  
"William {}".format(last)  
>> 'William Faulkner'
```

Não há a restrição de as chaves só poderem ser usadas uma vez – podemos usá-las na string quantas vezes quisermos: # <http://tinyurl.com/z6t6d8n>

```
author = "William Faulkner"  
year_born = "1897"
```

```
"{} was born in {}".format(author,  
year_born)  
>> 'William Faulkner was born in 1897.'
```

O método **format** será útil se você estiver criando uma string a partir de entradas do usuário: # <http://tinyurl.com/gnrdsj9>

```
n1 = input("Enter a noun:")  
v = input("Enter a verb:")  
adj = input("Enter an adj:")  
n2 = input("Enter a noun:")
```

```
r = ""The {} {} the {} {}  
"".format(n1,  
v,  
adj,  
n2)  
print(r)
```

```
>> Enter a noun: Seu programa solicita ao usuário que insira dois substantivos,  
um verbo e um adjetivo e, em seguida, usa o método format para criar uma nova  
string com as entradas e a exibe.
```

## Método split

As strings têm um método chamado **split** que você poderá usar para dividir uma string em duas ou mais strings. Você passará uma string como parâmetro para o método **split** e ele a usará para dividir a string original em várias strings. Por exemplo, podemos dividir a string "I jumped over the puddle. It was 12 feet!" em duas strings diferentes passando um ponto (.) como parâmetro para o método **split**: # <http://tinyurl.com/he8u28o>

```
"Hello.Yes!".split(".")
>> ['Hello', ' Yes!']
```

O resultado é uma lista com dois itens: uma string composta de todos os caracteres anteriores ao ponto e outra composta de todos os caracteres posteriores ao ponto. A string passada para **split** (nesse caso, um ponto) não é incluída no resultado.

## Método join

O método **join** permite adicionar novos caracteres entre cada caractere de uma string: # <http://tinyurl.com/h2pjkso>

```
first_three = "abc"
result = "+".join(first_three)
result
>> 'a+b+c'
```

Você pode transformar uma lista de strings em uma única string chamando o método **join** em uma string vazia e passando a lista como parâmetro: # <http://tinyurl.com/z49e3up>

```
words = ["The",
"fox",
"jumped",
"over",
"the",
"fence",
"."]
one = "".join(words)
one
>> Thefoxjumpedoverthefence.
```

Podemos criar uma string em que cada palavra apareça separada por um espaço chamando o método **join** em uma string com um espaço: #

<http://tinyurl.com/h4qq5oy>

```
words = ["The",  
"fox",  
"jumped",  
"over",  
"the",  
"fence",  
"."]  
one = " ".join(words)  
one
```

```
>> The fox jumped over the fence .
```

## Remoção de espaços

Você pode usar o método **strip** para remover os espaços em branco iniciais e finais de uma string: # <http://tinyurl.com/jfndhgx>

```
s = " The "  
s = s.strip()  
s
```

```
>> 'The'
```

## Método replace

O método **replace** substitui cada ocorrência de uma string por outra string. O primeiro parâmetro é a string a ser substituída e o segundo é a string que substituirá as ocorrências: # <http://tinyurl.com/zha4uwo>

```
equ = "All animals are equal."  
equ = equ.replace("a", "@")  
print(equ)
```

```
>> All @nim@ls @re equ@l.
```

## Busca de um índice

Você pode obter o índice da primeira ocorrência de um caractere em uma string com o método **index**. Passe o caractere que está procurando como parâmetro e o método **index** retornará o índice da primeira ocorrência desse caractere na string: # <http://tinyurl.com/hzc6asc>

```
"animals".index("m")
```

```
>> 3
```

O Python lançará uma exceção se o método `index` não encontrar uma ocorrência: # <http://tinyurl.com/jmtc984>

```
"animals".index("z")
```

```
>> ValueError: substring not found Se você não tiver certeza de se encontrará uma ocorrência, pode usar a manipulação de exceções: # http://tinyurl.com/zl6q4fd
```

```
try:
```

```
"animals".index("z")
```

```
except:
```

```
print("Not found.")
```

```
>> Not found.
```

## Palavra-chave in

A palavra-chave `in` verifica se uma string existe em outra string e retorna `True` ou `False`: # <http://tinyurl.com/hsnygwz>

```
"Cat" in "Cat in the hat."
```

```
>> True
```

```
# http://tinyurl.com/z9b3e97
```

```
"Bat" in "Cat in the hat."
```

```
>> False
```

Insira a palavra-chave `not` na frente de `in` para verificar se uma string não existe em outra string: # <http://tinyurl.com/jz8sygd>

```
"Potter" not in "Harry"
```

```
>> True
```

## Escape de strings

Se você usar aspas dentro de uma string, verá uma mensagem de erro de sintaxe: # <http://tinyurl.com/zj6hc4r>

```
# esse código não funciona
```

```
"She said "Surely.""
```

```
>> SyntaxError: invalid syntax Você pode corrigir esse erro colocando barras
```

invertidas antes das aspas: # <http://tinyurl.com/jdsrr7e>

```
"She said \"Surely.\""
```

```
>> 'She said "Surely."'
```

```
# http://tinyurl.com/zr7o7d7
```

```
'She said \"Surely.\"'
```

```
>> 'She said "Surely."'
```

**Escapar** uma string é inserir um símbolo na frente de um caractere que normalmente teria um significado especial em Python (aqui, as aspas), permitindo que a linguagem saiba que, nesse caso, as aspas representam um caractere e não o significado especial. O Python usa uma barra invertida para o escape.

Não é preciso escapar aspas simples dentro de uma string com aspas duplas: # <http://tinyurl.com/hoef63o>

```
"She said 'Surely.'"
```

```
>> "She said 'Surely.'"
```

Você também pode inserir aspas duplas dentro de aspas simples, o que é mais fácil do que escapar as aspas duplas: # <http://tinyurl.com/zkgfawo>

```
'She said "Surely."'
```

```
>> 'She said "Surely."'
```

## Nova linha

A inserção de `\n` dentro de uma string representa uma nova linha (New Line): # <http://tinyurl.com/zyrhaeg>

```
print("line1\nline2\nline3")
```

```
>> line1
```

```
>> line2
```

```
>> line3
```

## Fatiamento

**Fatiar** é uma maneira de retornar um novo iterável a partir de um subconjunto dos itens de outro iterável. A sintaxe do fatiamento é: `[iterável][índice_inicial:índice_final]`

O **índice inicial** é onde começará o fatiamento e o **índice final** é onde ele terminará.

Veja como fatiar uma lista:

```
# http://tinyurl.com/h2rqj2a
```

```
fict = ["Tolstoy",  
"Camus",  
"Orwell",  
"Huxley",  
"Austin"]  
fict[0:3]
```

```
>> ['Tolstoy', 'Camus', 'Orwell']
```

No fatiamento, o item pertencente ao índice inicial é incluído, mas o índice final só inclui o item anterior a ele. Portanto, se você quiser fazer o fatiamento de "Tolstoy" (índice 0) a "Orwell" (índice 2), terá de fatiar do índice 0 ao índice 3.

Aqui está um exemplo do fatiamento de uma string:

```
# http://tinyurl.com/hug9euj
```

```
ivan = ""In place of death there \  
was light.""
```

```
ivan[0:17]  
ivan[17:33]
```

```
>> 'In place of death'  
>> ' there was light.'
```

Se o seu índice inicial for 0, você poderá deixá-lo vazio: # <http://tinyurl.com/judcpx4>

```
ivan = ""In place of death there \  
was light.""
```

```
ivan[:17]
```

```
>> 'In place of death'
```

Se seu índice final for o índice do último item do iterável, você também poderá deixá-lo vazio: # <http://tinyurl.com/zqoscn4>

```
ivan = ""In place of death there \  
was light.""
```

```
ivan[17:]
```

```
>> ' there was light.'
```

Se deixarmos o índice inicial e o índice final vazios, o iterável original será retornado: # <http://tinyurl.com/zqvugoc>

```
ivan = ""In place of death there \  
was light.""
```

```
ivan[:]  
>> "In place of death there was light."
```

## Vocabulário

**Escapar:** Inserir um símbolo na frente de um caractere que normalmente teria um significado especial em Python para permitir que a linguagem saiba que, nesse caso, o caractere deve representar o próprio caractere e não o significado especial.

**Fatiamento:** Uma maneira de retornar um novo iterável a partir de um subconjunto dos itens de outro iterável.

**Índice inicial:** Índice de onde começará o fatiamento.

**Índice final:** Índice de onde terminará o fatiamento.

**Índice negativo:** Índice (que deve ser um número negativo) que podemos usar para procurar itens em um iterável da direita para a esquerda em vez de da esquerda para a direita.

## Desafios

1. Exiba cada caractere da string "Camus".
2. Escreva um programa que colete duas strings com um usuário, insira-as na string "Yesterday I wrote a [resposta\_um]. I sent it to [resposta\_dois]!" e exiba uma nova string.
3. Use um método para tornar a string "aldous Huxley was born in 1894." gramaticalmente correta fazendo com que a primeira letra da frase seja maiúscula.
4. Pegue a string "Where now? Who now? When now?" e chame um método que retorne uma lista com esta aparência: ["Where now?", "Who now?", "When now?"].
5. Pegue a lista ["The", "fox", "jumped", "over", "the", "fence", "."] e transforme-a em uma string gramaticalmente correta. É preciso que haja um espaço entre cada palavra, mas nenhum espaço entre a palavra **fence** e o ponto que vem depois dela. (Não se esqueça de que você conheceu um método que transforma uma lista de strings em uma única string).



6. Substitua cada ocorrência de "s" em "A screaming comes across the sky." por um cifrão.
7. Use um método para encontrar o primeiro índice do caractere "m" na string "Hemingway".
8. Encontre um diálogo em seu livro favorito (contendo aspas) e transforme-o em uma string.
9. Crie a string "three three three" usando a concatenação e, em seguida, crie-a novamente usando a multiplicação.
10. Fatie a string "It was a bright cold day in April, and the clocks were striking thirteen." para que só inclua os caracteres existentes antes da vírgula.

Soluções: <http://tinyurl.com/hapm4dx>.

# CAPÍTULO 7

## Loops

*“Oitenta por cento do sucesso é aparecer.”*

### – Woody Allen

O segundo programa que introduzi neste livro exibiu **Hello, World!** 100 vezes. Ele fez isso usando um loop: um trecho de código que executa instruções continuamente até uma condição definida no código ser atendida. Neste capítulo, você conhecerá os loops e aprenderá a usá-los.

### Loops for

Nesta seção, você aprenderá a usar um loop **for**: um loop usado para percorrer um iterável. Esse processo chama-se **iterar**. Você pode usar um loop **for** para definir as instruções que serão executadas uma vez para cada item de um iterável e para acessar e manipular cada item a partir das instruções que definir. Por exemplo, você poderia usar um loop **for** para iterar por uma lista de strings e utilizar o método **upper** para exibir cada string com todos os seus caracteres em maiúsculas.

Você pode definir um loop **for** usando a sintaxe `for [nome_da_variável] in [nome_do_iterável]: [instruções]`

onde `[nome_da_variável]` é um nome de variável de sua escolha atribuído ao valor de cada item do iterável e `[instruções]` é o código a ser executado a cada iteração do loop. Aqui está um exemplo que usa um loop **for** para iterar pelos caracteres de uma string: # <http://tinyurl.com/jya6kpm>

```
name = "Ted"
```

```
for character in name:
```

```
print(character)
```

```
>> T
```

```
>> e >> d
```

A cada iteração do loop, a variável `character` será atribuída a um item do iterável `name`. Na primeira iteração do loop, `T` será exibido porque a variável `character` será atribuída ao valor do primeiro item do iterável `name`. Na segunda iteração do loop, `e` será exibido porque a variável `character` será atribuída ao valor do segundo item do iterável `name`. Esse processo continuará até cada item do iterável ter sido atribuído à variável `character`.

Veja um exemplo do uso de um loop **for** para a iteração pelos itens de uma lista: # <http://tinyurl.com/zeftpg8>

```
shows = ["GOT",
```

```
"Narcos",
```

```
"Vice"]
for show in shows:
    print(show)
```

```
>> GOT
```

>> Narcos >> Vice A seguir temos um exemplo do uso de um loop for para a iteração pelos itens de uma tupla: # <http://tinyurl.com/gpr5a6e>

```
coms = ("A. Development",
        "Friends",
        "Always Sunny")
for show in coms:
    print(show)
```

>> A. Development >> Friends >> Always Sunny E agora um exemplo que usa um loop for para iterar pelas chaves de um dicionário: # <http://tinyurl.com/jk7do9b>

```
people = {"G. Bluth II":
           "A. Development",
           "Barney":
           "HIMYM",
           "Dennis":
           "Always Sunny"
```

```
}
```

```
for character in people:
    print(character)
```

>> Dennis >> Barney >> G. Bluth II Você pode usar loops for para alterar os itens de um iterável mutável, como os de uma lista: # <http://tinyurl.com/j8wvp8c>

```
tv = ["GOT",
      "Narcos",
      "Vice"]
i = 0
for show in tv:
    new = tv[i]
    new = new.upper()
    tv[i] = new
    i += 1
```

```
print(tv)
```

```
>> ['GOT', 'NARCOS', 'VICE']
```

Nesse exemplo, você usou um loop **for** para iterar pela lista **tv**. Você registrou o item atual da lista usando uma **variável de índice**: uma variável que contém um inteiro que representa um índice de um iterável. A variável de índice **i** começa em **0**

e é incrementada sempre que o loop itera. Você usou a variável de índice para obter o item atual da lista e então o armazenou na variável **new**. Em seguida, chamou o método **upper** em **new**, salvou o resultado e usou sua variável de índice para substituir o item atual por ele. Para concluir, você incrementou **i** para poder procurar o item seguinte da lista na próxima iteração do loop.

Já que o acesso a cada item e ao seu índice em um iterável é muito comum, o Python tem outra sintaxe que você pode usar para isso: # <http://tinyurl.com/z45g63j>.

```
tv = ["GOT", "Narcos",  
"Vice"]  
for i, show in enumerate(tv):  
    new = tv[i]  
    new = new.upper()  
    tv[i] = new
```

```
print(tv)
```

```
>> ['GOT', 'NARCOS', 'VICE']
```

Em vez de iterar por **tv**, você passou **tv** para **enumerate** e iterou pelo resultado, o que lhe permitiu adicionar uma nova variável **i** para registrar o índice atual.

Você pode usar os loops **for** para mover dados entre iteráveis mutáveis. Por exemplo, poderia usar dois loops **for** para pegar todas as strings de duas listas diferentes, passar seus caracteres para letras maiúsculas e inseri-las em uma nova lista: # <http://tinyurl.com/zcvgklh>

```
tv = ["GOT", "Narcos",  
"Vice"]  
coms = ["Arrested Development",  
"friends",  
"Always Sunny"]  
all_shows = []
```

```
for show in tv:  
    show = show.upper()  
    all_shows.append(show)
```

```
for show in coms:  
    show = show.upper()  
    all_shows.append(show)
```

```
print(all_shows)
```

```
>> ['GOT', 'NARCOS', 'VICE', 'ARRESTED DEVELOPMENT', 'FRIENDS', 'ALWAYS SUNNY']
```

Nesse exemplo existem três listas: **tv**, **coms** e **all\_shows**. No primeiro loop, você iterou por todos os itens da lista **tv**, usou o método **upper** para colocar cada item em letras maiúsculas e usou o método **append** para adicionar cada item à lista **all\_shows**. No segundo loop, fez o mesmo com a lista **coms**. Quando você exibir a lista **all\_shows**, ela conterá os itens das duas listas, todos em letras maiúsculas.

## Função range

Você pode usar a função interna **range** para criar uma sequência de inteiros e empregar um loop **for** para percorrê-los. A função **range** recebe dois parâmetros: o número onde a sequência começa e o número onde ela acaba. A sequência de inteiros retornada pela função **range** inclui o primeiro parâmetro (o número indicativo do começo), mas não o segundo (o número que indica o término). Aqui está um exemplo do uso da função **range** para a criação de uma sequência de números e para a iteração por eles: # <http://tinyurl.com/hh5t8rw>

```
for i in range(1, 11):
    print(i)

>> 1
...
>> 9
>> 10
```

Nesse exemplo, você usou um loop **for** para exibir cada número do iterável retornado pela função **range**. Geralmente, os programadores chamam de **i** a variável usada na iteração em uma lista de inteiros.

## Loops while

Nesta seção, você aprenderá a usar um loop **while**: um loop que executa um código enquanto uma expressão for avaliada como **True**. A sintaxe do loop **while** é **while [expressão]: [código\_a\_ser\_executado]**

onde **[expressão]** representa a expressão que determina se o loop continuará ou não a ser executado e **[código\_a\_ser\_executado]** representa o código que o loop deve executar enquanto puder: # <http://tinyurl.com/j2gwlcy>

```
x = 10
while x > 0:
    print('{}' .format(x))
    x -= 1
print("Happy New Year!")
```

```
>> 10
>> 9
>> 8
>> 7
>> 6
>> 5
>> 4
>> 3
>> 2
>> 1
>> Happy New Year!
```

O loop **while** executou seu código enquanto a expressão que você definiu em seu cabeçalho, `x > 0`, estava sendo avaliada como **True**. Na primeira iteração do loop, `x` é igual a **10** e a expressão `x > 0` é avaliada como **True**. O código do loop **while** exibiu o valor de `x` e o decrementou em **1** unidade. `x` passou a ser igual a **9**. Na iteração seguinte do loop, `x` é exibido novamente e é decrementado para **8**. Esse processo continua até `x` ser decrementado para **0**, momento em que `x > 0` é avaliada como **False** e o loop termina. O Python executou então a linha de código seguinte ao loop e exibiu **Happy New Year!**

Se você definir um loop **while** com uma expressão que seja sempre avaliada como **True**, seu loop será executado infinitamente. Um loop que nunca termina é chamado de **loop infinito**. Veja um exemplo de um loop infinito (prepare-se para pressionar **Ctrl+C** em seu teclado no shell Python para interromper a execução do loop infinito): # <http://tinyurl.com/hcwfk8>

```
while True:
    print("Hello, World!")
>> Hello, World!
...
```

Já que um loop **while** é executado enquanto a expressão definida em seu cabeçalho é avaliada como **True** – e **True** sempre será igual a **True** – esse loop será executado infinitamente.

## Instrução **break**

Você pode usar uma instrução **break** – uma instrução com a palavra-chave **break** – para encerrar um loop. O loop a seguir será executado 100 vezes: # <http://tinyurl.com/zrdh88c>

```
for i in range(0, 100):  
    print(i)  
  
>> 0  
>> 1  
...
```

Se você adicionar uma instrução **break**, o loop só será executado uma vez: # <http://tinyurl.com/zhxf3uk>

```
for i in range(0, 100):  
    print(i)  
    break  
  
>> 0
```

Assim que o Python chegar à instrução **break**, o loop terminará. Você pode usar um loop **while** e a palavra-chave **break** para escrever um programa que continue solicitando entradas do usuário até que ele digite **q** para sair: # <http://tinyurl.com/jmak8tr>

```
qs = ["What is your name?",  
      "What is your fav. color?",  
      "What is your quest?"]  
n = 0  
while True:  
    print("Type q to quit")  
    a = input(qs[n])  
    if a == "q":  
        break  
    n = (n + 1) % 3
```

Type q to quit What is your name?

A cada iteração pelo loop, o programa faz ao usuário uma das perguntas da lista **qs**. **n** é uma variável de índice. A cada iteração do loop você está atribuindo **n** à avaliação da expressão  $(n + 1) \% 3$ , o que permite que percorra por um período indefinido cada pergunta da lista **qs**. Na primeira iteração do loop, **n** começa em **0**. Em seguida, **n** recebe o valor da expressão  $(0 + 1) \% 3$ , que tem **1** como resultado. A variável **n** recebe então o valor de  $(1 + 1) \% 3$ , que é **2**, porque sempre que o primeiro número de uma expressão que usa o operador de módulo é menor do que o segundo, a resposta é o primeiro número. Para concluir, **n** recebe o valor de  $(2 + 1) \% 3$ , que volta a ser **0**.

## Instrução continue

Você pode usar uma instrução **continue** – uma instrução com a palavra-chave

**continue** – para interromper a iteração atual de um loop e passar para a sua próxima iteração. Digamos que você quisesse exibir todos os números de **1** a **5**, exceto o número **3**. Pode fazer isso usando um loop **for** e uma instrução **continue**: # <http://tinyurl.com/hflun4p>

```
for i in range(1, 6):  
    if i == 3:  
        continue  
    print(i)  
  
>> 1  
>> 2  
>> 4  
>> 5
```

Nesse loop, quando **i** é igual **3**, a instrução **continue** é executada e, em vez de fazer o loop terminar completamente – como a palavra-chave **break** faria – ele persiste. O loop passa para a próxima iteração, saltando qualquer código que teria sido executado. Quando **i** é igual a **3**, e o Python executa a instrução **continue**, o número **3** não é exibido.

Você pode obter o mesmo resultado usando um loop **while** e uma instrução **continue**: # <http://tinyurl.com/gp7forl>

```
i = 1  
while i <= 5:  
    if i == 3:  
        i += 1  
        continue  
    print(i)  
    i += 1  
  
>> 1  
>> 2  
>> 4  
>> 5
```

## Loops aninhados

É possível combinar os loops de várias maneiras. Por exemplo, você poderia ter um loop dentro de outro loop, que estaria dentro de um loop existente em outro loop, que faria parte de ainda outro loop. Não há limite para o número de loops que podemos ter dentro de outros loops, embora seja recomendável limitar isso. Quando um loop está dentro de outro loop, o segundo loop está aninhado no primeiro loop. Nessa situação, o loop que tem outro loop chama-se **loop externo** e o loop



aninhado chama-se **loop interno**. Quando você tiver um loop aninhado, o loop interno percorrerá seu iterável uma vez para cada iteração do loop externo: # <http://tinyurl.com/gqjxjtq>

```
for i in range(1, 3):  
    print(i)  
    for letter in ["a", "b", "c"]: print(letter)
```

```
>> 1
```

```
>> a >> b >> c >> 2
```

```
>> a >> b >> c 0
```

loop for aninhado iterará pela lista ["a", "b", "c"] o mesmo número de vezes que o loop externo for executado – nesse caso, duas vezes. Se você alterasse o loop externo para que fosse executado três vezes, o loop interno também iteraria pela sua lista três vezes.

Você pode usar dois loops **for** para somar cada número de uma lista com todos os números de outra lista: # <http://tinyurl.com/z7duawp>

```
list1 = [1, 2, 3, 4]  
list2 = [5, 6, 7, 8]  
added = []  
for i in list1:  
    for j in list2:  
        added.append(i + j)
```

```
print(added)
```

```
>> [6, 7, 8, 9, 7, 8, 9, 10, 8, 9, 10, 11, 9, 10, 11, 12]
```

O primeiro loop itera por cada inteiro de **list1**. Para cada item dessa lista, o segundo loop percorre cada inteiro de seu próprio iterável, soma-o ao inteiro de **list1** e acrescenta o resultado à lista **added**. Chamei a variável de **j** no segundo loop **for** porque usei o nome de variável **i** no primeiro loop.

Podemos aninhar um loop **for** dentro de um loop **while** e vice-versa: # <http://tinyurl.com/hnprmmv>

```
while input('y or n?') != 'n':  
    for i in range(1, 6):  
        print(i)
```

```
>> y or n?y 1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
y or n?y
```

1  
2  
3  
4  
5

y or n?n

>>> Esse programa exibirá os números 1-5 até o usuário inserir n.

## Vocabulário

**Instrução break:** Instrução com a palavra-chave **break** usada para encerrar um loop.

**Instrução continue:** Instrução com a palavra-chave **continue** usada para interromper a iteração atual de um loop e passar para a sua próxima iteração.

**Iterar:** Usar um loop para acessar cada item de um iterável.

**Loop:** Trecho de código que executa instruções continuamente até uma condição definida no código ser atendida.

**Loop externo:** Um loop com outro loop aninhado dentro dele.

**Loop for:** Loop usado para percorrer um iterável, como uma string, lista, tupla ou dicionário.

**Loop infinito:** Um loop que nunca termina.

**Loop interno:** Um loop aninhado em outro loop.

**Loop while:** Loop que executa um código enquanto uma expressão for avaliada como True.

**Variável de índice:** Variável que contém um inteiro que representa um índice de um iterável.

## Desafios

1. Exiba cada item da lista a seguir: ["The Walking Dead", "Entourage", "The Sopranos", "The Vampire Diaries"].
2. Exiba todos os números de 25 a 50.
3. Exiba cada item da lista do primeiro desafio e seus índices.
4. Escreva um programa com um loop infinito (com a opção de digitar q para sair) e uma lista de números. A cada iteração do loop, peça ao usuário para fornecer um número da lista e informe se o seu palpite estava ou não correto.
5. Multiplique todos os números da lista [8, 19, 148, 4] por todos os números da lista [9, 1, 33, 83] e acrescente cada resultado a uma terceira lista.

Soluções: <http://tinyurl.com/z2m2ll5>.

## CAPÍTULO 8

### Módulos

*“A perseverança e a atitude fizeram maravilhas em todas as épocas.”*

#### – George Washington

Suponhamos que você escrevesse um programa com 10.000 linhas de código. Se você inserisse todo o código em um único arquivo, seria difícil percorrê-lo. Sempre que houvesse um erro ou exceção, você teria de rolar por 10.000 linhas de código para encontrar a linha que está causando o problema. Os programadores resolvem essa questão dividindo os programas grandes em várias partes, chamadas de **módulos** – outro nome dado a um arquivo Python com código – contendo cada segmento. O Python permite usar o código de um módulo em outro módulo. Ele também tem **módulos internos**, módulos que foram embutidos na linguagem e que contêm funcionalidades importantes. Neste capítulo, você conhecerá os módulos e aprenderá a usá-los.

### Módulos internos importantes

Para usar um módulo, primeiro você precisa **importá-lo**: o que significa escrever um código para que o Python saiba onde procurá-lo. Você pode importar um módulo com a sintaxe `import [nome_módulo]`. Substitua `[nome_módulo]` pelo nome do módulo que está importando. Uma vez que você tiver importado um módulo, poderá usar suas variáveis e funções.

O Python tem muitos módulos, incluindo um módulo `math` que contém funcionalidades relacionadas à matemática. Você pode encontrar uma lista de todos os módulos internos do Python em <https://docs.python.org/3/py-modindex.html>. Veja como importar o módulo `math` do Python: # <http://tinyurl.com/h3ds93u>

```
import math
```

Depois que você importar um módulo, poderá usar os códigos dele com a sintaxe `[nome_módulo].[código]`, substituindo `[nome_módulo]` pelo nome do módulo que importou e `[código]` pelo nome da função ou variável que deseja usar. A seguir temos um exemplo da importação e do uso da função `pow` do módulo `math`, que recebe dois parâmetros, `x` e `y`, e eleva `x` a `y`: # <http://tinyurl.com/hyjo59s>

```
import math
```

```
math.pow(2, 3)
```

```
>> 8.0
```

Primeiro, importe o módulo **math** no início de seu arquivo. Você deve importar todos os seus módulos no início de seu arquivo para tornar fácil ver quais módulos está usando em seu programa. Em seguida, chame a função **pow** com **math.pow(2, 3)**. A função retornará **8.0** como resultado.

O módulo **random** é outro módulo interno. Você pode usar uma função dele chamada **randint** para gerar um inteiro aleatório: passe dois inteiros para a função e ela retornará um inteiro aleatório existente entre eles: # <http://tinyurl.com/hr3fppn>

```
# A saída pode não ser 52 quando você # executar o código - ela é aleatória!
```

```
import random
```

```
random.randint(0,100)
```

```
>> 52
```

Você pode usar o módulo interno **statistics** para calcular a média, a mediana e a moda em um iterável de números: # <http://tinyurl.com/jrnznnoy>

```
import statistics
```

```
# média
```

```
nums = [1, 5, 33, 12, 46, 33, 2]
```

```
statistics.mean(nums)
```

```
# mediana
```

```
statistics.median(nums)
```

```
# moda
```

```
statistics.mode(nums)
```

```
>> 18.857142857142858
```

```
>> 12
```

```
>> 33
```

Use o módulo interno **keyword** para verificar se uma string é uma palavra-chave do Python: # <http://tinyurl.com/zjphfho>

```
import keyword
```

```
keyword.iskeyword("for") keyword.iskeyword("football") >> True >> False
```

## Importação de outros módulos

Nesta seção, você criará um módulo, o importará para outro módulo e usará o seu

código. Primeiro, crie uma nova pasta em seu computador chamada **tstp**. Dentro dessa pasta, crie um arquivo chamado **hello.py**. Adicione o código a seguir a **hello.py** e salve o arquivo: # <http://tinyurl.com/z5v9hk3>

```
def print_hello():  
    print("Hello")
```

Dentro de sua pasta **tstp**, crie outro arquivo Python chamado **project.py**. Adicione o código a seguir a **project.py** e salve o arquivo: # <http://tinyurl.com/j4xv728>

```
import hello
```

```
hello.print_hello()
```

```
>> Hello
```

Nesse exemplo, você usou a palavra-chave **import** para utilizar o código de seu primeiro módulo no segundo módulo.

Quando você importar um módulo, todo o código dele será executado. Crie um módulo chamado **module1.py** com o código a seguir: # <http://tinyurl.com/zgyddhp>

```
# código de module1  
print("Hello!")  
  
>> Hello!
```

O código de **module1.py** será executado quando você importá-lo para outro módulo chamado **module2.py**: # <http://tinyurl.com/jamt9dy>

```
# código de module2  
import hello  
  
>> Hello!
```

Esse comportamento pode ser inconveniente. Por exemplo, você poderia ter um código de teste em seu módulo que não quisesse executar ao importá-lo. É possível resolver esse problema inserindo todo o código do módulo dentro da instrução **if \_\_name\_\_ == "\_\_main\_\_":**. Portanto, você poderia alterar o código de **module1.py** mostrado no exemplo anterior para o código descrito a seguir: # <http://tinyurl.com/j2xdzc7>

```
# código de module1  
if __name__ == "__main__": print("Hello!")  
  
>> Hello!
```

Quando você executar esse programa, a saída continuará sendo a mesma. No entanto, quando importá-lo de **module2.py**, o código de **module1.py** não será mais executado e **Hello!** não será exibido: # <http://tinyurl.com/jjccxds>

```
# código de module2
import hello
```

## Vocabulário

**Importar:** Escrever um código que permita que o Python saiba onde procurar um módulo que você planeja usar.

**Módulo:** Outro nome dado para um arquivo Python contendo código.

**Módulo interno:** Módulos que vêm com o Python e que contêm funcionalidades importantes.

## Desafios

1. Chame uma função diferente do módulo **statistics**.
2. Crie um módulo chamado **cubed** com uma função que receba um número como parâmetro e retorne o número ao cubo. Importe e chame a função a partir de outro módulo.

Soluções: <http://tinyurl.com/hlnsdot>.

## CAPÍTULO 9

### Arquivos

*“Acredito realmente que a autoaprendizagem é o único tipo de aprendizagem que existe.”*

#### – Isaac Asimov

Podemos usar o Python para trabalhar com arquivos. Por exemplo, você pode usar o Python para ler e gravar dados em um arquivo. **Ler** dados em um arquivo significa acessar os dados desse arquivo. **Gravar** dados em um arquivo significa adicionar ou alterar dados no arquivo. Neste capítulo, você aprenderá os aspectos básicos do trabalho com arquivos.

### Gravação em arquivos

A primeira etapa do trabalho com um arquivo é abri-lo com a função interna **open** do Python. A função **open** recebe dois parâmetros: uma string representando o caminho (path) do arquivo a ser aberto e outra string que representa o modo no qual ele será aberto.

O caminho que leva a um arquivo, ou o **caminho de arquivo**, representa o local onde um arquivo reside no computador. Por exemplo, *Usersbob/st.txt* é o caminho de um arquivo chamado **st.txt**. Cada palavra separada por uma barra é o nome de uma pasta. Juntas, elas representam a localização de um arquivo. Se um caminho só tiver o nome do arquivo (sem pastas separadas por barras), o Python o procurará em qualquer que seja a pasta a partir da qual você estiver executando seu programa. Você não deve escrever um caminho de arquivo por conta própria. Os sistemas operacionais de padrão Unix e o Windows usam um tipo diferente de barra em seus caminhos de arquivo. Para evitar problemas com seu programa ao trabalhar em diferentes sistemas operacionais, você deve sempre criar caminhos de arquivo usando o módulo **os** interno do Python. Sua função **path** recebe cada pasta de um caminho de arquivo como parâmetro e cria o caminho automaticamente:

```
# http://tinyurl.com/hkqfkar
```

```
import os
os.path.join("Users",
"bob",
"st.txt")
```

```
>> 'Users/bob/st.txt'
```

Criar caminhos de arquivo com a função **path** assegura que eles funcionem em qualquer sistema operacional. Mesmo assim, trabalhar com caminhos de arquivo pode ser complicado. Acesse <http://theselftaughtprogrammer.io/filepaths> se tiver



problemas.

O modo que você passar para a função **open** determinará as ações que poderão ser executadas no arquivo que for aberto. Estes são alguns dos modos nos quais você poderá abrir um arquivo:

- "r" abre um arquivo somente para leitura.
- "w" abre um arquivo somente para gravação. Se o arquivo existir, ele será sobrescrito. Se ele não existir, um novo arquivo será criado para gravação.
- "w+" abre um arquivo para leitura e gravação. Se o arquivo existir, ele será sobrescrito. Se ele não existir, um novo arquivo será criado para leitura e gravação.<sup>1</sup>

A função **open** retorna um objeto, chamado **objeto de arquivo (file object)**, que você poderá usar para ler e/ou gravar no arquivo. Quando você usar o modo "w", a função **open** criará um novo arquivo, se ele ainda não existir, no diretório em que seu programa estiver sendo executado.

Agora você pode usar o método **write** no objeto de arquivo para fazer gravações e o método **close** para fechar o arquivo. Se você abrir um arquivo usando o método **open**, deve fechá-lo com o método **close**. Se usar o método **open** em vários arquivos e se esquecer de fechá-los, isso pode causar problemas em seu programa. Aqui está um exemplo da abertura de um arquivo, da gravação feita nele e de seu fechamento:

```
# http://tinyurl.com/zfgczj5
```

```
st = open("st.txt", "w")  
st.write("Hi from Python!")  
st.close()
```

Nesse exemplo, você usou a função **open** para abrir o arquivo e salvar o objeto de arquivo que ela retornou na variável **st**. Em seguida, chamou o método **write** em **st**, que recebeu uma string como parâmetro e a gravou no novo arquivo que o Python criou. Para concluir, você fechou seu arquivo chamando o método **close** no objeto de arquivo.

## Fechamento automático de arquivos

Existe uma segunda sintaxe importante para a abertura de arquivos que evita termos de nos lembrar de fechá-los. Para usar essa sintaxe, você deve inserir todo o código que precisa de acesso ao objeto de arquivo dentro de uma instrução **with**: uma instrução composta com uma ação que ocorre automaticamente quando o Python sai da instrução.

A sintaxe de abertura de um arquivo com o uso de uma instrução **with** é

```
with open([caminho_arquivo],[modo]) as [nome_da_variável]: [seu_código]
```

[*caminho\_arquivo*] representa o caminho de arquivo, [*modo*] representa o modo no qual o arquivo será aberto, [*nome\_da\_variável*] representa o nome da variável à qual o objeto de arquivo será atribuído e [*seu\_código*] representa o código que terá acesso à variável à qual o objeto de arquivo foi atribuído.

Quando você usar essa sintaxe para abrir um arquivo, ele será fechado automaticamente após o último bloco do comandos de [*seu\_código*] ser executado. A seguir temos um exemplo da seção anterior usando essa nova sintaxe para abrir, gravar e fechar um arquivo:

```
# http://tinyurl.com/jt9guu2
```

```
with open("st.txt", "w") as f:  
f.write("Hi from Python!")
```

Enquanto você estiver dentro da instrução **with**, poderá acessar o objeto de arquivo – nesse caso, ele foi chamado de **f**. Assim que o Python terminar de executar todo o código da instrução **with**, ele fechará o arquivo para você.

## Leitura em arquivos

Se quiser ler o arquivo, passe "r" como segundo parâmetro para **open**. Em seguida, chame o método **read** no objeto de arquivo e ele retornará um iterável contendo cada linha do arquivo:

```
# http://tinyurl.com/hmuamr7
```

```
# certifique-se de ter  
# criado o arquivo  
# do exemplo anterior
```

```
with open("st.txt", "r") as f:  
print(f.read())
```

```
>> Hi from Python!
```

Você só pode chamar **read** em um arquivo uma vez, sem ser preciso fechá-lo e reabri-lo para acessar seu conteúdo, logo, deve salvar o conteúdo do arquivo em uma variável ou contêiner se for usá-lo posteriormente em seu programa. Veja como salvar o conteúdo do arquivo do exemplo anterior em uma lista:

```
# http://tinyurl.com/hkzhxdz
```

```
my_list = list()
```

```
with open("st.txt", "r") as f:
```

```
my_list.append(f.read())
```

```
print(my_list)
```

```
>> ['Hi from Python!']
```

Agora você pode acessar esses dados posteriormente em seu programa.

## Arquivos CSV

O Python vem com um módulo interno que permite trabalhar com **arquivos CSV**. Um arquivo CSV tem uma extensão **.csv** que separa dados usando vírgulas (CSV é a abreviação de Comma Separated Values, Valores Separados por Vírgulas). Programas que gerenciam planilhas como o Excel com frequência usam arquivos CSV. Cada trecho de dados separado por uma vírgula em um arquivo CSV representa uma célula e cada linha do arquivo representa uma linha da planilha. Um **delimitador** é um símbolo, como uma vírgula ou uma barra vertical (|), usado para separar dados em um arquivo CSV. Este é o conteúdo de um arquivo CSV chamado **self\_taught.csv**:

```
one,two,three
```

```
four,five,six
```

Você poderia carregar esse arquivo no Excel e tanto um, quanto dois, quanto três receberiam células na primeira linha da planilha, assim como quatro, cinco e seis receberiam células na segunda linha.

Você pode usar uma instrução **with** para abrir um arquivo CSV, mas é preciso usar o módulo **csv** dentro dela para converter o objeto de arquivo em objeto **csv**. O módulo **csv** tem um método chamado **writer** que recebe um objeto de arquivo e um delimitador. O método **writer** retorna um objeto **csv** que tem um método chamado **writerow**. O método **writerow** recebe uma lista como parâmetro, e você pode usá-lo para fazer gravações em um arquivo CSV. Cada item da lista será gravado – separado pelo delimitador que você passar para o método **writer** – em uma linha no arquivo CSV. O método **writerow** só cria uma linha, logo, você terá de chamá-lo duas vezes para criar duas linhas:

```
# http://tinyurl.com/go9wepf
```

```
import csv
```

```
with open("st.csv", "w", newline='') as f:
```

```
w = csv.writer(f,
```

```
delimiter=",")
```

```
w.writerow(["one",
```

```
"two",
```

```
"three"])
```

```
w.writerow(["four",  
"five",  
"six"])
```

Esse programa criará um novo arquivo chamado **st.csv**, e quando você o abrir em um editor de texto, sua aparência será esta:

```
one,two,three  
four,five,six
```

Se você carregar o arquivo no Excel (ou no Planilhas Google, uma alternativa ao Excel que é gratuita), as vírgulas desaparecerão, mas um, dois e três serão células na linha um, e quatro, cinco e seis serão células na linha dois.

Você também pode usar o módulo **csv** para ler o conteúdo de um arquivo. Para ler em um arquivo CSV, primeiro passe **"r"** como segundo parâmetro para a função **open** para abrir o arquivo para leitura. Em seguida, dentro da instrução **with**, chame o método **reader**, passando o objeto de arquivo e uma vírgula como o delimitador, e ele retornará um iterável que você poderá usar para acessar cada linha do arquivo.

```
# http://tinyurl.com/gvcdgxf
```

```
# certifique-se de ter criado  
# o arquivo do exemplo  
# anterior
```

```
import csv
```

```
with open("st.csv", "r") as f:  
    r = csv.reader(f, delimiter=",")  
    for row in r:  
        print(",".join(row))
```

```
>> one,two,three  
>> four,five,six
```

Nesse exemplo, você abriu **st.csv** para leitura e o converteu em um objeto **csv** usando o método **reader**. Em seguida, iterou pelo objeto **csv** usando um loop. A cada interação do loop, você chamou o método **join** em uma vírgula para adicionar a vírgula entre cada trecho de dados do arquivo e exibir o conteúdo da maneira como ele aparece no arquivo original (separado por vírgulas).

## Vocabulário

**Arquivo CSV:** Arquivo com extensão **.csv** que separa dados usando vírgulas (CSV é a abreviação de Comma Separated Values). Frequentemente usado em programas que gerenciam planilhas, como o Excel.

**Caminho (path) de arquivo:** O local onde um arquivo reside em seu computador.

**Delimitador:** Símbolo, como uma vírgula, usado para separar dados em um arquivo CSV.

**Instrução with:** Instrução composta com uma ação que ocorre automaticamente quando o Python sai dela.

**Gravar:** Adicionar ou alterar dados no arquivo.

**Ler:** Acessar o conteúdo do arquivo.

**Objeto de arquivo:** Objeto que você pode usar para ler ou gravar em um arquivo.

## Desafios

1. Encontre um arquivo em seu computador e exiba seu conteúdo usando o Python.
2. Escreva um programa que faça uma pergunta a um usuário e salve a resposta em um arquivo.
3. Pegue os itens desta lista de listas, `[["Top Gun", "Risky Business", "Minority Report"], ["Titanic", "The Revenant", "Inception"], ["Training Day", "Man on Fire", "Flight"]]`, e grave-os em um arquivo CSV. Os dados de cada lista devem ser uma linha no arquivo, com cada item da lista separado por uma vírgula.

Soluções: <http://tinyurl.com/hll6t3q>.

---

[https://www.tutorialspoint.com/python/python\\_files\\_io.htm](https://www.tutorialspoint.com/python/python_files_io.htm)

# CAPÍTULO 10

## Juntando tudo

*“Tudo que aprendi, aprendi nos livros.”*

### – Abraham Lincoln

Neste capítulo, você combinará os conceitos que aprendeu até agora e criará um jogo baseado em texto, o clássico Forca. Se você nunca jogou Forca, o jogo funciona assim: 1. O Jogador Um escolhe uma palavra secreta e desenha um traço para cada letra (você usará um underscore (\_) para representar cada traço).

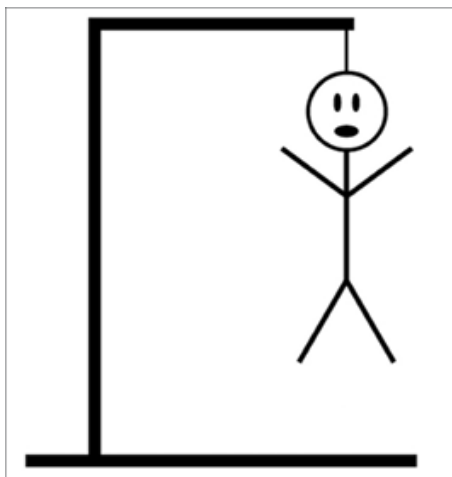
2. O Jogador Dois tenta adivinhar a palavra fornecendo uma letra de cada vez.

3. Se o Jogador Dois adivinhar uma letra corretamente, o Jogador Um substituirá o underscore correspondente pela letra certa. Nessa versão do jogo, se uma letra aparecer duas vezes em uma palavra, ela terá de ser adivinhada duas vezes.

OU

Se o palpite do Jogador Dois for incorreto, o Jogador Um desenhará uma parte do corpo de um boneco enforcado (começando com a cabeça).

4. Se o Jogador Dois completar a palavra antes do desenho do enforcado ser concluído, ele será o vencedor. Caso contrário, terá perdido.



Em seu programa, o computador será o Jogador Um e a pessoa que tentará adivinhar será o Jogador Dois. Está pronto para criar a Forca?

## Forca

Este é o começo do código da Forca:

```
# http://tinyurl.com/jhrvs94
```

```
def hangman(word):
```

```
    wrong = 0
```

```
stages = ["", "_____", "|", "| |", "| 0", "| /|\ ", "| / \ ", "| "]

]
```

```
rletters = list(word)
board = ["__"] * len(word) win = False
print("Welcome to Hangman")
```

Primeiro, crie uma função chamada `hangman` para armazenar o jogo. A função receberá uma variável chamada `word` como parâmetro; essa será a palavra que o Jogador Dois precisará adivinhar. Você usará outra variável, `wrong`, para registrar quantas letras incorretas o Jogador Dois forneceu.

A variável **stages** é uma lista preenchida com strings que você usará para desenhar o enforcado. Conforme o Python for exibindo cada string da lista **stages** em uma nova linha, um desenho de um enforcado se formará. A variável **rletters** é uma lista contendo cada caractere da variável **word** que registrará que letras ainda faltam ser adivinhadas.

A variável **board** é uma lista de strings usada para registrar as dicas que você está exibindo para o Jogador Dois, por exemplo, `c__t` se a palavra for **cat** (e o Jogador Dois já tiver adivinhado corretamente o **c** e o **t**). Você usou `["__"] * len(word)` para preencher a lista **board**, com dois underscores para cada caractere da variável **word**. Portanto, se a palavra for **cat**, **board** começará com `["__", "__", "__"]`.

Você também tem uma variável **win**, que começa como **False**, para registrar se o Jogador Dois já ganhou o jogo. Em seguida, você está exibindo "Welcome to Hangman".

A próxima parte de seu código é um loop que mantém o jogo em andamento: # <http://tinyurl.com/ztrp5jc>

```
while wrong < len(stages) - 1: print("\n")
msg = "Guess a letter"
char = input(msg)
if char in rletters:
cind = rletters \
.index(char)
board[cind] = char
rletters[cind] = '$'
else:
wrong += 1
print((" ".join(board))) e = wrong + 1
print("\n"
.join(stages[0: e]))
if "__" not in board: print("You win!") print(" ".join(board)) win = True
break
```

Seu loop (e o jogo) continuará enquanto a variável **wrong** for menor do que

**len(stages)** - 1. A variável **wrong** registra o número de letras erradas que o Jogador Dois forneceu, portanto, assim que o Jogador Dois fornecer mais letras erradas do que o número de strings que compõem o enforcado (o número de strings da lista **stages**), o jogo terminará. Você tem de subtrair 1 do tamanho da lista **stages** para compensar o fato de ela começar a contagem em 0 e **wrong** começar a contagem em 1.

Uma vez que você estiver dentro do loop, exiba um espaço em branco para dar uma boa aparência ao jogo quando ele aparecer no shell. Em seguida, colete os palpites do Jogador Dois com a função interna **input** e armazene o valor na variável **guess**.

Se **guess** estiver contida em **rletters** (a lista que registra as letras da palavra que o Jogador Dois ainda não adivinhou), o jogador terá adivinhado corretamente. Se o jogador der um palpite incorreto, você terá de atualizar a lista **board**, que será usada posteriormente no jogo para exibir as letras que faltam. Se o Jogador Dois adivinhasse **c**, você teria de alterar a lista **board** para **["c", "\_\_", "\_\_"]**.

Para fazer isso, use o método **index** na lista **rletters** para obter o primeiro índice da letra que o Jogador Dois adivinhou e use-o para substituir o underscore de **board** nesse índice pela letra adivinhada corretamente.

Existe um problema aqui. Já que **index** só retorna o primeiro índice do caractere que você está procurando, seu código não funcionará se a variável **word** tiver mais de uma ocorrência do mesmo caractere. Para resolver isso, modifique **rletters** substituindo o caractere que foi adivinhado corretamente por um cifrão, para que na próxima iteração do loop, a função **index** encontre a próxima ocorrência da letra (se houver) e não pare na primeira ocorrência.

Se, por outro lado, se o jogador fornecer uma letra incorreta, você incrementará **wrong** em 1.

Em seguida, você está exibindo o placar e o enforcado usando as listas **board** e **stages**. O código que exibe o placar é **' '.join(board)**.

Exibir o enforcado é mais complicado. Quando cada uma das strings de sua lista **stages** for exibida em uma nova linha, um desenho completo de um enforcado aparecerá. Você pode criar o enforcado inteiro fazendo a exibição com **'\n'.join(stages)**, que adicionará uma nova linha a cada string da lista **stages** para que cada string seja exibida em uma linha separada.

Para exibir seu enforcado em qualquer que seja o estágio em que o jogo estiver, fatie a lista **stages**. Comece no estágio 0 e faça o fatiamento até o estágio em que você está (representado pela variável **wrong**) mais um. Você deve adicionar 1 porque quando estiver fatiando, a última fatia não será incluída no resultado. Essa fatia só fornecerá as strings necessárias para a exibição da versão do enforcado na qual você está atualmente.



Para concluir, verifique se o Jogador Dois ganhou o jogo. Se não houver mais underscores na lista **board**, ele adivinhou todas as letras e ganhou. Se o Jogador Dois tiver vencido, exiba "You win! It was:" e a palavra que ele adivinhou corretamente. Você também pode configurar a variável **win** com **True**, o que o tirará de seu loop.

Uma vez que você sair do loop, se o Jogador Dois tiver vencido, não faça nada – o programa terá terminado. Se ele perder, a variável **win** será **False**. Se isso ocorrer, exiba o enforcado inteiro e "You lose!", seguido da palavra que ele não conseguiu adivinhar: # <http://tinyurl.com/zqklqxo>

```
if not win:
    print("\n"
        .join(stages[0: \
wrong]))
    print("You lose! It was {}".format(word))
```

Aqui está o código completo:

```
# http://tinyurl.com/h9q2cpc
```

```
def hangman(word):
    wrong = 0
    stages = ["", "_____", "|", "| |", "| 0", "| /|\ ", "| / \ ", "| "]

    ]
```

```
rletters = list(word)
board = ["__"] * len(word) win = False
print("Welcome to Hangman") while wrong < len(stages) - 1: print("\n")
msg = "Guess a letter"
char = input(msg)
if char in rletters:
    cind = rletters \
        .index(char)
    board[cind] = char
    rletters[cind] = '$'
else:
    wrong += 1
    print((" ".join(board))) e = wrong + 1
    print("\n"
        .join(stages[0: e]))
    if "__" not in board: print("You win!") print(" ".join(board)) win = True
    break
if not win:
    print("\n"
```

```
.join(stages[0: \
wrong]))
print("You lose! It was {}".format(word))
```

```
hangman("cat")
```

## Desafio

1. Modifique o jogo para que uma palavra seja selecionada aleatoriamente em uma lista de palavras.

Solução: <http://tinyurl.com/j7rb8or>.

# CAPÍTULO 11

## Prática

*“A prática não cria a perfeição. A prática cria a mielina<sup>1</sup> e a mielina cria a perfeição.”*

### – Daniel Coyle

Se este for seu primeiro livro de programação, recomendo que passe algum tempo praticando antes de ler a próxima seção. A seguir mostro alguns recursos que podem ser examinados e que fornecem aconselhamento sobre o que fazer se você não conseguir avançar.

## Leitura

<http://programmers.stackexchange.com/questions/44177/what-is-the-single-most-effective-thing-you-did-to-improve-your-programming-skill>

## Outros recursos

Organizei uma lista de recursos de programação em <http://www.theselftaughtprogrammer.io/resources>.

## Como obter ajuda

Se você não conseguir avançar, tenho algumas sugestões. Primeiro, publique sua pergunta (em inglês) no grupo do Facebook SelfTaught Programmers que fica em <https://www.facebook.com/groups/selftaughtprogrammers>. O grupo é uma comunidade de programadores (e aspirantes a programador) amigáveis que poderão ajudar a responder qualquer pergunta que você tiver.

Também recomendo acessar <http://www.stackoverflow.com>, um site no qual você poderá postar perguntas de programação e obter respostas de membros da comunidade.

Criei um curso online (em inglês) baseado neste livro que está disponível em [goseelftaught.com](http://goseelftaught.com) e que você também pode achar útil.

Aprender a confiar na ajuda de outras pessoas foi uma lição importante para mim. Tentar descobrir o que fazer é uma parte essencial do processo de aprendizagem, mas a certa altura torna-se improdutivo. No passado, quando trabalhava em projetos, eu costumava passar do ponto de produtividade. Atualmente, quando isso ocorre, publico uma pergunta online, se a resposta já não tiver sido publicada. Todas as vezes que postei uma pergunta online, alguém respondeu. Em relação a isso, palavras não são suficientes para descrever o quanto a comunidade de programação

é útil e amigável.

---

1 N.T.: A mielina é uma substância gordurosa de aparência esbranquiçada, formada por lipídios e proteínas. Ela é formada por várias camadas concêntricas de membrana plasmática que dão origem à bainha de mielina ao redor dos axônios. A função da bainha de mielina é aumentar a velocidade de condução do impulso nervoso.

## PARTE II

# **Introdução à programação orientada a objetos**

# CAPÍTULO 12

## Paradigmas de programação

*“Só existem dois tipos de linguagens: aquelas sobre as quais as pessoas reclamam e as que ninguém usa.”*

### – Bjarne Stroustrup

Um **paradigma de programação** é um estilo de programação. Existem muitos paradigmas de programação. Para programar profissionalmente, você precisa aprender o paradigma da programação orientada a objetos ou o da programação funcional. Neste capítulo examinaremos a programação procedural, a programação funcional e a programação orientada a objetos – com ênfase na programação orientada a objetos.

### Estado

Uma das diferenças básicas entre os diversos paradigmas de programação é a manipulação do **estado**. Estado é o valor das variáveis de um programa enquanto ele está sendo executado. **Estado global** é o valor das variáveis globais de um programa no momento de sua execução.

### Programação procedural

Na Parte I, você programou usando o paradigma da **programação procedural**: um estilo de programação no qual escrevemos uma sequência de etapas que avançam em direção a uma solução – com cada etapa mudando o estado do programa. Na programação procedural, o código é escrito para “fazer isso e depois aquilo”: # <http://tinyurl.com/jv2rrl8>

```
x = 2
y = 4
z = 8
xyz = x + y + z
xyz
>> 14
```

Cada linha de código desse exemplo altera o estado do programa. Primeiro, definimos **x**, depois **y**, e, em seguida, **z**. Para concluir, definimos o valor de **xyz**.

Quando programamos de forma procedural, armazenamos os dados em variáveis globais e os manipulamos com funções: # <http://tinyurl.com/gldykam>

```
rock = []
```

```
country = []
```

```
def collect_songs():  
    song = "Enter a song."  
    ask = "Type r or c. q to quit"
```

```
while True:  
    genre = input(ask)  
    if genre == "q":  
        break
```

```
if genre == "r":  
    rk = input(song)  
    rock.append(rk)
```

```
elif genre == "c":  
    cy = input(song)  
    country.append(cy)
```

```
else:  
    print("Invalid.")  
    print(rock)  
    print(country)
```

```
collect_songs()
```

>> Type r or c. q to quit: A programação procedural é boa para a criação de programas pequenos como esse, mas já que você está armazenando todo o estado em variáveis globais, terá problemas quando seu programa crescer. O problema da dependência de variáveis globais é que elas causam erros inesperados. À medida que seu programa crescer, você começará a usar variáveis globais em várias funções e será impossível rastrear todos os locais em que uma variável global foi modificada. Por exemplo, uma função poderia alterar o valor de uma variável global, e posteriormente no programa, uma segunda função acabaria alterando a mesma variável global, porque o programador que escreveu a segunda função se esqueceu de que a primeira função já a modificou. Essa situação ocorre com frequência e adultera os dados do programa.

Conforme a complexidade do seu programa crescer, o número de variáveis globais existentes nele aumentará. Se somarmos isso ao aumento no número de funções das quais o programa precisará para manipular novas funcionalidades, todas modificando as variáveis globais, a manutenção do programa se tornará rapidamente algo impossível de ser feito. Além disso, essa abordagem de programação gera **efeitos colaterais**. Um dos efeitos colaterais é a alteração do estado de uma variável global. Quando programamos de forma procedural,

geralmente nos deparamos com efeitos colaterais indesejados, como incrementar acidentalmente uma variável duas vezes.

Esse problema levou ao desenvolvimento dos paradigmas da programação orientada a objetos e funcional, e ambos usam abordagens diferentes para resolvê-lo.

## Programação funcional

A **programação funcional** é originária do cálculo lambda: a menor linguagem de programação universal do mundo (criada pelo matemático Alonzo Church). A programação funcional resolve os problemas que surgem na programação procedural eliminando o estado global. Um programador funcional usa funções que não utilizam ou alteram o estado global, o único estado que empregam são os parâmetros passados para elas. O resultado que uma função retorna geralmente é passado para outra função. Portanto, o programador funcional pode evitar o estado global passando-o de uma função para outra. A eliminação do estado global remove os efeitos colaterais e os problemas que vêm com eles.

Existe muito jargão na programação funcional, e Mary Rose Cook é mais direta em sua definição: “O código funcional é caracterizado por uma coisa: a ausência de efeitos colaterais. Ele não depende de dados de fora da função atual e não altera dados que existam fora dela”.<sup>1</sup> Mary dá prosseguimento à sua definição com um exemplo de uma função que tem efeitos colaterais: # <http://tinyurl.com/gu9jpco>

```
a = 0
```

```
def increment():  
    global a  
    a += 1
```

E de uma função sem efeitos colaterais:

```
# http://tinyurl.com/z27k2yL
```

```
def increment(a):  
    return a + 1
```

A primeira função tem efeitos colaterais porque depende de dados de fora dela, e altera dados que estão fora da função atual – ela incrementou uma variável global. A segunda função não tem efeitos colaterais porque não depende de ou altera nenhum dado existente fora dela.

Uma vantagem da programação funcional é que ela elimina toda uma categoria de erros causados pelo estado global (não existe estado global na programação funcional). Uma desvantagem dela é que certos problemas são mais fáceis de



conceber com o estado. Por exemplo, é mais fácil conceber o projeto de uma interface de usuário com o estado global do que sem ele. Se você quiser escrever um programa com um botão que alterne uma imagem entre ser exibida para o usuário e ficar invisível, será mais fácil pensar em como criar esse botão escrevendo um programa com estado global. Você poderia criar uma variável global que fosse **True** ou **False** e ocultasse ou revelasse a imagem, dependendo de seu valor atual. É mais difícil conceber um botão como esse sem o estado global.

## Programação orientada a objetos

O paradigma de programação **orientado a objetos** também resolve os problemas que surgem na programação procedural eliminando o estado global, mas em vez de armazenar o estado em funções, ele o armazena em objetos. Na programação orientada a objetos, as **classes** definem um conjunto de objetos que podem interagir uns com os outros. As classes são um mecanismo para o programador classificar e agrupar objetos semelhantes. Pense em um saco de laranjas. Cada laranja é um objeto. Todas as laranjas têm os mesmos atributos, como cor e peso, mas os valores desses atributos variam de uma laranja para a outra. Você pode usar uma classe para modelar as laranjas e criar objetos de tipo laranja com diferentes valores. Por exemplo, poderia definir uma classe que permitisse criar um objeto de tipo laranja que fosse laranja escuro e pesasse 283 gramas e um objeto de tipo laranja que fosse laranja claro e pesasse 340 gramas.

Cada objeto é uma **instância** de uma classe. Se você definir uma classe chamada **Orange**, e criar dois objetos **Orange**, cada um será uma instância da classe **Orange**; eles terão o mesmo tipo de dado – **Orange**. Podemos usar os termos objeto e instância de maneira intercambiável. Quando você definir uma classe, todas as instâncias dessa classe serão semelhantes: todas terão os atributos definidos na classe da qual são instância, como a cor e o peso de uma classe que representasse uma laranja – mas cada instância pode ter valores diferentes para esses atributos.

Em Python, uma classe é uma instrução composta com um cabeçalho e blocos de instruções. Você pode definir uma classe com a sintaxe **class [nome]: [blocos]** onde **[nome]** é o nome da classe e **[blocos]** são os blocos de instruções definidos para ela. Por convenção, as classes em Python sempre começam com letra maiúscula e são escritas em camelCase – o que significa que se uma classe tiver mais de uma palavra, as primeiras letras de todas as palavras devem ser maiúsculas **DestaForma**, em vez de serem separadas por um underscore (a convenção para os nomes de funções). Um bloco de instruções de uma classe pode ser uma instrução simples ou uma instrução composta chamada de **método**. Os métodos são como as funções, mas são definidos dentro de uma classe e só podem ser chamados em um objeto que a classe criar (como você fez na Parte I quando chamou métodos como

"hello".upper() em strings). Os nomes dos métodos, como os das funções, devem ser todos em letras minúsculas com as palavras separadas por underscores (\_).

Você pode definir os métodos com a mesma sintaxe das funções, mas com duas diferenças: deve definir o método como um bloco de instruções em uma classe e ele deve aceitar pelo menos um parâmetro (exceto em casos especiais). Por convenção, devemos sempre nomear o primeiro parâmetro de um método como **self**. Você deve definir pelo menos um parâmetro quando criar um método, porque quando o método for chamado em um objeto, o Python passará automaticamente o objeto que o chamou como parâmetro do método: # <http://tinyurl.com/zrmjape>

```
class Orange:
def __init__(self):
print("Created!")
```

Você pode usar **self** para definir uma **variável de instância**: uma variável pertencente a um objeto. Se você criar vários objetos, todos poderão ter variáveis de instância com diferentes valores. As variáveis de instância podem ser definidas com a sintaxe **self.[nome\_da\_variável] = [valor\_da\_variável]**. Normalmente, definimos as variáveis de instância dentro de um método especial chamado **\_\_init\_\_** (que significa inicializar) que o Python chama quando criamos um objeto.

A seguir temos um exemplo de uma classe que representa uma laranja (Orange): # <http://tinyurl.com/hrf6cus>

```
class Orange:
def __init__(self, w, c):
self.weight = w
self.color = c
print("Created!")
```

O código de **\_\_init\_\_** será executado quando você criar um objeto **Orange** (o que não ocorre nesse exemplo) e gerará duas variáveis de instância: **weight** e **color**. Você pode usar essas variáveis como variáveis comuns em qualquer método de sua classe. Quando você criar um objeto **Orange**, o código de **\_\_init\_\_** também exibirá **Created!** Qualquer método que esteja entre dois underscores, como **\_\_init\_\_**, é chamado de **método mágico**: um método que o Python usa para fins especiais como a criação de um objeto.

Você pode criar um novo objeto **Orange** com a mesma sintaxe usada para chamar uma função – **[nome\_da\_classe]([parâmetros])**, substituindo **[nome\_da\_classe]** pelo nome da classe que deseja usar para criar o objeto e substituindo **[parâmetros]** pelos parâmetros que **\_\_init\_\_** aceita. Não é preciso passar **self**; o Python o passará automaticamente. A criação de um novo objeto é chamada de **instanciar uma classe**:

# <http://tinyurl.com/jlc7pvk>

```
class Orange:
def __init__(self, w, c):
self.weight = w
self.color = c
print("Created!")
```

```
or1 = Orange(10, "dark orange")
print(or1)
```

```
>> Created!
```

```
>> <__main__.Orange object at 0x101a787b8>
```

Após a definição da classe, você instanciou a classe Orange com o código Orange(10, "dark orange") e Created! foi exibido. Em seguida, você exibiu o próprio objeto Orange e o Python informou que esse é um objeto Orange e forneceu sua localização na memória (a localização na memória exibida em seu computador não será a mesma desse exemplo).

Uma vez que você tiver criado um objeto, poderá obter o valor de suas variáveis de instância com a sintaxe `[nome_do_objeto].[nome_da_variável]` : # <http://tinyurl.com/grwzeo4>

```
class Orange:
def __init__(self, w, c):
self.weight = w
self.color = c
print("Created!")
```

```
or1 = Orange(10, "dark orange")
print(or1.weight)
print(or1.color)
```

```
>> Created!
```

```
>> 10
```

```
>> dark orange
```

Você pode alterar o valor de uma variável de instância com a sintaxe `[nome_do_objeto].[nome_da_variável] = [novo_valor]`

# <http://tinyurl.com/jsxgw44>

```
class Orange:
def __init__(self, w, c):
self.weight = w
self.color = c
print("Created!")
```

```
or1 = Orange(10, "dark orange")
```

```
or1.weight = 100
or1.color = "light orange"
```

```
print(or1.weight)
print(or1.color)
```

```
>> Created!
```

```
>> 100
```

```
>> light orange Inicialmente, as variáveis de instância color e weight tinham os valores "dark orange" e 10, mas você conseguiu alterá-los para "light orange" e 100.
```

Você pode usar a classe **Orange** para criar várias laranjas: # <http://tinyurl.com/jrmxlm0>

```
class Orange:
def __init__(self, w, c):
self.weight = w
self.color = c
print("Created!")
```

```
or1 = Orange(4, "light orange")
or2 = Orange(8, "dark orange")
or3 = Orange(14, "yellow")
```

```
>> Created!
```

```
>> Created!
```

```
>> Created!
```

Uma laranja não tem apenas propriedades físicas, como a cor e o peso. As laranjas também passam por processos, como apodrecer (rot), que você pode modelar com os métodos. Veja como permitir que um objeto **Orange** apodreça: # <http://tinyurl.com/zcp32pz>

```
class Orange():
def __init__(self, w, c):
    """weights are in oz"""
self.weight = w
self.color = c
self.mold = 0
print("Created!")
```

```
def rot(self, days, temp):
self.mold = days * temp
```

```
orange = Orange(6, "orange")
```

```
print(orange.mold)
orange.rot(10, 98)
print(orange.mold)

>> Created!
>> 0
>> 98.0
```

O método **rot** recebe dois parâmetros: o número de dias desde que alguém colheu a laranja e a temperatura durante esse período. Quando você o chamar, o método usará uma fórmula para incrementar a variável de instância **mold** (mofo), o que funciona porque podemos alterar o valor de qualquer variável de instância dentro de qualquer método. Agora, a laranja pode apodrecer.

Você pode definir vários métodos em uma classe. Aqui está um exemplo da modelagem de um retângulo com um método para calcular sua área e outro para alterar seu tamanho: # <http://tinyurl.com/j28qoox>

```
class Rectangle():
def __init__(self, w, l):
self.width = w
self.len = l

def area(self):
return self.width * self.len

def change_size(self, w, l):
self.width = w
self.len = l
```

```
rectangle = Rectangle(10, 20)
print(rectangle.area())
rectangle.change_size(20, 40)
print(rectangle.area())

>> 200
>> 800
```

Nesse exemplo, os objetos **Rectangle** têm duas variáveis de instância: **len** e **width**. O método **area** retorna a área do objeto **Rectangle** com a multiplicação das variáveis de instância e o método **change\_size** as altera atribuindo os números que o chamador passou como parâmetros.

A programação orientada a objetos tem muitas vantagens. Ela encoraja a reutilização de código e, portanto, diminui o tempo gasto no desenvolvimento e na manutenção. Ela também ajuda a dividir os problemas em várias partes, o que resulta em um código fácil de editar. Uma desvantagem da programação orientada a

objetos é que a criação de programas demanda esforço adicional porque seu projeto geralmente envolve muito planejamento.

## Vocabulário

**Classes:** Mecanismo que permite que o programador classifique e agrupe objetos semelhantes.

**Efeito colateral:** Alteração do estado de uma variável global.

**Estado:** Valor das variáveis de um programa enquanto ele está sendo executado.

**Estado global:** Valor das variáveis globais de um programa enquanto ele está sendo executado.

**Instância:** Todo objeto é uma instância de uma classe. Cada instância de uma classe tem o mesmo tipo de todas as outras instâncias dessa classe.

**Instanciar uma classe:** Criar um novo objeto usando uma classe.

**Métodos:** Os métodos são os blocos de instruções de uma classe. Eles são como as funções, mas são definidos dentro de uma classe e só podem ser chamados em um objeto que a classe criar.

**Método mágico:** Método que o Python usa em diferentes situações, como inicializar um objeto.

**Orientado a objetos:** Paradigma de programação no qual definimos objetos que interagem uns com os outros.

**Paradigma de programação:** Um estilo de programação.

**Programação funcional:** A programação funcional resolve o problema que surge na programação procedural eliminando o estado global ao passá-lo de uma função para a outra.

**Programação procedural:** Estilo de programação no qual escrevemos uma sequência de etapas que avançam em direção a uma solução – com cada etapa alterando o estado do programa.

**Variáveis de instância:** Variáveis que pertencem a um objeto.

## Desafios

1. Defina uma classe chamada **Apple** com quatro variáveis de instância para representar quatro atributos de uma maçã.
2. Crie uma classe **Circle** com um método chamado **area** que calcule e retorne sua área. Em seguida, crie um objeto **Circle**, chame **area** nele e exiba o resultado. Use a função **pi** do módulo interno **math** do Python.

3. Crie uma classe **Triangle** com um método chamado **area** que calcule e retorne sua área. Em seguida, crie um objeto **Triangle**, chame **area** nele e exiba o resultado.
4. Crie uma classe **Hexagon** com um método chamado **calculate\_perimeter** que calcule e retorne seu perímetro. Em seguida, crie um objeto **Hexagon**, chame **calculate\_perimeter** nele e exiba o resultado.

Soluções: <http://tinyurl.com/gpqe62e>.

---

<sup>1</sup> <https://maryrosecook.com/blog/post/a-practical-introduction-to-functionalprogramming>

# CAPÍTULO 13

## Os quatro pilares da programação orientada a objetos

*“Um bom design adiciona valor mais rápido do que adiciona custo.”*

– **Thomas C. Gale**

Existem quatro conceitos principais na programação orientada a objetos: encapsulamento, abstração, polimorfismo e herança. Juntos, eles formam os **quatro pilares da programação orientada a objetos**. Todos os quatro conceitos devem estar presentes em uma linguagem de programação para ela ser considerada uma linguagem totalmente orientada a objetos, como Python, Java e Ruby. Neste capítulo, você conhecerá cada um dos quatro pilares da programação orientada a objetos.

### Encapsulamento

O termo **encapsulamento** faz referência a dois conceitos. O primeiro é o de que na programação orientada a objetos, os objetos agrupam variáveis (estado) e métodos (para alterar o estado ou fazer cálculos que usem o estado) na mesma unidade – o próprio objeto: # <http://tinyurl.com/j74o5rh>

```
class Rectangle():
    def __init__(self, w, l):
        self.width = w
        self.len = l
```

```
    def area(self):
        return self.width * self.len
```

Nesse caso, as variáveis de instância **len** e **width** contêm o estado do objeto. O estado do objeto está agrupado na mesma unidade (o objeto) do método **area**. O método usa o estado do objeto para retornar a área do retângulo.

O segundo conceito, o encapsulamento propriamente dito, se refere à ocultação dos dados internos de uma classe para impedir que o cliente, o código de fora da classe que usa o objeto, acesse-os diretamente: # <http://tinyurl.com/jtz28ha>

```
class Data:
    def __init__(self):
        self.nums = [1, 2, 3, 4, 5]
```

```
    def change_data(self, index, n): self.nums[index] = n
```

A classe **Data** tem uma variável de instância chamada **nums** que contém uma lista de



inteiros. Uma vez que você criar um objeto **Data**, poderá alterar os itens de **nums** de duas maneiras: usando o método **change\_data** ou acessando diretamente a variável de instância **nums** usando o objeto **Data**: # <http://tinyurl.com/huczqr5>

```
class Data:
def __init__(self):
self.nums = [1, 2, 3, 4, 5]

def change_data(self, index, n): self.nums[index] = n
```

```
data_one = Data()
data_one.nums[0] = 100
print(data_one.nums)
```

```
data_two = Data()
data_two.change_data(0, 100)
print(data_two.nums)
```

```
>> [100, 2, 3, 4, 5]
```

```
>> [100, 2, 3, 4, 5]
```

As duas maneiras de alterar um item na variável de instância **nums** funcionam, mas o que aconteceria se você decidisse transformar a variável **nums** em uma tupla em vez de uma lista? Se fizer essa alteração, qualquer código cliente que tentar alterar os itens da variável **nums**, como você fez com **nums[0] = 100**, não funcionará mais, porque as tuplas são imutáveis.

Muitas linguagens de programação resolvem esse problema permitindo que os programadores definam **variáveis privadas** e **métodos privados**: variáveis e métodos que os objetos podem acessar no código que implementa os métodos, mas que o cliente não pode acessar. As variáveis e métodos privados serão úteis quando houver um método ou uma variável que sua classe use internamente, mas você estiver planejando alterar a implementação de seu código posteriormente (ou quiser preservar a flexibilidade dessa opção), e, portanto, preferir evitar que alguém que estiver usando a classe o utilize já que ele poderá mudar (e então quebraria o código do cliente). As variáveis privadas são um exemplo do segundo conceito ao qual o encapsulamento faz referência; elas ocultam os dados internos de uma classe para impedir que o cliente os acesse diretamente. As **variáveis públicas**, por outro lado, são variáveis que um cliente pode acessar.

O Python não tem variáveis privadas. Todas as suas variáveis são públicas. O Python resolve de outra maneira o problema que as variáveis privadas solucionam – usando convenções de nomeação. Em Python, se você tiver uma variável ou método que o chamador não possa acessar, seu nome deve ser antecedido por um underscore. Os

programadores Python sabem que se o nome de um método ou variável começar com um underscore, eles não devem usá-lo (embora possam fazê-lo por sua conta e risco): # <http://tinyurl.com/jkaorle>

```
class PublicPrivateExample:
```

```
def __init__(self):
```

```
self.public = "safe"
```

```
self._unsafe = "unsafe"
```

```
def public_method(self):
```

```
# os clientes podem usar isso
```

```
pass
```

```
def _unsafe_method(self):
```

```
# os clientes não devem usar isso pass
```

Os programadores que forem utilizar esse código saberão que é seguro usar a variável **self.public**, mas não devem usar a variável **self.\_unsafe** porque ela começa com um underscore, e se a usarem, o farão por sua conta e risco. A pessoa que estiver fazendo a manutenção desse código não precisa manter a variável **self.\_unsafe**, porque os chamadores não devem acessá-la. Também saberão que é seguro usar o método **public\_method**, mas não o método **\_unsafe\_method**, porque seu nome começa com um underscore.

## Abstração

**Abstração** é o processo de “eliminar ou remover características de algo para reduzi-lo a um conjunto de características essenciais”.<sup>1</sup> Você usará a abstração na programação orientada a objetos quando modelar objetos usando classes e omitir detalhes desnecessários.

Digamos que você estivesse modelando uma pessoa. Uma pessoa é algo complexo: ela tem a cor do cabelo, a cor dos olhos, altura, peso, raça, gênero *etc.* Se você criar uma classe para representar uma pessoa, alguns desses detalhes podem não ser relevantes para o problema a ser resolvido. Um exemplo de abstração seria a criação de uma classe **Person** (pessoa), mas com a omissão de alguns atributos que uma pessoa tem, como a cor dos olhos e a altura. Os objetos **Person** que sua classe criar serão abstrações de pessoas. Serão a representação de uma pessoa resumida apenas às características essenciais necessárias para o problema que você estiver resolvendo.

## Polimorfismo

**Polimorfismo** é “a habilidade (em programação) de apresentar a mesma interface

para formas (tipos de dados) subjacentes distintas”.<sup>2</sup> Uma interface é uma função ou um método. Aqui está um exemplo da apresentação da mesma interface para diferentes tipos de dados: # <http://tinyurl.com/hrxd7gn>

```
print("Hello, World!") print(200)
print(200.1)
>> Hello, World!
>> 200
>> 200.1
```

Você apresentou a mesma interface, a função `print`, para três tipos de dados diferentes: uma string, um inteiro e um número de ponto flutuante. Não precisou definir e chamar três funções separadas (como `print_string` para exibir strings, `print_int` para exibir inteiros e `print_float` para exibir números de ponto flutuante) para exibir três tipos de dados diferentes; em vez disso, pôde usar a função `print` para apresentar uma única interface para exibir todos eles.

A função interna `type` retorna o tipo de dado de um objeto: # <http://tinyurl.com/gnxq24x>

```
type("Hello, World!")
type(200)
type(200.1)
>> <class 'str'> >> <class 'int'> >> <class 'float'> Suponhamos que você quisesse
escrever um programa que criasse três objetos que se desenhasssem: triângulos,
quadrados e círculos. Você pode atingir esse objetivo definindo três classes
diferentes, Triangle, Square e Circle, e um método chamado draw para cada uma
delas. Triangle.draw() desenhará um triângulo. Square.draw() desenhará um
quadrado. E Circle.draw() desenhará um círculo. Com esse design, cada um dos
objetos terá uma interface draw que saberá como desenhá-lo. Você apresentou a
mesma interface para três tipos de dados diferentes.
```

Se o Python não suportasse o polimorfismo, você precisaria de um método para desenhar cada forma: talvez `draw_triangle` para desenhar um objeto `Triangle`, `draw_square` para desenhar um objeto `Square` e `draw_circle` para desenhar um objeto `Circle`.

Além disso, se você tivesse uma lista desses objetos e quisesse desenhar cada um, precisaria verificar cada objeto para obter seu tipo e, em seguida, chamar o método correto para esse tipo, o que tornaria o programa maior, mais difícil de ler e gravar e mais frágil. Também seria mais difícil aperfeiçoar o programa, porque sempre que você adicionasse uma nova forma, precisaria procurar todos os lugares no código em que desenhava as formas e incluir uma verificação (para saber que método usar) para esse novo tipo de forma, além de uma chamada a essa nova função `draw`. Veja um exemplo do desenho de formas com e sem polimorfismo: # Não execute

```
# Desenhando formas
# sem polimorfismo
shapes = [tr1, sq1, cr1]
for a_shape in shapes:
if type(a_shape) == "Triangle": a_shape.draw_triangle()
if type(a_shape) == "Square": a_shape.draw_square()
if type(a_shape) == "Circle": a_shape.draw_circle()
```

```
# Desenhando formas
# com polimorfismo
shapes = [tr1,
sw1,
cr1]
for a_shape in shapes:
a_shape.draw()
```

Se você quisesse adicionar uma nova forma à lista **shapes** sem polimorfismo, precisaria modificar o código no loop **for** para procurar o tipo **a\_shape** e chamar seu método **draw**. Com uma interface polimórfica uniforme, no futuro você poderá incluir quantas classes de formas quiser na lista **shapes** e a forma poderá se desenhar sem nenhum código adicional.

## Herança

A **herança** em programação é semelhante à herança genética. Na herança genética herdamos atributos, como a cor dos olhos dos pais. Da mesma forma, quando criamos uma classe, ela pode herdar os métodos e variáveis de outra classe. A classe de onde vem a herança é a **classe pai** e a classe que herda é a **classe filha**. Nesta seção, você modelará formas usando a herança. Aqui está uma classe que modela uma forma: # <http://tinyurl.com/zrnqeo3>

```
class Shape():
def __init__(self, w, l):
self.width = w
self.len = l

def print_size(self):
print("{} by {}".format(self.width, self.len))
```

```
my_shape = Shape(20, 25)
my_shape.print_size()
>> 20 by 25
```

Com essa classe, você pode criar objetos **Shape** com largura (**width**) e comprimento (**len**). Além disso, os objetos **Shape** têm o método **print\_size**, que exibe sua largura e comprimento.

Você pode definir uma classe filha que herde os elementos de uma classe pai passando o nome da classe pai como parâmetro para a classe filha ao criá-la. O exemplo a seguir cria uma classe **Square** que herda os elementos da classe **Shape**: # <http://tinyurl.com/j8lj35s>

```
class Shape():
def __init__(self, w, l):
self.width = w
self.len = l
```

```
def print_size(self):
print("{} by {}".format(self.width, self.len))
```

```
class Square(Shape):
pass
```

```
a_square = Square(20,20)
a_square.print_size()
```

```
>> 20 by 20
```

Já que você passou a classe **Shape** como parâmetro para a classe **Square**, esta herdará as variáveis e os métodos da classe **Shape**. O único bloco de instruções que você definiu na classe **Square** foi a palavra-chave **pass**, que solicita que o Python não faça nada.

Devido à herança, você pode criar um objeto **Square**, passar uma largura e um comprimento para ele e chamar o método **print\_size** nele sem escrever nenhum código (além de **pass**) na classe **Square**. Essa redução no código é importante porque evitar a repetição torna o programa menor e mais gerenciável.

Uma classe filha é como qualquer outra classe; você pode definir métodos e variáveis nela sem afetar a classe pai: # <http://tinyurl.com/hwjdcy9>

```
class Shape():
def __init__(self, w, l):
self.width = w
self.len = l
```

```
def print_size(self):
print("{} by {}".format(self.width, self.len))
```

```
"".format(self.width, self.len))
```

```
class Square(Shape):  
def area(self):  
return self.width * self.len
```

```
a_square = Square(20, 20)  
print(a_square.area())
```

```
>> 400
```

Quando uma classe filha herdar um método de uma classe pai, você poderá sobrescrevê-lo definindo um novo método com o mesmo nome do método herdado. A possibilidade de uma classe filha alterar a implementação de um método herdado de sua classe pai chama-se **sobrescrita de método**.

```
# http://tinyurl.com/hy9m8ht
```

```
class Shape():  
def __init__(self, w, l):  
self.width = w  
self.len = l
```

```
def print_size(self):  
print("{} by {}".format(self.width, self.len))
```

```
class Square(Shape):  
def area(self):  
return self.width * self.len
```

```
def print_size(self):  
print("I am {} by {}".format(self.width, self.len))
```

```
a_square = Square(20, 20)  
a_square.print_size()
```

```
>> I am 20 by 20
```

Nesse caso, já que você definiu um método chamado **print\_size**, o método recém-definido sobrescreverá o método do pai que tem o mesmo nome e exibirá uma nova mensagem quando você o chamar.

## Composição

Agora que você conhece os quatro pilares da programação orientada a objetos,

abordarei mais um conceito importante: a **composição**. A composição modela o relacionamento “tem um” armazenando um objeto como uma variável em outro objeto. Por exemplo, você pode usar a composição para representar o relacionamento entre um cão e seu dono (um cão tem um dono). Para modelar isso, primeiro defina as classes que representarão cães e pessoas: # <http://tinyurl.com/zqg488n>

```
class Dog():
def __init__(self,
name,
breed,
owner):
self.name = name
self.breed = breed
self.owner = owner
```

```
class Person():
def __init__(self, name):
self.name = name
```

Em seguida, quando você criar um objeto **Dog**, passe um objeto **Person** como o parâmetro **owner** (dono): # <http://tinyurl.com/zlzefd4>

```
# Continuação do
# último exemplo
```

```
mick = Person("Mick Jagger") stan = Dog("Stanley", "Bulldog",
mick)
print(stan.owner.name)
```

```
>> Mick Jagger Agora, o objeto stan chamado "Stanley" tem um dono - um objeto
Person chamado "Mick Jagger" - armazenado na variável de instância owner.
```

## Vocabulário

**Abstração:** O processo de “eliminar ou remover características de algo para reduzi-lo a um conjunto de características essenciais”.<sup>2</sup>

**Classe filha:** Classe que herda os atributos.

**Classe pai:** Classe de onde vem a herança.

**Cliente:** Código de fora da classe que usa o objeto.

**Composição:** A composição modela o relacionamento “tem um” armazenando um objeto como uma variável em outro objeto.

**Encapsulamento:** O encapsulamento faz referência a dois conceitos. O primeiro conceito é o de que na programação orientada a objetos, os objetos agrupam variáveis (estado) e métodos (para a alteração do estado) na mesma unidade – o

próprio objeto. O segundo conceito é a ocultação dos dados internos de uma classe para impedir que o cliente, a pessoa que está usando o código, acesse-os.

**Herança:** Na herança genética, herdamos atributos, como a cor dos olhos dos pais. Da mesma forma, quando criamos uma classe, ela pode herdar métodos e variáveis de outra classe.

**Os quatro pilares da programação orientada a objetos:** Os quatro conceitos principais da programação orientada a objetos: herança, polimorfismo, abstração e encapsulamento.

**Polimorfismo:** Polimorfismo é “a habilidade (em programação) de apresentar a mesma interface para formas (tipos de dados) subjacentes distintas”.<sup>4</sup>

**Sobrescrita de método:** Possibilidade de uma classe filha alterar a implementação de um método herdado de sua classe pai.

## Desafios

1. Crie as classes **Rectangle** e **Square** com um método chamado **calculate\_perimeter** que calcule o perímetro das formas que elas representam. Crie objetos **Rectangle** e **Square** e chame o método em ambos.
2. Defina um método em sua classe **Square** chamado **change\_size** que permita passar um número e aumente ou diminua (se o número for negativo) cada lado de um objeto **Square** de acordo com esse número.
3. Crie uma classe chamada **Shape**. Defina um método nela chamado **what\_am\_i** que exiba "I am a shape" quando chamado. Altere suas classes **Square** e **Rectangle** dos desafios anteriores para que herdem de **Shape**, crie objetos **Square** e **Rectangle** e chame o novo método em ambos.
4. Crie uma classe chamada **Horse** e uma classe chamada **Rider**. Use a composição para modelar um cavalo (horse) que tenha um cavaleiro (rider).

Soluções: <http://tinyurl.com/hz9qdh3>.

---

<sup>1</sup> <http://whatis.techtarget.com/definition/abstraction>

<sup>2</sup> <http://stackoverflow.com/questions/1031273/what-is-polymorphism-what-is-it-forand-how-is-it-used>

<sup>3</sup> <http://whatis.techtarget.com/definition/abstraction>

<sup>4</sup> <http://stackoverflow.com/questions/1031273/what-is-polymorphism-what-is-it-forand-how-is-it-used>



# CAPÍTULO 14

## Mais programação orientada a objetos

*“Trate seu código como poesia e resuma-o o máximo possível.”*

### – Ilya Dorman

Neste capítulo, abordarei conceitos adicionais relacionados à programação orientada a objetos.

### Variáveis de classe versus variáveis de instância

Em Python, as classes são objetos. Essa ideia vem do Smalltalk, uma linguagem de programação influente que foi pioneira na programação orientada a objetos. Cada classe em Python é um objeto que é uma instância do “tipo” da classe: # <http://tinyurl.com/h7ypzmd>

```
class Square:  
    pass
```

```
print(Square)
```

```
>> <class '__main__.Square'>
```

 Nesse exemplo, a classe Square é um objeto e você o exibiu.

As classes têm dois tipos de variáveis: **variáveis de classe** e **variáveis de instância**. As variáveis que você viu até agora eram variáveis de instância, definidas com a sintaxe `self.[nome_da_variável] = [valor_da_variável]`. As variáveis de instância pertencem a objetos: # <http://tinyurl.com/zmnf47e>

```
class Rectangle():  
    def __init__(self, w, l): self.width = w  
    self.len = l
```

```
def print_size(self):  
    print("{} by {}".format(self.width, self.len))
```

```
my_rectangle = Rectangle(10, 24) my_rectangle.print_size() >> 10 by 24
```

No exemplo anterior, `width` e `len` são variáveis de instância.

As variáveis de classe pertencem ao objeto que o Python cria para cada definição de classe e aos objetos que eles criam. Você pode definir as variáveis de classe como variáveis comuns (mas deve defini-las dentro de uma classe). Podemos acessá-las com objetos da classe e com um objeto criado com um objeto da classe. Elas são

acessadas da mesma forma que acessamos as variáveis de instância (com a inclusão de **self**. antes do nome da variável). As variáveis de classe são úteis; elas nos permitem compartilhar dados entre todas as instâncias de uma classe sem depender de variáveis globais: # <http://tinyurl.com/gu9unfc>

```
class Rectangle():
    recs = []

    def __init__(self, w, l): self.width = w
    self.len = l
    self.recs.append((self.width, self.len))
```

```
def print_size(self):
    print("{} by {}".format(self.width, self.len))
```

```
r1 = Rectangle(10, 24)
r2 = Rectangle(20, 40)
r3 = Rectangle(100, 200)
print(Rectangle.recs)

>> [(10, 24), (20, 40), (100, 200)]
```

Nesse exemplo, você adicionou uma variável de classe chamada **recs** à classe **Rectangle**. Ela foi definida fora do método **\_\_init\_\_** porque o Python só chama o método **\_\_init\_\_** quando criamos um objeto e você quer poder acessar a variável de classe usando o objeto da classe (que não chama o método **\_\_init\_\_**).

Em seguida, você criou três objetos **Rectangle**. Sempre que um objeto **Rectangle** é criado, o código do método **\_\_init\_\_** acrescenta uma tupla contendo a largura e o comprimento do objeto recém-criado à lista **recs**. Com esse código, sempre que você criar um novo objeto **Rectangle**, ele será adicionado automaticamente à lista **recs**. Usando uma variável de classe, você está compartilhando dados entre os diferentes objetos criados por uma classe, sem precisar usar uma variável global.

## Métodos mágicos

Toda classe em Python herda elementos de uma classe pai chamada **Object**. O Python utiliza os métodos herdados de **Object** em diferentes situações – como quando exibimos um objeto: # <http://tinyurl.com/ze8yr7s>

```
class Lion:
    def __init__(self, name): self.name = name
```

```
lion = Lion("Dilbert") print(lion)
```

>> <\_\_main\_\_.Lion object at 0x101178828> Quando você exibir um objeto Lion, o Python chamará nele um método mágico chamado `__repr__` herdado de `Object` e exibirá o que quer que o método `__repr__` retornar. Você pode sobrescrever o método herdado `__repr__` para alterar o que será exibido: # <http://tinyurl.com/j5rocqm>

```
class Lion:
```

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

```
def __repr__(self):
```

```
return self.name
```

```
lion = Lion("Dilbert") print(lion)
```

>> Dilbert Já que você sobrescreveu o método `__repr__` herdado de `Object` e o alterou para retornar o nome do objeto Lion, quando exibir um objeto Lion, seu nome - nesse caso, Dilbert - será exibido em vez de algo como <\_\_main\_\_.Lion object at 0x101178828> que o método `__repr__` teria retornado.

Os operandos de uma expressão devem ter um método mágico que o operador possa usar para avaliar a expressão. Por exemplo, na expressão `2 + 2` cada objeto de tipo inteiro tem um método mágico chamado `__add__` que o Python chamará quando avaliar a expressão. Se você definir um método `__add__` em uma classe, poderá usar os objetos que ela criar como operandos em uma expressão com o operador de adição: # <http://tinyurl.com/hlmhrwv>

```
class AlwaysPositive:
```

```
def __init__(self, number): self.n = number
```

```
def __add__(self, other): return abs(self.n +  
other.n)
```

```
x = AlwaysPositive(-20)
```

```
y = AlwaysPositive(10)
```

```
print(x + y)
```

```
>> 10
```

Os objetos `AlwaysPositive` podem ser usados como operandos em uma expressão com o operador de adição porque você definiu o método `__add__`. Quando o Python avaliar uma expressão com o operador de adição, ele chamará o método `__add__` no objeto de primeiro operando, passará o objeto de segundo operando para `__add__` como parâmetro e retornará o resultado.

Nesse caso, `__add__` está usando a função interna `abs` para retornar o valor absoluto

de dois números somados em uma expressão. Já que você definiu `__add__` desta forma, dois objetos **AlwaysPositive** avaliados em uma expressão com o operador de adição sempre retornarão o valor absoluto da soma de dois objetos; portanto, o resultado da expressão será sempre positivo.

## Palavra-chave is

A palavra-chave **is** retornará **True** se dois objetos forem o mesmo objeto; caso contrário, ela retornará **False**: # <http://tinyurl.com/gt28gww>

```
class Person:
def __init__(self):
self.name = 'Bob'
```

```
bob = Person()
same_bob = bob
print(bob is same_bob)
```

```
another_bob = Person()
print(bob is another_bob) >> True >> False
```

Quando você usar a palavra-chave **is** em uma expressão com os objetos **bob** e **same\_bob** como operadores, a expressão será avaliada como **True** porque as duas variáveis apontam para o mesmo objeto **Person**. Quando você criar um novo objeto **Person** e compará-lo com o objeto **bob** original, a expressão será avaliada como **False** porque as variáveis apontam para objetos **Person** diferentes.

Use a palavra-chave **is** para verificar se uma variável é **None**: # <http://tinyurl.com/jjettn2>

```
x = 10
if x is None:
print("x is None :( ") else:
print("x is not None")
x = None
if x is None:
print("x is None :( ") else:
print("x is not None") >> x is not None >> x is None :(
```

## Vocabulário

**Método privado:** Método que um objeto pode acessar, mas o cliente não.

**Variável de classe:** Uma variável de classe pertence a um objeto da classe e aos objetos que ele criar.

**Variável de instância:** Uma variável de instância pertence a um objeto.

**Variável pública:** Variável que um cliente pode acessar.

**Variáveis privadas:** Variável que um objeto pode acessar, mas o cliente não.

## Desafios

1. Adicione uma variável de classe **square\_list** a uma classe chamada **Square** para que, sempre que você criar um novo objeto **Square**, ele seja adicionado à lista.
2. Altere a classe **Square** para que, quando você exibir um objeto **Square**, uma mensagem seja exibida informando o comprimento de cada um dos quatro lados da forma. Por exemplo, se você criar um quadrado com **Square(29)** e exibi-lo, o Python deve exibir **29 by 29 by 29 by 29**.
3. Escreva uma função que receba dois objetos como parâmetros e retorne **True** se eles forem o mesmo objeto e **False** se não forem.

Soluções: <http://tinyurl.com/j9qjnep>.

# CAPÍTULO 15

## Juntando tudo

*“É tudo conversa até o código ser executado.”*

### – Ward Cunningham

Neste capítulo, você criará o popular jogo de cartas Guerra (WAR). No jogo Guerra, cada jogador retira uma carta do baralho e o que tiver a carta mais alta ganha. Você criará o jogo definindo classes para representar uma carta, um baralho, um jogador e, para concluir, o próprio jogo.

## Cartas

Esta é a classe que modela as cartas do jogo: # <http://tinyurl.com/jj22qv4>

```
class Card:
    suits = ("spades", "hearts",
            "diamonds",
            "clubs")

    values = (None, None, "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack",
            "Queen", "King", "Ace")
    def __init__(self, v, s): """suit + value are ints"""
        self.value = v
        self.suit = s

    def __lt__(self, c2):
        if self.value < c2.value: return True
        if self.value == c2.value: if self.suit < c2.suit: return True
        else:
            return False
        return False

    def __gt__(self, c2):
        if self.value > c2.value: return True
        if self.value == c2.value: if self.suit > c2.suit: return True
        else:
            return False
        return False

    def __repr__(self):
        v = self.values[self.value] +\ " of " + \
            self.suits[self.suit]
        return v
```

A classe **Card** tem duas variáveis de classe, **suits** e **values**. **suits** é uma lista de strings representando todos os naipes que uma carta pode ter: **spades**, **hearts**, **diamonds**, **clubs** (Espadas, Copas, Ouros, Paus). **values** é a lista das strings que representam os diferentes valores numéricos de uma carta: **2-10**, **Jack**, **Queen**, **King** e **Ace** (2 a 10, Valete, Dama, Rei e Ás). Os itens dos dois primeiros índices da lista **values** são **None**, para que as strings da lista coincidam com o índice que as representa – portanto, a string "2" da lista **values** fica no índice 2.

Os objetos **Card** têm duas variáveis de instância: **suit** e **value** – cada uma é representada por um inteiro. Juntas, as variáveis de instância representam qual é o tipo de carta do objeto **Card**. Por exemplo, você pode obter um 2 de copas criando um objeto **Card** e passando para ele os parâmetros 2 (para o naipe) e 1 (para o valor – 1 porque copas fica no índice 1 da lista **suits**).

As definições dos métodos mágicos **\_\_lt\_\_** e **\_\_gt\_\_** permitem comparar dois objetos **Card** em uma expressão com o uso dos operadores **>** (maior que) e **<** (menor que). O código desses métodos determina se a carta é maior ou menor que a outra carta passada como parâmetro. O código dos métodos mágicos também pode manipular casos em que as cartas tiverem o mesmo valor – por exemplo, se as duas cartas forem 10. Se isso ocorrer, os métodos usarão o valor dos naipes no desempate. Os naipes estão organizados por ordem de força na lista **suits** – com o naipe mais forte por último e, portanto, recebendo o índice mais alto, e o naipe menos poderoso recebendo o índice mais baixo.

```
# http://tinyurl.com/j6donnr
```

```
card1 = Card(10, 2)
card2 = Card(11, 3)
print(card1 < card2) >> True # http://tinyurl.com/hc9ktlr
```

```
card1 = Card(10, 2)
card2 = Card(11, 3)
print(card1 > card2) >> False
```

O último método da classe **Card** é o método mágico **\_\_repr\_\_**. Seu código usa as variáveis de instância **value** e **suit** para procurar o valor e o naipe da carta nas listas **values** e **suits**, e os retorna para exibirmos a carta que um objeto **Card** representa: # <http://tinyurl.com/z57hc75>

```
card = Card(3, 2)
print(card)

>> 3 of diamonds
```

## Baralho

Agora você precisa definir uma classe para representar um baralho de cartas: #

<http://tinyurl.com/jz8zfz7>

```
from random import shuffle
class Deck:
def __init__(self):
self.cards = []
for i in range(2, 15): for j in range(4):
self.cards\
.append(Card(i,
j))
shuffle(self.cards)
```

```
def rm_card(self):
if len(self.cards) == 0: return
return self.cards.pop()
```

Quando você inicializar o objeto Deck, os dois loops for de `__init__` criarão objetos Card representando todas as cartas de um baralho de 52 cartas e os acrescentarão à lista cards. O primeiro loop é de 2 a 15 porque o primeiro valor de uma carta é 2 e o último é 14 (o Ás). A cada iteração do loop interno, uma nova carta será criada usando o inteiro do loop externo como valor (isto é, 14 para um Ás) e o inteiro do loop interno como naipe (por exemplo, 2 para Copas). Esse processo criará 52 cartas – uma carta para cada combinação de naipe e valor. Após o método criar as cartas, o método shuffle do módulo random reorganizará os itens aleatoriamente na lista cards, imitando o embaralhar das cartas.

Nosso baralho tem outro método, chamado `rm_card`, que remove e retorna uma carta da lista `cards` ou retorna `None` se ela estiver vazia. Você pode usar a classe `Deck` para criar um novo baralho de cartas e exibir cada carta existente nele: # <http://tinyurl.com/hsv5n6p>

```
deck = Deck()
for card in deck.cards: print(card)
>> 4 of spades >> 8 of hearts ...
```

## Jogador

Você precisa de uma classe para representar cada participante do jogo e registrar suas cartas e quantas rodadas eles venceram: # <http://tinyurl.com/gwyrt2s>

```
class Player:
def __init__(self, name): self.wins = 0
self.card = None
self.name = name
```

A classe `Player` tem três variáveis de instância: `wins` para registrar quantas rodadas um jogador venceu, `card` para representar a carta que um jogador tem atualmente e `name` para registrar o nome de um jogador.



## Jogo

Para concluir, você precisa de uma classe para representar o jogo: # <http://tinyurl.com/huwq8mw>

```
class Game:
    def __init__(self):
        name1 = input("p1 name ") name2 = input("p2 name ") self.deck = Deck()
        self.p1 = Player(name1) self.p2 = Player(name2)
        def wins(self, winner): w = "{} wins this round"
        w = w.format(winner)
        print(w)

    def draw(self, p1n, p1c, p2n, p2c): d = "{} drew {} {} drew {}"
    d = d.format(p1n,
    p1c,
    p2n,
    p2c)
    print(d)

    def play_game(self):
        cards = self.deck.cards print("beginning War!") while len(cards) >= 2: m = "q to
        quit. Any " + \ "key to play:"
        response = input(m)
        if response == 'q': break
        p1c = self.deck.rm_card() p2c = self.deck.rm_card() p1n = self.p1.name
        p2n = self.p2.name
        self.draw(p1n,
        p1c,
        p2n,
        p2c)
        if p1c > p2c:
            self.p1.wins += 1
            self.wins(self.p1.name) else:
            self.p2.wins += 1
            self.wins(self.p2.name)
        win = self.winner(self.p1, self.p2)
        print("War is over.{} wins"
        .format(win))

    def winner(self, p1, p2): if p1.wins > p2.wins: return p1.name
    if p1.wins < p2.wins: return p2.name
    return "It was a tie!"
```

Quando você criar o objeto de jogo, o Python chamará o método `__init__`, e a função `input` coletará o nome dos dois jogadores e os armazenará nas variáveis

**name1** e **name2**. Em seguida, você criará um novo objeto **Deck**, o armazenará na variável de instância **deck** e criará dois objetos **Player** usando os nomes de **name1** e **name2**.

O método **play\_game** da classe **Game** começa o jogo. Há um loop no método que mantém o jogo em andamento enquanto existirem duas ou mais cartas no baralho e enquanto a variável **response** não for igual a **q**. A cada iteração do loop, a entrada do usuário é atribuída à variável **response**. O jogo continuará até o usuário digitar "**q**" ou até que haja menos de duas cartas no baralho.

Duas cartas são desenhadas a cada iteração do loop e o método **play\_game** atribui a primeira carta a **p1** e a segunda a **p2**. Em seguida, ele exibe o nome de cada jogador e a carta que eles retiraram, compara as duas cartas para ver qual é a mais alta, incrementa a variável de instância **wins** para o jogador que tiver a carta mais alta e exibe uma mensagem informando quem venceu.

A classe **Game** também tem um método chamado **winner** que recebe dois objetos de jogador, examina o número de rodadas que eles venceram e retorna o jogador que venceu mais rodadas.

Quando o objeto **Deck** ficar sem cartas, o método **play\_game** exibirá uma mensagem informando que a guerra acabou, chamará o método **winner** (passando tanto **p1** quanto **p2**) e exibirá o resultado – o nome do jogador que venceu.

## Guerra

Aqui está o jogo completo:

```
# http://tinyurl.com/ho7364a
```

```
from random import shuffle
class Card:
    suits = ["spades", "hearts",
            "diamonds",
            "clubs"]
    values = [None, None, "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack",
            "Queen", "King", "Ace"]
```

```
def __init__(self, v, s): """suit + value are ints"""
    self.value = v
    self.suit = s
```

```
def __lt__(self, c2):
    if self.value < c2.value: return True
    if self.value == c2.value: if self.suit < c2.suit: return True
    else:
```

```
return False
return False
```

```
def __gt__(self, c2):
if self.value > c2.value: return True
if self.value == c2.value: if self.suit > c2.suit: return True
else:
return False
return False
```

```
def __repr__(self):
v = self.values[self.value] +\ " of " + \
self.suits[self.suit]
return v
```

```
class Deck:
def __init__(self):
self.cards = []
for i in range(2, 15): for j in range(4):
self.cards\
.append(Card(i,
j))
shuffle(self.cards)
```

```
def rm_card(self):
if len(self.cards) == 0: return
return self.cards.pop()
class Player:
def __init__(self, name): self.wins = 0
self.card = None
self.name = name
```

```
class Game:
def __init__(self):
name1 = input("p1 name ") name2 = input("p2 name ") self.deck = Deck()
self.p1 = Player(name1) self.p2 = Player(name2)
def wins(self, winner): w = "{} wins this round"
w = w.format(winner)
print(w)
```

```
def draw(self, p1n, p1c, p2n, p2c): d = "{} drew {} {} drew {}"
d = d.format(p1n,
p1c,
p2n,
p2c)
```

```
print(d)
```

```
def play_game(self):
cards = self.deck.cards print("beginning War!") while len(cards) >= 2: m = "q to
quit. Any " + \ "key to play:"
response = input(m)
if response == 'q': break
p1c = self.deck.rm_card() p2c = self.deck.rm_card() p1n = self.p1.name
p2n = self.p2.name
self.draw(p1n,
p1c,
p2n,
p2c)
if p1c > p2c:
self.p1.wins += 1
self.wins(self.p1.name) else:
self.p2.wins += 1
self.wins(self.p2.name)
win = self.winner(self.p1, self.p2)
print("War is over.{} wins"
.format(win))
```

```
def winner(self, p1, p2): if p1.wins > p2.wins: return p1.name
if p1.wins < p2.wins: return p2.name
return "It was a tie!"
```

```
game = Game()
game.play_game()
```

```
>> "p1 name "
```

```
...
```

PARTE III

# **Introdução às ferramentas de programação**

# CAPÍTULO 16

## Bash

*“Não consigo pensar em uma função que eu quisesse exercer que não fosse programar computadores. Todo dia crio padrões e estruturas a partir de algo sem forma e resolvo vários pequenos enigmas durante o processo.”*

### – Peter Van Der Linden

Neste capítulo, você aprenderá a usar uma **interface de linha de comando** chamada **Bash**. A interface de linha de comando é um programa no qual digitamos instruções para o sistema operacional executar. O Bash é uma implementação específica de uma interface de linha de comando que vem com a maioria dos sistemas operacionais de padrão Unix (Unix-like). De agora em diante, usarei os termos interface de linha de comando e **linha de comando** de maneira intercambiável.

Quando consegui meu primeiro emprego como programador, cometi o erro de passar todo o meu tempo praticando programação. É claro que é preciso ser um programador talentoso para programar profissionalmente. No entanto, também é preciso ter várias outras habilidades, como saber usar a linha de comando. A linha de comando será o “centro de controle” para tudo o que você fizer que não envolver escrever códigos.

Posteriormente neste livro, por exemplo, você aprenderá a usar gerenciadores de pacotes para instalar programas de outras pessoas e sistemas de controle de versões para colaborar com outros programadores. Você operará essas duas ferramentas a partir da linha de comando. Além disso, quase todos os softwares escritos atualmente envolvem o acesso a dados pela internet e a maioria dos servidores web mundiais executa o Linux. Esses servidores não têm uma interface de usuário; só podemos acessá-los pela linha de comando.

A linha de comando, os gerenciadores de pacotes, as expressões regulares e o controle de versões são ferramentas básicas do arsenal de um programador. Todas as pessoas das equipes nas quais trabalhei eram especialistas nesses assuntos.

Quando você estiver programando profissionalmente, também exigirão que você seja especialista nessas áreas. Levei muito tempo para chegar a esse ponto e queria ter começado a aprender a usar essas ferramentas mais cedo.

## Acompanhamento

Se você estiver usando o Ubuntu ou o Unix, seu computador já terá vindo com o Bash. O Windows, entretanto, vem com uma interface de linha de comando chamada **Prompt de Comando** (que você não poderá usar neste capítulo). A versão mais recente do Windows 10 vem com o Bash. Você pode encontrar instruções sobre

como usar o Bash no Windows 10 em <http://theselftaughtprogrammer.io/windows10bash>.

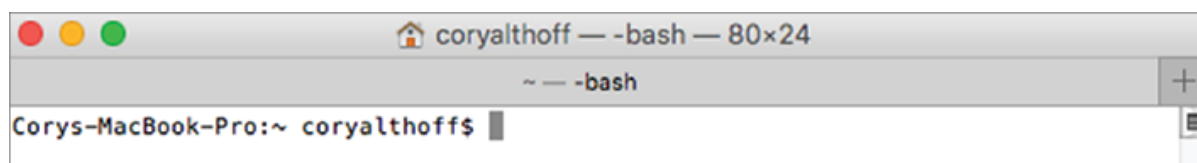
Se você estiver usando o Windows, poderá utilizar a Amazon AWS para obter um servidor web gratuito executando o Ubuntu. É fácil instalar um servidor, e a AWS é amplamente empregada no mundo da programação, logo, isso lhe dará uma experiência importante. Acesse <http://theselftaughtprogrammer.io/aws> para começar.

Se você estiver usando o Windows, e não quiser instalar um servidor, pode seguir os exemplos acessando <http://theselftaughtprogrammer.io/bashapp>, onde encontrará um link para uma aplicação web que emula o Bash e que pode ser utilizada para o acompanhamento da maioria dos exemplos.

Depois deste capítulo, você poderá seguir os exemplos dos dois capítulos seguintes usando o Prompt de Comando do Windows. Para encontrá-lo, procure **Prompt de Comando** na **Janela Executar**.

## Busca do Bash

Você pode encontrar o Bash em seu computador procurando **Terminal** no ícone intitulado **Search your computer and online resources** (pesquise seu computador e recursos online) se estiver usando o Ubuntu ou na busca com o Spotlight se estiver usando um Mac.



## Comandos

O Bash é semelhante ao shell Python. Você digitará comandos, que são como as funções do Python, no Bash. Em seguida, digitará um espaço e os parâmetros (se houver) que deseja passar para o comando. Pressione a tecla **Enter** e o Bash retornará o resultado. O comando **echo** é parecido com a função **print** do Python.

Sempre que você se deparar com um cifrão seguido de um comando, neste livro ou em alguma documentação de programação, significa que é preciso digitar o comando na linha de comando:

```
# http://tinyurl.com/junx62n
```

```
$ echo Hello, World!
```

```
>> Hello, World!
```

Primeiro, você digitou o comando **echo** no Bash, seguido de um espaço e de **Hello, World!** como parâmetro. Quando pressionar **Enter**, **Hello, World!** será exibido no

Bash.

Você pode usar os programas que instalou, como o Python, a partir da linha de comando. Insira o comando **python3** (quanto escrevi este texto, a aplicação web Bash não vinha com o Python 3. Digite **python** para usar o Python 2):

```
# http://tinyurl.com/htoospk
```

```
$ python3
```

Agora poderá executar o código Python:

```
# http://tinyurl.com/jk2acua
```

```
print("Hello, World!")
```

```
>> Hello, World!
```

Insira **exit()** para sair do Python.

## Comandos recentes

Você pode rolar pelos seus comandos recentes pressionando as setas para cima e para baixo no Bash. Para ver uma lista de todos os seus comandos recentes use o comando **history**:

```
# http://tinyurl.com/go2spbt
```

```
$ history
```

```
>> 1. echo Hello, World!
```

## Caminhos relativos versus absolutos

Um sistema operacional é composto de diretórios e arquivos. Um **diretório** é apenas outra palavra para indicar uma pasta em seu computador. Todos os diretórios e arquivos têm um caminho, um endereço onde o diretório ou o arquivo existe no sistema operacional. Quando você usar o Bash, estará sempre em um diretório, localizado em um caminho (path) específico. Você pode usar o comando **pwd**, que é a abreviação de print **working directory** (o diretório de trabalho é o diretório em que você está atualmente), para exibir o nome do diretório em que está:

```
# http://tinyurl.com/hptsqhp
```

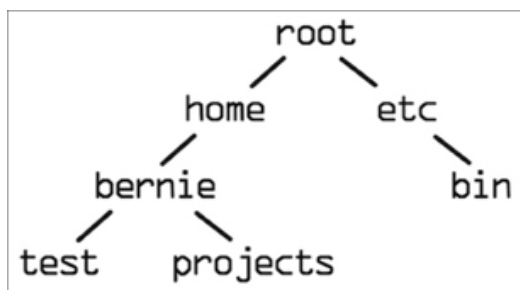
```
$ pwd
```

```
>> Userscoryalthoff
```

O sistema operacional representa seus diretórios, e a localização do diretório do usuário, com uma árvore. Em ciência da computação, uma árvore é um conceito importante chamado estrutura de dados (abordado na Parte IV). Em uma árvore há



uma raiz no topo. A raiz pode ter galhos, cada um dos galhos pode ter mais galhos, esses galhos também podem ter galhos, e assim por diante. A imagem a seguir é um exemplo de uma árvore que representa os diretórios de um sistema operacional:



Cada galho da árvore é um diretório, inclusive a raiz. A árvore mostra como os diretórios se conectam. Sempre que estiver usando o Bash, você estará em um local na árvore de seu sistema operacional. Um caminho (path) é uma maneira de expressar esse local. Existem duas maneiras de fornecer o caminho de um arquivo ou diretório em um sistema operacional de padrão Unix: com um caminho absoluto ou com um caminho relativo.

Um caminho absoluto fornece a localização de um arquivo ou diretório começando pelo diretório raiz. Ele é composto pelo nome dos diretórios da árvore, na ordem de sua proximidade com a raiz e separados por barras. O caminho absoluto do diretório **bernie** (no sistema operacional ilustrado na imagem anterior) é **homebernie**. A primeira barra representa o diretório raiz. O diretório **home** vem depois dela. Em seguida, há outra barra e o diretório **bernie**.

Outra maneira de especificar um local em seu computador é com um caminho relativo. Em vez de começar no diretório raiz, o caminho relativo começa com o diretório de trabalho atual. Se o caminho não começar com uma barra, o Bash saberá que você está usando um caminho relativo. Se você estivesse no diretório **home** da imagem de árvore do exemplo anterior, o caminho relativo do diretório **projects** seria **bernie/projects**. Ainda a partir do diretório **home**, o caminho relativo de **bernie** seria simplesmente **bernie**. Se você estivesse no diretório **root**, o caminho relativo de **projects** seria **home/bernie/projects**.

## Navegação

Você pode mudar de diretório passando um caminho absoluto ou relativo como parâmetro para o comando **cd**. Insira o comando **cd** seguido do caminho absoluto / para navegar para o diretório raiz do seu sistema operacional:

```
# http://tinyurl.com/hjgz79h
```

```
$ cd /
```

Para saber onde você está, use o comando **pwd**:

```
# http://tinyurl.com/j6ax35s
```

```
$ pwd
```

```
>> /
```

O comando para a listagem de diretórios, **ls**, exibe os diretórios e as pastas do diretório de trabalho atual:

```
# http://tinyurl.com/gw4d5yw
```

```
$ ls
```

```
>> bin dev initrd.img lost+found ...
```

Você pode criar um novo diretório passando o seu nome para o comando de criação de diretórios **mkdir** (make directory). Os nomes de diretório não podem ter espaços. Navegue para seu diretório home (~ é um atalho para o diretório home em sistemas operacionais de padrão Unix) e use o comando **mkdir** para criar um novo diretório chamado **tstp**:

```
# http://tinyurl.com/zavhjeq
```

```
$ cd ~
```

```
$ mkdir tstp
```

Verifique se o diretório foi criado com o comando **ls**.

```
# http://tinyurl.com/hneq2f6
```

```
$ ls
```

```
>> tstp
```

Agora use o comando **cd** para entrar no diretório **tstp** passando o caminho relativo de **tstp** como parâmetro:

```
# http://tinyurl.com/zp3nb2l
```

```
$ cd tstp
```

Você pode usar o comando **cd** seguido de dois pontos para mover-se um diretório para trás (um nível para cima na árvore):

```
# http://tinyurl.com/z2gevk2
```

```
$ cd ..
```

Para excluir um diretório, use o comando de remoção de diretórios **rmdir** (remove directory). Use-o para remover o diretório **tstp**:

```
# http://tinyurl.com/jkjj06s
```

```
$ rmdir tstp
```

Para concluir, verifique se excluiu o diretório com o comando **ls**.

```
# http://tinyurl.com/z32xn2n
```

```
$ ls
```

## Flags

Os comandos têm um conceito chamado flags que permite que o emissor do comando altere seu comportamento. As flags são opções para os comandos que podem ter um valor **True** ou **False**. Por padrão, inicialmente todas as flags de um comando são configuradas com **False**. Se você adicionar uma flag a um comando, o Bash configurará seu valor com **True** e o comportamento do comando mudará. Para configurar uma flag com **True**, insira um (-) ou dois (- -) hífen na frente do nome da flag (dependendo do sistema operacional).

Por exemplo, você pode adicionar **--author** ao comando **ls** para configurar a flag **author** com **True**. A inclusão dessa flag no comando **ls** alterará seu comportamento. Quando você adicionar a flag, o comando **ls** exibirá todos os diretórios e arquivos de um diretório, mas também exibirá o nome do autor, a pessoa que os criou.

No Unix, use um único hífen na frente de uma flag:

```
# http://tinyurl.com/j4y5kz4
```

```
$ ls -author
```

```
>> drwx-----+ 13 coryalthoff 442B Sep 16 17:25 Pictures
```

```
>> drwx-----+ 25 coryalthoff 850B Nov 23 18:09 Documents
```

E no Linux, use dois:

```
# http://tinyurl.com/hu9c54q
```

```
$ ls --author
```

```
>> drwx-----+ 13 coryalthoff 442B Sep 16 17:25 Pictures
```

```
>> drwx-----+ 25 coryalthoff 850B Nov 23 18:09 Documents
```

## Arquivos ocultos

Seu sistema operacional e muitos programas existentes em seu computador armazenam dados em arquivos ocultos. Arquivos ocultos são arquivos que, por padrão, não são exibidos para os usuários porque se alterados podem afetar os programas que dependem deles. Esses arquivos começam com um ponto, por exemplo, **.hidden**. Você pode visualizar os arquivos ocultos adicionando a flag **-a**,

que é a abreviação de all (todos), ao comando **ls**. O comando **touch** cria um novo arquivo a partir da linha de comando.

Já que o comando **touch** cria um novo arquivo, use-o para criar um arquivo oculto chamado **.self\_taught**:

```
# http://tinyurl.com/hfawo8t
```

```
$ touch .self_taught
```

Verifique se consegue vê-lo com os comandos **ls** e **ls -a**.

## Pipes

Em sistemas operacionais de padrão Unix, o caractere de barra vertical (|) chama-se pipe. Você pode usar um pipe para passar a saída de um comando como entrada para outro comando. Por exemplo, você pode usar a saída do comando **ls** como entrada do comando **less** (certifique-se de não estar em um diretório vazio):

```
# http://tinyurl.com/zjne9f5
```

```
$ ls | less
```

```
>> Applications ...
```

O resultado é um arquivo de texto com a saída de **ls** aberta no programa **less** (pressione **q** para sair de less).

## Variáveis de ambiente

Variáveis de ambiente são variáveis armazenadas no sistema operacional que os programas podem usar para obter dados sobre o ambiente em que eles estão sendo executados, como o nome do computador no qual o programa está em execução ou o nome do usuário do sistema operacional que o está executando. Você pode criar uma nova variável de ambiente no Bash com a sintaxe **export nome\_da\_variável=[valor\_da\_variável]**. Para referenciar uma variável de ambiente no Bash é preciso inserir um cifrão na frente do seu nome:

```
# http://tinyurl.com/jjbc9v2
```

```
$ export x=100
```

```
$ echo $x
```

```
>> 100
```

Uma variável de ambiente criada desta forma só existirá na janela do Bash na qual você a criar. Se você sair da janela do Bash onde criou a variável de ambiente, reabri-la e digitar **echo \$x**, o Bash não exibirá mais **100** porque a variável de ambiente **x** não existirá mais.

Você pode fazer uma variável de ambiente persistir adicionando-a a um arquivo oculto usado pelos sistemas operacionais de padrão Unix, localizado em seu diretório **home**, chamado **.profile**. Use sua GUI para navegar para seu diretório home. Você pode encontrar o caminho de arquivo de seu diretório home a partir da linha de comando com **pwd ~**. Use um editor de texto para criar um arquivo chamado **.profile**. Digite **export x=100** na primeira linha do arquivo e salve-o. Feche e reabra o Bash, e deve conseguir exibir a variável de ambiente **x**:

```
# http://tinyurl.com/j5wjwdf
```

```
$ echo $x
```

```
>> 100
```

A variável persistirá enquanto estiver em seu arquivo **.profile**. Você pode excluir a variável removendo-a desse arquivo.

## Usuários

Os sistemas operacionais podem ter vários usuários. Um usuário é uma pessoa que usa o sistema operacional. Cada usuário recebe um nome e uma senha, o que permite que ele faça login e use o sistema operacional. Cada usuário também tem um conjunto de permissões: operações que ele tem permissão para executar. Você pode exibir o nome do usuário de seu sistema operacional com o comando **whoami** (os exemplos desta seção não funcionarão no Bash no Windows ou na aplicação web Bash):

```
$ whoami
```

```
>> coryalthoff
```

Normalmente, somos o usuário que criamos quando instalamos o sistema operacional. No entanto, esse usuário não é o mais poderoso do sistema operacional. O usuário de nível mais alto, que é o que tem o conjunto de permissões mais alto, chama-se usuário root. Todo sistema tem um usuário root que pode, por exemplo, criar e excluir outros usuários.

Por razões de segurança, geralmente não fazemos login como usuário root. Em vez disso, colocamos o comando **sudo** (superuser do) antes dos comandos que precisamos emitir como usuário root. **sudo** nos permite emitir comandos como usuário root sem comprometer a segurança do sistema como ocorreria se fizéssemos login como root. Aqui está um exemplo de uso do comando **echo** com **sudo**:

```
$ sudo echo Hello, World!
```

```
>> Hello, World!
```

Se você tiver definido uma senha em seu computador, será solicitado a fornecê-la quando emitir um comando com **sudo**. **sudo** remove as proteções que nos impedem

de prejudicar o sistema operacional, logo, nunca emita um comando com **sudo** a não ser que tenha certeza de que ele não danificará seu sistema operacional.

## Saiba mais

Abordei apenas os aspectos básicos do Bash neste capítulo. Para saber mais sobre o uso do Bash, acesse <http://theselftaughtprogrammer.io/bash>.

## Vocabulário

**Bash:** Programa que vem com a maioria dos sistemas operacionais de padrão Unix no qual digitamos instruções para o sistema operacional executar.

**Caminho:** Uma maneira de representar a localização de um arquivo ou diretório no sistema operacional.

**Caminho absoluto:** Localização de um arquivo começando no diretório raiz.

**Caminho relativo:** Localização de um arquivo ou diretório começando no diretório de trabalho atual.

**Diretório:** Outro termo usado para designar uma pasta existente no computador.

**Diretório de trabalho:** O diretório no qual você está atualmente.

**Interface de linha de comando:** Uma interface de linha de comando é um programa no qual digitamos instruções para o sistema operacional executar.

**Linha de comando:** Outro nome dado à interface de linha de comando.

**Permissões:** Operações que os usuários do sistema operacional têm permissão para executar.

**Pipe:** O caractere |. Em sistemas operacionais de padrão Unix, podemos usar um pipe para passar a saída de um comando como entrada de outro comando.

**Prompt de comando:** Interface de linha de comando que vem com o Windows.

**\$PATH:** Quando digitamos um comando no shell de comando Bash, ele o procura em todos os diretórios armazenados em uma variável de ambiente chamada \$PATH.

**Usuário:** Uma pessoa que usa o sistema operacional.

**Usuário root:** O usuário de nível mais alto, aquele que tem o conjunto de permissões mais alto.

**Variáveis de ambiente:** Variáveis nas quais o sistema operacional e outros programas armazenam dados.

## Desafios

1. Exiba **Selftaught** no Bash.
2. Navegue para seu diretório home a partir de outro diretório usando um caminho absoluto e um relativo.
3. Crie uma variável de ambiente chamada **\$python\_projects** que seja um caminho absoluto para o diretório no qual você mantém seus arquivos Python. Salve a variável em seu arquivo **.profile** e, em seguida, use o comando **cd \$python\_projects** para navegar para esse local.

Soluções: <http://tinyurl.com/zdeyg8y>.

# CAPÍTULO 17

## Expressões regulares

*“Falar é fácil. Mostre-me o código.”*

### – Linus Torvalds

Muitas linguagens de programação e sistemas operacionais suportam **expressões regulares**: uma “sequência de caracteres que definem um padrão de busca”.<sup>1</sup> As expressões regulares são úteis porque podemos usá-las para procurar um padrão complexo em um arquivo ou em outros dados. Por exemplo, você poderia usar uma expressão regular para procurar todos os números existentes em um arquivo. Neste capítulo, você aprenderá a definir expressões regulares e a passá-las para **grep**, um comando de sistemas operacionais de padrão Unix que procura padrões em um arquivo e retorna qualquer texto encontrado que corresponda ao padrão. Também aprenderá a usar expressões regulares para procurar padrões nas strings em Python.

### Preparação

Para começar, crie um arquivo chamado **zen.txt**. Na linha de comando (certifique-se de estar dentro do diretório no qual criou **zen.txt**), insira o comando **python3 -c "import this"**. Ele exibirá *The Zen of Python* (O Zen do Python), um poema de Tim Peters:

Beautiful is better than ugly. (*Bonito é melhor que feio*)

**Explicit is better than implicit.** (*Explícito é melhor que implícito*)

Simple is better than complex. (*Simples é melhor que complexo*)

**Complex is better than complicated.** (*Complexo é melhor que complicado*)

Flat is better than nested. (*Linear é melhor que aninhado*)

**Sparse is better than dense.** (*Esparso é melhor que denso*)

Readability counts. (*Legibilidade faz diferença*)

**Special cases aren't special enough to break the rules.** (*Casos especiais não são especiais o suficiente para quebrar as regras*)

Although practicality beats purity. (*Ainda que praticidade vença a pureza*)

**Errors should never pass silently.** (*Erros nunca devem passar silenciosamente*)

Unless explicitly silenced. (*A menos que sejam explicitamente silenciados*)



**In the face of ambiguity, refuse the temptation to guess.** (*Diante da ambiguidade, recuse a tentação de adivinhar*)

There should be one – and preferably only one – obvious way to do it. (*Deveria haver um – e preferencialmente só um – modo óbvio para fazer algo*)

**Although that way may not be obvious at first unless you're Dutch.** (*Embora esse modo possa não parecer óbvio a princípio a menos que você seja holandês*)

Now is better than never. (*Agora é melhor que nunca*)

**Although never is often better than *right* now.** (*Embora nunca frequentemente seja melhor que já*)

If the implementation is hard to explain, it's a bad idea. (*Se a implementação é difícil de explicar, é uma má ideia*)

**If the implementation is easy to explain, it may be a good idea.** (*Se a implementação é fácil de explicar, pode ser uma boa ideia*)

Namespaces are one honking great idea – let's do more of those! (*Namespaces são uma grande ideia – vamos ter mais dessas!*)

A flag `-c` informará ao Python que você passará para ele uma string contendo código Python. O Python executará então o código. Quando o Python executar `import this`, ele exibirá *The Zen of Python* (uma mensagem oculta em código como esse poema chama-se **Easter egg**). Insira a função `exit()` no Bash para sair do Python e, em seguida, copie e cole *The Zen of Python* no arquivo `zen.txt`.

Por padrão, no Ubuntu o comando `grep` exibe palavras coincidentes em vermelho na sua saída, mas no Unix não é assim. Se você usar um Mac, pode alterar isso definindo as variáveis de ambiente a seguir no Bash: `# http://tinyurl.com/z9prphe`

```
$ export GREP_OPTIONS='--color=always'
```

Lembre-se, a definição de uma variável de ambiente no Bash não é permanente, logo, se você sair do Bash, terá de definir as variáveis de ambiente novamente na próxima vez que o abrir. Você pode adicionar as variáveis de ambiente ao seu arquivo `.profile` para torná-las permanentes.

## Busca simples

O comando `grep` recebe dois parâmetros: uma expressão regular e o caminho do arquivo no qual o padrão definido na expressão regular será procurado. O tipo mais simples de padrão para a busca com uma expressão regular é o da busca simples,

uma string de palavras que coincida com a mesma string de palavras. Para ver um exemplo de uma busca simples, insira o comando a seguir no diretório no qual você criou o arquivo `zen.txt`: # <http://tinyurl.com/jgh3x4c>

```
$ grep Beautiful zen.txt
```

```
>> Beautiful is better than ugly.
```

No comando que você executou, o primeiro parâmetro, **Beautiful**, é a expressão regular, e o segundo parâmetro, **zen.txt**, é o caminho do arquivo onde ela será procurada. O Bash exibiu a linha “**Beautiful is better than ugly.**” com **Beautiful** em vermelho porque essa é a palavra que corresponde à expressão regular.

Se você alterar a expressão regular do exemplo anterior de **Beautiful** para **beautiful**, o **grep** não encontrará nada: # <http://tinyurl.com/j2z6t2r>

```
$ grep beautiful zen.txt
```

Você pode ignorar a diferença entre letras maiúsculas e minúsculas com a flag **-i**: # <http://tinyurl.com/zchmrdq>

```
$ grep -i beautiful zen.txt
```

```
>> Beautiful is better than ugly.
```

Por padrão, o **grep** exibirá a linha inteira (do arquivo) na qual encontrou uma ocorrência. Você pode adicionar a flag **-o** para exibir apenas as palavras que correspondam ao padrão que foi passado: # <http://tinyurl.com/zfcdnmX>

```
$ grep -o Beautiful zen.txt
```

```
>> Beautiful Podemos usar expressões regulares em Python com sua biblioteca interna re (regular expressions). O módulo re vem com um método chamado findall. Ele recebe uma expressão regular e uma string como parâmetros e retorna uma lista com todos os itens da string que corresponderem ao padrão: # http://tinyurl.com/z9q2286
```

```
import re
```

```
l = "Beautiful is better than ugly."
```

```
matches = re.findall("Beautiful", l)
print(matches)
```

```
>> ['Beautiful']
```

Nesse exemplo, o método **findall** encontrou uma ocorrência (**Beautiful**) e retornou uma lista exibindo-a como primeiro item.

Você pode ignorar a diferença entre letras maiúsculas e minúsculas no método

`findall` passando `re.IGNORECASE` para ele como terceiro parâmetro: # <http://tinyurl.com/jzeonne>

```
import re
```

```
l = "Beautiful is better than ugly."
```

```
matches = re.findall("beautiful", l,  
re.IGNORECASE)
```

```
print(matches)
```

```
>> ['Beautiful']
```

## Busca no começo e no fim

Podemos criar expressões regulares que busquem padrões complexos adicionando a elas caracteres especiais que em vez de procurar um caractere, definam uma regra. Por exemplo, você pode usar o caractere de acento circunflexo (^) para criar uma expressão regular que só encontre um padrão se ele ocorrer no começo de uma linha: # <http://tinyurl.com/gleyzan>

```
$ grep ^If zen.txt
```

```
>> If the implementation is hard to explain, it's a bad idea.
```

```
>> If the implementation is easy to explain, it may be a good idea.
```

Da mesma forma, você pode usar o cifrão (\$) para só encontrar as linhas que terminem com um padrão: # <http://tinyurl.com/zkvpc2r>

```
$ grep idea.$ zen.txt
```

```
>> If the implementation is hard to explain, it's a bad idea.
```

```
>> If the implementation is easy to explain, it may be a good idea.
```

Nesse caso, `grep` ignorou **"Namespaces are one honking great idea -- let's do more of those!"** porque, embora a linha contenha a palavra `idea`, ela não termina com `idea`.

Aqui está um exemplo do uso do símbolo de acento circunflexo (^) em Python (você precisa passar `re.MULTILINE` como terceiro parâmetro para `findall` para procurar ocorrências em todas as linhas de uma string com várias linhas): # <http://tinyurl.com/zntqzc9>

```
import re
```

```
zen = """Although never is often better than
```

```
right now.  
If the implementation  
is hard to explain,  
it's a bad idea.  
If the implementation  
is easy to explain,  
it may be a good  
idea. Namespaces  
are one honking  
great idea -- let's  
do more of those!  
"""
```

```
m = re.findall("^If",  
zen,  
re.MULTILINE)  
print(m)  
  
>> ['If', 'If']
```

## Busca de vários caracteres

Você pode definir um padrão que busque vários caracteres inserindo-os dentro de colchetes em uma expressão regular. Se você inserir `[abc]` em uma expressão regular, ela procurará **a**, **b** ou **c**. No próximo exemplo, em vez de procurar texto em seu arquivo `zen.txt`, você procurará uma ocorrência em uma string passando-a para `grep` com o pipe: # <http://tinyurl.com/jf9qzuz>

```
$ echo Two too. | grep -i t[ow]o
```

```
>> Two too
```

A saída do comando `echo` é passada para `grep` como entrada e, portanto, você não precisa especificar o parâmetro de arquivo para `grep`. O comando exibe tanto `two` quanto `too`, porque a expressão regular está procurando um `t`, seguido de um `o` ou um `w`, seguido de um `o`.

Em Python:

```
# http://tinyurl.com/hg9sw3u
```

```
import re
```

```
string = "Two too."
```

```
m = re.findall("t[ow]o",  
string,  
re.IGNORECASE)  
print(m)  
  
>> ['Two', 'too']
```

## Busca de dígitos

Você pode procurar dígitos em uma string com `[[:digit:]]`: # <http://tinyurl.com/gm8o6gb>

```
$ echo 123 hi 34 hello. | grep [[:digit:]]
```

```
>> 123 hi 34 hello.
```

E com `\d` em Python: # <http://tinyurl.com/z3hr4q8>

```
import re
```

```
line = "123?34 hello?"
```

```
m = re.findall("\d",  
line,  
re.IGNORECASE)
```

```
print(m)
```

```
>> ['1', '2', '3', '3', '4']
```

## Repetição

O símbolo de asterisco (\*) adiciona repetição às expressões regulares. Com um asterisco, “the preceding item will be matched zero or more times” (o item anterior será procurado uma ou mais vezes).<sup>2</sup> Por exemplo, você pode usar um asterisco para encontrar `tw` seguido de qualquer quantidade da letra “o”: # <http://tinyurl.com/j8vbwq8>

```
$ echo two twoo not too. | grep -o two*
```

```
>> two >> twoo
```

Em uma expressão regular, um ponto (.) corresponde a qualquer caractere. Se você colocar um asterisco depois do ponto, ele instruirá a expressão regular a procurar qualquer caractere zero ou mais vezes. Você pode usar um ponto seguido de um asterisco para procurar tudo que existe entre dois caracteres: # <http://tinyurl.com/h5x6cal>

```
$ echo __hello__there | grep -o __.*__
```

```
>> __hello__
```

A expressão regular `__.*__` procura qualquer caractere que exista entre (e incluindo) os dois underscores duplos. Um asterisco é **ganancioso** (greedy), o que significa que ele tentará encontrar o máximo de texto que puder. Por exemplo, se você adicionar mais palavras com underscores duplos, a expressão regular do exemplo anterior procurará tudo o que existe do primeiro ao último underscore: #

<http://tinyurl.com/j9v9t24>

```
$ echo __hi__bye__hi__there | grep -o __.*__  
>> __hi__bye__hi__
```

Nem sempre vamos querer procurar padrões gananciosamente. Podemos colocar um ponto de interrogação depois de um asterisco para que a expressão regular seja **não gananciosa**. Uma expressão regular não gananciosa procura o menor número de ocorrências possível. Nesse caso, ela pararia de procurar no primeiro underscore duplo que encontrasse, em vez de encontrar tudo entre a primeira e a última ocorrência. Grep não tem uma busca não gananciosa, mas em Python você pode usar um ponto de interrogação para usar a busca não gananciosa: # <http://tinyurl.com/j399sq9>

```
import re  
  
t = "__one__ __two__ __three__"  
  
found = re.findall("__.*?", t)  
for match in found:  
    print(match)  
  
>> __one__  
>> __two__  
>> __three__
```

Podemos usar a busca não gananciosa em Python para criar o jogo Mad Libs (se você não se lembra do Mad Libs, é um jogo com um parágrafo de texto com várias palavras faltando que os jogadores são solicitados a fornecer): # <http://tinyurl.com/ze6oyua>

```
import re  
  
text = """Giraffes have aroused the curiosity of __PLURAL_NOUN__  
since earliest times. The  
giraffe is the tallest of all  
living __PLURAL_NOUN__, but  
scientists are unable to  
explain how it got its long  
__PART_OF_THE_BODY__. The  
giraffe's tremendous height,  
which might reach __NUMBER__  
__PLURAL_NOUN__, comes from  
it legs and __BODYPART__.  
"""
```

```
def mad_libs(mls):
    """
    :param mls: String
    with parts the user
    should fill out surrounded
    by double underscores.
    Underscores cannot
    be inside hint e.g., no
    __hint_hint__ only
    __hint__.
    """
    hints = re.findall("__.*?__", mls)
    if hints is not None:
        for word in hints:
            q = "Enter a {}"\
                .format(word)
            new = input(q)
            mls = mls.replace(word,
                               new,
                               1)
        print('\n')
        mls = mls.replace("\n", "")
        print(mls)
    else:
        print("invalid mls")
```

```
mad_libs(text)
```

```
>> enter a __PLURAL_NOUN__
```

Nesse exemplo, você usou o método `re.findall` para obter uma lista de todas as palavras da variável `text` inseridas entre underscores duplos (cada uma é uma dica do tipo de palavra que o usuário precisa substituir). Em seguida, percorreu a lista com um loop e usou cada dica para solicitar à pessoa que está usando o programa para fornecer uma nova palavra. Você criou então uma nova string, substituindo a dica pela palavra fornecida pelo usuário. No fim do loop, a nova string está sendo exibida com todas as palavras que você coletou do usuário.

## Escape

Você pode escapar caracteres (ignorar o significado de um caractere e procurá-lo) em expressões regulares como fez anteriormente com as strings em Python, prefixando um caractere na expressão com uma barra invertida (`\`): # <http://tinyurl.com/zkbumfj>.

```
$ echo I love $ | grep \$
```

```
>> I love $
```

Normalmente, o cifrão significa que uma ocorrência só é válida se ocorrer no fim da linha, mas já que você escapou o símbolo, alternativamente sua expressão regular procurará o cifrão.

Em Python, ficaria assim:

```
# http://tinyurl.com/zy7pr4l
```

```
import re
```

```
line = "I love $"
```

```
m = re.findall("\\$",  
line,  
re.IGNORECASE)
```

```
print(m)
```

```
>> ['$']
```

## Ferramenta de expressão regular

Fazer uma expressão regular encontrar um padrão é complexo. Acesse <http://theselftaughtprogrammer.io/regex> para ver uma lista de ferramentas que o ajudarão a criar expressões regulares perfeitas.

## Vocabulário

**Easter egg (ovo de Páscoa):** Uma mensagem oculta em código.

**Expressões regulares:** “Sequência de caracteres que define um padrão de busca”.<sup>3</sup>

**Gananciosa:** Uma expressão regular que é gananciosa tenta encontrar o máximo de texto que puder.

**Não gananciosa:** Uma expressão regular não gananciosa procura a menor quantidade de ocorrências possível.

## Desafios

1. Escreva uma expressão regular que procure a palavra Dutch em *The Zen of Python*.
2. Crie uma expressão regular que encontre todos os dígitos da string "Arizona 479, 501, 870. California 209, 213, 650".
3. Crie uma expressão regular que encontre qualquer palavra que comece com



qualquer caractere e seja seguida de duas letras “o”. Em seguida, use o módulo `re` do Python para encontrar `boo` e `loo` na frase "The ghost that says boo haunts the loo".

Soluções: <http://tinyurl.com/jmlkvxm>.

---

1 [https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression)

2 [http://tldp.org/LDP/Bash-Beginners-Guide/html/sect\\_04\\_01.html](http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_04_01.html)

3 [https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression)

# CAPÍTULO 18

## Gerenciadores de pacotes

*“Todo programador é um autor.”*

### – Sercan Leylek

Um **gerenciador de pacotes** é um programa que instala e gerencia outros programas. Eles são úteis porque com frequência precisamos usar outros programas para criar novos softwares. Por exemplo, os desenvolvedores web costumam usar um **web framework**: um programa que ajuda a criar um site. Os programadores usam gerenciadores de pacotes para instalar web frameworks, assim como vários outros programas. Neste capítulo, você aprenderá a usar o gerenciador de pacotes **pip**.

## Pacotes

Um **pacote** é um software “empacotado” para distribuição – ele inclui os arquivos que compõem o programa real, assim como **metadados**: dados sobre dados, como o nome do software, o número de versão, e **dependências** – os programas dos quais um programa depende para ser executado apropriadamente. Você pode usar um gerenciador de pacotes para baixar um pacote e instalá-lo como um programa em seu computador. O gerenciador de pacotes manipulará o download de qualquer dependência que o pacote tiver.

## Gerenciador de pacotes pip

Nesta seção, mostrarei como usar o pip, um gerenciador de pacotes para Python, para você baixar pacotes Python. Uma vez que você tiver baixado um pacote com o pip, poderá importá-lo como um módulo em um programa Python. Primeiro, verifique se o pip está instalado em seu computador abrindo o Bash, ou o Prompt de Comando se estiver usando o Windows, e inserindo o comando **pip3**:

```
# http://tinyurl.com/hmookdf
```

```
$ pip3
```

```
>> Usage: pip3 <command> [options]
```

```
Commands:
```

```
install Install packages.
```

```
download Download packages. ...
```

Quando você inserir o comando, uma lista de opções deve ser exibida. O pip virá com o Python quando você baixá-lo, mas isso não ocorria em versões anteriores. Você verá uma mensagem de erro “command not found” (ou algo semelhante, dependendo do seu shell) se o pip não estiver instalado em seu computador. Acesse

<http://www.theseftaughtprogrammer.io> para ver instruções sobre sua instalação.

Você pode instalar um novo pacote com `pip3 install [nome_pacote]`. O pip instalará novos pacotes em uma pasta em seu diretório Python chamada **site-packages**. Para encontrar uma lista de todos os pacotes Python disponíveis para download acesse <https://pypi.python.org/pypi>. Existem duas maneiras de especificar o pacote a ser baixado – com o nome do pacote, ou com o nome do pacote seguido de dois sinais de igualdade (==), e do número da versão que você deseja baixar. Se você usar o nome do pacote, o pip baixará a versão mais recente. A segunda opção permite baixar uma versão específica do pacote, em vez da mais atual. Veja como instalar o Flask, um pacote Python para a criação de sites no Ubuntu e Unix:

```
# http://tinyurl.com/hchso7u
```

```
$ sudo pip3 install Flask==0.11.1
```

```
>> Password:
```

```
>> Successfully installed flask-0.11.1
```

No Windows é preciso usar a linha de comando como administrador. Clique com o botão direito do mouse no ícone do prompt de comando e selecione **Run as administrator** (Executar como administrador).

Dentro do Prompt de Comando, insira:

```
# http://tinyurl.com/hyxm3vt
```

```
$ pip3 install Flask==0.11.1
```

```
>> Successfully installed flask-0.11.1
```

Com esse comando, o pip instalará o módulo **Flask** na pasta **site-packages** de seu computador.

Agora você pode importar o módulo **Flask** em um programa. Crie um novo arquivo Python, adicione o código a seguir e execute o programa:

```
# http://tinyurl.com/h59sdyu
```

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route('/')
```

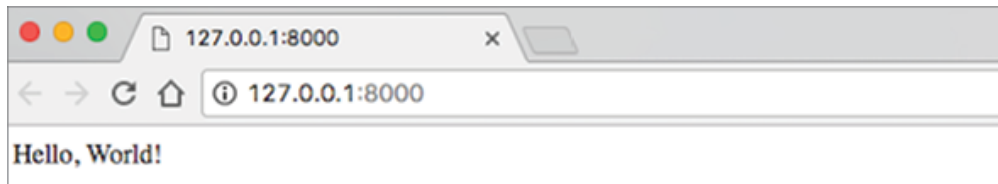
```
def index():
```

```
    return "Hello, World!"
```

```
app.run(port='8000')
```

```
>> * Running on http://127.0.0.1:8000/ (Press CTRL+C to quit)
```

Navegue então para `http://127.0.0.1:8000/` em seu navegador web e você deve ver um site exibindo **Hello, World!**



O módulo **Flask** permite criar um servidor web e um site rapidamente. Acesse <https://flask.palletsprojects.com/en/2.2.x/> para saber mais sobre como esse exemplo funciona.

Você pode visualizar os pacotes que instalou com o comando **pip3 freeze**:

```
# http://tinyurl.com/zxgcqeh
```

```
Pip3 freeze
```

```
>> Flask==0.11.11
```

```
...
```

Para concluir, você pode desinstalar um programa com **pip3 uninstall** `[package_name]`. Desinstale o Flask com o comando a seguir:

```
# http://tinyurl.com/ht8mleo
```

```
Pip3 uninstall flask
```

```
...
```

```
>> Proceed (y/n)? y ...
```

O Flask foi desinstalado, o que você pode verificar com o comando **pip3 freeze**.

## Ambientes virtuais

Você pode querer instalar seus pacotes Python em um **ambiente virtual** em vez de instalar todos os pacotes em **site-packages**. Os ambientes virtuais nos permitem manter separados os pacotes Python de diferentes projetos de programação. Você pode saber mais sobre os ambientes virtuais em <http://docs.python-guide.org/en/latest/dev/virtualenvs>.

## Vocabulário

**Ambiente virtual:** Você pode usar um ambiente virtual para manter separados os pacotes Python de diferentes projetos de programação.

**Apt-get:** Gerenciador de pacotes que vem com o Ubuntu.

**Dependências:** Os programas dos quais um programa depende para ser executado apropriadamente.

**Gerenciador de pacotes:** Programa que instala e gerencia outros programas.

**Metadados:** Dados sobre dados.

**Pacote:** Software “empacotado” para distribuição.

**Pip:** Gerenciador de pacotes para Python.

**PyPI:** Site que hospeda pacotes Python.

**Site-packages:** Pasta contida na `$PYTHONPATH`. É nessa pasta que o pip instala pacotes.

**\$PYTHONPATH:** O Python procura módulos em uma lista de pastas armazenada em uma variável de ambiente chamada `$PYTHONPATH`.

**Web framework:** Programa que ajuda a criar um site.

## **Desafio**

1. Encontre um pacote no PyPI (<https://pypi.python.org>) e baixe-o com o pip.

Solução: <http://tinyurl.com/h96qbw2>.

# CAPÍTULO 19

## Controle de versões

*“Recuso-me a fazer coisas que os computadores podem fazer.”*

### – Olin Shivers

Escrever software é uma atividade realizada em equipe. Na realização de um projeto com outra pessoa (ou com uma equipe inteira), todos os envolvidos têm de poder fazer alterações no **codebase** – as pastas e os arquivos que compõem o software – e é preciso manter essas alterações sincronizadas. Vocês poderiam enviar emails periodicamente uns para os outros com suas alterações e combinar versões diferentes por conta própria, mas seria tedioso.

Além disso, o que aconteceria se vocês dois fizessem alterações na mesma parte do projeto? Como decidiriam a alteração de quem usar? Esses são os tipos de problemas que um **sistema de controle de versões** resolve. Um sistema de controle de versões é um programa projetado para nos ajudar a colaborar facilmente em projetos com outros programadores.

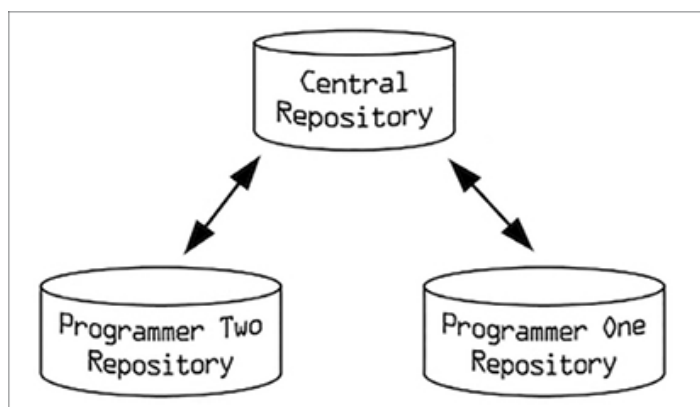
O **Git** e o **SVN** são dois sistemas de controle de versões populares. Normalmente, usamos um sistema de controle de versões com um serviço que armazena o software na nuvem. Neste capítulo, você usará o Git para inserir o software no **GitHub**, um site que armazena o código na nuvem.

## Repositórios

Um **repositório** é uma estrutura de dados criada por um sistema de controle de versões, como o Git, que registra todas as alterações feitas no projeto de programação. Uma **estrutura de dados** é uma maneira de organizar e armazenar informações: as listas e os dicionários são exemplos de estruturas de dados (você aprenderá mais sobre as estruturas de dados na Parte IV). Quando vemos um repositório, sua aparência é a de um diretório com arquivos. Você usará o Git para interagir com a estrutura de dados que registrará as alterações feitas no projeto.

Quando você estiver trabalhando em um projeto gerenciado pelo Git, haverá vários repositórios (geralmente um para cada pessoa que trabalha no projeto). Todas as pessoas envolvidas no projeto costumam ter um repositório em seu computador chamado **repositório local**, que registra as alterações feitas por elas. Também há um **repositório central**, hospedado em um site como o GitHub, com o qual todos os repositórios locais se comunicam para permanecer em sincronia uns com os outros (cada repositório é totalmente separado). Um programador que estiver trabalhando no projeto poderá atualizar o repositório central com as alterações que fizer em seu repositório local e atualizar seu repositório local com as alterações mais recentes que

outros programadores fizeram no repositório central. Se você estiver trabalhando em um projeto com outro programador, a configuração será semelhante a esta:

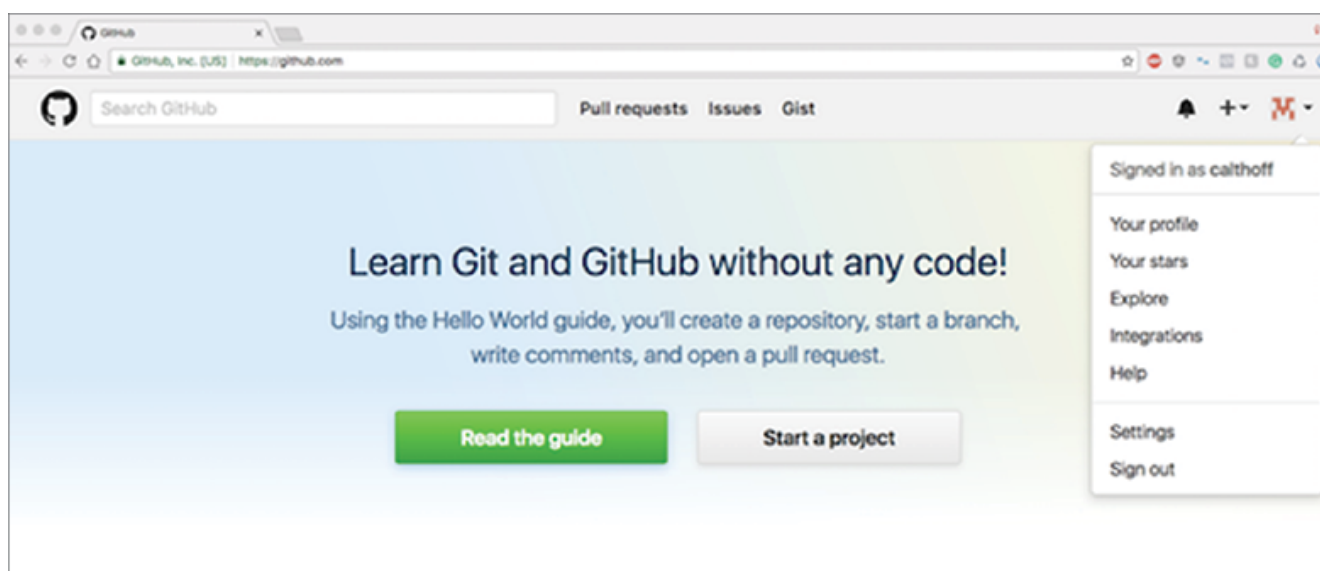


Você pode criar um novo repositório central a partir do site do GitHub (ou da linha de comando). Após ter criado um repositório central, poderá usar o Git para gerar um repositório local que se comunique com ele.

## Como começar

Se o GitHub alterar o layout do seu site, as instruções desta seção mudarão. Se isso ocorrer, fornecerei novas instruções em <http://theselftaughtprogrammer.io/git>. Para começar, você precisa criar uma conta do GitHub em <https://github.com/join>. Para criar um novo repositório no GitHub, faça login em sua conta (após criá-la) e clique no botão + no canto superior direito da tela. Clique em **New repository** no menu suspenso. Forneça o nome **hangman** para o repositório. Selecione a opção **Public** e marque a caixa **Initialize the repository with a README**. Agora clique em **Create repository**.

No GitHub, pressione o botão do canto superior direito e selecione **Your profile**.



Você verá o nome do seu repositório: **hangman**. Clique nele. Essa parte do site é seu repositório central. Você também verá um botão contendo **Clone Or Download**. Quando

clicar nele, verá um link. Salve esse link.

Antes de prosseguir é preciso instalar o Git. Você pode encontrar instruções de instalação em <https://www.git-scm.com/book/en/v2/Getting-Started-Installing-Git>.

Após instalar o Git, você poderá usá-lo a partir da linha de comando. Digite **git** na linha de comando:

```
# http://tinyurl.com/gS9d5hf
```

```
$ git
```

```
>>usage: git [--version] [--help] [-C <path>] [-c name=value] ...
```

Se sua saída for parecida com a desse exemplo, você instalou corretamente o Git.

Agora você poderá usar o link que encontrou anteriormente para baixar um repositório local em seu computador com o comando **git clone** [*url\_do\_repositório*]. O repositório será baixado em qualquer que for o diretório a partir do qual você emitir o comando. Copie o link, ou pressione o botão de cópia do link na área de transferência, e passe-o para o comando **git clone**:

```
# http://tinyurl.com/hvmq98m
```

```
$ git clone [url_do_repositório]
```

```
>> Cloning into 'hangman'... remote: Counting objects: 3, done. remote: Total 3
(delta 0), reused 0 (delta 0), pack-reused 0 Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
```

Use **ls** para verificar o repositório local baixado:

```
# http://tinyurl.com/gp4o9qv
```

```
$ ls
```

```
>> hangman
```

Você deve ver um diretório chamado **hangman**. Esse diretório é seu repositório local.

## Pushing e pulling

Existem duas ações principais que você pode executar com o Git. A primeira é atualizar seu repositório central com alterações de seu repositório local, chamado de **pushing**. A segunda é atualizar seu repositório local com novas alterações de seu repositório central, chamado **pulling**.

O comando **git remote -v** (uma flag comum que geralmente exibe informações adicionais e é a abreviação de “verboso”) exibe a URL do repositório central onde o repositório local está fazendo inserções (pushing) e extrações (pulling). Entre em seu diretório **hangman** e use o comando **git remote**:

```
# http://tinyurl.com/jscq6pj
```



```
$ cd hangman
$ git remote -v
>> origin [sua_url]/hangman.git (fetch)
>> origin [sua_url]/hangman.git (push)
```

A primeira linha da saída é a URL do repositório central do qual o projeto extrairá dados, já a segunda é a URL do repositório central no qual o projeto inserirá dados. Normalmente, fazemos inserções e extrações no mesmo repositório central para que as URLs sejam as mesmas.

## Exemplo de pushing

Nesta seção, você fará uma alteração no repositório local **hangman** que criou e clonou em seu computador e, em seguida, inserirá essa alteração em seu repositório central hospedado no GitHub.

Mova seu arquivo Python que contém o código do desafio que você concluiu no fim da Parte I para o diretório **hangman**. Agora seu repositório local tem um arquivo que não existe no repositório central – ele está fora de sincronia com o repositório central.

Você transferirá as alterações do repositório local para o repositório central em três etapas. Primeiro, **prepare (stage)** os arquivos: informe ao Git quais são os arquivos modificados que deseja inserir no repositório central.

O comando **git status** exibe o estado atual do projeto em relação ao repositório para que você possa decidir quais arquivos deseja preparar. O comando exibe os arquivos do repositório local que diferem dos arquivos do repositório central. Se você não preparar um arquivo, ele aparecerá em vermelho. Quando você preparar um arquivo, ele aparecerá em verde. Certifique-se de estar em seu diretório **hangman** e insira o comando **git status**:

```
# http://tinyurl.com/jvcr59w
```

```
$ git status
```

```
>> On branch master Your branch is up-to-date with 'origin/master'. Untracked
files: (use "git add <file>..." to include in what will be committed)
hangman.py
```

Você deve ver o arquivo **hangman.py** em vermelho. Para preparar um arquivo, use o comando **git add [arquivo]**:

```
# http://tinyurl.com/hncnyz9
```

```
$ git add hangman.py
```

Agora use o comando **git status** para confirmar que preparou o arquivo:

```
# http://tinyurl.com/jeuug7j
```

```
$ git status
```

```
>> On branch master Your branch is up-to-date with 'origin/master'. Changes to be committed: (use "git reset HEAD <file>..." to unstage)
```

```
new file: hangman.py
```

O arquivo `hangman.py` está com a cor verde porque você o preparou.

Você pode cancelar a preparação de um arquivo sem fazer alterações no repositório central com a sintaxe `git reset [caminho_do_arquivo]`. Cancele a preparação de `hangman.py` com:

```
# http://tinyurl.com/hh6xxvw
```

```
$ git reset hangman.py.
```

Confirme o cancelamento da preparação:

```
$ git status
```

```
>> On branch master Your branch is up-to-date with 'origin/master'. Untracked files: (use "git add <file>..." to include in what will be committed)
```

```
hangman.py
```

Prepare o arquivo novamente:

```
# http://tinyurl.com/gowe7hp
```

```
$ git add hangman.py
```

```
$ git status
```

```
>>On branch master Your branch is up-to-date with 'origin/master'. Changes to be committed: (use "git reset HEAD <file>..." to unstage)
```

```
new file: hangman.py
```

Uma vez que você tiver preparado os arquivos com os quais deseja atualizar seu repositório central, estará pronto para passar para a próxima etapa, a **confirmação** dos arquivos – fornecendo um comando para o Git a fim de registrar as alterações feitas no repositório local. Você pode confirmar os arquivos com a sintaxe `git commit -m [sua_mensagem]`. Esse comando cria um **commit**: uma versão do projeto que o Git salvará. A flag `-m` indica que você adicionará uma mensagem ao commit para ajudá-lo a lembrar-se das alterações que está fazendo e da razão para tê-las feito (essa mensagem é como um comentário). Na próxima etapa, você enviará as alterações para o repositório central no GitHub, onde poderá visualizar a mensagem:

```
# http://tinyurl.com/gmn92p6
```

```
$ git commit -m "my first commit"
```

```
>> 1 file changed, 1 insertion(+) create mode 100644 hangman.py
```

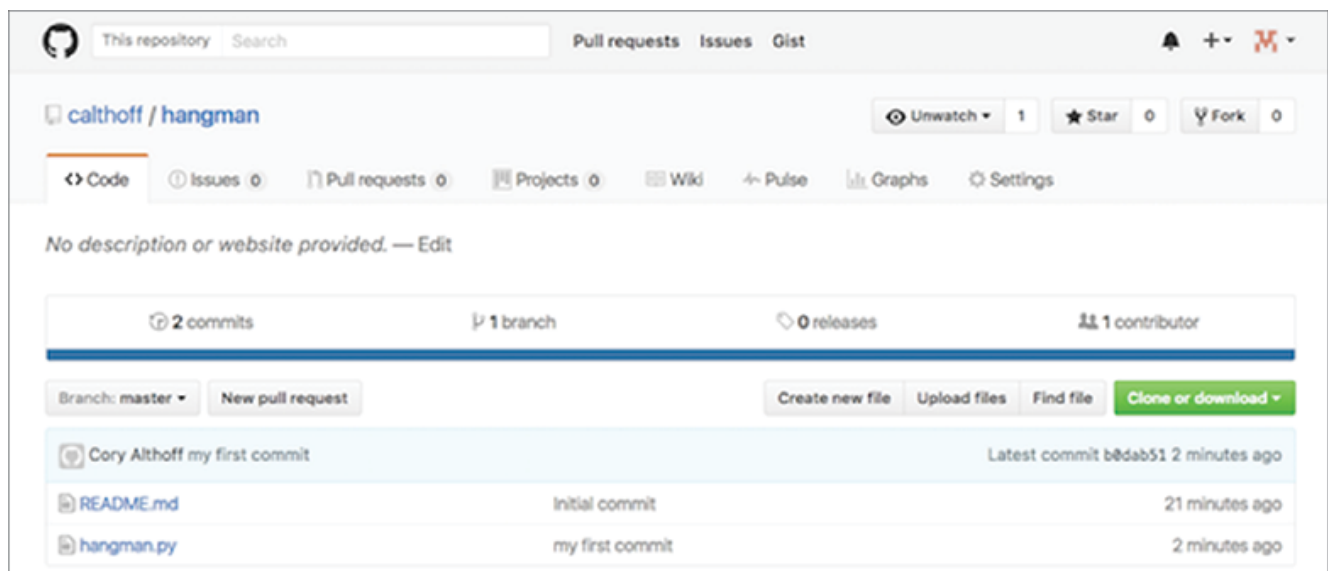
Após confirmar os arquivos, você estará pronto para a última etapa. Agora poderá enviar suas alterações para o repositório central com o comando **git push origin master**:

```
# http://tinyurl.com/hy98yq9
```

```
$ git push origin master
```

```
>> 1 file changed, 1 insertion(+) create mode 100644 hangman.py Corys-MacBook-Pro:hangman coryalthoff$ git push origin master Counting objects: 3, done. Delta compression using up to 4 threads. Compressing objects: 100% (2/2), done. Writing objects: 100% (3/3), 306 bytes | 0 bytes/s, done. Total 3 (delta 0), reused 0 (delta 0) To https://github.com/coryalthoff/hangman.git f5d44da..b0dab51 master -> master
```

Depois que você inserir o nome de usuário e a senha na linha de comando, o programa **git** enviará suas alterações para o GitHub. Se você examinar seu repositório central no site do GitHub, verá **hangman.py**, assim como a mensagem que incluiu no commit.



## Exemplo de pulling

Nesta seção, você atualizará seu repositório local extraíndo as alterações do repositório central. Será preciso fazer isso sempre que você quiser atualizar o repositório local com as alterações que outro programador tiver feito no repositório central.

Vá até seu repositório central e pressione o botão **Create new file**. Crie um arquivo chamado **new.py** e pressione o botão **Commit new file**. Esse arquivo ainda não está no repositório local, logo, este está fora de sincronia com o repositório central. Você pode atualizar o repositório local com as alterações do repositório central usando o comando **git pull origin master**:

```
# http://tinyurl.com/gqf2xue
```

```
$ git pull origin master
```

```
>>remote: Counting objects: 3, done. remote: Compressing objects: 100% (2/2),  
done. remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 Unpacking  
objects: 100% (3/3), done. From https://github.com/coryalthoff/hangman  
b0dab51..8e032f5 master -> origin/master Updating b0dab51..8e032f5 Fast-forward  
new.py | 1 + 1 file changed, 1 insertion(+) create mode 100644 new.py
```

O Git aplicou as alterações do repositório central ao repositório local. O arquivo **new.py** que você criou no repositório central agora deve estar no repositório local. Confirme com **ls**:

```
$ ls
```

```
>> README.md hangman.py new.py
```

## Reversão de versões

O Git salvará seu projeto sempre que você confirmar um arquivo. Com o Git, você pode fazer a reversão para qualquer commit anterior – pode fazer o projeto “retroceder”. Por exemplo, você pode voltar seu projeto para um commit que fez na semana passada. Todos os arquivos e pastas serão os mesmos da semana anterior. Em seguida, pode saltar imediatamente para frente para um commit mais recente. Cada commit tem um **número do commit**: uma sequência de caracteres única que o Git usa para identificar um commit.

Você pode visualizar o histórico de commits do seu projeto com o comando **git log**, que exibirá todos os seus commits:

```
# http://tinyurl.com/h2m7ahs
```

```
$ git log
```

```
>> commit 8e032f54d383e5b7fc640a3686067ca14fa8b43f Author: Cory Althoff  
<coryedwardalthoff@gmail.com> Date: Thu Dec 8 16:20:03 2016 -0800  
Create new.py  
commit b0dab51849965144d78de21002464dc0f9297fdc Author: Cory Althoff  
<coryalthoff@Corys-MacBook-Pro.local> Date: Thu Dec 8 16:12:10 2016 -0800  
my first commit  
commit f5d44dab1418191f6c2bbfd4a2b2fcf74ef5a68f Author: Cory Althoff  
<coryedwardalthoff@gmail.com> Date: Thu Dec 8 15:53:25 2016 -0800 Initial commit
```

Você deve ver três commits. Seu primeiro commit ocorreu quando você criou o repositório central. O segundo commit ocorreu quando você atualizou o repositório central com o arquivo **hangman.py**. O terceiro commit ocorreu quando você criou o arquivo **new.py**. Cada commit tem um número. Você pode passar seu projeto para outro commit fornecendo o número do commit ao comando **git checkout**. Nesse

exemplo, eu poderia reverter meu projeto para como ele era quando o criei com o comando `git checkout f5d44dab1418191f6c2bbfd4a2b2fcf74ef5a68f`.

## diff

O comando `git diff` exibe a diferença entre um arquivo no repositório local versus sua versão no repositório central. Crie um novo arquivo chamado `hello_world.py` em seu repositório local e adicione a ele o código `print("Hello, World!")`.

Agora prepare o arquivo:

```
# http://tinyurl.com/h6msygd
```

```
$ git add hello_world.py
```

Verifique se está tudo certo:

```
# http://tinyurl.com/zg4d8vd
```

```
$ git status
```

```
>> Changes to be committed: (use "git reset HEAD <file>..." to unstage) new file:
hello_world.py
```

E confirme-o:

```
# http://tinyurl.com/ztc8zs
```

```
$ git commit -m "adding new file"
```

```
>> 1 file changed, 1 insertion(+) create mode 100644 hello_world.py
```

Envie suas alterações para o repositório central:

```
# http://tinyurl.com/zay2vct
```

```
$ git push origin master
```

```
>> Counting objects: 3, done. Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done. Writing objects: 100% (3/3), 383 bytes | 0
bytes/s, done. Total 3 (delta 0), reused 0 (delta 0) To
https://github.com/coryalthoff/hangman.git 8e032f5..6f679b1 master -> master
```

Adicione `print("Hello!")` à segunda linha do arquivo `hello_world.py` no repositório local. Agora esse arquivo está diferente do arquivo do repositório central. Insira o comando `git diff` para ver a diferença:

```
# http://tinyurl.com/znvj9r8
```

```
$ git diff hello_world.py
```

```
>> diff --git a/hello_world.py b/hello_world.py index b376c99..83f9007 100644 ---
a/hello_world.py +++ b/hello_world.py -1 +1,2 print("Print, Hello World!")
+print("Hello!")
```

O Git realçará `print("Hello!")` em verde porque trata-se de uma nova linha de código. O operador de adição (+) indica que essa linha é nova. Se você tivesse removido algum código, o código excluído estaria em vermelho e seria antecedido de um operador de subtração (-).

## Próximas etapas

Neste capítulo, abordei os recursos do Git que você usará com mais frequência. Uma vez que você tiver dominado os aspectos básicos, recomendo que passe algum tempo aprendendo os recursos mais avançados do Git como a ramificação e a mesclagem em <http://theselftaughtprogrammer.io/git>.

## Vocabulário

**Codebase:** Pastas e arquivos que compõem um software.

**Commit:** Uma versão do seu projeto que o Git salvará.

**Confirmar:** Fornecer um comando que solicite ao Git para registrar as alterações que você fez no seu repositório.

**Estrutura de dados:** Uma maneira de organizar e armazenar informações. As listas e os dicionários são exemplos de estruturas de dados.

**Git:** Um popular sistema de controle de versões.

**GitHub:** Site que armazena códigos na nuvem.

**Número do commit:** Uma sequência de caracteres única que o Git usa para identificar um commit.

**Preparar:** Informar ao Git que arquivos (com modificações) você deseja enviar para o repositório central.

**Pulling:** Atualizar o repositório local com as alterações feitas no repositório central.

**Pushing:** Atualizar o repositório central com as alterações feitas no repositório local.

**Repositório:** Estrutura de dados criada por um sistema de controle de versões, como o Git, que registra as alterações feitas no projeto de programação.

**Repositório central:** Repositório hospedado em um site como o GitHub com o qual todos os repositórios locais se comunicam para ficar em sincronia uns com os outros.

**Repositório local:** O repositório que fica no seu computador.

**Sistema de controle de versões:** Programa projetado para permitir uma fácil colaboração em projetos com outros programadores.

**SVN:** Um popular sistema de controle de versões.

## **Desafios**

1. Crie um novo repositório no GitHub. Insira todos os arquivos Python dos exercícios que você fez até agora em um diretório em seu computador e envie-os para o novo repositório.

# CAPÍTULO 20

## Juntando tudo

*“A magia do mito e da lenda tornou-se real em nossa época. Alguém digita o encanto correto em um teclado e a exibição em uma tela ganha vida, mostrando coisas que nunca existiram e nem poderiam existir...”*

### – Frederick Brooks

Neste capítulo, você verá como a programação é poderosa criando um **web scraper**: um programa que extrai dados de um site. Depois que criar um web scraper, você poderá coletar dados no maior conjunto de informações que existe. O poder dos web scrapers, e a facilidade de criá-los, foi uma das razões para eu ter me tornado um entusiasta da programação, e espero que produza o mesmo efeito em você.

## HTML

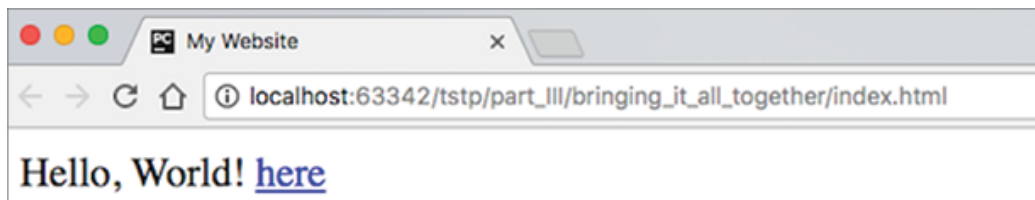
Antes de você criar um web scraper, precisará de um curso rápido de **HTML** (Hypertext Markup Language). O HTML é uma das tecnologias fundamentais com as quais os programadores criam sites com o CSS e o JavaScript. É uma linguagem que dá estrutura a um site, e é composto das tags que um navegador web usará para criar o layout das páginas web. Você pode desenvolver um site inteiro com HTML, mas ele não será interativo ou terá uma aparência muito boa, porque é o JavaScript que torna os sites interativos e é o CSS que lhes dá estilo, mas mesmo assim será um site. Aqui está um site que exibe o texto **Hello, World!**

```
# http://tinyurl.com/jptzkvp
```

```
<!--Este é um comentário em HTML.
Salve esse arquivo como index.html--> <!-- http://tinyurl.com/h3bjuov -->
<html lang="en"> <head>
<meta charset="UTF-8"> <title>My Website</title> </head>
<body>
Hello, World!
<a href="https://www.google.com"/> click here</a>
</body>
</html>
```

Salve esse HTML em um arquivo. Abra o arquivo com seu navegador web clicando nele (você pode ter de clicar com o botão direito do mouse e mudar o programa padrão para abrir o arquivo com um navegador web como o Chrome). Após abrir o arquivo com o navegador web, você verá um site exibindo **Hello World!** com um link para o Google.





Seu navegador web usará as diferentes **tags HTML** do arquivo para exibir esse site. Uma tag HTML (apenas tag na abreviação) é como uma palavra-chave na programação – ela solicita ao navegador que faça algo. A maioria das tags tem uma tag inicial e uma tag de fechamento, com frequência com texto entre elas. Por exemplo, seu navegador exibirá o texto existente entre as tags `<title>` `</title>` em sua aba. Podemos ter tags dentro de tags; tudo que existe entre `<head>``</head>` são metadados sobre a página web, enquanto o texto entre `<body>``</body>` é que compõe realmente o site. Juntas, as tags `<a>``</a>` criam um link, e essas tags podem conter dados. O texto `href="https://www.google.com"` dentro da tag `<a>` permite que o navegador saiba com que site ele deve se conectar. Existem muitos outros detalhes relacionados ao HTML, mas com essas informações você está pronto para criar seu primeiro web scraper.

## Scraping no Google Notícias

Nesta seção, você criará um web scraper que buscará todos os artigos do Google Notícias extraíndo todas as tags `<item>``</item>` do seu feed RSS. O Google Notícias usa essas tags para armazenar itens de diferentes fontes de notícias, e cada item de notícia contém (além de alguns dados adicionais) um título e o link de uma URL que conduz à fonte de notícias original. Você usará o módulo **BeautifulSoup** para fazer o **parsing** do XML (um parente próximo do HTML) do Google Notícias. Fazer o parsing significa pegar um formato, como o XML ou o HTML, e usar uma linguagem de programação para lhe dar estrutura. Por exemplo, transformar os dados em um objeto. Para começar, use o comando a seguir para instalar o módulo **BeautifulSoup** no Ubuntu e no Unix: # <http://tinyurl.com/z4fzfzf>

```
$ sudo pip3 install beautifulsoup4==4.4.1
```

```
>> Successfully installed beautifulsoup4-4.4.1
```

E no Windows (abra a linha de comando como administrador): # <http://tinyurl.com/hk3kxgr>

```
$ pip3 install beautifulsoup4==4.4.1
```

```
>> Successfully installed beautifulsoup4-4.4.1
```

O Python tem um módulo interno chamado **urllib** para o trabalho com URLs. Adicione o código a seguir a um novo arquivo Python: # <http://tinyurl.com/jmgysar8>

```
import urllib.request
from bs4 import BeautifulSoup
class Scraper:
    def __init__(self,
site):
self.site = site
```

```
def scrape(self):
pass
```

O método `__init__` recebe como parâmetro o site a partir do qual será feito o scraping. A classe **Scraper** tem um método chamado **scrape** que você chamará sempre que quiser fazer o scraping de dados no site que foi passado.

Adicione o código a seguir ao método **scrape**: # <http://tinyurl.com/h5eywoa>

```
def scrape(self):
r = urllib.request\
.urlopen(self.site)
xml = r.read()
```

A função `urlopen()` faz uma solicitação a um site e retorna um objeto **Response** que contém seu XML armazenado com dados adicionais. A função `response.read()` retorna o XML do objeto **Response**. Todo o XML do site está na variável `xml`.

Agora você está pronto para fazer o parsing do HTML. Adicione uma nova linha à função **scrape** que crie um objeto **BeautifulSoup**, e passe a variável `xml` e a string `"html.parser"` (porque o parsing do XML do Google Notícias pode ser feito como HTML) como parâmetros: # <http://tinyurl.com/hvjulxh>

```
def scrape(self):
r = urllib.request\
.urlopen(self.site)
xml = r.read()
parser = "html.parser"
sp = BeautifulSoup(xml,
parser)
```

O objeto **BeautifulSoup** se encarregará da parte mais difícil e fará o parsing do XML. Adicione então à função **scrape** o código que chama o método `find_all` no objeto **BeautifulSoup**. Passe `"item"` como parâmetro (isso instruirá a função a procurar tags `<item></item>`) e o método retornará todos os itens de notícias do HTML ou XML que você baixou: # <http://tinyurl.com/zwrxjkk>

```
def scrape(self):
r = urllib.request\
.urlopen(self.site)
xml = r.read()
parser = "html.parser"
sp = BeautifulSoup(xml,
parser)
for item in sp.find_all("item"): title = item.find("title") if title is None:
continue
else:
print("\n" + title.text) O método find_all retorna um iterável contendo os
objetos item que encontrou. A cada iteração do loop for, a variável item recebe o
valor de um novo objeto Tag. Cada item é composto de tags aninhadas contendo
várias informações, mas você só quer o valor da tag title, que contém o título do
item de notícias como texto. Você pode obtê-lo chamando o método find e passando
"title" como parâmetro. Para concluir, verifique se a variável title contém dados
e se não é None, e, se contiver, exiba-a. Aqui está o web scraper completo: #
http://tinyurl.com/j55s7hm
```

```
import urllib.request
from bs4 import BeautifulSoup
class Scraper:
def __init__(self, site):
self.site = site
```

```
def scrape(self):
r = urllib.request\
.urlopen(self.site)
xml = r.read()
parser = "html.parser"
sp = BeautifulSoup(xml,
parser)
for item in sp.find_all("item"): title = item.find("title") if title is None:
continue
else:
print("\n" + title.text)
news = "https://news.google.com/news/rss/headlines"
Scraper(news).scrape()
```

Quando você executar seu programa, a saída deve ficar parecida com esta: • SpaceX Crew Dragon to Bring 2 NASA Astronauts Home – The New York Times • Mars rover launched to look for signs of past life – DAWN.com • How Woody Vines Do the Twist – The New York Times • Rite Aid Used Facial Recognition in Stores for Nearly a Decade – WIRED

Agora que você pode coletar manchetes no Google Notícias, as possibilidades são infinitas. Você poderia escrever um programa para analisar as palavras mais usadas

nas manchetes. Poderia criar um programa para analisar a impressão causada pelas manchetes e ver se há alguma relação com o mercado de ações. Com o web scraping todas as informações que existem estão ao seu alcance, e espero que isso o empolgue tanto quanto me empolga.

## Vocabulário

**HTML:** Linguagem que dá estrutura a um site.

**Parse:** Parsing significa pegar um formato como o HTML e usar uma linguagem de programação para lhe dar estrutura. Por exemplo, transformar os dados em um objeto.

**Tag HTML:** É como uma palavra-chave na programação – solicita ao navegador que faça algo.

**Web scraper:** Programa que extrai dados de um site.

## Desafio

1. Modifique seu scraper para que ele salve as manchetes em um arquivo.

Solução do desafio: <http://tinyurl.com/gkv6fuh>.

## PARTE IV

# Introdução à ciência da computação

# CAPÍTULO 21

## Estruturas de dados

*“Acredito que, na verdade, a diferença entre um bom e um mau programador é se ele considera mais importante seu código ou suas estruturas de dados. Maus programadores se preocupam com o código. Bons programadores se preocupam com as estruturas de dados e seus relacionamentos.”*

### – Linus Torvalds

## Estruturas de dados

Uma **estrutura de dados** é um formato usado para armazenar e organizar informações. As estruturas de dados são fundamentais para a programação e a maioria das linguagens já vem com elas internamente. Você já sabe como usar várias das estruturas de dados internas do Python, como as listas, as tuplas e os dicionários. Neste capítulo, você aprenderá a criar mais duas estruturas de dados: as pilhas e as filas.

## Pilhas

Uma **pilha (stack)** é uma estrutura de dados. Como em uma lista, você pode adicionar e remover itens em uma pilha, mas ao contrário do que ocorre na lista, só pode adicionar e remover o último item. Se você tivesse a lista `[1, 2, 3]`, poderia remover qualquer item. Se tivesse uma pilha com os mesmos itens, só poderia remover o último item, `3`. Se você remover o `3`, a pilha ficará com a aparência `[1, 2]`. Agora você pode remover o `2`. Após você remover o `2`, poderá remover o `1` e a pilha ficará vazia. A remoção de um item de uma pilha é chamada de **popping**. Se você inserir o `1` novamente na pilha, ela ficará com a aparência `[1]`. Se inserir o dois na pilha, sua aparência mudará para `[1, 2]`. A inserção de um item em uma pilha é chamada de **pushing**. Esse tipo de estrutura de dados, na qual o último item inserido é o primeiro a ser removido, chama-se **estrutura de dados LIFO (last-in-first-out, último a entrar, primeiro a sair)**.

Poderíamos considerar a abordagem LIFO como uma pilha de pratos. Se você empilhasse cinco pratos um em cima do outro, precisaria remover todos os outros pratos para chegar ao do fim da pilha. Pense em cada dado de uma pilha como um prato; para acessá-los você precisaria extrair os dados do topo.

Nesta seção, você criará uma pilha. O Python tem uma biblioteca com as duas estruturas de dados que abordarei neste capítulo, mas se você mesmo as criar, verá como elas funcionam. A pilha terá cinco métodos: **is\_empty**, **push**, **pop** e **size**. O método **is\_empty** retorna **True** se a pilha estiver vazia; caso contrário retorna **False**.

**push** adiciona um item ao topo da pilha. **pop** remove e retorna o item do topo da pilha. **peek** retorna o item do topo da pilha, mas não o remove. **size** retorna um inteiro que representa o número de itens da pilha. Aqui está uma pilha implementada em Python: # <http://tinyurl.com/zk24ps6>

```
class Stack:
def __init__(self):
self.items = []
```

```
def is_empty(self):
return self.items == []
```

```
def push(self, item):
self.items.append(item)
```

```
def pop(self):
return self.items.pop()
```

```
def peek(self):
last = len(self.items)-1
return self.items[last]
```

```
def size(self):
return len(self.items)
```

Se você criar uma nova pilha, ela estará vazia e o método **is\_empty** retornará **True**: # <http://tinyurl.com/jfybm4v>

```
stack = Stack()
print(stack.is_empty())
>> True Quando você adicionar um novo item à pilha, is_empty retornará False: #
http://tinyurl.com/zsexcal
```

```
stack = Stack()
stack.push(1)
print(stack.is_empty())
>> False Chame o método pop para remover um item da pilha e is_empty retornará
True novamente: # http://tinyurl.com/j72kswr
```

```
stack = Stack()
stack.push(1)
item = stack.pop()
```

```
print(item)
print(stack.is_empty())
```

```
>> 1
```

```
>> True Para concluir, você pode examinar o conteúdo de uma pilha e obter seu tamanho: # http://tinyurl.com/zle7sno
```

```
stack = Stack()
```

```
for i in range(0, 6):
    stack.push(i)
```

```
print(stack.peek())
print(stack.size())
```

```
>> 5
```

```
>> 6
```

## Inversão de uma string com uma pilha

Uma pilha pode inverter um iterável, porque o que quer que você insira na pilha será removido na ordem inversa. Nesta seção, você resolverá um problema comum em entrevistas de programação – a inversão de uma string com o uso de uma pilha, por meio da sua inserção e depois com a remoção: # <http://tinyurl.com/zoosvqg>

```
class Stack:
    def __init__(self):
        self.items = []
```

```
    def is_empty(self):
        return self.items == []
```

```
    def push(self, item):
        self.items.append(item)
```

```
    def pop(self):
        return self.items.pop()
```

```
    def peek(self):
        last = len(self.items)-1
        return self.items[last]
```

```
    def size(self):
```



```
return len(self.items)
```

```
stack = Stack()  
for c in "Hello":  
    stack.push(c)
```

```
reverse = ""
```

```
for i in range(len(stack.items)): reverse += stack.pop()
```

```
print(reverse)
```

```
>> olleH
```

Primeiro, você percorreu cada caractere da string **"Hello"** e o inseriu em uma pilha. Em seguida, iterou pela pilha. Você removeu cada item da pilha e o inseriu na variável **reverse**. Após a iteração terminar, a palavra original está invertida e o programa está exibindo **olleH**.

## Filas

A **fila (queue)** é outra estrutura de dados. Uma fila também é como uma lista, onde você pode adicionar e remover itens. Ela também é como uma pilha porque só podemos adicionar e remover itens em uma ordem específica. Ao contrário da pilha, na qual o primeiro item inserido é o último a sair, a fila é uma **estrutura de dados FIFO (first-in-first-out, primeiro a entrar, primeiro a sair)**: o primeiro item adicionado é o primeiro a ser removido.

Pense em uma estrutura de dados FIFO como uma fila de pessoas esperando para comprar ingressos para o cinema. A primeira pessoa da fila será a primeira a conseguir os ingressos, a segunda pessoa da fila será a segunda a comprar, e assim por diante.

Nesta seção, você criará uma fila com quatro métodos: **enqueue**, **dequeue**, **is\_empty** e **size**. **enqueue** adiciona um novo item à fila; **dequeue** remove um item da fila; **is\_empty** retorna **True** se a fila estiver vazia, caso contrário retorna **False**; e **size** retorna o número de itens da fila: # <http://tinyurl.com/zrg24hj>.

```
class Queue:  
    def __init__(self):  
        self.items = []
```

```
    def is_empty(self):
```

```
return self.items == []
```

```
def enqueue(self, item):  
self.items.insert(0, item)
```

```
def dequeue(self):  
return self.items.pop()
```

```
def size(self):  
return len(self.items)
```

Se você criar uma nova fila, vazia, o método `is_empty` retornará `True`: # <http://tinyurl.com/j3ck9jl>

```
a_queue = Queue()  
print(a_queue.is_empty())  
>> True Adicione itens e verifique o tamanho da fila:  
# http://tinyurl.com/jzjrg8s
```

```
a_queue = Queue()
```

```
for i in range(5):  
a_queue.enqueue(i)
```

```
print(a_queue.size())  
>> 5
```

Remova cada item da fila:

```
# http://tinyurl.com/jazkh8b
```

```
a_queue = Queue()
```

```
for i in range(5):  
a_queue.enqueue(i)
```

```
for i in range(5):  
print(a_queue.dequeue())
```

```
print()
```

```
print(a_queue.size())
```

```
>> 0
>> 1
>> 2
>> 3
>> 4
>> >> 0
```

## Fila para ingressos

Uma fila pode simular pessoas esperando para comprar ingressos para ir ao cinema:

# <http://tinyurl.com/jnw56zx>

```
import time
import random
```

```
class Queue:
def __init__(self):
self.items = []
```

```
def is_empty(self):
return self.items == []
```

```
def enqueue(self, item):
self.items.insert(0, item)
```

```
def dequeue(self):
return self.items.pop()
```

```
def size(self):
return len(self.items)
```

```
def simulate_line(self,
till_show,
max_time):
pq = Queue()
tix_sold = []
```

```
for i in range(100):
pq.enqueue("person"
+ str(i))
```

```
t_end = time.time()\
```

```

+ till_show
now = time.time()
while now < t_end \
and not pq.is_empty():
now = time.time()
r = random.\
randint(0,
max_time)
time.sleep(r)
person = pq.dequeue()
print(person)
tix_sold.append(person)

```

```

return tix_sold

```

```

queue = Queue()
sold = queue.simulate_line(5, 1) print(sold)

```

```

>> person0

```

```

...

```

```

>> ['person0', 'person1', 'person2']

```

Primeiro, você criou uma função chamada **simulate\_line**, que simula a venda de ingressos para as pessoas de uma fila. A função recebe dois parâmetros: **till\_show** e **max\_time**. O primeiro parâmetro é um inteiro que representa quantos segundos faltam até a sessão começar e não haver mais tempo para a compra de ingressos. O segundo parâmetro também é um inteiro, que representa o período de tempo mais longo (em segundos) que uma pessoa pode ter de esperar para comprar um ingresso.

Na função, você criou uma fila nova e vazia e uma lista vazia. A lista registrará as pessoas que compraram um ingresso. Em seguida, você preencheu a fila com 100 strings, começando com "**person0**" e terminando com "**person99**". Cada string da fila representa uma pessoa que está na fila esperando para comprar um ingresso.

O módulo interno **time** tem uma função chamada **time**. Ela retorna um float que representa quantos segundos se passaram desde o valor de **epoch**, um momento no tempo (1º de janeiro de 1970) usado como referência. Se eu chamar a função **time** agora, ela retornará **1481849664.256039**, o período em segundos que se passou desde o valor de epoch. Se após um segundo eu a chamar novamente, o float que a função retornará será incrementado em 1 unidade.

A variável **t\_end** calcula o resultado da função **time** mais o período em segundos passado na variável **till\_show**. A combinação dos dois cria um momento no futuro.

O loop **while** é executado até a função **time** retornar um resultado maior que **t\_end** ou até a fila estar vazia.

Em seguida, você interrompeu o código Python por um período de tempo aleatório para estimular que cada venda de ingresso se estenda por um período de tempo diferente. Você fez isso chamando a função **sleep** no módulo interno **time** para que o código Python não faça nada durante um número de segundos aleatório entre **0** e **max\_time**.

Depois da pausa causada pela função **sleep**, você removeu da fila uma string que representa uma pessoa e a inseriu na lista **tix\_sold**, que representa que a pessoa comprou um ingresso.

O resultado do seu código é uma função que pode vender ingressos para as pessoas de uma fila, vendendo mais ou menos ingressos dependendo dos parâmetros passados e da casualidade.

## Vocabulário

**Estrutura de dados:** Formato usado para armazenar e organizar informações.

**Estrutura de dados primeiro a entrar, primeiro a sair (first-in-first-out):** Estrutura de dados na qual o primeiro item adicionado é o primeiro a ser removido.

**Estrutura de dados último a entrar, primeiro a sair (last-in-first-out):** Estrutura de dados na qual o último item inserido é o primeiro a ser removido.

**Epoch:** Momento no tempo usado como referência.

**Pilha:** Estrutura de dados último a entrar, primeiro a sair.

**Popping:** Remover um item de uma pilha.

**Pushing:** Inserir um item em uma pilha.

**FIFO (first-in-first-out):** Primeiro a entrar, primeiro a sair.

**Fila:** Estrutura de dados primeiro a entrar, primeiro a sair.

**LIFO (last-in-first-out):** Último a entrar, primeiro a sair.

## Desafios

1. Inverta a string "**yesterday**" usando uma pilha.
2. Use uma pilha para criar uma nova lista com os itens da lista a seguir invertidos: **[1, 2, 3, 4, 5]**.

Soluções: <http://tinyurl.com/j7d7nx2>.

# CAPÍTULO 22

## Algoritmos

*“Um algoritmo é como uma receita.”*

### – Waseem Latif

Este capítulo é uma breve introdução aos algoritmos. Um **algoritmo** é uma série de etapas que podem ser seguidas para a solução de um problema. O problema poderia ser uma busca em uma lista ou a exibição da letra da canção “99 Bottles of Beer on the Wall.”

### FizzBuzz

Finalmente chegou a hora de aprendermos a resolver o FizzBuzz, a popular questão solicitada em entrevistas, projetada para eliminar candidatos: Escreva um programa que exiba os números de 1 a 100, mas para múltiplos de três, em vez do número exiba “Fizz”, e para múltiplos de cinco exiba “Buzz”. Para múltiplos tanto de três quanto de cinco exiba “FizzBuzz”.

Para resolver esse problema, você precisa de uma maneira de verificar se um número é múltiplo de três, múltiplo de cinco, as duas coisas, ou nenhuma delas. Se um número for múltiplo de três, quando você dividi-lo por três não haverá resto. O mesmo se aplica ao número cinco. O operador de módulo (%) retorna o resto. Você pode resolver esse problema iterando pelos números e verificando se cada número é divisível por três e cinco, apenas por três, apenas por cinco ou por nenhum dos dois:

# <http://tinyurl.com/jroprmn>

```
def fizz_buzz():
    for i in range(1, 101):
        if i % 3 == 0 \
            and i % 5 == 0:
            print("FizzBuzz")
        elif i % 3 == 0:
            print("Fizz")
        elif i % 5 == 0:
            print("Buzz")
        else:
            print(i)
```

```
fizz_buzz()
```

```
>> 1
```

```
>> 2
```

```
>> Fizz ...
```

Você começou iterando pelos números de 1 a 100. Em seguida, verificou se o número é divisível por 3 e 5. É importante fazer isso primeiro, porque se o número for divisível por ambos, você terá de exibir **FizzBuzz** e prosseguir para a próxima iteração do loop. Se você verificasse primeiro se um número é divisível apenas por 3 ou apenas por 5, e esse número fosse encontrado, não poderia exibir **Fizz** ou **Buzz** e prosseguir para a próxima iteração do loop, porque o número também poderia ser divisível tanto por 3 quanto por 5, caso em que seria incorreto exibir **Fizz** ou **Buzz**; seria preciso exibir **FizzBuzz**.

Uma vez que tiver sido verificado se um número é divisível por 3 e 5, a ordem desses testes não importará mais, porque você saberá que ele não é divisível por ambos. Se o número for divisível por 3 ou 5, você poderá interromper o algoritmo e exibir **Fizz** ou **Buzz**. Se um número passar pelas três primeiras condições, você saberá que ele não é divisível por 3, por 5 ou por ambos e poderá exibi-lo.

## Busca sequencial

Um **algoritmo de busca** encontra informações em uma estrutura de dados como a lista. A **busca sequencial** é um algoritmo de busca simples que verifica cada item de uma estrutura de dados para ver se ele é o item que está sendo procurado.

Se ao jogar cartas você já procurou uma carta específica no baralho, provavelmente fez uma busca sequencial para encontrá-la. Você examinou cada carta do baralho, e quando ela não era a que estava sendo procurada, passou para a próxima carta. Quando você finalmente chegou à carta desejada, interrompeu a busca. Se você percorresse o baralho inteiro sem encontrar a carta, também pararia, porque perceberia que ele não a contém. Aqui está um exemplo de uma busca sequencial em Python: # <http://tinyurl.com/zer9esp>

```
def ss(number_list, n):
    found = False
    for i in number_list:
        if i == n:
            found = True
            break
    return found
```

```
numbers = range(0, 100)
s1 = ss(numbers, 2)
print(s1)
s2 = ss(numbers, 202)
print(s2)
```

```
>> True >> False
```

Primeiro você configurou a variável `found` com `False`. Essa

variável registra se o algoritmo encontrou ou não o número que está sendo procurado. Em seguida, percorreu cada número da lista e verificou se ele era o número procurado. Se o encontrar, você configurará `found` com `True`, sairá do loop e retornará a variável `found`, que será `True`.

Se você não encontrar o número que está procurando, passará para o próximo número da lista. Se percorrer a lista inteira, retornará a variável `found`. `found` será `False` se o número não estiver na lista.

## Palíndromo

Um **palíndromo** é uma palavra que tem a mesma grafia da esquerda para a direita ou vice-versa. Você pode escrever um algoritmo que verifique se uma palavra é um palíndromo invertendo todas as letras da palavra e verificando se a palavra invertida e a palavra original são iguais. Se forem, a palavra é um palíndromo: # <http://tinyurl.com/jffr7pr>

```
def palindrome(word):  
    word = word.lower()  
    return word[::-1] == word
```

```
print(palindrome("Mother")) print(palindrome("Mom")) >> False >> True  
O método lower removerá os caracteres maiúsculos da palavra que estiver sendo verificada.  
O Python trata M e m como caracteres diferentes e queremos que eles sejam  
tratados como o mesmo caractere.
```

O código `word[::-1]` inverterá a palavra. `[::-1]` é a sintaxe do Python que retorna uma fatia de um iterável inteiro invertida. Você inverterá a palavra para poder compará-la com a original. Se elas forem iguais, a função retornará `True`, porque a palavra é um palíndromo. Caso contrário, a função retornará `False`.

## Anagrama

Um **anagrama** é uma palavra criada pela reorganização das letras de outra palavra. A palavra `iceman` é um anagrama de `cinema`, porque podemos reorganizar as letras de uma das duas palavras para formar a outra. Você pode determinar se duas palavras são anagramas ordenando as letras de cada palavra alfabeticamente e verificando se elas são iguais: # <http://tinyurl.com/hxplj3z>

```
def anagram(w1, w2):  
    w1 = w1.lower()  
    w2 = w2.lower()  
    return sorted(w1) == sorted(w2)
```



```
print(anagram("iceman", "cinema")) print(anagram("leaf", "tree")) >> True >> False
```

Primeiro você chamou o método `lower` nas duas palavras para que a diferenciação entre letras maiúsculas e minúsculas não afete o resultado. Em seguida, passou as duas palavras para o método `sorted` do Python. O método `sorted` retornará as palavras ordenadas alfabeticamente. Para concluir, você comparou os resultados. Se as palavras ordenadas forem iguais, seu algoritmo retornará `True`. Caso contrário, ele retornará `False`.

## Contagem das ocorrências de caracteres

Nesta seção, você escreverá um algoritmo que retornará quantas vezes cada caractere ocorre em uma string. O algoritmo percorrerá cada caractere da string e registrará em um dicionário quantas vezes ele ocorre: # <http://tinyurl.com/zknqlde>

```
def count_characters(string):  
    count_dict = {}  
    for c in string:  
        if c in count_dict:  
            count_dict[c] += 1  
        else:  
            count_dict[c] = 1  
    print(count_dict)
```

```
count_characters("Dynasty") >> {'D': 1, 't': 1, 'n': 1, 'a': 1, 's': 1, 'y': 2}
```

Nesse algoritmo, você percorreu cada caractere de uma string passada como o parâmetro `string`. Se o caractere já estiver no dicionário `count_dict`, você incrementará seu valor em 1 unidade.

Caso contrário, você adicionará o caractere ao dicionário e configurará seu valor com 1. No fim do loop `for`, `count_dict` conterá um par chave-valor para cada caractere da string. O valor de cada chave será o número de vezes que o caractere ocorreu na string.

## Recursão

A **recursão** é um método de solução de problemas que divide o problema em partes cada vez menores até ele poder ser resolvido facilmente. Até agora, você resolveu problemas usando **algoritmos iterativos**. Os algoritmos iterativos resolvem problemas repetindo etapas, normalmente usando um loop. Os **algoritmos recursivos** dependem de funções que chamem a si mesmas. Qualquer problema que você possa resolver iterativamente poderá ser resolvido recursivamente; no entanto, às vezes um algoritmo recursivo é uma solução mais elegante.

Um algoritmo recursivo é escrito dentro de uma função. A função deve ter um **caso**

**base:** uma condição que encerra um algoritmo recursivo para impedi-lo de continuar infinitamente. Dentro da função, a função chamará a si mesma. Sempre que a função chamar a si mesma, ela chegará mais perto do caso base. O caso base acabará sendo atendido, o problema será resolvido e a função parará de chamar a si mesma. Um algoritmo que seguir essas regras estará obedecendo às três leis da recursão: 1. Um algoritmo recursivo deve ter um caso base.

2. Um algoritmo recursivo deve alterar seu estado e mover-se em direção ao caso base.

3. Um algoritmo recursivo deve chamar a si mesmo, recursivamente.

Aqui está um algoritmo recursivo que exibe a letra da popular canção folk “99 Bottles of Beer on the Wall”: # <http://tinyurl.com/z49qe4s>

```
def bottles_of_beer(bob):  
    """ Prints 99 Bottle of Beer on the  
    Wall lyrics.  
    :param bob: Must  
    be a positive  
    integer.  
    """  
    if bob < 1:  
        print("""No more  
        bottles  
        of beer  
        on the wall.  
        No more  
        bottles of  
        beer.""")  
        return  
    tmp = bob  
    bob -= 1  
    print("{} bottles of beer on the  
    wall. {} bottles  
    of beer. Take one  
    down, pass it  
    around, {} bottles  
    of beer on the  
    wall.  
    """.format(tmp,  
    tmp,  
    bob))  
    bottles_of_beer(bob)
```

```
bottles_of_beer(99)
```

>> 99 bottles of beer on the wall. 99 bottles of beer.

Take one down, pass it around, 98 bottles of beer on the wall. 98 bottles of beer on the wall. 98 bottles of beer.

Take one down, pass it around, 97 bottles of beer on the wall.

...

No more bottles of beer on the wall. No more bottles of beer.

Nesse exemplo, a primeira lei da recursão foi atendida com o caso base a seguir: # <http://tinyurl.com/h4k3ytt>

```
if bob < 1:
    print("""No more
bottles
of beer
on the wall.
No more
bottles of
beer.""")
    return
```

Quando a variável **bob** for menor que **1**, a função retornará e parará de chamar a si mesma.

A linha **bob -= 1** atende à segunda lei da recursão porque diminuir a variável **bob** faz o algoritmo mover-se em direção ao caso base. Nesse exemplo, você passou o número **99** para sua função como parâmetro. O caso base será atendido quando a variável **bob** for menor que **1**, e sempre que a função chamar a si mesma, ela se moverá em direção ao caso base.

A última lei da recursão é atendida com:

```
# http://tinyurl.com/j7zwm8t
```

```
bottles_of_beer(bob)
```

Essa linha assegura que, enquanto o caso base não for atendido, a função continue chamando a si mesma. Sempre que a função chamar a si mesma, passará para ela própria um parâmetro que foi decrementado em **1** unidade e, portanto, avançará em direção ao caso base. Na primeira vez que a função chamar a si mesma com essa linha, passará para ela própria **98** como parâmetro, em seguida passará **97**, depois **96**, até finalmente passar um parâmetro menor que **1**, o que atenderá ao caso base e “**No more bottles of beer on the wall. No more bottles of beer.**” será exibido. A função alcançará então a palavra-chave **return**, o que interromperá o algoritmo.

A recursão é reconhecidamente um dos conceitos mais complicados para novos programadores entenderem. Se inicialmente for confuso para você, não se preocupe

– continue praticando. E lembre-se: para entender a recursão, você precisa entender a recursão.<sup>1</sup>

## Vocabulário

**Algoritmo:** Uma série de etapas que podem ser seguidas para a resolução de um problema.

**Algoritmo de busca:** Algoritmo que encontra informações em uma estrutura de dados (como uma lista).

**Algoritmo iterativo:** Os algoritmos iterativos resolvem problemas repetindo etapas, normalmente usando um loop.

**Algoritmo recursivo:** Os algoritmos recursivos resolvem problemas usando funções que chamam a si mesmas.

**Anagrama:** Palavra criada pela reorganização das letras de outra palavra.

**Busca sequencial:** Algoritmo de busca simples que encontra informações em uma estrutura de dados verificando cada item para ver se ele coincide com o item que está sendo procurado.

**Caso base:** Condição que encerra um algoritmo recursivo.

**Palíndromo:** Palavra que tem a mesma grafia da esquerda para a direita ou vice-versa.

**Recursão:** Método de solução de problemas que divide o problema em partes cada vez menores até ele poder ser resolvido facilmente.

## Desafio

1. Cadastre-se para ter uma conta na plataforma <http://leetcode.com> e tente resolver três de seus problemas de algoritmos de nível fácil.

---

<sup>1</sup> N.T.: Pode parecer uma colocação estranha do autor, mas me parece que aqui ele está usando o conceito de recursão, ou seja, usando uma ação que chama a si própria.

## PARTE V

# Conseguir um emprego

## CAPÍTULO 23

### Melhores práticas de programação

*“Codifique sempre como se a pessoa que for fazer a manutenção de seu código fosse um psicopata violento que sabe onde você mora.”*

#### – John Woods

**Código de produção** é o código que existe em um produto que as pessoas usam. Quando colocamos um software em **produção**, isso significa que estamos disponibilizando-o para todos. Neste capítulo, abordarei alguns princípios gerais de programação que o ajudarão a escrever um código pronto para a produção. Muitos desses princípios são originários de *O Programador Pragmático* de Andy Hunt e Dave Thomas, um livro que melhorou muito a qualidade do meu código.

#### Escreva código como último recurso

Sua missão como engenheiro de software é escrever a menor quantidade de código possível. Quando você tiver um problema, a primeira coisa em que deve pensar não deve ser “Como posso resolver isso?”. Deve ser “Alguém já resolveu esse problema e posso usar sua solução?”. Se você estiver tentando resolver um problema comum, alguém pode já ter descoberto a solução. Comece procurando uma solução online. Só após ter certeza de que ninguém conseguiu resolver o problema é que você deve tentar resolvê-lo por conta própria.

#### DRY

**DRY** é um princípio de programação cujas letras são as iniciais de Don't Repeat Yourself (não se repita). Não repita o mesmo, ou quase o mesmo, código em um programa. Em vez disso, insira o código em uma função que possa manipular várias situações.

#### Ortogonalidade

A **ortogonalidade** é outro princípio de programação importante popularizado pelo livro *O Programador Pragmático*. Hunt e Thomas explicam: “Em computação, esse termo significa algo como uma independência ou separação. Duas ou mais coisas são ortogonais quando alterações feitas em uma delas não afetam nenhuma das outras. Em um sistema bem projetado, o código do banco de dados deve ser ortogonal em relação à interface de usuário: temos de poder alterar a interface sem afetar o banco de dados e trocar de banco de dados sem alterar a interface”. Você pode colocar isso em prática lembrando-se sempre de que “a não deve afetar b”. Se

Se você tiver dois módulos – módulo **a** e módulo **b** – o módulo **a** não deve fazer alterações em elementos do módulo **b** e vice-versa. Se você projetar um sistema no qual **a** afete **b**, que então afete **c**, que por sua vez afete **d**, tudo sairá rapidamente de controle e o sistema passará a ser não gerenciável.

## **Cada dado deve ter apenas uma representação**

Quando você tiver um dado, só deve armazená-lo em um local. Por exemplo, digamos que você estivesse criando um software que operasse com números de telefone. Se duas funções precisarem usar uma lista de códigos de área, certifique-se de que haja apenas uma lista em seu programa. Não é recomendável que existam duas listas de códigos de área duplicadas, uma para cada função. Em vez disso, crie uma variável global para armazenar os códigos de área. Melhor ainda seria armazenar as informações em um arquivo ou em um banco de dados.

O problema da duplicação de dados é que em algum momento você precisará alterá-los e terá de se lembrar de fazê-lo em todos os locais em que os duplicou. Se você alterar seus códigos de área em uma função e esquecer da outra função que também usa os dados, o programa não funcionará apropriadamente. Você pode evitar isso tendo apenas uma representação para cada dado.

## **As funções devem executar uma única ação**

Cada função que você escrever deve executar uma única, e apenas uma, ação. Se você perceber que suas funções estão ficando muito longas, considere se elas estão executando mais de uma tarefa. Limitar as funções a executarem uma única tarefa apresenta várias vantagens. Seu código será mais fácil de ler porque o nome da função descreverá exatamente o que ela faz. Se o código não estiver funcionando, será mais fácil depurá-lo porque cada função será responsável por uma tarefa específica, logo, você poderá isolar e diagnosticar rapidamente a função que não está funcionando. Programadores famosos resumiram melhor essa questão: “Uma complexidade tão grande no software pode estar ocorrendo por você querer que uma única coisa faça duas coisas”.

## **Se estiver demorando muito, provavelmente você cometeu um erro**

Se você não estiver trabalhando em algo obviamente complexo, como trabalhar com grandes quantidades de dados, e seu programa estiver demorando muito para ser carregado, assuma que fez algo errado.

## **Faça as coisas da melhor maneira na primeira vez**

Se você estiver programando e pensar “Sei que existe uma maneira melhor de fazer isso, mas estou no meio da codificação e não quero parar para descobrir como fazer melhor”, não continue codificando. Pare. Faça melhor.

## Siga as convenções

Reservar algum tempo para aprender as convenções da nova linguagem de programação o ajudará a ler com mais rapidez os códigos escritos nela. O PEP 8 é um conjunto de diretrizes para a criação de código Python e você deve lê-lo. Ele inclui as regras para a extensão do código Python para novas linhas. Está disponível em <https://www.python.org/dev/peps/pep-0008/>.

## Use um IDE poderoso

Até agora, você usou o IDLE, o IDE que vem com o Python, para escrever seus códigos. No entanto, o IDLE é apenas um dos muitos IDEs disponíveis, e não recomendo que você o use por muito tempo porque ele não é tão poderoso. Por exemplo, se você abrir um projeto Python em um IDE melhor, haverá diferentes abas para cada arquivo Python. No IDLE, é preciso abrir uma nova janela para cada arquivo, o que é tedioso e dificulta a alternância entre arquivos.

Uso um IDE chamado PyCharm criado pela JetBrains. Eles oferecem uma versão gratuita e uma profissional. Criei uma lista dos recursos do PyCharm que me ajudam a ser mais rápido:

1. Se você quiser ver a definição de uma variável, função ou objeto, o PyCharm tem um atalho que salta para o código que o definiu (mesmo se ele estiver em um arquivo diferente). Também há um atalho que volta para a página inicial.
2. O PyCharm tem um recurso que salva um histórico local, o que aumentou muito minha produtividade. Ele salva automaticamente uma nova versão do projeto sempre que este muda. Você pode usar o PyCharm como sistema de controle de versões local sem precisar fazer a inserção em um repositório. Não é preciso fazer nada, isso acontece automaticamente. Antes de saber da existência desse recurso, com frequência eu resolvia um problema, mudava a solução e então decidia que queria voltar para a solução original. Se não enviasse a solução inicial para o GitHub, eu a perdia e era preciso reescrevê-la. Com esse recurso, você pode voltar 10 minutos no tempo e recarregar seu projeto exatamente como ele era. Se mudar de ideia novamente, poderá se alternar entre diferentes soluções quantas vezes quiser.
3. Você pode estar executando muitas operações de copiar e colar código em seu fluxo de trabalho. No PyCharm, em vez de copiar e colar, podemos mover o código para cima e para baixo na página em que estivermos.



4. O PyCharm suporta sistemas de controle de versões como o Git e o SVN. Em vez de acessar a linha de comando, você pode usar o Git a partir do PyCharm. Quanto menos você precisar se alternar entre seu IDE e a linha de comando, mais produtivo será.
5. O PyCharm tem uma linha de comando e um shell Python internos.
6. O PyCharm tem um **depurador** interno. Um depurador é um programa que nos permite interromper a execução do código e percorrer o programa linha a linha para ver os valores das variáveis em diferentes partes dele.

Se você estiver interessado em aprender a usar o PyCharm, a JetBrains tem um tutorial disponível em <https://www.jetbrains.com/help/pycharm/2016.1/quick-start-guide.html>.

## Logging

**Logging** é a prática de registrar dados quando o software é executado. Você pode usar o logging para ajudá-lo a depurar seu programa e obter insights adicionais sobre o que ocorreu quando ele foi executado. O Python vem com um módulo **logging** que permite fazer registros no console ou em um arquivo.

Se acontecer algo errado em seu programa, não deixe passar como se nada tivesse ocorrido – você deve registrar informações sobre o que ocorreu para examinar posteriormente. O logging também é útil para a coleta e a análise de dados. Por exemplo, você pode configurar um servidor web para registrar dados – incluindo a data e a hora – sempre que ele receber uma solicitação. Você poderia armazenar todos os seus logs em um banco de dados, criar outro programa para analisar os dados e gerar um gráfico para exibir as horas do dia em que as pessoas visitam seu site.

O blogger Henrik Warne escreveu: “Uma das diferenças entre um programador bom e um ruim é que o bom programador adiciona logging e ferramentas e esses recursos facilitam depurar o programa quando algo falha”. Você pode aprender a usar o módulo **logging** do Python em <https://docs.python.org/3/howto/logging.html>.

## Teste

**Testar** um programa significa verificar se ele “atende aos requisitos que orientaram seu design e desenvolvimento, responde corretamente a todos os tipos de entradas, executa suas funções em tempo aceitável, é suficientemente usável, pode ser instalado e executado nos ambientes pretendidos e atinge o resultado geral que seus stakeholders<sup>1</sup> esperam.”<sup>2</sup> Para testar seus programas, os programadores escrevem mais programas.

Em um ambiente de produção, testar não é opcional. Você deve considerar incompleto qualquer programa que queira colocar em produção até ter escrito testes para ele. No entanto, se escrever um programa que nunca mais usará, testar pode ser perda de tempo. Se você estiver escrevendo um programa que outras pessoas usarão, deve criar testes. Como vários programadores famosos disseram “Código não testado é código quebrado”. Você pode aprender a usar o módulo `unittest` do Python em <https://docs.python.org/3/library/unittest.html>.

## Revisões de código

Em uma **revisão de código**, alguém lê o código e fornece feedback. Você deve fazer o maior número de revisões de código que puder – principalmente como programador autodidata. Mesmo se você seguir todas as práticas recomendadas descritas neste capítulo, haverá coisas que fará incorretamente. É preciso que alguém com experiência leia seu código e lhe informe sobre os erros cometidos para que você possa corrigi-los.

O Code Review é um site onde você poderá obter revisões de código feitas por uma comunidade de programadores. Qualquer pessoa pode acessar o Code Review e postar seu código. Outros membros da comunidade Stack Exchange revisarão o código, darão feedback sobre onde você agiu bem e oferecerão sugestões úteis de como melhorar. Você pode visitar o Code Review em <http://codereview.stackexchange.com/>.

## Segurança

A segurança é um tópico que o programador autodidata costuma ignorar. Provavelmente não haverá perguntas sobre segurança nas entrevistas e ela não será importante para os programas que você escrever enquanto estiver aprendendo a programar. No entanto, quando você conseguir seu primeiro emprego como programador, será diretamente responsável pela segurança do código que escrever. Nesta seção, fornecerei algumas dicas para manter seu código seguro.

Anteriormente, você aprendeu a usar **sudo** para emitir um comando como usuário root. Nunca execute um programa a partir da linha de comando usando **sudo** se não precisar porque um hacker terá acesso root se comprometer o programa. Você também deve desativar logins como root se estiver gerenciando um servidor. Todo hacker sabe que existe uma conta root, logo, ela é um alvo fácil no ataque a um sistema.

Presuma sempre que a entrada do usuário é maliciosa. Vários tipos de ataques maliciosos dependem da exploração de programas que aceitam entradas do usuário, logo, você também deve presumir que todas as entradas de usuário são maliciosas e

programar levando isso em consideração.

Outra estratégia para manter seu software seguro é reduzir a **superfície de ataque** – as diferentes áreas do programa nas quais os invasores poderiam extrair dados ou atacar o sistema. Tornando a área de ataque a menor possível, você reduzirá a probabilidade de existência de vulnerabilidades no programa. Algumas estratégias para a redução da superfície de ataque: evite armazenar dados confidenciais se não precisar, dê aos usuários o menor nível de acesso que puder, use a menor quantidade de bibliotecas de terceiros possível (quanto menor for a quantidade de código, menor será a quantidade de explorações possíveis) e livre-se de recursos que não estejam mais sendo usados (menos código, menos explorações).

Evitar fazer login no sistema como usuário root, não confiar na entrada do usuário e reduzir a superfície de ataque são etapas importantes para assegurar que os programas sejam seguros. No entanto, esses são apenas pontos de partida. Você deve sempre tentar pensar como um hacker. Como um hacker exploraria seu código? Pensar assim pode ajudá-lo a encontrar vulnerabilidades que de outra forma você deixaria passar. Há tantas questões envolvendo a segurança que seria impossível abordá-las neste livro, portanto, pense sempre nisso e tente aprender cada vez mais sobre ela. Bruce Schneier tem uma ótima descrição – “A segurança é um estado de espírito”.

## Vocabulário

**Código de produção:** Código em um produto que as pessoas usam.

**DRY:** Princípio de programação que significa Don't Repeat Yourself (Não se repita).

**Depurador:** Um depurador é um programa que nos permite interromper a execução do código e percorrer o programa linha a linha para ver os valores das variáveis em diferentes partes dele.

**Logging:** A prática de registrar dados quando o software é executado.

**Ortogonalidade:** “Em computação, esse termo significa algo como uma independência ou separação. Duas ou mais coisas são ortogonais quando alterações feitas em uma delas não afetam nenhuma das outras. Em um sistema bem projetado, o código do banco de dados deve ser ortogonal em relação à interface de usuário: temos de poder alterar a interface sem afetar o banco de dados e trocar de banco de dados sem alterar a interface.”

**Produção:** Quando colocamos o software em produção, isso significa disponibilizá-lo para todos.

**Revisão de código:** Quando alguém lê o código e fornece feedback.

**Superfície de ataque:** As diferentes áreas do programa nas quais os invasores

poderiam extrair dados ou atacar o sistema.

**Testar:** Verificar se o programa “atende aos requisitos que orientaram seu design e desenvolvimento, responde corretamente a todos os tipos de entradas, executa suas funções em tempo aceitável, é suficientemente usável, pode ser instalado e executado nos ambientes pretendidos e atinge o resultado geral que seus stakeholders esperam”.<sup>3</sup>

---

<sup>1</sup> N.T.: Grupos de pessoas ou organizações que podem ter interesse nas ações do programa.

<sup>2</sup> [https://en.wikipedia.org/wiki/Software\\_testing](https://en.wikipedia.org/wiki/Software_testing)

<sup>3</sup> [https://en.wikipedia.org/wiki/Software\\_testing](https://en.wikipedia.org/wiki/Software_testing)

## CAPÍTULO 24

### Seu primeiro emprego em programação

*“Cuidado com ‘o mundo real’. Esse apelo vindo de alguém será sempre um convite para não desafiarmos as suposições que nele estão implícitas.”*

#### – Edsger W. Dijkstra

A última parte deste livro será dedicada a ajudá-lo em sua carreira. Conseguir seu primeiro emprego em programação demandará algum esforço adicional, mas se você seguir minhas recomendações, não deve ter problemas. Felizmente, uma vez que você conseguir seu primeiro emprego como programador e ganhar alguma experiência, quando chegar a hora de procurar seu próximo emprego, os recrutadores é que o procurarão.

### Escolha um caminho

Quando você se candidatar a um emprego em programação, será esperado que conheça um conjunto específico de tecnologias, dependendo da área de atuação relacionada à vaga. Embora não haja problema em ser um generalista (um programador que lide com muitas coisas) enquanto aprendemos a programar, e seja possível conseguir um emprego sendo um programador generalista, é recomendável dedicar-se a uma área da programação que você aprecie e tornar-se um especialista nela. Seguir um caminho na programação facilitará conseguir um emprego.

Os desenvolvimentos web e mobile são dois caminhos muito populares em programação. Existem duas áreas de especialização para esses caminhos: front-end e back-end. O front-end de uma aplicação é a parte que vemos – como a GUI de uma aplicação web. O back-end é o que não podemos ver – a parte que fornece dados para o front-end. As vagas abertas em programação serão oferecidas com nomes como “Programador back-end Python”, o que significa que estão procurando alguém que programe o back-end de um site e esteja familiarizado com Python. A descrição da vaga listará as tecnologias que o candidato precisa conhecer, junto com qualquer habilidade adicional necessária.

Algumas empresas têm uma equipe dedicada ao front-end e outra dedicada ao back-end. Outras só contratam desenvolvedores full stack – programadores que conseguem trabalhar tanto no front-end quanto no back-end; no entanto, isso só se aplica a empresas que criam sites ou aplicações.

Existem várias outras áreas em programação nas quais você pode trabalhar, como segurança, engenharia de plataforma e ciência de dados. As descrições de vagas em sites que listam empregos em programação são um bom local para aprender mais sobre os requisitos das diferentes áreas da programação. O Python Job Board,

encontrado em <https://www.python.org/jobs> é um bom ponto de partida. Leia os requisitos de algumas vagas, assim como as tecnologias que elas usam, para ter uma ideia do que você precisa aprender para ser competitivo para o tipo de emprego que deseja.

## **Ganhando a experiência inicial**

Antes de você ser contratado para seu primeiro emprego em programação precisará de experiência. No entanto, como ganhar experiência em programação se ninguém o contratará sem ela? Há algumas maneiras de resolver esse problema. Você pode se envolver com o movimento open source começando um projeto ou contribuindo com os milhares de projetos open source do GitHub.

Outra opção é trabalhar como freelancer. Crie um perfil em um site, como o Upwork, e comece a se candidatar para a execução de pequenos projetos de programação. Recomendo encontrar alguém que você conheça e que precise que seja realizada alguma tarefa de programação, pedir que essa pessoa se cadastre em uma conta no Upwork, e convencê-la para que o contrate oficialmente no site para poder deixar uma boa avaliação pelo trabalho. Até você ter pelo menos uma boa avaliação em um site como o Upwork, será difícil ser contratado para projetos. Quando as pessoas souberem da conclusão de pelo menos um trabalho com sucesso, será mais fácil ser contratado porque você já terá alguma credibilidade.

## **Como conseguir uma entrevista**

Uma vez que você ganhar experiência em programação por meio de trabalhos open source ou como freelancer, terá chegado a hora de começar a participar de entrevistas. Acredito que a maneira mais eficiente de conseguir uma entrevista seja usando o LinkedIn. Se não tiver uma conta no LinkedIn, crie-a para começar uma rede de relacionamentos com possíveis empregadores. Descreva resumidamente quem você é no topo do perfil e certifique-se de realçar suas habilidades em programação. Por exemplo, muitas pessoas incluem algo como “Linguagens de Programação: Python, JavaScript” no início do perfil, o que ajuda a direcionar para recrutadores que estejam procurando essas palavras-chave. Não se esqueça de inserir sua experiência com o movimento open source ou como freelancer como sua atividade mais recente.

Quando você concluir seu perfil, comece a conectar-se com recrutadores técnicos – existem muitos recrutadores técnicos no LinkedIn. Eles estão sempre procurando novos talentos e ficarão ansiosos para conversar com você. Uma vez que aceitarem seu convite, entre em contato e pergunte se estão contratando para alguma vaga aberta.

## A entrevista

Se um recrutador achar que você é um bom candidato para a vaga para a qual ele está contratando, enviará uma mensagem no LinkedIn solicitando uma entrevista por telefone. A entrevista será com o recrutador, logo, geralmente ela não é técnica, embora recrutadores já me tenham feito perguntas técnicas cujas respostas eles memorizaram durante as primeiras entrevistas. A conversa será sobre as tecnologias que você conhece e sua experiência anterior e para saber se você se enquadra na cultura da empresa.

Se você se sair bem, avançará para a segunda etapa – uma entrevista técnica por telefone – na qual falará com membros da equipe de engenharia. Eles farão as mesmas perguntas da primeira entrevista. No entanto, dessa vez as perguntas serão acompanhadas de um teste técnico pelo telefone. Os engenheiros fornecerão o endereço de um site no qual eles postaram perguntas de programação e lhe pedirão para respondê-las.

Se você passar na segunda etapa, pode haver uma terceira entrevista. Normalmente, a terceira entrevista é em pessoa nas dependências da empresa. Como nas duas primeiras entrevistas, você se reunirá com vários engenheiros da equipe. Eles farão perguntas sobre suas habilidades e experiência e passarão mais testes técnicos. Pode ocorrer de você ter de ficar para o almoço para que seja verificado como interage com a equipe. A terceira etapa é aquela em que ocorrem os famosos testes de codificação no quadro branco. Se a empresa para a qual estiver sendo feita a entrevista usar o quadro branco, você será solicitado a resolver vários problemas de programação. Recomendo comprar um quadro branco e praticar antecipadamente porque resolver um problema de programação em um quadro branco é muito mais difícil do que resolvê-lo em um computador.

## Atalhos para o sucesso na entrevista

A maioria das entrevistas de programação dá ênfase a dois assuntos – estruturas de dados e algoritmos. Para passar em sua entrevista, você sabe exatamente o que deve fazer – ficar muito bom nessas duas áreas específicas da ciência da computação. Felizmente, isso o ajudará a tornar-se um programador melhor.

Você pode reduzir ainda mais as perguntas com as quais terá de se preocupar pensando na entrevista do ponto de vista do entrevistador. Pense na situação em que seu entrevistador se encontra; dizem que um software nunca termina, e é verdade. Seu entrevistador deve ter outras coisas para fazer e não vai querer dedicar muito tempo a entrevistar candidatos. Ele vai desperdiçar seu precioso tempo criando perguntas originais sobre programação? Provavelmente não. Vai procurar “perguntas sobre programação para entrevistas” no Google e fará uma das primeiras

perguntas que encontrar. Essa situação faz com que as mesmas perguntas surjam com frequência nas entrevistas – e existem alguns recursos muito bons que ajudam a respondê-las! É altamente recomendável usar o LeetCode – nele encontrei todas as perguntas que já me fizeram em entrevistas de programação.



## CAPÍTULO 25

### Trabalho em equipe

*“Você não conseguirá ter bons softwares sem uma boa equipe, e a maioria das equipes de software se comporta como famílias disfuncionais.”*

#### – Jim McCarthy

Por vir de um passado autodidata, você estará acostumado a programar sozinho. Ao fazer parte de uma empresa, terá de aprender a trabalhar em equipe. Mesmo se você fundar uma empresa, acabará contratando outros programadores, momento em que precisará aprender a trabalhar como parte de uma equipe. Programar é uma atividade realizada em equipe, e como ocorre em qualquer equipe esportiva, é preciso ter boas relações com os colegas de equipe. Este capítulo fornecerá algumas dicas para um trabalho bem-sucedido em um ambiente de equipe.

#### Domine o básico

Quando uma empresa o contratar, seus integrantes exigirão que você seja competente nas habilidades abordadas neste livro. Não é suficiente apenas ler o livro – também é preciso dominar os conceitos. Seus colegas de equipe ficarão frustrados se tiverem de ajudá-lo constantemente com o básico.

#### Não pergunte sobre o que puder encontrar no Google

Como um membro novo e autodidata de uma equipe de programação, você terá muito a aprender e precisará fazer várias perguntas. Perguntar é uma ótima maneira de aprender, mas você precisa ter certeza de fazer as perguntas certas. Só faça uma pergunta se tiver passado pelo menos cinco minutos procurando a resposta no Google. Se fizer muitas perguntas, cujas respostas poderia ter descoberto facilmente por conta própria, acabará aborrecendo seus colegas de equipe.

#### Alteração de código

Ao ler este livro, você mostrará que é o tipo de pessoa que está constantemente tentando melhorar. Infelizmente, nem todos na equipe compartilharão de seu entusiasmo em se tornar um programador melhor. Muitos programadores não querem continuar aprendendo – sentem-se bem trabalhando sem buscar um nível ideal.

Códigos mal projetados são predominantes, principalmente em startups, onde a entrega rápida do código com frequência é mais importante do que a entrega de um código de alta qualidade. Se estiver nessa situação, tome cuidado. Alterar o código de

alguém pode ferir seu ego. E, o que é pior, se passar muito tempo corrigindo o código de outras pessoas, não terá tempo para contribuir com novos projetos e pode parecer que não está trabalhando com dedicação suficiente. A melhor maneira de evitar esse ambiente é perguntar cuidadosamente sobre a cultura de engenharia de qualquer empresa na qual estiver fazendo entrevistas. Se mesmo assim vivenciar essa situação, é melhor ouvir o que Edward Yourdon tem a dizer: “Se você achar que sua gerência não sabe o que está fazendo, ou que sua organização produz software de baixa qualidade e isso pode deixá-lo em uma situação embaraçosa, saia”.

## **Síndrome do impostor**

Qualquer pessoa que programa em algum momento se sente sobrecarregada, e mesmo se trabalhar arduamente, encontrará coisas que não conhece. Para um programador autodidata, é particularmente fácil sentir-se inadequado porque alguém pedirá que você faça algo sobre o qual nunca ouviu falar, ou você pode achar que existem muitos conceitos na ciência da computação que ainda não entende. Esses momentos acontecem com todos – e não apenas com você.

Fiquei surpreso quando um amigo com diploma de mestrado em ciência da computação da Universidade de Stanford me disse que também se sentia assim. Disse-me que todos os envolvidos em seu programa lidaram com a síndrome do impostor. Ele notou que eles regiam de uma entre duas maneiras: mostravam humildade e admitiam quando não sabiam algo – e se esforçavam para aprender – ou fingiam que sabiam tudo (mas na verdade não sabiam) e não davam continuidade à aprendizagem. Lembre-se de que você chegou onde está trabalhando arduamente, e não há problema em não saber tudo; ninguém sabe. Seja modesto, estude sem descanso qualquer coisa que não entender e ninguém o deterá.

## CAPÍTULO 26

### Leitura adicional

*“Os melhores programadores não são tão melhores do que os simplesmente bons. Eles são uma ordem de magnitude melhores, o que pode ser medido por qualquer que seja o padrão: criatividade conceitual, velocidade, genialidade no design ou habilidade para resolver problemas.”*

#### – Randall E. Stross

O artigo “ABC: Always Be Coding” de David Byttow dá bons conselhos sobre como conseguir um emprego como engenheiro de software. O título já diz tudo – Always Be Coding (codifique sempre). Você pode encontrar o artigo em <https://medium.com/always-be-coding/abc-always-be-coding-d5f8051afce2#.2hjho0px7>. Se você combinar o ABC com um novo acrônimo que criei – ABL – Always Be Learning (aprenda sempre) – pode ter certeza de que terá uma carreira fantástica. Neste capítulo, descreverei alguns recursos de programação que achei útil.

### Os clássicos

Existem alguns livros de programação que são considerados leitura obrigatória. O *Programador Pragmático*, de Andy Hunt e Dave Thomas; *Padrões de Projeto*, de Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm (os padrões de projeto são um assunto importante que não pude abordar); *Code Complete*, de Steve McConnell; *Compiladores: Princípios, Técnicas e Ferramentas*, de Alfred Aho, Jeffrey Ullman, Monica S. Lam e Ravi Sethi; e *Introduction to Algorithms*, da MIT Press. Também é altamente recomendável *Problem Solving with Data Structures and Algorithms*, uma excelente introdução gratuita e interativa aos algoritmos, de Bradley N. Miller e David L. Ranum, muito mais fácil de entender do que *Introduction to Algorithms*, do MIT.

### Aulas online

Aulas de codificação online são outra maneira de melhorar suas habilidades em programação. Você pode encontrar todas as minhas recomendações de aulas em <http://theselftaughtprogrammer.io/courses> (em inglês).

### Hacker News

Hacker News é uma plataforma de notícias enviadas por usuários hospedada no site da incubadora tecnológica Y Combinator, que fica em <https://news.ycombinator.com>. Ela o ajudará a ficar atualizado com as mais novas tendências e tecnologias.

## CAPÍTULO 27

### Próximas etapas

*“Ama o pequeno ofício que aprendeste e contenta-te com isso.”*

#### – Marcus Aurelius

Em primeiro lugar – obrigado por comprar este livro. Espero que ele o tenha ajudado a tornar-se um programador melhor. Agora que você terminou a leitura, é hora de por mãos à obra. Para onde ir depois? Estruturas de dados e algoritmos. Acesse a LeetCode e pratique com os algoritmos. Em seguida, pratique um pouco mais! Neste capítulo, fornecerei algumas considerações finais sobre como você pode continuar a melhorar como programador (após terminar de praticar a criação de algoritmos).

### Encontre um mentor

Um mentor o ajudará a levar suas habilidades em programação para o próximo nível. Uma das dificuldades de aprender a programar é que existem muitas coisas que podemos fazer abaixo do nível ideal sem o saber. Mencionei anteriormente que você pode combater isso fazendo revisões de código. Um mentor pode fazer revisões de código com você para ajudar a melhorar seu processo de codificação, recomendar livros e ensinar conceitos de programação que você não entenda.

### Tente aprofundar-se

Existe um conceito em programação chamado “caixa-preta”, que se refere a algo que você usa, mas não sabe como funciona. Quando você começar a programar, tudo será uma caixa-preta. Uma das melhores maneiras de melhorar na programação é abrir todas as caixas-pretas que você encontrar e tentar entender como elas funcionam. Um de meus amigos me disse que foi uma grande descoberta quando ele percebeu que a linha de comando também era um programa. Abrir uma caixa-preta é o que chamo de aprofundar-se.

Escrever este livro me ajudou a me aprofundar. Havia certos conceitos que eu achava que entendia, mas então percebi que não conseguia explicá-los. Precisei me aprofundar. Não pare na primeira resposta, leia todas as explicações que puder encontrar sobre um tópico. Faça perguntas e leia diferentes opiniões online.

Outra maneira de aprofundar-se é criar coisas que você queira entender melhor. Está com dificuldades para entender o controle de versões? Crie um sistema de controle de versões simples em seu tempo livre. Reservar um tempo para executar um projeto como esse vale o investimento – ele melhorará sua compreensão de

qualquer coisa com a qual você estiver tendo dificuldades.

## Outros conselhos

Uma vez encontrei um fórum no qual estavam discutindo diferentes maneiras de tornar-se um programador melhor. Surpreendentemente, a resposta mais votada foi: faça outra coisa que não seja programar. Percebi que isso era verdade – ler livros como *O Código do Talento*, de Daniel Coyle, me tornou um programador melhor porque ele descreve exatamente o que é preciso fazer para dominar qualquer habilidade. Preste atenção em coisas não relacionadas a programar que você possa trazer para sua atividade de programação.

O último conselho que darei para você é que passe o máximo de tempo que puder lendo o código de outras pessoas. É uma das melhores maneiras de se aperfeiçoar como programador. Quando você estiver estudando, tente encontrar um equilíbrio entre escrever e ler código. No começo será difícil ler o código de outras pessoas, mas é importante porque você pode aprender muito com outros programadores.

Espero que você tenha gostado de ler este livro tanto quanto gostei de escrevê-lo. Fique à vontade para conversar comigo (em inglês) pelo email [cory@theselftaughtprogrammer.io](mailto:cory@theselftaughtprogrammer.io) seja qual for o motivo. Também tenho uma newsletter de programação na qual você pode se cadastrar em <http://theselftaughtprogrammer.io> e um grupo do Facebook, que fica em <https://www.facebook.com/groups/selftaughtprogrammers>, onde é possível entrar em contato comigo e com uma comunidade de outras pessoas que estão aprendendo a programar. Se gostou deste livro, publique uma avaliação na Amazon em <https://www.amazon.com/dp/B01M01YDQA#customerReviews>; isso levará mais pessoas a poderem lê-lo, e eu aprecio cada avaliação que recebo. Muita sorte no restante de sua jornada!