

Python Fluente

dados e funções

Luciano Ramalho

Sumário

Dedication	1
Prefácio	2
Para quem é esse livro	2
Para quem esse livro não é	2
Como ler este livro	3
Abordagem "mão na massa"	5
Ponto de vista: minha perspectiva pessoal	5
Conteúdo na na Web	5
Convenções usadas no livro	6
Usando os exemplos de código	6
O'Reilly Online Learning	7
Como entrar em contato	7
Agradecimentos	8
Sobre esta tradução	10
Histórico das traduções	11
Parte I: Estruturas de dados	13
1. O modelo de dados de Python	14
1.1. Novidades nesse capítulo	15
1.2. Um baralho pythônico	15
1.3. Como os métodos especiais são utilizados	19
1.4. Visão geral dos métodos especiais	25
1.5. Por que len não é um método?	27
1.6. Resumo do capítulo	27
1.7. Para saber mais	28
2. Uma coleção de sequências	30
2.1. Novidades neste capítulo	30
2.2. Uma visão geral das sequências embutidas	31
2.3. Compreensões de listas e expressões geradoras	33
2.4. Tuplas não são apenas listas imutáveis	38
2.5. Desempacotando sequências e iteráveis	42
2.6. Pattern matching com sequências	46
2.7. Fatiamento	54
2.8. Usando + e * com sequências	57
2.9. list.sort versus a função embutida sorted	62
2.10. Quando uma lista não é a resposta	64
2.11. Resumo do capítulo	74
2.12. Leitura complementar	75
3. Dicionários e conjuntos	80

3.1. Novidades nesse capítulo.....	80
3.2. A sintaxe moderna dos dicts	81
3.3. Pattern matching com mapeamentos	83
3.4. A API padrão dos tipos de mapeamentos.....	85
3.5. Tratamento automático de chaves ausentes.....	91
3.6. Variações de dict.....	96
3.7. Mapeamentos imutáveis	100
3.8. Views de dicionários.....	101
3.9. Consequências práticas da forma como dict funciona.....	102
3.10. Teoria dos conjuntos.....	103
3.11. Consequências práticas da forma de funcionamento dos conjuntos	106
3.12. Operações de conjuntos em views de dict.....	109
3.13. Resumo do capítulo.....	111
3.14. Leitura complementar	111
4. Texto em Unicode versus Bytes	115
4.1. Novidades nesse capítulo.....	115
4.2. Questões de caracteres	116
4.3. Os fundamentos do byte.....	117
4.4. Codificadores/Decodificadores básicos.....	119
4.5. Entendendo os problemas de codificação/decodificação.....	121
4.6. Processando arquivos de texto.....	127
4.7. Normalizando o Unicode para comparações confiáveis	135
4.8. Ordenando texto Unicode	143
4.9. O banco de dados do Unicode.....	145
4.10. APIs de modo dual para str e bytes	149
4.11. Resumo do capítulo.....	151
4.12. Leitura complementar	152
5. Fábricas de classes de dados.....	157
5.1. Novidades nesse capítulo.....	157
5.2. Visão geral das fábricas de classes de dados.....	158
5.3. Tuplas nomeadas clássicas	162
5.4. Tuplas nomeadas com tipo	165
5.5. Introdução às dicas de tipo	166
5.6. Mais detalhes sobre @dataclass	171
5.7. A classe de dados como <i>cheiro no código</i>	181
5.8. Pattern Matching com instâncias de classes	183
5.9. Resumo do Capítulo.....	187
5.10. Leitura complementar	187
6. Referências, mutabilidade, e memória	192
6.1. Novidades nesse capítulo.....	192
6.2. Variáveis não são caixas.....	193

6.3. Identidade, igualdade e apelidos	194
6.4. A princípio, cópias são rasas.....	198
6.5. Parâmetros de função como referências	203
6.6. del e coleta de lixo	208
6.7. Peças que Python prega com imutáveis.....	210
6.8. Resumo do capítulo.....	211
6.9. Para saber mais	212
Parte II: Funções como objetos	216
7. Funções como objetos de primeira classe	217
7.1. Novidades nesse capítulo.....	217
7.2. Tratando uma função como um objeto	218
7.3. Funções de ordem superior.....	219
7.4. Funções anônimas.....	221
7.5. Os nove sabores de objetos invocáveis.....	222
7.6. Tipos invocáveis definidos pelo usuário	223
7.7. De parâmetros posicionais a parâmetros somente nomeados.....	225
7.8. Pacotes para programação funcional	227
7.9. Resumo do capítulo.....	232
7.10. Leitura complementar	233
8. Dicas de tipo em funções	236
8.1. Novidades nesse capítulo.....	237
8.2. Sobre tipagem gradual	237
8.3. Tipagem gradual na prática	238
8.4. Tipos são definidos pelas operações possíveis	243
8.5. Tipos próprios para anotações	248
8.6. Anotando parâmetros apenas posicionais e variádicos.....	275
8.7. Tipos imperfeitos e testes poderosos	276
8.8. Resumo do capítulo.....	277
8.9. Para saber mais	278

Dedication

Para Marta, com todo o meu amor.

Prefácio

Eis um plano: se uma pessoa usar um recurso que você não entende, mate-a. É mais fácil que aprender algo novo, e em pouco tempo os únicos programadores sobreviventes usarão apenas um subconjunto minúsculo e fácil de entender de Python 0.9.6 <piscadela marota>^[1].

— Tim Peters, lendário colaborador do CPython e autor do *Zen de Python*

"Python é uma linguagem fácil de aprender e poderosa." Essas são as primeiras palavras do tutorial oficial de Python 3.10 [<https://fpy.li/p-2>]. Isso é verdade, mas há uma pegadinha: como a linguagem é fácil de entender e de começar a usar, muitos programadores praticantes de Python se contentam apenas com uma fração de seus poderosos recursos.

Uma programadora experiente pode começar a escrever código Python útil em questão de horas. Conforme as primeiras horas produtivas se tornam semanas e meses, muitos desenvolvedores continuam escrevendo código Python com um forte sotaque das linguagens que aprenderam antes. Mesmo se Python for sua primeira linguagem, muitas vezes ela é apresentada nas universidades e em livros introdutórios evitando deliberadamente os recursos específicos da linguagem.

Como professor, ensinando Python para programadores experientes em outras linguagens, vejo outro problema: só sentimos falta daquilo que conhecemos. Vindo de outra linguagem, qualquer um é capaz de imaginar que Python suporta expressões regulares, e procurar esse tema na documentação. Mas se você nunca viu desempacotamento de tuplas ou descritores de atributos, talvez nunca procure por eles, e pode acabar não usando esses recursos, só por que são novos para você.

Este livro não é uma referência exaustiva de Python de A a Z. A ênfase está em recursos da linguagem característicos de Python ou incomuns em outras linguagens populares. Vamos nos concentrar principalmente nos aspectos centrais da linguagem e pacotes essenciais da biblioteca padrão. Apenas alguns exemplos mostram o uso de pacotes externos como FastAPI, httpx, e Curio.

Para quem é esse livro

Escrevi este livro para programadores que já usam Python e desejem se tornar fluentes em Python 3 moderno. Testei os exemplos em Python 3.10—e a maioria também em Python 3.9 e 3.8. Os exemplos que exigem especificamente Python 3.10 estão indicados.

Caso não tenha certeza se conhece Python o suficiente para acompanhar o livro, revise o tutorial oficial de Python [<https://fpy.li/4g>]. Tópicos tratados no tutorial não serão explicados aqui, exceto por alguns recursos mais novos.

Para quem esse livro não é

Se está começando a estudar Python, poderá achar difícil acompanhar este livro. Mais ainda, se você o ler muito cedo em sua jornada pela linguagem, pode ficar com a impressão que todo script Python precisa se valer de métodos especiais e truques de metaprogramação. Abstração prematura

é tão ruim quanto otimização prematura.

Para quem está aprendendo a programar, recomendo o livro *Pense em Python* [<https://fpy.li/4h>] de Allen Downey, disponível na Web.

Se já sabe programar e está aprendendo Python, o tutorial oficial de Python [<https://fpy.li/4g>] foi traduzido pela comunidade Python brasileira.

Como ler este livro

Esta edição impressa de *Python Fluente, Segunda Edição* está dividida em três volumes, contendo as cinco partes do original *Fluent Python, Second Edition* (O'Reilly, 2022), que foi publicado em um volume de 1012 páginas. O texto completo desta edição em português está disponível em pythonfluente.com [<https://pythonfluente.com>], em uma única página HTML para facilitar buscas e a leitura offline.

Recomendo que todos leiam o Capítulo 1. Após a leitura do capítulo "O modelo de dados de Python", o público principal deste livro não terá problema em pular diretamente para qualquer outra parte ou volume, mas muitas vezes assumo que você leu os capítulos precedentes de cada parte específica.

Tentei enfatizar o uso de classes e módulos que já existem antes de discutir como criar seus próprios. Por exemplo, no *Volume 1*, o Capítulo 2 trata dos tipos de sequências que estão prontas para serem usadas, incluindo algumas que não recebem muita atenção, como `collections.deque`. Criar sequências definidas pelo usuário só é discutido no *Volume 2, Parte III: Classes e Protocolos*, onde também vemos como usar as classes base abstratas (ABCs) de `collections.abc`. Criar suas próprias ABCs é discutido ainda mais tarde naquele volume, pois acredito na importância de estar confortável usando uma ABC antes de escrever uma.

Essa abordagem tem algumas vantagens. Primeiro, saber o que está disponível para uso imediato pode evitar que você reinvente a roda. Usamos as classes de coleções existentes com mais frequência que implementamos nossas próprias coleções, e podemos prestar mais atenção ao uso avançado de ferramentas prontas adiando a discussão sobre a criação de novas ferramentas. Também é mais provável herdarmos de ABCs existentes que criar uma nova ABC do zero. E, finalmente, acredito ser mais fácil entender as abstrações após vê-las em ação.

A desvantagem dessa estratégia são as referências a pontos futuros espalhadas por outros volumes. Espero que você possa tolerar esse inconveniente, agora que você sabe porque escolhi esse caminho.

Os três volumes e as 5 partes

O *Fluent Python (Second Edition)* original é dividido em 5 partes e foi publicado em um volume em inglês. Nesta edição em português brasileiro, dividimos as 5 partes em 3 volumes.

Volume 1—Estruturas de dados

Parte I—Estruturas de dados (capítulos 1-6)

O Capítulo 1 introduz o Modelo de Dados de Python e explica porque os métodos especiais (por

exemplo, `__repr__`) são a chave do comportamento consistente de objetos de todos os tipos. Os métodos especiais são tratados em maiores detalhes ao longo do livro. Os capítulos restantes dessa parte cobrem o uso de tipos coleção: sequências, mapeamentos e conjuntos, bem como a separação de `str` e `bytes`—causa de muitas celebrações entre usuários de Python 3, e de muita dor para usuários de Python 2 obrigados a migrar suas bases de código. Também são abordadas as fábricas de classe de alto nível na biblioteca padrão: fábricas de tuplas nomeadas e o decorador `@dataclass`. *Pattern matching* ("casamento de padrões")—novidade no Python 3.10—é tratada em seções do Capítulo 2, do Capítulo 3 e do Capítulo 5, que discutem padrões para sequências, padrões para mapeamentos e padrões para instâncias de classes. O último capítulo deste volume é sobre o ciclo de vida dos objetos: criação, referências, mutabilidade e coleta de lixo (*garbage collection*).

Parte II—Funções como objetos (capítulos 7-10)

Lá falamos sobre funções como objetos de primeira classe na linguagem: o significado disso, como isso afeta alguns padrões de projetos populares e como aproveitar as clausuras para implementar decoradores de função. Também são vistos aqui o conceito geral de invocáveis no Python, atributos de função, introspecção, anotação de parâmetros e a nova declaração `nonlocal` no Python 3. O *Cap. 8—Dicas de tipo em funções* introduz esse tópico importante e relativamente novo em Python.

Volume 2—Decoradores, classes e protocolos

Parte III—Classes e protocolos (capítulos 11-16)

Agora o foco se volta para a criação "manual" de classes—em contraste com o uso de fábricas de classe vistas no Capítulo 5. Como qualquer linguagem orientada a objetos, Python tem seu conjunto particular de recursos que podem ou não estar presentes na linguagem na qual você ou eu aprendemos programação baseada em classes. Os capítulos explicam como criar suas próprias coleções, classes base abstratas (ABCs) e protocolos, bem como as formas de lidar com herança múltipla e como implementar a sobrecarga de operadores, quando fizer sentido. O *Cap. 15—Mais dicas de tipo (vol. 2)* continua a conversa sobre dicas de tipo.

Volume 3—Concorrência e metaprogramação

Parte IV—Controle de fluxo (capítulos 17-21)

Nesta parte estudadis as instruções da linguagem e as bibliotecas que vão além do controle de fluxo tradicional (condicionais, laços e sub-rotinas). Começamos com os geradores, visitamos a seguir os gerenciadores de contexto e as corrotinas, incluindo a desafiadora mas poderosa sintaxe do `yield from`. O *Cap. 18—Instruções with, match, e blocos else (vol. 3)* inclui um exemplo significativo, usando *pattern matching* em um interpretador de linguagem simples mas funcional. O *Cap. 19—Modelos de concorrência em Python (vol. 3)* é novo, apresentando uma visão geral das alternativas para processamento concorrente e paralelo no Python, suas limitações, e como a arquitetura de software permite ao Python operar na escala da Web. Reescrevi o capítulo sobre *programação assíncrona*, para enfatizar os recursos centrais da linguagem—por exemplo, `await`, `async def`, `async for` e `async with`, e mostrar como eles são usados com `asyncio` e outros frameworks.

Parte V—Metaprogramação (capítulos 22-24)

Essa parte começa com uma revisão de técnicas para criação de classes com atributos criados

dinamicamente para lidar com dados semi-estruturados, tal como conjuntos de dados JSON. A seguir tratamos do mecanismo familiar das propriedades, antes de mergulhar no funcionamento do acesso a atributos de objetos no Python em um nível mais baixo, usando descritores. A relação entre funções, métodos e descritores é explicada. Ao longo daquele volume, a implementação passo a passo de uma biblioteca de validação de campos revela questões sutis, levando às ferramentas avançadas do capítulo final: decoradores de classes e metaclasses.

Abordagem "mão na massa"

Frequentemente usaremos o console interativo de Python para explorar a linguagem e as bibliotecas. Acho isso importante para enfatizar o poder dessa ferramenta de aprendizagem, especialmente para quem teve mais experiência com linguagens estáticas compiladas, que não oferecem um REPL.^[2]

Um dos pacotes padrão de testagem de Python, o *doctest* [<https://fpy.li/doctest>], funciona simulando sessões de console e verificando se as expressões resultam nas respostas exibidas. Usei *doctest* para verificar a maior parte do código desse livro, incluindo as listagens do console. Não é necessário usar ou sequer saber da existência do *doctest* para acompanhar o texto: a principal característica dos *doctests* é que eles imitam transcrições de sessões interativas no console de Python, assim qualquer pessoa pode reproduzir as demonstrações facilmente.

Algumas vezes vou explicar o que queremos realizar mostrando um *doctest* antes do código que implementa a solução. Estabelecer precisamente o quê deve ser feito, antes de pensar sobre como fazer, ajuda a focalizar nosso esforço de codificação. Escrever os testes previamente é a base de desenvolvimento dirigido por testes (TDD, *test-driven development*), e também acho essa técnica útil para ensinar.

Também escrevi testes de unidade para alguns dos exemplos maiores usando *pytest*—que acho mais fácil de usar e mais poderoso que o módulo *unittest* da biblioteca padrão. Você vai descobrir que pode verificar a maior parte do código do livro digitando `python3 -m doctest example_script.py` ou *pytest* no console de seu sistema operacional. A configuração do *pytest.ini*, na raiz do repositório do código de exemplo [<https://fpy.li/code>], assegura que *doctests* são coletados e executados pelo comando *pytest*.

Ponto de vista: minha perspectiva pessoal

Venho usando, ensinando e debatendo Python desde 1998, e gosto de estudar e comparar linguagens de programação, seus projetos e a teoria por trás delas. Ao final de alguns capítulos acrescentei uma seção "Ponto de vista", apresentando minha perspectiva sobre Python e outras linguagens. Você pode pular essas partes, se não tiver interesse em tais discussões. Seu conteúdo é inteiramente opcional.

Conteúdo na Web

Criei dois sites para este livro:

<https://pythonfluente.com>

O texto integral em português traduzido por Paulo Candido de Oliveira Filho. É que você está lendo agora.

<https://fluentpython.com>

Contém textos em inglês para ambas edições do livro, além de um glossário. É um material que eu cortei para não ultrapassar o limite de 1.000 páginas.

O repositório de exemplos de código está no GitHub [<https://fpy.li/code>].

Convenções usadas no livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica novos termos, URLs, endereços de email, nomes e extensões de arquivos ^[3].

Espaçamento constante

Usado para listagens de programas, bem como dentro de parágrafos para indicar elementos programáticos tais como nomes de variáveis ou funções, bancos de dados, tipos de dados, variáveis do ambiente, instruções e palavras-chave.

Observe que quando uma quebra de linha cai dentro de um termo de espaçoamento constante, o hífen não é utilizado—pois ele poderia ser erroneamente entendido como parte do termo.

Espaçamento constante em negrito

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

Espaçamento constante em itálico

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



Esse elemento é uma dica ou sugestão.



Este elemento é uma nota ou observação.



Este elemento é um aviso ou alerta.

Usando os exemplos de código

Todos os scripts e a maior parte dos trechos de código que aparecem no livro estão disponíveis no repositório de código de Python Fluente, no GitHub [<https://fpy.li/code>].

Se você tiver uma questão técnica ou algum problema para usar o código, por favor mande um email para bookquestions@oreilly.com.

Esse livro existe para ajudar você a fazer seu trabalho. Em geral, se o código exemplo está no livro,

você pode usá-lo em seus programas e na sua documentação. Não é necessário nos contactar para pedir permissão, a menos que você queira reproduzir uma parte significativa do código. Por exemplo, escrever um programa usando vários pedaços de código deste livro não exige permissão. Vender ou distribuir exemplos de livros da O'Reilly exige permissão. Responder uma pergunta citando este livro e código exemplo daqui não exige permissão. Incorporar uma parte significativa do código exemplo do livro na documentação de seu produto exige permissão.

Gostamos, mas em geral não exigimos, atribuição da fonte. Isto normalmente inclui o título, o autor, a editora e o ISBN. Por exemplo, “*Python Fluente*, 2ª ed., de Luciano Ramalho. Copyright 2022 Luciano Ramalho, 978-1-492-05635-5.”

Se você achar que seu uso dos exemplo de código está fora daquilo previsto na lei ou das permissões dadas acima, por favor entre em contato com permissions@oreilly.com.

O'Reilly Online Learning



Por mais de 40 anos, *O'Reilly Media* tem oferecido treinamento, conhecimento e ideias sobre tecnologia e negócios, ajudando empresas serem bem sucedidas.

Nossa rede sem igual de especialistas e inovadores compartilha conhecimento e sabedoria através de livros, artigos e de nossa plataforma online de aprendizagem. A plataforma de aprendizagem online da O'Reilly's oferece acesso sob demanda a treinamentos ao vivo, trilhas de aprendizagem profunda, ambientes interativos de programação e uma imensa coleção de textos e vídeos da O'Reilly e de mais de 200 outras editoras. Para maiores informações, visite <http://oreilly.com>.

Como entrar em contato

Por gentileza, envie comentários e perguntas sobre esse livro para o editor:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

Há uma página online para este livro, com erratas, exemplos e informação adicional, que pode ser acessada aqui: <https://fpy.li/p-4>.

<!--Don't forget to update the link above.-->

Envie email para bookquestions@oreilly.com, com comentários ou dúvidas técnicas sobre o livro.

Novidades e informações sobre nossos livros e cursos podem ser encontradas em <http://oreilly.com>.

Agradecimentos

Eu não esperava que atualizar um livro de Python cinco anos depois fosse um empreendimento de tal magnitude. Mas foi. Marta Mello, minha amada esposa, sempre esteve ao meu lado quando precisei. Meu querido amigo Leonardo Rochael me ajudou desde os primeiros rascunhos até a revisão técnica final, incluindo consolidar e revisar as sugestões dos outros revisores técnicos, de leitores e de editores. Honestamente, não sei se teria conseguido sem seu apoio, Marta e Leo. Muito, muito grato!

Jürgen Gmach, Caleb Hattingh, Jess Males, Leonardo Rochael e Miroslav Šedivý formaram a fantástica equipe de revisores técnicos da segunda edição. Eles revisaram o livro inteiro. Bill Behrman, Bruce Eckel, Renato Oliveira e Rodrigo Bernardo Pimentel revisaram capítulos específicos. Suas inúmeras sugestões, vindas de diferentes perspectivas, tornaram o livro muito melhor.

Muitos leitores me enviaram correções ou fizeram outras contribuições durante o pré-lançamento, incluindo: Guilherme Alves, Christiano Anderson, Konstantin Baikov, K. Alex Birch, Michael Boesl, Lucas Brunialti, Sergio Cortez, Gino Crecco, Chukwuerika Dike, Juan Esteras, Federico Fissore, Will Frey, Tim Gates, Alexander Hagerman, Chen Hanxiao, Sam Hyeong, Simon Ilincev, Parag Kalra, Tim King, David Kwast, Tina Lapine, Wanpeng Li, Guto Maia, Scott Martindale, Mark Meyer, Andy McFarland, Chad McIntire, Diego Rabatone Oliveira, Francesco Piccoli, Meredith Rawls, Michael Robinson, Federico Tula Rovaletti, Tushar Sadhwani, Arthur Constantino Scardua, Randal L. Schwartz, Avichai Sefati, Guannan Shen, William Simpson, Vivek Vashist, Jerry Zhang, Paul Zuradzki—e outros que pediram para não ter seus nomes mencionados, enviaram correções após a entrega da versão inicial ou foram omitidos porque eu não registrei seus nomes—mil desculpas.

Durante minha pesquisa, aprendi sobre tipagem, concorrência, *pattern matching* e metaprogramação interagindo com Michael Albert, Pablo Aguilar, Kaleb Barrett, David Beazley, J. S. O. Bueno, Bruce Eckel, Martin Fowler, Ivan Levkivskyi, Alex Martelli, Peter Norvig, Sebastian Rittau, Guido van Rossum, Carol Willing e Jelle Zijlstra.

Os editores da O'Reilly Jeff Bleiel, Jill Leonard e Amelia Blevins fizeram sugestões que melhoraram o fluxo do texto em muitas partes. Jeff Bleiel e o editor de produção Danny Elfmanbaum me apoiaram durante essa longa maratona.

As ideias e sugestões de cada um deles tornaram o livro melhor e mais preciso. Inevitavelmente, vão restar erros de minha própria criação no produto final. Me desculpo antecipadamente.

Por fim gostaria de estender meus sinceros agradecimento a meus colegas na Thoughtworks Brasil—e especialmente a meu mentor, Alexey Bôas, que apoiou este projeto de muitas formas até o fim.

Claro, todos os que me ajudaram a entender Python e a escrever a primeira edição merecem agora agradecimentos em dobro. Não haveria segunda edição sem o sucesso da primeira.

Agradecimentos da primeira edição

O tabuleiro e as peças de xadrez Bauhaus, criadas por Josef Hartwig, são um exemplo de um excelente design: belo, simples e claro. Guido van Rossum, filho de um arquiteto e irmão de

projetista de fonte magistral, criou um obra prima de design de linguagens. Adoro ensinar Python porque ele é belo, simples e claro.

Alex Martelli e Anna Ravenscroft foram os primeiros a verem o esquema desse livro, e me encorajaram a submetê-lo à O'Reilly para publicação. Seus livros me ensinaram Python idiomático e são modelos de clareza, precisão e profundidade em escrita técnica. Os 6,200+ posts de Alex no Stack Overflow [<https://fpy.li/p-7>] (EN) são uma fonte de boas ideias sobre a linguagem e seu uso apropriado.

Martelli e Ravenscroft foram também revisores técnicos deste livro, juntamente com Lennart Regebro e Leonardo Rochael. Todos nesta proeminente equipe de revisão técnica têm pelo menos 15 anos de experiência com Python, com muitas contribuições a projetos Python de alto impacto, em contato constante com outros desenvolvedores da comunidade. Em conjunto, eles me enviaram centenas de correções, sugestões, questões e opiniões, acrescentando imenso valor ao livro. Victor Stinner gentilmente revisou o *Cap. 21—Programação assíncrona* (vol. 3), trazendo seu conhecimento especializado, como um dos mantenedores do `asyncio`, para a equipe de revisão técnica. Foi um grande privilégio e um prazer colaborar com eles por estes muitos meses.

A editora Meghan Blanchette foi uma fantástica mentora, e me ajudou a melhorar a organização e o fluxo do texto do livro, me mostrando que partes estavam monótonas e evitando que eu atrasasse o projeto ainda mais. Brian MacDonald editou os capítulos da *Parte II* quando Meghan estava ausente. Adorei trabalhar com eles e com todos na O'Reilly, incluindo a equipe de suporte e desenvolvimento do Atlas (Atlas é a plataforma de publicação de livros da O'Reilly, que usei para escrever esse livro).

Mario Domenech Goulart deu sugestões numerosas e detalhadas, desde a primeira versão do livro. Também recebi muitas sugestões e comentários de Dave Pawson, Elias Dorneles, Leonardo Alexandre Ferreira Leite, Bruce Eckel, J. S. Bueno, Rafael Gonçalves, Alex Chiaranda, Guto Maia, Lucas Vido e Lucas Brunialti.

Ao longo dos anos, muitas pessoas me encorajaram a me tornar um autor, mas os mais persuasivos foram Rubens Prates, Aurelio Jargas, Rudá Moura e Rubens Altimari. Mauricio Bussab me abriu muitas portas, incluindo minha primeira experiência real na escrita de um livro. Renzo Nuccitelli apoiou este projeto de escrita o tempo todo, mesmo quando significou iniciar mais lentamente nossa parceria no python.pro.br.

A maravilhosa comunidade brasileira de Python é inteligente, generosa e divertida. O grupo Python Brasil [<https://fpy.li/p-9>] tem milhares de membros, e nossas conferências nacionais e regionais reúnem centenas de pessoas. Mas os mais influentes em minha jornada como pythonista foram Leonardo Rochael, Adriano Petrich, Daniel Vainsencher, Rodrigo RBP Pimentel, Bruno Gola, Leonardo Santagada, Jean Ferri, Rodrigo Senra, J. S. Bueno, David Kwast, Luiz Irber, Osvaldo Santana, Fernando Masanori, Henrique Bastos, Gustavo Niemayer, Pedro Werneck, Gustavo Barbieri, Lalo Martins, Danilo Bellini, e Pedro Kroger.

Dorneles Tremea foi um grande amigo, (e incrivelmente generoso com seu tempo e seu conhecimento), um hacker fantástico e o mais inspirador líder da Associação Python Brasil. Ele nos deixou cedo demais.

Meus estudantes, ao longo desses anos, me ensinaram muito através de suas perguntas, ideias,

feedbacks e soluções criativas para problemas. Érico Andrei e a Simples Consultoria tornaram possível que eu me concentrasse em ser um professor de Python pela primeira vez.

Martijn Faassen foi meu mentor de Grok e compartilhou ideias valiosas sobre Python e os neandertais. Seu trabalho e o de Paul Everitt, Chris McDonough, Tres Seaver, Jim Fulton, Shane Hathaway, Lennart Regebro, Alan Runyan, Alexander Limi, Martijn Pieters, Godefroid Chapelle e outros, dos planetas Zope, Plone e Pyramid, foram decisivos para minha carreira. Graças ao Zope e a surfar na primeira onda da web, pude começar a ganhar a vida com Python em 1998. José Octavio Castro Neves foi meu sócio na primeira software house baseada em Python do Brasil.

Tenho gurus demais na comunidade Python como um todo para listar todos aqui, mas além daqueles já mencionados, eu tenho uma dúvida com Steve Holden, Raymond Hettinger, A.M. Kuchling, David Beazley, Fredrik Lundh, Doug Hellmann, Nick Coghlan, Mark Pilgrim, Martijn Pieters, Bruce Eckel, Michele Simionato, Wesley Chun, Brandon Craig Rhodes, Philip Guo, Daniel Greenfeld, Audrey Roy e Brett Slatkin, por me ensinarem novas e melhores formas de ensinar Python.

A maior parte dessas páginas foi escrita no meu *home office* e em dois laboratórios: o CoffeeLab e o Garoa Hacker Clube. O CoffeeLab [<https://fpy.li/p-10>] é o quartel general dos geeks cafeinados na Vila Madalena, em São Paulo, Brasil. O Garoa Hacker Clube [<https://fpy.li/p-11>] é um espaço hacker aberto a todos: um laboratório comunitário onde qualquer um é livre para tentar novas ideias.

A comunidade Garoa me forneceu inspiração, infraestrutura e distração. Acho que Aleph gostaria desse lirô.

Minha mãe, Maria Lucia, e meu pai, Jairo, sempre me apoiaram de todas as formas. Gostaria que ele estivesse aqui para ver esse livro; e fico feliz de poder compartilhá-lo com ela.

Minha esposa, Marta Mello, suportou 15 meses de um marido que estava sempre trabalhando, mas continuou me apoiando e me guiando através dos momentos mais críticos do projeto, quando temi que poderia abandonar a maratona.

Agradeço a todos vocês, por tudo.

Sobre esta tradução

Python Fluente, Segunda Edição é uma tradução direta de *Fluent Python, Second Edition* (O'Reilly, 2022). Não é uma obra derivada de *Python Fluente* (Novatec, 2015).

A presente tradução foi autorizada pela O'Reilly Media para distribuição nos termos da licença CC BY-NC-ND [<https://fpy.li/4j>]. Os arquivos-fonte em formato *Asciidoc* estão no repositório público <https://github.com/pythonfluente/pythonfluente2e>.

Enquanto publicávamos a tradução ao longo de 2023, muitas correções foram enviadas por leitores como *issues* (defeitos) ou *pull requests* (correções) no repositório [<https://fpy.li/4k>]. Agradeço a todas as pessoas que colaboraram!



Correções e sugestões de melhorias são bem vindas! Para contribuir, veja os *issues* [<https://fpy.li/4m>] no repositório <https://github.com/pythonfluente/pythonfluente2e>.

Histórico das traduções

Escrevi a primeira e a segunda edições deste livro originalmente em inglês, para serem mais facilmente distribuídas no mercado internacional.

Cedi os direitos exclusivos para a O'Reilly Media, nos termos usuais de contratos com editoras famosas: elas ficam com a maior parte do lucro, o direito de publicar, e o direito de vender licenças para tradução em outros idiomas.

Até 2022, a primeira edição foi publicada nesses idiomas:

1. inglês,
2. português brasileiro,
3. chinês simplificado (China),
4. chinês tradicional (Taiwan),
5. japonês,
6. coreano,
7. russo,
8. francês,
9. polonês.

A ótima tradução PT-BR foi produzida e publicada no Brasil pela Editora Novatec em 2015, sob licença da O'Reilly.

Entre 2020 e 2022, atualizei e expandi bastante o livro para a segunda edição. Sou muito grato à liderança da Thoughtworks Brasil [<https://fpy.li/4n>] por terem me apoiado enquanto passei a maior parte de 2020 e 2021 pesquisando, escrevendo, e revisando esta edição.

Quando entreguei o manuscrito para a O'Reilly, negociei um adendo contratual para liberar a tradução da segunda edição em PT-BR com uma licença livre, como uma contribuição para comunidade Python lusófona.

A O'Reilly autorizou que essa tradução fosse publicada sob a licença CC BY-NC-ND: Creative Commons — Atribuição-NãoComercial-SemDerivações 4.0 Internacional [<https://fpy.li/4j>]. Com essa mudança contratual, a Editora Novatec não teve interesse em traduzir e publicar a segunda edição.

Felizmente encontrei meu querido amigo Paulo Cândido de Oliveira Filho (PC). Fomos colegas do ensino fundamental ao médio, e depois trabalhamos juntos como programadores em diferentes momentos e empresas. Hoje ele presta serviços editoriais, inclusive faz traduções com a excelente qualidade desta aqui.

Contratei PC para traduzir. Estou fazendo a revisão técnica, gerando os arquivos HTML com Asciidoc [https://fpy.li/4p] e publicando em <https://PythonFluente.com>. Estamos trabalhando diretamente a partir do *Fluent Python, Second Edition* da O'Reilly, sem aproveitar a tradução da

primeira edição, cujo copyright pertence à Novatec.

O copyright desta tradução pertence a mim.

Luciano Ramalho, São Paulo, 13 de março de 2023

[1] Mensagem para o grupo da Usenet comp.lang.python em 23 de dezembro de 2002: "Acrimony in c.l.p" [<https://fpy.li/p-1>] (EN).

[2] *Read-Eval-Print Loop*, o nome acadêmico de um console interativo que funciona como um laço lendo código, avaliando, e exibindo resultados.

[3] NT: Nesta edição em português também usamos *italico* em alguns termos mantidos em inglês ou traduções de termos cuja versão em português não é familiar

Parte I: Estruturas de dados

1. O modelo de dados de Python

O senso estético de Guido para o design de linguagens é incrível. Conheci muitos projetistas capazes de criar linguagens teoricamente lindas, que ninguém jamais usaria. Mas Guido é uma daquelas raras pessoas capaz de criar uma linguagem só um pouco menos teoricamente linda que, por isso mesmo, é uma delícia para programar.^[1]

— Jim Hugunin, criador do Jython, co-criador do AspectJ, e arquiteto do .Net DLR—Dynamic Language Runtime

Uma das melhores qualidades de Python é sua consistência. Após trabalhar com Python por algum tempo é possível intuir, de modo informado e correto, o funcionamento de recursos que você acabou de conhecer.

Entretanto, se você aprendeu outra linguagem orientada a objetos antes de Python, pode achar estranho usar `len(collection)` em vez de `collection.len()`. Essa peculiaridade é a ponta de um iceberg que, quando bem compreendido, é a chave para tudo aquilo que chamamos de *pythônico*. O iceberg se chama o Modelo de Dados de Python, e é a API que usamos para fazer nossos objetos lidarem bem com os recursos mais poderosos e característicos da linguagem.

É possível pensar no modelo de dados como uma descrição de Python na forma de um framework. Ele formaliza as interfaces dos elementos constituintes da própria linguagem, como sequências, funções, iteradores, corrotinas, classes, gerenciadores de contexto e assim por diante.

Quando usamos um framework, passamos um bom tempo programando métodos que são chamados pelo framework, e não pelas nossas classes. O mesmo acontece quando nos valemos do Modelo de Dados de Python para criar novas classes. O interpretador de Python invoca métodos especiais para realizar operações básicas sobre os objetos, muitas vezes acionadas por uma sintaxe especial. Os nomes dos métodos especiais são sempre precedidos e seguidos de dois sublinhados. Por exemplo, a sintaxe `obj[key]` está amparada no método especial `__getitem__`. Para resolver `my_collection[key]`, o interpretador chama `my_collection.__getitem__(key)`.

Implementamos métodos especiais quando queremos que nossos objetos suportem e interajam com elementos fundamentais da linguagem, como:

- Coleções
- Acesso a atributos
- Iteração (incluindo iteração assíncrona com `async for`)
- Sobrecarga (*overloading*) de operadores
- Invocação de funções e métodos
- Representação e formatação de strings
- Programação assíncrona usando `await`
- Criação e destruição de objetos

- Contextos gerenciados usando as instruções `with` ou `async with`

Mágica e o "dunder"

O termo *método mágico* é uma gíria usada para se referir aos métodos especiais, mas como falamos de um método específico, por exemplo `__getitem__`? Aprendi a dizer "dunder-getitem" com o autor e professor Steve Holden. "Dunder" é uma contração da frase em inglês "double underscore before and after" (*sublinhado duplo antes e depois*). Por isso os métodos especiais são também conhecidos como *métodos dunder*. O capítulo "Análise Léxica" [<https://fpy.li/2d>] de *A Referência da Linguagem Python* adverte: "Qualquer uso de nomes no formato `__*___` que não siga explicitamente o uso documentado, em qualquer contexto, está sujeito a quebra sem aviso prévio."



1.1. Novidades nesse capítulo

Esse capítulo sofreu poucas alterações desde a primeira edição, pois é uma introdução ao Modelo de Dados de Python, que é muito estável. As mudanças mais significativas foram:

- Métodos especiais que suportam programação assíncrona e outras novas funcionalidades foram acrescentados às tabelas em Seção 1.4.
- A Figura 2, mostrando o uso de métodos especiais em Seção 1.3.4, incluindo a classe base abstrata `collections.abc.Collection`, introduzida no Python 3.6.

Além disso, aqui e por toda essa segunda edição, adotei a sintaxe *f-string*, introduzida no Python 3.6, que é mais legível e muitas vezes mais conveniente que as notações de formatação de strings mais antigas: o método `str.format()` e o operador `%`.



Existe ainda uma razão para usar `my_fmt.format()`: quando a definição de `my_fmt` precisa vir de um lugar diferente daquele onde a operação de formatação precisa acontecer no código. Por exemplo, quando `my_fmt` tem múltiplas linhas e é melhor definida em uma constante, ou quando tem de vir de um arquivo de configuração ou de um banco de dados. Essas são necessidades reais, mas não acontecem com frequência.

1.2. Um baralho pythônico

O Exemplo 1 é simples, mas demonstra as possibilidades que se abrem com a implementação de apenas dois métodos especiais, `__getitem__` e `__len__`.

Exemplo 1. Um baralho como uma sequência de cartas

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
```

```

suits = 'spades diamonds clubs hearts'.split()

def __init__(self):
    self._cards = [Card(rank, suit) for suit in self.suits
                  for rank in self.ranks]

def __len__(self):
    return len(self._cards)

def __getitem__(self, position):
    return self._cards[position]

```

A primeira coisa a notar é o uso de `collections.namedtuple` para construir uma classe simples representando cartas individuais. Usamos `namedtuple` para criar classes de objetos que são apenas um agrupamento de atributos, sem métodos próprios, como um registro de banco de dados. Neste exemplo, a utilizamos para fornecer uma boa representação textual para as cartas em um baralho, como mostra a sessão no console:

```

>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')

```

Mas a parte central desse exemplo é a classe `FrenchDeck`. Ela é curta, mas poderosa. Primeiro, como qualquer coleção padrão de Python, uma instância de `FrenchDeck` responde à função `len()`, devolvendo o número de cartas naquele baralho:

```

>>> deck = FrenchDeck()
>>> len(deck)
52

```

Ler cartas específicas do baralho é fácil, graças ao método `__getitem__`. Por exemplo, a primeira e a última carta:

```

>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')

```

Deveríamos criar um método para obter uma carta aleatória? Não é necessário. Python já tem uma função que devolve um item aleatório de uma sequência: `random.choice`. Podemos usá-la em uma instância de `FrenchDeck`:

```

>>> from random import choice
>>> choice(deck)
Card(rank='3', suit='hearts')
>>> choice(deck)

```

```
Card(rank='K', suit='spades')
>>> choice(deck)
Card(rank='2', suit='clubs')
```

Acabamos de ver duas vantagens de usar os métodos especiais no contexto do Modelo de Dados de Python.

- Os usuários de suas classes não precisam memorizar nomes arbitrários de métodos para operações comuns ("Como obter o número de itens? É `.size()`, `.length()` ou outra coisa?")
- É mais fácil de aproveitar a rica biblioteca padrão de Python e evitar reinventar a roda, como no caso da função `random.choice`.

Mas não é só isso.

Como nosso `__getitem__` usa o operador `[]` de `self._cards`, nosso baralho suporta fatiamento automaticamente. Podemos olhar as três primeiras cartas no topo de um baralho, e depois pegar só os ases, iniciando com o índice 12 e pulando 13 cartas por vez:

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'),
 Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
 Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

E como já temos o método especial `__getitem__`, nosso baralho é um objeto iterável, ou seja, pode ser percorrido em um laço `for`:

```
>>> for card in deck: # doctest: +ELLIPSIS
...     print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
...
...
```

Também podemos iterar sobre o baralho na ordem inversa:

```
>>> for card in reversed(deck): # doctest: +ELLIPSIS
...     print(card)
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='Q', suit='hearts')
...
...
```



Reticências nos doctests

Sempre que possível, extraí as listagens do console de Python usadas neste livro

com o doctest [<https://fpy.li/2e>], para garantir a precisão. Quando a saída era longa demais, a parte omitida está marcada por reticências (...), como na última linha do trecho de código anterior.

Nesse casos, usei a diretiva `# doctest: +ELLIPSIS` para fazer o doctest funcionar. Ao experimentar esses exemplos no console iterativo, pode omitir todos os comentários de doctest.

A iteração muitas vezes é implícita. Python invoca o método `__contains__` da coleção para tratar o operador `in`: `student in team`. Mas se a coleção não fornece um método `__contains__`, o operador `in` realiza uma busca sequencial. No nosso caso, `in` funciona com nossa classe FrenchDeck porque ela é iterável. Veja a seguir:

```
>>> Card('Q', 'hearts') in deck
True
>>> Card('7', 'beasts') in deck
False
```

E o ordenamento? Um sistema comum de ordenar cartas é por seu valor numérico (ases sendo os mais altos) e depois por naipe, na ordem espadas (o mais alto), copas, ouros e paus (o mais baixo). Aqui está uma função que ordena as cartas com essa regra, devolvendo 0 para o 2 de paus e 51 para o Ás de espadas.

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)

def spades_high(card):
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

Podemos agora listar nosso baralho em ordem crescente de usando `spades_high` como critério de ordenação:

```
>>> for card in sorted(deck, key=spades_high): # doctest: +ELLIPSIS
...     print(card)
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
...
... (46 cards omitted)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```

Apesar da FrenchDeck herdar implicitamente da classe `object`, a maior parte de sua funcionalidade não é herdada, vem do modelo de dados e de composição. Ao implementar os métodos especiais `__len__` e `__getitem__`, nosso FrenchDeck se comporta como uma sequência Python padrão, podendo assim se beneficiar de recursos centrais da linguagem (por exemplo, iteração e fatiamento), e da

biblioteca padrão, como mostramos nos exemplos usando `random.choice`, `reversed`, e `sorted`. Graças à composição, as implementações de `__len__` e `__getitem__` podem delegar todo o trabalho para um objeto `list`, especificamente `self._cards`.

Como embaralhar as cartas?



Como foi implementado até aqui, um `FrenchDeck` não pode ser embaralhado, porque as cartas e suas posições não podem ser alteradas, exceto violando o encapsulamento e manipulando o atributo `_cards` diretamente. No [ch_ifaces_prot_abc] vamos corrigir isso acrescentando um método `__setitem__` de uma linha. Você consegue imaginar como ele seria implementado?

1.3. Como os métodos especiais são utilizados

A primeira coisa a saber sobre os métodos especiais é que eles são feitos para serem chamados pelo interpretador Python, e não por você. Você não escreve `my_object.__len__()`. Escreve `len(my_object)` e, se `my_object` é uma instância de uma classe definida por você, então Python chama o método `__len__` que você implementou.

Mas o interpretador pega um atalho quando está lidando com um tipo embutido como `list`, `str`, `bytearray`, ou extensões compiladas como os arrays do NumPy. As coleções de tamanho variável de Python escritas em C incluem uma struct^[2] chamada `PyVarObject`, com um campo `ob_size` que registra a quantidade de itens na coleção. Então, se `my_object` é uma instância de algum daqueles tipos embutidos, `len(my_object)` devolve diretamente valor do campo `ob_size`, e isso é mais rápido que chamar um método.

Na maior parte das vezes, a chamada a um método especial é implícita. Por exemplo, o comando `for i in x:` na verdade gera uma invocação de `iter(x)`, que por sua vez pode chamar `x.__iter__()` se esse método estiver disponível, ou usar `x.__getitem__()`, como no exemplo do `FrenchDeck`.

Em condições normais, seu código não deveria conter muitas chamadas diretas a métodos especiais. A menos que você esteja fazendo muita metaprogramação, implementar métodos especiais deve ser mais frequente que invocá-los explicitamente. O único método especial que é chamado frequentemente pelo seu código é `__init__`, para invocar a inicialização da superclasse na implementação do seu próprio `__init__`.

Geralmente, se você precisa invocar um método especial, é melhor chamar a função embutida relacionada (por exemplo, `len`, `iter`, `str`, etc.). Essas funções chamam o método especial correspondente, mas também fornecem outros serviços e—para tipos embutidos—são mais rápidas que chamadas a métodos. Veja, por exemplo, [iter_closer_look] no [ch_generators].

Na próxima seção veremos alguns dos usos mais importantes dos métodos especiais:

- Emular tipos numéricos
- Representar objetos na forma de strings
- Determinar o valor booleano de um objeto
- Implementar coleções

1.3.1. Emulando tipos numéricos

Vários métodos especiais permitem que objetos criados pelo usuário respondam a operadores como `+`. Vamos tratar disso com mais detalhes no [ch_op_overload]. Aqui nosso objetivo é continuar ilustrando o uso dos métodos especiais, através de outro exemplo simples.

Vamos implementar uma classe para representar vetores bi-dimensionais—isto é, vetores euclidianos como aqueles usados em matemática e física (veja a Figura 1).



O tipo embutido `complex` pode ser usado para representar vetores bi-dimensionais, mas nossa classe pode ser estendida para representar vetores N-dimensionais. Faremos isso em [ch_generators].

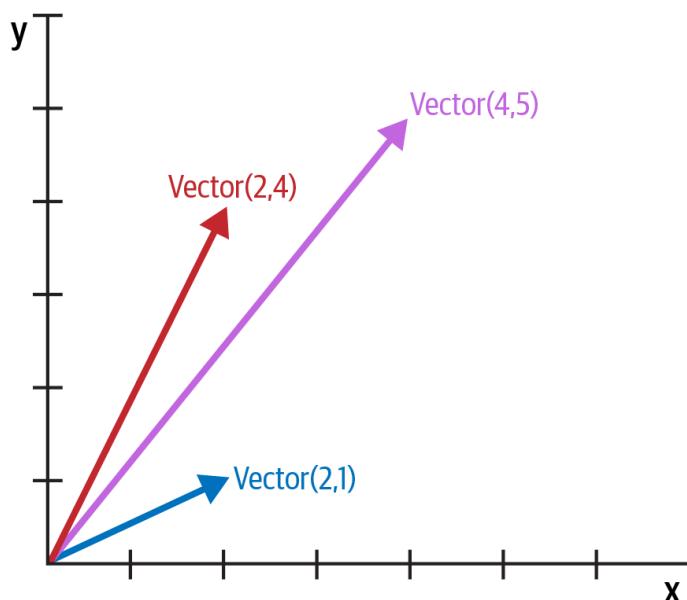


Figura 1. Soma de vetores bi-dimensionais; `Vector(2, 4) + Vector(2, 1)` devolve `Vector(4, 5)`.

Vamos começar a projetar a API para essa classe escrevendo uma sessão de console simulada, que depois podemos usar como um doctest. O trecho a seguir testa a adição de vetores ilustrada na Figura 1:

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
```

Observe como o operador `+` produz um novo objeto `Vector(4, 5)`.

A função embutida `abs` devolve o valor absoluto de números inteiros e de ponto flutuante, e a magnitude de números complexos. Então, por consistência, nossa API também usa `abs` para calcular a magnitude de um vetor:

```
>>> v = Vector(3, 4)
>>> abs(v)
```

Podemos também implementar o operador `*`, para realizar multiplicação escalar (isto é, multiplicar um vetor por um número para obter um novo vetor de mesma direção e magnitude multiplicada):

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0
```

O Exemplo 2 é uma classe `Vector` que implementa as operações descritas acima, usando os métodos especiais `__repr__`, `__abs__`, `__add__`, e `__mul__`.

Exemplo 2. Uma classe simples para representar um vetor 2D.

```
"""
vector2d.py: a simplistic class demonstrating some special methods

It is simplistic for didactic reasons. It lacks proper error handling,
especially in the ``__add__`` and ``__mul__`` methods.

This example is greatly expanded later in the book.
```

Addition::

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
```

Absolute value::

```
>>> v = Vector(3, 4)
>>> abs(v)
5.0
```

Scalar multiplication::

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0
```

"""

```
import math

class Vector:
```

```

def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

def __repr__(self):
    return f'Vector({self.x!r}, {self.y!r})'

def __abs__(self):
    return math.hypot(self.x, self.y)

def __bool__(self):
    return bool(abs(self))

def __add__(self, other):
    x = self.x + other.x
    y = self.y + other.y
    return Vector(x, y)

def __mul__(self, scalar):
    return Vector(self.x * scalar, self.y * scalar)

```

Implementamos cinco métodos especiais, além do costumeiro `__init__`. Veja que nenhum deles é chamado diretamente dentro da classe ou durante seu uso normal, ilustrado pelos doctests. Como mencionado antes, o interpretador Python é o único usuário frequente da maioria dos métodos especiais.

O Exemplo 2 implementa dois operadores: `+` e `*`, para demonstrar o uso básico de `__add__` e `__mul__`. No dois casos, os métodos criam e devolvem uma nova instância de `Vector`, e não modificam nenhum dos operandos: `self` e `other` são apenas lidos. Esse é o comportamento esperado de operadores infixos: criar novos objetos e não tocar em seus operandos. Vou falar mais sobre esse tópico no [ch_op_overload].



Da forma como está implementado, o Exemplo 2 permite multiplicar um `Vector` por um número, mas não um número por um `Vector`, violando a propriedade comutativa da multiplicação escalar. Vamos consertar isso com o método especial `__rmul__` no [ch_op_overload].

Nas seções seguintes vamos discutir os outros métodos especiais em `Vector`.

1.3.2. Representação como string

O método especial `__repr__` é chamado pela função embutida `repr` para obter a representação do objeto como uma string, para depuração. Sem um `__repr__` customizado, o console de Python mostraria uma instância de `Vector` como `<Vector object at 0x10e100070>`.

O console iterativo e o depurador chamam `repr` para exibir o resultado das expressões. O `repr` também é usado:

- Pelo marcador posicional `%r` na formatação clássica com o operador `%`. Ex.: `'%r' % my_obj`
- Pelo sinalizador de conversão `!r` na nova sintaxe de strings de formato [<https://fpy.li/2f>] usada nas *f-strings* e no método `str.format`. Ex: `f'{my_obj}!r'`

Note que a *f-string* no nosso `__repr__` usa `!r` para obter a representação padrão dos atributos a serem exibidos. Isso é uma boa prática, pois durante uma seção de depuração podemos ver a diferença entre `Vector(1, 2)` e `Vector('1', '2')`. Este segundo objeto não funcionaria no contexto desse exemplo, porque o construtor espera que os argumentos sejam números, não str.

A string devolvida por `__repr__` não deve ser ambígua e, se possível, deve corresponder ao código-fonte necessário para recriar o objeto representado. É por isso que nossa representação de `Vector` se parece com uma chamada ao construtor da classe, por exemplo `Vector(3, 4)`.

Por outro lado, `__str__` é chamado pela função embutida `str()` e usado automaticamente pela função `print`. Ele deve devolver uma string apropriada para ser exibida aos usuários finais da aplicação.

Algumas vezes a própria string devolvida por `__repr__` é adequada para exibir ao usuário, e você não precisa programar `__str__`, porque a implementação de `__str__` herdada da classe `object` já invoca `__repr__`. O Exemplo 65 é um dos muitos exemplos neste livro com um `__str__` customizado.



Programadores com experiência anterior em linguagens que contém o método `toString` tendem a implementar `__str__` e não `__repr__`. Se você for implementar apenas um desses métodos especiais, escolha `__repr__`.

"What is the difference between `__str__` and `__repr__` in Python?" (*Qual a diferença entre `__str__` e `__repr__` em Python?*) [<https://fpy.li/1-5>] (EN) é uma questão no Stack Overflow com excelentes contribuições dos pythonistas Alex Martelli e Martijn Pieters.

1.3.3. O valor booleano de um tipo customizado

Apesar de Python ter um tipo `bool`, a linguagem aceita qualquer objeto em um contexto booleano, tal como as expressões controlando instruções `if` ou `while`, ou como operandos de `and`, `or` e `not`. Para determinar se um valor `x` é *verdadeiro* ou *falso*, Python invoca `bool(x)`, que devolve somente `True` ou `False`.

Por padrão, instâncias de classes definidas pelo usuário são consideradas verdadeiras, a menos que `__bool__` ou `__len__` sejam implementadas. Basicamente, `bool(x)` chama `x.__bool__()` e usa o resultado. Se `__bool__` não está implementado, Python tenta invocar `x.__len__()`, e se esse último devolver zero, `bool` devolve `False`. Caso contrário, `bool` devolve `True`.

Nossa implementação de `__bool__` é simples: ela devolve `False` se a magnitude do vetor for zero, caso contrário devolve `True`. Convertemos a magnitude para um valor booleano usando `bool(abs(self))`, porque espera-se que `__bool__` devolva um booleano. Fora dos métodos `__bool__`, raramente é necessário chamar `bool()` explicitamente, porque qualquer objeto pode ser usado em um contexto booleano.

Observe que o método especial `__bool__` permite que seus objetos sigam as regras de teste do valor

verdade definidas no capítulo "Tipos Embutidos" [https://fpy.li/2g] da documentação da *Biblioteca Padrão de Python*.

Essa é uma implementação mais rápida de `Vector.__bool__`:

```
def __bool__(self):
    return bool(self.x or self.y)
```



Isso é mais difícil de ler, mas evita a jornada através de `abs`, `__abs__`, os quadrados, e a raiz quadrada. A conversão explícita para `bool` é necessária porque `__bool__` deve devolver um booleano, e `or` devolve um dos seus operandos no formato original: `x or y` resulta em `x` se `x` for verdadeiro, caso contrário resulta em `y`, qualquer que seja o valor deste último.

1.3.4. A API de Collection

A Figura 2 documenta as interfaces dos tipos de coleções essenciais na linguagem. Todas as classes no diagrama são ABCs—*classes base abstratas* (*ABC* é sigla para *Abstract Base Class*). As ABCs e o módulo `collections.abc` são tratados no [ch_ifaces_prot_abc]. O objetivo dessa pequena seção é dar uma visão panorâmica das interfaces das coleções mais importantes de Python, mostrando como elas são criadas a partir de métodos especiais.

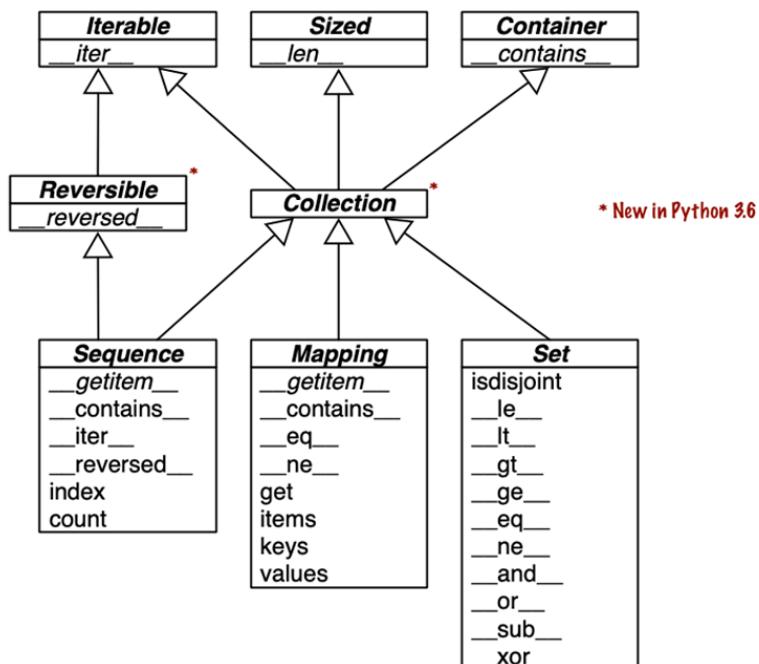


Figura 2. Diagrama de classes UML com os tipos fundamentais de coleções. Métodos com nome em itálico são abstratos, então precisam ser implementados pelas subclasses concretas, como `list` e `dict`. O restante dos métodos tem implementações concretas, então as subclasses podem herdá-los.

Cada uma das ABCs no topo da hierarquia tem um único método especial. A ABC `Collection` (introduzida no Python 3.6) unifica as três interfaces essenciais que toda coleção deveria implementar:

- `Iterable`, para suportar `for`, desempacotamento [https://fpy.li/2h], e outras formas de iteração

- Sized para suportar a função embutida `len`
- Container para suportar o operador `in`

Na verdade, Python não exige que classes concretas herdem de qualquer dessas ABCs. Qualquer classe que implemente `__len__` satisfaz a interface `Sized`.

Três especializações muito importantes de `Collection` são:

- Sequence, formalizando a interface de tipos embutidos como `list` e `str`
- Mapping, implementado por `dict`, `collections.defaultdict`, etc.
- Set, a interface dos tipos embutidos `set` e `frozenset`

Apenas `Sequence` é `Reversible`, porque sequências suportam o ordenamento arbitrário de seu conteúdo, ao contrário de mapeamentos(*mappings*) e conjuntos(*sets*).



Desde Python 3.7, o tipo `dict` é oficialmente "ordenado", mas isso só quer dizer que a ordem de inserção das chaves é preservada. Você não pode rearranjar as chaves em um `dict` da forma que quiser.

Todos os métodos especiais na ABC `Set` implementam operadores infixos. Por exemplo, `a & b` calcula a intersecção entre os conjuntos `a` e `b`, e é implementada no método especial `__and__`.

Os próximos dois capítulos vão tratar em detalhes das sequências, mapeamentos e conjuntos da biblioteca padrão.

Agora vamos considerar as duas principais categorias dos métodos especiais definidos no Modelo de Dados de Python.

1.4. Visão geral dos métodos especiais

O capítulo "Modelo de Dados" [<https://fpy.li/2j>] de *A Referência da Linguagem Python* lista mais de 80 nomes de métodos especiais. Mais da metade deles implementa operadores aritméticos, de comparação, ou bit-a-bit. Para ter uma visão geral do que está disponível, veja tabelas a seguir.

A Tabela 1 mostra nomes de métodos especiais, excluindo aqueles usados para implementar operadores infixos ou funções matemáticas fundamentais como `abs`. A maioria desses métodos será tratado ao longo do livro, incluindo as adições mais recentes: métodos especiais assíncronos como `__anext__` (acrescentado no Python 3.5), e o método de configuração de classes, `__init_subclass__` (do Python 3.6).

Tabela 1. Nomes de métodos especiais (excluindo operadores)

Categoria	Nomes dos métodos
Representação de string/bytes	<code>__repr__</code> <code>__str__</code> <code>__format__</code> <code>__bytes__</code> <code>__fspath__</code>
Conversão para número	<code>__bool__</code> <code>__complex__</code> <code>__int__</code> <code>__float__</code> <code>__hash__</code> <code>__index__</code>
Emulação de coleções	<code>__len__</code> <code>__getitem__</code> <code>__setitem__</code> <code>__delitem__</code> <code>__contains__</code>

Categoria	Nomes dos métodos
Iteração	<code>__iter__ __aiter__ __next__ __anext__ __reversed__</code>
Execução de chamável ou corrotina	<code>__call__ __await__</code>
Gerenciamento de contexto	<code>__enter__ __exit__ __aexit__ __aenter__</code>
Criação e destruição de instâncias	<code>__new__ __init__ __del__</code>
Gerenciamento de atributos	<code>__getattr__ __getattribute__ __setattr__ __delattr__ __dir__</code>
Descritores de atributos	<code>__get__ __set__ __delete__ __set_name__</code>
Classes base abstratas	<code>__instancecheck__ __subclasscheck__</code>
Metaprogramação de classes	<code>__prepare__ __init_subclass__ __class_getitem__ __mro_entries__</code>

Operadores infixos e numéricos são suportados pelos métodos especiais listados na Tabela 2. Aqui os nomes mais recentes são `__matmul__`, `__rmatmul__`, e `__imatmul__`, adicionados no Python 3.5 para suportar o uso de `@` como operador de multiplicação de matrizes, como veremos no [ch_op_overload].

Tabela 2. Nomes e símbolos de métodos especiais para operadores

Categoria do operador	Símbolos	Nomes de métodos
Unário numérico	<code>- + abs()</code>	<code>__neg__ __pos__ __abs__</code>
Comparação rica	<code>< <= == != > >=</code>	<code>__lt__ __le__ __eq__ __ne__ __gt__ __ge__</code>
Aritmético	<code>+ - * / // % @ divmod() round() ** pow()</code>	<code>__add__ __sub__ __mul__ __truediv__ __floordiv__ __mod__ __matmul__ __divmod__ __round__ __pow__</code>
Aritmética reversa	(operadores aritméticos com operandos invertidos)	<code>__radd__ __rsub__ __rmul__ __rtruediv__ __rfloordiv__ __rmod__ __rmatmul__ __rdivmod__ __rpow__</code>
Atribuição aritmética aumentada	<code>+= -= *= /= //= %= @= **=</code>	<code>__iadd__ __isub__ __imul__ __itruediv__ __ifloordiv__ __imod__ __imatmul__ __ipow__</code>
Bit a bit	<code>& ^ << >> ~</code>	<code>__and__ __or__ __xor__ __lshift__ __rshift__ __invert__</code>
Bit a bit reversa	(operadores bit a bit com os operandos invertidos)	<code>__rand__ __ror__ __rxor__ __rlshift__ __rrshift__</code>
Atribuição bit a bit aumentada	<code>&= = ^= <=>=</code>	<code>__iand__ __ior__ __ixor__ __ilshift__ __irshift__</code>

Python invoca um método especial de operador reverso no segundo argumento quando o método especial correspondente não pode ser usado no primeiro operando. Atribuições aumentadas são atalho combinando um operador infixo com uma atribuição de variável, por exemplo `a += b`.



O [ch_op_overload] explica em detalhes os operadores reversos e a atribuição aumentada.

1.5. Por que len não é um método?

Em 2013, fiz essa pergunta a Raymond Hettinger, um dos desenvolvedores principais do Python, e sua resposta foi basicamente uma citação do "The Zen of Python" (*O Zen do Python*) [<https://fpy.li/1-8>] (EN): "a praticidade vence a pureza." Na Seção 1.3, descrevi como `len(x)` roda muito rápido quando `x` é uma instância de um tipo embutido. Nenhum método é chamado para os objetos embutidos do CPython: o tamanho é simplesmente lido de um campo em uma struct C. Obter o número de itens em uma coleção é uma operação comum, e precisa funcionar de forma eficiente para tipos tão básicos e diferentes como `str`, `list`, `memoryview`, e assim por diante.

Em outras palavras, `len` não é chamado como um método porque recebe um tratamento especial como parte do Modelo de Dados de Python, da mesma forma que `abs`. Mas graças ao método especial `__len__`, também é possível fazer `len` funcionar com objetos nossas classes. Isso é um compromisso justo entre a necessidade de objetos embutidos eficientes e a consistência da linguagem. Também de "O Zen de Python": "Casos especiais não são especiais o bastante para quebrar as regras."

Pensar em `abs` e `len` como operadores unários nos deixa mais inclinados a perdoar seus aspectos funcionais, contrários à sintaxe de chamada de método que esperaríamos em uma linguagem orientada a objetos. De fato, Python herdou muito de sua sintaxe e estruturas de dados da linguagem ABC, onde existe o operador `#`, que equivale ao `len`: em ABC, `len(s)` escreve-se `#s`. Quando usado como operador infixo, `x#s` conta as ocorrências de `x` em `s`, que em Python obtemos com `s.count(x)`, para qualquer sequência `s`.



1.6. Resumo do capítulo

Ao implementar métodos especiais, as classes que você cria podem se comportar como os tipos embutidos, permitindo o estilo de programação expressivo que a comunidade considera *pythônico*.

Uma exigência básica para um objeto em Python é fornecer strings representando a si mesmo que possam ser usadas, uma para depuração e registro (*log*), outra para apresentar aos usuários finais. É para isso que os métodos especiais `__repr__` e `__str__` existem no modelo de dados.

Emular sequências, como mostrado com o exemplo do `FrenchDeck`, é um dos usos mais comuns dos métodos especiais. Por exemplo, bibliotecas de banco de dados frequentemente devolvem resultados de consultas na forma de coleções similares a sequências. Tirar o máximo proveito dos tipos de sequências existentes é o assunto do Capítulo 2. Como implementar suas próprias sequências será visto na [ch_seq_methods], onde criaremos uma extensão multidimensional da classe `Vector`.

Graças à sobrecarga de operadores, Python oferece uma rica seleção de tipos numéricos, desde os tipos embutidos até `decimal.Decimal` e `fractions.Fraction`, todos eles suportando operadores aritméticos infixos. As bibliotecas de ciência de dados `NumPy` suportam operadores infixos com

matrizes e tensores. A implementação de operadores—including operadores reversos e atribuição aumentada—será vista no [ch_op_overload], usando melhorias do exemplo `Vector`.

Também veremos o uso e a implementação da maioria dos outros métodos especiais do Modelo de Dados de Python ao longo deste livro.

1.7. Para saber mais

O capítulo "Modelo de Dados" [<https://fpy.li/2j>] em *A Referência da Linguagem Python* é a fonte canônica para o assunto desse capítulo e de uma boa parte deste livro.

Python in a Nutshell, 3rd ed. [<https://fpy.li/pynut3>] (EN), de Alex Martelli, Anna Ravenscroft, e Steve Holden (O'Reilly) tem uma excelente cobertura do modelo de dados. Sua descrição da mecânica de acesso a atributos é a mais competente que já vi, perdendo apenas para o próprio código-fonte em C do CPython. Martelli também é um contribuidor prolífico do Stack Overflow, com mais de 6200 respostas publicadas. Veja seu perfil de usuário no Stack Overflow [<https://fpy.li/1-9>].

David Beazley tem dois livros tratando do modelo de dados em detalhes, no contexto de Python 3: *Python Essential Reference* [<https://fpy.li/2k>] (EN), 4th ed. (Addison-Wesley), e *Python Cookbook*, 3rd ed [<https://fpy.li/pycook3>] (EN) (O'Reilly), colaborando com Brian K. Jones.

The Art of the Metaobject Protocol [<https://fpy.li/2m>] (EN) (MIT Press) de Gregor Kiczales, Jim des Rivieres, e Daniel G. Bobrow explica o conceito de um protocolo de metaobjetos, do qual o Modelo de Dados de Python é um exemplo.

Ponto de Vista

Modelo de dados ou modelo de objetos?

Aquilo que a documentação de Python chama de "Modelo de Dados de Python", a maioria dos autores diria que é o "Modelo de objetos de Python"

O *Python in a Nutshell*, 3rd ed. de Martelli, Ravenscroft, e Holden, e o *Python Essential Reference*, 4th ed., de David Beazley são os melhores livros sobre o Modelo de Dados de Python, mas se referem a ele como o "modelo de objetos." Na Wikipedia, a primeira definição de "modelo de objetos" [<https://fpy.li/1-10>] (EN) é: "as propriedades dos objetos em geral em uma linguagem de programação de computadores específica." É disso que o Modelo de Dados de Python trata. Neste livro, usarei "modelo de dados" porque esse é o título do capítulo de *A Referência da Linguagem Python* [<https://fpy.li/2j>] mais relevante para nossas discussões.

Métodos de "trouxas"

The Original Hacker's Dictionary (Dicionário Hacker Original) [<https://fpy.li/1-11>] (EN) define *mágica* como "algo ainda não explicado ou muito complicado para explicar" ou "uma funcionalidade, em geral não divulgada, que permite fazer algo que de outra forma seria impossível."

Ruby tem o equivalente aos métodos especiais, chamados de *métodos mágicos* naquela comunidade. Alguns na comunidade Python também adotam esse termo. Acredito que os

métodos especiais são o contrário de mágica. Python e Ruby oferecem a seus usuários um rico protocolo de metaobjetos integralmente documentado, permitindo que "trouxas" como você e eu possam emular muitas das funcionalidades disponíveis para os mantenedores que escrevem os interpretadores daquelas linguagens.

Por outro lado, pense em Go. Alguns objetos naquela linguagem tem funcionalidades que são mágicas, no sentido de não poderem ser emuladas em nossos próprios objetos definidos pelo usuário. Por exemplo, os arrays, strings e mapas de Go suportam o uso de colchetes para acesso a um item, na forma `a[i]`. Mas não há como fazer a notação `[]` funcionar com um novo tipo de coleção definida por você. Pior ainda, no ano 2021 Go não tem ainda o conceito de uma interface iterável ou um objeto iterador acessível ao usuário, daí sua sintaxe para `for/range` estar limitada a suportar cinco tipos "mágicos" embutidos, incluindo arrays, strings e mapas.

Talvez, no futuro, os projetistas de Go melhorem seu protocolo de metaobjetos. Em 2021, ele ainda é mais limitado do que Python, Ruby, e JavaScript oferecem.

Metaobjetos

The Art of the Metaobject Protocol (AMOP) (A Arte do protocolo de metaobjetos) é meu título favorito entre livros de computação. Mas o menciono aqui porque o termo *protocolo de metaobjetos* é útil para pensar sobre o Modelo de Dados de Python, e sobre recursos similares em outras linguagens. A parte *metaobjetos* se refere aos objetos que são os componentes essenciais da própria linguagem. Nesse contexto, *protocolo* é sinônimo de *interface*. Assim, um *protocolo de metaobjetos* é um sinônimo chique para modelo de objetos: uma API para os elementos fundamentais da linguagem.

Um protocolo de metaobjetos rico permite estender a linguagem para suportar novos paradigmas de programação. Gregor Kiczales, o primeiro autor do *AMOP*, mais tarde se tornou um pioneiro da programação orientada a aspecto, e o autor inicial do AspectJ, uma extensão de Java implementando aquele paradigma. A programação orientada a aspecto é mais fácil de implementar em uma linguagem dinâmica como Python, e alguns frameworks fazem exatamente isso. Um exemplo importante é a `zope.interface` [<https://fpy.li/1-12>] (EN), parte do framework Zope sobre o qual o sistema de gerenciamento de conteúdo Plone [<https://fpy.li/2n>] é construído.

[1] Traduzido de Story of Jython [<https://fpy.li/1-1>] (EN), prefácio de Jython Essentials [<https://fpy.li/1-2>] (EN), de Samuele Pedroni e Noel Rappin (O'Reilly).

[2] Uma struct do C é um tipo de registro com campos nomeados.

2. Uma coleção de sequências

Como podem ter notado, várias das operações mencionadas funcionam da mesma forma com textos, listas e tabelas. Coletivamente, textos, listas e tabelas são chamados de 'trens' (*trains*). [...] A instrução FOR também funciona, de forma genérica, com trens.

— Leo Geurts, Lambert Meertens, and Steven Pemberton, *ABC Programmer's Handbook* (Bosko Books) p.8

Antes de criar Python, Guido foi um dos desenvolvedores da linguagem ABC—um projeto de pesquisa de 10 anos para criar um ambiente de programação para iniciantes. A ABC introduziu várias ideias que hoje consideramos "pythônicas": operações genéricas com diferentes tipos de sequências, tipos tupla e mapeamento embutidos, estrutura de blocos por indentação, tipagem forte sem declaração de variáveis, entre outras. A usabilidade de Python não é acidental.

Python herdou da ABC o tratamento uniforme de sequências. Strings, listas, sequências de bytes, arrays, elementos XML e resultados vindos de bancos de dados compartilham um rico conjunto de operações comuns, incluindo iteração, fatiamento, ordenação e concatenação.

Entender a variedade de sequências disponíveis no Python evita reinventar a roda, e sua interface comum nos inspira a criar APIs que suportem e aproveitem bem os tipos de sequências existentes e futuros.

A maior parte deste capítulo trata das sequências em geral, desde a conhecida `list` até os tipos `str` e `bytes`, adicionados no Python 3. Tópicos específicos sobre listas, tuplas, arrays e filas também foram incluídos, mas os detalhes sobre strings Unicode e sequências de bytes são tratados no Capítulo 4. Além disso, a ideia aqui é falar sobre os tipos de sequências prontas para usar. A criação de novos tipos de sequência é o tema do [ch_seq_methods].

Os principais tópicos cobertos neste capítulo são:

- Compreensão de listas e os fundamentos das expressões geradoras
- Usar tuplas como registros ou como listas imutáveis
- Desempacotamento de sequências e padrões de sequências
- Fatiamento de sequências para leitura e escrita
- Tipos especializados de sequências, como arrays e filas

2.1. Novidades neste capítulo

A atualização mais importante desse capítulo é a Seção 2.6, primeira abordagem das instruções `match/case` introduzidas no Python 3.10.

As outras mudanças são aperfeiçoamentos da primeira edição:

- Um novo diagrama e uma nova descrição do funcionamento interno das sequências, contrastando contêineres e sequências planas.

- Uma comparação entre `list` e `tuple` quanto ao desempenho e ao armazenamento.
- Ressalvas sobre tuplas com elementos mutáveis, e como detectá-los se necessário.

Movi a discussão sobre tuplas nomeadas para a Seção 5.3 no Capítulo 5, onde elas são comparadas com `typing.NamedTuple` e `@dataclass`.



Para abrir espaço para conteúdo novo mantendo o número de páginas dentro do razoável, a seção "Managing Ordered Sequences with Bisect" ("Gerenciando sequências ordenadas com bisect") da primeira edição agora é um artigo [<https://fpy.li/bisect>] (EN) no site que complementa o livro.

2.2. Uma visão geral das sequências embutidas

A biblioteca padrão oferece uma boa seleção de tipos de sequências, implementadas em C:

Sequências contêiner

Podem armazenar itens de tipos diferentes, incluindo contêineres aninhados e objetos de qualquer tipo. Alguns exemplos: `list`, `tuple`, e `collections.deque`.

Sequências planas

Armazenaem itens de algum tipo simples, mas não outras coleções ou referências a objetos. Alguns exemplos: `str`, `bytes`, e `array.array`.

Uma *sequência contêiner* mantém referências para os objetos que contém, que podem ser de qualquer tipo, enquanto uma *sequência plana* armazena o valor de seu conteúdo em seu próprio espaço de memória, e não como objetos Python distintos. Veja a Figura 3.

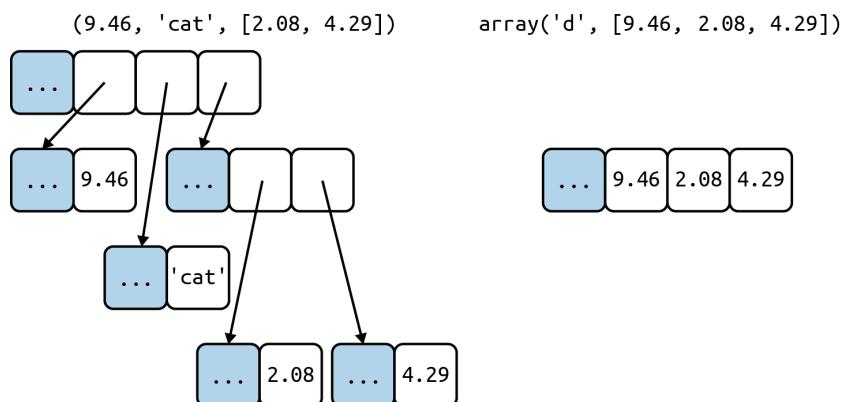


Figura 3. Diagramas de memória simplificados mostrando uma tuple e um array, cada uma com três itens. As células em cinza representam o cabeçalho de cada objeto Python na memória. A tuple tem um array de ponteiros para seus itens. Cada item é um objeto Python separado, possivelmente contendo também referências aninhadas a outros objetos Python, como aquela lista de dois itens. Por outro lado, um array Python é um único objeto, contendo um array da linguagem C com três números de ponto flutuante no formato nativo da CPU.

Dessa forma, sequências planas são mais compactas, mas só podem armazenar valores primitivos como bytes e números inteiros e de ponto flutuante.



Todo objeto Python na memória tem um cabeçalho com metadados. O objeto

Python mais simples, um float, tem um campo de valor e dois campos de metadados:

- ob_refcnt: a contagem de referências ao objeto
- ob_type: um ponteiro para o tipo do objeto
- ob_fval: um double de C mantendo o valor do float

No Python 64-bits, cada um desses campos ocupa 8 bytes. Por isso um array de números de ponto flutuante é mais compacto que uma tupla de números de ponto flutuante: o array é um único objeto contendo apenas o valor dos números, enquanto a tupla consiste de vários objetos—a própria tupla e cada objeto float que ela contém.

Outra forma de agrupar as sequências é por mutabilidade:

Sequências mutáveis

Por exemplo, list, bytearray, array.array e collections.deque.

Sequências imutáveis

Por exemplo, tuple, str, e bytes.

A Figura 4 ajuda a visualizar como as sequências mutáveis herdam todos os métodos das sequências imutáveis e implementam vários métodos adicionais. Os tipos embutidos de sequências na verdade não são subclasses das classes base abstratas (ABCs) Sequence e MutableSequence, mas sim *subclasses virtuais* registradas com aquelas ABCs—como veremos no [ch_ifaces_prot_abc]. Por serem subclasses virtuais, tuple e list passam nesses testes:

```
>>> from collections import abc
>>> issubclass(tuple, abc.Sequence)
True
>>> issubclass(list, abc.MutableSequence)
True
```

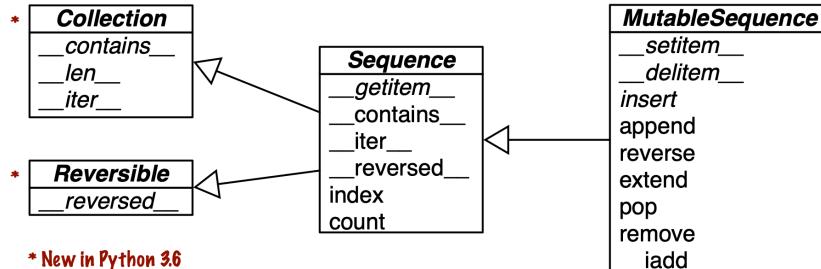


Figura 4. Diagrama de classe UML simplificado para algumas classes de collections.abc (as superclasses estão à esquerda; as setas de herança apontam das subclasses para as superclasses; nomes em itálico indicam classes e métodos abstratos).

Lembre-se dessas características básicas: mutável versus imutável; contêiner versus plana. Elas ajudam a extrapolar o que se sabe sobre um tipo de sequência para outros tipos.

O tipo mais fundamental de sequência é a lista: um contêiner mutável. Espero que você já esteja

muito familiarizada com listas, então vamos passar diretamente para a compreensão de listas, uma forma potente de criar listas que algumas vezes é subutilizada por sua sintaxe parecer, a princípio, estranha. Dominar as compreensões de listas abre as portas para expressões geradoras que—entre outros usos—podem produzir elementos para preencher sequências de qualquer tipo. Ambas são temas da próxima seção.

2.3. Compreensões de listas e expressões geradoras

Um jeito rápido de criar uma sequência é usando uma compreensão de lista (se o alvo é uma `list`) ou uma expressão geradora (para outros tipos de sequências). Se você não usa essas formas sintáticas diariamente, aposto que está perdendo oportunidades de escrever código mais legível e, muitas vezes, mais rápido também.

Se você duvida de minha alegação, sobre essas formas serem "mais legíveis", continue lendo. Vou tentar convencer você.



Por comodidade, muitos programadores Python se referem a compreensões de listas como *listcomps*, e a expressões geradoras como *genexps*. Usarei também esses dois termos.

2.3.1. Compreensões de lista e legibilidade

Em sua opinião qual desses exemplos é mais fácil de ler, o Exemplo 3 ou o Exemplo 4?

Exemplo 3. Cria uma lista de códigos Unicode a partir de uma string

```
>>> symbols = '$¢ƒ¥€¤'  
>>> codes = []  
>>> for symbol in symbols:  
...     codes.append(ord(symbol))  
...  
>>> codes  
[36, 162, 163, 165, 8364, 164]
```

Exemplo 4. Cria uma lista de códigos Unicode a partir de uma string, usando uma listcomp

```
>>> symbols = '$¢ƒ¥€¤'  
>>> codes = [ord(symbol) for symbol in symbols]  
>>> codes  
[36, 162, 163, 165, 8364, 164]
```

Qualquer um que saiba um pouco de Python consegue ler o Exemplo 3. Entretanto, após aprender sobre as listcomps, acho o Exemplo 4 mais legível, porque deixa sua intenção explícita.

Um loop `for` pode ser usado para muitas coisas diferentes: percorrer uma sequência para contar ou encontrar itens, computar valores agregados (somas, médias), ou inúmeras outras tarefas. O código no Exemplo 3 está criando uma lista. Uma listcomp, por outro lado, é mais clara. Seu objetivo é sempre criar uma nova lista.

Naturalmente, é possível abusar das comprehensões de lista para escrever código verdadeiramente incompreensível. Já vi código Python usando listcomps apenas para repetir um bloco de código por seus efeitos colaterais. Se você não vai fazer alguma coisa com a lista criada, não deveria usar essa sintaxe. Além disso, tente manter o código curto. Se uma compreensão ocupa mais de duas linhas, provavelmente seria melhor quebrá-la ou reescrevê-la como um bom e velho loop `for`. Avalie qual o melhor caminho: em Python, como em português, não existem regras absolutas para se escrever bem.



Dica de sintaxe

Em código Python, quebras de linha são ignoradas dentro de pares de `[]`, `{ }`, ou `()`. Então você pode usar múltiplas linhas para criar listas, listcomps, tuplas, dicionários, etc., sem necessidade de usar o marcador de continuação de linha `\`, que não funciona se após o `\` você acidentalmente digitar um espaço. Outro detalhe, quando aqueles pares de delimitadores são usados para definir um literal com uma série de itens separados por vírgulas, uma vírgula solta no final será ignorada. Daí, por exemplo, quando se codifica uma lista a partir de um literal com múltiplas linhas, é uma gentileza deixar uma vírgula após o último item. Isso torna um pouco mais fácil ao próximo programador acrescentar mais um item àquela lista, e reduz o ruído quando se lê os diffs.

Escopo local dentro de comprehensões e expressões geradoras

No Python 3, comprehensões de lista, expressões geradoras, e suas irmãs, as comprehensões de `set` e de `dict`, tem um escopo local para manter as variáveis criadas na condição `for`. Entretanto, variáveis atribuídas com o "operador morsa" ("Walrus operator"), `:=`, continuam acessíveis após aquelas comprehensões ou expressões retornarem—diferente das variáveis locais em uma função. A PEP 572—Assignment Expressions [<https://fpyli/pep572>] (EN) define o escopo do alvo de um `:=` como a função à qual ele pertence, exceto se houver uma declaração `global` ou `nonlocal` para aquele identificador.^[1]

```
>>> x = 'ABC'  
>>> codes = [ord(x) for x in x]  
>>> x ①  
'ABC'  
>>> codes  
[65, 66, 67]  
>>> codes = [last := ord(c) for c in x]  
>>> last ②  
67  
>>> c ③  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'c' is not defined
```

① `x` não foi sobreescrito: continua vinculado a 'ABC'.

② `last` permanece.

③ c desapareceu; ele só existiu dentro da listcomp.

Compreensões de lista criam listas a partir de sequências ou de qualquer outro tipo iterável, filtrando e transformando os itens. As funções embutidas `filter` e `map` podem fazer o mesmo, mas perdemos legibilidade, como veremos a seguir.

2.3.2. Listcomps versus map e filter

Listcomps fazem tudo que as funções map e filter fazem, sem os malabarismos exigidos pela funcionalidade limitada do lambda de Python.

Considere o Exemplo 5.

Exemplo 5. A mesma lista, criada por uma listcomp e por uma composição de map/filter

```
>>> symbols = '$¢£¥€¤'  
>>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]  
>>> beyond_ascii  
[162, 163, 165, 8364, 164]  
>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord, symbols)))  
>>> beyond_ascii  
[162, 163, 165, 8364, 164]
```

Eu imaginava que `map` e `filter` fossem mais rápidas que as `listcomps` equivalentes, mas Alex Martelli assinalou que não é o caso—pelo menos não nos exemplos acima. O script `listcomp_speed.py` [<https://fpy.li/2-1>] no repositório de código de Python Fluente [<https://fpy.li/code>] é um teste de desempenho simples, comparando `listcomp` com `filter/map`.

Vou falar mais sobre `map` e `filter` no Capítulo 7. Vamos agora ver o uso de `listcomps` para computar produtos cartesianos: uma lista contendo tuplas criadas a partir de todos os itens de duas ou mais listas.

2.3.3. Produtos cartesianos

Listcomps podem criar listas a partir do produto cartesiano de dois ou mais iteráveis. Os itens resultantes de um produto cartesiano são tuplas criadas com os itens de cada iterável na entrada, e a lista resultante tem o tamanho igual ao produto dos tamanhos dos iteráveis usados. Veja a Figura 5.

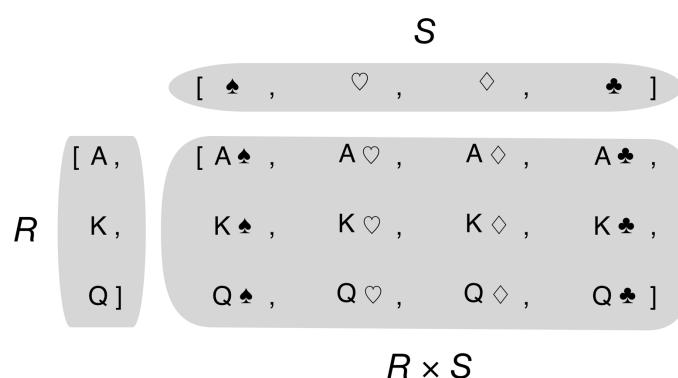


Figura 5. O produto cartesiano de 3 valores de cartas e 4 naipes é uma sequência de 12 itens.

Por exemplo, imagine que você precisa produzir uma lista de camisetas disponíveis em duas cores e três tamanhos. O Exemplo 6 mostra como produzir tal lista usando uma listcomp. O resultado tem seis itens.

Exemplo 6. Produto cartesiano usando uma compreensão de lista

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> tshirts = [(color, size) for color in colors for size in sizes] ①
>>> tshirts
[('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'),
 ('white', 'M'), ('white', 'L')]
>>> for color in colors: ②
...     for size in sizes:
...         print((color, size))
...
('black', 'S')
('black', 'M')
('black', 'L')
('white', 'S')
('white', 'M')
('white', 'L')
>>> tshirts = [(color, size) for size in sizes      ③
...                           for color in colors]
>>> tshirts
[('black', 'S'), ('white', 'S'), ('black', 'M'), ('white', 'M'),
 ('black', 'L'), ('white', 'L')]
```

① Isso gera uma lista de tuplas ordenadas por cor, depois por tamanho.

② Observe que a lista resultante é ordenada como se os loops for estivessem aninhados na mesma ordem que eles aparecem na listcomp.

③ Para ter os itens ordenados por tamanho e então por cor, apenas rearranje as cláusulas for; quebrar a listcomp em duas linhas torna mais fácil ver como o resultado será ordenado.

No Exemplo 1 (em Capítulo 1) usei a seguinte expressão para inicializar um baralho de cartas com uma lista contendo 52 cartas de todos os 13 valores possíveis para cada um dos quatro naipes, ordenada por naipe e então por valor:

```
self._cards = [Card(rank, suit) for suit in self.suits
               for rank in self.ranks]
```

Listcomps são mágicas de truque só: elas criam listas. Para gerar dados para outros tipos de sequências, uma genexp é o caminho. A próxima seção é uma pequena incursão às genexps, no contexto de criação de sequências que não são listas.

2.3.4. Expressões geradoras

Para inicializar tuplas, arrays e outros tipos de sequências, você também pode usar uma listcomp, mas uma genexp (expressão geradora) economiza memória, pois ela produz itens um de cada vez usando o protocolo iterador, em vez de criar uma lista inteira apenas para alimentar outro construtor.

As genexps usam a mesma sintaxe das listcomps, mas são delimitadas por parênteses em vez de colchetes.

O Exemplo 7 demonstra o uso básico de genexps para criar uma tupla e um array.

Exemplo 7. Inicializando uma tupla e um array a partir de uma expressão geradora

```
>>> symbols = '$¢¥€¤'  
>>> tuple(ord(symbol) for symbol in symbols) ①  
(36, 162, 163, 165, 8364, 164)  
>>> import array  
>>> array.array('I', (ord(symbol) for symbol in symbols)) ②  
array('I', [36, 162, 163, 165, 8364, 164])
```

① Se a expressão geradora é o único argumento em uma chamada de função, não há necessidade de duplicar os parênteses circundantes.

② O construtor de array espera dois argumentos, então os parênteses em torno da expressão geradora são obrigatórios. O primeiro argumento do construtor de array define o tipo de armazenamento usado para os números no array, como veremos na Seção 2.10.1.

O Exemplo 8 usa uma genexp com um produto cartesiano para gerar uma relação de camisetas de duas cores em três tamanhos. Diferente do Exemplo 6, aquela lista de camisetas com seis itens nunca é criada na memória: a expressão geradora alimenta o loop for produzindo um item por vez. Se as duas listas usadas no produto cartesiano tivessem mil itens cada uma, usar uma função geradora evitaria o custo de construir uma lista com um milhão de itens apenas para passar ao loop for.

Exemplo 8. Produto cartesiano em uma expressão geradora

```
>>> colors = ['black', 'white']  
>>> sizes = ['S', 'M', 'L']  
>>> for tshirt in (f'{c} {s}' for c in colors for s in sizes): ①  
...     print(tshirt)  
...  
black S  
black M  
black L  
white S  
white M  
white L
```

① A expressão geradora produz um item por vez; uma lista com todas as seis variações de

camisetas nunca é criada neste exemplo.



O [ch_generators] explica em detalhes o funcionamento de geradoras. A ideia aqui é apenas mostrar o uso de expressões geradoras para inicializar sequências diferentes de listas, ou produzir uma saída que não precise ser mantida na memória.

Vamos agora estudar outra sequência fundamental de Python: a tupla.

2.4. Tuplas não são apenas listas imutáveis

Alguns textos introdutórios de Python apresentam as tuplas como "listas imutáveis", mas isso é subestimá-las. Tuplas tem duas funções: elas podem ser usadas como listas imutáveis e também como registros sem nomes de campos. Esse uso algumas vezes é negligenciado, então vamos começar por ele.

2.4.1. Tuplas como registros

Tuplas podem conter registros: cada item na tupla contém os dados de um campo, e a posição do item indica seu significado.

Se você pensar em uma tupla apenas como uma lista imutável, a quantidade e a ordem dos elementos pode ser importante ou não, dependendo do contexto. Mas quando usamos uma tupla como uma coleção de campos, o número de itens em geral é fixo e sua ordem é sempre importante.

O Exemplo 9 mostra tuplas usadas como registros. Observe que, em todas as expressões, ordenar a tupla destruiria a informação, pois o significado de cada campo é dado por sua posição na tupla.

Exemplo 9. Tuplas usadas como registros

```
>>> lax_coordinates = (33.9425, -118.408056) ①
>>> city, year, pop, chg, area = ('Tokyo', 2003, 32_450, 0.66, 8014) ②
>>> traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'), ③
...     ('ESP', 'XDA205856')]
>>> for passport in sorted(traveler_ids): ④
...     print('%s/%s' % passport) ⑤
...
BRA/CE342567
ESP/XDA205856
USA/31195855
>>> for country, _ in traveler_ids: ⑥
...     print(country)
...
USA
BRA
ESP
```

① Latitude e longitude do Aeroporto Internacional de Los Angeles.

- ② Dados sobre Tóquio: nome, ano, população (em milhares), crescimento populacional (%) e área (km^2).
- ③ Uma lista de tuplas no formato (código_de_país, número_do_passaporte).
- ④ Iterando sobre a lista, `passport` é vinculado a cada tupla.
- ⑤ O operador de formatação `%` entende as tuplas e trata cada item como um campo separado.
- ⑥ A instrução `for` sabe como recuperar separadamente os itens de uma tupla—isto é chamado "desempacotamento" (*unpacking*). Aqui não estamos interessados no segundo item, então o atribuímos a `_`, uma variável descartável, apenas para coletar valores que não usaremos.



Em geral, usar `_` como variável descartável (*dummy variable*) é só uma convenção. É apenas um nome de variável estranho mas válido. Entretanto, em uma instrução `match/case`, o `_` é um coringa que corresponde a qualquer valor, mas não está vinculado a um valor. Veja a Seção 2.6. No console de Python, o resultado da instrução anterior é atribuído a `_`, exceto quando o resultado é `None`.

Muitas vezes pensamos em registros como estruturas de dados com campos nomeados. O Capítulo 5 apresenta duas formas de criar tuplas com campos nomeados.

Mas muitas vezes não é preciso se dar ao trabalho de criar uma classe apenas para nomear os campos, especialmente se você aproveitar o desempacotamento e evitar o uso de índices para acessar os campos. No Exemplo 9, atribuímos ('Tokyo', 2003, 32_450, 0.66, 8014) a `city`, `year`, `pop`, `chg`, `area` em uma única instrução. E daí o operador `%` atribuiu cada item da tupla `passport` para a posição correspondente da string de formato passada a `print`. Esses foram dois exemplos de *desempacotamento de tuplas*.



O termo "desempacotamento de tuplas" (*tuple unpacking*) é muito usado entre os pythonistas, mas *desempacotamento de iteráveis* é mais preciso e está ganhando popularidade, como no título da PEP 3132—Extended Iterable Unpacking (*Desempacotamento Estendido de Iteráveis*) [<https://fpy.li/2-2>].

A Seção 2.5 fala mais sobre desempacotamento, não apenas de tuplas, mas também de sequências e iteráveis em geral.

Agora vamos considerar o uso da classe `tuple` como uma variante imutável da classe `list`.

2.4.2. Tuplas como listas imutáveis

O interpretador Python e a biblioteca padrão fazem uso extensivo das tuplas como listas imutáveis, e você deve seguir o exemplo. Isso traz dois benefícios importantes:

Clareza

Quando você vê uma `tuple` no código, sabe que seu tamanho nunca mudará.

Desempenho

Uma `tuple` usa menos memória que uma `list` de mesmo tamanho, e permite ao Python realizar algumas otimizações.

Entretanto, lembre-se que a imutabilidade de uma tuple só se aplica às referências ali contidas. Referências em uma tupla não podem ser apagadas ou substituídas. Mas se uma daquelas referências apontar para um objeto mutável, e aquele objeto mudar, então o valor da tuple muda. O próximo trecho de código ilustra esse fato criando duas tuplas—`a` e `b`—que inicialmente são iguais. A Figura 6 representa a disposição inicial da tupla `b` na memória.

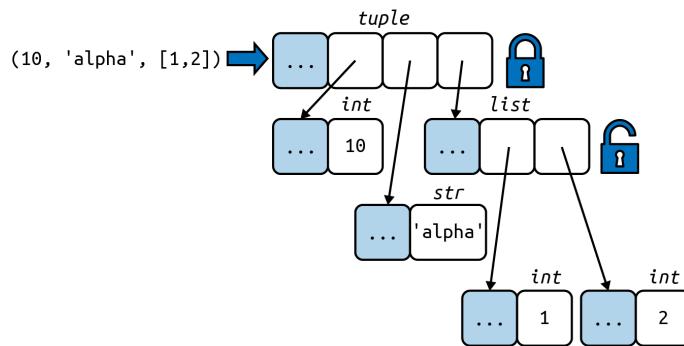


Figura 6. O conteúdo em si da tupla é imutável, mas isso significa apenas que as referências mantidas pela tupla vão sempre apontar para os mesmos objetos. Entretanto, se um dos objetos referenciados for mutável—uma lista, por exemplo—seu conteúdo pode mudar.

Quando o último item em `b` muda, `a` e `b` se tornam diferentes:

```
>>> a = (10, 'alpha', [1, 2])
>>> b = (10, 'alpha', [1, 2])
>>> a == b
True
>>> b[-1].append(99)
>>> a == b
False
>>> b
(10, 'alpha', [1, 2, 99])
```

Tuplas com itens mutáveis podem ser uma fonte de bugs. Se uma tupla contém qualquer item mutável, ela não pode ser usada como chave em um dict ou como elemento em um set. O motivo será explicado em Seção 3.4.1.

Se você quiser determinar explicitamente se uma tupla (ou qualquer outro objeto) tem um valor fixo, pode usar a função embutida `hash` para criar uma função `fixed`, assim:

```
>>> def fixed(o):
...     try:
...         hash(o)
...     except TypeError:
...         return False
...     return True
...
>>> tf = (10, 'alpha', (1, 2))
>>> tm = (10, 'alpha', [1, 2])
>>> fixed(tf)
True
```

```
>>> fixed(tm)
False
```

Vamos aprofundar essa questão em Seção 6.3.2.

Apesar dessa ressalva, as tuplas são frequentemente usadas como listas imutáveis. Elas oferecem algumas vantagens de desempenho, explicadas por uma das desenvolvedores principais de Python, Raymond Hettinger, em uma resposta à questão "Are tuples more efficient than lists in Python?" (*As tuplas são mais eficientes que as listas no Python?*) [<https://fpy.li/2-3>] no StackOverflow. Em resumo, Hettinger escreveu:

- Para avaliar uma tupla literal, o compilador Python gera bytecode para uma constante tupla em uma operação; mas para um literal lista, o bytecode gerado insere cada elemento como uma constante separada na pilha, e então cria a lista.
- Dada a tupla `t`, `tuple(t)` simplesmente devolve uma referência para a mesma `t`. Não há necessidade de cópia. Por outro lado, dada uma lista `l`, o construtor `list(l)` precisa criar uma nova cópia de `l`.
- Devido a seu tamanho fixo, uma instância de `tuple` tem alocado para si o espaço exato de memória que precisa. Em contrapartida, instâncias de `list` reservam memória adicional, para amortizar o custo de acréscimos futuros.
- As referências para os itens em uma tupla são armazenadas em um array na struct da tupla, enquanto uma lista mantém um ponteiro para um array de referências armazenada em outro lugar. Essa indireção é necessária porque, quando a lista cresce além do espaço alocado naquele momento, Python precisa reallocar o array de referências para criar espaço. A indireção adicional torna o cache da CPU menos eficiente.

2.4.3. Comparando os métodos de tuplas e listas

Quando usamos uma tupla como uma variante imutável de `list`, é bom saber o quanto similares são suas APIs. Como se pode ver na Tabela 3, `tuple` suporta todos os métodos de `list` que não envolvem adicionar ou remover itens, com uma exceção—`tuple` não possui o método `__reversed__`. Entretanto, `reversed(my_tuple)` funciona sem esse método; ele serve apenas para otimizar.

Tabela 3. Métodos e atributos encontrados em list ou tuple (os métodos implementados por object foram omitidos para economizar espaço)

	list	tuple	
<code>s.__add__(s2)</code>	•	•	<code>s + s2</code> —concatenação
<code>s.__iadd__(s2)</code>	•		<code>s += s2</code> —concatenação no mesmo lugar
<code>s.append(e)</code>	•		Acrescenta um elemento após o último
<code>s.clear()</code>	•		Apaga todos os itens
<code>s.__contains__(e)</code>	•	•	<code>e in s</code>
<code>s.copy()</code>	•		Cópia rasa da lista
<code>s.count(e)</code>	•	•	Conta as ocorrências de um elemento

	list	tuple	
s. <u>__delitem__</u> (p)	•		Remove o item na posição p
s.extend(it)	•		Acrescenta itens do iterável it
s. <u>__getitem__</u> (p)	•	•	s[p]—obtém o item na posição p
s. <u>__getnewargs__</u> ()		•	Suporte a serialização otimizada com pickle
s.index(e)	•	•	Encontra a posição da primeira ocorrência de e
s.insert(p, e)	•		Insere elemento e antes do item na posição p
s. <u>__iter__</u> ()	•	•	Obtém um iterador
s. <u>__len__</u> ()	•	•	len(s)—número de itens
s. <u>__mul__</u> (n)	•	•	s * n—concatenação repetida
s. <u>__imul__</u> (n)	•		s *= n—concatenação repetida no mesmo lugar
s. <u>__rmul__</u> (n)	•	•	n * s—concatenação repetida inversa ^[2]
s.pop([p])	•		Remove e devolve o último item ou o item na posição opcional p
s.remove(e)	•		Remove o primeiro elemento de valor igual a e
s.reverse()	•		Reverte, no lugar, a ordem dos itens
s. <u>__reversed__</u> ()	•		Obtém iterador para percorrer itens do último para o primeiro
s. <u>__setitem__</u> (p, e)	•		s[p] = e—coloca e na posição p, sobrescrevendo o item existente ^[3]
s.sort([key], [reverse])	•		Ordena os itens no lugar, com os argumentos nomeados opcionais key e reverse

Vamos agora examinar um tópico importante para a programação Python idiomática: tuplas, listas e desempacotamento iterável.

2.5. Desempacotando sequências e iteráveis

O desempacotamento é importante porque evita o uso de índices para acessar itens de sequências, o que causa muitos bugs. Além disso, o desempacotamento funciona tendo qualquer objeto iterável como fonte de dados—incluindo iteradores, que não suportam a notação de índice ([]). O único requisito é que o iterável produza exatamente um item por variável do lado esquerdo da atribuição, a menos que você use um asterisco (*) para capturar os itens em excesso, como explicado na Seção 2.5.1.

A forma mais visível de desempacotamento é a *atribuição paralela*; isto é, atribuir itens de um iterável a uma tupla de variáveis, como vemos nesse exemplo:

```
>>> lax_coordinates = (33.9425, -118.408056)
>>> latitude, longitude = lax_coordinates # unpacking
>>> latitude
```

```
33.9425
>>> longitude
-118.408056
```

Uma aplicação elegante de desempacotamento é permutar os valores de variáveis sem usar uma variável temporária:

```
>>> b, a = a, b
```

Outro exemplo de desempacotamento é prefixar um argumento com * ao chamar uma função:

```
>>> divmod(20, 8)
(2, 4)
>>> t = (20, 8)
>>> divmod(*t)
(2, 4)
>>> quotient, remainder = divmod(*t)
>>> quotient, remainder
(2, 4)
```

O código acima mostra outro uso do desempacotamento: permitir que funções devolvam múltiplos valores de forma conveniente para quem as chama. Em ainda outro exemplo, a função `os.path.split()` cria uma tupla (`path, last_part`) a partir de um caminho do sistema de arquivos:

```
>>> import os
>>> _, filename = os.path.split('/home/luciano/.ssh/id_rsa.pub')
>>> filename
'id_rsa.pub'
```

Outra forma de usar apenas alguns itens quando desempacotando é com a sintaxe *, que veremos a seguir.

2.5.1. Usando * para recolher itens em excesso

Definir parâmetros de função com `*args` para capturar argumentos em excesso é um recurso clássico de Python.

No Python 3, essa ideia foi estendida para se aplicar também à atribuição paralela:

```
>>> a, b, *rest = range(5)
>>> a, b, rest
(0, 1, [2, 3, 4])
>>> a, b, *rest = range(3)
>>> a, b, rest
(0, 1, [2])
>>> a, b, *rest = range(2)
```

```
>>> a, b, rest  
(0, 1, [])
```

No contexto da atribuição paralela, o prefixo * pode ser aplicado a exatamente uma variável, mas pode aparecer em qualquer posição:

```
>>> a, *body, c, d = range(5)  
>>> a, body, c, d  
(0, [1, 2], 3, 4)  
>>> *head, b, c, d = range(5)  
>>> head, b, c, d  
([0, 1], 2, 3, 4)
```

2.5.2. Desempacotando com * em chamadas de função e sequências literais

A PEP 448—Additional Unpacking Generalizations (*Generalizações adicionais de desempacotamento*) [<https://fpy.li/pep448>] (EN) introduziu uma sintaxe mais flexível para desempacotamento de iterável, melhor resumida em "O que há de novo no Python 3.5" [<https://fpy.li/2q>] (EN).

Em chamadas de função, podemos usar * múltiplas vezes:

```
>>> def fun(a, b, c, d, *rest):  
...     return a, b, c, d, rest  
...  
>>> fun(*[1, 2], 3, *range(4, 7))  
(1, 2, 3, 4, (5, 6))
```

O * pode também ser usado na definição de literais `list`, `tuple`, ou `set`, como visto nesses exemplos de "O que há de novo no Python 3.5" [<https://fpy.li/2q>] (EN):

```
>>> *range(4), 4  
(0, 1, 2, 3, 4)  
>>> [*range(4), 4]  
[0, 1, 2, 3, 4]  
>>> {*range(4), 4, *(5, 6, 7)}  
{0, 1, 2, 3, 4, 5, 6, 7}
```

A PEP 448 introduziu uma nova sintaxe similar para **, que veremos na Seção 3.2.2.

Por fim, outro importante aspecto do desempacotamento de tuplas: funciona com estruturas aninhadas.

2.5.3. Desempacotamento aninhado

O alvo de um desempacotamento pode usar aninhamento, por exemplo (a, b, (c, d)). Python fará a coisa certa se o valor tiver a mesma estrutura aninhada. O Exemplo 10 mostra o

desempacotamento aninhado em ação.

Exemplo 10. Desempacotando tuplas aninhadas para acessar a longitude

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)), ①
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]

def main():
    print(f'{"":>15} | {"latitude":>9} | {"longitude":>9}')
    for name, _, _, (lat, lon) in metro_areas: ②
        if lon <= 0: ③
            print(f'{name:>15} | {lat:9.4f} | {lon:9.4f}')

if __name__ == '__main__':
    main()
```

① Cada tupla contém um registro com quatro campos, o último deles um par de coordenadas.

② Ao atribuir o último campo a uma tupla aninhada, desempacotamos as coordenadas.

③ O teste `lon <= 0`: seleciona apenas cidades no hemisfério ocidental.

A saída do Exemplo 10 é:

	latitude	longitude
Mexico City	19.4333	-99.1333
New York-Newark	40.8086	-74.0204
São Paulo	-23.5478	-46.6358

O alvo da atribuição de um desempacotamento pode também ser uma lista, mas bons casos de uso aqui são raros. Aqui está o único que conheço: se você tem uma consulta de banco de dados que devolve um único registro (por exemplo, se o código SQL tem a instrução `LIMIT 1`), daí é possível desempacotar e ao mesmo tempo se assegurar que há apenas um resultado com o seguinte código:

```
>>> [record] = query_returning_single_row()
```

Se o registro contiver apenas um campo, é possível obtê-lo diretamente, assim:

```
>>> [[field]] = query_returning_single_row_with_single_field()
```

Ambos os exemplos acima podem ser escritos com tuplas, mas não esqueça da peculiaridade sintática, tuplas com um único item devem ser escritas com uma vírgula final. Então o primeiro alvo seria (`record,`) e o segundo ((`field,`)). Nos dois casos, esquecer aquela vírgula causa um bug

silencioso.^[4]

Agora vamos estudar *pattern matching*, que suporta maneiras ainda mais poderosas para desempacotar sequências.

2.6. Pattern matching com sequências

O novo recurso mais visível de Python 3.10 é o *pattern matching* (casamento de padrões) com a instrução `match/case`, proposta na PEP 634—Structural Pattern Matching: Specification (*Casamento Estrutural de Padrões: Especificação*) [<https://fpy.li/pep634>] (EN).

 Carol Willing, uma das desenvolvedoras principais de Python, escreveu uma excelente introdução ao *pattern matching* na seção "Casamento de padrão estrutural" [<https://fpy.li/2r>]^[5] em "O que há de novo no Python 3.10" [<https://fpy.li/2s>]. Você pode querer ler aquela revisão rápida. Neste livro, dividi o tratamento do casamento de padrões em diferentes capítulos, dependendo dos tipos de padrão: na Seção 3.3 e na Seção 5.8. E há um exemplo mais longo na [pattern_matching_case_study_sec].

Vamos ao primeiro exemplo do tratamento de sequências com `match/case`.

Imagine que você está construindo um robô que aceita comandos, enviados como sequências de palavras e números, como `BEEPER 440 3`. Após separar o comando em partes e analisar os números, você teria uma mensagem como `['BEEPER', 440, 3]`. Então, você poderia usar um método assim para interpretar mensagens naquele formato:

Exemplo 11. Método de uma classe Robot imaginária

```
def handle_command(self, message):
    match message: ①
        case ['BEEPER', frequency, times]: ②
            self.beep(times, frequency)
        case ['NECK', angle]: ③
            self.rotate_neck(angle)
        case ['LED', ident, intensity]: ④
            self.leds[ident].set_brightness(ident, intensity)
        case ['LED', ident, red, green, blue]: ⑤
            self.leds[ident].set_color(ident, red, green, blue)
        case _: ⑥
            raise InvalidCommand(message)
```

① A expressão após a palavra-chave `match` é o sujeito (*subject*). O sujeito contém os dados que Python vai comparar aos padrões em cada instrução `case`.

② Esse padrão casa com qualquer sujeito que seja uma sequência de três itens. O primeiro item deve ser a string `BEEPER`. O segundo e o terceiro itens podem ser qualquer coisa, e serão vinculados às variáveis `frequency` e `times`, nessa ordem.

③ Isso casa com qualquer sujeito com dois itens, se o primeiro for `'NECK'`.

- ④ Isso vai casar com uma sujeito de três itens começando com LED. Se o número de itens não for correspondente, Python segue para o próximo case.
- ⑤ Outro padrão de sequência começando com 'LED', agora com cinco itens—incluindo a constante 'LED'.
- ⑥ Esse é o case default. Vai casar com qualquer sujeito que não tenha sido capturado por um dos padrões precedentes. A variável _ é especial, como logo veremos.

Olhando superficialmente, `match/case` se parece instrução `switch/case` da linguagem C—mas isso é só uma pequena parte da sua funcionalidade.^[6] Uma melhoria fundamental do `match` sobre o `switch` é a *desestruturação*—uma forma mais avançada de desempacotamento. Desestruturação é uma palavra nova no vocabulário de Python, mas é usada com frequência na documentação de linguagens que suportam o *pattern matching*—como Scala e Elixir.

Como um primeiro exemplo de desestruturação, o Exemplo 12 mostra parte do Exemplo 10 reescrito com `match/case`.

Exemplo 12. Desestruturando tuplas aninhadas—requer Python ≥ 3.10

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]

def main():
    print(f'{"":>15} | {"latitude":>9} | {"longitude":>9}')
    for record in metro_areas:
        match record: ①
            case [name, _, _, (lat, lon)] if lon <= 0: ②
                print(f'{name:>15} | {lat:9.4f} | {lon:9.4f}')
```

① O sujeito desse `match` é `record`—isto é, cada uma das tuplas em `metro_areas`.

② Uma instrução `case` tem duas partes: um padrão e uma guarda opcional, com a palavra-chave `if`.

Em geral, um padrão de sequência casa com o sujeito se estas três condições forem verdadeiras:

1. O sujeito é uma sequência, *e*
2. O sujeito e o padrão tem o mesmo número de itens, *e*
3. Todos os itens correspondentes casam, incluindo os itens aninhados.

Por exemplo, o padrão `[name, _, _, (lat, lon)]` no Exemplo 12 casa com uma sequência de quatro itens, e o último item tem que ser uma sequência de dois itens.

Padrões de sequência podem ser escritos como tuplas e listas, mas a sintaxe usada não faz diferença: em um padrão de sequência, colchetes e parênteses tem o mesmo significado. Escrevi o padrão como uma lista com uma tupla aninhada de dois itens para evitar a repetição de colchetes

ou parênteses no Exemplo 12.

Um padrão de sequência pode casar com instâncias da maioria das subclasses reais ou virtuais de `collections.abc.Sequence`, com a exceção de `str`, `bytes`, e `bytearray`.

Instâncias de `str`, `bytes`, e `bytearray` não são tratadas como sequências no contexto de um `match/case`. Um sujeito de `match` de um desses tipos é tratado como um valor "atômico"—assim como o inteiro 987 é tratado como um único valor, e não como uma sequência de dígitos. Tratar aqueles três tipos como sequências poderia causar bugs devido a casamentos não intencionais. Se você quer usar um objeto daqueles tipos como um sujeito sequência, converta-o na instrução `match`. Por exemplo, veja `tuple(phone)` no trecho abaixo, que poderia ser usado para separar números de telefone por regiões do mundo com base no prefixo DDI:



```
match tuple(phone):
    case ['1', *rest]: # North America and Caribbean
        ...
    case ['2', *rest]: # Africa and some territories
        ...
    case ['3' | '4', *rest]: # Europe
        ...
```

Na biblioteca padrão, os seguintes tipos são compatíveis com padrões de sequência:

```
list      memoryview     array.array
tuple    range           collections.deque
```

Ao contrário do desempacotamento, padrões não desestruturam iteráveis que não sejam sequências (tal como os iteradores).

O símbolo `_` é especial nos padrões: ele casa com qualquer item naquela posição, mas nunca é vinculado ao valor daquele item. O valor é descartado. Além disso, o `_` é a única variável que pode aparecer mais de uma vez em um padrão.

Você pode vincular qualquer parte de um padrão a uma variável usando a palavra-chave `as`:

```
case [name, _, _, (lat, lon) as coord]:
```

Dado o sujeito `['Shanghai', 'CN', 24.9, (31.1, 121.3)]`, o padrão anterior vai casar e atribuir valores às seguintes variáveis:

Variável	Valor atribuído
name	'Shanghai'
lat	31.1
lon	121.3

Variável	Valor atribuído
coord	(31.1, 121.3)

Para tornar os padrões mais específicos, podemos incluir informação de tipo. Por exemplo, o seguinte padrão casa com a mesma estrutura de sequência aninhada do exemplo anterior, mas o primeiro item deve ser uma instância de str, e ambos os itens da tupla devem ser instâncias de float:

```
case [str(name), _, _, (float(lat), float(lon))]:
```



As expressões str(name) e float(lat) se parecem com chamadas a construtores, que usariam para converter name e lat para str e float. Mas no contexto de um padrão, aquela sintaxe faz uma checagem de tipos durante a execução do programa: o padrão acima vai casar com uma sequência de quatro itens, na qual o item 0 deve ser uma str e o item 3 deve ser um par de números de ponto flutuante. Além disso, a str no item 0 será vinculada à variável name e os números no item 3 serão vinculados a lat e lon, respectivamente. Assim, apesar de imitar a sintaxe de uma chamada de construtor, o significado de str(name) é totalmente diferente no contexto de um padrão. O uso de classes arbitrárias em padrões será tratado na Seção 5.8.

Por outro lado, se queremos casar qualquer sujeito sequência começando com uma str e terminando com uma sequência aninhada com dois números de ponto flutuante, podemos escrever:

```
case [str(name), *_, (float(lat), float(lon))]:
```

O *_ casa com qualquer número de itens, sem vinculá-los a uma variável. Usar *extra em vez de *_ vincularia os itens a extra como uma list com 0 ou mais itens.

A instrução de guarda opcional começando com if só é avaliada se o padrão casar, e pode se referir a variáveis vinculadas no padrão, como no Exemplo 12:

```
match record:
    case [name, _, _, (lat, lon)] if lon <= 0:
        print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')
```

O bloco aninhado com a instrução print só será executado se o padrão casar e a expressão guarda for verdadeira.



A desestruturação com padrões é tão expressiva que, algumas vezes, um match com um único case pode tornar o código mais simples. Guido van Rossum tem uma coleção de exemplos de case/match, incluindo um que ele chamou de "A very deep iterable and type match with extraction" (*Um match de iterável e tipo muito profundo, com extração*) [<https://fpy.li/2-10>] (EN).

O Exemplo 12 não é melhor que o Exemplo 10. É apenas um exemplo para contrastar duas formas de fazer a mesma coisa. O próximo exemplo mostra como o *pattern matching* contribui para a criação de código claro, conciso e eficaz.

2.6.1. Casando padrões de sequência em um interpretador

Peter Norvig, da Universidade de Stanford, escreveu o *lis.py* [<https://fpy.li/2-11>]: um interpretador de um subconjunto do dialeto Scheme da linguagem de programação Lisp, em 132 belas linhas de código Python legível. Peguei o código fonte de Norvig (publicado sob a licença MIT) e o atualizei para Python 3.10, para exemplificar o *pattern matching*. Nessa seção, vamos comparar uma parte fundamental do código de Norvig—que usa *if/elif* e desempacotamento—with uma nova versão usando *match/case*.

As duas funções principais do *lis.py* são *parse* e *evaluate*.^[7] O parser (*analisador sintático*) recebe as expressões entre parênteses do Scheme e devolve listas Python. Aqui estão dois exemplos:

```
>>> parse('(gcd 18 45)')
['gcd', 18, 45]
>>> parse('''
... (define double
...     (lambda (n)
...         (* n 2)))
... ''')
['define', 'double', ['lambda', ['n'], ['*', 'n', 2]]]
```

O avaliador recebe listas como essas e as executa. O primeiro exemplo está chamando uma função *gcd* com 18 e 45 como argumentos. Quando executada, ela computa o maior divisor comum (*gcd* são as iniciais do termo em inglês *greatest common divisor*) dos argumentos (que é 9). O segundo exemplo está definindo uma função chamada *double* com um parâmetro *n*. O corpo da função é a expressão $(* n 2)$. O resultado da chamada a uma função em Scheme é o valor da última expressão no corpo da função chamada.

Nosso foco aqui é a desestruturação de sequências, então não vou explicar as ações do avaliador. Veja a [pattern_matching_case_study_sec] para aprender mais sobre o funcionamento do *lis.py*.

O Exemplo 13 mostra o avaliador de Norvig com algumas pequenas modificações, abreviado para mostrar apenas os padrões de sequência.

Exemplo 13. Casando padrões sem match/case

```
def evaluate(exp: Expression, env: Environment) -> Any:
    "Evaluate an expression in an environment."
    if isinstance(exp, Symbol):      # variable reference
        return env[exp]
    # ... lines omitted
    elif exp[0] == 'quote':          # (quote exp)
        (_, x) = exp
        return x
    elif exp[0] == 'if':              # (if test conseq alt)
```

```

(_, test, consequence, alternative) = exp
if evaluate(test, env):
    return evaluate(consequence, env)
else:
    return evaluate(alternative, env)
elif exp[0] == 'lambda':          # (lambda (parm...) body...)
    (_, parms, *body) = exp
    return Procedure(parms, body, env)
elif exp[0] == 'define':
    (_, name, value_exp) = exp
    env[name] = evaluate(value_exp, env)
# ... more lines omitted

```

Observe como cada instrução `elif` verifica o primeiro item da lista, e então desempacota a lista, ignorando o primeiro item. O uso extensivo do desempacotamento sugere que Norvig é um fã do *pattern matching*, mas ele originalmente escreveu aquele código em Python 2 (apesar de agora ele funcionar com qualquer Python 3)

Usando `match/case` em Python ≥ 3.10 , podemos refatorar `evaluate`, como mostrado no Exemplo 14.

Exemplo 14. Pattern matching com match/case—requer Python ≥ 3.10

```

def evaluate(exp: Expression, env: Environment) -> Any:
    "Evaluate an expression in an environment."
    match exp:
        # ... lines omitted
        case ['quote', x]: ①
            return x
        case ['if', test, consequence, alternative]: ②
            if evaluate(test, env):
                return evaluate(consequence, env)
            else:
                return evaluate(alternative, env)
        case ['lambda', [*parms], *body] if body: ③
            return Procedure(parms, body, env)
        case ['define', Symbol() as name, value_exp]: ④
            env[name] = evaluate(value_exp, env)
        # ... more lines omitted
        case _: ⑤
            raise SyntaxError(lispstr(exp))

```

① Casa se o sujeito for uma sequência de dois itens começando com 'quote'.

② Casa se o sujeito for uma sequência de quatro itens começando com 'if'.

③ Casa se o sujeito for uma sequência com três ou mais itens começando com 'lambda'. A guarda assegura que `body` não esteja vazio.

④ Casa se o sujeito for uma sequência de três itens começando com 'define', seguido de uma instância de `Symbol`.

⑤ É uma boa prática ter um case para capturar qualquer outro sujeito. Neste exemplo, se exp não casar com nenhum dos padrões, a expressão está mal-formada, então gera um `SyntaxError`.

Sem o último case, para pegar tudo que tiver passado pelos anteriores, o bloco `match` não faz nada quando o sujeito não casa com algum case—e isso pode causar uma falha silenciosa.

Norvig deliberadamente evitou a checagem e o tratamento de erros em `lis.py`, para manter o código fácil de entender. Com *pattern matching*, podemos acrescentar mais verificações e ainda manter o programa legível. Por exemplo, no padrão '`define`', o código original não se assegura que `name` é uma instância de `Symbol`—isso exigiria um bloco `if`, uma chamada a `isinstance`, e mais código. O Exemplo 14 é mais curto e mais seguro que o Exemplo 13.

2.6.1.1. Padrões alternativos para `lambda`

Essa é a sintaxe de `lambda` no Scheme, usando a convenção sintática onde o sufixo `...` significa que o elemento pode aparecer zero ou mais vezes:

```
(lambda (parms...) body1 body2...)
```

Um padrão simples para o case de '`lambda`' seria esse:

```
case ['lambda', parms, *body] if body:
```

Entretanto, isso casa com qualquer valor na posição `parms`, incluindo o primeiro `x` nesse sujeito inválido:

```
['lambda', 'x', ['*', 'x', 2]]
```

A lista aninhada após a palavra-chave `lambda` do Scheme contém os nomes do parâmetros formais da função, e deve ser uma lista mesmo que contenha apenas um elemento. Ela pode também ser uma lista vazia, se função não receber parâmetros—como a `random.random()` de Python.

No Exemplo 14, tornei o padrão de '`lambda`' mais seguro usando um padrão de sequência aninhado:

```
case ['lambda', [*parms], *body] if body:  
    return Procedure(parms, body, env)
```

Em um padrão de sequência, o `*` pode aparecer apenas uma vez por sequência. Aqui temos duas sequências: a externa e a interna.

Acrescentando os caracteres `[*]` em torno de `parms` fez o padrão mais parecido com a sintaxe do Scheme da qual ele trata, e nos deu uma verificação estrutural adicional.

2.6.1.2. Sintaxe abreviada para definição de função

O Scheme tem uma sintaxe alternativa de `define`, para criar uma função nomeada sem usar um

lambda aninhado. Tal sintaxe funciona assim:

```
(define (name parms...) body1 body2...)
```

A palavra-chave `define` é seguida por uma lista com o nome da nova função e zero ou mais nomes de parâmetros. Após a lista vem o corpo da função, com uma ou mais expressões.

Acrescentar essas duas linhas ao `match` cuida da implementação:

```
case ['define', [Symbol() as name, *parms], *body] if body:  
    env[name] = Procedure(parms, body, env)
```

Eu colocaria esse `case` após o `case` da outra forma de `define` no Exemplo 14. A ordem desses `cases` de `define` é irrelevante nesse exemplo, pois nenhum sujeito pode casar com esses dois padrões: o segundo elemento deve ser um `Symbol` na forma original de `define`, mas deve ser uma sequência começando com um `Symbol` na sintaxe de `define` para definição de função.

Agora pense em quanto trabalho teríamos para adicionar o suporte a essa segunda sintaxe de `define` sem a ajuda do *pattern matching* no Exemplo 13. A instrução `match` faz mais que o `switch` das linguagens similares ao C.

O *pattern matching* é um exemplo de programação declarativa: o código descreve "o que" você quer casar, em vez de "como" casar. A forma do código segue a forma dos dados, como ilustra a Tabela 4.

Tabela 4. Algumas formas sintáticas do Scheme e os padrões de case para tratá-las

Sintaxe do Scheme	Padrão de sequência
<code>(quote exp)</code>	<code>['quote', exp]</code>
<code>(if test conseq alt)</code>	<code>['if', test, conseq, alt]</code>
<code>(lambda (parms...) body1 body2...)</code>	<code>['lambda', [*parms], *body] if body</code>
<code>(define name exp)</code>	<code>['define', Symbol() as name, exp]</code>
<code>(define (name parms...) body1 body2...)</code>	<code>['define', [Symbol() as name, *parms], *body] if body</code>

Espero que a refatoração do `evaluate` de Norvig com *pattern matching* tenha convencido você que `match/case` pode tornar seu código mais legível e mais seguro.



Veremos mais do `lis.py` na `[pattern_matching_case_study_sec]`, quando vamos revisar o exemplo completo de `match/case` em `evaluate`. Se você quiser aprender mais sobre o `lys.py` de Norvig, leia seu maravilhoso post "(How to Write a (Lisp) Interpreter (in Python))" (*Como Escrever um Interpretador (Lisp) em (Python)*) [<https://fpy.li/2-12>].

Isso conclui nossa primeira passagem por desempacotamento, desestruturação e *pattern matching* com sequências. Vamos tratar de outros tipos de padrões mais adiante, em outros capítulos.

Todo programador Python sabe que sequências podem ser fatiadas usando a sintaxe `s[a:b]`. Vamos

agora examinar alguns fatos menos conhecidos sobre fatiamento.

2.7. Fatiamento

Um recurso comum a `list`, `tuple`, `str`, e a todos os tipos de sequência em Python, é o suporte a operações de fatiamento, que são mais potentes do que a maioria das pessoas percebe.

Nesta seção descrevemos o *uso* dessas formas avançadas de fatiamento. Sua implementação em uma classe definida pelo usuário será tratada no [ch_seq_methods], mantendo nossa filosofia de tratar de classes prontas para usar nessa parte do livro, e da criação de novas classes na [classes_protocols_part].

2.7.1. Por que fatias e faixas excluem o último item?

A convenção pythônica de excluir o último item em fatias e faixas funciona bem com a indexação iniciada no zero usada no Python, no C e em muitas outras linguagens. Algumas vantagens desta convenção:

- É fácil ver o tamanho da fatia ou da faixa quando apenas a posição final é dada: tanto `range(3)` quanto `my_list[:3]` produzem três itens.
- É fácil calcular o tamanho de uma fatia ou de uma faixa quando o início e o fim são dados: basta subtrair `fim - início`.
- É fácil cortar uma sequência em duas partes em qualquer índice `x`, sem sobreposição: escreva `my_list[:x]` e `my_list[x:]`. Por exemplo:

```
>>> l = [10, 20, 30, 40, 50, 60]
>>> l[:2] # split at 2
[10, 20]
>>> l[2:]
[30, 40, 50, 60]
>>> l[:3] # split at 3
[10, 20, 30]
>>> l[3:]
[40, 50, 60]
```

Agora vamos olhar mais de perto a forma como Python interpreta a notação de fatiamento.

2.7.2. Objetos fatia

Isso não é segredo, mas vale a pena repetir, só para ter certeza: `s[a:b:c]` pode ser usado para especificar um passo ou salto `c`, fazendo com que a fatia resultante pule itens. O passo pode ser também negativo, devolvendo os itens em ordem inversa. Veja três exemplos:

```
>>> s = 'bicycle'
>>> s[::-3]
'bye'
>>> s[::-1]
```

```
'elcycib'  
>>> s[::-2]  
'eccb'
```

Vimos outro exemplo no Capítulo 1, quando usamos `deck[12::13]` para obter todos os ases de uma baralho não embaralhado:

```
>>> deck[12::13]  
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),  
 Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

A notação `a:b:c` só é válida entre `[]` quando usada como operador de indexação ou de subscrição (*subscript*), e produz um objeto fatia (*slice object*): `slice(a, b, c)`. Para avaliar a expressão `seq[start:stop:step]`, o Python chama `seq.__getitem__(slice(start, stop, step))`, como veremos na [how_slicing_works]. Mesmo se você não for implementar seus próprios tipos de sequência, saber dos objetos fatia é útil, porque eles permitem que você atribua nomes às fatias, da mesma forma que planilhas permitem dar nomes a faixas de células.

Suponha que você precise analisar um arquivo de dados como a fatura mostrada na Exemplo 15. Em vez de encher seu código de fatias explícitas fixas, você pode nomeá-las. Veja como isso torna legível o loop `for` no final do exemplo.

Exemplo 15. Itens de um arquivo tabular de fatura

```
>>> invoice = """  
... 0.....6.....40.....52...55.....  
... 1909 Pimoroni PiBrella           $17.50   3   $52.50  
... 1489 6mm Tactile Switch x20      $4.95    2   $9.90  
... 1510 Panavise Jr. - PV-201       $28.00   1   $28.00  
... 1601 PiTFT Mini Kit 320x240     $34.95   1   $34.95  
... """  
>>> SKU = slice(0, 6)  
>>> DESCRIPTION = slice(6, 40)  
>>> UNIT_PRICE = slice(40, 52)  
>>> QUANTITY = slice(52, 55)  
>>> ITEM_TOTAL = slice(55, None)  
>>> line_items = invoice.split('\n')[2:]  
>>> for item in line_items:  
...     print(item[UNIT_PRICE], item[DESCRIPTION])  
...  
$17.50  Pimoroni PiBrella  
$4.95   6mm Tactile Switch x20  
$28.00  Panavise Jr. - PV-201  
$34.95  PiTFT Mini Kit 320x240
```

Voltaremos aos objetos `slice` quando formos discutir a criação de suas próprias coleções, na [sliceable_sequence]. Enquanto isso, do ponto de vista do usuário, o fatiamento tem recursos adicionais, como fatias multidimensionais e a notação de reticências (...). Siga comigo.

2.7.3. Fatiamento multidimensional e reticências

O operador `[]` pode também receber múltiplos índices ou fatias separadas por vírgulas. Os métodos especiais `__getitem__` e `__setitem__`, que tratam o operador `[]`, recebem os índices em `a[i, j]` como uma tupla. Em outras palavras, para avaliar `a[i, j]`, Python chama `a.__getitem__((i, j))`.

Isso é usado, por exemplo, no pacote externo NumPy, onde itens de uma `numpy.ndarray` bi-dimensional podem ser recuperados usando a sintaxe `a[i, j]`, e uma fatia bi-dimensional é obtida com uma expressão como `a[m:n, k:l]`. O Exemplo 24, abaixo nesse mesmo capítulo, mostra o uso dessa notação.

Exceto por `memoryview`, os tipos embutidos de sequência de Python são uni-dimensionais, então aceitam só um índice ou fatia, e não uma tupla de índices ou fatias.^[8]

As reticências—escritas como três pontos finais (...) e não como … (Unicode U+2026)—são reconhecidas como um símbolo pelo parser de Python. Esse símbolo é um apelido para o objeto `Ellipsis`, a única instância da classe `ellipsis`.^[9] Dessa forma, ele pode ser passado como argumento para funções e como parte da especificação de uma fatia, como em `f(a, ..., z)` ou `a[i:\dots]`. O NumPy usa `...` como atalho ao fatiar arrays com muitas dimensões; por exemplo, se `x` é um array com quatro dimensões, `x[i, \dots]` é um atalho para `x[i, :, :, :, :]`. Veja "NumPy quickstart" [<https://fpy.li/2-13>] (EN) para saber mais sobre isso.

Desconheço usos de `Ellipsis` ou de índices multidimensionais na biblioteca padrão de Python. Esses recursos sintáticos existem para suportar tipos definidos pelo usuário ou extensões como o NumPy.

Fatias não são úteis apenas para extrair informações de sequências; elas podem também ser usadas para modificar sequências mutáveis no lugar—isto é, sem precisar reconstruí-las do zero.

2.7.4. Atribuindo a fatias

Sequências mutáveis podem ser enxertadas, extirpadas e, de várias maneiras modificadas no lugar com o uso da notação de fatias no lado esquerdo de uma instrução de atribuição ou como alvo de uma instrução de `del`. Os próximos exemplos dão uma ideia do poder dessa notação:

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:5] = [20, 30]
>>> l
[0, 1, 20, 30, 5, 6, 7, 8, 9]
>>> del l[5:7]
>>> l
[0, 1, 20, 30, 5, 8, 9]
>>> l[3::2] = [11, 22]
>>> l
[0, 1, 20, 11, 5, 22, 9]
>>> l[2:5] = 100 ①
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: can only assign an iterable  
>>> l[2:5] = [100]  
>>> l  
[0, 1, 100, 22, 9]
```

- ① Quando o alvo de uma atribuição é uma fatia, o lado direito deve ser um objeto iterável, mesmo que tenha apenas um item.

Todo programador sabe que a concatenação é uma operação frequente com sequências. Tutoriais introdutórios de Python explicam o uso de `+` e `*` para tal propósito, mas há detalhes sutis em seu funcionamento, como veremos a seguir.

2.8. Usando `+` e `*` com sequências

Em Python, podemos assumir que sequências suportem `+` e `*`. Em geral, os dois operandos de `+` devem ser sequências do mesmo tipo, e nenhum deles é modificado: uma nova sequência daquele mesmo tipo é criada como resultado da concatenação.

Para concatenar múltiplas cópias da mesma sequência, multiplique por um inteiro. Da mesma forma, uma nova sequência é criada:

```
>>> l = [1, 2, 3]  
>>> l * 5  
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]  
>>> 5 * 'abcd'  
'abcdabcdabcdabcdabcd'
```

Tanto `+` quanto `*` sempre criam um novo objeto, e nunca modificam seus operandos.



Tenha cuidado com expressões como `a * n` quando `a` é uma sequência contendo itens mutáveis, pois o resultado pode ser surpreendente. Por exemplo, tentar inicializar uma lista de listas como `my_list = [[]] * 3` vai resultar em uma lista com três referências para a mesma lista interna, que provavelmente não é o desejado.

A próxima seção fala das armadilhas ao se tentar usar `*` para inicializar uma lista de listas.

2.8.1. Criando uma lista de listas

Algumas vezes precisamos inicializar uma lista com um certo número de listas aninhadas—para, por exemplo, distribuir estudantes em uma lista de equipes, ou para representar casas no tabuleiro de um jogo. A melhor forma de fazer isso é com uma compreensão de lista, como no Exemplo 16.

Exemplo 16. Uma lista com três listas de tamanho 3 pode representar um tabuleiro de jogo da velha

```
>>> board = [['_'] * 3 for i in range(3)] ①  
>>> board  
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
```

```
>>> board[1][2] = 'X' ②
>>> board
[['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]
```

① Cria uma lista de três listas, cada uma com três itens. Inspeciona a estrutura criada.

② Coloca um "X" na linha 1, coluna 2, e verifica o resultado.

Um atalho tentador mas errado seria fazer algo como o Exemplo 17.

Exemplo 17. Uma lista com três referências para a mesma lista é inútil

```
>>> weird_board = [['_'] * 3] * 3 ①
>>> weird_board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> weird_board[1][2] = '0' ②
>>> weird_board
[['_', '_', '0'], ['_', '_', '0'], ['_', '_', '0']]
```

① A lista externa é feita de três referências para a mesma lista interna. Enquanto ela não é modificada, tudo parece correr bem.

② Colocar um "O" na linha 1, coluna 2, revela que todas as linhas são apelidos do mesmo objeto.

O problema com o Exemplo 17 é que ele se comporta, essencialmente, como o código abaixo:

```
row = '_' * 3
board = []
for i in range(3):
    board.append(row) ①
```

① A mesma `row` é anexada três vezes ao `board`.

Por outro lado, a compreensão de lista no Exemplo 16 equivale ao seguinte código:

```
>>> board = []
>>> for i in range(3):
...     row = '_' * 3 ①
...     board.append(row)
...
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[2][0] = 'X'
>>> board ②
[['_', '_', '_'], ['_', '_', '_'], ['X', '_', '_']]
```

① Cada iteração cria uma nova `row` e a acrescenta ao `board`.

② Como esperado, apenas a linha 2 é modificada.



Se o problema ou a solução mostrados nessa seção não estão claros para você, não se preocupe. Escrevi o Capítulo 6 para esclarecer a mecânica e os perigos das referências e dos objetos mutáveis.

Até aqui discutimos o uso dos operadores simples `+` e `*` com sequências, mas existem também os operadores `+=` e `*=`, que produzem resultados muito diferentes, dependendo da mutabilidade da sequência alvo. A próxima seção explica como eles funcionam.

2.8.2. Atribuição aumentada com sequências

Os operadores de atribuição aumentada `+=` e `*=` se comportam de formas muito diferentes, dependendo do primeiro operando. Para simplificar a discussão, vamos primeiro nos concentrar na adição aumentada (`+=`), mas os conceitos se aplicam a `*=` e a outros operadores de atribuição aumentada.

O método especial que faz `\+=` funcionar chama-se `__iadd__`, que significa *in-place addition* (adição no mesmo lugar).

Entretanto, se `__iadd__` não estiver implementado, Python usa `__add__` como antes de fazer a atribuição. Considere essa expressão simples:

```
>>> a += b
```

Se a implementar `__iadd__`, esse método será chamado. No caso de sequências mutáveis (por exemplo, `list`, `bytearray`, `array.array`), o objeto a será modificado no lugar (mesmo resultado de `a.extend(b)`). Porém, quando a não implementa `__iadd__`, a expressão `a += b` tem o mesmo efeito de `a = a + b`: a expressão `a + b` é avaliada antes, produzindo um novo objeto, que então é vinculado a a. Em outras palavras, a identidade do objeto vinculado à variável a pode ou não mudar, dependendo da existência de `__iadd__`.

Em geral, para sequências mutáveis, é razoável supor que `__iadd__` está implementado e que `+=` acontece *in-place*. Para sequências imutáveis, obviamente isso não pode acontecer.

O que acabei de escrever sobre `+=` também se aplica a `*=`, que é implementado via `__imul__`. Os métodos especiais `__iadd__` e `__imul__` são tratados no [ch_op_overload].

Veja uma demonstração de `*=` com uma sequência mutável e depois com uma sequência imutável:

```
>>> l = [1, 2, 3]
>>> id(l)
4311953800 ①
>>> l *= 2
>>> l
[1, 2, 3, 1, 2, 3]
>>> id(l)
4311953800 ②
>>> t = (1, 2, 3)
>>> id(t)
```

```
4312681568 ③
>>> t *= 2
>>> id(t)
4301348296 ④
```

- ① O ID da lista inicial.
- ② Após a multiplicação, a lista é o mesmo objeto, com novos itens anexados.
- ③ O ID da tupla inicial.
- ④ Após a multiplicação, uma nova tupla foi criada.

A concatenação repetida de sequências imutáveis é ineficiente, pois ao invés de apenas acrescentar novos itens, o interpretador tem que copiar toda a sequência alvo para criar um novo objeto com os novos itens concatenados.^[10]

Vimos casos de uso comuns para `+=`. A próxima seção mostra um caso surpreendente, que demonstra o real significado de "imutável" no contexto das tuplas.

2.8.3. Um quebra-cabeça com a atribuição `+=`

Tente responder sem usar o console: qual o resultado da avaliação das duas expressões no Exemplo 18?^[11]

Exemplo 18. Um enigma

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
```

O que acontece a seguir? Escolha a melhor alternativa:

1. `t` se torna `(1, 2, [30, 40, 50, 60])`.
2. É gerado um `TypeError` com a mensagem '`tuple`' object does not support item assignment (*o objeto tupla não suporta atribuição de itens*).
3. As alternativas A e B estão corretas.
4. Nenhuma das alternativas acima.

Quando vi isso, tinha certeza que a resposta era B, mas, na verdade é C: as alternativas A e B estão corretas!

O Exemplo 19 é a saída real em um console rodando Python 3.10.^[12]

Exemplo 19. O resultado inesperado: o item `t[2]` é modificado e uma exceção é gerada

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t
```

(1, 2, [30, 40, 50, 60])

O Online Python Tutor [<https://fpy.li/2-14>] (EN) é uma ferramenta online fantástica para visualizar em detalhes o funcionamento de Python. A Figura 7 é uma composição de duas capturas de tela, mostrando os estados inicial e final da tupla t do Exemplo 19.

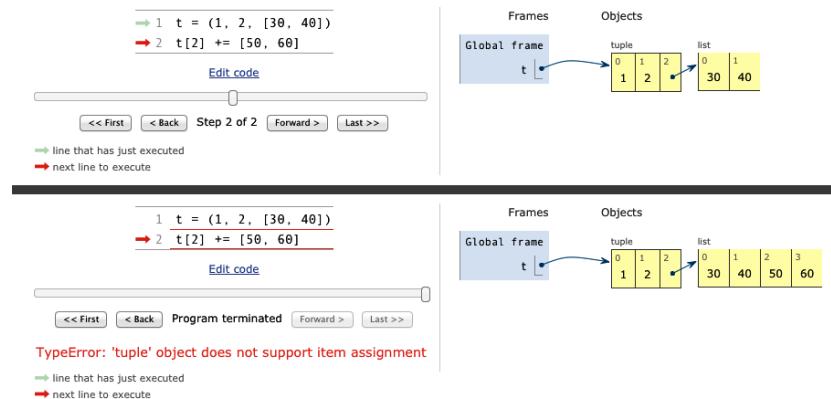


Figura 7. Estados inicial e final do enigma da atribuição de tuplas (diagrama gerado pelo Online Python Tutor).

Se olharmos o bytecode gerado pelo Python para a expressão `s[a] += b` (Exemplo 20), fica claro como isso acontece.

Exemplo 20. Bytecode para a expressão `s[a] += b`

```
>>> dis.dis('s[a] += b')
 1      0 LOAD_NAME                  0 (s)
      3 LOAD_NAME                  1 (a)
      6 DUP_TOP_TWO
      7 BINARY_SUBSCR               ①
      8 LOAD_NAME                  2 (b)
     11 INPLACE_ADD
     12 ROT_THREE
     13 STORE_SUBSCR               ②
     14 LOAD_CONST                 0 (None)
     17 RETURN_VALUE
```

① Coloca o valor de `s[a]` no TOS (*Top Of Stack*, topo da pilha).

② Executa `TOS += b`. Isso é bem sucedido se TOS se refere a um objeto mutável (no Exemplo 19 é uma lista).

③ Atribui `s[a] = TOS`. Isso falha se s é imutável (a tupla t no Exemplo 19).

Esse exemplo é um caso raro—nunca vi esse comportamento bizarro estragar o dia de alguém em mais de 20 anos trabalhando com Python.

Há três lições para tirar daqui:

- Evite colocar objetos mutáveis em tuplas.
- A atribuição aumentada não é uma operação atômica—acabamos de vê-la gerar uma exceção

após executar parte de seu trabalho.

- Inspecionar o bytecode de Python não é muito difícil, e pode ajudar a ver o que está acontecendo por debaixo dos panos.

Depois de ver as sutilezas do uso de `+` e `*` para concatenação, podemos mudar de assunto e tratar de outra operação essencial com sequências: ordenação.

2.9. `list.sort` versus a função embutida `sorted`

O método `list.sort` ordena uma lista no mesmo lugar—isto é, sem criar uma cópia. Ele devolve `None` para nos lembrar que muda a própria instância e não cria uma nova lista. Essa é uma convenção importante da API de Python: funções e métodos que mudam um objeto no mesmo lugar deve devolver `None`, para deixar claro a quem chamou que o receptor^[13] foi modificado, e que nenhum objeto novo foi criado. Um comportamento similar pode ser observado, por exemplo, na função `random.shuffle(s)`, que devolve `None` após embaralhar os itens de uma sequência mutável *in-place* (no lugar), isto é, mudando a posição dos itens dentro da própria sequência.



A convenção de devolver `None` para sinalizar mudanças no mesmo lugar tem uma desvantagem: não podemos encadear chamadas a esses métodos. Em contraste, métodos que devolvem novos objetos (todos os métodos de `str`, por exemplo) podem ser cascadeados no estilo de uma interface fluente. Veja o artigo "Fluent interface" [<https://fpy.li/2-15>] (EN) da Wikipedia em inglês para uma descrição mais detalhada deste tópico.

A função embutida `sorted`, por outro lado, cria e devolve uma nova lista. Ela aceita qualquer objeto iterável como um argumento, incluindo sequências imutáveis e geradores (veja o [ch_generators]). Independente do tipo do iterável passado a `sorted`, ela sempre cria e devolve uma nova lista.

Tanto `list.sort` quanto `sorted` podem receber dois argumentos nomeados opcionais:

reverse

Se `True`, os itens são devolvidos em ordem decrescente (isto é, invertendo a comparação dos itens). O default é `False`.

key

Uma função de um argumento que será aplicada a cada item, para produzir sua chave de ordenação. Por exemplo, ao ordenar uma lista de strings, `key=str.lower` pode ser usada para realizar uma ordenação sem levar em conta maiúsculas e minúsculas, e `key=len` irá ordenar as strings pela quantidade de caracteres. O default é a função identidade (isto é, os itens propriamente ditos são comparados).



Também se pode usar o parâmetro de palavra-chave opcional `key` com as funções embutidas `min()` e `max()`, e com outras funções da biblioteca padrão (por exemplo, `itertools.groupby()` e `heapq.nlargest()`).

Aqui estão alguns exemplos para esclarecer o uso dessas funções e dos argumentos nomeados. Os exemplos também demonstram que o algoritmo de ordenação de Python é estável (isto é, ele

preserva a ordem relativa de itens que resultam iguais na comparação):^[14]

```
>>> fruits = ['grape', 'raspberry', 'apple', 'banana']
>>> sorted(fruits)
['apple', 'banana', 'grape', 'raspberry'] ①
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ②
>>> sorted(fruits, reverse=True)
['raspberry', 'grape', 'banana', 'apple'] ③
>>> sorted(fruits, key=len)
['grape', 'apple', 'banana', 'raspberry'] ④
>>> sorted(fruits, key=len, reverse=True)
['raspberry', 'banana', 'grape', 'apple'] ⑤
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ⑥
>>> fruits.sort()                      ⑦
>>> fruits
['apple', 'banana', 'grape', 'raspberry'] ⑧
```

① Isso produz uma lista de strings ordenadas alfabeticamente.^[15]

② Inspecionando a lista original, vemos que ela não mudou.

③ Isso é a ordenação "alfabética" anterior, invertida.

④ Uma nova lista de strings, agora ordenada por tamanho. Como o algoritmo de ordenação é estável, "grape" e "apple," ambas com tamanho 5, estão em sua ordem original.

⑤ Essas são strings ordenadas por tamanho em ordem descendente. Não é o inverso do resultado anterior porque a ordenação é estável e então, novamente, "grape" aparece antes de "apple."

⑥ Até aqui, a ordenação da lista `fruits` original não mudou.

⑦ Isso ordena a lista no mesmo lugar, devolvendo `None` (que o console omite).

⑧ Agora `fruits` está ordenada.



Por default, Python ordena as strings lexicograficamente por código de caractere. Isso quer dizer que as letras maiúsculas ASCII virão antes das minúsculas, e caracteres não-ASCII dificilmente serão ordenados de forma razoável. A Seção 4.8 trata de maneiras corretas de ordenar texto da forma esperada por seres humanos.

Uma vez ordenadas, podemos realizar buscas em nossas sequências de forma muito eficiente. Um algoritmo de busca binária já é fornecido no módulo `bisect` da biblioteca padrão de Python. Aquele módulo também inclui a função `bisect.insort`, que você pode usar para garantir que suas sequências ordenadas permaneçam ordenadas. Há uma introdução ilustrada ao módulo `bisect` no post "Managing Ordered Sequences with Bisect" (*Gerenciando Sequências Ordenadas com Bisect*) [<https://fpy.li/bisect>] (EN) no site que complementa este livro.

Muito do que vimos até aqui neste capítulo se aplica a sequências em geral, não apenas a listas ou tuplas. Programadores Python às vezes usam excessivamente o tipo `list`, por ele ser tão conveniente—eu mesmo já fiz isso. Por exemplo, se você está processando grandes listas de

números, deveria considerar usar arrays em vez de listas. O restante do capítulo é dedicado a alternativas a listas e tuplas.

2.10. Quando uma lista não é a resposta

O tipo `list` é flexível e fácil de usar mas, dependendo dos requerimentos específicos, há opções melhores. Por exemplo, um `array` economiza muita memória se você precisa manipular milhões de valores de ponto flutuante. Por outro lado, se você está constantemente acrescentando e removendo itens das pontas opostas de uma lista, é bom saber que um `deque` (uma fila com duas pontas) é uma estrutura de dados FIFO^[16] mais eficiente.



Se seu código frequentemente verifica se um item está presente em uma coleção (por exemplo, `item in my_collection`), considere usar um `set` para `my_collection`, especialmente se ela contiver um número grande de itens. Um `set` é otimizado para verificação rápida de presença de itens. Eles também são iteráveis, mas não são sequências, porque a ordenação de itens de conjuntos não é especificada. Vamos falar deles no Capítulo 3.

O restante desse capítulo discute tipos mutáveis de sequências que, em muitos casos, podem substituir as listas. Começamos pelos arrays.

2.10.1. Arrays

Se uma lista contém apenas números, um `array.array` é um substituto mais eficiente. Arrays suportam todas as operações das sequências mutáveis (incluindo `.pop`, `.insert`, e `.extend`), bem como métodos adicionais para carregamento e armazenamento rápidos, como `.frombytes` e `.tofile`.

Um array de Python é quase tão enxuto quanto um array do C. Como mostrado na Figura 3, um array de valores `float` não contém objetos da classe `float`, mas apenas os bytes representando seus valores em código de máquina—como um array do tipo `double` na linguagem C. Ao criar um array, você fornece um código de tipo (*typecode*), uma letra que determina o tipo na linguagem C usado para armazenar cada item na memória. Por exemplo, `b` é o código de tipo para o que o C chama de `signed char`, um inteiro variando de -128 a 127. Se você criar uma `array('b')`, então cada item será armazenado em um único byte e será interpretado como um inteiro. Para grandes sequências de números, isso economiza muita memória. E Python não permite que você insira qualquer número que não corresponda ao tipo do array.

O Exemplo 21 mostra a criação, o armazenamento e o carregamento de um array de 10 milhões de números de ponto flutuante aleatórios.

Exemplo 21. Criando, armazenando e carregando uma grande array de números de ponto flutuante.

```
>>> from array import array ①
>>> from random import random
>>> floats = array('d', (random() for i in range(10**7))) ②
>>> floats[-1] ③
0.07802343889111107
>>> fp = open('floats.bin', 'wb')
>>> floats.tofile(fp) ④
```

```

>>> fp.close()
>>> floats2 = array('d') ⑤
>>> fp = open('floats.bin', 'rb')
>>> floats2.fromfile(fp, 10**7) ⑥
>>> fp.close()
>>> floats2[-1] ⑦
0.07802343889111107
>>> floats2 == floats ⑧
True

```

- ① Importa o tipo `array`.
- ② Cria um array de números de ponto flutuante de dupla precisão (código de tipo '`d`') a partir de qualquer objeto iterável—nesse caso, uma expressão geradora.
- ③ Inspeciona o último número no array.
- ④ Salva o array em um arquivo binário.
- ⑤ Cria um array vazio de números de ponto flutuante de dupla precisão
- ⑥ Lê 10 milhões de números do arquivo binário.
- ⑦ Inspeciona o último número no array.
- ⑧ Verifica a igualdade do conteúdo dos arrays

Como você pode ver, `array.tofile` e `array.fromfile` são fáceis de usar. Se você rodar o exemplo, verá que são também muito rápidos. Um pequeno experimento mostra que `array.fromfile` demora aproximadamente 0,1 segundos para carregar 10 milhões de números de ponto flutuante de dupla precisão de um arquivo binário criado com `array.tofile`. Isso é quase 60 vezes mais rápido que ler os números de um arquivo de texto, algo que também exige passar cada linha para a função embutida `float`. Salvar o arquivo com `array.tofile` é umas sete vezes mais rápido que escrever um número de ponto flutuante por vez em um arquivo de texto. Além disso, o tamanho do arquivo binário com 10 milhões de números de dupla precisão é de 80.000.000 bytes (8 bytes por número, nenhum byte a mais), enquanto o arquivo de texto ocupa 181.515.739 bytes para os mesmos dados.

Para o caso específico de arrays numéricas representando dados binários, tal como bitmaps de imagens, Python tem os tipos `bytes` e `bytearray`, discutidos na Capítulo 4.

Vamos encerrar essa seção sobre arrays com a Tabela 5, comparando as características de `list` e `array.array`.

Tabela 5. Métodos e atributos encontrados em `list` ou `array` (os métodos descontinuados de `array` e aqueles implementados também por `object` foram omitidos para preservar espaço)

	<code>list</code>	<code>array</code>	
<code>s.__add__(s2)</code>	•	•	<code>s + s2</code> —concatenação
<code>s.__iadd__(s2)</code>	•	•	<code>s += s2</code> —concatenação no mesmo lugar
<code>s.append(e)</code>	•	•	Acrescenta um elemento após o último
<code>s.byteswap()</code>		•	Permuta os bytes de todos os itens do array para conversão de <i>endianness</i> (ordem de interpretação bytes)

	list	array	
s.clear()	•		Apaga todos os itens
s.__contains__(e)	•	•	e in s
s.copy()	•		Cópia rasa da lista
s.__copy__()		•	Suporte a copy.copy
s.count(e)	•	•	Conta as ocorrências de um elemento
s.__deepcopy__(...)		•	Suporte otimizado a copy.deepcopy
s.__delitem__(p)	•	•	Remove o item na posição p
s.extend(it)	•	•	Acrescenta itens a partir do iterável it
s.frombytes(b)		•	Acrescenta itens de uma sequência de bytes, interpretada como valores compactos de máquina
s.fromfile(f, n)		•	Acrescenta n itens de um arquivo binário f, interpretado como valores compactos de máquina
s.fromlist(l)		•	Acrescenta itens de lista; se um deles causar um TypeError, nenhum item é acrescentado
s.__getitem__(p)	•	•	s[p]—obtém o item ou fatia na posição
s.index(e)	•	•	Encontra a posição da primeira ocorrência de e
s.insert(p, e)	•	•	Insere elemento e antes do item na posição p
s.itemsize		•	Tamanho em bytes de cada item do array
s.__iter__()	•	•	Obtém iterador
s.__len__()	•	•	len(s)—número de itens
s.__mul__(n)	•	•	s * n—concatenação repetida
s.__imul__(n)	•	•	s *= n—concatenação repetida no mesmo lugar
s.__rmul__(n)	•	•	n * s—concatenação repetida invertida ^[17]
s.pop([p])	•	•	Remove e devolve o item na posição p (default: o último)
s.remove(e)	•		Remove o primeiro elemento de valor igual a e
s.reverse()	•	•	Reverte a ordem dos itens no mesmo lugar
s.__reversed__()	•		Obtém iterador para percorrer itens do último até o primeiro
s.__setitem__(p, e)	•	•	s[p] = e—coloca e na posição p, sobrescrevendo item ou fatia existente
s.sort([key], [reverse])	•		Ordena itens no mesmo lugar, com os argumentos nomeados opcionais key e reverse
s.tobytes()		•	Devolve itens como valores de máquina compactos em um objeto bytes

	list	array	
s.tofile(f)		<ul style="list-style-type: none"> • Grava itens como valores de máquina compactos no arquivo binário f 	
s.tolist()		<ul style="list-style-type: none"> • Devolve os itens como objetos numéricos em uma list 	
s.typecode		<ul style="list-style-type: none"> • String de um caractere identificando o tipo em C dos itens 	

Até Python 3.10, o tipo array ainda não tem um método sort equivalente a list.sort(), que reordena os elementos na própria estrutura de dados, sem copiá-la. Se você precisa ordenar um array, use a função embutida sorted para reconstruir o array:



```
a = array.array(a.typecode, sorted(a))
```

Para manter a ordem de um array ordenado ao acrescentar novos itens, use a função bisect.insort [<https://fpy.li/2t>].

Se você trabalha muito com arrays e não conhece memoryview, pode estar desperdiçando memória e CPU. Veja o próximo tópico.

2.10.2. Views de memória

A classe embutida memoryview é um tipo sequência de memória compartilhada, que permite manipular fatias de arrays sem copiar bytes. Ela foi inspirada pela biblioteca NumPy (que discutiremos brevemente, na Seção 2.10.3). Travis Oliphant, autor principal da NumPy, responde assim à questão "When should a memoryview be used?" *Quando se deve usar uma memoryview?* [<https://fpy.li/2-17>]:

Uma memoryview é essencialmente uma estrutura de array Numpy generalizada dentro do próprio Python (sem a matemática). Ela permite compartilhar memória entre estruturas de dados (coisas como imagens PIL, bancos de dados SQLite, arrays da NumPy, etc.) sem copiar bytes. Isso é muito importante ao lidar com grandes conjuntos de dados.

Usando uma notação similar ao módulo array, o método memoryview.cast permite mudar a forma como múltiplos bytes são lidos ou escritos como unidades, sem a necessidade de alterar os bytes. memoryview.cast devolve um novo objeto memoryview, sempre compartilhando a mesma memória.

O Exemplo 22 mostra como criar views alternativas da mesmo array de 6 bytes, para operar com ele como uma matriz de 2x3 ou de 3x2.

Exemplo 22. Manipular 6 bytes de memória como views de 1×6, 2×3, e 3×2

```
>>> from array import array
>>> octets = array('B', range(6)) ①
```

```

>>> m1 = memoryview(octets) ②
>>> m1.tolist()
[0, 1, 2, 3, 4, 5]
>>> m2 = m1.cast('B', [2, 3]) ③
>>> m2.tolist()
[[0, 1, 2], [3, 4, 5]]
>>> m3 = m1.cast('B', [3, 2]) ④
>>> m3.tolist()
[[0, 1], [2, 3], [4, 5]]
>>> m2[1,1] = 22 ⑤
>>> m3[1,1] = 33 ⑥
>>> octets ⑦
array('B', [0, 1, 2, 33, 22, 5])

```

- ① Cria um array de 6 bytes (código de tipo 'B').
- ② Cria uma `memoryview` a partir daquele array, e a exporta como uma lista.
- ③ Cria uma nova `memoryview` a partir da anterior, mas com 2 linhas e 3 colunas.
- ④ Ainda outra `memoryview`, agora com 3 linhas e 2 colunas.
- ⑤ Sobrescreve o byte em `m2`, na linha 1, coluna 1 com 22.
- ⑥ Sobrescreve o byte em `m3`, na linha 1, coluna 1 com 33.
- ⑦ Mostra o array original, provando que a memória era compartilhada entre `octets`, `m1`, `m2`, e `m3`.

O fantástico poder de `memoryview` também pode ser usado para o mal. O Exemplo 23 mostra como mudar um único byte de um item em um array de inteiros de 16 bits.

Exemplo 23. Mudando o valor de um item em um array de inteiros de 16 bits trocando apenas o valor de um de seus bytes

```

>>> numbers = array.array('h', [-2, -1, 0, 1, 2])
>>> memv = memoryview(numbers) ①
>>> len(memv)
5
>>> memv[0] ②
-2
>>> memv_oct = memv.cast('B') ③
>>> memv_oct.tolist() ④
[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]
>>> memv_oct[5] = 4 ⑤
>>> numbers
array('h', [-2, -1, 1024, 1, 2]) ⑥

```

- ① Cria uma `memoryview` a partir de um array de 5 inteiros de 16 bits com sinal (tipo 'h').
- ② `memv` vê os mesmos 5 itens no array.
- ③ Cria `memv_oct`, transformando os elementos de `memv` em bytes (tipo 'B').
- ④ Exporta os elementos de `memv_oct` como uma lista de 10 bytes, para inspeção.

- ⑤ Atribui o valor 4 ao byte com offset 5.
- ⑥ Observe a mudança em `numbers`: um 4 no byte mais significativo de um inteiro de 2 bytes sem sinal é 1024.



Você pode ver um exemplo de inspeção de uma `memoryview` com o pacote `struct` em "Parsing binary records with `struct`" *Analizando registros binários com struct* [<https://fpy.li/2-18>] (EN).

Enquanto isso, se você está fazendo processamento numérico avançado com arrays, deveria estar usando as bibliotecas NumPy. Vamos agora fazer um breve passeio por elas.

2.10.3. NumPy

Nesse livro eu priorizo o que já existe na biblioteca padrão de Python, para que você a aproveite ao máximo. Mas a NumPy é tão importante que exige um desvio.

Graças a suas operações avançadas com arrays e matrizes, o Numpy permitiu que Python se tornasse uma das principais linguagens para aplicações de computação científica. A Numpy implementa tipos multidimensionais e homogêneos de arrays e matrizes, que podem conter não apenas números, mas também registros definidos pelo usuário. E fornece operações eficientes ao nível desses elementos.

A SciPy é uma biblioteca criada usando a NumPy, e oferece inúmeros algoritmos de computação científica, incluindo álgebra linear, cálculo numérico e estatística. A SciPy é rápida e confiável porque usa a popular base de código C e Fortran do Repositório Netlib [<https://fpy.li/2-19>]. Em outras palavras, a SciPy dá a cientistas o melhor de dois mundos: um prompt iterativo e as APIs de alto nível de Python, junto com funções estáveis e comprovadamente eficientes para processamento numérico, otimizadas em C e Fortran.

O Exemplo 24, uma demonstração muito rápida da Numpy, demonstra algumas operações básicas com arrays bi-dimensionais.

Exemplo 24. Operações básicas com linhas e colunas em uma `numpy.ndarray`

```
>>> import numpy as np ①
>>> a = np.arange(12) ②
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.shape ③
(12,)
>>> a.shape = 3, 4 ④
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a[2] ⑤
array([ 8,  9, 10, 11])
```

```

>>> a[2, 1] ⑥
9
>>> a[:, 1] ⑦
array([1, 5, 9])
>>> a.transpose() ⑧
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])

```

- ① Importa a NumPy, que precisa ser instalada previamente (ela não faz parte da biblioteca padrão de Python). Por convenção, `numpy` é importada como `np`.
- ② Cria e inspeciona um `numpy.ndarray` com inteiros de 0 a 11.
- ③ Inspeciona as dimensões do array: esse é um array com uma dimensão e 12 elementos.
- ④ Muda o formato do array, acrescentando uma dimensão e depois inspecionando o resultado.
- ⑤ Obtém a linha no índice 2
- ⑥ Obtém elemento na posição 2, 1.
- ⑦ Obtém a coluna no índice 1
- ⑧ Cria um novo array por transposição (permutando as colunas com as linhas)

A NumPy também suporta operações de alto nível para carregar, salvar e operar sobre todos os elementos de um `numpy.ndarray`:

```

>>> import numpy
>>> floats = numpy.loadtxt('floats-10M-lines.txt') ①
>>> floats[-3:] ②
array([ 3016362.69195522,   535281.10514262,   4566560.44373946])
>>> floats *= .5 ③
>>> floats[-3:]
array([ 1508181.34597761,   267640.55257131,   2283280.22186973])
>>> from time import perf_counter as pc ④
>>> t0 = pc(); floats /= 3; pc() - t0 ⑤
0.03690556302899495
>>> numpy.savetxt('floats-10M', floats) ⑥
>>> floats2 = numpy.load('floats-10M.npy', 'r+') ⑦
>>> floats2 *= 6
>>> floats2[-3:] ⑧
memmap([ 3016362.69195522,   535281.10514262,   4566560.44373946])

```

- ① Carrega 10 milhões de números de ponto flutuante de um arquivo de texto.
- ② Usa a notação de fatiamento de sequência para inspecionar os três últimos números.
- ③ Multiplica cada elemento no array `floats` por .5 e inspeciona novamente os três últimos elementos.
- ④ Importa o cronômetro de medida de tempo em alta resolução (disponível desde Python 3.3).

- ⑤ Divide cada elemento por 3; o tempo decorrido para dividir os 10 milhões de números de ponto flutuante é menos de 40 milissegundos.
- ⑥ Salva o array em um arquivo binário `.npy`.
- ⑦ Carrega os dados como um arquivo mapeado na memória em outro array; isso permite o processamento eficiente de fatias do array, mesmo que ele não caiba inteiro na memória.
- ⑧ Inspeciona os três últimos elementos após multiplicar cada elemento por 6.

Isso foi apenas um aperitivo.

A NumPy e a SciPy são bibliotecas formidáveis, e estão na base de outras ferramentas fantásticas, como a Pandas [<https://fpy.li/2-20>] (EN)—que implementa tipos eficientes de arrays capazes de conter dados não-numéricos, e fornece funções de importação/exportação em vários formatos diferentes, como `.csv`, `.xls`, `dumps` SQL, HDF5, etc.—e a scikit-learn [<https://fpy.li/2-21>] (EN), o conjunto de ferramentas para Aprendizagem de Máquina mais usado atualmente. A maior parte das funções da NumPy e da SciPy são implementadas em C ou C++, e conseguem aproveitar todos os núcleos de CPU disponíveis, pois podem liberar a GIL (Global Interpreter Lock, *Trava Global do Interpretador*) de Python. O projeto Dask [<https://fpy.li/dask>] suporta a paralelização do processamento da NumPy, da Pandas e da scikit-learn para grupos (*clusters*) de máquinas. Esses pacotes merecem livros inteiros. Este não é um desses livros, mas nenhuma revisão das sequências de Python estaria completa sem pelo menos uma breve passagem pelos arrays da NumPy.

Tendo visto as sequências planas—arrays padrão e arrays da NumPy—vamos agora nos voltar para um grupo completamente diferente de substitutos para a boa e velha `list`: filas (*queues*).

2.10.4. Deques e outras filas

Os métodos `.append` e `.pop` tornam uma `list` usável como uma pilha (*stack*) ou uma fila (*queue*) (usando `.append` e `.pop(0)`, se obtém o comportamento FIFO de uma fila). Mas inserir e remover da cabeça de uma lista (a posição com índice 0) é caro, pois a lista toda precisa ser deslocada na memória.

A classe `collections.deque` é uma fila *thread-safe* (segura para usar com threads), otimizada para inserção e remoção rápida nas duas pontas. É também a estrutura preferencial se você precisa manter uma lista de "últimos itens vistos" ou coisa semelhante, pois um deque pode ser delimitado—isto é, criado com um tamanho máximo fixo. Se um deque delimitado está cheio, quando se adiciona um novo item, o item na ponta oposta é descartado. O Exemplo 25 mostra algumas das operações típicas com um deque.

Exemplo 25. Usando um deque

```
>>> from collections import deque
>>> dq = deque(range(10), maxlen=10) ①
>>> dq
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.rotate(3) ②
>>> dq
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
>>> dq.rotate(-4)
```

```

>>> dq
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
>>> dq.appendleft(-1) ③
>>> dq
deque([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.extend([11, 22, 33]) ④
>>> dq
deque([3, 4, 5, 6, 7, 8, 9, 11, 22, 33], maxlen=10)
>>> dq.extendleft([10, 20, 30, 40]) ⑤
>>> dq
deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)

```

- ① O argumento opcional `maxlen` determina o número máximo de itens permitidos nessa instância de `deque`; isso estabelece o valor de um atributo de instância `maxlen`, somente de leitura.
- ② Rotacionar com $n > 0$ retira itens da direita e os recoloca pela esquerda; quando $n < 0$, os itens são retirados pela esquerda e anexados pela direita.
- ③ Acrescentar itens a um `deque` cheio (`len(d) == d maxlen`) elimina itens da ponta oposta. Na linha seguinte, note que o `0` foi descartado.
- ④ Acrescentar três itens à direita derruba `-1`, `1`, e `2` da extremidade esquerda.
- ⑤ Observe que `extendleft(iter)` acrescenta cada item sucessivo do argumento `iter` do lado esquerdo do `deque`, então a posição final dos itens é invertida.

A Tabela 6 compara os métodos específicos de `list` e `deque` (omitindo aqueles que também aparecem em `object`).

Veja que `deque` implementa a maioria dos métodos de `list`, acrescentando alguns específicos ao seu modelo, como `popleft` e `rotate`. Mas há um custo oculto: remover itens do meio de um `deque` não é rápido. A estrutura é realmente otimizada para acréscimos e remoções pelas pontas.

As operações `append` e `popleft` são atômicas, então `deque` pode ser usado de forma segura como uma fila FIFO em aplicações multithread sem a necessidade de travas.

Tabela 6. Métodos implementados em list ou deque (aqueles também implementados por object foram omitidos para preservar espaço)

	<code>list</code>	<code>deque</code>	
<code>s.__add__(s2)</code>	•		<code>s + s2</code> —concatenação
<code>s.__iadd__(s2)</code>	•	•	<code>s += s2</code> —concatenação no mesmo lugar
<code>s.append(e)</code>	•	•	Acrescenta um elemento à direita (após o último)
<code>s.appendleft(e)</code>		•	Acrescenta um elemento à esquerda (antes do primeiro)
<code>s.clear()</code>	•	•	Apaga todos os itens
<code>s.__contains__(e)</code>	•		<code>e in s</code>
<code>s.copy()</code>	•		Cópia rasa da lista
<code>s.__copy__()</code>		•	Suporte a <code>copy.copy</code> (cópia rasa)

	list	deque	
s.count(e)	•	•	Conta ocorrências de um elemento
s.__delitem__(p)	•	•	Remove item na posição p
s.extend(i)	•	•	Acrescenta item do iterável i pela direita
s.extendleft(i)		•	Acrescenta item do iterável i pela esquerda
s.__getitem__(p)	•	•	s[p]—obtém item ou fatia na posição
s.index(e)	•		Encontra a primeira ocorrência de e
s.insert(p, e)	•		Insere elemento e antes do item na posição p
s.__iter__()	•	•	Obtém iterador
s.__len__()	•	•	len(s)—número de itens
s.__mul__(n)	•		s * n—concatenação repetida
s.__imul__(n)	•		s *= n—concatenação repetida no mesmo lugar
s.__rmul__(n)	•		`n * s`—concatenação repetida invertida ^[18]
s.pop()	•	•	Remove e devolve último item ^[19]
s.popleft()		•	Remove e devolve primeiro item
s.remove(e)	•		Remove o primeiro elemento de valor igual a e
s.reverse()	•	•	Inverte a ordem do itens no mesmo lugar
s.__reversed__()	•	•	Obtém iterador para percorrer itens, do último para o primeiro
s.rotate(n)		•	Move n itens de um lado para o outro
s.__setitem__(p, e)	•	•	s[p] = e—coloca e na posição p, sobrescrevendo item ou fatia existentes
s.sort([key], [reverse])	•		Ordena os itens no mesmo lugar, com os argumentos nomeados opcionais key e reverse

Além de deque, outros pacotes da biblioteca padrão de Python implementam filas:

queue

Fornece as classes sincronizadas (isto é, seguras para se usar com múltiplas threads) SimpleQueue, Queue, LifoQueue, e PriorityQueue. Essas classes podem ser usadas para comunicação segura entre threads. Todas, exceto SimpleQueue, podem ser delimitadas passando um argumento maxsize maior que 0 ao construtor. Entretanto, elas não descartam um item para abrir espaço, como faz deque. Em vez disso, quando a fila está lotada, a inserção de um novo item bloqueia quem tentou inserir—isto é, ela espera até alguma outra thread criar espaço retirando um item da fila, algo útil para limitar o número de threads ativas.

multiprocessing

Implementa sua própria SimpleQueue, não-delimitada, e Queue, delimitada, muito similares àquelas no pacote queue, mas projetadas para comunicação entre processos. Uma fila especializada, JoinableQueue, serve para gerenciamento de tarefas.

asyncio

Fornece Queue, LifoQueue, PriorityQueue, e JoinableQueue com APIs inspiradas pelas classes nos módulos queue e multiprocessing, mas adaptadas para gerenciar tarefas em programação assíncrona.

heapq

Diferente do últimos três módulos, heapq não implementa a classe queue, mas oferece funções como heappush e heappop, que permitem o uso de uma sequência mutável como uma fila do tipo heap ou como uma fila de prioridade.

Aqui termina nossa revisão das alternativas ao tipo list, e também nossa exploração dos tipos sequência em geral—exceto pelas especificidades de str e das sequências binárias, que tem seu próprio capítulo (Capítulo 4).

2.11. Resumo do capítulo

Dominar o uso dos tipos sequência da biblioteca padrão é um pré-requisito para escrever código Python conciso, eficiente e idiomático.

As sequências de Python são geralmente categorizadas como mutáveis ou imutáveis, mas também é útil considerar um outro eixo: sequências planas e sequências contêiner. As primeiras são mais compactas, mais rápidas e mais fáceis de usar, mas estão limitadas a armazenar dados atômicos como números, caracteres e bytes. As sequências contêiner são mais flexíveis, mas podem surpreender quando contêm objetos mutáveis. Então, quando armazenando estruturas de dados aninhadas, é preciso ter cuidado para usar tais sequências da forma correta.

Infelizmente Python não tem um tipo de sequência contêiner imutável infalível: mesmo as tuplas "imutáveis" podem ter seus valores modificados quando contêm itens mutáveis como listas ou objetos definidos pelo usuário.

Compreensões de lista e expressões geradoras são notações poderosas para criar e inicializar sequências. Se você ainda não se sente confortável com essas técnicas, gaste o tempo necessário para aprender seu uso básico. Não é difícil, e você logo vai estar gostando delas.

As tuplas no Python tem dois papéis: como registros de campos sem nome e como listas imutáveis. Ao usar uma tupla como uma lista imutável, lembre-se que só é garantido que o valor de uma tupla será fixo se todos os seus itens também forem imutáveis. Chamar hash(t) com a tupla como argumento é uma forma rápida de se assegurar que seu valor é fixo. Se t contiver itens mutáveis, um TypeError é gerado.

Quando uma tupla é usada como registro, o desempacotamento de tuplas é a forma mais segura e legível de extrair seus campos. Além das tuplas, * funciona com listas e iteráveis em vários contextos, e alguns de seus casos de uso apareceram no Python 3.5 com a PEP 448—Additional Unpacking Generalizations (*Generalizações de Desempacotamento Adicionais*) [<https://fpy.li/pep448>] (EN). Python 3.10 introduziu o casamento de padrões com match/case, suportando um tipo de desempacotamento mais poderoso, conhecido como desestruturação.

O fatiamento de sequências é um dos recursos de sintaxe preferidos de Python, e é ainda mais poderoso do que muita gente pensa. Fatiamento multidimensional e a notação de reticências (...),

como usados no NumPy, podem também ser suportados por sequências definidas pelo usuário. Atribuir a fatias é uma forma muito expressiva de editar sequências mutáveis.

Concatenação repetida, como em `seq * n`, é conveniente e, tomado cuidado, pode ser usada para inicializar listas de listas contendo itens imutáveis. Atribuição aumentada com `+=` e `*=` se comporta de forma diferente com sequências mutáveis e imutáveis. No último caso, esses operadores necessariamente criam novas sequências. Mas se a sequência alvo é mutável, ela em geral é modificada no lugar—mas nem sempre, depende de como a sequência é implementada.

O método `sort` e a função embutida `sorted` são fáceis de usar e flexíveis, graças ao argumento opcional `key`: uma função para calcular o critério de ordenação. E aliás, `key` também pode ser usado com as funções embutidas `min` e `max`.

Além de listas e tuplas, a biblioteca padrão de Python oferece `array.array`. Apesar da NumPy e da SciPy não serem parte da biblioteca padrão, se você faz qualquer tipo de processamento numérico em grandes conjuntos de dados, estudar mesmo uma pequena parte dessas bibliotecas pode levar você muito longe.

Terminamos com uma visita à versátil `collections.deque`, que é segura para usar com threads. Comparamos sua API com a de `list` na Tabela 6 e mencionamos as outras implementações de filas na biblioteca padrão.

2.12. Leitura complementar

O capítulo 1, "Data Structures" (*Estruturas de Dados*) do Python Cookbook [https://fpy.li/pycook3] (EN) (O'Reilly), de David Beazley e Brian K. Jones, traz muitas receitas usando sequências, incluindo a "Recipe 1.11. Naming a Slice" (*Receita 1.11. Nomeando uma Fatia*), onde aprendi o truque de atribuir fatias a variáveis para melhorar a legibilidade, como ilustrado no nosso Exemplo 15.

A segunda edição do *Python Cookbook* foi escrita para Python 2.4, mas a maior parte de seu código funciona com Python 3, e muitas das receitas dos capítulos 5 e 6 lidam com sequências. O livro foi editado por Alex Martelli, Anna Ravenscroft, e David Ascher, e inclui contribuições de dúzias de pythonistas. A terceira edição foi reescrita do zero, e se concentra mais na semântica da linguagem—especialmente no que mudou no Python 3—enquanto o volume mais antigo enfatiza a pragmática (isto é, como aplicar a linguagem a problemas da vida real). Apesar de algumas das soluções da segunda edição não serem mais a melhor abordagem, honestamente acho que vale a pena ter à mão as duas edições do *Python Cookbook*.

O "HowTo - Ordenação" [https://fpy.li/2v] oficial de Python tem vários exemplos de técnicas avançadas de uso de `sorted` e `list.sort`.

A PEP 3132—Extended Iterable Unpacking (*Desempacotamento Estendido de Iterável*) [https://fpy.li/2-2] (EN) é a fonte canônica para ler sobre o novo uso da sintaxe `*extra` no lado esquerdo de atribuições paralelas. Se você quiser dar uma olhada no processo de evolução de Python, "Missing *-unpacking generalizations" (*As generalizações esquecidas de * no desempacotamento*) [https://fpy.li/2-24] (EN) é um tópico do bug tracker propondo melhorias na notação de desempacotamento iterável. PEP 448—Additional Unpacking Generalizations (*Generalizações de Desempacotamento Adicionais*) [https://fpy.li/pep448] (EN) foi o resultado de discussões ocorridas naquele tópico.

Como mencionei na Seção 2.6, o texto introdutório "Casamento de padrão estrutural" [<https://fpy.li/2r>], de Carol Willing, no "O que há de novo no Python 3.10" [<https://fpy.li/2s>], é uma ótima introdução a esse novo grande recurso, em mais ou menos 1.400 palavras (isso é menos de 5 páginas quando o Firefox converte o HTML em PDF). PEP 636—Structural Pattern Matching: Tutorial (*Casamento de Padrões Estrutural: Tutorial*) [<https://fpy.li/pep636>] (EN) também é bom, porém mais longo. A mesma PEP 636 inclui o "Appendix A—Quick Intro" (*Apêndice A-Introdução Rápida*) [<https://fpy.li/2-27>] (EN). Ele é menor que a introdução de Willing, porque omite as considerações gerais sobre os motivos pelos quais o casamento de padrões é útil. Se você precisar de mais argumentos para se convencer ou convencer outros que o casamento de padrões foi bom para o Python, leia as 22 páginas de PEP 635—Structural Pattern Matching: Motivation and Rationale (*Casamento de Padrões Estrutural: Motivação e Justificativa*) [<https://fpy.li/pep635>] (EN).

O post de Eli Bendersky em seu blog, "Less copies in Python with the buffer protocol and memoryviews" (*Menos cópias em Python, com o protocolo de buffer e memoryviews*) [<https://fpy.li/2-28>] inclui um pequeno tutorial sobre `memoryview`.

Há muitos livros tratando da NumPy no mercado, e muitos não mencionam "NumPy" no título. Dois exemplos são o *Python Data Science Handbook* [<https://fpy.li/2-29>], escrito por Jake VanderPlas e de acesso aberto, e a segunda edição do *Python for Data Analysis* [<https://fpy.li/2-30>], de Wes McKinney.

"A Numpy é toda sobre vetorização". Essa é a frase de abertura do livro de acesso aberto *From Python to NumPy* [<https://fpy.li/2-31>], de Nicolas P. Rougier. Operações vetorizadas aplicam funções matemáticas a todos os elementos de um array sem um loop explícito escrito em Python. Elas podem operar em paralelo, usando instruções especiais de vetor presentes em CPUs modernas, tirando proveito de múltiplos núcleos ou delegando para a GPU, dependendo da biblioteca. O primeiro exemplo no livro de Rougier mostra um aumento de velocidade de 500 vezes, após a refatoração de uma bela classe pythônica, usando um método gerador, em uma pequena e feroz função que chama um par de funções de vetor da NumPy.

Para aprender a usar `deque` (e outras coleções), veja os exemplos e as receitas práticas em "Tipos de dados de contêineres" [<https://fpy.li/2w>], na documentação de Python.

A melhor defesa da convenção de Python de excluir o último item `range` e fatias foi escrita pelo grande Edsger W. Dijkstra, em uma nota curta intitulada "Why Numbering Should Start at Zero" (*Porque a Numeração Deve Começar em Zero*) [<https://fpy.li/2-32>]. O assunto da nota é notação matemática, mas ela é relevante para Python porque Dijkstra explica, com humor e rigor, porque uma sequência como 2, 3, ..., 12 deveria sempre ser expressa como $2 \leq i < 13$. Todas as outras convenções razoáveis são refutadas, bem como a ideia de deixar cada usuário escolher uma convenção. O título se refere à indexação baseada em zero, mas a nota na verdade é sobre porque é desejável que '`ABCDE[1:3]`' signifique '`BC`' e não '`CD`', e porque faz todo sentido escrever `range(2, 13)` para produzir 2, 3, 4, ..., 12. E, por sinal, a nota foi escrita à mão, mas é linda e totalmente legível. A letra de Dijkstra é tão cristalina que alguém criou uma fonte [<https://fpy.li/2-33>] a partir de suas anotações.

Ponto de Vista

A natureza das tuplas

Em 2012, apresentei um poster sobre a linguagem ABC na PyCon US. Antes de criar Python,

Guido van Rossum tinha trabalhado no interpretador ABC, então ele veio ver meu pôster. Entre outras coisas, falamos sobre como os *compounds* (compostos) da ABC, predecessores das tuplas de Python. Compostos também suportam atribuição paralela e são usados como chaves compostas em dicionários (ou *tabelas*, no jargão da ABC). Entretanto, compostos não são sequências, Eles não são iteráveis, e não é possível obter um campo por índice, muitos menos fatiá-los. Ou você manuseia o composto inteiro ou extrai os campos individuais usando atribuição paralela, e é isso.

Disse a Guido que essas limitações tornavam muito claro o principal propósito dos compostos: ele são apenas registros sem campos nomeados. Sua resposta: "Fazer as tuplas se comportarem como sequências foi uma gambiarra."

Isso ilustra a abordagem pragmática que tornou Python mais prático e mais bem sucedido que a ABC. Da perspectiva de um implementador de linguagens, fazer as tuplas se comportarem como sequências custa pouco. Como resultado, o principal caso de uso de tuplas como registros não é tão óbvio, mas ganhamos listas imutáveis—mesmo que seu tipo não seja tão claramente nomeado como seria `frozenlist`.

Sequências planas versus sequências contêineres

Para realçar os diferentes modelos de memória dos tipos de sequências usei os termos *sequência contêiner* e *sequência plana*. A palavra "contêiner" vem da própria documentação do "Modelo de Dados" [<https://fpy.li/2x>]:

Alguns objetos contêm referências a outros objetos; eles são chamados de contêineres.

Usei o termo "sequência contêiner" para ser específico, porque existem contêineres em Python que não são sequências, como `dict` e `set`. Sequências contêineres podem ser aninhadas porque elas podem conter objetos de qualquer tipo, incluindo seu próprio tipo.

Por outro lado, *sequências planas* são tipos de sequências que não podem ser aninhadas, pois só podem conter valores atômicos como inteiros, números de ponto flutuante ou caracteres.

Adotei o termo *sequência plana* porque precisava de algo para contrastar com "sequência contêiner."

Apesar do uso anterior da palavra "containers" na documentação oficial, há uma classe abstrata em `collections.abc` chamada `Container`. Aquela ABC tem apenas um método, `__contains__`—o método especial por trás do operador `in`. Isso significa que arrays e strings, que não são contêineres no sentido tradicional, são subclasses virtuais de `Container`, porque implementam `__contains__`. Isso é só mais um exemplo de humanos usando uma mesma palavra para significar coisas diferentes. Nesse livro, vou escrever "contêiner" com minúscula e em português para "um objeto que contém referências para outros objetos" e `Container` com a inicial maiúscula em fonte mono espaçada para me referir a `collections.abc.Container`.

Listas bagunçadas

Textos introdutórios de Python costumam enfatizar que listas podem conter objetos de

diferentes tipos, mas na prática esse recurso não é muito útil: colocamos itens em uma lista para processá-los mais tarde, o que implica o suporte, da parte de todos os itens, a pelo menos alguma operação em comum (isto é, eles devem todos "grasnar", independente de serem ou não 100% patos, geneticamente falando). Por exemplo, não é possível ordenar uma lista em Python 3 a menos que os itens ali contidos sejam comparáveis:

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

Diferente das listas, as tuplas muitas vezes mantêm itens de tipos diferentes. Isso é natural: se cada item em uma tupla é um campo, então cada campo pode ter um tipo diferente.

'key' é brilhante

O argumento opcional `key` de `list.sort`, `sorted`, `max`, e `min` é uma grande ideia. Outras linguagens forçam você a fornecer uma função de comparação com dois argumentos, como a função descontinuada de Python 2 `cmp(a, b)`. Usar `key` é mais simples e mais eficiente. É mais simples porque basta definir uma função de um único argumento que recupera ou calcula o critério a ser usado para ordenar seus objetos; isso é mais fácil que escrever uma função de dois argumentos para devolver -1, 0, 1. Também é mais eficiente, porque a função `key` é invocada apenas uma vez por item, enquanto a comparação de dois argumentos é chamada a cada vez que o algoritmo de ordenação precisa comparar dois itens. Claro, Python também precisa comparar as chaves ao ordenar, mas aquela comparação é feita em código C otimizado, não em uma função Python escrita por você.

Por sinal, usando `key` podemos ordenar uma lista bagunçada de números e strings "parecidas com números". Só precisamos decidir se queremos tratar todos os itens como inteiros ou como strings:

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l, key=int)
[0, '1', 5, 6, '9', 14, 19, '23', 28, '28']
>>> sorted(l, key=str)
[0, '1', 14, 19, '23', 28, '28', 5, 6, '9']
```

A Oracle, o Google, e a Conspiração Timbot

O algoritmo de ordenação usado em `sorted` e `list.sort` é o Timsort, um algoritmo adaptativo que troca de estratégia de ordenação (entre *merge sort* e *insertion sort*), dependendo de quanto ordenados os dados já estão. Isso é eficiente porque dados reais tendem a ter séries de itens ordenados. Há um artigo da Wikipedia [<https://fpy.li/2y>] sobre ele.

O Timsort foi usado no CPython pela primeira vez em 2002. Desde 2009, o Timsort também é usado para ordenar arrays tanto em Java padrão quanto no Android, um fato que ficou muito conhecido quando a Oracle usou parte do código relacionado ao Timsort como evidência da

violação da propriedade intelectual da Sun pelo Google. Por exemplo, veja essa ordem do Juiz William Alsup [<https://fpy.li/2-36>] (EN) de 2012. Em 2021, a Suprema Corte dos Estados Unidos decidiu que o uso do código de Java pelo Google é "fair use"^[20]

O Timsort foi inventado por Tim Peters, um dos desenvolvedores principais de Python, e tão produtivo que se acredita que ele seja uma inteligência artificial, o Timbot. Você pode ler mais sobre essa teoria da conspiração em "Python Humor" [<https://fpy.li/2-37>] (EN). Tim também escreveu "The Zen of Python": `import this`.

[1] Agradeço à leitora Tina Lapine por apontar essa informação.

[2] Operadores invertidos são explicados no [ch_op_overload].

[3] Também usado para sobreescriver uma sub-sequência. Veja a Seção 2.7.4.

[4] Agradeço ao revisor técnico Leonardo Rochael por esse exemplo.

[5] NT: A tradução em português da documentação de Python adotou o termo "casamento de padrões" no lugar de *pattern matching*. O termo em inglês é usado nas comunidades brasileiras de linguagens que implementam *pattern matching* há muitos anos, como por exemplo Scala, Elixir e Haskell. Naturalmente mantivemos os títulos originais nos links externos.

[6] Na minha opinião, uma sucessão `if/elif/elif/…/else` funciona muito bem no lugar de `switch/case`. E ela não sofre dos problemas de `fallthrough` (*cascadeamento*) [<https://fpy.li/2-8>] (EN) e de `dangling else` (*o else errante*) [<https://fpy.li/2-9>] (EN), que alguns projetistas de linguagens copiaram irracionalmente do C—décadas após sabermos que tais problemas causam inúmeros bugs.

[7] A última é chamada `eval` no código original; a renomeei para evitar confusão com a função embutida `eval` de Python.

[8] Na Seção 2.10.2 vamos mostrar que `views` da memória construídas de forma especial podem ter mais de uma dimensão.

[9] Não, eu não escrevi ao contrário: o nome da classe `ellipsis` realmente se escreve só com minúsculas, e a instância é um objeto embutido chamado `Ellipsis`, da mesma forma que `bool` é em minúsculas mas suas instâncias são `True` e `False`.

[10] `str` é uma exceção a essa descrição. Como criar strings com `+=` em loops é tão comum em bases de código reais, o CPython foi otimizado para esse caso de uso. Instâncias de `str` são alocadas na memória com espaço extra, então a concatenação não exige a cópia da string inteira a cada operação.

[11] Agradeço a Leonardo Rochael e Cesar Kawakami por compartilharem esse enigma na Conferência PythonBrasil de 2013.

[12] Alguns leitores sugeriram que a operação no exemplo pode ser realizada com `t[2].extend([50, 60])`, sem erros. Eu sei disso, mas a intenção aqui é mostrar o comportamento estranho do operador `+=` nesse caso.

[13] Receptor (`receiver`) é o alvo de uma chamada a um método, o objeto vinculado a `self` no corpo do método.

[14] O principal algoritmo de ordenação de Python se chama Timsort, em homenagem a seu criador, Tim Peters. Para curiosidades sobre o Timsort, veja o Ponto de Vista.

[15] As palavras nesse exemplo estão ordenadas em ordem alfabética porque são 100% constituídas de caracteres ASCII em letras minúsculas. Veja o aviso após o exemplo.

[16] Sigla em inglês para "First in, first out" (primeiro a entrar, primeiro a sair), o comportamento padrão de filas.

[17] Operadores invertidos são explicados no [ch_op_overload].

[18] Operadores invertidos são explicados no [ch_op_overload].

[19] `a_list.pop(p)` permite remover da posição `p`, mas `deque` não suporta essa opção.

[20] NT: Conceito da lei de copyright norte-americana que permite, em determinadas circunstâncias, o uso sem autorização prévia de partes da propriedade intelectual de outros. Em geral traduzido como "uso razoável" ou "uso aceitável". Essa doutrina não faz parte da lei brasileira.

3. Dicionários e conjuntos

Python é feito basicamente de dicionários cobertos por muitas camadas de açúcar sintático

— Lalo Martins, pioneiro do nomadismo digital e pythonista

Usamos dicionários em todos os nossos programas Python. Se não diretamente em nosso código, então indiretamente, pois o tipo `dict` é um elemento fundamental da implementação de Python. Atributos de classes e de instâncias, espaços de nomes de módulos e argumentos nomeados de funções são alguns dos elementos fundamentais de Python representados na memória por dicionários. O `_builtins__._dict_` armazena todos os tipos, funções e objetos embutidos.

Por seu papel crucial, os dict de Python são extremamente otimizados—e continuam recebendo melhorias. As *tabelas de hash* são o motor por trás do alto desempenho dos dict de Python.

Outros tipos embutidos baseados em tabelas de hash são `set` e `frozenset`. Eles oferecem uma API mais completa e operadores mais robustos que os conjuntos que você pode ter encontrado em outras linguagens populares. Em especial, os conjuntos de Python implementam todas as operações fundamentais da teoria dos conjuntos, como união, intersecção, testes de subconjuntos, etc. Com eles, podemos expressar algoritmos de forma mais declarativa, evitando o excesso de loops e condicionais aninhados.

Aqui está um breve esquema do capítulo:

- A sintaxe moderna para criar e manipular dict e mapeamentos, incluindo desempacotamento aumentado e *pattern matching* (casamento de padrões)
- Métodos comuns dos tipos de mapeamentos
- Tratamento especial para chaves ausentes
- Variantes de `dict` na biblioteca padrão
- Os tipos `set` e `frozenset`
- As implicações das tabelas de hash no comportamento de conjuntos e dicionários.

3.1. Novidades nesse capítulo

A maior parte das mudanças nessa segunda edição se concentra em novos recursos relacionados a tipos de mapeamento:

- A Seção 3.2 fala da sintaxe aperfeiçoada de desempacotamento e de diferentes maneiras de mesclar mapeamentos—incluindo os operadores `|` e `|=`, suportados pelos dict desde Python 3.9.
- A Seção 3.3 ilustra o manuseio de mapeamentos com `match/case`, recurso que surgiu no Python 3.10.
- A Seção 3.6.1 agora se concentra nas pequenas mas ainda relevantes diferenças entre `dict` e `OrderedDict`—levando em conta que, desde Python 3.6, `dict` passou a manter a ordem de

inserção das chaves.

- Novas seções sobre os objetos view devolvidos por `dict.keys`, `dict.items`, e `dict.values`: a Seção 3.8 e a Seção 3.12.

A implementação interna de `dict` e `set` ainda está alicerçada em tabelas de hash, mas o código de `dict` teve duas otimizações importantes, que economizam memória e preservam a ordem de inserção das chaves. A Seção 3.9 e a Seção 3.11 resumem o que você precisa saber sobre isso para usar bem essas estruturas.

Após acrescentar mais de 200 páginas a essa segunda edição, transferi a seção opcional "Internals of sets and dicts" (*As entranhas dos sets e dos dicts*) [<https://fpy.li/hashint>] (EN) para o <http://fluentpython.com>, o site que complementa o livro. O post de 18 páginas [<https://fpy.li/hashint>] (EN) foi atualizado e expandido, e inclui explicações e diagramas sobre:



- O algoritmo de tabela de hash e as estruturas de dados, começando por seu uso em `set`, que é mais simples de entender.
- A otimização de memória que preserva a ordem de inserção de chaves em instâncias de `dict` (desde Python 3.6).
- O layout do compartilhamento de chaves em dicionários que mantêm atributos de instância—o `__dict__` de objetos definidos pelo usuário (otimização implementada no Python 3.3).

3.2. A sintaxe moderna dos dicts

As próximas seções descrevem os recursos avançados de sintaxe para criação, desempacotamento e processamento de mapeamentos. Alguns desses recursos não são novos na linguagem, mas podem ser novidade para você. Outros requerem Python 3.9 (como o operador `|`) ou Python 3.10 (como `match/case`). Vamos começar por um dos melhores e mais antigos desses recursos.

3.2.1. Compreensões de dict

Desde Python 2.7, a sintaxe das `listcomps` e `genexps` foi adaptada para compreensões de `dict` (e também compreensões de `set`, que veremos em breve). Uma *dictcomp* (compreensão de `dict`) cria uma instância de `dict`, recebendo pares `key:value` de qualquer iterável. O Exemplo 26 mostra o uso de compreensões de `dict` para criar dois dicionários a partir de uma mesma lista de tuplas.

Exemplo 26. Exemplos de compreensões de dict

```
>>> dial_codes = [①
...     (880, 'Bangladesh'),
...     (55, 'Brazil'),
...     (86, 'China'),
...     (91, 'India'),
...     (62, 'Indonesia'),
...     (81, 'Japan'),
...     (234, 'Nigeria'),
```

```

...     (92, 'Pakistan'),
...     (7, 'Russia'),
...     (1, 'United States'),
... ]
>>> country_dial = {country: code for code, country in dial_codes} ②
>>> country_dial
{'Bangladesh': 880, 'Brazil': 55, 'China': 86, 'India': 91, 'Indonesia': 62,
'Japan': 81, 'Nigeria': 234, 'Pakistan': 92, 'Russia': 7, 'United States': 1}
>>> {code: country.upper() ③
...     for country, code in sorted(country_dial.items())
...     if code < 70}
{55: 'BRAZIL', 62: 'INDONESIA', 7: 'RUSSIA', 1: 'UNITED STATES'}

```

① Um iterável de pares chave-valor como `dial_codes` pode ser passado diretamente para o construtor de `dict`, mas...

② ...aqui permutamos os pares: `country` é a chave, e `code` é o valor.

③ Ordenando `country_dial` por nome, revertendo novamente os pares, colocando os valores em maiúsculas e filtrando os itens com `code < 70`.

Se você já usa `listcomps`, as `dictcomps` são um próximo passo natural. Caso contrário, a propagação da sintaxe de compreensão mostra que agora é mais valioso que nunca se tornar fluente nessa técnica.

3.2.2. Desempacotando mapeamentos

A PEP 448—Additional Unpacking Generalizations (*Generalizações de Desempacotamento Adicionais*) [<https://fpy.li/pep448>] melhorou o suporte ao desempacotamento de mapeamentos de duas formas, desde Python 3.5.

Primeiro, podemos aplicar `**` a mais de um argumento em uma chamada de função. Isso funciona quando todas as chaves são strings e únicas, para todos os argumentos (porque argumentos nomeados duplicados são proibidos):

```

>>> def dump(**kwargs):
...     return kwargs
...
>>> dump(**{'x': 1}, y=2, **{'z': 3})
{'x': 1, 'y': 2, 'z': 3}

```

Em segundo lugar, `**` pode ser usado dentro de um literal `dict`—também múltiplas vezes:

```

>>> {'a': 0, **{'x': 1}, 'y': 2, **{'z': 3, 'x': 4}}
{'a': 0, 'x': 4, 'y': 2, 'z': 3}

```

Nesse caso, chaves duplicadas são permitidas. Cada ocorrência sobrescreve ocorrências anteriores—observe o valor mapeado para `x` no exemplo.

Essa sintaxe também pode ser usada para mesclar mapas, mas isso pode ser feito de outras formas. Siga comigo.

3.2.3. Fundindo mapeamentos com |

Desde a versão 3.9, Python suporta o uso de | e |= para mesclar mapeamentos. Isso faz todo sentido, já que estes são também os operadores de união de conjuntos.

O operador | cria um novo mapeamento:

```
>>> d1 = {'a': 1, 'b': 3}
>>> d2 = {'a': 2, 'b': 4, 'c': 6}
>>> d1 | d2
{'a': 2, 'b': 4, 'c': 6}
```

O tipo do novo mapeamento normalmente será o mesmo do operando da esquerda—no exemplo, d1—but ele pode ser do tipo do segundo operando se tipos definidos pelo usuário estiverem envolvidos na operação, dependendo das regras de sobrecarga de operadores, que exploraremos no [ch_op_overload].

Para atualizar mapeamentos existentes no mesmo lugar, use |=. Retomando o exemplo anterior, ali d1 não foi modificado. Mas aqui sim:

```
>>> d1
{'a': 1, 'b': 3}
>>> d1 |= d2
>>> d1
{'a': 2, 'b': 4, 'c': 6}
```



Se você precisa manter código rodando no Python 3.8 ou anterior, a seção "Motivation" (*Motivação*) [<https://fpy.li/3-1>] (EN) da PEP 584—Add Union Operators To dict (*Acrescentar Operadores de União a dict*) [<https://fpy.li/pep584>] (EN) inclui um bom resumo das outras formas de mesclar mapeamentos.

Agora vamos ver como o pattern matching se aplica aos mapeamentos.

3.3. Pattern matching com mapeamentos

A instrução `match/case` suporta sujeitos que sejam objetos mapeamento. Padrões para mapeamentos se parecem com literais `dict`, mas podem casar com instâncias de qualquer subclasse real ou virtual de `collections.abc.Mapping`.^[1]

No Capítulo 2 nos concentrarmos apenas nos padrões de sequência, mas tipos diferentes de padrões podem ser combinados e aninhados. Graças à desestruturação, o pattern matching é uma ferramenta poderosa para processar registros estruturados como sequências e mapeamentos aninhados, que frequentemente precisamos ler de APIs JSON ou bancos de dados que suportam registros ou campos semi-estruturados, como o MongoDB, o EdgeDB, ou o PostgreSQL. O Exemplo

27 demonstra isso.

As dicas de tipo simples em `get_creators` tornam claro que ela recebe um dict e devolve uma list.

Exemplo 27. creator.py: get_creators() extrai o nome dos criadores em registros de mídia

```
def get_creators(record: dict) -> list:
    match record:
        case {'type': 'book', 'api': 2, 'authors': [*names]}: ①
            return names
        case {'type': 'book', 'api': 1, 'author': name}: ②
            return [name]
        case {'type': 'book'}: ③
            raise ValueError(f"Invalid 'book' record: {record!r}")
        case {'type': 'movie', 'director': name}: ④
            return [name]
        case _: ⑤
            raise ValueError(f'Invalid record: {record!r}')
```

① Casa com qualquer mapeamento na forma `'type': 'book'`, `'api' :2`, e uma chave `'authors'` mapeada para uma sequência. Devolve os itens da sequência, como uma nova list.

② Casa com qualquer mapeamento na forma `'type': 'book'`, `'api' :1`, e uma chave `'author'` mapeada para qualquer objeto. Devolve aquele objeto dentro de uma list.

③ Qualquer outro mapeamento na forma `'type': 'book'` é inválido e gera um `ValueError`.

④ Casa qualquer mapeamento na forma `'type': 'movie'` e uma chave `'director'` mapeada para um único objeto. Devolve o objeto dentro de uma list.

⑤ Qualquer outro sujeito é inválido e gera um `ValueError`.

O Exemplo 27 mostra algumas práticas úteis para lidar com dados semi-estruturados, como registros JSON:

- Incluir um campo descrevendo o tipo de registro (por exemplo, `'type': 'movie'`)
- Incluir um campo identificando a versão do schema (por exemplo, `'api': 2`), para permitir evoluções futuras das APIs públicas.
- Ter cláusulas `case` para processar registros inválidos de um tipo específico (por exemplo, `'book'`), bem como um `case` final para capturar tudo que tenha passado pelas condições anteriores.

Agora vamos ver como `get_creators` se comporta com alguns doctests concretos:

```
>>> b1 = dict(api=1, author='Douglas Hofstadter',
...             type='book', title='Gödel, Escher, Bach')
>>> get_creators(b1)
['Douglas Hofstadter']
>>> from collections import OrderedDict
>>> b2 = OrderedDict(api=2, type='book',
...                   title='Python in a Nutshell',
```

```

...         authors='Martelli Ravenscroft Holden'.split())
>>> get_creators(b2)
['Martelli', 'Ravenscroft', 'Holden']
>>> get_creators({'type': 'book', 'pages': 770})
Traceback (most recent call last):
...
ValueError: Invalid 'book' record: {'type': 'book', 'pages': 770}
>>> get_creators('Spam, spam, spam')
Traceback (most recent call last):
...
ValueError: Invalid record: 'Spam, spam, spam'

```

Observe que a ordem das chaves nos padrões é irrelevante, mesmo se o sujeito for um `OrderedDict` como `b2`.

Diferente de patterns de sequência, patterns de mapeamento funcionam com matches parciais. Nos doctests, os sujeitos `b1` e `b2` incluem uma chave `'title'`, que não aparece em nenhum padrão `'book'`, mas mesmo assim casam.

Não há necessidade de usar `**extra` para casar pares chave-valor adicionais, mas se você quiser capturá-los como um `dict`, pode prefixar uma variável com `**`. Ela precisa ser a última do padrão, e `**_` é proibido, pois seria redundante. Um exemplo simples:

```

>>> food = dict(category='ice cream', flavor='vanilla', cost=199)
>>> match food:
...     case {'category': 'ice cream', **details}:
...         print(f'Ice cream details: {details}')
...
Ice cream details: {'flavor': 'vanilla', 'cost': 199}

```

Na Seção 3.5, vamos estudar o `defaultdict` e outros mapeamentos onde buscas com chaves via `__getitem__` (isto é, `d[chave]`) funcionam porque itens ausentes são criados na hora. No contexto do pattern matching, um `match` é bem sucedido apenas se o sujeito já possui as chaves necessárias no início do bloco `match`.



O tratamento automático de chaves ausentes não é acionado porque o pattern matching sempre usa o método `d.get(key, sentinel)`—onde o sentinel `default` é um marcador com valor especial, que não pode aparecer nos dados do usuário.

Vistas a sintaxe e a estrutura, vamos estudar a API dos mapeamentos.

3.4. A API padrão dos tipos de mapeamentos

O módulo `collections.abc` contém as ABCs `Mapping` e `MutableMapping`, descrevendo as interfaces de `dict` e de tipos similares. Veja a Figura 8.

A maior utilidade dessas ABCs é documentar e formalizar as interfaces padrão para os mapeamentos, e servir e critério para testes com `isinstance` em código que precise suportar

mapeamentos de forma geral:

```
>>> my_dict = {}
>>> isinstance(my_dict, abc.Mapping)
True
>>> isinstance(my_dict, abc.MutableMapping)
True
```



Usar `isinstance` com uma ABC é muitas vezes melhor que verificar se um argumento de função é do tipo concreto `dict`, porque daí tipos alternativos de mapeamentos podem ser usados. Vamos discutir isso em detalhes no [ch_ifaces_prot_abc].

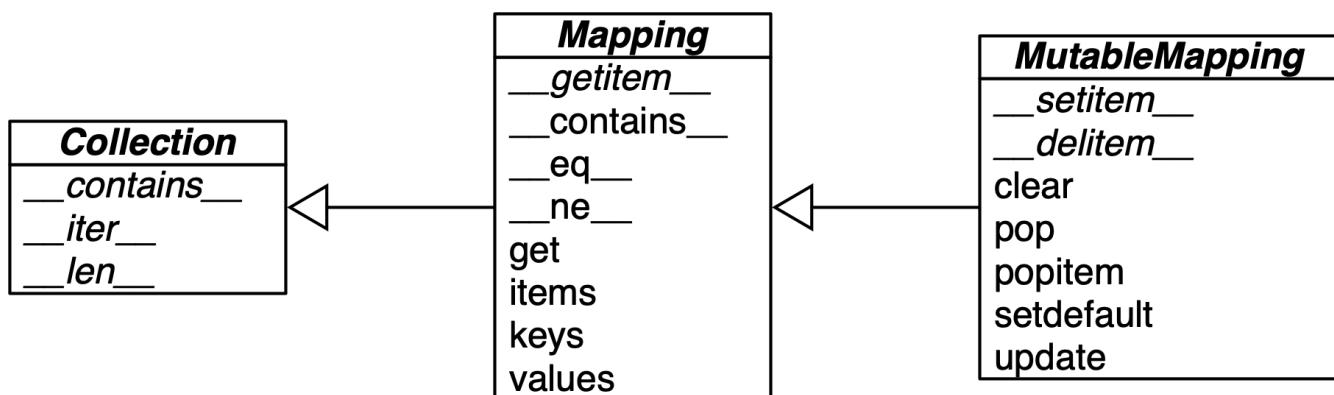


Figura 8. Diagrama de classe simplificado para `MutableMapping` e suas superclasses de `collections.abc` (as setas de herança apontam das subclasses para as superclasses; nomes em itálico indicam classes e métodos abstratos)

Para implementar uma mapeamento customizado, é mais fácil estender `collections.UserDict`, ou envolver um `dict` por composição, ao invés de criar uma subclasse dessas ABCs. A classe `collections.UserDict` e todas as classes concretas de mapeamentos da biblioteca padrão encapsulam o `dict` básico em suas implementações, que por sua vez é criado sobre uma tabela de hash. Assim, todas elas compartilham a mesma limitação: as chaves precisam ser *hashable* (os valores não precisam ser *hashable*, só as chaves). Se você precisa de uma recapitulação, a próxima seção explica isso.

3.4.1. O que é *hashable*?

Aqui está parte da definição de *hashable*, adaptado do *Glossário de Python* [<https://fpy.li/32>]:

Um objeto é *hashable* se tem um código de hash que nunca muda durante seu ciclo de vida (precisa ter um método `__hash__`) e pode ser comparado com outros objetos (precisa ter um método `__eq__`). Objetos *hashable* que são comparados como iguais devem ter o mesmo código de hash.^[2]

Tipos numéricos e os tipos planos imutáveis `str` e `bytes` são todos *hashable*. Tipos contêineres são *hashable* se forem imutáveis e se todos os objetos por eles contidos forem também *hashable*. Um `frozenset` é sempre *hashable*, pois todos os elementos que ele contém devem ser, por definição,

hashable. Uma tuple é hashable apenas se todos os seus itens também forem. Observe as tuplas tt, tl, and tf:

```
>>> tt = (1, 2, (30, 40))
>>> hash(tt)
8027212646858338501
>>> tl = (1, 2, [30, 40])
>>> hash(tl)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> tf = (1, 2, frozenset([30, 40]))
>>> hash(tf)
-4118419923444501110
```

O código de hash de um objeto pode ser diferente dependendo da versão de Python, da arquitetura da máquina, e pelo *sal* acrescentado ao cálculo do hash por razões de segurança.^[3] O código de hash de um objeto corretamente implementado tem a garantia de ser constante apenas dentro de um processo Python.

Tipos definidos pelo usuário são hashble por default, pois seu código de hash é seu id(), e o método __eq__() herdado da classe object apenas compara os IDs dos objetos. Se um objeto implementar seu próprio __eq__(), que leve em consideração seu estado interno, ele será hashable apenas se seu __hash__() sempre devolver o mesmo código de hash. Na prática, isso exige que __eq__() e __hash__() levem em conta apenas atributos de instância que nunca mudem durante a vida do objeto.

Vamos agora revisar a API dos tipos de mapeamento mais comumente usado no Python: dict, defaultdict, e OrderedDict.

3.4.2. Revisão dos métodos mais comuns dos mapeamentos

A API básica para mapeamentos é bem completa. A Tabela 7 mostra os métodos implementados por dict e pelas variantes mais usadas: defaultdict e OrderedDict, ambas classes definidas no módulo collections.

Tabela 7. Métodos do tipos de mapeamento dict, collections.defaultdict, e collections.OrderedDict (métodos comuns de object omitidos por concisão); argumentos opcionais então entre [...]

	dict	default dict	Ordered Dict	
d.clear()	•	•	•	Remove todos os itens.
d.__contains__(k)	•	•	•	k in d
d.copy()	•	•	•	Cópia rasa
d.__copy__()		•		Suporte a copy.copy(d).
d.default_factory		•		Chamável invocado por __missing__ para definir valores ausentes ^[4]

	dict	default dict	Ordered Dict	
d.__delitem__(k)	•	•	•	del d[k]—remove item com chave k
d.fromkeys(it, [initial])	•	•	•	Novo mapeamento com chaves no iterável it, com um valor inicial opcional (o default é None)
d.get(k, [default])	•	•	•	Obtém item com chave k, devolve default ou None se k não existir
d.__getitem__(k)	•	•	•	d[k]—obtém item com chave k
d.items()	•	•	•	Obtém uma <i>view</i> dos itens—pares (chave, valor)
d.__iter__()	•	•	•	Obtém iterador das chaves
d.keys()	•	•	•	Obtém <i>view</i> das chaves
d.__len__()	•	•	•	len(d)—número de itens
d.__missing__(k)		•		Chamado quando __getitem__ não consegue encontrar a chave
d.move_to_end(k, [last])			•	Move k para primeira ou última posição (last é True por default).
d.__or__(other)	•	•	•	Suporte a d1 d2 para criar um novo dict, fundindo d1 e d2 (Python ≥ 3.9)
d.__ior__(other)	•	•	•	Suporte a d1 = d2 para atualizar d1 com d2 (Python ≥ 3.9)
d.pop(k, [default])	•	•	•	Remove e devolve valor em k, ou default ou None, se k não existir
d.popitem()	•	•	•	Remove e devolve, na forma (chave, valor), o último item inserido ^[5]
d.__reversed__()	•	•	•	Suporte a reverse(d)—devolve um iterador de chaves, da última para a primeira a serem inseridas
d.__ror__(other)	•	•	•	Suporte a other dd—operador de união invertido (Python ≥ 3.9) ^[6]
d.setdefault(k, [default])	•	•	•	Se k in d, devolve d[k]; senão, atribui d[k] = default e devolve isso
d.__setitem__(k, v)	•	•	•	d[k] = v—coloca v em k
d.update(m, **kwargs)	•	•	•	Atualiza d com itens de um mapeamento ou iterável de pares (chave, valor)
d.values()	•	•	•	Obtém uma <i>view</i> dos valores

A forma como d.update(m) lida com seu primeiro argumento, m, é um excelente exemplo de *duck typing* (*tipagem pato*): ele primeiro verifica se m possui um método keys e, em caso afirmativo, assume que m é um mapeamento. Caso contrário, update() reverte para uma iteração sobre m,

presumindo que seus itens são pares (chave, valor). O construtor da maioria dos mapeamentos de Python usa internamente a lógica de update(), o que quer dizer que eles podem ser inicializados por outros mapeamentos ou a partir de qualquer objeto iterável que produza pares (chave, valor).

Um método útil dos mapeamentos é setdefault(). Ele evita buscas redundantes de chaves quando precisamos atualizar o valor em um item no mesmo lugar. A próxima seção mostra como ele pode ser usado.

3.4.3. Inserindo ou atualizando valores mutáveis

Alinhada à filosofia de *fallar rápido* de Python, a consulta a um dict com d[k] gera um erro quando k não é uma chave existente. Pythonistas sabem que d.get(k, default) é uma alternativa a d[k] quando receber um valor default é mais conveniente que tratar um KeyError. Entretanto, se você está buscando um valor mutável e quer atualizá-lo, há um jeito melhor.

Considere um script para indexar texto, produzindo um mapeamento no qual cada chave é uma palavra, e o valor é uma lista das posições onde aquela palavra ocorre, como mostrado no Exemplo 28.

Exemplo 28. Saída parcial do Exemplo 29 processando o texto "Zen of Python"; cada linha mostra uma palavra e uma lista de ocorrências na forma de pares (line_number, column_number) (número da linha, número da coluna).

```
$ python3 index0.py zen.txt
a [(19, 48), (20, 53)]
Although [(11, 1), (16, 1), (18, 1)]
ambiguity [(14, 16)]
and [(15, 23)]
are [(21, 12)]
aren [(10, 15)]
at [(16, 38)]
bad [(19, 50)]
be [(15, 14), (16, 27), (20, 50)]
beats [(11, 23)]
Beautiful [(3, 1)]
better [(3, 14), (4, 13), (5, 11), (6, 12), (7, 9), (8, 11), (17, 8), (18, 25)]
...
```

O Exemplo 29 é um script aquém do ideal, para mostrar um caso onde dict.get não é a melhor maneira de lidar com uma chave ausente. Ele foi adaptado de um exemplo de Alex Martelli.^[7]

Exemplo 29. index0.py usa dict.get para obter e atualizar uma lista de ocorrências de palavras de um índice (uma solução melhor é apresentada no Exemplo 30)

```
"""Build an index mapping word -> list of occurrences"""

import re
import sys

WORD_RE = re.compile(r'\w+')
```

```

index = []
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
            # this is ugly; coded like this to make a point
            occurrences = index.get(word, []) ①
            occurrences.append(location) ②
            index[word] = occurrences ③

# display in alphabetical order
for word in sorted(index, key=str.upper): ④
    print(word, index[word])

```

- ① Obtém a lista de ocorrências de word, ou [] se a palavra não for encontrada.
- ② Acrescenta uma nova localização a occurrences.
- ③ Coloca a occurrences modificada no dict index; isso exige uma segunda busca em index.
- ④ Não estou chamando str.upper no argumento key= de sorted, apenas passando uma referência àquele método, para que a função sorted possa usá-lo para normalizar as palavras antes de ordená-las.^[8]

As três linhas tratando de occurrences no Exemplo 29 podem ser substituídas por uma única linha usando dict.setdefault. O Exemplo 30 fica mais próximo do código apresentado por Alex Martelli.

Exemplo 30. index.py usa dict.setdefault para obter e atualizar uma lista de ocorrências de uma palavra em uma única linha de código; compare com o Exemplo 29

```

"""Build an index mapping word -> list of occurrences"""

import re
import sys

WORD_RE = re.compile(r'\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
            index.setdefault(word, []).append(location) ①

# display in alphabetical order
for word in sorted(index, key=str.upper):

```

```
print(word, index[word])
```

- ① Obtém a lista de ocorrências de `word`, ou a define como `[]`, se não for encontrada; `setdefault` devolve o valor, então ele pode ser atualizado sem uma segunda busca.

Em outras palavras, o resultado final desta linha...

```
my_dict.setdefault(key, []).append(new_value)
```

...é o mesmo que executar...

```
if key not in my_dict:  
    my_dict[key] = []  
my_dict[key].append(new_value)
```

...exceto que este último trecho de código executa pelo menos duas buscas por `key`—três se a chave não for encontrada—enquanto `setdefault` faz tudo isso com uma única busca.

Uma questão relacionada, o tratamento de chaves ausentes em qualquer busca (e não apenas para inserção de valores), é o assunto da próxima seção.

3.5. Tratamento automático de chaves ausentes

Algumas vezes é conveniente que os mapeamentos devolvam algum valor padronizado quando se busca por uma chave ausente. Há duas abordagem principais para esse fim: uma é usar um `defaultdict` em vez de um `dict` simples. A outra é criar uma subclasse de `dict` ou de qualquer outro tipo de mapeamento e acrescentar um método `__missing__`. Vamos ver as duas soluções a seguir.

3.5.1. defaultdict: outra perspectiva sobre as chaves ausentes

Uma instância de `collections.defaultdict` cria itens com um valor default sob demanda, sempre que uma chave ausente é buscada usando a sintaxe `d[k]`. O Exemplo 31 usa `defaultdict` para fornecer outra solução elegante para o índice de palavras do Exemplo 30.

Funciona assim: ao instanciar um `defaultdict`, você fornece um chamável que produz um valor default sempre que `__getitem__` recebe uma chave inexistente como argumento.

Por exemplo, dado um `defaultdict` criado por `dd = defaultdict(list)`, se 'new-key' não estiver em `dd`, a expressão `dd['new-key']` segue os seguintes passos:

1. Chama `list()` para criar uma nova lista.
2. Insere a lista em `dd` usando 'new-key' como chave.
3. Devolve uma referência para aquela lista.

O chamável que produz os valores default é mantido em um atributo de instância chamado `default_factory`.

Exemplo 31. index_default.py: usando um defaultdict em vez do método setdefault

```
"""Build an index mapping word -> list of occurrences"""

import collections
import re
import sys

WORD_RE = re.compile(r'\w+')

index = collections.defaultdict(list)      ①
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
            index[word].append(location) ②

# display in alphabetical order
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

① Cria um defaultdict com o construtor de list como default_factory.

② Se word não está inicialmente no index, o default_factory é chamado para produzir o valor ausente, que neste caso é uma list vazia, que então é atribuída a index[word] e devolvida, de forma que a operação .append(location) é sempre bem sucedida.

Se nenhum default_factory é fornecido, o KeyError usual é gerado para chaves ausentes.



O default_factory de um defaultdict só é invocado para fornecer valores default para chamadas a `__getitem__`, não para outros métodos. Por exemplo, se dd é um defaultdict e k uma chave ausente, dd[k] chamará default_factory para criar um valor default, mas dd.get(k) vai devolver None, e k in dd resulta False.

O mecanismo que faz defaultdict funcionar, chamando default_factory, é o método especial `__missing__`, apresentado a seguir.

3.5.2. O método `__missing__`

Por trás da forma como os mapeamentos lidam com chaves ausentes está o método muito apropriadamente chamado `__missing__`.^[9] Esse método não é definido na classe base dict, mas dict está ciente de sua possibilidade: se você criar uma subclasse de dict e incluir um método `__missing__`, o dict.`__getitem__` padrão vai chamar seu método sempre que uma chave não for encontrada, em vez de gerar um KeyError.

Suponha que você queira um mapeamento onde as chaves são convertidas para str quando são procuradas. Um caso de uso concreto seria uma biblioteca para dispositivos IoT (Internet of Things, *Internet das Coisas*)^[10], onde uma placa programável com portas genéricas programáveis (por

exemplo, uma Raspberry Pi ou uma Arduino) é representada por uma classe "Placa" com um atributo `minha_placa.portas`, que é uma mapeamento dos identificadores das portas físicas para objetos de software portas. O identificador da porta física pode ser um número ou uma string como "A0" ou "P9_12". Por consistência, é desejável que todas as chaves em `placa.portas` seja strings, mas também é conveniente buscar uma porta por número, como em `meu-arduino.porta[13]`, para evitar que iniciantes tropeçem quando quiserem fazer piscar o LED na porta 13 de seus Arduinos. O Exemplo 32 mostra como tal mapeamento funcionaria.

Exemplo 32. Ao buscar por uma chave não-string, StrKeyDict0 a converte para str quando ela não é encontrada

Tests for item retrieval using 'd[key]' notation::

```
>>> d = StrKeyDict0([('2', 'two'), ('4', 'four')])
>>> d['2']
'two'
>>> d[4]
'four'
>>> d[1]
Traceback (most recent call last):
...
KeyError: '1'
```

Tests for item retrieval using 'd.get(key)' notation::

```
>>> d.get('2')
'two'
>>> d.get(4)
'four'
>>> d.get(1, 'N/A')
'N/A'
```

Tests for the 'in' operator::

```
>>> 2 in d
True
>>> 1 in d
False
```

O Exemplo 33 implementa a classe `StrKeyDict0`, que passa nos doctests acima.



Uma forma melhor de criar uma mapeamento definido pelo usuário é criar uma subclasse de `collections.UserDict` em vez de `dict` (como faremos no Exemplo 34). Aqui criamos uma subclasse de `dict` apenas para mostrar que `_missing_` é suportado pelo método embutido `dict.__getitem__`.

Exemplo 33. StrKeyDict0 converte chaves não-string para string no momento da consulta (vejas os testes no Exemplo 32)

```
class StrKeyDict0(dict): ①

    def __missing__(self, key):
        if isinstance(key, str): ②
            raise KeyError(key)
        return self[str(key)] ③

    def get(self, key, default=None):
        try:
            return self[key] ④
        except KeyError:
            return default ⑤

    def __contains__(self, key):
        return key in self.keys() or str(key) in self.keys() ⑥
```

① StrKeyDict0 herda de dict.

② Verifica se key já é uma str. Se é, e está ausente, gera um KeyError.

③ Cria uma str de key e a procura.

④ O método get delega para __getitem__ usando a notação self[key]; isso dá oportunidade para nosso __missing__ agir.

⑤ Se um KeyError foi gerado, __missing__ já falhou, então devolvemos o default.

⑥ Procura pela chave não-modificada (a instância pode conter chaves não-str), depois por uma str criada a partir da chave.

Considere por um momento o motivo do teste isinstance(key, str) ser necessário na implementação de __missing__.

Sem aquele teste, nosso método __missing__ funcionaria bem com qualquer chave k—str ou não—sempre que str(k) produzisse uma chave existente. Mas se str(k) não for uma chave existente, teríamos uma recursão infinita. Na última linha de __missing__, self[str(key)] chamaria __getitem__, passando aquela chave str, e __getitem__, por sua vez, chamaria __missing__ novamente.

O método __contains__ também é necessário para que o comportamento nesse exemplo seja consistente, pois a operação k in d o chama, mas o método herdado de dict não invoca __missing__ com chaves ausentes. Há um detalhe útil em nossa implementação de __contains__: não verificamos a existência da chave da forma pythônica normal—k in d—porque str(key) in self chamaria __contains__ recursivamente. Evitamos isso procurando a chave explicitamente em self.keys().

Uma busca como k in my_dict.keys() é eficiente em Python 3 mesmo para mapeamentos muito grandes, porque dict.keys() devolve uma view, que é similar a um set, como veremos na Seção 3.12. Entretanto, lembre-se que k in my_dict faz o mesmo trabalho, e é mais rápido porque evita a

busca nos atributos para encontrar o método `.keys`.

Eu tinha uma razão específica para usar `self.keys()` no método `__contains__` do Exemplo 33. A verificação da chave não-modificada `key in self.keys()` é necessária por correção, pois `StrKeyDict` não obriga todas as chaves no dicionário a serem do tipo `str`. Nossa único objetivo com esse exemplo simples foi fazer a busca "mais amigável", e não forçar tipos.



Classes definidas pelo usuário derivadas de mapeamentos da biblioteca padrão podem ou não usar `__missing__` como alternativa em sua implementação de `__getitem__`, `get`, ou `__contains__`, como explicado na próxima seção.

3.5.3. O uso inconsistente de `__missing__` na biblioteca padrão

Considere os seguintes cenários, e como eles afetam a busca de chaves ausentes:

subclasse de dict

Uma subclasse de `dict` que implemente apenas `__missing__` e nenhum outro método. Nesse caso, `__missing__` pode ser chamado apenas em `d[k]`, que usará o `__getitem__` herdado de `dict`.

subclasse de `collections.UserDict`

Da mesma forma, uma subclasse de `UserDict` que implemente apenas `__missing__` e nenhum outro método. O método `get` herdado de `UserDict` chama `__getitem__`. Isso significa que `__missing__` pode ser chamado para tratar de consultas com `d[k]` e com `d.get(k)`.

subclasse de `abc.Mapping` com o `__getitem__` mais simples possível

Uma subclasse mínima de `abc.Mapping`, implementando `__missing__` e os métodos abstratos obrigatórios, incluindo uma implementação de `__getitem__` que não chama `__missing__`. O método `__missing__` nunca é acionado nessa classe.

subclasse de `abc.Mapping` com `__getitem__` chamando `__missing__`

Uma subclasse mínima de `abc.Mapping`, implementando `__missing__` e os métodos abstratos obrigatórios, incluindo uma implementação de `__getitem__` que chama `__missing__`. O método `__missing__` é acionado nessa classe para consultas por chaves ausentes feitas com `d[k]`, `d.get(k)`, e `k in d`.

Veja `missing.py` [<https://fpy.li/3-7>] no repositório de exemplos de código para demonstrações dos cenários descritos acima.

Os quatro cenários que acabo de descrever supõem implementações mínimas. Se a sua subclasse implementa `__getitem__`, `get`, e `__contains__`, então você pode ou não fazer tais métodos usarem `__missing__`, dependendo de suas necessidades. O ponto aqui é mostrar que é preciso ter cuidado ao criar subclasses dos mapeamentos da biblioteca padrão para usar `__missing__`, porque as classes base suportam comportamentos default diferentes. Não se esqueça que o comportamento de `setdefault` e `update` também é afetado pela consulta de chaves. E por fim, dependendo da lógica de seu `__missing__`, pode ser necessário implementar uma lógica especial em `__setitem__`, para evitar inconsistências ou comportamentos surpreendentes. Veremos um exemplo disso na Seção 3.6.5.

Até aqui tratamos dos tipos de mapeamentos `dict` e `defaultdict`, mas a biblioteca padrão traz outras implementações de mapeamentos, que discutiremos a seguir.

3.6. Variações de dict

Nessa seção falaremos brevemente sobre os tipos de mapeamentos incluídos na biblioteca padrão diferentes de `defaultdict`, já visto na Seção 3.5.1.

3.6.1. `collections.OrderedDict`

Agora que o `dict` embutido também mantém as chaves ordenadas (desde Python 3.6), o motivo mais comum para usar `OrderedDict` é escrever código compatível com versões anteriores de Python. Dito isso, a documentação lista algumas diferenças entre `dict` e `OrderedDict` que ainda persistem e que cito aqui:

- A operação de igualdade para `OrderedDict` verifica a igualdade da ordenação.
- O método `popitem()` de `OrderedDict` tem uma assinatura diferente, que aceita um argumento opcional especificando qual item será devolvido.
- `OrderedDict` tem um método `move_to_end()`, que reposiciona de um elemento para uma ponta do dicionário de forma eficiente.
- O `dict` comum foi projetado para ser muito bom nas operações de mapeamento. Preservar a ordem de inserção era uma preocupação secundária.
- `OrderedDict` foi projetado para ser bom em operações de reordenamento. Eficiência espacial, velocidade de iteração e o desempenho de operações de atualização eram preocupações secundárias.
- Em termos do algoritmo, um `OrderedDict` lida melhor que um `dict` com operações frequentes de reordenamento. Isso o torna adequado para monitorar acessos recentes (em um cache LRU^[11], por exemplo).

3.6.2. `collections.ChainMap`

Uma instância de `ChainMap` mantém uma lista de mapeamentos que podem ser consultados como se fossem um mapeamento único. A busca é realizada em cada mapa incluído, na ordem em que eles aparecem na chamada ao construtor, e é bem sucedida assim que a chave é encontrada em um daqueles mapeamentos. Por exemplo:

```
>>> d1 = dict(a=1, b=3)
>>> d2 = dict(a=2, b=4, c=6)
>>> from collections import ChainMap
>>> chain = ChainMap(d1, d2)
>>> chain['a']
1
>>> chain['c']
6
```

A instância de `ChainMap` não cria cópias dos mapeamentos, mantém referências para eles. Atualizações ou inserções a um `ChainMap` afetam apenas o primeiro mapeamento passado. Continuando do exemplo anterior:

```
>>> chain['c'] = -1
>>> d1
{'a': 1, 'b': 3, 'c': -1}
>>> d2
{'a': 2, 'b': 4, 'c': 6}
```

Um ChainMap é útil na implementação de linguagens com escopos aninhados, onde cada mapeamento representa um contexto de escopo, desde o escopo aninhado mais interno até o mais externo. A seção "Objetos ChainMap" [https://fpy.li/33], na documentação de collections, apresenta vários exemplos do uso de Chainmap, incluindo esse trecho inspirado nas regras básicas de consulta de variáveis no Python:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

O [environment_class_ex] mostra uma subclasse de ChainMap usada para implementar um interpretador parcial da linguagem de programação Scheme.

3.6.3. collections.Counter

Um mapeamento que mantém uma contagem para cada chave. Atualizar uma chave existente adiciona à sua contagem. Isso pode ser usado para contar instâncias de objetos hashable ou como um *multiset* (conjunto múltiplo), discutido adiante nessa seção. Counter implementa os operadores + e - para combinar contagens, e outros métodos úteis tal como o `most_common([n])`, que devolve uma lista ordenada de tuplas com os *n* itens mais comuns e suas contagens; veja a documentação [https://fpy.li/34].

Aqui temos um Counter usado para contar as letras em palavras:

```
>>> ct = collections.Counter('abracadabra')
>>> ct
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.update('aaaaazzz')
>>> ct
Counter({'a': 10, 'z': 3, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.most_common(3)
[('a', 10), ('z', 3), ('b', 2)]
```

Observe que as chaves 'b' e 'r' estão empatadas em terceiro lugar, mas `ct.most_common(3)` mostra apenas três contagens.

Para usar `collections.Counter` como um conjunto múltiplo, trate cada chave como um elemento de um conjunto, e a contagem será o número de ocorrências daquele elemento no conjunto.

3.6.4. shelve.Shelf

O módulo `shelve` na biblioteca padrão fornece armazenamento persistente a um mapeamento de chaves em formato string para objetos Python serializados no formato binário `pickle`. O nome curioso, `shelve`, faz sentido quando você percebe que potes de `pickle` são armazenadas em prateleiras.^[12]

A função de módulo `shelve.open` devolve uma instância de `shelve.Shelf`—um banco de dados DBM simples de chave-valor, baseado no módulo `dbm`, com as seguintes características:

- `shelve.Shelf` é uma subclasse de `abc.MutableMapping`, então fornece os métodos essenciais esperados de um tipo mapeamento.
- Além disso, `shelve.Shelf` fornece alguns outros métodos de gerenciamento de E/S, como `sync` e `close`.
- Uma instância de `Shelf` é um gerenciador de contexto, então é possível usar um bloco `with` para garantir que ela seja fechada após o uso.
- Chaves e valores são salvos sempre que um novo valor é atribuído a uma chave.
- As chaves devem ser strings.
- Os valores devem ser objetos que o módulo `pickle` possa serializar.

A documentação para os módulos `shelve` [<https://fpy.li/35>], `dbm` [<https://fpy.li/36>] (EN), e `pickle` [<https://fpy.li/37>] traz mais detalhes e também algumas ressalvas.



O `pickle` de Python é fácil de usar nos casos mais simples, mas tem vários inconvenientes. Leia o "Pickle's nine flaws" [<https://fpy.li/3-13>], de Ned Batchelder, antes de adotar qualquer solução envolvendo `pickle`. Em seu post, Ned menciona outros formatos de serialização que podem ser considerados como alternativas.

As classes `OrderedDict`, `ChainMap`, `Counter`, e `Shelf` podem ser usadas diretamente, mas também podem ser customizadas por subclasses. `UserDict`, por outro lado, foi planejada apenas como uma classe base a ser estendida.

3.6.5. Criando subclasses de UserDict em vez de dict

É melhor criar um novo tipo de mapeamento estendendo `collections.UserDict` em vez de `dict`. Percebemos isso quando tentamos estender nosso `StrKeyDict` do Exemplo 33 para assegurar que qualquer chave adicionada ao mapeamento seja armazenada como `str`.

A principal razão pela qual é melhor criar uma classe subclasse de `UserDict` em vez de `dict` é que o tipo embutido tem alguns atalhos de implementação, que acabam nos obrigando a sobreescrivar métodos que poderíamos apenas herdar de `UserDict` sem maiores problemas.^[13]

Observe que `UserDict` não herda de `dict`, mas usa uma composição: a classe tem uma instância interna de `dict`, chamada `data`, que mantém os itens propriamente ditos. Isso evita recursão indesejada quando escrevemos métodos especiais, como `__setitem__`, e simplifica a programação de `__contains__`, quando comparado com o Exemplo 33.

Graças a `UserDict`, o `StrKeyDict` (Exemplo 34) é mais conciso que o `StrKeyDict0` (Exemplo 33), mas ainda faz melhor: ele armazena todas as chaves como `str`, evitando surpresas desagradáveis se a instância for criada ou atualizada com dados contendo chaves de outros tipos (que não `string`).

Exemplo 34. `StrKeyDict` sempre converte chaves que não sejam strings para str na inserção, atualização e busca

```
import collections

class StrKeyDict(collections.UserDict): ①

    def __missing__(self, key): ②
        if isinstance(key, str):
            raise KeyError(key)
        return self[str(key)]

    def __contains__(self, key):
        return str(key) in self.data ③

    def __setitem__(self, key, item):
        self.data[str(key)] = item ④
```

① `StrKeyDict` estende `UserDict`.

② `__missing__` é exatamente igual ao do Exemplo 33.

③ `__contains__` é mais simples: podemos assumir que todas as chaves armazenadas são `str`, e podemos operar sobre `self.data` em vez de invocar `self.keys()`, como fizemos em `StrKeyDict0`.

④ `__setitem__` converte qualquer `key` para uma `str`. Esse método é mais fácil de sobrescrever quando podemos delegar para o atributo `self.data`.

Como `UserDict` estende `abc.MutableMapping`, o restante dos métodos que fazem de `StrKeyDict` um mapeamento completo são herdados de `UserDict`, `MutableMapping`, ou `Mapping`. Estes últimos contém vários métodos concretos úteis, apesar de serem classes base abstratas (ABCs). Os seguinte métodos são dignos de nota:

`MutableMapping.update`

Esse método poderoso pode ser chamado diretamente, mas também é usado por `__init__` para criar a instância a partir de outros mapeamentos, de iteráveis de pares (`chave, valor`), e de argumentos nomeados. Como usa `self[chave] = valor` para adicionar itens, ele termina por invocar nossa implementação de `__setitem__`.

`Mapping.get`

No `StrKeyDict0` (Exemplo 33), precisamos codificar nosso próprio `get` para devolver os mesmos resultados de `__getitem__`, mas no Exemplo 34 herdamos `Mapping.get`, que é implementado exatamente como `StrKeyDict0.get` (consulte o código-fonte de Python [<https://fpy.li/3-14>]).



Antoine Pitrou escreveu a PEP 455—Adding a key-transforming dictionary to collections (*Acrescentando um dicionário com transformação de chaves a*

collections) [https://fpy.li/pep455] (EN) e um patch para aperfeiçoar o módulo collections com uma classe TransformDict, que é mais genérico que StrKeyDict e preserva as chaves como fornecidas antes de aplicar a transformação. A PEP 455 foi rejeitada em maio de 2015—veja a mensagem de rejeição [https://fpy.li/3-15] (EN) de Raymond Hettinger. Para experimentar com a TransformDict, extraí o patch de Pitrou do issue18986 [https://fpy.li/3-16] (EN) para um módulo independente (*03-dict-set/transformdict.py* [https://fpy.li/3-17] disponível no repositório de código da segunda edição do *Fluent Python* [https://fpy.li/code]).

Sabemos que existem tipos de sequências imutáveis, mas e mapeamentos imutáveis? Não existe um tipo real desses na biblioteca padrão, mas um substituto está disponível. É o que vem a seguir.

3.7. Mapeamentos imutáveis

Os tipos de mapeamentos disponíveis na biblioteca padrão são todos mutáveis, mas pode ser desejável impedir que os usuários mudem um mapeamento por acidente. Um caso de uso concreto pode ser encontrado, novamente, em uma biblioteca de programação de hardware como a *Pingo*, mencionada na Seção 3.5.2: o mapeamento `board.pins` representa as portas de GPIO (General Purpose Input/Output, Entrada/Saída Genérica) em um dispositivo. Dessa forma, seria útil evitar atualizações descuidadas de `board.pins`, pois o hardware não pode ser modificado via software: qualquer mudança no mapeamento o tornaria inconsistente com a realidade física do dispositivo.

O módulo `types` oferece uma classe invólucro (*wrapper*) chamada `MappingProxyType` que, dado um mapeamento, devolve uma instância de `mappingproxy`, que é um proxy somente para leitura (mas dinâmico) do mapeamento original. Isso significa que atualizações ao mapeamento original são refletidas no `mappingproxy`, mas nenhuma mudança pode ser feita através desse último. Veja uma breve demonstração no Exemplo 35.

Exemplo 35. MappingProxyType cria uma instância somente de leitura de mappingproxy a partir de um dict

```
>>> from types import MappingProxyType
>>> d = {1: 'A'}
>>> d_proxy = MappingProxyType(d)
>>> d_proxy
mappingproxy({1: 'A'})
>>> d_proxy[1] ①
'A'
>>> d_proxy[2] = 'x' ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment
>>> d[2] = 'B'
>>> d_proxy ③
mappingproxy({1: 'A', 2: 'B'})
>>> d_proxy[2]
'B'
>>>
```

- ① Os items em `d` podem ser vistos através de `d_proxy`.
- ② Não é possível fazer modificações através de `d_proxy`.
- ③ `d_proxy` é dinâmica: qualquer mudança em `d` é refletida ali.

Isso pode ser usado assim na prática, no cenário da programação de hardware: o construtor em uma subclasse concreta `Board` preencheria um mapeamento privado com os objetos porta, e o exporia aos clientes da API via um atributo público `.portas`, implementado como um `mappingproxy`. Dessa forma os clientes não poderiam acrescentar, remover ou modificar as portas por acidente.

A seguir estudaremos `views`—que permitem operações de alto desempenho em um `dict`, sem cópias desnecessárias dos dados.

3.8. Views de dicionários

Objetos `dict` implementam os métodos `.keys()`, `.values()`, e `.items()`, que devolvem instâncias de classes chamadas `dict_keys`, `dict_values`, e `dict_items`, respectivamente. Essas `views` de dicionário são projeções somente para leitura de estruturas de dados internas usadas na implementação de `dict`. Elas evitam o uso de memória adicional dos métodos equivalentes no Python 2, que construían listas, duplicando dados já presentes no `dict`. E também substituem os métodos antigos que devolviam iteradores.

O Exemplo 36 mostra algumas operações básicas suportadas por todas as `views` de dicionários.

Exemplo 36. O método `.values()` devolve uma view dos valores em um `dict`

```
>>> d = dict(a=10, b=20, c=30)
>>> values = d.values()
>>> values
dict_values([10, 20, 30]) ①
>>> len(values) ②
3
>>> list(values) ③
[10, 20, 30]
>>> reversed(values) ④
<dict_reversevalueiterator object at 0x10e9e7310>
>>> values[0] ⑤
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_values' object is not subscriptable
```

- ① O `repr` de um objeto `view` mostra seu conteúdo.
- ② Podemos consultar a `len` de uma `view`.
- ③ `Views` são iteráveis, então é fácil criar listas a partir delas.
- ④ `Views` implementam `__reversed__`, devolvendo um iterador customizado.
- ⑤ Não é possível usar `[]` para obter itens individuais de uma `view`.

Um objeto `view` é um proxy dinâmico. Se o `dict` fonte é atualizado, as mudanças podem ser vistas

imediatamente através de uma view existente. Continuando do Exemplo 36:

```
>>> d['z'] = 99
>>> d
{'a': 10, 'b': 20, 'c': 30, 'z': 99}
>>> values
dict_values([10, 20, 30, 99])
```

As classes `dict_keys`, `dict_values`, e `dict_items` são internas: elas não estão disponíveis via `__builtins__` ou qualquer módulo da biblioteca padrão, e mesmo que você obtenha uma referência para uma delas, não pode usar essa referência para criar uma view do zero no seu código Python:

```
>>> values_class = type({}.values())
>>> v = values_class()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'dict_values' instances
```

A classe `dict_values` é a view de dicionário mais simples—ela implementa apenas os métodos especiais `__len__`, `__iter__`, e `__reversed__`. Além desses métodos, `dict_keys` e `dict_items` implementam vários métodos dos `sets`, quase tantos quanto a classe `frozenset`. Após vermos os conjuntos, voltaremos a estudar `dict_keys` e `dict_items`, na Seção 3.12.

Agora vamos ver algumas regras e dicas baseadas na forma como `dict` é implementado debaixo dos panos.

3.9. Consequências práticas da forma como dict funciona

A implementação da tabela de hash do `dict` de Python é muito eficiente, mas é importante entender os efeitos práticos desse design:

- Chaves devem ser objetos `hashable`. Eles devem implementar métodos `__hash__` e `__eq__` apropriados, como descrito na Seção 3.4.1.
- O acesso aos itens através da chave é muito rápido. Mesmo que um `dict` tenha milhões de chaves, Python pode localizar uma chave diretamente, computando o código hash da chave e derivando um deslocamento do índice na tabela de hash, com um possível ônus de um pequeno número de tentativas até encontrar a entrada correspondente.
- A ordenação das chaves é preservada, como efeito colateral de um layout de memória mais compacto para `dict` no CPython 3.6, que se tornou um recurso oficial da linguagem no 3.7.
- Apesar de seu novo layout compacto, os `dicts` apresentam, inevitavelmente, um uso adicional significativo de memória. A estrutura de dados interna mais compacta para um contêiner seria um array de ponteiros para os itens.^[14] Comparado a isso, uma tabela de hash precisa armazenar mais dados para cada entrada e, para manter a eficiência, Python precisa manter pelo menos um terço das linhas da tabela de hash vazias.

- Para economizar memória, evite criar atributos de instância fora do método `__init__`.

Essa última dica, sobre atributos de instância, é consequência do comportamento default de Python, de armazenar atributos de instância em um atributo `__dict__` especial, que é um dict vinculado a cada instância.^[15] Desde a implementação da PEP 412—Key-Sharing Dictionary (*Dicionário de Compartilhamento de Chaves*) [<https://fpy.li/pep412>] (EN), no Python 3.3, instâncias de uma classe podem compartilhar uma tabela de hash comum, armazenada com a classe. Essa tabela de hash comum é compartilhada pelo `__dict__` de cada nova instância que, quando `__init__` retorna, tenha os mesmos nomes de atributos que a primeira instância a ser criada naquela classe. O `__dict__` de cada instância então pode manter apenas seus próprios valores de atributos como um array de ponteiros. Acrescentar um atributo de instância após o `__init__` obriga Python a criar uma nova tabela de hash só para o `__dict__` daquela instância (que era o comportamento default antes de Python 3.3). De acordo com a PEP 412, essa otimização reduz o uso da memória entre 10% e 20% em programas orientados as objetos. Os detalhes das otimizações do layout compacto e do compartilhamento de chaves são bastante complexos. Para saber mais, leia "Internals of sets and dicts" [<https://fpy.li/hashint>] (EN).

Agora vamos estudar conjuntos.

3.10. Teoria dos conjuntos

Conjuntos não são novidade no Python, mas ainda são um tanto subutilizados. O tipo `set` e seu irmão imutável, `frozenset`, surgiram inicialmente como módulos na biblioteca padrão de Python 2.3, e foram promovidos a tipos embutidos no Python 2.6.



Nesse livro, uso a palavra "conjunto" para me referir tanto a `set` quanto a `frozenset`.

Um conjunto é uma coleção de objetos únicos. Uma grande utilidade dos conjuntos é descartar itens duplicados:

```
>>> l = ['spam', 'spam', 'eggs', 'spam', 'bacon', 'eggs']
>>> set(l)
{'eggs', 'spam', 'bacon'}
>>> list(set(l))
['eggs', 'spam', 'bacon']
```

Para remover elementos duplicados preservando a ordem da primeira ocorrência de cada item, você pode fazer isso com um dict simples, assim:



```
>>> dict.fromkeys(l).keys()
dict_keys(['spam', 'eggs', 'bacon'])
>>> list(dict.fromkeys(l).keys())
['spam', 'eggs', 'bacon']
```

Elementos de um conjunto devem ser *hashable*. O tipo `set` não é hashable, então não é possível

criar um set com instâncias aninhadas de set. Mas frozenset é hashable, então você pode ter instâncias de frozenset dentro de um set.

Além de garantir que cada elemento é único, os tipos conjunto implementam muitas operações entre conjuntos como operadores infixos. Assim, dados dois conjuntos `a` e `b`, `a | b` devolve sua união, `a & b` calcula a intersecção, `a - b` a diferença, e `a ^ b` a diferença simétrica. Quando bem utilizadas, as operações de conjuntos podem reduzir tanto a contagem de linhas quanto o tempo de execução de programas Python, ao mesmo tempo em que tornam o código mais legível e mais fácil de entender—pela remoção de loops e lógica condicional.

Por exemplo, imagine que você tem um grande conjunto de endereços de email (o "palheiro"—haystack) e um conjunto menor de endereços (as "agulhas"—needles'), e precisa contar quantas agulhas existem no palheiro. Graças à intersecção de set (o operador `&`), é possível codar isso em uma expressão simples (veja o Exemplo 37).

Exemplo 37. Conta as ocorrências de agulhas (needles) em um palheiro (haystack), ambos do tipo set

```
found = len(needles & haystack)
```

Sem o operador de intersecção, seria necessário escrever o Exemplo 38 para realizar a mesma tarefa executada pelo Exemplo 37.

Exemplo 38. Conta as ocorrências de agulhas (needles) em um palheiro (haystack); mesmo resultado final do Exemplo 37

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

O Exemplo 37 é um pouco mais rápido que o Exemplo 38. Por outro lado, o Exemplo 38 funciona para quaisquer objetos iteráveis needles e haystack, enquanto o Exemplo 37 exige que ambos sejam conjuntos. Mas se você não tem conjuntos à mão, pode sempre criá-los na hora, como mostra o Exemplo 39.

Exemplo 39. Conta as ocorrências de agulhas (needles) em um palheiro (haystack); essas linhas funcionam para qualquer tipo iterável

```
found = len(set(needles) & set(haystack))

# another way:
found = len(set(needles).intersection(haystack))
```

Claro, há o custo extra envolvido na criação dos conjuntos no Exemplo 39, mas se ou as needles ou o haystack já forem um set, a alternativa no Exemplo 39 pode ser mais barata que o Exemplo 38.

Qualquer dos exemplos acima é capaz de buscar 1000 elementos em um haystack de 10 milhões de itens em cerca de 0,3 milisegundos—isso é cerca de 0,3 microsegundos por elemento.

Além do teste de existência extremamente rápido (graças à tabela de hash), os tipos embutidos `set` e `frozenset` oferecem uma rica API para criar novos conjuntos ou, no caso de `set`, para modificar conjuntos existentes. Vamos discutir essas operações em breve, após uma observação sobre sintaxe.

3.10.1. Sets literais

A sintaxe de literais `set`—`{1}`, `{1, 2}`, etc.—parece muito com a notação matemática, mas tem uma importante exceção: não há notação literal para o set vazio, então precisamos nos lembrar de escrever `set()`.



Peculiaridade sintática

Para criar um set vazio, usamos o construtor sem argumentos: `set()`. Se você escrever `{}`, vai criar um dict vazio—isso não mudou no Python 3.

No Python 3, a representação padrão dos sets como strings sempre usa a notação `{...}`, exceto para o conjunto vazio:

```
>>> s = {1}
>>> type(s)
<class 'set'>
>>> s
{1}
>>> s.pop()
1
>>> s
set()
```

A sintaxe do set literal, como `{1, 2, 3}`, é mais rápida e mais legível que uma chamada ao construtor (por exemplo, `set([1, 2, 3])`). Essa última forma é mais lenta porque, para avaliá-la, Python precisa buscar o nome `set` para obter seu construtor, daí criar uma lista e, finalmente, passá-la para o construtor. Por outro lado, para processar um literal como `{1, 2, 3}`, o Python roda um bytecode especializado, `BUILD_SET`.^[16]

Não há sintaxe especial para representar literais `frozenset`—eles só podem ser criados chamando seu construtor. Sua representação padrão como string no Python 3 se parece com uma chamada ao construtor de `frozenset` com um argumento `set`. Observe a saída no console:

```
>>> frozenset(range(10))
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

E por falar em sintaxe, a ideia das listcomps foi adaptada para criar conjuntos também.

3.10.2. Compreensões de conjuntos

Compreensões de conjuntos (`setcomps`) apareceram há bastante tempo, no Python 2.7, junto com as `dictcomps` que vimos na Seção 3.2.1. O Exemplo 40 mostra procedimento.

Exemplo 40. Cria um conjunto de caracteres Latin-1 que tenham a palavra "SIGN" em seus nomes Unicode

```
>>> from unicodedata import name ①
>>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i), '')} ②
{'§', '=', '¢', '#', '¤', '<', '¥', 'µ', '×', '$', '¶', 'ƒ', '©',
 '°', '+', '÷', '±', '>', '¬', '®', '%'}
```

① Importa a função `name` de `unicodedata` para obter os nomes dos caracteres.

② Cria um conjunto de caracteres com códigos entre 32 e 255 que contenham a palavra 'SIGN' em seus nomes.

A ordem da saída muda a cada processo Python, devido ao hash "salgado", mencionado na Seção 3.4.1.

Questões de sintaxe à parte, vamos considerar agora o comportamento dos conjuntos.

3.11. Consequências práticas da forma de funcionamento dos conjuntos

Os tipos `set` e `frozenset` são ambos implementados com um tabela de hash. Isso tem os seguintes efeitos:

- Elementos de conjuntos tem que ser objetos `hashable`. Eles precisam implementar métodos `__hash__` e `__eq__` adequados, como descrito na Seção 3.4.1.
- O teste de existência de um elemento é muito eficiente. Um conjunto pode ter milhões de elementos, mas um elemento pode ser localizado diretamente, computando o código hash da chave e derivando um deslocamento do índice, com o possível ônus de um pequeno número de tentativas até encontrar a entrada correspondente ou exaurir a busca.
- Conjuntos usam mais memória que um array de ponteiros para seus elementos—que é uma estrutura mais compacta, porém menos eficientes para buscas quando seu tamanho cresce além de uns poucos elementos.
- A ordem dos elementos depende da ordem de inserção, mas não de forma útil ou confiável. Se dois elementos são diferentes mas tem o mesmo código hash, sua posição depende de qual elemento foi inserido primeiro.
- Acrescentar elementos a um conjunto muda a ordem dos elementos existentes. Isso ocorre porque o algoritmo se torna menos eficiente se a tabela de hash tiver mais de dois terços de ocupação, então Python pode mover e redimensionar a tabela conforme ela cresce. Quando isso acontece, os elementos são reinseridos e sua ordem relativa pode mudar.

Veja o post "Internals of sets and dicts" [<https://fpy.li/hashint>] (EN) no <http://fluentpython.com> para maiores detalhes.

Agora vamos revisar a vasta seleção de operações oferecidas pelos conjuntos.

3.11.1. Operações de conjuntos

A Figura 9 dá uma visão geral dos métodos disponíveis em conjuntos mutáveis e imutáveis. Muitos deles são métodos especiais que sobrecarregam operadores, como `&` and `>=`. A Tabela 8 mostra os operadores matemáticos de conjuntos que tem operadores ou métodos correspondentes no Python. Note que alguns operadores e métodos realizam mudanças no mesmo lugar sobre o conjunto alvo (por exemplo, `&=`, `difference_update`, etc.). Tais operações não fazem sentido no mundo ideal dos conjuntos matemáticos, e também não são implementadas em `frozenset`.



Os operadores infixos na Tabela 8 exigem que os dois operandos sejam conjuntos, mas todos os outros métodos recebem um ou mais argumentos iteráveis. Por exemplo, para produzir a união de quatro coleções, `a, b, c, e d`, você pode chamar `a.union(b, c, d)`, onde `a` precisa ser um `set`, mas `b, c, e d` podem ser iteráveis de qualquer tipo que produza itens hashable. Para criar um novo conjunto com a união de quatro iteráveis, desde Python 3.5 você pode escrever `{*a, *b, *c, *d}` ao invés de atualizar um conjunto existente, graças à PEP 448—Additional Unpacking Generalizations (*Generalizações de Desempacotamento Adicionais*) [<https://fpy.li/pep448>].

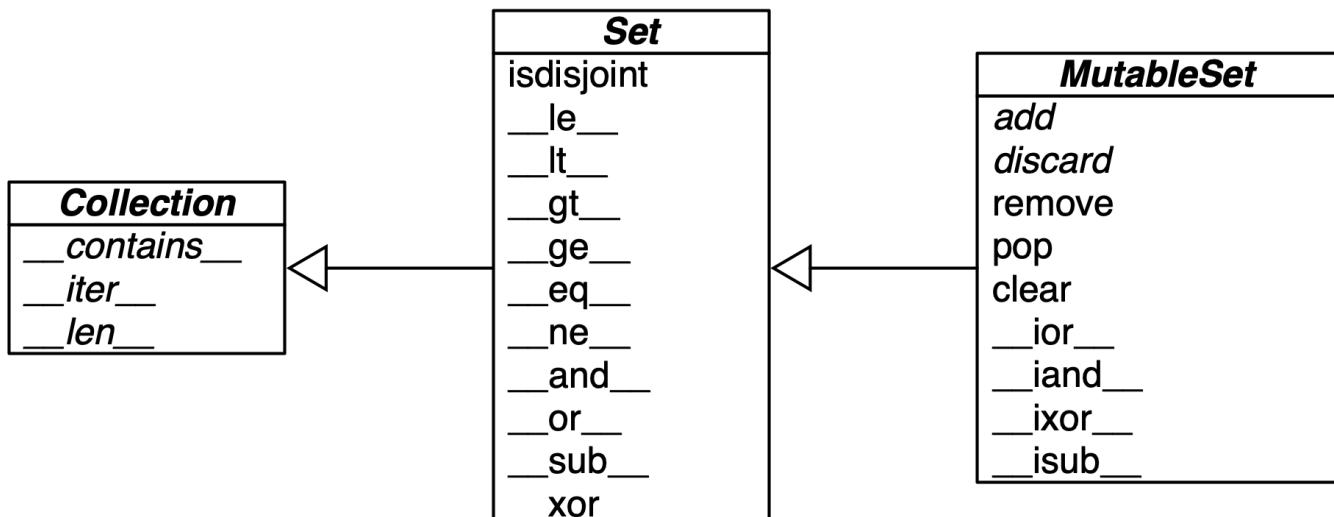


Figura 9. Diagrama de classes UML simplificado para `MutableSet` e suas superclasses em `collections.abc` (nomes em itálico são classes e métodos abstratos; métodos de operadores reversos foram omitidos por concisão).

Tabela 8. Operações matemáticas com conjuntos: esses métodos produzem um novo conjunto ou atualizam o conjunto alvo no mesmo lugar, se ele for mutável

operador	método	
<code>s & z</code>	<code>s.__and__(z)</code>	Intersecção de <code>s</code> e <code>z</code>
<code>z & s</code>	<code>s.__rand__(z)</code>	Operador <code>&</code> invertido
	<code>s.intersection(it, ...)</code>	Intersecção de <code>s</code> e todos os conjuntos construídos a partir de iteráveis <code>it</code> , etc.
<code>s &= z</code>	<code>s.__iand__(z)</code>	<code>s</code> atualizado com a intersecção de <code>s</code> e <code>z</code>
	<code>s.intersection_update(it, ...)</code>	<code>s</code> atualizado com a intersecção de <code>s</code> e todos os conjuntos construídos a partir de iteráveis <code>it</code> , etc.

operador	método	
s z	s.__or__(z)	União de s e z
z s	s.__ror__(z)	invertido
	s.union(it, ...)	União de s e todos os conjuntos construídos a partir de iteráveis it, etc.
s = z	s.__ior__(z)	s atualizado com a união de s e z
	s.update(it, ...)	s atualizado com a união de s e todos os conjuntos construídos a partir de iteráveis it, etc.
s - z	s.__sub__(z)	Complemento relativo ou diferença entre s e z
z - s	s.__rsub__(z)	Operador - invertido
	s.difference(it, ...)	Diferença entre s e todos os conjuntos construídos a partir de iteráveis it, etc.
s -= z	s.__isub__(z)	s atualizado com a diferença entre s e z
	s.difference_update(it, ...)	s atualizado com a diferença entre s e todos os conjuntos construídos a partir de iteráveis it, etc.
s ^ z	s.__xor__(z)	Diferença simétrica (o complemento da intersecção s & z)
z ^ s	s.__rxor__(z)	Operador ^ invertido
	s.symmetric_difference(it)	Complemento de s & set(it)
s ^= z	s.__ixor__(z)	s atualizado com a diferença simétrica de s e z
	s.symmetric_difference_update(it, ...)	s atualizado com a diferença simétrica de s e todos os conjuntos construídos a partir de iteráveis it, etc.

A Tabela 9 lista predicados de conjuntos: operadores e métodos que devolvem True ou False.

Tabela 9. Operadores e métodos de comparação de conjuntos que devolvem um booleano

operador	método	
e in s	s.__contains__(e)	Elemento e é membro de s
s <= z	s.__le__(z)	s é um subconjunto do conjunto z
	s.issubset(it)	s é um subconjunto do conjunto criado a partir do iterável it
s < z	s.__lt__(z)	s é um subconjunto próprio ^[17] do conjunto z
s >= z	s.__ge__(z)	s é um superconjunto do conjunto z
	s.issuperset(it)	s é um superconjunto do conjunto criado a partir do iterável it
s > z	s.__gt__(z)	s é um superconjunto próprio do conjunto z
	s.isdisjoint(z)	s e z são disjuntos (não tem elementos em comum)

Além de operadores e métodos derivados da teoria matemática dos conjuntos, os tipos conjunto implementam outros métodos para tornar seu uso prático, resumidos na Tabela 10.

Tabela 10. Métodos adicionais de conjuntos

	set	frozenset	
s.add(e)	•		Adiciona elemento e a s
s.clear()	•		Remove todos os elementos de s
s.copy()	•	•	Cópia rasa de s
s.discard(e)	•		Remove elemento e de s, se existir
s.__iter__()	•	•	Obtém iterador de s
s.__len__()	•	•	len(s)
s.pop()	•		Remove e devolve um elemento de s, gerando um KeyError se s estiver vazio
s.remove(e)	•		Remove elemento e de s, gerando um KeyError se e não existir em s

Isso encerra nossa visão geral dos recursos dos conjuntos. Como prometido na Seção 3.8, vamos agora ver como dois dos tipos de views de dicionários se comportam de forma muito similar a um frozenset.

3.12. Operações de conjuntos em views de dict

A Tabela 11 mostra como os objetos view devolvidos pelos métodos .keys() e .items() de dict são notavelmente similares a um frozenset.

Tabela 11. Métodos implementados por frozenset, dict_keys, e dict_items

	frozenset	dict_keys	dict_items	
s.__and__(z)	•	•	•	s & z (interseção de s e z)
s.__rand__(z)	•	•	•	operador & invertido
s.__contains__()	•	•	•	e in s
s.copy()	•			Cópia rasa de s
s.difference(it, ...)	•			Diferença entre s e os iteráveis it, etc.
s.intersection(it, ...)	•			Intersecção de s e dos iteráveis it, etc.
s.isdisjoint(z)	•	•	•	s e z são disjuntos (não tem elementos em comum)
s.issubset(it)	•			s é um subconjunto do iterável it
s.issuperset(it)	•			s é um superconjunto do iterável it
s.__iter__()	•	•	•	obtém iterador para s
s.__len__()	•	•	•	len(s)
s.__or__(z)	•	•	•	s z (união de s e z)

	frozenset	dict_keys	dict_items	
s.__ror__()	•	•	•	Operador invertido
s.__reversed__()		•	•	Obtém iterador para s com a ordem invertida
s.__rsub__(z)	•	•	•	Operador - invertido
s.__sub__(z)	•	•	•	s - z (diferença entre s e z)
s.symmetric_difference(it)	•			Complemento de s & set(it)
s.union(it, ...)	•			União de s e dos iteráveis `it`, etc.
s.__xor__()	•	•	•	s ^ z (diferença simétrica de s e z)
s.__rxor__()	•	•	•	Operador ^ invertido

Especificamente, `dict_keys` e `dict_items` implementam os métodos especiais para suportar as poderosas operações de conjuntos `&` (intersecção), `|` (união), `-` (diferença), and `^` (diferença simétrica).

Por exemplo, usando `&` é fácil obter as chaves que aparecem em dois dicionários:

```
>>> d1 = dict(a=1, b=2, c=3, d=4)
>>> d2 = dict(b=20, d=40, e=50)
>>> d1.keys() & d2.keys()
{'b', 'd'}
```

Observe que o valor devolvido por `&` é um `set`. Melhor ainda: os operadores de conjuntos em `views` de dicionários são compatíveis com instâncias de `set`. Veja isso:

```
>>> s = {'a', 'e', 'i'}
>>> d1.keys() & s
{'a'}
>>> d1.keys() | s
{'a', 'c', 'b', 'd', 'i', 'e'}
```



Uma `view` obtida de `dict_items` só funciona como um conjunto se todos os valores naquele `dict` são `hashable`. Tentar executar operações de conjuntos sobre uma `view` devolvida por `dict_items` que contenha valores não-`hashable` gera um `TypeError: unhashable type 'T'`, sendo `T` o tipo do valor rejeitado.

Por outro lado, uma `view` devolvida por `dict_keys` sempre pode ser usada como um conjunto, pois todas as chaves são `hashable`—por definição.

Usar operações de conjunto com `views` pode evitar a necessidade de muitos loops e ifs quando seu código precisa inspecionar o conteúdo de dicionários. Deixe a eficiente implementação de Python em C trabalhar para você!

Com isso, encerramos esse capítulo.

3.13. Resumo do capítulo

Dicionários são a pedra fundamental de Python. Ao longo dos anos, a sintaxe literal familiar `{k1: v1, k2: v2}` foi aperfeiçoada para suportar desempacotamento com `**` e pattern matching, bem como com comprehensões de dict.

Além do dict básico, a biblioteca padrão oferece mapeamentos práticos prontos para serem usados, como o `defaultdict`, o `ChainMap`, e o `Counter`, todos definidos no módulo `collections`. Com a nova implementação de dict, o `OrderedDict` não é mais tão útil quanto antes, mas deve permanecer na biblioteca padrão para manter a compatibilidade retroativa—e por suas características específicas ausentes em dict, tal como a capacidade de levar em consideração o ordenamento das chaves em uma comparação `==`. Também no módulo `collections` está o `UserDict`, uma classe base fácil de usar na criação de mapeamentos customizados.

Dois métodos poderosos disponíveis na maioria dos mapeamentos são `setdefault` e `update`. O método `setdefault` pode atualizar itens que mantenham valores mutáveis—por exemplo, em um dict de valores `list`—evitando uma segunda busca pela mesma chave. O método `update` permite inserir ou sobrescrever itens em massa a partir de qualquer outro mapeamento, desde iteráveis que forneçam pares (`chave, valor`) até argumentos nomeados. Os construtores de mapeamentos também usam `update` internamente, permitindo que instâncias sejam inicializadas a partir de outros mapeamentos, de iteráveis e de argumentos nomeados. Desde Python 3.9 também podemos usar o operador `|=` para atualizar uma mapeamento e o operador `|` para criar um novo mapeamento a partir a união de dois mapeamentos.

Um gancho elegante na API de mapeamento é o método `__missing__`, que permite customizar o que acontece quando uma chave não é encontrada ao se usar a sintaxe `d[k]` syntax, que invoca `__getitem__`.

O módulo `collections.abc` oferece as classes base abstratas `Mapping` e `MutableMapping` como interfaces padrão, úteis para checagem de tipo durante a execução. O `MappingProxyType`, do módulo `types`, cria uma fachada imutável para um mapeamento que você precise proteger de modificações acidentais. Existem também ABCs para `Set` e `MutableSet`.

Views de dicionários foram uma grande novidade no Python 3, eliminando o uso desnecessário de memória dos métodos `.keys()`, `.values()`, e `.items()` de Python 2, que criavam listas duplicando os dados na instância alvo de dict. Além disso, as classes `dict_keys` e `dict_items` suportam os operadores e métodos mais úteis de `frozenset`.

3.14. Leitura complementar

Na documentação da Biblioteca Padrão de Python, a seção "collections—Tipos de dados de contêineres" [https://fpy.li/2w] inclui exemplos e receitas práticas para vários tipos de mapeamentos. O código-fonte do módulo, `Lib/collections/__init__.py`, é uma excelente referência para qualquer um que deseja criar novos tipos de mapeamentos ou entender a lógica dos tipos existentes. O capítulo 1 do *Python Cookbook, 3rd ed.* [https://fpy.li/pycook3] (O'Reilly), de David Beazley e Brian K. Jones traz 20 receitas práticas e perpicazes usando estruturas de dados—a maioria mostrando

formas inteligentes de usar dict.

Greg Gandenberger defende a continuidade do uso de `collections.OrderedDict`, com os argumentos de que "explícito é melhor que implícito," compatibilidade retroativa, e o fato de algumas ferramentas e bibliotecas presumirem que a ordenação das chaves de um dict é irrelevante—nesse post: "Python Dictionaries Are Now Ordered. Keep Using OrderedDict" (*Os dicionários de Python agora são ordenados. Continue a usar OrderedDict*) [<https://fpy.li/3-18>] (EN).

A PEP 3106—Revamping `dict.keys()`, `.values()` and `.items()` (*Renovando dict.keys(), .values() e .items()*) [<https://fpy.li/pep3106>] (EN) foi onde Guido van Rossum apresentou o recurso de views de dicionário para Python 3. No resumo, ele afirma que a ideia veio da Java Collections Framework.

O PyPy [<https://fpy.li/3-19>] foi o primeiro interpretador Python a implementar a proposta de Raymond Hettinger para dicts compactos, e eles escreverem em seu blog sobre isso, em "Faster, more memory efficient and more ordered dictionaries on PyPy" (*Dicionários mais rápidos, mais eficientes em termos de memória e mais ordenados no PyPy*) [<https://fpy.li/3-20>] (EN), reconhecendo que um layout similar foi adotado no PHP 7, como descrito em PHP's new hashtable implementation (*A nova implementação de tabelas de hash de PHP*) [<https://fpy.li/3-21>] (EN). É sempre muito bom quando criadores citam trabalhos anteriores de outros.

Na PyCon 2017, Brandon Rhodes apresentou "The Dictionary Even Mightier" (*O dicionário, ainda mais poderoso*) [<https://fpy.li/3-22>] (EN), uma continuação de sua apresentação animada clássica "The Mighty Dictionary" (*O poderoso dicionário*) [<https://fpy.li/3-23>] (EN)—incluindo colisões de hash animadas! Outro vídeo atual mas mais aprofundado sobre o funcionamento interno do dict de Python é "Modern Dictionaries" (*Dicionários modernos*) [<https://fpy.li/3-24>] (EN) de Raymond Hettinger, onde ele conta que após não conseguir convencer os desenvolvedores principais de Python sobre os dicts compactos, ele persuadiu a equipe do PyPy, eles os adotaram, a ideia ganhou força, e finalmente foi adicionada [<https://fpy.li/3-25>] ao CPython 3.6 por INADA Naoki. Para saber todos os detalhes, dê uma olhada nos extensos comentários no código-fonte do CPython para `Objects/dictobject.c` [<https://fpy.li/3-26>] (EN) e no documento de design em `Objects/dictnotes.txt` [<https://fpy.li/3-27>] (EN).

A justificativa para a adição de conjuntos ao Python está documentada na PEP 218—Adding a Built-In Set Object Type (*Adicionando um objeto embutido de tipo conjunto*) [<https://fpy.li/pep218>]. Quando a PEP 218 foi aprovada, nenhuma sintaxe literal especial foi adotada para conjuntos. Os literais set foram criados para Python 3 e implementados retroativamente no Python 2.7, assim como as compreensões de dict e set. Na PyCon 2019, apresentei "Set Practice: learning from Python's set types" (*A Prática dos Conjuntos: aprendendo com os tipos conjunto de Python*) [<https://fpy.li/3-29>] (EN), video [<https://fpy.li/3-28>], [slides]) "Set Practice: learning from Python's set types" [<https://fpy.li/3-28>]^[18], descrevendo casos de uso de conjuntos em programas reais, falando sobre o design de sua API, e sobre a implementação da `uintset` [<https://fpy.li/3-30>], uma classe de conjunto para elementos inteiros, usando um vetor de bits ao invés de uma tabela de hash, inspirada por um exemplo do capítulo 6 do excelente *The Go Programming Language* (A Linguagem de Programação Go) [<https://fpy.li/38>] (EN), de Alan Donovan e Brian Kernighan (Addison-Wesley).

A revista *Spectrum*, do IEEE, tem um artigo sobre Hans Peter Luhn, um prolífico inventor que patenteou um conjunto de cartões interligados que permitiam selecionar receitas de coquetéis a partir dos ingredientes disponíveis, entre inúmeras outras invenções, incluindo... tabelas de hash! Veja "Hans Peter Luhn and the Birth of the Hashing Algorithm" (*Hans Peter Luhn e o Nascimento do*

Algoritmo de Hash) [<https://fpy.li/3-31>].

Ponto de Vista

Açúcar sintático

Meu amigo Geraldo Cohen certa vez observou que Python é "simples e correto."

Puristas de linguagens de programação gostam de desprezar a sintaxe como algo desimportante.

Syntactic sugar causes cancer of the semicolon.^[19]

— Alan Perlis

A sintaxe é a interface de usuário de uma linguagem de programação, então tem muita importância na prática.

Antes de encontrar Python, fiz um pouco de programação para a web usando Perl e PHP. A sintaxe para mapeamentos nessas linguagens é muito útil. Sinto muita falta dela quando tenho que usar Java ou C.

Uma boa sintaxe para mapeamentos literais é muito conveniente para configuração, para implementações guiadas por tabelas, e para conter dados para prototipagem e testes. Essa foi uma das lições que os projetistas do Go aprenderam com as linguagens dinâmicas. A falta de uma boa forma de expressar dados estruturados no código empurrou a comunidade Java a adotar o prolixo e excessivamente complexo XML como formato de dados.

JSON foi proposto como "The Fat-Free Alternative to XML" (*A alternativa sem gordura ao XML*) [<https://fpy.li/3-32>] e se tornou um imenso sucesso, substituindo XML em vários contextos. Uma sintaxe concisa para listas e dicionários resulta em um excelente formato para troca de dados.

PHP e Ruby imitaram a sintaxe de hash do Perl, usando => para ligar chaves a valores. JavaScript usa : como Python. Por que usar dois caracteres, quando um já é legível o bastante?

O JSON veio de JavaScript, mas por acaso também é quase um subconjunto exato da sintaxe de Python. O JSON é compatível com Python, exceto por usar true, false, e null em vez de True, False, e None.

Armin Ronacher tuitou [<https://fpy.li/3-33>] que gosta de brincar com o espaço de nomes global de Python, para acrescentar apelidos compatíveis com o JSON para o True, o False, e o None de Python, pois daí ele pode colar trechos de JSON diretamente no console. Sua ideia básica:

```
>>> true, false, null = True, False, None
>>> fruit = {
...     "type": "banana",
...     "avg_weight": 123.2,
```

```

...     "edible_peel": false,
...     "species": ["acuminata", "balbisiana", "paradisiaca"],
...     "issues": null,
... }
>>> fruit
{'type': 'banana', 'avg_weight': 123.2, 'edible_peel': False,
'species': ['acuminata', 'balbisiana', 'paradisiaca'], 'issues': None}

```

A sintaxe que todo mundo agora usa para trocar dados é a sintaxe de dict e list de Python. Agora temos uma sintaxe agradável com a conveniência da preservação da ordem de inserção.

Simples e correto.

[1] Uma subclasse virtual é qualquer classe registrada com uma chamada ao método `.register()` de uma ABC, como explicado na [virtual_subclass_sec]. Um tipo implementado através da API Python/C também serve, se tiver bit de marcação específico setado no header. Veja Py_TPFLAGS_MAPPING [<https://fpy.li/2z>] (EN).

[2] O verbete para "hashable" no *Glossário de Python* [<https://fpy.li/32>] usa o termo "valor de hash" em vez de *código de hash*. Prefiro *código de hash* porque "valor" é um conceito frequentemente usado no contexto de mapeamentos, onde itens são compostos de chaves e valores. Então pode ser confuso se referir ao código de hash como um valor. Nesse livro usarei apenas *código de hash*.

[3] Veja a PEP 456—Secure and interchangeable hash algorithm (Algoritmo de hash seguro e intercambiável) [<https://fpy.li/pep456>] (EN) para saber mais sobre as implicações de segurança e as soluções adotadas.

[4] `default_factory` não é um método, mas um atributo chamável definido pelo usuário quando um `defaultdict` é instanciado.

[5] `OrderedDict.popitem(last=False)` remove o primeiro item inserido (FIFO). O argumento nomeado `last` não é suportado por `dict` ou `defaultdict`, pelo menos até Python 3.10b3.

[6] Operadores invertidos são tratados no `[ch_op_overload]`.

[7] O script original aparece no slide 41 da apresentação de Martelli, "Re-learning Python" (*Reaprendendo Python*) [<https://fpy.li/3-5>] (EN). O script é, na verdade, uma demonstração de `dict.setdefault`, como visto no nosso Exemplo 30.

[8] Isso é um exemplo do uso de um método como uma função de `__init__` de uma classe, o assunto do Capítulo 7.

[9] NT: "Missing" significa ausente, perdido ou desaparecido

[10] Uma biblioteca dessas é a *Pingo.io* [<https://fpy.li/3-6>], que não está mais em desenvolvimento ativo.

[11] NT: Least Recently Used, *Menos Recentemente Usado*, esquema de cache que descarta o item armazenado que esteja há mais tempo sem requisições

[12] NT: "to shelve" é "colocar na prateleira", "pickle" também significa "conserva" e "pickles" é literalmente *picles*. O trocadilho dos desenvolvedores de Python é sobre colocar pickles em shelves.

[13] O problema exato de se criar subclasses de `dict` e de outros tipos embutidos é tratado na [subclass_builtin_woes].

[14] É assim que as tuplas são armazenadas.

[15] A menos que classe tenha um atributo `__slots__`, como explicado na [slots_section].

[16] Isso pode ser interessante, mas não é super importante. Essa diferença de desempenho vai ocorrer apenas quando um conjunto literal for avaliado, e isso acontece no máximo uma vez por processo Python—quando um módulo é compilado pela primeira vez. Se você estiver curiosa, importe a função `dis` do módulo `dis`, e a use para inspecionar os bytecodes de um set literal—por exemplo `dis('set([1])')`—e uma chamada ao construtor `set`—`dis('set([1])')`

[17] NT: Na teoria dos conjuntos, A é um *subconjunto próprio* de B se A é subconjunto de B e A é diferente de B.

[18] The slides for this presentation are available at <https://fpy.li/3-29>.)

[19] NT: Explicando o trocadilho intraduzível: "colon", em inglês, designa "a parte central do intestino grosso"; "semicolon", por outro lado, é "ponto e vírgula". A frase diz, literalmente, "Açúcar sintético causa câncer no ponto e vírgula", que faz sentido em inglês pela proximidade sonora das palavras.

4. Texto em Unicode versus Bytes

Humanos usam texto. Computadores falam em bytes.^[1]

— Esther Nam e Travis Fischer

Python 3 introduziu uma forte distinção entre strings de texto humano e sequências de bytes em estado bruto. A conversão automática de sequências de bytes para texto Unicode ficou para trás no Python 2. Este capítulo trata de strings Unicode, sequências de bytes, e das codificações usadas para converter umas nas outras.

Dependendo do que você faz com Python, pode achar que entender o Unicode não é importante. Isso é improvável, mas mesmo que seja o caso, não há como escapar da separação entre `str` e `bytes`, que agora exige conversões explícitas. Como um bônus, você descobrirá que os tipos especializados de sequências binárias `bytes` e `bytearray` oferecem recursos que a classe `str` "pau para toda obra" de Python 2 não oferecia.

Nesse capítulo, veremos os seguintes tópicos:

- Caracteres, pontos de código e representações binárias
- Recursos exclusivos das sequências binárias: `bytes`, `bytearray`, e `memoryview`
- Codificando para o Unicode completo e para conjuntos de caracteres legados
- Evitando e tratando erros de codificação
- Melhores práticas para lidar com arquivos de texto
- A armadilha da codificação default e questões de E/S padrão
- Comparações seguras de texto Unicode com normalização
- Funções utilitárias para normalização, *case folding* (equiparação maiúsculas/minúsculas) e remoção de sinais diacríticos por força bruta
- Ordenação correta de texto Unicode com `locale` e a biblioteca `pyuca`
- Metadados de caracteres do banco de dados Unicode
- APIs duais, que processam `str` e `bytes`

4.1. Novidades nesse capítulo

O suporte ao Unicode no Python 3 sempre foi muito completo e estável, então o acréscimo mais notável é a Seção 4.9.1, descrevendo um utilitário de linha de comando para busca no banco de dados Unicode—uma forma de encontrar gatinhos sorridentes ou hieróglifos do Egito antigo.

Vale a pena mencionar que o suporte a Unicode no Windows ficou melhor e mais simples desde Python 3.6, como veremos na Seção 4.6.1.

Vamos começar então com os conceitos não-tão-novos mas fundamentais de caracteres, pontos de código e bytes.

Para essa segunda edição, expandi a seção sobre o módulo `struct` e o publiquei online em "Parsing binary records with struct" (*Analisando registros binários com struct*) [<https://fpy.li/4-3>], (EN) no fluentpython.com [<http://fluentpython.com>] (EN) o website que complementa o livro em inglês.



Lá você também vai encontrar o "Building Multi-character Emojis" (*Criando emojis multi-caractere*) [<https://fpy.li/4-4>] (EN), descrevendo como combinar caracteres Unicode para criar bandeiras de países, bandeiras de arco-íris, pessoas com tonalidades de pele diferentes e ícones de diferentes tipos de famílias.

4.2. Questões de caracteres

O conceito de "string" é simples: uma string é uma sequência de caracteres. O problema está na definição de "caractere".

Em 2023, a melhor definição de "caractere" que temos é um caractere Unicode. Consequentemente, os itens que compõe um `str` de Python 3 são caracteres Unicode, como os itens de um objeto `unicode` no Python 2. Em contraste, os itens de uma `str` no Python 2 são bytes, assim como os itens num objeto `bytes` de Python 3.

O padrão Unicode separa explicitamente a identidade dos caracteres de representações binárias específicas:

- A identidade de um caractere é chamada deponto de código (*code point*). É um número de 0 a 1.114.111 (na base 10), formatado no padrão Unicode como 4 a 6 dígitos hexadecimais precedidos pelo prefixo "U+", de U+0000 a U+10FFFF. Por exemplo, o ponto de código da letra A é U+0041, o símbolo do Euro é U+20AC, e o símbolo musical da clave de sol corresponde ao ponto de código U+1D11E. Cerca de 13% dos pontos de código possíveis tem caracteres atribuídos no Unicode 13, a versão do padrão usada no Python 3.10.
- Os bytes específicos que representam um caractere dependem da codificação (*encoding*) usada. Uma codificação, nesse contexto, é um algoritmo que converte pontos de código para sequências de bytes, e vice-versa. O ponto de código para a letra A (U+0041) é codificado como um único byte, \x41, na codificação UTF-8, ou como os bytes \x41\x00 na codificação UTF-16LE. Em um outro exemplo, o UTF-8 exige três bytes para codificar o símbolo do Euro (U+20AC): \xe2\x82\xac. Mas no UTF-16LE o mesmo ponto de código é representado em dois bytes: \xac\x20.

Converter pontos de código para bytes é *codificar*; converter bytes para pontos de código é *decodificar*. Veja o Exemplo 41.

Exemplo 41. Codificando e decodificando

```
>>> s = 'caf '  
>>> len(s) ①  
4  
>>> b = s.encode('utf8') ②  
>>> b  
b'caf\xc3\xa9' ③
```

```
>>> len(b) ④
5
>>> b.decode('utf8') ⑤
'café'
```

- ① A str 'café' tem quatro caracteres Unicode.
- ② Codifica str para bytes usando a codificação UTF-8.
- ③ bytes literais são prefixados com um b.
- ④ bytes b tem cinco bytes (o ponto de código para "é" é codificado com dois bytes em UTF-8).
- ⑤ Decodifica bytes para str usando a codificação UTF-8.



Um jeito fácil de memorizar a distinção entre `.decode()` e `.encode()` é se convencer que sequências de bytes podem ser enigmáticos dumps de código de máquina, ao passo que objetos str Unicode são texto "humano". Daí que faz sentido *decodificar* bytes em str, para obter texto legível por seres humanos, e *codificar* str em bytes, para armazenamento ou transmissão.

Apesar do str de Python 3 ser praticamente igual ao tipo unicode de Python 2 com um novo nome, o bytes de Python 3 não é meramente o velho str renomeado, e há também o tipo estreitamente relacionado bytearray. Então vale a pena examinar os tipos de sequências binárias antes de avançar para questões de codificação/decodificação.

4.3. Os fundamentos do byte

Os novos tipos de sequências binárias são diferentes do str de Python 2 em vários aspectos. A primeira coisa importante é que existem dois tipos embutidos básicos de sequências binárias: o tipo imutável bytes, introduzido no Python 3, e o tipo mutável bytearray, introduzido há tempos, no Python 2.6^[2]. A documentação de Python algumas vezes usa o termo genérico "byte string" (*string de bytes*, na documentação em português) para se referir a bytes e bytearray.

Cada item em bytes ou bytearray é um inteiro entre 0 e 255, e não uma string de um caractere, como no str de Python 2. Entretanto, uma fatia de uma sequência binária sempre produz uma sequência binária do mesmo tipo—incluindo fatias de tamanho 1. Veja o Exemplo 42.

Exemplo 42. Uma sequência de cinco bytes, como bytes e como bytearray

```
>>> cafe = bytes('café', encoding='utf_8') ①
>>> cafe
b'caf\xc3\xa9'
>>> cafe[0] ②
99
>>> cafe[:1] ③
b'c'
>>> cafe_arr = bytearray(cafe)
>>> cafe_arr ④
bytearray(b'caf\xc3\xa9')
>>> cafe_arr[-1:] ⑤
```

```
bytearray(b'\xa9')
```

- ① bytes pode ser criado a partir de uma str, dada uma codificação.
- ② Cada item é um inteiro em range(256).
- ③ Fatiadas de bytes também são bytes—mesmo fatiadas de um único byte.
- ④ A sintaxe literal para bytearray é bytearray(...) com um literal bytes como argumento.
- ⑤ Uma fatia de bytearray também é um bytearray.



O fato de my_bytes[0] obter um int mas my_bytes[:1] devolver uma sequência de bytes de tamanho 1 só é surpreendente porque estamos acostumados com o tipo str de Python, onde s[0] == s[:1]. Para todos os outros tipos de sequência no Python, um item não é o mesmo que uma fatia de tamanho 1.

Apesar de sequências binárias serem na verdade sequências de inteiros, sua notação literal reflete o fato delas frequentemente embutirem texto ASCII. Assim, quatro formas diferentes de apresentação são utilizadas, dependendo do valor de cada byte:

- Para bytes com código decimais de 32 a 126—do espaço ao ~ (til)—é usado o próprio caractere ASCII.
- Para os bytes correspondendo a tab, quebra de linha, carriage return (CR) e \, são usadas as sequências de escape \t, \n, \r, e \\.
- Se os dois delimitadores de string, ' e ", aparecem na sequência de bytes, a sequência inteira é delimitada com ', e qualquer ' dentro da sequência é precedida do caractere de escape, assim \'.^[3]
- Para qualquer outro valor do byte, é usada uma sequência de escape hexadecimal (por exemplo, \x00 é o byte nulo).

É por isso que no Exemplo 42 vemos b'caf\xc3\xa9': os primeiros três bytes, b'caf', estão na faixa de impressão do ASCII, ao contrário dos dois últimos.

Tanto bytes quanto bytearray suportam todos os métodos de str, exceto aqueles relacionados à formatação (format, format_map) e aqueles que dependem de dados Unicode, incluindo casefold, isdecimal, isidentifier, isnumeric, isprintable, e encode. Isso significa que você pode usar os métodos conhecidos de string, como endswith, replace, strip, translate, upper e dezenas de outros, com sequências binárias—mas com argumentos bytes em vez de str. Além disso, as funções de expressões regulares no módulo re também funcionam com sequências binárias, se a regex for compilada a partir de uma sequência binária ao invés de uma str. Desde Python 3.5, o operador % voltou a funcionar com sequências binárias.^[4]

As sequências binárias têm um método de classe que str não possui, chamado fromhex, que cria uma sequência binária a partir da análise de pares de dígitos hexadecimais, separados opcionalmente por espaços:

```
>>> bytes.fromhex('31 4B CE A9')
```

```
b'1K\xce\x99'
```

As outras formas de criar instâncias de bytes ou bytearray são chamadas a seus construtores com:

- Uma str e um argumento nomeado encoding
- Um iterável que forneça itens com valores entre 0 e 255
- Um objeto que implemente o protocolo de buffer (por exemplo bytes, bytearray, memoryview, array.array), que copia os bytes do objeto fonte para a recém-criada sequência binária



Até Python 3.5, era possível chamar bytes ou bytearray com um único inteiro, para criar uma sequência daquele tamanho inicializada com bytes nulos. Essa assinatura foi descontinuada no Python 3.5 e removida no Python 3.6. Veja a PEP 467—Minor API improvements for binary sequences (*Pequenas melhorias na API para sequências binárias*) (EN) [<https://fpy.li/pep467>].

Criar uma sequência binária a partir de um objeto tipo buffer é uma operação de baixo nível que pode envolver conversão de tipos. Veja uma demonstração no Exemplo 43.

Exemplo 43. Inicializando bytes a partir de dados brutos de um array

```
>>> import array  
>>> numbers = array.array('h', [-2, -1, 0, 1, 2]) ①  
>>> octets = bytes(numbers) ②  
>>> octets  
b'\xfe\xff\xff\x00\x00\x01\x00\x02\x00' ③
```

① O typecode 'h' cria um array de *short integers* (inteiros de 16 bits).

② octets mantém uma cópia dos bytes que compõem numbers.

③ Esses são os 10 bytes que representam os 5 inteiros pequenos.

Criar um objeto bytes ou bytearray a partir de qualquer fonte tipo buffer vai sempre copiar os bytes. Já objetos memoryview permitem compartilhar memória entre estruturas de dados binários, como vimos na Seção 2.10.2.

Após essa exploração básica dos tipos de sequências de bytes de Python, vamos ver como eles são convertidos de e para strings.

4.4. Codificadores/Decodificadores básicos

A distribuição de Python inclui mais de 100 *codecs* (encoders/decoders, _codificadores/_decodificadores) para conversão de texto para bytes e vice-versa. Cada codec tem um nome, como 'utf_8', e muitas vezes apelidos, como 'utf8', 'utf-8', e 'U8', que você pode usar como o argumento de codificação em funções como open(), str.encode(), bytes.decode(), e assim por diante. O Exemplo 44 mostra o mesmo texto codificado como três sequências de bytes diferentes.

Exemplo 44. A string "El Niño" codificada com três codecs, gerando sequências de bytes muito diferentes

```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print(codec, 'El Niño'.encode(codec), sep='\t')
...
latin_1 b'El Ni\xf1o'
utf_8  b'El Ni\xc3\xb1o'
utf_16 b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

A Figura 10 mostra um conjunto de codecs gerando bytes a partir de caracteres como a letra "A" e o símbolo musical da clave de sol. Observe que as últimas três codificações tem bytes múltiplos e tamanho variável.

char.	code point	ascii	latin1	cp1252	cp437	gb2312	utf-8	utf-16le
A	U+0041	41	41	41	41	41	41	41 00
¿	U+00BF	*	BF	BF	A8	*	C2 BF	BF 00
Ã	U+00C3	*	C3	C3	*	*	C3 83	C3 00
á	U+00E1	*	E1	E1	A0	A8 A2	C3 A1	E1 00
Ω	U+03A9	*	*	*	EA	A6 B8	CE A9	A9 03
€	U+06BF	*	*	*	*	*	DA BF	BF 06
“	U+201C	*	*	93	*	A1 B0	E2 80 9C	1C 20
€	U+20AC	*	*	80	*	*	E2 82 AC	AC 20
Γ	U+250C	*	*	*	DA	A9 B0	E2 94 8C	0C 25
氣	U+6C14	*	*	*	*	C6 F8	E6 B0 94	14 6C
氣	U+6C23	*	*	*	*	*	E6 B0 A3	23 6C
♪	U+1D11E	*	*	*	*	*	F0 9D 84 9E	34 D8 1E DD

Figura 10. Doze caracteres, seus pontos de código, e sua representação binária (em hexadecimal) em 7 codificações diferentes (asteriscos indicam que o caractere não pode ser representado naquela codificação).

Todos aqueles asteriscos na Figura 10 deixam claro que algumas codificações, como o ASCII e mesmo o multi-byte GB2312, não conseguem representar todos os caracteres Unicode. As codificações UTF, por outro lado, foram projetadas para lidar com todos os pontos de código possíveis.

Escolhi as codificações apresentadas na Figura 10 como uma amostra representativa:

latin1 a.k.a. iso8859_1

Importante por ser a base de outras codificações, tal como a cp1252 e o próprio Unicode (observe que os valores binários do latin1 aparecem nos bytes do cp1252 e nos pontos de código).

cp1252

Um superconjunto útil de latin1, criado pela Microsoft, acrescentando símbolos convenientes como as aspas curvas e o € (euro); alguns aplicativos de Windows chamam essa codificação de "ANSI", mas ela nunca foi um padrão ANSI real.

cp437

O conjunto de caracteres original do IBM PC, com caracteres de desenho de caixas. Incompatível

com o latin1, que surgiu depois.

gb2312

Padrão legado para codificar ideogramas chineses simplificados usados na República da China; uma das várias codificações criadas para línguas asiáticas.

utf-8

A codificação de 8 bits mais comum na web. Em julho de 2021, o W³ Techs: Usage statistics of character encodings for websites [https://fpy.li/4-5] informa que 97% dos sites usam UTF-8, um grande avanço sobre os 81,4% de setembro de 2014, quando escrevi este capítulo na primeira edição.

utf-16le

Uma forma do esquema de codificação UTF de 16 bits; todas as codificações UTF-16 suportam pontos de código acima de U+FFFF, através de sequências de escape chamadas "pares substitutos".



A UTF-16 sucedeu a codificação de 16 bits original do Unicode 1.0—a UCS-2—há muito tempo, em 1996. Mas a UCS-2 obsoleta ainda é usada em muitos sistemas, apesar de ter sido descontinuada por suportar apenas pontos de código até U+FFFF. Em 2021, mas de 57% dos pontos de código alocados ficam acima de U+FFFF, incluindo os importantíssimos emojis.

Após completar essa revisão das codificações mais comuns, vamos agora tratar das questões relativas a operações de codificação e decodificação.

4.5. Entendendo os problemas de codificação/decodificação

Apesar de existir uma exceção genérica, `UnicodeError`, o erro relatado pelo Python em geral é mais específico: ou é um `UnicodeEncodeError` (ao converter uma `str` para sequências binárias) ou é um `UnicodeDecodeError` (ao ler uma sequência binária para uma `str`). Carregar módulos de Python também pode gerar um `SyntaxError`, quando a codificação da fonte for inesperada. Vamos ver como tratar todos esses erros nas próximas seções.



A primeira coisa a observar quando aparece um erro de Unicode é o tipo exato da exceção. É um `UnicodeEncodeError`, um `UnicodeDecodeError`, ou algum outro erro (por exemplo, `SyntaxError`) mencionando um problema de codificação? Para resolver o problema, você primeiro precisa entendê-lo.

4.5.1. Tratando o `UnicodeEncodeError`

A maioria dos codecs não-UTF entendem apenas um pequeno subconjunto dos caracteres Unicode. Ao converter texto para bytes, um `UnicodeEncodeError` será gerado se um caractere não estiver definido na codificação alvo, a menos que seja fornecido um tratamento especial, passando um argumento `errors` para o método ou função de codificação. O comportamento para tratamento de erro é apresentado no Exemplo 45.

Exemplo 45. Encoding to bytes: success and error handling

```
>>> city = 'São Paulo'  
>>> city.encode('utf_8') ①  
b'S\xc3\xa3o Paulo'  
>>> city.encode('utf_16')  
b'\xff\xfeS\x00\xe3\x00\x00 \x00P\x00a\x00u\x00l\x00o\x00'  
>>> city.encode('iso8859_1') ②  
b'S\xe3o Paulo'  
>>> city.encode('cp437') ③  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    File "/.../lib/python3.4/encodings/cp437.py", line 12, in encode  
        return codecs.charmap_encode(input,errors,encoding_map)  
UnicodeEncodeError: 'charmap' codec can't encode character '\xe3' in  
position 1: character maps to <undefined>  
>>> city.encode('cp437', errors='ignore') ④  
b'So Paulo'  
>>> city.encode('cp437', errors='replace') ⑤  
b'S?o Paulo'  
>>> city.encode('cp437', errors='xmlcharrefreplace') ⑥  
b'S&#227;o Paulo'
```

① As codificações UTF lidam com qualquer str

② iso8859_1 também funciona com a string 'São Paulo'.

③ cp437 não consegue codificar o 'ã' ("a" com til). O método default de tratamento de erro ('strict') gera um UnicodeEncodeError.

④ O método de tratamento errors='ignore' pula os caracteres que não podem ser codificados; isso normalmente é uma péssima ideia, levando a perda silenciosa de informação.

⑤ Ao codificar, errors='replace' substitui os caracteres não-codificáveis por um '?'; aqui também há perda de informação, mas é mais fácil perceber que algo está faltando.

⑥ 'xmlcharrefreplace' substitui os caracteres não-codificáveis por uma entidade XML. Se você não pode usar UTF e não pode perder informação, essa é a única opção.



O tratamento de erros de codecs é extensível. Você pode registrar novas strings para o argumento errors passando um nome e uma função de tratamento de erros para a função codecs.register_error. Veja documentação de codecs.register_error [<https://fpy.li/39>] (EN).

O ASCII é um subconjunto comum a todas as codificações que conheço, então a codificação deveria sempre funcionar se o texto for composto exclusivamente por caracteres ASCII. Python 3.7 trouxe um novo método booleano, str.isascii() [<https://fpy.li/4-7>], para verificar se seu texto Unicode é 100% ASCII. Se for, você deve ser capaz de codificá-lo para bytes em qualquer codificação sem gerar um UnicodeEncodeError.

4.5.2. Tratando o UnicodeDecodeError

Nem todo byte contém um caractere ASCII válido, e nem toda sequência de bytes é um texto corretamente codificado em UTF-8 ou UTF-16; assim, se você presumir uma dessas codificações ao converter um sequência binária para texto, pode receber um `UnicodeDecodeError`, se bytes inesperados forem encontrados.

Por outro lado, várias codificações de 8 bits antigas, como a '`cp1252`', a '`iso8859_1`' e a '`koi8_r`' são capazes de decodificar qualquer série de bytes, incluindo ruído aleatório, sem reportar qualquer erro. Portanto, se seu programa presumir a codificação de 8 bits errada, ele vai decodificar lixo silenciosamente.



Caracteres truncados ou distorcidos são conhecidos como "gremlins" ou "mojibake" (モジベーク—"texto modificado" em japonês).

O Exemplo 46 ilustra a forma como o uso do codec errado pode produzir gremlins ou um `UnicodeDecodeError`.

Exemplo 46. Decodificando de str para bytes: sucesso e tratamento de erro

```
>>> octets = b'Montr\xe9al' ①
>>> octets.decode('cp1252') ②
'Montréal'
>>> octets.decode('iso8859_7') ③
'Montr\x91al'
>>> octets.decode('koi8_r') ④
'Montr\x91al'
>>> octets.decode('utf_8') ⑤
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 5:
invalid continuation byte
>>> octets.decode('utf_8', errors='replace') ⑥
'Montr\x91al'
```

① A palavra "Montréal" codificada em latin1; '\xe9' é o byte para "é".

② Decodificar com Windows 1252 funciona, pois esse codec é um superconjunto de latin1.

③ ISO-8859-7 foi projetado para a língua grega, então o byte '\xe9' é interpretado incorretamente, mas nenhum erro é gerado.

④ KOI8-R é foi projetado para o russo. Agora '\xe9' significa a letra "И" do alfabeto cirílico.

⑤ O codec 'utf_8' detecta que octets não é UTF-8 válido, e gera um `UnicodeDecodeError`.

⑥ Usando 'replace' para tratamento de erro, o '\xe9' é substituído por "\ufffd" (ponto de código #U+FFFFD), o caractere oficial do Unicode chamado REPLACEMENT CHARACTER, criado exatamente para representar caracteres desconhecidos.

4.5.3. SyntaxError ao carregar módulos com codificação inesperada

UTF-8 é a codificação default para código-fonte no Python 3, da mesma forma que ASCII era o default no Python 2. Se você carregar um módulo `.py` contendo dados que não estejam em UTF-8, sem declaração codificação, receberá uma mensagem como essa:

```
SyntaxError: Non-UTF-8 code starting with '\xe1' in file ola.py on line
1, but no encoding declared; see https://python.org/dev/peps/pep-0263/
for details
```

Como o UTF-8 está amplamente instalado em sistemas GNU/Linux e macOS, um cenário onde isso tem mais chance de ocorrer é na abertura de um arquivo `.py` criado no Windows, com `cp1252`. Observe que esse erro ocorre mesmo no Python para Windows, pois a codificação default para fontes de Python 3 é UTF-8 em todas as plataformas.

Para resolver esse problema, acrescente o comentário mágico `coding` no início do arquivo, como no Exemplo 47.

Exemplo 47. 'ola.py': um "Hello, World!" em português

```
# coding: cp1252

print('Olá, Mundo!')
```



Agora que o código fonte de Python 3 não está mais limitado ao ASCII, e por default usa a excelente codificação UTF-8, a melhor "solução" para código fonte em codificações antigas como '`cp1252`' é converter tudo para UTF-8 de uma vez, e não se preocupar com os comentários `coding`. Se seu editor não suporta UTF-8, é hora de trocar de editor.

Suponha que você tem um arquivo de texto, seja ele código-fonte ou poesia, mas não sabe qual codificação foi usada. Como detectar a codificação correta? Respostas na próxima seção.

4.5.4. Como descobrir a codificação de uma sequência de bytes

Como descobrir a codificação de uma sequência de bytes? Resposta curta: não é possível. Você precisa ser informado.

Alguns protocolos de comunicação e formatos de arquivo, como o HTTP e o XML, contêm cabeçalhos que nos dizem explicitamente como o conteúdo está codificado. Você pode ter certeza que algumas sequências de bytes não estão em ASCII, pois elas contêm bytes com valores acima de 127, e o modo como o UTF-8 e o UTF-16 são construídos também limita as sequências de bytes possíveis.

O hack do Leo para adivinhar uma decodificação UTF-8

(Os próximos parágrafos vieram de uma nota escrita pelo revisor técnico Leonardo Rochael

no rascunho desse livro.)

Pela forma como o UTF-8 foi projetado, é quase impossível que uma sequência aleatória de bytes, ou mesmo uma sequência não-aleatória de bytes de uma codificação diferente do UTF-8, seja accidentalmente decodificada como lixo no UTF-8, ao invés de gerar um `UnicodeDecodeError`.

As razões para isso são que as sequências de escape do UTF-8 nunca usam caracteres ASCII, e tais sequências de escape tem padrões de bits que tornam muito difícil que dados aleatórios sejam UTF-8 válido por acidente.

Portanto, se você consegue decodificar alguns bytes contendo códigos > 127 como UTF-8, a maior probabilidade é de sequência estar em UTF-8.

Trabalhando com os serviços online brasileiros, alguns baseados em back-ends antigos, ocasionalmente precisei implementar uma estratégia de decodificação que tentava decodificar via UTF-8 e tratava um `UnicodeDecodeError` decodificando via `cp1252`. Uma estratégia feia, mas efetiva.

Entretanto, considerando que as linguagens humanas também tem suas regras e restrições, uma vez que você supõe que uma série de bytes é um *texto humano simples*, pode ser possível intuir sua codificação usando heurística e estatística. Por exemplo, se bytes com valor `b'\x00'` bytes forem comuns, é provável que seja uma codificação de 16 ou 32 bits, e não um esquema de 8 bits, pois caracteres nulos em texto simples são erros. Quando a sequência de bytes `'b'\x20\x00'` aparece com frequência, é mais provável que esse seja o caractere de espaço (U+0020) na codificação UTF-16LE, e não o obscuro caractere U+2000 (EN QUAD)—seja lá o que for isso.

É assim que o pacote "Chardet—The Universal Character Encoding Detector (*Chardet—O Detector Universal de Codificações de Caracteres*)" [<https://fpy.li/4-8>] trabalha para descobrir cada uma das mais de 30 codificações suportadas. *Chardet* é uma biblioteca Python que pode ser usada em seus programas, mas que também inclui um utilitário de linha de comando, `chardetect`. Veja como ele analisa o código fonte desse capítulo:

```
$ chardetect 04-text-byte.asciidoc
04-text-byte.asciidoc: utf-8 with confidence 0.99
```

Apesar de sequências binárias de texto codificado normalmente não trazerem dicas sobre sua codificação, os formatos UTF podem usar um marcador de ordem dos bytes. Isso é explicado a seguir.

4.5.5. BOM: um gremlin útil

No Exemplo 44, você pode ter notado um par de bytes extra no início de uma sequência codificada em UTF-16. Aqui estão eles novamente:

```
>>> u16 = 'El Niño'.encode('utf_16')
>>> u16
```

```
b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

Os bytes são `b'\xff\xfe'`. Isso é um *BOM*—sigla para byte-order mark (marcador de ordem de bytes)—indicando a ordenação de bytes "little-endian" da CPU Intel onde a codificação foi realizada.

Em uma máquina *little-endian*, para cada ponto de código, o byte menos significativo aparece primeiro: a letra 'E', ponto de código U+0045 (decimal 69), é codificado nas posições 2 e 3 dos bytes como 69 e 0:

```
>>> list(u16)
[255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
```

Em uma CPU *big-endian*, a codificação seria invertida; 'E' seria codificado como 0 e 69.

Para evitar confusão, a codificação UTF-16 precede o texto a ser codificado com o caractere especial invisível ZERO WIDTH NO-BREAK SPACE (U+FEFF). Em um sistema *little-endian*, isso é codificado como `b'\xff\xfe'` (decimais 255, 254). Como, por design, não existe um caractere U+FFFE em Unicode, a sequência de bytes `b'\xff\xfe'` tem que ser o ZERO WIDTH NO-BREAK SPACE em uma codificação *little-endian*, e então o codec sabe qual ordenação de bytes usar.

Há uma variante do UTF-16—o UTF-16LE—que é explicitamente *little-endian*, e outra que é explicitamente *big-endian*, o UTF-16BE. Se você usá-los, um BOM não será gerado:

```
>>> u16le = 'El Niño'.encode('utf_16le')
>>> list(u16le)
[69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
>>> u16be = 'El Niño'.encode('utf_16be')
>>> list(u16be)
[0, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111]
```

Se o BOM estiver presente, supõe-se que ele será filtrado pelo codec UTF-16, então recebemos apenas o conteúdo textual efetivo do arquivo, sem o ZERO WIDTH NO-BREAK SPACE inicial.

O padrão Unicode diz que se um arquivo é UTF-16 e não tem um BOM, deve-se presumir que ele é UTF-16BE (*big-endian*). Entretanto, a arquitetura x86 da Intel é *little-endian*, daí que há uma grande quantidade de UTF-16 *little-endian* e sem BOM no mundo.

Toda essa questão de ordenação dos bytes (*endianness*) só afeta codificações que usam palavras de máquina com mais de um byte, como UTF-16 e UTF-32. Uma grande vantagem do UTF-8 é produzir a mesma sequência independente da ordenação dos bytes, então um BOM não é necessário. No entanto, algumas aplicações Windows (em especial o Notepad) mesmo assim acrescentam o BOM a arquivos UTF-8—e o Excel depende do BOM para detectar um arquivo UTF-8, caso contrário ele presume que o conteúdo está codificado com uma página de código do Windows. Essa codificação UTF-8 com BOM é chamada UTF-8-SIG no registro de codecs de Python. O caractere U+FEFF codificado em UTF-8-SIG é a sequência de três bytes `b'\xef\xbb\xbf'`. Então, se um arquivo começa com aqueles três bytes, é provavelmente um arquivo UTF-8 com um BOM.

A dica de Caleb sobre o UTF-8-SIG

Caleb Hattingh—um dos revisores técnicos—sugere sempre usar o codec UTF-8-SIG para ler arquivos UTF-8. Isso é inofensivo, pois o UTF-8-SIG lê corretamente arquivos com ou sem um BOM, e não devolve o BOM propriamente dito. Para escrever arquivos, recomendo usar UTF-8, para interoperabilidade integral. Por exemplo, scripts Python podem ser tornados executáveis em sistemas Unix, se começarem com o comentário: `#!/usr/bin/env python3`. Os dois primeiros bytes do arquivo precisam ser `b'#!'` para isso funcionar, mas o BOM quebra essa convenção. Se você tem o requerimento específico de exportar dados para aplicativos que precisam do BOM, use o UTF-8-SIG, mas esteja ciente do que diz a documentação sobre codecs [<https://fpy.li/3a>] (EN) de Python: "No UTF-8, o uso do BOM é desencorajado e, em geral, deve ser evitado."



Vamos agora ver como tratar arquivos de texto no Python 3.

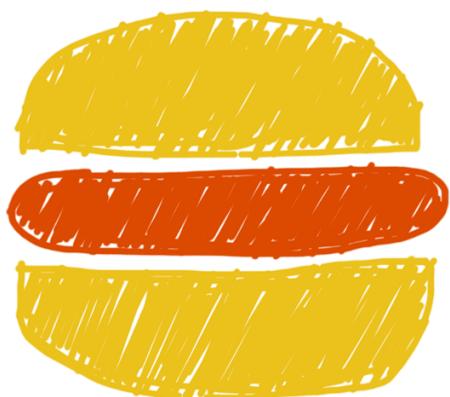
4.6. Processando arquivos de texto

A melhor prática para lidar com E/S de texto é o "Sanduíche de Unicode" (*Unicode sandwich*) (Figura 11).^[5] Isso significa que os bytes devem ser decodificados para str o mais cedo possível na entrada (por exemplo, ao abrir um arquivo para leitura). O "recheio" do sanduíche é a lógica do negócio de seu programa, onde o tratamento do texto é feito somente com objetos str. Evite codificar ou decodificar em diferentes estágios do processamento. Na saída, as str são codificadas para bytes o mais tarde possível. A maioria dos frameworks web funciona assim, e raramente tocamos em bytes ao usá-los. No Django, por exemplo, suas views devem produzir str em Unicode; o próprio Django se encarrega de codificar a resposta para bytes, usando UTF-8 como default.

Python 3 torna mais fácil seguir o conselho do sanduíche de Unicode, pois a função `open()` decodifica na leitura e a codifica na escrita, ao lidar com arquivos em modo texto. Dessa forma, tudo que você recebe de `my_file.read()` e passa para `my_file.write(text)` são objetos str.

Assim, usar arquivos de texto é aparentemente simples. Mas se você confiar nas codificações default, pode acabar levando uma mordida.

The Unicode sandwich



`bytes → str`

Decode bytes on input,

`100% str`

process text only,

`str → bytes`

encode text on output.

Figura 11. O sanduíche de Unicode: a melhor prática para processamento de texto.

Observe a sessão de console no Exemplo 48. Você consegue ver o erro?

Exemplo 48. Uma questão de plataforma na codificação (você pode ou não ver o problema se tentar isso na sua máquina)

```
>>> open('cafe.txt', 'w', encoding='utf_8').write('café')
4
>>> open('cafe.txt').read()
'cafÃ©'
```

O erro: especifiquei a codificação UTF-8 ao escrever o arquivo, mas não fiz isso na leitura, então Python assumiu a codificação default do Windows—página de código 1252—e os bytes finais foram decodificados como os caracteres 'Ã©' ao invés de 'é'.

Executei o Exemplo 48 no Python 3.8.1, 64 bits, no Windows 10 (build 18363). Os mesmos comandos rodando em um GNU/Linux ou um macOS recentes funcionam perfeitamente, pois a codificação default desses sistemas é UTF-8, dando a falsa impressão que tudo está bem. Se o argumento de codificação fosse omitido ao abrir o arquivo para escrita, a codificação default do *locale* seria usada, e poderíamos ler o arquivo corretamente usando a mesma codificação. Mas aí o script geraria arquivos com bytes diferentes dependendo da plataforma, ou mesmo das configurações do *locale* na mesma plataforma, criando problemas de compatibilidade.



Código que precisa rodar em múltiplas máquinas ou múltiplas ocasiões não deve depender de defaults de codificação. Sempre passe um argumento `encoding=` explícito ao abrir arquivos de texto, pois o default pode mudar de uma máquina para outra ou de um dia para o outro.

Um detalhe curioso no Exemplo 48 é que a função `write` na primeira instrução informa que foram escritos quatro caracteres, mas na linha seguinte são lidos cinco caracteres. O Exemplo 49 é uma versão estendida do Exemplo 48, e explica esse e outros detalhes.

Exemplo 49. Uma inspeção mais atenta do Exemplo 48 rodando no Windows revela o bug e a solução do problema

```
>>> fp = open('cafe.txt', 'w', encoding='utf_8')
>>> fp ①
<_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf_8'>
>>> fp.write('café') ②
4
>>> fp.close()
>>> import os
>>> os.stat('cafe.txt').st_size ③
5
>>> fp2 = open('cafe.txt')
>>> fp2 ④
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>
>>> fp2.encoding ⑤
'cp1252'
>>> fp2.read() ⑥
```

```
'caf o'
>>> fp3 = open('cafe.txt', encoding='utf_8') ⑦
>>> fp3
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf_8'>
>>> fp3.read() ⑧
'caf '
>>> fp4 = open('cafe.txt', 'rb') ⑨
>>> fp4 ⑩
<_io.BufferedReader name='cafe.txt'>
>>> fp4.read() ⑪
b'caf\xc3\xa9'
```

- ① Por default, `open` usa o modo texto e devolve um objeto `TextIOWrapper` com uma codificação específica.
- ② O método `write` de um `TextIOWrapper` devolve o número de caracteres Unicode escritos.
- ③ `os.stat` diz que o arquivo tem 5 bytes; o UTF-8 codifica 'é' com 2 bytes, `0xc3` e `0xa9`.
- ④ Abrir um arquivo de texto sem uma codificação explícita devolve um `TextIOWrapper` com a codificação configurada para um default do locale.
- ⑤ Um objeto `TextIOWrapper` tem um atributo de codificação que pode ser inspecionado: neste caso, `cp1252`.
- ⑥ Na codificação `cp1252` do Windows, o byte `0xc3` é um "Ã" (A maiúsculo com til), e `0xa9` é o símbolo de copyright.
- ⑦ Abrindo o mesmo arquivo com a codificação correta.
- ⑧ O resultado esperado: os mesmos quatro caracteres Unicode para 'caf '.
- ⑨ A flag '`rb`' abre um arquivo para leitura em modo binário.
- ⑩ O objeto devolvido é um `BufferedReader`, e não um `TextIOWrapper`.
- ⑪ Ler do arquivo obtém bytes, como esperado.



N o abra arquivos de texto no modo bin rio, a menos que seja necess rio analisar o conte do do arquivo para determinar sua codifica o—e mesmo assim, voc  deveria estar usando o Chardet em vez de reinventar a roda (veja a Se o 4.5.4).

Programas comuns s o deveriam usar o modo bin rio para abrir arquivos bin rios, como arquivos de imagens raster ou bitmaps.

O problema no Exemplo 49 vem de se confiar numa configura o default ao se abrir um arquivo de texto. H  v rias fontes de tais defaults, como mostra a pr oxima se o.

4.6.1. Cuidado com os defaults de codifica o

V rias configura es afetam os defaults de codifica o para E/S no Python. Veja o script `default_encodings.py` script no Exemplo 50.

Exemplo 50. Explorando os defaults de codificação

```
import locale
import sys

expressions = """
    locale.getpreferredencoding()
    type(my_file)
    my_file.encoding
    sys.stdout.isatty()
    sys.stdout.encoding
    sys.stdin.isatty()
    sys.stdin.encoding
    sys.stderr.isatty()
    sys.stderr.encoding
    sys.getdefaultencoding()
    sys.getfilesystemencoding()
"""

my_file = open('dummy', 'w')

for expression in expressions.split():
    value = eval(expression)
    print(f'{expression:>30} -> {value!r}')
```

A saída do Exemplo 50 no GNU/Linux (Ubuntu 14.04 a 19.10) e no macOS (10.9 a 10.14) é idêntica, mostrando que UTF-8 é usado em toda parte nesses sistemas:

```
$ python3 default_encodings.py
locale.getpreferredencoding() -> 'UTF-8'
    type(my_file) -> <class '_io.TextIOWrapper'>
    my_file.encoding -> 'UTF-8'
    sys.stdout.isatty() -> True
    sys.stdout.encoding -> 'utf-8'
    sys.stdin.isatty() -> True
    sys.stdin.encoding -> 'utf-8'
    sys.stderr.isatty() -> True
    sys.stderr.encoding -> 'utf-8'
    sys.getdefaultencoding() -> 'utf-8'
    sys.getfilesystemencoding() -> 'utf-8'
```

No Windows, porém, a saída é o Exemplo 51.

Exemplo 51. Codificações default, no PowerShell do Windows 10 (a saída é a mesma no cmd.exe)

```
> chcp ①
Active code page: 437
> python default_encodings.py ②
locale.getpreferredencoding() -> 'cp1252' ③
```

```

        type(my_file) -> <class '_io.TextIOWrapper'>
    my_file.encoding -> 'cp1252' ④
    sys.stdout.isatty() -> True ⑤
    sys.stdout.encoding -> 'utf-8' ⑥
    sys.stdin.isatty() -> True
    sys.stdin.encoding -> 'utf-8'
    sys.stderr.isatty() -> True
    sys.stderr.encoding -> 'utf-8'
    sys.getdefaultencoding() -> 'utf-8'
    sys.getfilesystemencoding() -> 'utf-8'

```

- ① chcp mostra a página de código ativa para o console: 437.
- ② Executando `default_encodings.py`, com a saída direcionada para o console.
- ③ `locale.getpreferredencoding()` é a configuração mais importante.
- ④ Arquivos de texto usam ``locale.getpreferredencoding()`` por default.
- ⑤ A saída está direcionada para o console, então `sys.stdout.isatty()` é True.
- ⑥ Agora, `sys.stdout.encoding` não é a mesma que a página de código informada por chcp!

O suporte a Unicode no próprio Windows e no Python para Windows melhorou desde que escrevi a primeira edição deste livro. O Exemplo 51 costumava informar quatro codificações diferentes no Python 3.4 rodando no Windows 7. As codificações para `stdout`, `stdin`, e `stderr` costumavam ser iguais à da página de código ativa informada pelo comando `chcp`, mas agora são todas utf-8, graças à PEP 528—Change Windows console encoding to UTF-8 (*Mudar a codificação do console no Windows para UTF-8*) [<https://fpy.li/pep528>] (EN), implementada no Python 3.6, e ao suporte a Unicode no PowerShell do `cmd.exe` (desde o Windows 1809, de outubro de 2018).^[6] É esquisito que o `chcp` e o `sys.stdout.encoding` reportem coisas diferentes quando o `stdout` está escrevendo no console, mas é ótimo podermos agora escrever strings Unicode sem erros de codificação no Windows—a menos que o usuário redirecione a saída para um arquivo, como veremos adiante. Isso não significa que todos os seus emojis favoritos vão aparecer: isso também depende da fonte usada pelo console.

Outra mudança foi a PEP 529—Change Windows filesystem encoding to UTF-8 (*Mudar a codificação do sistema de arquivos do Windows para UTF-8*) [<https://fpy.li/pep529>], também implementada no Python 3.6, que mudou a codificação do sistema de arquivos (usada para representar nomes de diretórios e de arquivos), da codificação proprietária MBCS da Microsoft para UTF-8.

Entretanto, se a saída do Exemplo 50 for redirecionada para um arquivo, assim...

```
Z:\>python default_encodings.py > encodings.log
```

...aí o valor de `sys.stdout.isatty()` se torna `False`, e `sys.stdout.encoding` é determinado por `locale.getpreferredencoding()`, 'cp1252' naquela máquina—mas `sys.stdin.encoding` e `sys.stderr.encoding` seguem como utf-8.



No Exemplo 52, usei a expressão de escape '`\N{}`' para literais Unicode, escrevendo o nome oficial do caractere dentro do `\N{}`. Isso é bastante prolixo, mas explícito e seguro: Python gera um `SyntaxError` se o nome não existir—bem melhor

que escrever um número hexadecimal que pode estar errado, mas isso só será descoberto mais tarde. De qualquer forma, você provavelmente vai querer escrever um comentário explicando os códigos numéricos dos caracteres, então a verbosidade do \N{} é fácil de aceitar.

Isso significa que um script como o Exemplo 52 funciona quando está escrevendo no console, mas pode falhar quando a saída é redirecionada para um arquivo.

Exemplo 52. stdout_check.py

```
import sys
from unicodedata import name

print(sys.version)
print()
print('sys.stdout.isatty():', sys.stdout.isatty())
print('sys.stdout.encoding:', sys.stdout.encoding)
print()

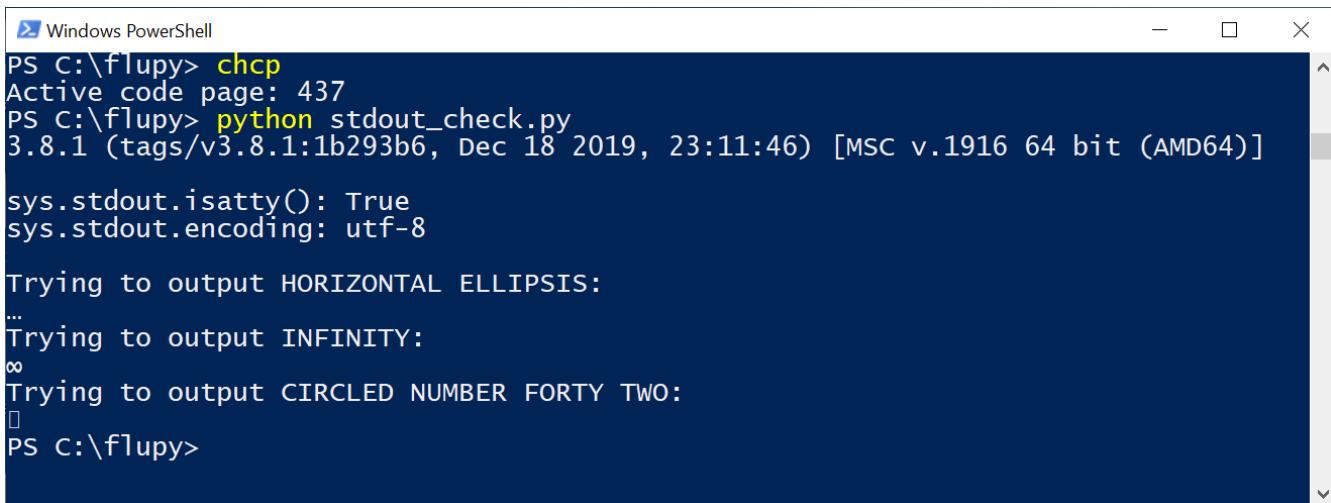
test_chars = [
    '\N{HORIZONTAL ELLIPSIS}',      # exists in cp1252, not in cp437
    '\N{INFINITY}',                # exists in cp437, not in cp1252
    '\N{CIRCLED NUMBER FORTY TWO}', # not in cp437 or in cp1252
]

for char in test_chars:
    print(f'Trying to output {name(char)}:')
    print(char)
```

O Exemplo 52 mostra o resultado de uma chamada a `sys.stdout.isatty()`, o valor de `sys.stdout.encoding`, e esses três caracteres:

- '...' HORIZONTAL ELLIPSIS—existe no CP 1252 mas não no CP 437.
- 'ꝑ' INFINITY—existe no CP 437 mas não no CP 1252.
- 'ꝑ' CIRCLED NUMBER FORTY TWO—não existe nem no CP 1252 nem no CP 437.

Quando executo `stdout_check.py` no PowerShell ou no `cmd.exe`, funciona como visto na Figura 12.



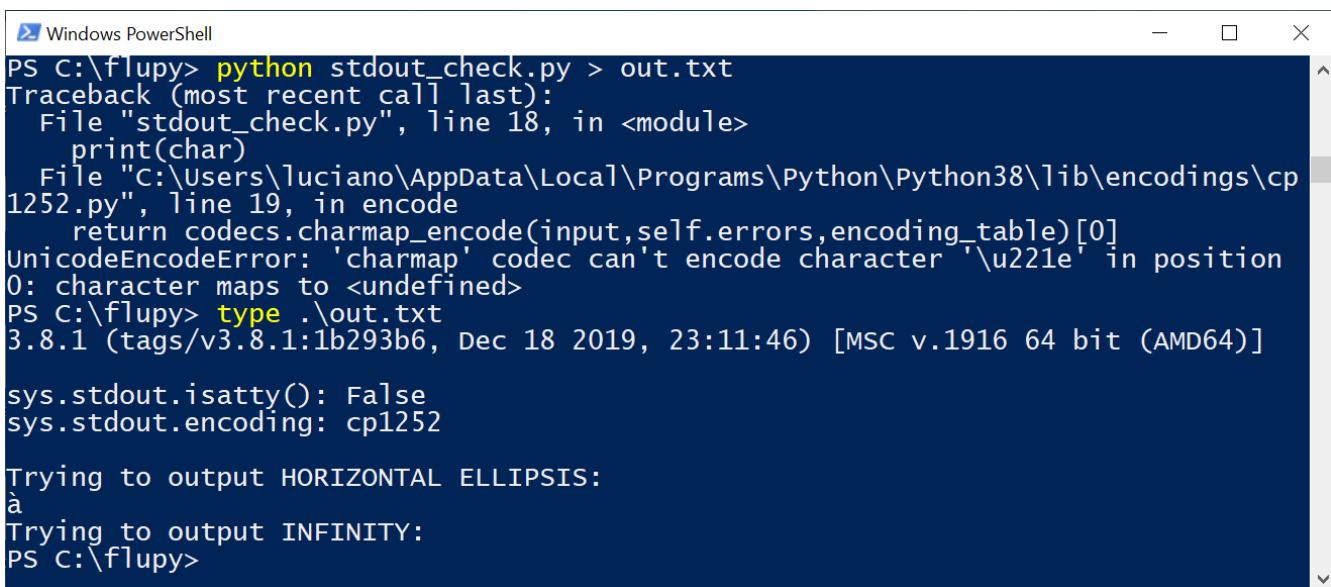
```
Windows PowerShell
PS C:\flupy> chcp
Active code page: 437
PS C:\flupy> python stdout_check.py
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]
sys.stdout.isatty(): True
sys.stdout.encoding: utf-8

Trying to output HORIZONTAL ELLIPSIS:
...
Trying to output INFINITY:
∞
Trying to output CIRCLED NUMBER FORTY TWO:
□
PS C:\flupy>
```

Figura 12. Executando `stdout_check.py` no PowerShell.

Apesar de `chcp` informar o código ativo como 437, `sys.stdout.encoding` é UTF-8, então tanto HORIZONTAL ELLIPSIS quanto INFINITY são escritos corretamente. O CIRCLED NUMBER FORTY TWO é substituído por um retângulo, mas nenhum erro é gerado. Presume-se que ele seja reconhecido como um caractere válido, mas a fonte do console não tem o glifo para mostrá-lo.

Entretanto, quando redireciono a saída de `stdout_check.py` para um arquivo, o resultado é o da Figura 13.



```
Windows PowerShell
PS C:\flupy> python stdout_check.py > out.txt
Traceback (most recent call last):
  File "stdout_check.py", line 18, in <module>
    print(char)
  File "C:\Users\luciano\AppData\Local\Programs\Python\Python38\lib\encodings\cp1252.py", line 19, in encode
    return codecs.charmap_encode(input, self.errors, encoding_table)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u221e' in position
0: character maps to <undefined>
PS C:\flupy> type .\out.txt
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]

sys.stdout.isatty(): False
sys.stdout.encoding: cp1252

Trying to output HORIZONTAL ELLIPSIS:
à
Trying to output INFINITY:
PS C:\flupy>
```

Figura 13. Executando `stdout_check.py` no PowerShell, redirecionando a saída.

O primeiro problema demonstrado pela Figura 13 é o `UnicodeEncodeError` mencionando o caractere '\u221e', porque `sys.stdout.encoding` é 'cp1252'—uma página de código que não tem o caractere INFINITY.

Lendo `out.txt` com o comando `type`—ou um editor de Windows como o VS Code ou o Sublime Text—mostra que, ao invés do HORIZONTAL ELLIPSIS, consegui um 'à' (LATIN SMALL LETTER A WITH GRAVE). Acontece que o valor binário 0x85 no CP 1252 significa '...', mas no CP 437 o mesmo valor binário representa o 'à'. Então, pelo visto, a página de código ativa tem alguma importância, não de uma forma razoável ou útil, mas como uma explicação parcial para uma experiência ruim com o Unicode.



Para realizar esses experimentos, usei um laptop configurado para o mercado norte-americano, rodando Windows 10 OEM. Versões de Windows localizadas para outros países podem ter configurações de codificação diferentes. No Brasil, por exemplo, o console do Windows usa a página de código 850 por default—e não a 437.

Para encerrar esse enlouquecedor tópico de codificações default, vamos dar uma última olhada nas diferentes codificações no Exemplo 51:

- Se você omitir o argumento `encoding` ao abrir um arquivo, o default é dado por `locale.getpreferredencoding()` ('cp1252' no Exemplo 51).
- Antes de Python 3.6, a codificação de `sys.stdout|stdin|stderr` costumava ser determinada pela variável do ambiente `PYTHONIOENCODING` [<https://fpy.li/3b>]—agora essa variável é ignorada, a menos que `PYTHONLEGACYWINDOWSSTDI0` [<https://fpy.li/3c>] seja definida como uma string não-vazia. Caso contrário, a codificação da E/S padrão será UTF-8 para E/S interativa, ou definida por `locale.getpreferredencoding()`, se a entrada e a saída forem redirecionadas para ou de um arquivo.
- `sys.getdefaultencoding()` é usado internamente pelo Python em conversões implícitas de dados binários de ou para `str`. Não há suporte para mudar essa configuração.
- `sys.getfilesystemencoding()` é usado para codificar/decodificar nomes de arquivo (mas não o conteúdo dos arquivos). Ele é usado quando `open()` recebe um argumento `str` para um nome de arquivo; se o nome do arquivo é passado como um argumento `bytes`, ele é entregue sem modificação para a API do sistema operacional.



Já faz muito anos que, no GNU/Linux e no macOS, todas essas codificações são definidas como UTF-8 por default, então a E/S entende e exibe todos os caracteres Unicode. No Windows, não apenas codificações diferentes são usadas no mesmo sistema, elas também são, normalmente, páginas de código como 'cp850' ou 'cp1252', que suportam só o ASCII com 127 caracteres adicionais (que por sua vez são diferentes de uma codificação para a outra). Assim, usuários de Windows têm mais chances de encontrar erros de codificação.

Resumindo, a configuração de codificação mais importante devolvida por `locale.getpreferredencoding()` é a default para abrir arquivos de texto e para `sys.stdout/stdin/stderr`, quando eles são redirecionados para arquivos. Entretanto, a documentação [<https://fpy.li/3d>] diz (em parte):

`locale.getpreferredencoding(do_setlocale=True)`

Retorna a codificação da localidade usada para dados de texto, de acordo com as preferências do usuário. As preferências do usuário são expressas de maneira diferente em sistemas diferentes e podem não estar disponíveis programaticamente em alguns sistemas, portanto, essa função retorna apenas um palpite. [...]

Assim, o melhor conselho sobre defaults de codificação é: não confie neles.

Você evitara muitas dores de cabeça se seguir o conselho do sanduíche de Unicode, e sempre tratar codificações de forma explícita em seus programas. Infelizmente, o Unicode é trabalhoso mesmo se você converter seus bytes para `str` corretamente. As duas próximas seções tratam de assuntos que são simples no reino do ASCII, mas ficam muito complexos no planeta Unicode: normalização de texto (isto é, transformar o texto em uma representação uniforme para comparações) e ordenação.

4.7. Normalizando o Unicode para comparações confiáveis

Comparações de strings são dificultadas pelo fato do Unicode ter combinações de caracteres: sinais diacríticos e outras marcações que são sobrepostas ao caractere anterior, ambos aparecendo juntos como um só caractere quando impressos.

Por exemplo, a palavra "café" pode ser composta de duas formas, usando quatro ou cinco pontos de código, mas o resultado parece exatamente o mesmo:

```
>>> s1 = 'café'
>>> s2 = 'cafe\N{COMBINING ACUTE ACCENT}'
>>> s1, s2
('café', 'café')
>>> len(s1), len(s2)
(4, 5)
>>> s1 == s2
False
```

Colocar `COMBINING ACUTE ACCENT` (U+0301) após o "e" resulta em "é". No padrão Unicode, sequências como 'é' e 'e\u0301' são chamadas de "equivalentes canônicas", e se espera que as aplicações as tratem como iguais. Mas Python vê duas sequências de pontos de código diferentes, e não as considera iguais.

A solução é a `unicodedata.normalize()`. O primeiro argumento para essa função é uma dessas quatro strings: 'NFC', 'NFD', 'NFKC', e 'NFKD'. Vamos começar pelas duas primeiras.

A Forma Normal C (NFC) combina os ponto de código para produzir a string equivalente mais curta, enquanto a NFD decompõe, expandindo os caracteres compostos em caracteres base e separando caracteres combinados. Ambas as normalizações fazem as comparações funcionarem da forma esperada, como mostra o próximo exemplo:

```
>>> from unicodedata import normalize
>>> s1 = 'café'
>>> s2 = 'cafe\N{COMBINING ACUTE ACCENT}'
>>> len(s1), len(s2)
(4, 5)
>>> len(normalize('NFC', s1)), len(normalize('NFC', s2))
(4, 4)
>>> len(normalize('NFD', s1)), len(normalize('NFD', s2))
(5, 5)
```

```
>>> normalize('NFC', s1) == normalize('NFC', s2)
True
>>> normalize('NFD', s1) == normalize('NFD', s2)
True
```

Drivers de teclado normalmente geram caracteres compostos, então o texto digitado pelos usuários estará na NFC por default. Entretanto, por segurança, pode ser melhor normalizar as strings com `normalize('NFC', user_text)` antes de salvá-las. A NFC também é a forma de normalização recomendada pelo W3C em "Character Model for the World Wide Web: String Matching and Searching" (*Modelo de Caracteres para a World Wide Web: Casamento de Strings e Busca*) [<https://fpy.li/4-15>] (EN).

Alguns caracteres singulares são normalizados pela NFC em um outro caractere singular. O símbolo para o ohm (Ω), a unidade de medida de resistência elétrica, é normalizado para a letra grega ômega maiúscula. Eles são visualmente idênticos, mas diferentes quando comparados, então a normalização pode evitar surpresas:

```
>>> from unicodedata import normalize, name
>>> ohm = '\u03a9'
>>> name(ohm)
'OHM SIGN'
>>> ohm_c = normalize('NFC', ohm)
>>> name(ohm_c)
'GREEK CAPITAL LETTER OMEGA'
>>> ohm == ohm_c
False
>>> normalize('NFC', ohm) == normalize('NFC', ohm_c)
True
```

As outras duas formas de normalização são a NFKC e a NFKD, a letra K significando "compatibilidade". Essas são formas mais fortes de normalização, afetando os assim chamados "caracteres de compatibilidade". Apesar de um dos objetivos do Unicode ser a existência de um único ponto de código "canônico" para cada caractere, alguns caracteres aparecem mais de uma vez, para manter compatibilidade com padrões pré-existentes. Por exemplo, o MICRO SIGN, μ (U+00B5), foi adicionado para permitir a conversão bi-direcional com o latin1, que o inclui, apesar do mesmo caractere ser parte do alfabeto grego com o ponto de código U+03BC (GREEK SMALL LETTER MU). Assim, o símbolo de micro é considerado um "caractere de compatibilidade".

Nas formas NFKC e NFKD, cada caractere de compatibilidade é substituído por uma "decomposição de compatibilidade" de um ou mais caracteres, que é considerada a representação "preferencial", mesmo se ocorrer alguma perda de formatação—idealmente, a formatação deveria ser responsabilidade de alguma marcação externa, não parte do Unicode. Para exemplificar, a decomposição de compatibilidade da fração um meio, ' $\frac{1}{2}$ ' (U+00BD), é a sequência de três caracteres '1/2', e a decomposição de compatibilidade do símbolo de micro, ' μ ' (U+00B5), é o mu minúsculo, ' μ ' (U+03BC).^[7]

É assim que a NFKC funciona na prática:

```

>>> from unicodedata import normalize, name
>>> half = '\N{VULGAR FRACTION ONE HALF}'
>>> print(half)
½
>>> normalize('NFKC', half)
'1/2'
>>> for char in normalize('NFKC', half):
...     print(char, name(char), sep='\t')
...
1  DIGIT ONE
/  FRACTION SLASH
2  DIGIT TWO
>>> four_squared = '4²'
>>> normalize('NFKC', four_squared)
'42'
>>> micro = 'μ'
>>> micro_kc = normalize('NFKC', micro)
>>> micro, micro_kc
('μ', 'μ')
>>> ord(micro), ord(micro_kc)
(181, 956)
>>> name(micro), name(micro_kc)
('MICRO SIGN', 'GREEK SMALL LETTER MU')

```

Ainda que '1/2' seja um substituto razoável para '½', e o símbolo de micro ser realmente a letra grega mu minúscula, converter '4²' para '42' muda o sentido. Uma aplicação poderia armazenar '4²' como '4²', mas a função `normalize` não sabe nada sobre formatação. Assim, NFKC ou NFKD podem perder ou distorcer informações, mas podem produzir representações intermediárias convenientes para buscas ou indexação.

Infelizmente, com o Unicode tudo é sempre mais complicado do que parece à primeira vista. Para o VULGAR FRACTION ONE HALF, a normalização NFKC produz 1 e 2 unidos pelo FRACTION SLASH, em vez do SOLIDUS, também conhecido como "barra" ("slash" em inglês)—o familiar caractere com código decimal 47 em ASCII. Portanto, buscar pela sequência ASCII de três caracteres '1/2' não encontraria a sequência Unicode normalizada.



As normalizações NFKC e NFKD causam perda de dados e devem ser aplicadas apenas em casos especiais, como busca e indexação, e não para armazenamento permanente do texto.

Ao preparar texto para busca ou indexação, há outra operação útil: *case folding*^[8], nosso próximo assunto.

4.7.1. Case Folding

Case folding é essencialmente a conversão de todo o texto para minúsculas, com algumas transformações adicionais. A operação é suportada pelo método `str.casefold()`.

Para qualquer string `s` contendo apenas caracteres latin1, `s.casefold()` produz o mesmo resultado

de `s.lower()`, com apenas duas exceções—o símbolo de micro, 'μ', é trocado pela letra grega mu minúscula (que é exatamente igual na maioria das fontes) e a letra alemã *Eszett* (ß), também chamada "s agudo" (*scharfes S*) se torna "ss":

```
>>> micro = 'μ'
>>> name(micro)
'MICRO SIGN'
>>> micro_cf = micro.casefold()
>>> name(micro_cf)
'GREEK SMALL LETTER MU'
>>> micro, micro_cf
('μ', 'μ')
>>> eszett = 'ß'
>>> name(eszett)
'LATIN SMALL LETTER SHARP S'
>>> eszett_cf = eszett.casefold()
>>> eszett, eszett_cf
('ß', 'ss')
```

Há quase 300 pontos de código para os quais `str.casefold()` e `str.lower()` devolvem resultados diferentes.

Como acontece com qualquer coisa relacionada ao Unicode, *case folding* é um tópico complexo, com muitos casos linguísticos especiais, mas os desenvolvedores de Python fizeram um grande esforço para apresentar uma solução que, espera-se, funcione para a maioria dos usuários.

Nas próximas seções vamos colocar nosso conhecimento sobre normalização para trabalhar, desenvolvendo algumas funções utilitárias.

4.7.2. Funções utilitárias para casamento de texto normalizado

Como vimos, é seguro usar a NFC e a NFD, e ambas permitem comparações razoáveis entre strings Unicode. A NFC é a melhor forma normalizada para a maioria das aplicações, e `str.casefold()` é a opção certa para comparações indiferentes a maiúsculas/minúsculas.

Se você precisa lidar com texto em muitas línguas diferentes, seria muito útil acrescentar às suas ferramentas de trabalho um par de funções como `nfc_equal` e `fold_equal`, do Exemplo 53.

Exemplo 53. normeq.py: normalized Unicode string comparison

```
"""
Utility functions for normalized Unicode string comparison.

Using Normal Form C, case sensitive:

>>> s1 = 'caf  '
>>> s2 = 'cafe  \u0301'
>>> s1 == s2
False
```

```
>>> nfc_equal(s1, s2)
True
>>> nfc_equal('A', 'a')
False
```

Using Normal Form C with case folding:

```
>>> s3 = 'Stra e'  
>>> s4 = 'strasse'  
>>> s3 == s4  
False  
>>> nfc_equal(s3, s4)  
False  
>>> fold_equal(s3, s4)  
True  
>>> fold_equal(s1, s2)  
True  
>>> fold_equal('A', 'a')  
True
```

三

```
from unicodedata import normalize

def nfc_equal(str1, str2):
    return normalize('NFC', str1) == normalize('NFC', str2)

def fold_equal(str1, str2):
    return (normalize('NFC', str1).casefold() ==
            normalize('NFC', str2).casefold())
```

Além da normalização e do *case folding* do Unicode—ambos partes desse padrão—algumas vezes faz sentido aplicar transformações mais profundas, como por exemplo mudar 'café' para 'cafe'. Vamos ver quando e como na próxima seção.

4.7.3. "Normalização" extrema: removendo sinais diacríticos

O tempero secreto da busca do Google inclui muitos truques, mas um deles aparentemente é ignorar sinais diacríticos (acentos e cedilhas, por exemplo), pelo menos em alguns contextos. Remover sinais diacríticos não é uma forma regular de normalização, pois muitas vezes muda o sentido das palavras e pode produzir falsos positivos em uma busca. Mas ajuda a lidar com alguns fatos da vida: as pessoas às vezes são preguiçosas ou desconhecem o uso correto dos sinais diacríticos, e regras de ortografia mudam com o tempo, levando acentos a desaparecerem e reaparecerem nas línguas vivas.

Além do caso da busca, eliminar os acentos torna as URLs mais legíveis, pelo menos nas línguas latinas. Veja a URL do artigo da Wikipedia sobre a cidade de São Paulo:

```
https://en.wikipedia.org/wiki/S%C3%A3o\_Paulo
```

O trecho %C3%A3 é a renderização em UTF-8 de uma única letra, o "ã" ("a" com til). A forma a seguir é mais fácil de reconhecer, mesmo com a ortografia incorreta:

```
https://en.wikipedia.org/wiki/Sao\_Paulo
```

Para remover todos os sinais diacríticos de uma str, você pode usar uma função como a do Exemplo 54.

Exemplo 54. simplify.py: função para remover todas as marcações combinadas

```
import unicodedata
import string

def shave_marks(txt):
    """Remove all diacritic marks"""
    norm_txt = unicodedata.normalize('NFD', txt) ①
    shaved = ''.join(c for c in norm_txt
                     if not unicodedata.combining(c)) ②
    return unicodedata.normalize('NFC', shaved) ③
```

① Decompõe todos os caracteres em caracteres base e marcações combinadas.

② Filtra e retira todas as marcações combinadas.

③ Recompõe todos os caracteres.

Exemplo 55 mostra alguns usos para shave_marks.

Exemplo 55. Dois exemplos de uso da shave_marks do Exemplo 54

```
>>> order = 'Herr Voß: ½ cup of Etker™ caffè latte à bowl of açaí.'
>>> shave_marks(order)
'Herr Voß: ½ cup of Etker™ caffe latte à bowl of acai.' ①
>>> Greek = 'Ζέφυρος, Ζέφυρος'
>>> shave_marks(Greek)
'Ζεφύρος, Ζεφύρος' ②
```

① Apenas as letras "è", "ç", e "í" foram substituídas.

② Tanto "é" quanto "é" foram substituídas.

A função shave_marks do Exemplo 54 funciona bem, mas talvez vá longe demais. Frequentemente, a razão para remover os sinais diacríticos é transformar texto de uma língua latina para ASCII puro, mas shave_marks também troca caracteres não-latinos—como letras gregas—que nunca se tornarão ASCII apenas pela remoção de acentos. Então faz sentido analisar cada caractere base e remover as marcações anexas apenas se o caractere base for uma letra do alfabeto latino. É isso que o Exemplo

56 faz.

Exemplo 56. Função para remover marcações combinadas de caracteres latinos (comando de importação omitidos, pois isso é parte do módulo simplify.py do Exemplo 54)

```
def shave_marks_latin(txt):
    """Remove all diacritic marks from Latin base characters"""
    norm_txt = unicodedata.normalize('NFD', txt) ①
    latin_base = False
    preserve = []
    for c in norm_txt:
        if unicodedata.combining(c) and latin_base: ②
            continue # ignore diacritic on Latin base char
        preserve.append(c) ③
        # if it isn't a combining char, it's a new base char
        if not unicodedata.combining(c): ④
            latin_base = c in string.ascii_letters
    shaved = ''.join(preserve)
    return unicodedata.normalize('NFC', shaved) ⑤
```

- ① Decompõe todos os caracteres em caracteres base e marcações combinadas.
 - ② Pula as marcações combinadas quando o caractere base é latino.
 - ③ Caso contrário, mantém o caractere original.
 - ④ Detecta um novo caractere base e determina se ele é latino.
 - ⑤ Recompõe todos os caracteres.

Um passo ainda mais radical substituiria os símbolos comuns em textos de línguas ocidentais (por exemplo, aspas curvas, travessões, os círculos de *bullet points*, etc) em seus equivalentes ASCII. É isso que a função `asciize` faz no Exemplo 57.

Exemplo 57. Transforma alguns símbolos tipográficos ocidentais em ASCII (este trecho também é parte do simplify.py do Exemplo 54)

```

multi_map.update(single_map) ③

def dewinize(txt):
    """Replace Win1252 symbols with ASCII chars or sequences"""
    return txt.translate(multi_map) ④

def asciize(txt):
    no_marks = shave_marks_latin(dewinize(txt))      ⑤
    no_marks = no_marks.replace('ß', 'ss')            ⑥
    return unicodedata.normalize('NFKC', no_marks) ⑦

```

- ① Cria uma tabela de mapeamento para substituição de caractere para caractere.
- ② Cria uma tabela de mapeamento para substituição de string para caractere.
- ③ Funde as tabelas de mapeamento.
- ④ `dewinize` não afeta texto em ASCII ou latin1, apenas os acréscimos da Microsoft ao latin1 no cp1252.
- ⑤ Aplica `dewinize` e remove as marcações de sinais diacríticos.
- ⑥ Substitui o *Eszett* por "ss" (não estamos usando *case folding* aqui, pois queremos preservar maiúsculas e minúsculas).
- ⑦ Aplica a normalização NFKC para compor os caracteres com seus pontos de código de compatibilidade.

O Exemplo 58 mostra a `asciize` em ação.

Exemplo 58. Dois exemplos usando `asciize`, do Exemplo 57

```

>>> order = 'Herr Voß: ½ cup of Etker™ caffè latte  bowl of açaí.'
>>> dewinize(order)
'"Herr Voß: - ½ cup of OEtker(TM) caffè latte - bowl of açaí."' ①
>>> asciize(order)
'"Herr Voss: - ½ cup of OEtker(TM) caffe latte - bowl of acai."' ②

```

- ① `dewinize` substitui as aspas curvas, os *bullets*, e o ™ (símbolo de marca registrada).
- ② `asciize` aplica `dewinize`, remove os sinais diacríticos e substitui o 'ß'.



Cada idioma tem suas próprias regras para remoção de sinais diacríticos. Por exemplo, os alemães trocam o 'ü' por 'ue'. Nossa função `asciize` não é tão refinada, então pode ou não ser adequada para seu idioma. Contudo, ela é aceitável para o português.

Resumindo, as funções em *simplify.py* vão bem além da normalização padrão, e realizam uma cirurgia profunda no texto, com boas chances de mudar seu sentido. Só você pode decidir se deve ir tão longe, conhecendo o idioma alvo, os seus usuários e a forma como o texto transformado será utilizado.

Isso conclui nossa discussão sobre normalização de texto Unicode.

Vamos agora ordenar nossos pensamentos sobre ordenação no Unicode.

4.8. Ordenando texto Unicode

Python ordena sequências de qualquer tipo comparando um por um os itens em cada sequência. Para strings, isso significa comparar pontos de código. Infelizmente, isso produz resultados inaceitáveis para qualquer um que use caracteres não-ASCII.

Considere ordenar uma lista de frutas cultivadas no Brazil:

```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açaí', 'caju', 'cajá']
```

As regras de ordenação variam entre diferentes locales, mas em português e em muitas línguas que usam o alfabeto latino, acentos e cedilhas raramente fazem diferença na ordenação.^[9] Então "cajá" é lido como "caja," e deve vir antes de "caju."

A lista fruits ordenada deveria ser:

```
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

O modo padrão de ordenar texto não-ASCII em Python é usar a função `locale.strxfrm` que, de acordo com a documentação do módulo `locale` [<https://fpy.li/3e>] "transforma uma string em uma que pode ser usada em comparações com reconhecimento de localidade."

Para poder usar `locale.strxfrm`, você deve primeiro definir um *locale* adequado para sua aplicação, e rezar para que o SO o suporte. A sequência de comando no Exemplo 59 pode funcionar para você.

Exemplo 59. locale_sort.py: Usando a função locale.strxfrm como chave de ordenamento

```
import locale
my_locale = locale.setlocale(locale.LC_COLLATE, 'pt_BR.UTF-8')
print(my_locale)
fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
sorted_fruits = sorted(fruits, key=locale.strxfrm)
print(sorted_fruits)
```

Executando o Exemplo 59 no GNU/Linux (Ubuntu 19.10) com o *locale* pt_BR.UTF-8 instalado, consigo o resultado correto:

```
'pt_BR.UTF-8'
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

Portanto, você precisa chamar `setlocale(LC_COLLATE, «your_locale»)` antes de usar `locale.strxfrm` como a chave de ordenação.

Porém, aqui vão algumas ressalvas:

- Como as configurações de *locale* são globais, não é recomendado chamar `setlocale` em uma biblioteca. Sua aplicação ou framework deveria definir o *locale* no início do processo, e não mudá-lo mais depois disso.
- O *locale* desejado deve estar instalado no SO, caso contrário `setlocale` gera uma exceção de `locale.Error: unsupported locale setting`.
- Você tem que saber como escrever corretamente o nome do *locale*.
- O *locale* precisa ser corretamente implementado pelos desenvolvedores do SO. Tive sucesso com o Ubuntu 19.10, mas não no macOS 10.14. No macOS, a chamada `setlocale(LC_COLLATE, 'pt_BR.UTF-8')` devolve a string '`pt_BR.UTF-8`' sem qualquer reclamação. Mas `sorted(fruits, key=locale.strxfrm)` produz o mesmo resultado incorreto de `sorted(fruits)`. Também tentei os *locales* fr_FR, es_ES, e de_DE no macOS, mas `locale.strxfrm` nunca fez seu trabalho direito.^[10]

Portanto, a solução da biblioteca padrão para ordenação internacionalizada funciona, mas parece ter suporte adequado apenas no GNU/Linux (talvez também no Windows, se você for um especialista). Mesmo assim, ela depende das configurações do *locale*, criando dores de cabeça na implantação.

Felizmente, há uma solução mais simples: a biblioteca *pyuca*, disponível no PyPI.

4.8.1. Ordenando com o Algoritmo de Ordenação do Unicode

James Tauber, contribuidor muito ativo do Django, deve ter sentido as dores de lidar com *locale*. Ele criou a *pyuca* [<https://fpy.li/4-17>], uma implementação integralmente em Python do Algoritmo de Ordenação do Unicode (UCA, sigla em inglês para *Unicode Collation Algorithm*). O Exemplo 60 mostra como ela é fácil de usar.

Exemplo 60. Utilizando o método pyuca.Collator.sort_key

```
>>> import pyuca
>>> coll = pyuca.Collator()
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted_fruits = sorted(fruits, key=coll.sort_key)
>>> sorted_fruits
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

Isso é simples e funciona no GNU/Linux, no macOS, e no Windows, pelo menos com a minha pequena amostra.

A *pyuca* não leva o *locale* em consideração. Se você precisar customizar a ordenação, pode fornecer um caminho para uma tabela própria de ordenação para o construtor `Collator()`. Sem qualquer configuração adicional, a biblioteca usa o *allkeys.txt* [<https://fpy.li/4-18>], incluído no projeto. Esse arquivo é apenas uma cópia da Default Unicode Collation Element Table (*Tabela Default de Ordenação de Elementos Unicode*) [<https://fpy.li/4-19>] do *Unicode.org*.

PyICU: A recomendação do Miro para ordenação com Unicode

(O revisor técnico Miroslav Šedivý é um poliglota e um especialista em Unicode. Eis o que ele escreveu sobre a *pyuca*.)



A *pyuca* tem um algoritmo de ordenação que não respeita o padrão de ordenação de linguagens individuais. Por exemplo, [a letra] Ä em alemão fica entre o A e o B, enquanto em sueco ela vem depois do Z. Dê uma olhada na PyICU [<https://fpy.li/4-20>], que funciona sem modificar o locale do processo. Ela também é necessária se você quiser mudar a capitalização de ï/İ em turco. A PyICU inclui uma extensão que precisa ser compilada, então pode ser mais difícil de instalar em alguns sistemas que a *pyuca*, que é toda feita em Python.

E por sinal, aquela tabela de ordenação é um dos muitos arquivos de dados que formam o banco de dados do Unicode, nosso próximo assunto.

4.9. O banco de dados do Unicode

O padrão Unicode fornece todo um banco de dados—na forma de vários arquivos de texto estruturados—que inclui não apenas a tabela mapeando pontos de código para nomes de caracteres, mas também metadados sobre os caracteres individuais e como eles se relacionam. Por exemplo, o banco de dados do Unicode registra se um caractere pode ser impresso, se é uma letra, um dígito decimal ou algum outro símbolo numérico. É assim que os métodos de `str.isalpha`, `isprintable`, `isdecimal` e `isnumeric` funcionam. `str.casefold` também usa informação de uma tabela do Unicode.



A função `unicodedata.category(char)` devolve uma categoria de `char` com duas letras, do banco de dados do Unicode. Os métodos de alto nível de `str` são mais fáceis de usar. Por exemplo, `label.isalpha()` [<https://fpy.li/3f>] devolve `True` se todos os caracteres em `label` pertencerem a uma das seguintes categorias: `Lm`, `Lt`, `Lu`, `Ll`, or `Lo`. Para descobrir o que esses códigos significam, veja "General Category" [<https://fpy.li/4-22>] (EN) no artigo "Unicode character property" [<https://fpy.li/4-23>] (EN) da Wikipedia em inglês.

4.9.1. Encontrando caracteres por nome

O módulo `unicodedata` tem funções para obter os metadados de caracteres, incluindo `unicodedata.name()`, que devolve o nome oficial do caractere no padrão. A Figura 14 demonstra essa função.^[11]

```
>>> from unicodedata import name
>>> name('A')
'LATIN CAPITAL LETTER A'
>>> name('ã')
'LATIN SMALL LETTER A WITH TILDE'
>>> name('♚')
'BLACK CHESS QUEEN'
>>> name('😺')
'GRINNING CAT FACE WITH SMILING EYES'
```

Figura 14. Explorando `unicodedata.name()` no console de Python.

Você pode usar a função `name()` para criar aplicações que permitem aos usuários buscarem caracteres por nome. A Figura 15 demonstra o script de linha de comando `cf.py`, que recebe como argumentos uma ou mais palavras, e lista os caracteres que tem aquelas palavras em seus nomes Unicode oficiais. O código fonte completo de `cf.py` aparece no Exemplo 61.

```
$ ./cf.py cat smiling
U+1F638 😺 GRINNING CAT FACE WITH SMILING EYES
U+1F63A 😻 SMILING CAT FACE WITH OPEN MOUTH
U+1F63B 😻 SMILING CAT FACE WITH HEART-SHAPED EYES
```

Figura 15. Usando `cf.py` para encontrar gatos sorridentes.



O suporte a emojis varia muito entre sistemas operacionais e aplicativos. Nos últimos anos, o terminal do macOS tem oferecido o melhor suporte para emojis, seguido por terminais gráficos GNU/Linux modernos. O `cmd.exe` e o PowerShell do Windows agora suportam saída Unicode, mas enquanto escrevo essa seção, em janeiro de 2020, eles ainda não mostram emojis—pelo menos não sem configurações adicionais. O revisor técnico Leonardo Rochael me falou sobre um novo terminal para Windows da Microsoft [<https://fpy.li/4-24>], de código aberto, que pode ter um suporte melhor a Unicode que os consoles antigos da Microsoft. Não tive tempo de testar.

No Exemplo 61, observe que o comando `if`, na função `find`, usa o método `.issubset()` para testar rapidamente se todas as palavras no conjunto `query` aparecem na lista de palavras criada a partir do nome do caractere. Graças à rica API de conjuntos de Python, não precisamos de um loop `for` aninhado e de outro `if` para implementar essa verificação.

Exemplo 61. `cf.py`: o utilitário de busca de caracteres

```
#!/usr/bin/env python3
import sys
import unicodedata

START, END = ord(' '), sys.maxunicode + 1          ①

def find(*query_words, start=START, end=END):        ②
    query = {w.upper() for w in query_words}          ③
```

```

for code in range(start, end):
    char = chr(code)                                ④
    name = unicodedata.name(char, None)              ⑤
    if name and query.issubset(name.split()):        ⑥
        print(f'U+{code:04X}\t{char}\t{name}')       ⑦

def main(words):
    if words:
        find(*words)
    else:
        print('Please provide words to find.')

if __name__ == '__main__':
    main(sys.argv[1:])

```

- ① Configura os defaults para a faixa de pontos de código da busca.
- ② `find` aceita `query_words` e somente argumentos nomeados (opcionais) para limitar a faixa da busca, facilitando os testes.
- ③ Converte `query_words` em um conjunto de strings capitalizadas.
- ④ Obtém o caractere Unicode para `code`.
- ⑤ Obtém o nome do caractere, ou `None` se o ponto de código não estiver atribuído a um caractere.
- ⑥ Se há um nome, separa esse nome em uma lista de palavras, então verifica se o conjunto `query` é um subconjunto daquela lista.
- ⑦ Mostra uma linha com o ponto de código no formato U+9999, o caractere e seu nome.

O módulo `unicodedata` tem outras funções interessantes. A seguir veremos algumas delas, relacionadas a obter informação de caracteres com significado numérico.

4.9.2. O sentido numérico de caracteres

O módulo `unicodedata` inclui funções para determinar se um caractere Unicode representa um número e, se for esse o caso, seu valor numérico em termos humanos—em contraste com o número de seu ponto de código.

O Demo do banco de dados Unicode de metadados de caracteres numéricos demonstra o uso de `unicodedata.name()` e `unicodedata.numeric()`, junto com os métodos `.isdecimal()` e `.isnumeric()` de `str`.

*Demo do banco de dados Unicode de metadados de caracteres numéricos
(as notas explicativas descrevem cada coluna da saída)*

```

import unicodedata
import re

re_digit = re.compile(r'\d')

```

```

sample = '1\xbc\xb2\u0969\u136b\u216b\u2466\u2480\u3285'

for char in sample:
    print(f'U+{ord(char):04x}',           ①
          char.center(6),                ②
          're_dig' if re_digit.match(char) else '-', ③
          'isdig' if char.isdigit() else '-', ④
          'isnum' if char.isnumeric() else '-', ⑤
          f'{unicodedata.numeric(char):5.2f}', ⑥
          unicodedata.name(char),            ⑦
          sep='\t')

```

- ① Ponto de código no formato U+0000.
- ② O caractere, centralizado em uma str de tamanho 6.
- ③ Mostra `re_dig` se o caractere casa com a regex `r'\d'`.
- ④ Mostra `isdig` se `char.isdigit()` é True.
- ⑤ Mostra `isnum` se `char.isnumeric()` é True.
- ⑥ Valor numérico formatado com tamanho 5 e duas casa decimais.
- ⑦ O nome Unicode do caractere.

Executar o Demo do banco de dados Unicode de metadados de caracteres numéricos gera a Figura 16, se a fonte do seu terminal incluir todos aqueles símbolos.

Character	Code Point	re_dig	isdig	isnum	Numeric Value	Name
U+0031	1	re_dig	isdig	isnum	1.00	DIGIT ONE
U+00bc	¼	-	-	isnum	0.25	VULGAR FRACTION ONE QUARTER
U+00b2	²	-	isdig	isnum	2.00	SUPERSCRIPT TWO
U+0969	୩	re_dig	isdig	isnum	3.00	DEVANAGARI DIGIT THREE
U+136b	߳	-	isdig	isnum	3.00	ETHIOPIC DIGIT THREE
U+216b	XII	-	-	isnum	12.00	ROMAN NUMERAL TWELVE
U+2466	⑦	-	isdig	isnum	7.00	CIRCLED DIGIT SEVEN
U+2480	(୯)	-	-	isnum	13.00	PARENTHEΣIZED NUMBER THIRTEEN
U+3285	ଓ	-	-	isnum	6.00	CIRCLED IDEOGRAPH SIX

Figura 16. Terminal do macOS mostrando os caracteres numéricos e metadados correspondentes; `re_dig` significa que o caractere casa com a expressão regular `r'\d'`.

A sexta coluna da Figura 16 é o resultado da chamada a `unicodedata.numeric(char)` com o caractere. Ela mostra que o Unicode sabe o valor numérico de símbolos que representam números. Assim, se você quiser criar uma aplicação de planilha que suporta dígitos tamil ou numerais romanos, vá fundo!

A Figura 16 mostra que a expressão regular `r'\d'` casa com o dígito "1" e com o dígito 3 da escrita devanágari, mas não com alguns outros caracteres considerados dígitos pela função `isdigit`. O módulo `re` não é tão conhecedor de Unicode quanto deveria ser. O novo módulo `regex`, disponível no PyPI, foi projetado para um dia substituir o `re`, e fornece um suporte melhor ao Unicode.^[12] Voltaremos ao módulo `re` na próxima seção.

Ao longo desse capítulo, usamos várias funções de `unicodedata`, mas há muitas outras que não mencionamos. Veja a documentação da biblioteca padrão para o módulo `unicodedata` [<https://fpy.li/3g>].

A seguir vamos dar uma rápida passada pelas APIs de modo dual, com funções que aceitam argumentos `str` ou `bytes` e dão a eles tratamento especial dependendo do tipo.

4.10. APIs de modo dual para `str` e `bytes`

A biblioteca padrão de Python tem funções que aceitam argumentos `str` ou `bytes` e se comportam de forma diferente dependendo do tipo recebido. Alguns exemplos podem ser encontrados nos módulos `re` e `os`.

4.10.1. `str` versus `bytes` em expressões regulares

Se você criar uma expressão regular com `bytes`, padrões tal como `\d` e `\w` vão casar apenas com caracteres ASCII; por outro lado, se esses padrões forem passados como `str`, eles vão casar com dígitos Unicode ou letras além do ASCII. O Exemplo 62 e a Figura 17 comparam como letras, dígitos ASCII, superescritos e dígitos tamil casam em padrões `str` e `bytes`.

Exemplo 62. ramanujan.py: compara o comportamento de expressões regulares simples como str e como bytes

```
import re

re_numbers_str = re.compile(r'\d+')      ①
re_words_str = re.compile(r'\w+')
re_numbers_bytes = re.compile(rb'\d+')    ②
re_words_bytes = re.compile(rb'\w+')

text_str = ("Ramanujan saw \u0be7\u0bed\u0be8\u0bef" ③
           " as 1729 = 13 + 123 = 93 + 103.")     ④

text_bytes = text_str.encode('utf_8')   ⑤

print(f'Text\n {text_str!r}')
print('Numbers')
print(' str :', re_numbers_str.findall(text_str))    ⑥
print(' bytes:', re_numbers_bytes.findall(text_bytes)) ⑦
print('Words')
print(' str :', re_words_str.findall(text_str))       ⑧
print(' bytes:', re_words_bytes.findall(text_bytes))   ⑨
```

① As duas primeiras expressões regulares são do tipo `str`.

② As duas últimas são do tipo `bytes`.

③ Texto Unicode para ser usado na busca, contendo os dígitos tamil para 1729 (a linha lógica continua até o símbolo de fechamento de parênteses).

④ Essa string é unida à anterior no momento da compilação (veja "2.4.2. String literal concatenation" (*Concatenação de strings literais*) [<https://fpy.li/3h>] em *A Referência da Linguagem*

Python).

- ⑤ Uma string bytes é necessária para a busca com as expressões regulares bytes.
- ⑥ O padrão str r'\d+' casa com os dígitos ASCII e tamil.
- ⑦ O padrão bytes rb'\d+' casa apenas com os bytes ASCII para dígitos.
- ⑧ O padrão str r'\w+' casa com letras, superescritos e dígitos tamil e ASCII.
- ⑨ O padrão bytes rb'\w+' casa apenas com bytes ASCII para letras e dígitos.

```
$ python3 ramanujan.py
1. bash
Text
'Ramanujan saw களைக்க as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
str : ['களைக்க', '1729', '1', '12', '9', '10']
bytes: [b'1729', b'1', b'12', b'9', b'10']
Words
str : ['Ramanujan', 'saw', 'களைக்க', 'as', '1729', '1³', '12³', '9³', '10³']
bytes: [b'Ramanujan', b'saw', b'ஈஸ்', b'1729', b'1', b'12', b'9', b'10']
$
```

Figura 17. Captura de tela da execução de ramanujan.py do Exemplo 62.

O Exemplo 62 é um exemplo trivial para destacar um ponto: você pode usar expressões regulares com str ou bytes, mas nesse último caso os bytes fora da faixa do ASCII são tratados como caracteres que não representam dígitos nem palavras.

Para expressões regulares str, há uma marcação re.ASCII, que faz \w, \W, \b, \B, \d, \D, \s, e \S casarem apenas com ASCII. Veja a documentação do módulo re [<https://fpy.li/3j>] para mais detalhes.

Outro módulo importante é o os.

4.10.2. str versus bytes nas funções de os

O kernel do GNU/Linux não conhece Unicode então, no mundo real, você pode encontrar nomes de arquivo compostos de sequências de bytes que não são válidas em nenhum esquema razoável de codificação, e não podem ser decodificados para str. Esta é uma situação comum em servidores de arquivo com clientes usando uma variedade de diferentes SO.

Para mitigar esse problema, todas as funções do módulo os que aceitam nomes de arquivo ou caminhos podem receber seus argumentos como str ou bytes. Se uma dessas funções é chamada com um argumento str, o argumento será automaticamente convertido usando o codec informado por sys.getfilesystemencoding(), e a resposta do SO será decodificada com o mesmo codec. Isso é quase sempre o que se deseja, mantendo a melhor prática do sanduíche de Unicode.

Mas se você precisa lidar com (e provavelmente corrigir) nomes de arquivo que não podem ser processados daquela forma, você pode passar argumentos bytes para as funções de os, e receber bytes de volta. Esse recurso permite que você processe qualquer nome de arquivo ou caminho, independente de quantos gremlins encontrar. Veja o Exemplo 63.

Exemplo 63. listdir com argumentos str e bytes, e os resultados

```
>>> os.listdir('.') ①
['abc.txt', 'digits-of-π.txt']
>>> os.listdir(b'.') ②
[b'abc.txt', b'digits-of-\xcf\x80.txt']
```

① O segundo nome de arquivo é "digits-of-π.txt" (com a letra grega pi).

② Dado um argumento byte, listdir devolve nomes de arquivos como bytes: b'\xcf\x80' é a codificação UTF-8 para a letra grega pi.

Para ajudar no processamento manual de sequências str ou bytes que são nomes de arquivos ou caminhos, o módulo os fornece funções especiais de codificação e decodificação, os.fsencode(name_or_path) e os.fsdecode(name_or_path). Ambas as funções aceitam argumentos dos tipos str, bytes ou, desde Python 3.6, um objeto que implemente a interface os.PathLike.

O Unicode é um buraco de coelho bem fundo. É hora de encerrar nossa exploração de str e bytes.

4.11. Resumo do capítulo

Começamos o capítulo descartando a noção de que 1 caractere == 1 byte. A medida que o mundo adota o Unicode, precisamos manter o conceito de strings de texto separado das sequências binárias que as representam em arquivos, e Python 3 reforça essa separação.

Após uma breve passada pelos tipos de dados sequências binárias—bytes, bytearray, e memoryview—mergulhamos na codificação e na decodificação, com uma amostragem dos codecs importantes, seguida por abordagens para prevenir ou lidar com os abomináveis UnicodeEncodeError, UnicodeDecodeError e os SyntaxError causados pela codificação errada em arquivos de código-fonte de Python.

A seguir consideramos a teoria e a prática de detecção de codificação na ausência de metadados: em teoria, não pode ser feita, mas na prática o pacote Chardet consegue realizar esse feito para uma grande quantidade de codificações populares. Marcadores de ordem de bytes foram apresentados como a única dica de codificação encontrada em arquivos UTF-16 e UTF-32—algumas vezes também em arquivos UTF-8.

Na seção seguinte, demonstramos como abrir arquivos de texto, uma tarefa fácil exceto por uma armadilha: o argumento nomeado encoding= não é obrigatório quando se abre um arquivo de texto, mas deveria ser. Se você não especificar a codificação, terminará com um programa que consegue produzir "texto puro" que é incompatível entre diferentes plataformas, devido a codificações default conflitantes. Expusemos então as diferentes configurações de codificação usadas pelo Python, e como detectá-las.

Uma triste realidade para usuários de Windows é o fato dessas configurações muitas vezes terem valores diferentes dentro da mesma máquina, e desses valores serem mutuamente incompatíveis; usuários do GNU/Linux e do macOS, por outro lado, vivem em um lugar mais feliz, onde o UTF-8 é o default por (quase) toda parte.

O Unicode fornece múltiplas formas de representar alguns caracteres, então a normalização é um pré-requisito para a comparação de textos. Além de explicar a normalização e o *case folding*, apresentamos algumas funções úteis que podem ser adaptadas para as suas necessidades, incluindo transformações drásticas como a remoção de todos os acentos. Vimos como ordenar corretamente texto Unicode, usando o módulo padrão `locale`—com algumas restrições—e uma alternativa que não depende de complexas configurações de `locale`: a biblioteca externa `pyuca`.

Usamos o banco de dados do Unicode para programar um utilitário de linha de comando que busca caracteres por nome—em 28 linhas de código, graças ao poder de Python. Demos uma olhada em outros metadados do Unicode, e vimos rapidamente as APIs de modo dual, onde algumas funções podem ser chamadas com argumentos `str` ou `bytes`, produzindo resultados diferentes.

4.12. Leitura complementar

A palestra de Ned Batchelder na PyCon US 2012, "Pragmatic Unicode, or, How Do I Stop the Pain?" (*Unicode Pragmático, ou, Como Eu Fiz a Dor Sumir?*) [<https://fpy.li/4-28>] (EN), foi marcante. Ned é tão profissional que forneceu uma transcrição completa da palestra, além dos slides e do vídeo.

"Character encoding and Unicode in Python: How to (□□□) with dignity" (*Codificação de caracteres e o Unicode no Python: como (□□□) com dignidade*) (slides [<https://fpy.li/4-1>], vídeo [<https://fpy.li/4-2>]) (EN) foi uma excelente palestra de Esther Nam e Travis Fischer na PyCon 2014, e foi onde encontrei a concisa epígrafe desse capítulo: "Humanos usam texto. Computadores falam em bytes."

Lennart Regebro—um dos revisores técnicos da primeira edição desse livro—compartilha seu "Useful Mental Model of Unicode (UMMU)" (*Modelo Mental Útil do Unicode*) em um post curto, "Unconfusing Unicode: What Is Unicode?" (*Desconfundindo o Unicode: O Que É O Unicode?*) [<https://fpy.li/4-31>] (EN). O Unicode é um padrão complexo, então o UMMU de Lennart é realmente um ponto de partida útil.

O "Unicode HOWTO" [<https://fpy.li/3k>] oficial na documentação de Python aborda o assunto por vários ângulos diferentes, de uma boa introdução histórica a detalhes de sintaxe, codecs, expressões regulares, nomes de arquivo, e boas práticas para E/S sensível ao Unicode (isto é, o sanduíche de Unicode), com vários links adicionais de referências em cada seção.

O Chapter 4, "Strings" (*Capítulo 4, "Strings"*) [<https://fpy.li/4-33>], do excelente livro *Dive into Python 3* [<https://fpy.li/4-34>] (EN) de Mark Pilgrim (Apress), também fornece uma ótima introdução ao suporte a Unicode no Python 3. No mesmo livro, o Capítulo 15 [<https://fpy.li/4-35>] descreve como a biblioteca Chardet foi portada de Python 2 para Python 3, um valioso estudo de caso, dado que a mudança do antigo tipo `str` para o novo `bytes` é a causa da maioria das dores da migração, e esta é uma preocupação central em uma biblioteca projetada para detectar codificações.

Se você conhece Python 2 mas é novo no Python 3, o artigo "What's New in Python 3.0" (*O quê há de novo no Python 3.0*) [<https://fpy.li/4-36>] (EN), de Guido van Rossum, tem 15 pontos resumindo as mudanças, com vários links. Guido inicia com uma afirmação brutal: "Tudo o que você achava que sabia sobre dados binários e Unicode mudou." O post de Armin Ronacher em seu blog, "The Updated Guide to Unicode on Python" (*O Guia Atualizado do Unicode no Python*) [<https://fpy.li/4-37>], é bastante profundo e realça algumas das armadilhas do Unicode no Python (Armin não é um grande fã de Python 3).

O capítulo 2 ("Strings and Text" *Strings e Texto*) do Python Cookbook, 3rd ed. [<https://fpy.li/pycook3>] (EN) (O'Reilly), de David Beazley e Brian K. Jones, tem várias receitas tratando de normalização de Unicode, sanitização de texto, e execução de operações orientadas para texto em sequências de bytes. O capítulo 5 trata de arquivos e E/S, e inclui a "Recipe 5.17. Writing Bytes to a Text File" (*Receita 5.17. Escrevendo Bytes em um Arquivo de Texto*), mostrando que sob qualquer arquivo de texto há sempre uma sequência binária que pode ser acessada diretamente quando necessário. Mais tarde no mesmo livro, o módulo `struct` é usado em "Recipe 6.11. Reading and Writing Binary Arrays of Structures" (*Receita 6.11. Lendo e Escrevendo Arrays Binárias de Estruturas*).

O blog "Python Notes" de Nick Coghlan tem dois posts muito relevantes para esse capítulo: "Python 3 and ASCII Compatible Binary Protocols" (*Python 3 e os Protocolos Binários Compatíveis com ASCII*) [<https://fpy.li/4-38>] (EN) e "Processing Text Files in Python 3" (*Processando Arquivos de Texto em Python 3*) [<https://fpy.li/4-39>] (EN). Fortemente recomendado.

Uma lista de codificações suportadas pelo Python está disponível em "Standard Encodings" [<https://fpy.li/3m>] (EN), na documentação do módulo `codecs`. Se você precisar obter aquela lista de dentro de um programa, pode ver como isso é feito no script `/Tools/unicode/listcodecs.py` [<https://fpy.li/4-41>], que acompanha o código-fonte do CPython.

Os livros *Unicode Explained* (*Unicode Explicado*) [<https://fpy.li/4-42>] (EN), de Jukka K. Korpela (O'Reilly) e *Unicode Demystified* (*Unicode Desmisticificado*) [<https://fpy.li/4-43>], de Richard Gillam (Addison-Wesley) não são específicos sobre Python, mas foram muito úteis para meu estudo dos conceitos do Unicode. *Programming with Unicode* (*Programando com Unicode*) [<https://fpy.li/4-44>], de Victor Stinner, é um livro gratuito e publicado pelo próprio autor tratando de Unicode em geral, bem como de ferramentas e APIs no contexto dos principais sistemas operacionais e algumas linguagens de programação, incluindo Python.

As páginas do W3C "Case Folding: An Introduction" (*Case Folding: Uma Introdução*) [<https://fpy.li/4-45>] (EN) e "Character Model for the World Wide Web: String Matching" (*O Modelo de Caracteres para a World Wide Web: Casamento de Strings*) [<https://fpy.li/4-15>] (EN) tratam de conceitos de normalização, a primeira uma suave introdução e a segunda uma nota de um grupo de trabalho escrita no seco jargão dos padrões—o mesmo tom do "Unicode Standard Annex #15—Unicode Normalization Forms" (*Anexo 15 do Padrão Unicode—Formas de Normalização do Unicode*) [<https://fpy.li/4-47>] (EN). A seção "Frequently Asked Questions, Normalization" (*Perguntas Frequentes, Normalização*) [<https://fpy.li/4-48>] (EN) do *Unicode.org* [<https://fpy.li/4-49>] é mais fácil de ler, bem como o "NFC FAQ" [<https://fpy.li/4-50>] (EN) de Mark Davis—autor de vários algoritmos do Unicode e presidente do Unicode Consortium quando essa seção foi escrita.

Em 2016, o Museu de Arte Moderna (MoMA) de New York adicionou à sua coleção o emoji original [<https://fpy.li/4-51>] (EN), os 176 emojis desenhados por Shigetaka Kurita em 1999 para a NTT DOCOMO—a provedora de telefonia móvel japonesa. Indo mais longe no passado, a *Emojipedia* [<https://fpy.li/4-52>] (EN) publicou o artigo "Correcting the Record on the First Emoji Set" (*Corrigindo o Registro [Histórico] sobre o Primeiro Conjunto de Emojis*) [<https://fpy.li/4-53>] (EN), atribuindo à SoftBank do Japão o mais antigo conjunto conhecido de emojis, implantado em celulares em 1997. O conjunto da SoftBank é a fonte de 90 emojis que hoje fazem parte do Unicode, incluindo o U+1F4A9 (PILE OF POO). O *emojitracker.com* [<https://fpy.li/4-54>], de Matthew Rothenberg, é um painel ativo mostrando a contagem do uso de emojis no Twitter, atualizado em tempo real. Quando escrevo isso, FACE WITH TEARS OF JOY (U+1F602) é o emoji mais popular no Twitter, com mais de 3.313.667.315 ocorrências registradas.

Ponto de vista

Nomes não-ASCII no código-fonte: você deveria usá-los?

Python 3 permite identificadores não-ASCII no código-fonte:

```
>>> ação = 'PBR' # ação = stock  
>>> ε = 10**-6    # ε = epsilon
```

Algumas pessoas não gostam dessa ideia. O argumento mais comum é que se limitar aos caracteres ASCII torna a leitura e a edição do código mais fácil para todo mundo. Esse argumento erra o alvo: você quer que seu código-fonte seja legível e editável pela audiência pretendida, e isso pode não ser "todo mundo". Se o código pertence a uma corporação multinacional, ou se é um código aberto e você deseja contribuidores de todo o mundo, os identificadores devem ser em inglês, e então tudo o que você precisa é do ASCII.

Mas se você é uma professora no Brasil, seus alunos vão achar mais fácil ler código com variáveis e nomes de função em português, e escritos corretamente. E eles não terão nenhuma dificuldade para digitar as cedilhas e as vogais acentuadas em seus teclados localizados.

Agora que Python pode interpretar nomes em Unicode, e que o UTF-8 é a codificação padrão para código-fonte, não vejo motivo para codificar identificadores em português sem acentos, como fazíamos no Python 2, por necessidade—a menos que seu código tenha que rodar também no Python 2. Se os nomes estão em português, excluir os acentos não vai tornar o código mais legível para ninguém.

Esse é meu ponto de vista como um brasileiro falante de português, mas acredito que se aplica além de fronteiras e a outras culturas: escolha a linguagem humana que torna o código mais legível para sua equipe, e então use todos os caracteres necessários para a ortografia correta.

O que é "texto puro"?

Para qualquer um que lide diariamente com texto em línguas diferentes do inglês "texto puro" não significa "ASCII". O Glossário do Unicode [<https://fpy.li/4-55>] (EN) define *texto puro* dessa forma:

Texto codificado por computador que consiste apenas de uma sequência de pontos de código de um dado padrão, sem qualquer outra informação estrutural ou de formatação.

Essa definição começa muito bem, mas não concordo com a parte após a vírgula. HTML é um ótimo exemplo de um formato de texto puro que inclui informação estrutural e de formatação. Mas ele ainda é texto puro, porque cada byte em um arquivo desse tipo está lá para representar um caractere de texto, em geral usando UTF-8. Não há bytes com significado não-textual, como você encontra em documentos .png ou .xls, onde a maioria dos bytes

representa valores binários compactos, como valores RGB ou números de ponto flutuante. No texto puro, números são representados como sequências de caracteres de dígitos.

Estou escrevendo esse livro em um formato de texto puro chamado—ironicamente—AsciiDoc [<https://fpy.li/4-56>], que é parte do conjunto de ferramentas da plataforma de publicação de livros Atlas [<https://fpy.li/4-57>] da O'Reilly. Os arquivos fonte de AsciiDoc são texto puro, mas aceitam UTF-8, e não só ASCII. Se fosse o contrário, escrever esse capítulo teria sido realmente doloroso. Apesar do nome equivocado, o AsciiDoc é muito bom.

O mundo do Unicode está em constante expansão e, nas fronteiras, as ferramentas de apoio nem sempre existem. Nem todos os caracteres que eu queria exibir estavam disponíveis nas fontes usadas para renderizar o livro. Por isso tive que usar capturas de tela do terminal em alguns exemplos desse capítulo. Por outro lado, os terminais do Ubuntu e do macOS exibem a maioria do texto Unicode muito bem—incluindo os caracteres japoneses para a palavra "mojibake": モジベイ.

Como os ponto de código numa str são representados na memória?

A documentação oficial de Python evita falar sobre como os pontos de código de uma str são armazenados na memória. Realmente, é um detalhe de implementação. Em teoria, não importa: qualquer que seja a representação interna, toda str precisa ser codificada para bytes na saída.

Na memória, Python 3 armazena cada str como uma sequência de pontos de código, usando um número fixo de bytes por ponto de código, para permitir o acesso direto eficiente a qualquer caractere ou fatia.

Desde Python 3.3, ao criar um novo objeto str o interpretador verifica os caracteres no objeto, e escolhe o layout de memória mais econômico que seja adequado para aquela str em particular: se existirem apenas caracteres na faixa latin1, aquela str vai usar apenas um byte por ponto de código. Caso contrário, podem ser usados dois ou quatro bytes por ponto de código, dependendo da str. Isso é uma simplificação; para saber todos os detalhes, dê uma olhada an PEP 393—Flexible String Representation (*Representação Flexível de Strings*) [<https://fpy.li/pep393>] (EN).

A representação flexível de strings é similar à forma como o tipo int funciona no Python 3: se um inteiro cabe em uma palavra da máquina, ele será armazenado em uma palavra da máquina. Caso contrário, o interpretador muda para uma representação de tamanho variável, como aquela do tipo long de Python 2. É bom ver as boas ideias se espalhando.

Entretanto, sempre podemos contar com Armin Ronacher para encontrar problemas no Python 3. Ele me explicou porque, na prática, essa não é uma ideia tão boa assim: basta um único RAT (U+1F400) para inflar um texto, que de outra forma seria inteiramente ASCII, e transformá-lo em um array sugadora de memória, usando quatro bytes por caractere, quando um byte seria o suficiente para todos os caracteres exceto o RAT. Além disso, por causa de todas as formas como os caracteres Unicode se combinam, a capacidade de buscar um caractere arbitrário pela posição é superestimada—e extrair fatias arbitrárias de texto Unicode é no mínimo ingênuo, e muitas vezes errado, produzindo mojibake. Com os emojis se tornando mais populares, esses problemas só vão piorar.

[1] Slide 12 da palestra "Character Encoding and Unicode in Python" (*Codificação de Caracteres e Unicode no Python*) na PyCon 2014 (slides [<https://fpy.li/4-1>] (EN), vídeo [<https://fpy.li/4-2>] (EN)).

[2] Python 2.6 e o 2.7 também tinham um bytes, mas ele era só um apelido (*alias*) para o tipo str.

[3] Trívia: O caractere ASCII "aspas simples", que por default Python usa como delimitador de strings, na verdade se chama APOSTROPHE no padrão Unicode. As aspas simples reais são assimétricas: a da esquerda é U+2018 e a da direita U+2019.

[4] Ele não funcionava de Python 3.0 ao 3.4, causando muitas dores de cabeça nos desenvolvedores que lidam com dados binários. A decisão está documentada na PEP 461—Adding % formatting to bytes and bytearray (*Aumentando formatação com % a bytes e bytearray*) [<https://fpy.li/pep461>]. (EN)

[5] A primeira vez que vi o termo "Unicode sandwich" (*sanduíche de Unicode*) foi na excelente apresentação de Ned Batchelder, "Pragmatic Unicode" (*Unicode pragmático*) (EN) [<https://fpy.li/4-10>] na US PyCon 2012.

[6] Fonte: "Windows Command-Line: Unicode and UTF-8 Output Text Buffer" (*A Linha de Comando do Windows: O Buffer de Saída de Texto para Unicode e UTF-8*) [<https://fpy.li/4-11>].

[7] Curiosamente, o símbolo de micro é considerado um "caractere de compatibilidade", mas o símbolo de ohm não. O resultado disso é que a NFC não toca no símbolo de micro, mas muda o símbolo de ohm para ômega maiúsculo, ao passo que a NFKC e a NFKD mudam tanto o ohm quanto o micro para caracteres gregos.

[8] NT: algo como "dobra" ou "mudança" de caixa.

[9] Sinais diacríticos afetam a ordenação apenas nos raros casos em que eles são a única diferença entre duas palavras—nesse caso, a palavra com o sinal diacrítico é colocada após a palavra sem o sinal na ordenação.

[10] De novo, eu não consegui encontrar uma solução, mas encontrei outras pessoas relatando o mesmo problema. Alex Martelli, um dos revisores técnicos, não teve problemas para usar setlocale e locale.strxfrm em seu Mac com o macOS 10.9. Em resumo: cada caso é um caso.

[11] Aquilo é uma imagem—não uma listagem de código—porque, no momento em que esse capítulo foi escrito, os emojis não tem um bom suporte no sistema de publicação digital da O'Reilly.

[12] Embora não tenha se saído melhor que o re para identificar dígitos nessa amostra em particular.

5. Fábricas de classes de dados

Classes de dados são como crianças. São um bom ponto de partida mas, para participarem como um objeto adulto, precisam assumir alguma responsabilidade.^[1]

— Martin Fowler & Kent Beck

Python oferece algumas formas de criar uma classe simples, apenas uma coleção de campos, com pouca ou nenhuma funcionalidade adicional. Esse padrão é conhecido como "classe de dados"—e `dataclasses` é um dos pacotes que suporta tal modelo. Este capítulo trata de três diferentes fábricas de classes que podem ser utilizadas como atalhos para escrever classes de dados:

`collections.namedtuple`

A forma mais simples—disponível desde o Python 2.6.

`typing.NamedTuple`

Uma alternativa que requer dicas de tipo nos campos—desde o Python 3.5, com a sintaxe `class` adicionada no 3.6.

`@dataclasses.dataclass`

Um decorador de classe que permite mais customização que as alternativas anteriores, acrescentando várias opções e, potencialmente, mais complexidade—desde o Python 3.7.

Após falar sobre essas fábricas de classes, vamos discutir o motivo de *classe de dados* ser também o nome um *code smell* (odor no código): um padrão de programação que pode ser um sintoma de um mau design orientado a objetos.

A classe `typing.TypedDict` pode parecer sintaticamente com um fábrica de classes de dados. Ela usa uma sintaxe similar, e é descrita pouco após `typing.NamedTuple` na documentação do módulo `typing` [<https://fpy.li/3n>] (EN) de Python 3.11.



Entretanto, `TypedDict` não cria classes concretas que possam ser instanciadas. Ela é apenas a sintaxe para escrever dicas de tipo para parâmetros de função e variáveis que aceitarão valores de mapeamentos como registros, enquanto suas chaves serão os nomes dos campos. Veremos mais sobre isso na [typeddict_sec] do [ch_more_types].

5.1. Novidades nesse capítulo

Este capítulo é novo nessa segunda edição do *Python Fluente*. A Seção 5.3 era parte do capítulo 2 da primeira edição, mas o restante do capítulo é inteiramente inédito.

Vamos começar por uma visão geral, por alto, das três fábricas de classes.

5.2. Visão geral das fábricas de classes de dados

Considere uma classe simples, representando um par de coordenadas geográficas, como aquela no Exemplo 64.

Exemplo 64. class/coordinates.py

```
class Coordinate:  
  
    def __init__(self, lat, lon):  
        self.lat = lat  
        self.lon = lon
```

A tarefa da classe `Coordinate` é manter os atributos latitude e longitude. Escrever o `__init__` padrão fica cansativo muito rápido, especialmente se sua classe tiver mais que alguns poucos atributos: cada um deles é mencionado três vezes! E aquele código repetitivo não fornece algumas funcionalidades básicas que esperamos de um objeto Python:

```
>>> from coordinates import Coordinate  
>>> moscow = Coordinate(55.76, 37.62)  
>>> moscow  
<coordinates.Coordinate object at 0x107142f10> ①  
>>> location = Coordinate(55.76, 37.62)  
>>> location == moscow ②  
False  
>>> (location.lat, location.lon) == (moscow.lat, moscow.lon) ③  
True
```

① O `__repr__` herdado de `object` não é muito útil.

② O `==` não faz sentido; o método `__eq__` herdado de `object` compara os IDs dos objetos.

③ Comparar duas coordenadas exige a comparação explícita de cada atributo.

As fábricas de classes de dados tratadas nesse capítulo fornecem automaticamente os métodos `__init__`, `__repr__`, e `__eq__` necessários, além de outros recursos úteis.

 Nenhuma das fábricas de classes discutidas aqui depende de herança para funcionar. Tanto `collections.namedtuple` quanto `typing.NamedTuple` criam subclasses de `tuple`. O `@dataclass` é um decorador de classe, não afeta de forma alguma a hierarquia de classes. Cada um deles utiliza técnicas diferentes de metaprogramação para injetar métodos e atributos de dados na classe em construção.

Aqui está uma classe `Coordinate` criada com uma `namedtuple`—uma função que fabrica uma subclasse de `tuple` com o nome e os campos especificados:

```
>>> from collections import namedtuple  
>>> Coordinate = namedtuple('Coordinate', 'lat lon')
```

```
>>> issubclass(Coordinate, tuple)
True
>>> moscow = Coordinate(55.756, 37.617)
>>> moscow
Coordinate(lat=55.756, lon=37.617) ①
>>> moscow == Coordinate(lat=55.756, lon=37.617) ②
True
```

① Um `__repr__` útil.

② Um `__eq__` que faz sentido.

A `typing.NamedTuple`, mais recente, oferece a mesma funcionalidade e acrescenta anotações de tipo a cada campo:

```
>>> import typing
>>> Coordinate = typing.NamedTuple('Coordinate',
...     [('lat', float), ('lon', float)])
>>> issubclass(Coordinate, tuple)
True
>>> typing.get_type_hints(Coordinate)
{'lat': <class 'float'>, 'lon': <class 'float'>}
```

Uma tupla nomeada e com dicas de tipo pode também ser construída passando os campos como argumentos nomeados, assim:



```
Coordinate = typing.NamedTuple('Coordinate', lat=float, lon=float)
```

Além de ser mais legível, essa forma permite fornecer o mapeamento de campos e tipos como `**fields_and_types`.

Desde Python 3.6, `typing.NamedTuple` pode também ser usada em uma instrução `class`, com as anotações de tipo escritas como descrito na PEP 526—Syntax for Variable Annotations (*Sintaxe para Anotações de Variáveis*) [<https://fpy.li/pep526>] (EN). É mais legível, e torna fácil sobrescrever métodos ou acrescentar métodos novos. O Exemplo 65 é a mesma classe `Coordinate`, com um par de atributos `float` e um `__str__` customizado, para mostrar a coordenada no formato 55.8°N, 37.6°E.

Exemplo 65. typing_namedtuple/coordinates.py

```
from typing import NamedTuple

class Coordinate(NamedTuple):
    lat: float
    lon: float

    def __str__(self):
        ns = 'N' if self.lat >= 0 else 'S'
        we = 'E' if self.lon >= 0 else 'W'
```

```
return f'{abs(self.lat):.1f}°{ns}, {abs(self.lon):.1f}°{we}'
```

Apesar de `NamedTuple` aparecer na declaração `class` como uma superclasse, ela não é. `typing.NamedTuple` usa a funcionalidade avançada de uma metaclasses^[2] para customizar a criação da classe do usuário. Confira isso:



```
>>> issubclass(Coordinate, typing.NamedTuple)
False
>>> issubclass(Coordinate, tuple)
True
```

No método `__init__` gerado por `typing.NamedTuple`, os campos aparecem como parâmetros e na mesma ordem em que aparecem na declaração `class`.

Assim como `typing.NamedTuple`, o decorador `dataclass` suporta a sintaxe da PEP 526 [<https://fpy.li/pep526>] (EN) para declarar atributos de instância. O decorador lê as anotações das variáveis e gera métodos automaticamente para sua classe. Como comparação, veja a classe `Coordinate` equivalente escrita com a ajuda do decorador `dataclass`, como mostra o Exemplo 66.

Exemplo 66. dataclass/coordinates.py

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Coordinate:
    lat: float
    lon: float

    def __str__(self):
        ns = 'N' if self.lat >= 0 else 'S'
        we = 'E' if self.lon >= 0 else 'W'
        return f'{abs(self.lat):.1f}°{ns}, {abs(self.lon):.1f}°{we}'
```

Observe que o corpo das classes no Exemplo 65 e no Exemplo 66 são idênticos—a diferença está na própria declaração `class`. O decorador `@dataclass` não depende de herança ou de uma metaclasses, então não deve interferir no uso desses mecanismos pelo usuário.^[3] A classe `Coordinate` no Exemplo 66 é uma subclasse de `object`.

5.2.1. Principais recursos

As diferentes fábricas de classes de dados tem muito em comum, como resume a Tabela 12.

Tabela 12. Recursos selecionados, comparando as três fábricas de classes de dados; x é uma instância de uma classe de dados daquele tipo

	namedtuple	NamedTuple	dataclass
instâncias mutáveis	NÃO	NÃO	SIM

	namedtuple	NamedTuple	dataclass
sintaxe de declaração de classe	NÃO	SIM	SIM
criar um dict	x._asdict()	x._asdict()	dataclasses.asdict(x)
obter nomes dos campos	x._fields	x._fields	[f.name for f in dataclasses.fields(x)]
obter defaults	x._field_defaults	x._field_defaults	[f.default for f in dataclasses.fields(x)]
obter tipos dos campos	N/A	x.__annotations__	x.__annotations__
nova instância com modificações	x.replace(...)	x.replace(...)	dataclasses.replace(x, ...)
nova classe durante a execução	namedtuple(...)	NamedTuple(...)	dataclasses.make_dataclass(...)



As classes criadas por `typing.NamedTuple` e `@dataclass` tem um atributo `__annotations__`, contendo as dicas de tipo dos campos. Entretanto, ler `__annotations__` diretamente não é recomendado. Em vez disso, a melhor prática recomendada para obter tal informação é chamar `inspect.get_annotations(MyClass)` [<https://fpy.li/3p>] (a partir de Python 3.10—EN) ou `typing.get_type_hints(MyClass)` [<https://fpy.li/3z>] (Python 3.5 a 3.9). Isso porque tais funções fornecem serviços adicionais, como a resolução de referências futuras nas dicas de tipo. Voltaremos a isso bem mais tarde neste livro, na [problems_annot_runtime_sec].

Vamos agora detalhar aqueles recursos principais.

5.2.1.1. Instâncias mutáveis

A diferença fundamental entre essas três fábricas de classes é que `collections.namedtuple` e `typing.NamedTuple` criam subclasses de `tuple`, e portanto as instâncias são imutáveis. Por default, `@dataclass` produz classes mutáveis. Mas o decorador aceita o argumento nomeado `frozen=True`, que aparece no Exemplo 66. Quando `frozen=True`, a classe vai gerar uma exceção se você tentar atribuir um valor a um campo após a instância ter sido inicializada.

5.2.1.2. Sintaxe de declaração de classe

Apenas `typing.NamedTuple` e `dataclass` suportam a sintaxe da instrução `class`, tornando mais fácil acrescentar métodos e docstrings à classe que está sendo criada.

5.2.1.3. Construir um dict

As duas variantes de tuplas nomeadas fornecem um método de instância (`._asdict`), para construir um objeto `dict` a partir dos campos de uma instância de classe de dados. O módulo `dataclasses` fornece uma função para fazer o mesmo: `dataclasses.asdict`.

5.2.1.4. Obter nomes dos campos e valores default

Todas as três fábricas de classes permitem que você obtenha os nomes dos campos e os valores default (que podem ser configurados para cada campo). Nas classes de tuplas nomeadas, aqueles metadados estão nos atributos de classe `._fields` e `._fields_defaults`. Você pode obter os mesmos metadados em uma classe decorada com `dataclass` usando a função `fields` do módulo `dataclasses`. Ela devolve uma tupla de objetos `Field` com vários atributos, incluindo `name` e `default`.

5.2.1.5. Obter os tipos dos campos

Classes definidas com a ajuda de `typing.NamedTuple` e `@dataclass` contêm um mapeamento dos nomes dos campos para seus tipos, o atributo de classe `__annotations__`. Como já mencionado, use a função `typing.get_type_hints` em vez de ler diretamente de `__annotations__`.

5.2.1.6. Nova instância com modificações

Dada uma instância de tupla nomeada `x`, a chamada `x._replace(**kwargs)` devolve uma nova instância com os valores de alguns atributos modificados, de acordo com os argumentos nomeados incluídos na chamada. A função de módulo `dataclasses.replace(x, **kwargs)` faz o mesmo para uma instância de uma classe decorada com `dataclass`.

5.2.1.7. Nova classe durante a execução

Apesar da sintaxe de declaração de classe ser mais legível, ela é estática, registrada no código-fonte. Um framework pode ter a necessidade de criar classes de dados durante a execução. Para tanto, podemos usar a sintaxe default de chamada de função de `collections.namedtuple`, que também é suportada por `typing.NamedTuple`. O módulo `dataclasses` oferece a função `make_dataclass`, com o mesmo propósito.

Após essa visão geral dos principais recursos das fábricas de classes de dados, vamos examinar cada uma delas mais de perto, começando pela mais simples.

5.3. Tuplas nomeadas clássicas

A função `collections.namedtuple` é uma fábrica que cria subclasses de `tuple`, acrescidas de nomes de campos, um nome de classe, e um `__repr__` informativo. Classes criadas com `namedtuple` podem ser usadas onde quer que uma tupla seja necessária. Na verdade, muitas funções da biblioteca padrão, que antes devolviam tuplas simples, agora devolvem tuplas nomeadas, sem afetar de forma alguma o código que as utiliza.



Cada instância de uma classe criada por `namedtuple` usa exatamente a mesma quantidade de memória usada por uma tupla, pois os nomes dos campos são armazenados na classe.

O Exemplo 67 mostra como poderíamos definir uma tupla nomeada para registrar informações sobre uma cidade.

Exemplo 67. Definindo e usando um tipo tupla nomeada

```
>>> from collections import namedtuple
>>> City = namedtuple('City', 'name country population coordinates') ①
>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667)) ②
>>> tokyo
City(name='Tokyo', country='JP', population=36.933, coordinates=(35.689722,
139.691667))
>>> tokyo.population ③
36.933
>>> tokyo.coordinates
(35.689722, 139.691667)
>>> tokyo[1]
'JP'
```

- ① São necessários dois parâmetros para criar uma tupla nomeada: um nome de classe e uma lista de nomes de campos, que podem ser passados como um iterável de strings ou como uma única string com os nomes delimitados por espaços.
- ② Na inicialização de uma instância, os valores dos campos devem ser passados como argumentos posicionais separados (uma tuple, por outro lado, é inicializada com um único iterável)
- ③ É possível acessar os campos por nome ou por posição.

Como uma subclasse de tuple, City herda métodos úteis, tal como `__eq__` e os métodos especiais para operadores de comparação—incluindo `__lt__`, que permite ordenar listas de instâncias de City.

Uma tupla nomeada oferece alguns atributos e métodos além daqueles herdados de tuple. O Exemplo 68 demonstra os mais úteis dentre eles: o atributo de classe `_fields`, o método de classe `_make(iterable)`, e o método de instância `_asdict()`.

Exemplo 68. Atributos e métodos das tuplas nomeadas (continuando do exemplo anterior)

```
>>> City._fields ①
('name', 'country', 'population', 'location')
>>> Coordinate = namedtuple('Coordinate', 'lat lon')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935, Coordinate(28.613889, 77.208889))
>>> delhi = City._make(delhi_data) ②
>>> delhi._asdict() ③
{'name': 'Delhi NCR', 'country': 'IN', 'population': 21.935,
'location': Coordinate(lat=28.613889, lon=77.208889)}
>>> import json
>>> json.dumps(delhi._asdict()) ④
'{"name": "Delhi NCR", "country": "IN", "population": 21.935,
"location": [28.613889, 77.208889]}'
```

- ① `_fields` é uma tupla com os nomes dos campos da classe.
- ② `_make()` cria uma City a partir de um iterável; `City(*delhi_data)` faria o mesmo.

- ③ `._asdict()` devolve um `dict` criado a partir da instância de tupla nomeada.
- ④ `._asdict()` é útil para serializar os dados no formato JSON, por exemplo.



Até Python 3.7, o método `_asdict` devolvia um `OrderedDict`. Desde Python 3.8, ele devolve um `dict` simples—o que não é problema, agora que podemos confiar na ordem de inserção das chaves. Se você precisar de um `OrderedDict`, a documentação do `_asdict` [<https://fpy.li/3q>] (EN) recomenda criar um com o resultado: `OrderedDict(x._asdict())`.

Desde Python 3.7, a `namedtuple` aceita o argumento nomeado `defaults`, fornecendo um iterável de N valores default para cada um dos N campos mais à direita na definição da classe. O Exemplo 69 mostra como definir uma tupla nomeada `Coordinate` com um valor default para o campo `reference`.

Exemplo 69. Atributos e métodos das tuplas nomeadas, continuando do Exemplo 68

```
>>> Coordinate = namedtuple('Coordinate', 'lat lon reference', defaults=['WGS84'])
>>> Coordinate(0, 0)
Coordinate(lat=0, lon=0, reference='WGS84')
>>> Coordinate._field_defaults
{'reference': 'WGS84'}
```

Na Seção 5.2.1.2, mencionei que é mais fácil programar métodos com a sintaxe de classe suportada por `typing.NamedTuple` and `@dataclass`. Você também pode acrescentar métodos a uma `namedtuple`, mas é um remendo. Pule a próxima caixinha se você não estiver interessada em gambiarra.

Remendando uma tupla nomeada para injetar um método

Lembre como criamos a classe `Card` class no Exemplo 1 do Capítulo 1:

```
Card = collections.namedtuple('Card', ['rank', 'suit'])
```

Mas tarde no Capítulo 1, escrevi uma função `spades_high`, para ordenação. Seria bom que aquela lógica estivesse encapsulada em um método de `Card`, mas acrescentar `spades_high` a `Card` sem usar uma declaração `class` exige um remendo rápido: definir a função e então atribuí-la a um atributo de classe. O Exemplo 70 mostra como isso é feito:

Exemplo 70. frenchdeck.doctest: Acrescentando um atributo de classe e um método a Card, a namedtuple da Seção 1.2

```
>>> Card.suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0) ①
>>> def spades_high(card):
...     rank_value = FrenchDeck.ranks.index(card.rank)                      ②
...     suit_value = card.suit_values[card.suit]
...     return rank_value * len(card.suit_values) + suit_value
...
>>> Card.overall_rank = spades_high                                         ③
>>> lowest_card = Card('2', 'clubs')
```

```
>>> highest_card = Card('A', 'spades')
>>> lowest_card.overall_rank()                                ④
0
>>> highest_card.overall_rank()
51
```

- ① Acrescenta um atributo de classe com valores para cada naipe.
- ② A função spades_high vai se tornar um método; o primeiro argumento não precisa ser chamado de self. Ao ser invocada como método, ela receberá a instância no primeiro argumento.
- ③ Anexa a função à classe Card como um método chamado overall_rank.
- ④ Funciona!

Para uma melhor legibilidade e para ajudar na manutenção futura, é muito melhor programar métodos dentro de uma declaração class. Mas é bom saber que essa gambiarra é possível, pois às vezes pode ser útil.^[4]

Isso foi apenas um pequeno desvio para demonstrar o poder de uma linguagem dinâmica.

Agora vamos ver a variante typing.NamedTuple.

5.4. Tuplas nomeadas com tipo

A classe Coordinate com um campo default, do Exemplo 69, pode ser escrita usando typing.NamedTuple, como se vê no Exemplo 71.

Exemplo 71. typing_namedtuple/coordinates2.py

```
from typing import NamedTuple

class Coordinate(NamedTuple):
    lat: float          ①
    lon: float
    reference: str = 'WGS84'  ②
```

- ① Todo campo de instância precisa ter uma anotação de tipo.
- ② O campo de instância reference é anotado com um tipo e um valor default.

As classes criadas por typing.NamedTuple não tem qualquer método além daqueles que collections.namedtuple também gera, e aqueles herdados de tuple. A única diferença é a presença do atributo de classe __annotations__, que Python ignora durante a execução do programa.

Dado que o principal recurso de typing.NamedTuple são as anotações de tipo, vamos dar uma rápida olhada nisso antes de continuar nossa exploração das fábricas de classes de dados.

5.5. Introdução às dicas de tipo

Dicas de tipo (*type hints*, também chamadas anotações de tipo) são formas de declarar o tipo esperado dos argumentos de funções, dos valores devolvidos em `return`, das variáveis e dos atributos de objetos.



Essa é uma introdução muito breve sobre dicas de tipo, suficiente apenas para que a sintaxe e o propósito das anotações usadas nas declarações de `typing.NamedTuple` e `@dataclass` façam sentido. Vamos tratar de anotações de tipo nas assinaturas de função no Capítulo 8 e de anotações mais avançadas no [ch_more_types]. Aqui vamos ver principalmente dicas com tipos embutidos simples, como `str`, `int`, e `float`, que são provavelmente os tipos mais comuns usados para anotar campos em classes de dados.

A primeira coisa que você precisa saber sobre dicas de tipo é que elas não são checadas pelo compilador de bytecode ou pelo interpretador de Python.

5.5.1. Nenhum efeito durante a execução

Pense nas dicas de tipo de Python como "documentação que pode ser utilizada por IDEs e checadores de tipos".

Isso porque as dicas de tipo não tem qualquer impacto sobre o comportamento de programas em Python durante a execução. Veja o Exemplo 72.

Exemplo 72. Python não checa dicas de tipo durante a execução de um programa

```
>>> import typing
>>> class Coordinate(typing.NamedTuple):
...     lat: float
...     lon: float
...
>>> trash = Coordinate('Ni!', None)
>>> print(trash)
Coordinate(lat='Ni!', lon=None)    ①
```

① Eu avisei: não há checagem de tipos durante a execução!

Se você incluir o código do Exemplo 72 em um módulo de Python, ela vai rodar e exibir uma `Coordinate` sem sentido, sem gerar erro ou aviso:

```
$ python3 nocheck_demo.py
Coordinate(lat='Ni!', lon=None)
```

O objetivo primário das dicas de tipo é ajudar os checadores de tipos externos, como o Mypy [<https://fpy.li/mypy>] ou o verificador de tipo embutido do PyCharm IDE [<https://fpy.li/5-5>]. Essas são ferramentas de análise estática: elas verificam código-fonte Python "parado", não código em execução.

Para observar o efeito das dicas de tipo, é necessário executar alguma dessas ferramentas sobre seu código—como um linter (*analisador de código*). Por exemplo, eis o quê o Mypy tem a dizer sobre o exemplo anterior:

```
$ mypy nocheck_demo.py
nocheck_demo.py:8: error: Argument 1 to "Coordinate" has
incompatible type "str"; expected "float"
nocheck_demo.py:8: error: Argument 2 to "Coordinate" has
incompatible type "None"; expected "float"
```

Como se vê, dada a definição de `Coordinate`, o Mypy percebe que os dois argumentos para criar um instância devem ser do tipo `float`, mas atribuição a `trash` usa uma `str` e `None`.^[5]

Vamos falar agora sobre a sintaxe e o significado das dicas de tipo.

5.5.2. Sintaxe de anotação de variáveis

Tanto `typing.NamedTuple` quanto `@dataclass` usam a sintaxe de anotações de variáveis definida na PEP 526 [<https://fpy.li/pep526>] (EN). Vamos ver aqui uma pequena introdução àquela sintaxe, no contexto da definição de atributos em declarações `class`.

A sintaxe básica da anotação de variáveis é :

```
var_name: some_type
```

A seção "Acceptable type hints" (*Dicas de tipo aceitáveis*) [<https://fpy.li/5-6>] na PEP 484 explica o que são tipos aceitáveis. Porém, no contexto da definição de uma classe de dados, os tipos mais úteis geralmente serão os seguintes:

- Uma classe concreta, por exemplo `str` ou `FrenchDeck`.
- Um tipo de coleção parametrizado, como `list[int]`, `tuple[str, float]`, etc.
- `typing.Optional`, por exemplo `Optional[str]`—para declarar um campo que pode ser uma `str` ou `None`, que também pode ser escrito como `str | None`.

Você também pode inicializar uma variável com um valor. Em uma declaração de `typing.NamedTuple` ou `@dataclass`, aquele valor se tornará o default daquele atributo quando o argumento correspondente for omitido na chamada de inicialização:

```
var_name: some_type = a_value
```

5.5.3. O significado das anotações de variáveis

Vimos na Seção 5.5.1 que dicas de tipo não tem qualquer efeito durante a execução de um programa. Mas no momento da importação—quando um módulo é carregado—o Python as lê para construir o dicionário `__annotations__`, que `typing.NamedTuple` e `@dataclass` então usam para

aprimorar a classe.

Vamos começar essa exploração no Exemplo 73, com uma classe simples, para mais tarde ver que recursos adicionais são acrescentados por `typing.NamedTuple` e `@dataclass`.

Exemplo 73. meaning/demo_plain.py: uma classe básica com dicas de tipo

```
class DemoPlainClass:  
    a: int          ①  
    b: float = 1.1  ②  
    c = 'spam'      ③
```

- ① a se torna um registro em `__annotations__`, mas é então descartada: nenhum atributo chamado a é criado na classe.
- ② b é salvo como uma anotação, e também se torna um atributo de classe com o valor 1.1.
- ③ c é só um bom e velho atributo de classe básico, sem uma anotação.

Podemos checar isso no console, primeiro lendo o `__annotations__` da `DemoPlainClass`, e daí tentando obter os atributos chamados a, b, e c:

```
>>> from demo_plain import DemoPlainClass  
>>> DemoPlainClass.__annotations__  
{'a': <class 'int'>, 'b': <class 'float'>}  
>>> DemoPlainClass.a  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: type object 'DemoPlainClass' has no attribute 'a'  
>>> DemoPlainClass.b  
1.1  
>>> DemoPlainClass.c  
'spam'
```

Observe que o atributo especial `__annotations__` é criado pelo interpretador para registrar dicas de tipo que aparecem no código-fonte—mesmo em uma classe básica.

O a sobrevive apenas como uma anotação, não se torna um atributo da classe, porque nenhum valor é atribuído a ele.^[6] O b e o c são armazenados como atributos de classe porque são vinculados a valores.

Nenhum desses três atributos estará em uma nova instância de `DemoPlainClass`. Se você criar um objeto `o = DemoPlainClass()`, `o.a` vai gerar um `AttributeError`, enquanto `o.b` e `o.c` vão obter os atributos de classe com os valores 1.1 e 'spam'—que é apenas o comportamento normal de um objeto Python.

5.5.3.1. Inspecionando uma `typing.NamedTuple`

Agora vamos examinar uma classe criada com `typing.NamedTuple` (Exemplo 74), usando os mesmos atributos e anotações da `DemoPlainClass` do Exemplo 73.

Exemplo 74. meaning/demo_nt.py: uma classe criada com typing.NamedTuple

```
import typing

class DemoNTClass(typing.NamedTuple):
    a: int      ①
    b: float = 1.1  ②
    c = 'spam'    ③
```

- ① a se torna uma anotação e também um atributo de instância.
- ② b é outra anotação, mas também se torna um atributo de instância com o valor default 1.1.
- ③ c é só um bom e velho atributo de classe comum; não será mencionado em nenhuma anotação.

Inspecionando a DemoNTClass, temos o seguinte:

```
>>> from demo_nt import DemoNTClass
>>> DemoNTClass.__annotations__
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoNTClass.a
<_collections._tuplegetter object at 0x101f0f940>
>>> DemoNTClass.b
<_collections._tuplegetter object at 0x101f0f8b0>
>>> DemoNTClass.c
'spam'
```

Aqui vemos as mesmas anotações para a e b que vimos no Exemplo 73. Mas typing.NamedTuple cria os atributos de classe a e b. O atributo c é apenas um atributo de classe simples, com o valor 'spam'.

Os atributos de classe a e b são descritores (*descriptors*)—um recurso avançado tratado no [ch_descriptors]. Por ora, pense neles como similares a um *getter* de propriedades do objeto^[7]: métodos que não exigem o operador explícito de chamada () para obter um atributo de instância. Na prática, isso significa que a e b vão funcionar como atributos de instância somente para leitura—o que faz sentido, se lembrarmos que instâncias de DemoNTClass são apenas tuplas enfeitadas, e tuplas são imutáveis.

A DemoNTClass também recebe uma docstring customizada:

```
>>> DemoNTClass.__doc__
'DemoNTClass(a, b)'
```

Vamos examinar uma instância de DemoNTClass:

```
>>> nt = DemoNTClass(8)
>>> nt.a
8
>>> nt.b
```

```
1.1  
>>> nt.c  
'spam'
```

Para criar `nt`, precisamos passar pelo menos o argumento `a` para `DemoNTClass`. O construtor também aceita um argumento `b`, mas como este último tem um valor default (de 1.1), ele é opcional. Como esperado, o objeto `nt` possui os atributos `a` e `b`; ele não tem um atributo `c`, mas Python obtém `c` da classe, como de hábito.

Se você tentar atribuir valores para `nt.a`, `nt.b`, `nt.c`, ou mesmo para `nt.z`, vai gerar uma exceção `AttributeError`, com mensagens de erro sutilmente diferentes. Tente fazer isso, e reflita sobre as mensagens.

5.5.3.2. Inspecionando uma classe decorada com `@dataclass`

Vamos agora examinar o Exemplo 75.

Exemplo 75. meaning/demo_dc.py: uma classe decorada com @dataclass

```
from dataclasses import dataclass  
  
@dataclass  
class DemoDataClass:  
    a: int          ①  
    b: float = 1.1  ②  
    c = 'spam'      ③
```

① `a` se torna uma anotação, e também um atributo de instância controlado por um descritor.

② `b` é outra anotação, e também se torna um atributo de instância com um descritor e um valor default de 1.1.

③ `c` é apenas um atributo de classe comum; nenhuma anotação se refere a ele.

Podemos então verificar o `__annotations__`, o `__doc__`, e os atributos `a`, `b`, `c` em `DemoDataClass`:

```
>>> from demo_dc import DemoDataClass  
>>> DemoDataClass.__annotations__  
{'a': <class 'int'>, 'b': <class 'float'>}  
>>> DemoDataClass.__doc__  
'DemoDataClass(a: int, b: float = 1.1)'  
>>> DemoDataClass.a  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: type object 'DemoDataClass' has no attribute 'a'  
>>> DemoDataClass.b  
1.1  
>>> DemoDataClass.c  
'spam'
```

O `__annotations__` e o `__doc__` não guardam surpresas. Entretanto, não há um atributo chamado `a` em `DemoDataClass`—diferente do que ocorre na `DemoNTClass` do Exemplo 74, que inclui um descritor para obter a das instâncias da classe, como atributos somente para leitura (aquele misterioso `<collections.tuplegetter>`). Isso ocorre porque o atributo `a` só existirá nas instâncias de `DemoDataClass`. Será um atributo público, que poderemos obter e definir, a menos que a classe seja *frozen*. Mas `b` e `c` existem como atributos de classe, com `b` contendo o valor default para o atributo de instância `b`, enquanto `c` é apenas um atributo de classe que não será vinculado a instâncias.

Vejamos como se parece uma instância de `DemoDataClass`:

```
>>> dc = DemoDataClass(9)
>>> dc.a
9
>>> dc.b
1.1
>>> dc.c
'spam'
```

Novamente, `a` e `b` são atributos de instância, e `c` é um atributo de classe obtido através da instância.

Como mencionado, instâncias de `DemoDataClass` são mutáveis—e nenhuma checagem de tipos é realizada durante a execução:

```
>>> dc.a = 10
>>> dc.b = 'oops'
```

Podemos fazer atribuições ainda mais ridículas:

```
>>> dc.c = 'whatever'
>>> dc.z = 'secret stash'
```

Agora a instância `dc` tem um atributo `c`—mas isso não muda o atributo de classe `c`. E podemos adicionar um novo atributo `z`. Isso é o comportamento normal de Python: instâncias podem ter seus próprios atributos, que não aparecem na classe.^[8]

5.6. Mais detalhes sobre `@dataclass`

Até agora, só vimos exemplos simples do uso de `@dataclass`. Esse decorador aceita vários argumentos nomeados. Esta é sua assinatura:

```
@dataclass(*, init=True, repr=True, eq=True, order=False,
           unsafe_hash=False, frozen=False)
```

O `*` na primeira posição significa que os parâmetros restantes são todos parâmetros nomeados. A Tabela 13 os descreve.

Tabela 13. Parâmetros nomeados aceitos pelo decorador `@dataclass`

opção	efeito	default	notas
<code>init</code>	Gera o <code>__init__</code>	True	Ignorado se o <code>__init__</code> for implementado pelo usuário.
<code>repr</code>	Gera o <code>__repr__</code>	True	Ignorado se o <code>__repr__</code> for implementado pelo usuário.
<code>eq</code>	Gera o <code>__eq__</code>	True	Ignorado se o <code>__eq__</code> for implementado pelo usuário.
<code>order</code>	Gera <code>__lt__</code> , <code>__le__</code> , <code>__gt__</code> , <code>__ge__</code>	False	Se True, causa uma exceção se <code>eq=False</code> , ou se qualquer dos métodos de comparação que seriam gerados estiver definido ou herdado.
<code>unsafe_hash</code>	Gera o <code>__hash__</code>	False	Semântica complexa e várias restrições—veja a documentação de <code>dataclass</code> [https://fpy.li/3r].
<code>frozen</code>	Cria instâncias "imutáveis"	False	As instâncias estarão razoavelmente protegidas contra mudanças acidentais, mas não serão realmente imutáveis. ^[9]

Os defaults são realmente as configurações mais úteis para os casos de uso mais comuns. As opções mais prováveis de serem modificadas de seus defaults são:

`frozen=True`

Protege as instâncias da classe de modificações acidentais.

`order=True`

Permite ordenar as instâncias da classe de dados.

Dada a natureza dinâmica de objetos Python, não é muito difícil para um programador curioso contornar a proteção oferecida por `frozen=True`. Mas os truques necessários são fáceis de perceber em uma revisão do código.

Se tanto o argumento `eq` quanto o `frozen` forem `True`, `@dataclass` produz um método `__hash__` adequado, e daí as instâncias serão *hashable*. O `__hash__` gerado usará dados de todos os campos que não forem individualmente excluídos usando uma opção de campo, que veremos na Seção 5.6.1. Se `frozen=False` (o default), `@dataclass` definirá `__hash__` como `None`, sinalizando que as instâncias não são *hashable*, e portanto sobrescrevendo o `__hash__` de qualquer superclasse.

A PEP 557—Data Classes (*Classe de Dados*) [<https://fpy.li/pep557>] (EN) diz o seguinte sobre `unsafe_hash`:

Apesar de não ser recomendado, você pode forçar `@dataclass` a criar um método `__hash__` com `unsafe_hash=True`. Pode ser esse o caso, se sua classe for logicamente imutável e mesmo assim possa ser modificada. Este é um caso de uso especializado e deve ser considerado com cuidado.

Deixo o `unsafe_hash` por aqui. Se você achar que precisa usar essa opção, leia a documentação de `dataclasses.dataclass` [<https://fpy.li/3r>].

Outras customizações da classe de dados gerada podem ser feitas no nível dos campos.

5.6.1. Opções de campo

Já vimos a opção de campo mais básica: fornecer (ou não) um valor default junto com a dica de tipo. Os campos de instância declarados se tornarão parâmetros no `__init__` gerado. Python não permite parâmetros sem um default após parâmetros com defaults. Então, após declarar um campo com um valor default, cada um dos campos seguintes deve também ter um default.

Valores default mutáveis são a fonte mais comum de bugs entre desenvolvedores Python iniciantes. Em definições de função, um valor default mutável é facilmente corrompido, quando uma invocação da função modifica o default, mudando o comportamento nas invocações posteriores—um tópico que vamos explorar na Seção 6.5.1 (no Capítulo 6). Atributos de classe são frequentemente usados como valores default de atributos para instâncias, inclusive em classes de dados. E o `@dataclass` usa os valores default nas dicas de tipo para gerar parâmetros com defaults no `__init__`. Para prevenir bugs, o `@dataclass` rejeita a definição de classe que aparece no Exemplo 76.

Exemplo 76. dataclass/club_wrong.py: essa classe gera um ValueError

```
@dataclass
class ClubMember:
    name: str
    guests: list = []
```

Se você carregar o módulo com aquela classe `ClubMember`, o resultado será esse:

```
$ python3 club_wrong.py
Traceback (most recent call last):
  File "club_wrong.py", line 4, in <module>
    class ClubMember:
        ...several lines omitted...
ValueError: mutable default <class 'list'> for field guests is not allowed:
use default_factory
```

A mensagem do `ValueError` explica o problema e sugere uma solução: usar a `default_factory`. O Exemplo 77 mostra como corrigir a `ClubMember`.

Exemplo 77. dataclass/club.py: essa definição de ClubMember funciona

```
from dataclasses import dataclass, field

@dataclass
class ClubMember:
    name: str
```

```
guests: list = field(default_factory=list)
```

No campo `guests` do Exemplo 77, em vez de uma lista literal, o valor `default` é definido chamando a função `dataclasses.field` com `default_factory=list`.

O parâmetro `default_factory` permite que você forneça uma função, classe ou qualquer outro invocável, que será chamado sem argumentos, para gerar um valor `default` a cada vez que uma instância da classe de dados for criada. Dessa forma, cada instância de `ClubMember` terá sua própria `list`—ao invés de todas as instâncias compartilharem a mesma `list` da classe, que raramente é o que queremos, e muitas vezes é um bug.



É bom que `@dataclass` rejeite definições de classe com uma `list` default em um campo. Entretanto, entenda que isso é uma solução parcial, que se aplica apenas a `list`, `dict` e `set`. Outros valores mutáveis usados como `default` não serão rejeitados por `@dataclass`. É sua responsabilidade entender o problema e se lembrar de usar uma `factory` default para definir valores `default` mutáveis.

Se você estudar a documentação do módulo `dataclasses` [<https://fpy.li/3s>], verá um campo `list` definido com uma sintaxe nova, como no Exemplo 78.

Exemplo 78. dataclass/club_generic.py: essa definição de ClubMember é mais precisa

```
from dataclasses import dataclass, field

@dataclass
class ClubMember:
    name: str
    guests: list[str] = field(default_factory=list) ①
```

① `list[str]` significa "uma lista de `str`."

A nova sintaxe `list[str]` é um tipo genérico parametrizado: desde Python 3.9, o tipo embutido `list` aceita aquela notação com colchetes para especificar o tipo dos itens da lista.



Antes de Python 3.9, as coleções embutidas não suportavam a notação de tipagem genérica. Como uma solução temporária, há tipos correspondentes de coleções no módulo `typing`. Se você precisa de uma dica de tipo para uma `list` parametrizada no Python 3.8 ou anterior, você tem que importar e usar o tipo `List` de `typing`: `List[str]`. Leia mais sobre isso na caixa Suporte a tipos de coleção descontinuados.

Vamos tratar dos tipos genéricos no Capítulo 8. Por ora, observe que o Exemplo 77 e o Exemplo 78 estão ambos corretos, e que o verificador de tipagem Mypy não reclama de nenhuma das duas definições de classe.

A diferença é que aquele `guests: list` significa que `guests` pode ser uma `list` de objetos de qualquer tipo, enquanto `guests: list[str]` diz que `guests` deve ser uma `list` na qual cada item é uma `str`. Isso permite que o checador de tipos encontre (alguns) bugs em código que insira itens inválidos na lista, ou que leia itens dela.

A `default_factory` é possivelmente a opção mais comum da função `field`, mas há várias outras, listadas na Tabela 14.

Tabela 14. Argumentos nomeados aceitos pela função `field`

opção	default	significado
default	<code>_MISSING_TYPE</code> <small>[10]</small>	Valor default para o campo
default_factory	<code>_MISSING_TYPE</code>	função com 0 parâmetros usada para produzir um valor default
init	<code>True</code>	Incluir o campo nos parâmetros de <code>__init__</code>
repr	<code>True</code>	Incluir o campo em <code>__repr__</code>
compare	<code>True</code>	Usar o campo nos métodos de comparação <code>__eq__</code> , <code>__lt__</code> , etc.
hash	<code>None</code> ^[11]	Incluir o campo no cálculo de <code>__hash__</code>
metadata	<code>None</code>	Mapeamento com dados definidos pelo usuário; ignorado por <code>@dataclass</code>

A opção `default` existe porque a chamada a `field` toma o lugar do valor `default` na anotação do campo. Se você quisesse criar um campo `athlete` com o valor `default=False`, e também omitir aquele campo do método `__repr__`, escreveria o seguinte:

```
@dataclass
class ClubMember:
    name: str
    guests: list = field(default_factory=list)
    athlete: bool = field(default=False, repr=False)
```

5.6.2. Processamento pós-inicialização

O método `__init__` gerado por `@dataclass` apenas recebe os argumentos passados e os atribui (ou seus valores `default`, se o argumento não estiver presente) aos atributos de instância, que são campos da instância. Mas pode ser necessário fazer mais que isso para inicializar a instância. Se for esse o caso, você pode fornecer um método `__post_init__`. Quando esse método existir, `@dataclass` acrescentará código ao `__init__` gerado para invocar `__post_init__` como o último passo da inicialização.

Casos de uso comuns para `__post_init__` são validação e o cálculo de valores de campos baseado em outros campos. Vamos estudar um exemplo simples, que usa `__post_init__` pelos dois motivos.

Primeiro, dê uma olhada no comportamento esperado de uma subclasse de `ClubMember`, chamada `HackerClubMember`, como descrito por doctests no Exemplo 79.

Exemplo 79. `dataclass/hackerclub.py: doctests para HackerClubMember`

```
"""
``HackerClubMember`` objects accept an optional ``handle`` argument::
```

```
>>> anna = HackerClubMember('Anna Ravenscroft', handle='AnnaRaven')
>>> anna
HackerClubMember(name='Anna Ravenscroft', guests=[], handle='AnnaRaven')
```

If ``handle`` is omitted, it's set to the first part of the member's name::

```
>>> leo = HackerClubMember('Leo Rochael')
>>> leo
HackerClubMember(name='Leo Rochael', guests=[], handle='Leo')
```

Members must have a unique handle. The following ``leo2`` will not be created, because its ``handle`` would be 'Leo', which was taken by ``leo``:::

```
>>> leo2 = HackerClubMember('Leo DaVinci')
Traceback (most recent call last):
...
ValueError: handle 'Leo' already exists.
```

To fix, ``leo2`` must be created with an explicit ``handle``::

```
>>> leo2 = HackerClubMember('Leo DaVinci', handle='Neo')
>>> leo2
HackerClubMember(name='Leo DaVinci', guests=[], handle='Neo')
"""
```

Observe que precisamos fornecer `handle` como um argumento nomeado, pois `HackerClubMember` herda `name` e `guests` de `ClubMember`, e acrescenta o campo `handle`. A docstring gerada para `HackerClubMember` mostra a ordem dos campos na chamada de inicialização:

```
>>> HackerClubMember.__doc__
'HackerClubMember(name: str, guests: list = <factory>, handle: str = '')'
```

Aqui `<factory>` apenas indica que um invocável vai produzir o valor default para `guests` (no nosso caso, a fábrica é a classe `list`). O ponto é o seguinte: para fornecer um `handle` mas não `guests`, precisamos passar `handle` como um argumento nomeado.

A seção "Herança [https://fpy.li/3t] na documentação do módulo `dataclasses` explica como a ordem dos campos é analisada quando existem vários níveis de herança.



No [ch_inheritance] vamos falar sobre o uso indevido da herança, especialmente quando as superclasses não são abstratas. Criar uma hierarquia de classes de dados é, em geral, uma má ideia, mas nos serviu bem aqui para tornar o Exemplo 80 mais curto, e permitir que nos concentrássemos na declaração do campo `handle` e na validação com `__post_init__`.

O Exemplo 80 mostra a implementação.

Exemplo 80. *dataclass/hackerclub.py*: código para `HackerClubMember`

```
from dataclasses import dataclass
from club import ClubMember

@dataclass
class HackerClubMember(ClubMember):
    all_handles = set()                                ①
    handle: str = ''                                  ②
    def __post_init__(self):
        cls = self.__class__                            ④
        if self.handle == '':
            self.handle = self.name.split()[0]          ⑤
        if self.handle in cls.all_handles:             ⑥
            msg = f'handle {self.handle!r} already exists.'
            raise ValueError(msg)
        cls.all_handles.add(self.handle)                ⑦
```

- ① `HackerClubMember` estende `ClubMember`.
- ② `all_handles` é um atributo de classe.
- ③ `handle` é um campo de instância do tipo `str`, com uma string vazia como valor default; isso o torna opcional.
- ④ Obtém a classe da instância.
- ⑤ Se `self.handle` é a string vazia, a define como a primeira parte de `name`.
- ⑥ Se `self.handle` está em `cls.all_handles`, gera um `ValueError`.
- ⑦ Insere o novo `handle` em `cls.all_handles`.

O Exemplo 80 funciona como esperado, mas não é satisfatório pra um verificador estático de tipos. A seguir veremos a razão disso, e como resolver o problema.

5.6.3. Atributos de classe tipados

Se verificarmos os tipos de Exemplo 80 com o Mypy, seremos repreendidos:

```
$ mypy hackerclub.py
hackerclub.py:37: error: Need type annotation for "all_handles"
(hint: "all_handles: Set[<type>] = ...")
Found 1 error in 1 file (checked 1 source file)
```

Infelizmente, a dica fornecida pelo Mypy (versão 0.910 quando essa seção foi revisada) não é muito útil no contexto do uso de `@dataclass`. Primeiro, ele sugere usar `Set`, mas desde Python 3.9 podemos usar `set`—sem a necessidade de importar `Set` de `typing`. E mais importante, se acrescentarmos uma dica de tipo como `set[...]` a `all_handles`, `@dataclass` vai encontrar essa anotação e transformar `all_handles` em um campo de instância. Vimos isso acontecer na Seção 5.5.3.2.

A forma de contornar esse problema definida na PEP 526—Syntax for Variable Annotations (*Sintaxe para Anotações de Variáveis*) [<https://fpy.li/5-11>] (EN) é horrível. Para criar uma variável de classe com uma dica de tipo, precisamos usar um pseudo-tipo chamado `typing.ClassVar`, que aproveita a notação de tipos genéricos ([]) para definir o tipo da variável e também para declará-la como um atributo de classe.

Para fazer felizes tanto o checador de tipos quanto o `@dataclass`, deveríamos declarar o `all_handles` do Exemplo 80 assim:

```
all_handles: ClassVar[set[str]] = set()
```

Aquela dica de tipo está dizendo o seguinte:

`all_handles` é um atributo de classe do tipo `set-de-str`, com um `set` vazio como valor default.

Para escrever aquela anotação precisamos também importar `ClassVar` do módulo `typing`.

O decorador `@dataclass` não se importa com os tipos nas anotações, exceto em dois casos, e este é um deles: se o tipo for `ClassVar`, um campo de instância não será gerado para aquele atributo.

O outro caso onde o tipo de um campo é relevante para `@dataclass` é quando declaramos *variáveis apenas de inicialização*, nosso próximo tópico.

5.6.4. Variáveis de inicialização que não são campos

Algumas vezes pode ser necessário passar para `__init__` argumentos que não são campos de instância. Tais argumentos são chamados "argumentos apenas de inicialização" (*init-only variables*) pela documentação de `dataclasses` [<https://fpy.li/3v>]. Para declarar um argumento desses, o módulo `dataclasses` oferece o pseudo-tipo `InitVar`, que usa a mesma sintaxe de `typing.ClassVar`. O exemplo dado na documentação é uma classe de dados com um campo inicializado a partir de um banco de dados, e o objeto banco de dados precisa ser passado para o `__init__`.

O Exemplo 81 mostra o código que ilustra a seção "Variáveis de inicialização apenas" [<https://fpy.li/3v>].

Exemplo 81. Exemplo da documentação do módulo `dataclasses` [<https://fpy.li/3v>]

```
@dataclass
class C:
    i: int
    j: int | None = None
    database: InitVar[DatabaseType | None] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

Veja como o atributo `database` é declarado. `InitVar` vai evitar que `@dataclass` trate `database` como um campo normal. Ele não será definido como um atributo de instância, e a função `dataclasses.fields` não vai listá-lo. Entretanto, `database` será um dos argumentos aceitos pelo `__init__` gerado, e também será passado para o `__post_init__`. Ao escrever aquele método é preciso adicionar o argumento correspondente à sua assinatura, como mostra o Exemplo 81.

Esse longo tratamento de `@dataclass` cobriu os recursos mais importantes desse decorador—alguns deles apareceram em seções anteriores, como na Seção 5.2.1, onde falamos em paralelo das três fábricas de classes de dados. A documentação de `dataclasses` [<https://fpy.li/3v>] e a PEP 526—Syntax for Variable Annotations (*Sintaxe para Anotações de Variáveis*) [<https://fpy.li/pep526>] (EN) têm todos os detalhes.

Na próxima seção apresento um exemplo mais completo com o `@dataclass`.

5.6.5. Exemplo de `@dataclass`: o registro de recursos do Dublin Core

Frequentemente as classes criadas com o `@dataclass` vão ter mais campos que os exemplos muito curtos apresentados até aqui. O Dublin Core [<https://fpy.li/5-12>] (EN) oferece a fundação para um exemplo mais típico de `@dataclass`.

O Dublin Core é um esquema de metadados que visa descrever objetos digitais, como vídeos, sons, imagens, textos e sites na web. Aplicações de Dublin Core utilizam XML e o RDF (Resource Description Framework).^[12]

— Dublin Core na Wikipedia

O padrão define 15 campos opcionais; a classe `Resource`, no Exemplo 82, usa 8 deles.

Exemplo 82. `dataclass/resource.py`: código de `Resource`, uma classe baseada nos termos do Dublin Core

```
from dataclasses import dataclass, field
from typing import Optional
from enum import Enum, auto
from datetime import date

class ResourceType(Enum): ①
    BOOK = auto()
    EBOOK = auto()
    VIDEO = auto()

@dataclass
class Resource:
    """Media resource description."""
    identifier: str ②
    title: str = '<untitled>' ③
    creators: list[str] = field(default_factory=list)
    date: Optional[date] = None ④
    type: ResourceType = ResourceType.BOOK ⑤
```

```

description: str = ''
language: str = ''
subjects: list[str] = field(default_factory=list)

```

- ① Esse Enum vai fornecer valores de um tipo seguro para o campo Resource.type.
- ② identifier é o único campo obrigatório.
- ③ title é o primeiro campo com um default. Isso obriga todos os campos abaixo dele a fornecerem defaults.
- ④ O valor de date pode ser uma instância de datetime.date ou None.
- ⑤ O default do campo type é ResourceType.BOOK.

O Exemplo 83 mostra um doctest, para demonstrar como um registro Resource aparece no código.

Exemplo 83. dataclass/resource.py: código de Resource, uma classe baseada nos termos do Dublin Core

```

>>> description = 'Improving the design of existing code'
>>> book = Resource('978-0-13-475759-9', 'Refactoring, 2nd Edition',
...     ['Martin Fowler', 'Kent Beck'], date(2018, 11, 19),
...     ResourceType.BOOK, description, 'EN',
...     ['computer programming', 'OOP'])
>>> book # doctest: +NORMALIZE_WHITESPACE
Resource(identifier='978-0-13-475759-9', title='Refactoring, 2nd Edition',
creators=['Martin Fowler', 'Kent Beck'], date=datetime.date(2018, 11, 19),
type=<ResourceType.BOOK: 1>, description='Improving the design of existing code',
language='EN', subjects=['computer programming', 'OOP'])

```

O `__repr__` gerado pelo `@dataclass` é razoável, mas podemos torná-lo mais legível. Esse é o formato que queremos para `repr(book)`:

```

>>> book # doctest: +NORMALIZE_WHITESPACE
Resource(
    identifier = '978-0-13-475759-9',
    title = 'Refactoring, 2nd Edition',
    creators = ['Martin Fowler', 'Kent Beck'],
    date = datetime.date(2018, 11, 19),
    type = <ResourceType.BOOK: 1>,
    description = 'Improving the design of existing code',
    language = 'EN',
    subjects = ['computer programming', 'OOP'],
)

```

O Exemplo 84 é o código para o `__repr__`, produzindo o formato que aparece no trecho anterior. Esse exemplo usa `dataclass.fields` para obter os nomes dos campos da classe de dados.

Exemplo 84. dataclass/resource_repr.py: código para o método `__repr__`, implementado na classe Resource do Exemplo 82

```
def __repr__(self):
    cls = self.__class__
    cls_name = cls.__name__
    indent = ' ' * 4
    res = [f'{cls_name}(']                                ①
    for f in fields(cls):
        value = getattr(self, f.name)                  ②
        res.append(f'{indent}{f.name} = {value!r},')     ③
    res.append(')')                                    ④
    return '\n'.join(res)                            ⑤
```

- ① Inicializa a lista `res`, para criar a string de saída com o nome da classe e o parênteses abrindo.
- ② Para cada campo `f` na classe...
- ③ ...obtém o atributo nomeado da instância.
- ④ Anexa uma linha indentada com o nome do campo e `repr(value)`—é isso que o `!r` faz.
- ⑤ Acrescenta um parênteses fechando.
- ⑥ Cria uma string de múltiplas linhas a partir de `res`, e devolve essa string.

Com esse exemplo, inspirado pelo espírito de Dublin, Ohio, concluímos nosso passeio pelas fábricas de classes de dados de Python.

Classes de dados são úteis, mas podem estar sendo usadas de forma excessiva em seu projeto. A próxima seção explica isso.

5.7. A classe de dados como *cheiro no código*

Independente de você implementar uma classe de dados escrevendo todo o código ou aproveitando as facilidades oferecidas por alguma das fábricas de classes descritas nesse capítulo, fique alerta: isso pode sinalizar um problema em seu design.

No livro *Refactoring: Improving the Design of Existing Code, 2nd ed.* [<https://fpy.li/42>] (Addison-Wesley), Martin Fowler e Kent Beck apresentam um catálogo de "cheiros no código"^[13]—padrões no código que podem indicar a necessidade de refatoração. O verbete entitulado "Data Class" (*Classe de Dados*) começa assim:

Essas são classes que tem campos, métodos para obter e definir os campos, e nada mais. Tais classes são recipientes burros de dados, e muitas vezes são manipuladas de forma excessivamente detalhada por outras classes.

No site pessoal de Fowler, há um post muito esclarecedor chamado "Code Smell" (*Cheiro no Código*) [<https://fpy.li/5-14>] (EN). Esse texto é muito relevante para nossa discussão, pois o autor usa a *classe de dados* como um exemplo de *cheiro no código*, e sugere alternativas para lidar com ele. Abaixo está a

tradução integral daquele artigo.^[14]

Cheiros no Código

De Martin Fowler

Um cheiro no código é um indicação superficial que frequentemente corresponde a um problema mais profundo no sistema. O termo foi inventado por Kent Beck, enquanto ele me ajudava com meu livro, *Refactoring* [<https://fpy.li/5-15>].

A rápida definição acima contém um par de detalhes sutis. Primeiro, um cheiro é, por definição, algo rápido de detectar—é "cheiroso", como eu disse recentemente. Um método longo é um bom exemplo disso—basta olhar o código e ver mais de uma dúzia de linhas de Java para meu nariz se contrair.

O segundo detalhe é que cheiros nem sempre indicam um problema. Alguns métodos longos são bons. É preciso ir mais fundo para ver se há um problema subjacente ali. Cheiros não são inherentemente ruins por si só—eles frequentemente são o indicador de um problema, não o problema propriamente dito.

Os melhores cheiros são algo fácil de detectar e que, na maioria das vezes, leva a problemas realmente interessantes. Classes de dados (classes contendo só dados e nenhum comportamento [próprio]) são um bom exemplo. Você olha para elas e se pergunta que comportamento deveria fazer parte daquela classe. Então você começa a refatorar, para incluir ali aquele comportamento. Muitas vezes, algumas perguntas simples e essas refatorações iniciais são um passo vital para transformar um objeto anêmico em alguma coisa que realmente tenha classe.

Uma coisa boa sobre cheiros é sua facilidade de detecção por pessoas inexperientes, mesmo aquelas pessoas que não conhecem o suficiente para avaliar se há mesmo um problema ou, se existir, para corrigi-lo. Soube de um líder de uma equipe de desenvolvimento que elege um "cheiro da semana", e pede às pessoas que procurem aquele cheiro e o apresentem para colegas mais experientes. Fazer isso com um cheiro por vez é uma ótima maneira de ensinar gradualmente os membros da equipe a serem programadores melhores.

A principal ideia da programação orientada a objetos é manter o comportamento e os dados juntos, na mesma unidade de código: uma classe. Se uma classe é largamente utilizada mas não tem qualquer comportamento próprio significativo, é bem provável que o código que interage com as instâncias dessa classe esteja espalhado (ou mesmo duplicado) em métodos e funções ao longo de todo o sistema—uma receita para dores de cabeça na manutenção. Por isso, as refatorações de Fowler para lidar com uma classe de dados envolvem trazer responsabilidades de volta para a classe.

Levando o que foi dito acima em consideração, há alguns cenários comuns onde faz sentido ter um classe de dados com pouco ou nenhum comportamento.

5.7.1. A classe de dados como um esboço

Nesse cenário, a classe de dados é uma implementação simplista inicial de uma classe, para dar início a um novo projeto ou módulo. Com o tempo, a classe deve ganhar seus próprios métodos, deixando de depender de métodos de outras classes para operar sobre suas instâncias. O esboço é temporário; ao final do processo, sua classe pode se tornar totalmente independente da fábrica usada inicialmente para criá-la.

Python também é muito usado para resolução rápida de problemas e para experimentação, e nesses casos faz sentido preservar o esboço.

5.7.2. A classe de dados como representação intermediária

Uma classe de dados pode ser útil para criar registros que serão exportados para o JSON ou algum outro formato de intercomunicação, ou para manter dados que acabaram de ser importados, cruzando alguma fronteira do sistema. Todas as fábricas de classes de dados de Python oferecem um método ou função para converter uma instância em um dict simples, e você sempre pode invocar o construtor com um dict, usado para passar argumentos nomeados expandidos com **. Um dict assim é muito similar a um registro JSON.

Nesse cenário, as instâncias da classe de dados devem ser tratadas como objetos imutáveis—mesmo que os campos sejam mutáveis, não deveriam ser modificados nessa forma intermediária. Mudá-los significa perder o principal benefício de manter os dados e o comportamento próximos. Quando o processo de importação/exportação exigir mudança nos valores, você deve implementar seus próprios métodos de fábrica, em vez de usar os métodos "as dict" existentes ou os construtores padrão.

Vamos agora mudar de assunto e aprender como escrever padrões que "casam" com instâncias de classes arbitrárias, não apenas com as sequências e mapeamentos que vimos nas seções Seção 2.6 e Seção 3.3.

5.8. Pattern Matching com instâncias de classes

Padrões de classe são projetados para "casar" com instâncias de classes por tipo e—opcionalmente—por atributos. O sujeito de um padrão de classe pode ser uma instância de qualquer classe, não apenas instâncias de classes de dados.^[15]

Há três variantes de padrões de classes: simples, nomeado e posicional. Vamos estudá-las nessa ordem.

5.8.1. Padrões de classe simples

Já vimos um exemplo de padrões de classe simples usados como sub-padrões na Seção 2.6:

```
case [str(name), _, _, (float(lat), float(lon))]:
```

Aquele padrão casa com uma sequência de quatro itens, onde o primeiro item deve ser uma instância de str e o último item deve ser uma sequência de duas instâncias de float.

A sintaxe dos padrões de classe se parece com a invocação de um construtor. Abaixo temos um padrão de classe que "casa" com valores float sem vincular uma variável (o corpo do case pode ser referir a x diretamente, se necessário):

```
match x:  
    case float():  
        do_something_with(x)
```

Mas isso aqui possivelmente será um bug no seu código:

```
match x:  
    case float: # DANGER!!!  
        do_something_with(x)
```

No exemplo anterior, case float: "casa" com qualquer sujeito, pois Python entende float como uma variável, que é então vinculada ao sujeito.

A sintaxe float(x) do padrão simples é um caso especial que se aplica apenas a onze tipos embutidos "abençoados", listados no final da seção "Class Patterns" (*Padrões de Classe*) [<https://fpy.li/5-16>] (EN) da PEP 634—Structural Pattern Matching: Specification [<https://fpy.li/pep634>] (*Pattern Matching Estrutural: Especificação*).

```
bool  bytearray  bytes  dict  float  frozenset  int  list  set  str  tuple
```

Nessas classes, a variável que parece um argumento do construtor—por exemplo, o x em float(x)—é vinculada a toda a instância do sujeito ou à parte do sujeito que "casa" com um sub-padrão, como exemplificado por str(name) no padrão de sequência que vimos antes:

```
case [str(name), _, _, (float(lat), float(lon))]:
```

Se a classe não é um daqueles onze tipos embutidos abençoados, então essas variáveis parecidas com argumentos representam padrões a serem casados com atributos de uma instância daquela classe.

5.8.2. Padrões de classe nomeados

Para entender como usar padrões de classe nomeados, observe a classe City e suas cinco instâncias no Exemplo 85, abaixo.

Exemplo 85. A classe City e algumas instâncias

```
import typing  
  
class City(typing.NamedTuple):  
    continent: str  
    name: str
```

```

country: str

cities = [
    City('Asia', 'Tokyo', 'JP'),
    City('Asia', 'Delhi', 'IN'),
    City('North America', 'Mexico City', 'MX'),
    City('North America', 'New York', 'US'),
    City('South America', 'São Paulo', 'BR'),
]

```

Dadas essas definições, a seguinte função devolve uma lista de cidades asiáticas:

```

def match_asian_cities():
    results = []
    for city in cities:
        match city:
            case City(continent='Asia'):
                results.append(city)
    return results

```

O padrão `City(continent='Asia')` encontra qualquer instância de `City` onde o atributo `continent` seja igual a `'Asia'`, independente do valor dos outros atributos.

Para coletar o valor do atributo `country`, você poderia escrever:

```

def match_asian_countries():
    results = []
    for city in cities:
        match city:
            case City(continent='Asia', country=cc):
                results.append(cc)
    return results

```

O padrão `City(continent='Asia', country=cc)` encontra as mesmas cidades asiáticas, como antes, mas agora a variável `cc` está vinculada ao atributo `country` da instância. Isso inclusive funciona se a variável do padrão também se chamar `country`:

```

match city:
    case City(continent='Asia', country=country):
        results.append(country)

```

Padrões de classe nomeados são bastante legíveis, e funcionam com qualquer classe que possua atributos de instância públicos. Mas eles são um tanto prolixos.

Padrões de classe posicionais são mais convenientes em alguns casos, mas exigem suporte explícito da classe do sujeito, como veremos a seguir.

5.8.3. Padrões de classe posicionais

Dadas as definições do Exemplo 85, a seguinte função devolveria uma lista de cidades asiáticas, usando um padrão de classe posicional:

```
def match_asian_cities_pos():
    results = []
    for city in cities:
        match city:
            case City('Asia'):
                results.append(city)
    return results
```

O padrão `City('Asia')` encontra qualquer instância de `City` na qual o valor do primeiro atributo seja `Asia`, independente do valor dos outros atributos.

Se você quiser obter o valor do atributo `country`, poderia escrever:

```
def match_asian_countries_pos():
    results = []
    for city in cities:
        match city:
            case City('Asia', _, country):
                results.append(country)
    return results
```

O padrão `City('Asia', _, country)` encontra as mesmas cidades de antes, mas agora variável `country` está vinculada ao terceiro atributo da instância.

Eu falei do "primeiro" ou do "terceiro" atributos, mas o quê isso realmente significa?

`City` (ou qualquer classe) funciona com padrões posicionais graças a um atributo de classe especial chamado `__match_args__`, que as fábricas de classe vistas nesse capítulo criam automaticamente. Esse é o valor de `__match_args__` na classe `City`:

```
>>> City.__match_args__
('continent', 'name', 'country')
```

Como se vê, `__match_args__` declara os nomes dos atributos na ordem em que eles serão usados em padrões posicionais.

Na [positional_pattern_implement_sec] vamos escrever código para definir `__match_args__` em uma classe que criaremos sem a ajuda de uma fábrica de classes.



Você pode combinar argumentos nomeados e posicionais em um padrão. Alguns, mas não todos, os atributos de instância disponíveis para o `match` podem estar listados no `__match_args__`. Dessa forma, algumas vezes pode ser necessário usar

argumentos nomeados em um padrão, além dos argumentos posicionais.

Chegou a hora de resumir o capítulo.

5.9. Resumo do Capítulo

O tópico principal desse capítulo foram as fábricas de classes de dados `collections.namedtuple`, `typing.NamedTuple`, e `dataclasses.dataclass`. Vimos como cada uma delas gera classes de dados a partir de descrições, fornecidas como argumentos a uma função fábrica ou, no caso das duas últimas, a partir de uma declaração `class` com dicas de tipo. Especificamente, ambas as variantes de tupla produzem subclasses de `tuple`, acrescentando apenas a capacidade de acessar os campos por nome, e criando também um atributo de classe `_fields`, que lista os nomes dos campos na forma de uma tupla de strings.

A seguir colocamos lado a lado os principais recursos de cada uma das três fábricas de classes, incluindo como extrair dados da instância como um `dict`, como obter os nomes e valores default dos campos, e como criar uma nova instância a partir de uma instância existente.

Isso levou ao nosso primeiro contato com dicas de tipo, especialmente aquelas usadas para anotar atributos em uma declaração `class`, usando a notação introduzida no Python 3.6 com a PEP 526—*Syntax for Variable Annotations (Sintaxe para Anotações de Variáveis)* [<https://fpy.li/pep526>] (EN). O aspecto provavelmente mais surpreendente das dicas de tipo em geral é o fato delas não terem efeito durante a execução. Python continua sendo uma linguagem dinâmica. Ferramentas externas, como o Mypy, são necessárias para aproveitar a informação de tipagem na detecção de erros via análise estática do código-fonte. Após um resumo básico da sintaxe da PEP 526, estudamos os efeitos das anotações em uma classe simples e em classes criadas por `typing.NamedTuple` e por `@dataclass`.

A seguir falamos sobre os recursos mais usados dentre os oferecidos por `@dataclass`, e sobre a opção `default_factory` da função `dataclasses.field`. Também demos uma olhada nas dicas de pseudo-tipo especiais `typing.ClassVar` e `dataclasses.InitVar`, importantes no contexto das classes de dados. Esse tópico central foi concluído com um exemplo baseado no schema Dublin Core, ilustrando como usar `dataclasses.fields` para iterar sobre os atributos de uma instância de `Resource` em um `__repr__` customizado.

Então alertamos contra os possíveis usos abusivos das classes de dados, frustrando um princípio básico da programação orientada a objetos: os dados e as funções que acessam os dados devem estar juntos na mesma classe. Classes sem uma lógica podem ser um sinal de uma lógica fora de lugar.

Na última seção, vimos como o *pattern matching* funciona com instâncias de qualquer classe como sujeitos—e não apenas das classes criadas com as fábricas apresentadas nesse capítulo.

5.10. Leitura complementar

A documentação padrão de Python para as fábricas de classes de dados vistas aqui é muito boa, e inclui muitos pequenos exemplos.

Em especial para `@dataclass`, a maior parte da PEP 557—Data Classes (*Classes de Dados*) [<https://fpy.li/pep557>] (EN) foi copiada para a documentação do módulo `dataclasses` [<https://fpy.li/3s>]. Entretanto, algumas seções informativas da PEP 557 [<https://fpy.li/pep557>] não foram copiadas, incluindo "Why not just use `namedtuple`?" (*Por que simplesmente não usar `namedtuple`?*) [<https://fpy.li/5-18>], "Why not just use `typing.NamedTuple`?" (*Por que simplesmente não usar `typing.NamedTuple`?*) [<https://fpy.li/5-19>], e a seção "Rationale" (*Justificativa*) [<https://fpy.li/5-20>], que termina com a seguinte Q&A:

Quando não é apropriado usar Classes de Dados?

Quando for exigida compatibilidade da API com tuplas de dicts. Quando for exigida validação de tipo além daquela oferecida pelas PEPs 484 e 526, ou quando for exigida validação ou conversão de valores.

— Eric V. Smith, PEP 557 "Justificativa"

Em *RealPython.com* [<https://fpy.li/5-21>], Geir Arne Hjelle escreveu *Ultimate guide to data classes in Python 3.7* [<https://fpy.li/5-22>] (EN) guia bem completo.

Na PyCon US 2018, Raymond Hettinger apresentou "Dataclasses: The code generator to end all code generators" [<https://fpy.li/5-23>] (EN, video).

Para mais recursos e funcionalidade avançada, incluindo validação, o projeto `attrs` [<https://fpy.li/5-24>] (EN), liderado por Hynek Schlawack, surgiu anos antes de `dataclasses` e oferece mais facilidades, com a promessa de "trazer de volta a alegria de criar classes, liberando você do tedioso trabalho de implementar protocolos de objeto (também conhecidos como métodos *dunder*)".

A influência do `attrs` sobre o `@dataclass` é reconhecida por Eric V. Smith na PEP 557. Isso provavelmente inclui a mais importante decisão de Smith sobre a API: o uso de um decorador de classe em vez de uma classe base ou de uma metaclass para realizar a tarefa.

Glyph—fundador do projeto Twisted—escreveu uma excelente introdução à `attrs` em "The One Python Library Everyone Needs" (*A Biblioteca Python que Todo Mundo Precisa Ter*) [<https://fpy.li/5-25>] (EN). A documentação da `attrs` inclui uma discussão sobre alternativas [<https://fpy.li/5-26>].

O autor de livros, instrutor e cientista maluco da computação Dave Beazley escreveu o *cluegen* [<https://fpy.li/5-27>], um outro gerador de classes de dados. Se você já assistiu alguma palestra do David, sabe que ele é um mestre na metaprogramação Python a partir de princípios básicos. Então achei inspirador descobrir, no arquivo *README.md* do *cluegen*, o caso de uso concreto que o motivou a criar uma alternativa ao `@dataclass` de Python, e sua filosofia de apresentar uma abordagem para resolver o problema, ao invés de fornecer uma ferramenta: a ferramenta pode inicialmente ser mais rápida de usar, mas a abordagem é mais flexível e pode ir tão longe quanto você quiser.

Sobre a *classe de dados* como um cheiro no código, a melhor fonte que encontrei foi livro de Martin Fowler, *Refactoring* ("Refatorando"), 2^a ed. A versão mais recente não traz a citação da epígrafe deste capítulo, "Classes de dados são como crianças...", mas apesar disso é a melhor edição do livro mais famoso de Fowler, em especial para pythonistas, pois os exemplos são em JavaScript moderno, que é mais próximo de Python que de Java—a linguagem usada na primeira edição.

O site *Refactoring Guru* (Guru da Refatoração) [<https://fpy.li/5-28>] (EN) também tem uma descrição do data class code smell (*classe de dados como cheiro no código*) [<https://fpy.li/5-29>] (EN).

Ponto de vista

O verbete para "Guido" [<https://fpy.li/5-30>] no "The Jargon File" [<https://fpy.li/5-31>] (EN) é sobre Guido van Rossum. Entre outras coisas, ele diz:

Diz a lenda que o atributo mais importante de Guido, além do próprio Python, é a máquina do tempo de Guido, um aparelho que dizem que ele possui por causa da frequência irritante com que pedidos de usuários por novos recursos recebem como resposta "Acabei de implementar isso noite passada..."

Por um longo tempo, uma das peças ausentes da sintaxe de Python foi uma forma rápida e padronizada de declarar atributos de instância em uma classe. Muitas linguagens orientadas a objetos incluem esse recurso. Aqui está parte da definição da classe Point em Smalltalk:

```
Object subclass: #Point
  instanceVariableNames: 'x y'
  classVariableNames: ''
  package: 'Kernel-BasicObjects'
```

A segunda linha lista os nomes dos atributos de instância x e y. Se existissem atributos de classe, eles estariam na terceira linha.

Python sempre teve uma forma fácil de declarar um atributo de classe, se ele tiver um valor inicial. Mas atributos de instância são mais comuns, e os programadores Python têm sido obrigados a olhar dentro do método `__init__` para encontrá-los, sempre temerosos que podem existir atributos de instância sendo criados em outro lugar na classe—ou mesmo por funções e métodos de outras classes.

Agora temos o `@dataclass`, viva!

Mas ele traz seus próprios problemas

Primeiro, quando você usa `@dataclass`, as dicas de tipo não são opcionais. Pelos últimos sete anos, desde a PEP 484—Type Hints (*Dicas de Tipo*) [<https://fpy.li/pep484>] (EN), prometeram que elas sempre seriam opcionais. Agora temos um novo recurso importante na linguagem que exige dicas de tipo. Se você não gosta de toda essa tendência de tipagem estática, pode querer usar a `attrs` [<https://fpy.li/5-24>] em vez de `@dataclass`.

Em segundo lugar, a sintaxe da PEP 526 [<https://fpy.li/pep526>] (EN) para anotar atributos de instância e de classe inverte a convenção consagrada para declarações de classe: tudo que era declarado no nível superior de um bloco `class` era um atributo de classe (métodos também são atributos de classe). Com a PEP 526 e o `@dataclass`, qualquer atributo declarado no nível superior com uma dica de tipo se torna um atributo de instância:

```
@dataclass  
class Spam:  
    repeat: int # instance attribute
```

Aqui, `repeat` também é um atributo de instância:

```
@dataclass  
class Spam:  
    repeat: int = 99 # instance attribute
```

Mas se não houver dicas de tipo, subitamente estamos de volta ao cenário em que declarações no nível superior da classe pertencem apenas à classe:

```
@dataclass  
class Spam:  
    repeat = 99 # class attribute!
```

Por fim, se você quiser anotar aquele atributo de classe com um tipo, não pode usar tipos comuns, porque então ele se tornará um atributo de instância. Você tem que recorrer a aquela anotação usando o pseudo-tipo `ClassVar`, que é uma gambiarra:

```
@dataclass  
class Spam:  
    repeat: ClassVar[int] = 99 # aargh!
```

Aqui estamos falando sobre uma exceção da exceção da regra. Me parece algo muito pouco pythônico.

Não tomei parte nas discussões que levaram à PEP 526 ou à PEP 557—Data Classes (*Classes de Dados*) [<https://fpy.li/pep557>], mas aqui está uma sintaxe alternativa que eu gostaria de ver:

```
@dataclass  
class HackerClubMember:  
    .name: str  
    .guests: list = field(default_factory=list)  
    .handle: str = ''  
  
    all_handles = set()
```

① Atributos de instância devem ser declarados com um prefixo `.` (ponto).

② Qualquer nome de atributo que não tenha um prefixo `.` é um atributo de classe (como sempre foram).

A gramática da linguagem teria que mudar para acomodar isso. Mas acho essa forma mais

legível, e ela evita o problema da exceção-da-exceção.

Queria poder pegar a máquina do tempo de Gudo emprestada e voltar a 2017, para convencer os desenvolvedores principais a aceitarem essa ideia.

[1] Fonte: *Refactoring, First Edition*, capítulo 3, seção *Bad Smells in Code, Data Class* (Mau cheiro no código, classe de dados), p. 87.

[2] As metaclasses são um dos assuntos tratados no [ch_class_metaprogramming].

[3] Decoradores de classe são discutidos no [ch_class_metaprogramming], na seção "Metaprogramação de classes", junto com as metaclasses. Ambos são formas de customizar o comportamento de uma classe além do que seria possível com herança.

[4] Se você conhece Ruby, sabe que injetar métodos é uma técnica bastante conhecida, apesar de controversa, entre *rubystas*. Em Python isso não é tão comum, pois não funciona com nenhum dos tipos embutidos—str, list, etc. Considero essa limitação de Python uma bênção.

[5] No contexto das dicas de tipo, None não é o singleton `NoneType`, mas um apelido para o próprio `NoneType`. Se paramos para pensar, isso é estranho, mas agrada nossa intuição e torna as anotações de valores devolvidos por uma função mais fáceis de ler, no caso comum de funções que devolvem `None`.

[6] O conceito de *undefined*, um dos erros mais tolos no design de Javascript, não existe no Python. Obrigado, Guido!

[7] NT: Um *getter* é um método que devolve o valor um atributo do objeto. Para propriedades mutáveis, o *getter* vem geralmente acompanhado por um *setter*, que modifica a mesma propriedade. Os nomes derivam dos verbos em inglês *get* (obter, receber) e *set* (definir, estabelecer).

[8] Definir um atributo após o `__init__` prejudica a otimização de uso de memória com o compartilhamento das chaves do `__dict__`, mencionada na Seção 3.9.

[9] O `@dataclass` emula a imutabilidade criando um `__setattr__` e um `__delattr__` que geram um `dataclass.FrozenInstanceError`—uma subclasse de `AttributeError`—quando o usuário tenta definir ou apagar o valor de um campo.

[10] `dataclass._MISSING_TYPE` é um valor sentinela, indicando que a opção não foi fornecida. Ele existe para que se possa definir `None` como um valor default efetivo, um caso de uso comum.

[11] A opção `hash=None` significa que o campo será usado em `__hash__` apenas se `compare=True`.

[12] Fonte: O artigo Dublin Core [<https://fpy.li/3w>] na Wikipedia.

[13] NT: *Code smell* em geral não é traduzido na literatura em português—uma tradução quase literal seria "fedor no código". Uma tradução mais gentil pode ser "cheiro no código", adotado aqui. Mais gentil e menos enviesada: um "cheiro no código" nem sempre é indicação de um problema.

[14] Tenho a felicidade de ter Martin Fowler como colega de trabalho na Thoughtworks, estou precisando de apenas 20 minutos para obter sua permissão.

[15] Trato desse conteúdo aqui por ser o primeiro capítulo sobre classes definidas pelo usuário, e acho que *pattern matching* com classes é um assunto muito importante para esperar até a Parte II: Funções como objetos do livro. Minha filosofia: é mais importante saber como usar classes que como defini-las.

6. Referências, mutabilidade, e memória

“Você está triste,” disse o Cavaleiro em um tom de voz ansioso: “deixe eu cantar para você uma canção reconfortante. [...] O nome da canção se chama ‘OLHOS DE HADOQUE’.”

“Oh, esse é o nome da canção?”, disse Alice, tentando parecer interessada.

“Não, você não entendeu,” retorquiu o Cavaleiro, um pouco irritado. “É assim que o nome É CHAMADO. O nome na verdade é ‘O ENVELHECIDO HOMEM VELHO.’”

— Adaptado de “Alice Através do Espelho e o que Ela Encontrou Lá”, de Lewis Carroll

Alice e o Cavaleiro dão o tom do que veremos nesse capítulo. O tema é a distinção entre objetos e seus nomes; um nome não é o objeto; o nome é outra coisa.

Começamos o capítulo apresentando uma metáfora para variáveis em Python: variáveis são rótulos, não caixas. Se variáveis de referência não são novidade para você, a analogia pode ainda ser útil para ilustrar questões de *aliasing* (“apelidamento”) para alguém.

Depois discutimos os conceitos de identidade, valor e apelidamento de objetos. Uma característica surpreendente das tuplas é revelada: elas são imutáveis, mas seus valores podem mudar. Isso leva a uma discussão sobre cópias rasas e profundas. Referências e parâmetros de funções são o tema seguinte: o problema com parâmetros mutáveis por default e formas seguras de lidar com argumentos mutáveis passados para nossas funções por clientes.

As últimas seções do capítulo tratam de coleta de lixo (*garbage collection*), a instrução `del` e de algumas otimizações com com objetos imutáveis em Python.

É um capítulo bastante árido, mas os tópicos tratados podem explicar muitos bugs sutis em programas reais em Python, além de boas práticas para evitá-los.

6.1. Novidades nesse capítulo

Os tópicos tratados aqui são muito estáveis e fundamentais. Não foi introduzida nenhuma mudança digna de nota nesta segunda edição.

Acrescentei um exemplo usando `is` para testar a existência de um objeto sentinela, e um aviso sobre o mau uso do operador `is` no final de Seção 6.3.1.

Este capítulo estava na Parte IV, mas decidi abordar esses temas mais cedo, pois eles funcionam melhor como o encerramento da Parte II, “Estruturas de Dados”, que como abertura de “Práticas de Orientação a Objetos”



A seção sobre “Referências Fracas” da primeira edição deste livro agora é um post em [fluentpython.com \[https://fpy.li/weakref\]](https://fpy.li/weakref).

Vamos começar desaprendendo que uma variável é como uma caixa onde você guarda dados.

6.2. Variáveis não são caixas

Em 1997, fiz um curso de verão sobre Java no MIT. A professora, Lynn Stein^[1] apontou que a metáfora comum, de “variáveis como caixas”, na verdade atrapalha o entendimento de variáveis de referência em linguagens orientadas a objetos. As variáveis em Python são como variáveis de referência em Java; uma metáfora melhor é pensar em uma variável como uma etiqueta que dá nome a um objeto. O exemplo e a figura a seguir ajudam a entender o motivo disso.

O Exemplo 86 é uma interação simples que não pode ser explicada por “variáveis como caixas”.

Exemplo 86. As variáveis a e b mantêm referências para a mesma lista, não cópias da lista.

```
>>> a = [1, 2, 3] ①
>>> b = a          ②
>>> a.append(4)   ③
>>> b             ④
[1, 2, 3, 4]
```

- ① Cria uma lista [1, 2, 3] e a vincula à variável a.
- ② Vincula a variável b ao mesmo valor referenciado por a.
- ③ Modifica a lista referenciada por a, anexando um novo item.
- ④ É possível ver o efeito através da variável b. Se você pensar em b como uma caixa que guardava uma cópia de [1, 2, 3] da caixa a, este comportamento não faz sentido.

A Figura 18 explica porque a metáfora da caixa está errada em Python, enquanto etiquetas apresentam uma imagem mais útil para entender como variáveis funcionam.

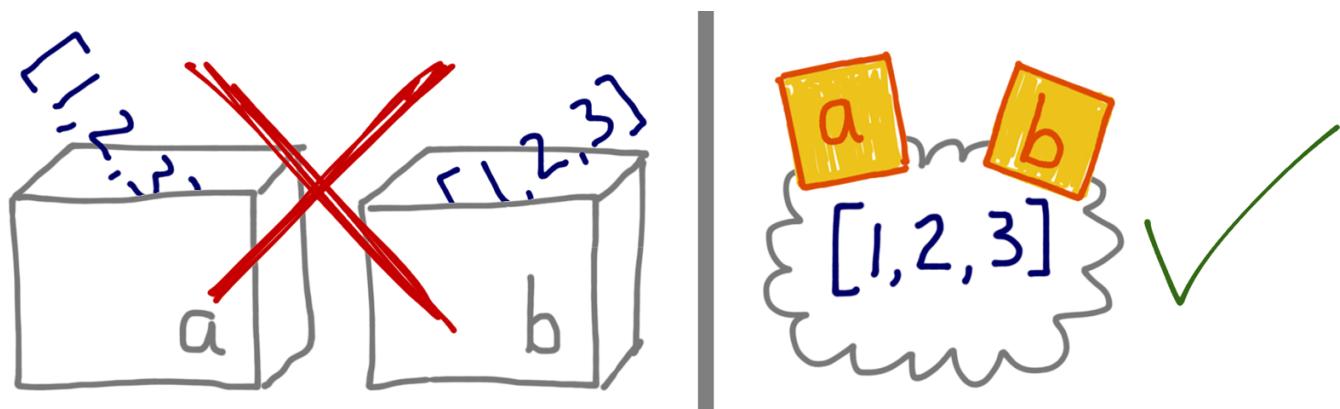


Figura 18. Se você imaginar variáveis como caixas, não é possível entender a atribuição em Python; por outro lado, imagine variáveis como etiquetas autocolantes e o Exemplo 86 é facilmente explicável.

Assim, a instrução `b = a` não copia o conteúdo de uma caixa a para uma caixa b. Ela cola uma nova etiqueta b no objeto que já tem a etiqueta a.

A professora Stein também falava sobre atribuição de uma maneira bastante específica. Por exemplo, quando discutia sobre um objeto representando uma gangorra em uma simulação, ela dizia: “A variável g foi atribuída à gangorra”, mas nunca “A gangorra foi atribuída à variável g”.

Com variáveis de referência, faz mais sentido dizer que a variável é atribuída a um objeto, não o contrário. Afinal, o objeto é criado antes da atribuição. Exemplo 87 prova que o lado direito de uma atribuição é processado primeiro.

Já que o verbo “atribuir” é usado de diferentes maneiras, uma “vincular” é uma alternativa melhor: a declaração de atribuição em Python `x = ...` vincula o nome `x` ao objeto criado ou referenciado no lado direito. E o objeto precisa existir antes que um nome possa ser vinculado a ele, como demonstra Exemplo 87.

Exemplo 87. Variáveis são vinculadas a objetos somente após os objetos serem criados

```
>>> class Gizmo:  
...     def __init__(self):  
...         print(f'Gizmo id: {id(self)}')  
...  
>>> x = Gizmo()  
Gizmo id: 4301489152 ①  
>>> y = Gizmo() * 10 ②  
Gizmo id: 4301489432 ③  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unsupported operand type(s) for *: 'Gizmo' and 'int'  
>>>  
>>> dir() ④  
['Gizmo', '__builtins__', '__doc__', '__loader__', '__name__',  
 '__package__', '__spec__', 'x']
```

① A saída `Gizmo id: ...` é um efeito colateral da criação de uma instância de `Gizmo`.

② Multiplicar uma instância de `Gizmo` levanta uma exceção.

③ Aqui está a prova que um segundo `Gizmo` foi de fato instanciado antes que a multiplicação fosse tentada.

④ Mas a variável `y` nunca foi criada, porque a exceção aconteceu enquanto a parte direita da atribuição estava sendo executada.



Para entender uma atribuição em Python, leia primeiro o lado direito: é ali que o objeto é criado ou recuperado. Depois disso, a variável do lado esquerdo é vinculada ao objeto, como uma etiqueta colada a ele. Esqueça as caixas.

Como variáveis são apenas meras etiquetas, nada impede que um objeto tenha várias etiquetas vinculadas a si. Quando isso acontece, você tem *apelidos* (aliases), nosso próximo tópico.

6.3. Identidade, igualdade e apelidos

Lewis Carroll é o pseudônimo literário do Prof. Charles Lutwidge Dodgson. O Sr. Carroll não é apenas igual ao Prof. Dodgson, eles são exatamente a mesma pessoa. Exemplo 88 expressa essa ideia em Python.

Exemplo 88. charles e lewis se referem ao mesmo objeto

```
>>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}
>>> lewis = charles ①
>>> lewis is charles
True
>>> id(charles), id(lewis) ②
(4300473992, 4300473992)
>>> lewis['balance'] = 950 ③
>>> charles
{'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950}
```

① lewis é um apelido para charles.

② O operador `is` e a função `id` confirmam essa afirmação.

③ Adicionar um item a lewis é o mesmo que adicionar um item a charles.

Entretanto, suponha que um impostor—vamos chamá-lo de Dr. Alexander Pedachenko—diga que é o verdadeiro Charles L. Dodgson, nascido em 1832. Suas credenciais podem ser as mesmas, mas o Dr. Pedachenko não é o Prof. Dodgson. Figura 19 ilustra esse cenário.

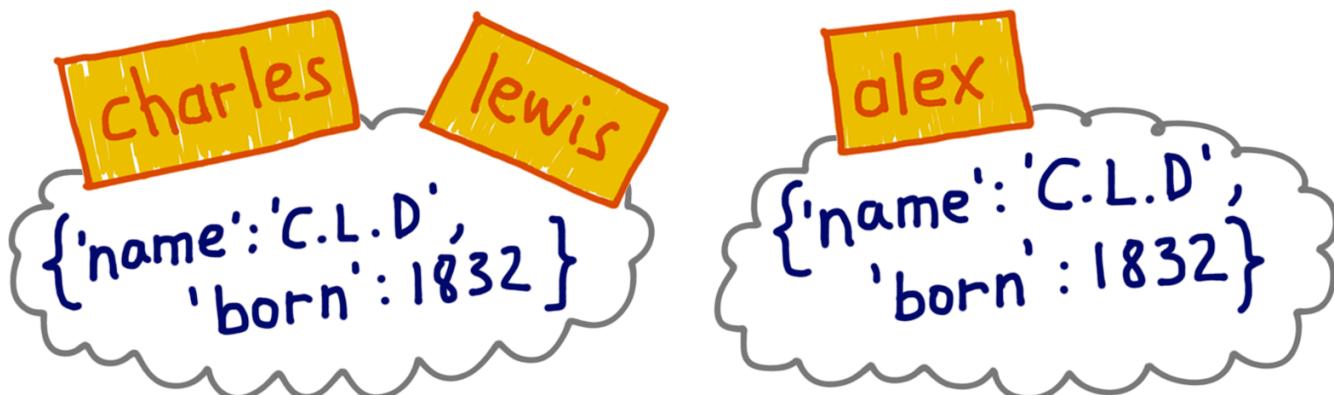


Figura 19. charles e lewis estão vinculados ao mesmo objeto; alex está vinculado a um objeto diferente de valor igual.

Exemplo 89 implementa e testa o objeto alex como apresentado em Figura 19.

Exemplo 89. alex e charles são iguais quando comparados, mas alex não é charles

```
>>> alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950} ①
>>> alex == charles ②
True
>>> alex is not charles ③
True
```

① alex é uma referência a um objeto que é uma réplica do objeto vinculado a charles.

② Os objetos são iguais quando comparados devido à implementação de `__eq__` na classe `dict`.

③ Mas são objetos distintos. Essa é a forma pythônica de escrever a negação de uma comparação de identidade: `a is not b`.

Exemplo 88 é um exemplo de *apelidamento* (aliasing). Naquele código, `lewis` e `charles` são apelidos: duas variáveis vinculadas ao mesmo objeto. Por outro lado, `alex` não é um apelido para `charles`: essas variáveis estão vinculadas a objetos diferentes. Os objetos vinculados a `alex` e `charles` tem o mesmo *valor* — é isso que `==` compara — mas tem identidades diferentes.

Na Referência da Linguagem Python [<https://fpy.li/2x>], está escrito:

A identidade de um objeto nunca muda após ele ter sido criado; você pode pensar nela como o endereço do objeto na memória. O operador `is` compara a identidade de dois objetos; a função `id()` retorna um inteiro representando essa identidade.

O verdadeiro significado do ID de um objeto depende da implementação da linguagem. Em CPython, `id()` retorna o endereço de memória do objeto, mas outro interpretador Python pode retornar algo diferente. O ponto fundamental é que o ID será sempre um valor numérico único, e ele nunca mudará durante a vida do objeto.

Na prática, raramente usamos a função `id()` quando programamos. A verificação de identidade é feita, na maior parte das vezes, com o operador `is`, que compara os IDs dos objetos, então nosso código não precisa chamar `id()` explicitamente. A seguir vamos falar sobre `is` versus `==`.



Para o revisor técnico Leonardo Rochael, o uso mais frequente de `id()` ocorre durante o processo de debugging, quando o `repr()` de dois objetos são semelhantes, mas você precisa saber se duas referências são apelidos ou apontam para objetos diferentes. Se as referências estão em contextos diferentes—por exemplo, em stack frames diferentes—pode não ser viável usar `is`.

6.3.1. Escolhendo Entre `==` e `is`

O operador `==` compara os valores de objetos (os dados que eles contêm), enquanto `is` compara suas identidades.

Quando estamos programando, em geral, nos preocupamos mais com os valores que com as identidades dos objetos, então `==` aparece com mais frequência que `is` em programas Python.

Entretanto, se você estiver comparando uma variável com um singleton (um objeto único) faz mais sentido usar `is`. O caso mais comum, de longe, é verificar se a variável está vinculada a `None`. Esta é a forma recomendada de fazer isso:

```
x is None
```

E a forma apropriada de escrever sua negação é:

```
x is not None
```

`None` é o singleton mais comum que testamos com `is`. Objetos sentinelas são outro exemplo de

singletons que testamos com `is`. Veja um modo de criar e testar um objeto sentinelas:

```
END_OF_DATA = object()
# ... many lines
def traverse(...):
    # ... more lines
    if node is END_OF_DATA:
        return
    # etc.
```

O operador `is` é mais rápido que `==`, pois não pode ser sobreescrito. Daí Python não precisa encontrar e invocar métodos especiais para calcular seu resultado e o processamento é tão simples quanto comparar dois IDs, que são números inteiros. Por outro lado, `a == b` é açúcar sintático para `a.__eq__(b)`. O método `__eq__`, herdado de `object`, compara os IDs dos objetos, então produz o mesmo resultado de `is`. Mas a maioria dos tipos embutidos sobreescrava `__eq__` com implementações mais úteis, que levam em consideração os valores dos atributos dos objetos. A determinação da igualdade pode envolver muito processamento—por exemplo, quando se comparam coleções grandes ou estruturas aninhadas com muitos níveis.



Normalmente estamos mais interessados na igualdade que na identidade de objetos. Checar se o objeto é `None` é o único caso de uso comum do operador `is`. A maioria dos outros usos que vejo quando reviso código estão errados. Se você não estiver seguro, use `==`. Em geral, é o que você quer, e ele também funciona com `None`, ainda que não tão rápido.

Para concluir essa discussão de identidade versus igualdade, vamos ver como o tipo notoriamente imutável `tuple` não é assim tão invariável quanto você poderia supor.

6.3.2. A imutabilidade relativa das tuplas

As tuplas, como a maioria das coleções em Python—lists, dicts, sets, etc.—são contêiners: armazenam referências para objetos.^[2]

Se os itens referenciados forem mutáveis, eles podem mudar, mesmo que tupla em si não mude. Em outras palavras, a imutabilidade das tuplas, na verdade se refere ao conteúdo físico da estrutura de dados tuple (isto é, as referências que ela mantém), e não se estende aos objetos referenciados.

O Exemplo 90 ilustra uma situação em que o valor de uma tupla muda como resultado de mudanças em um objeto mutável ali referenciado. O que não pode nunca mudar em uma tupla é a identidade dos itens que ela contém.

Exemplo 90. t1 e t2 inicialmente são iguais, mas a mudança em um item mutável dentro da tupla t1 as torna diferentes

```
>>> t1 = (1, 2, [30, 40]) ①
>>> t2 = (1, 2, [30, 40]) ②
>>> t1 == t2 ③
```

```

True
>>> id(t1[-1]) ④
4302515784
>>> t1[-1].append(99) ⑤
>>> t1
(1, 2, [30, 40, 99])
>>> id(t1[-1]) ⑥
4302515784
>>> t1 == t2 ⑦
False

```

- ① `t1` é imutável, mas `t1[-1]` é mutável.
- ② Cria a tupla `t2`, cujos itens são iguais àqueles de `t1`.
- ③ Apesar de serem objetos distintos, quando comparados `t1` e `t2` são iguais, como esperado.
- ④ Obtém o ID da lista na posição `t1[-1]`.
- ⑤ Modifica diretamente a lista `t1[-1]`.
- ⑥ O ID de `t1[-1]` não mudou, apenas seu valor.
- ⑦ `t1` e `t2` agora são diferentes

Essa imutabilidade relativa das tuplas está por trás do enigma Seção 2.8.3. Essa também é razão pela qual não é possível gerar o hash de algumas tuplas, como vimos em Seção 3.4.1.

A distinção entre igualdade e identidade tem outras implicações quando você precisa copiar um objeto. Uma cópia é um objeto igual com um ID diferente. Mas se um objeto contém outros objetos, é preciso que a cópia duplique os objetos internos ou eles podem ser compartilhados? Não há uma resposta única. A seguir discutimos esse ponto.

6.4. A princípio, cópias são rasas

A forma mais fácil de copiar uma lista (ou a maioria das coleções mutáveis nativas) é usando o construtor padrão do próprio tipo. Por exemplo:

```

>>> l1 = [3, [55, 44], (7, 8, 9)]
>>> l2 = list(l1) ①
>>> l2
[3, [55, 44], (7, 8, 9)]
>>> l2 == l1 ②
True
>>> l2 is l1 ③
False

```

- ① `list(l1)` cria uma cópia de `l1`.
- ② As cópias são iguais...
- ③ ...mas se referem a dois objetos diferentes.

Para listas e outras sequências mutáveis, o atalho `l2 = l1[:]` também cria uma cópia.

Contudo, tanto o construtor quanto `[:]` produzem uma *cópia rasa* (shallow copy). Isto é, o contêiner externo é duplicado, mas a cópia é preenchida com referências para os mesmos itens contidos no contêiner original. Isso economiza memória e não causa qualquer problema se todos os itens forem imutáveis. Mas se existirem itens mutáveis, isso pode gerar surpresas desagradáveis.

Em Exemplo 91 criamos uma lista contendo outra lista e uma tupla, e então fazemos algumas mudanças para ver como isso afeta os objetos referenciados.



Se você tem um computador conectado à internet disponível, recomendo fortemente que você assista à animação interativa do Exemplo 91 em Online Python Tutor [<https://fpy.li/6-3>]. No momento em que escrevo, o link direto para um exemplo pronto no *pythontutor.com* não estava funcionando de forma estável. Mas a ferramenta é ótima, então vale a pena gastar seu tempo copiando e colando o código.

Exemplo 91. Criando uma cópia rasa de uma lista contendo outra lista; copie e cole esse código para vê-lo animado no Online Python Tutor

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
l2 = list(l1)      ①
l1.append(100)    ②
l1[1].remove(55)  ③
print('l1:', l1)
print('l2:', l2)
l2[1] += [33, 22] ④
l2[2] += (10, 11) ⑤
print('l1:', l1)
print('l2:', l2)
```

① `l2` é uma cópia rasa de `l1`. Este estado está representado em Figura 20.

② Concatenar `100` a `l1` não tem qualquer efeito sobre `l2`.

③ Aqui removemos `55` da lista interna `l1[1]`. Isso afeta `l2`, pois `l2[1]` está associado à mesma lista em `l1[1]`.

④ Para um objeto mutável como a lista referida por `l2[1]`, o operador `+=` altera a lista diretamente. Essa mudança é visível em `l1[1]`, que é um apelido para `l2[1]`.

⑤ `+=` em uma tupla cria uma nova tupla e reassocia a variável `l2[2]` a ela. Isso é equivalente a fazer `l2[2] = l2[2] + (10, 11)`. Agora as tuplas na última posição de `l1` e `l2` não são mais o mesmo objeto. Veja Figura 21.

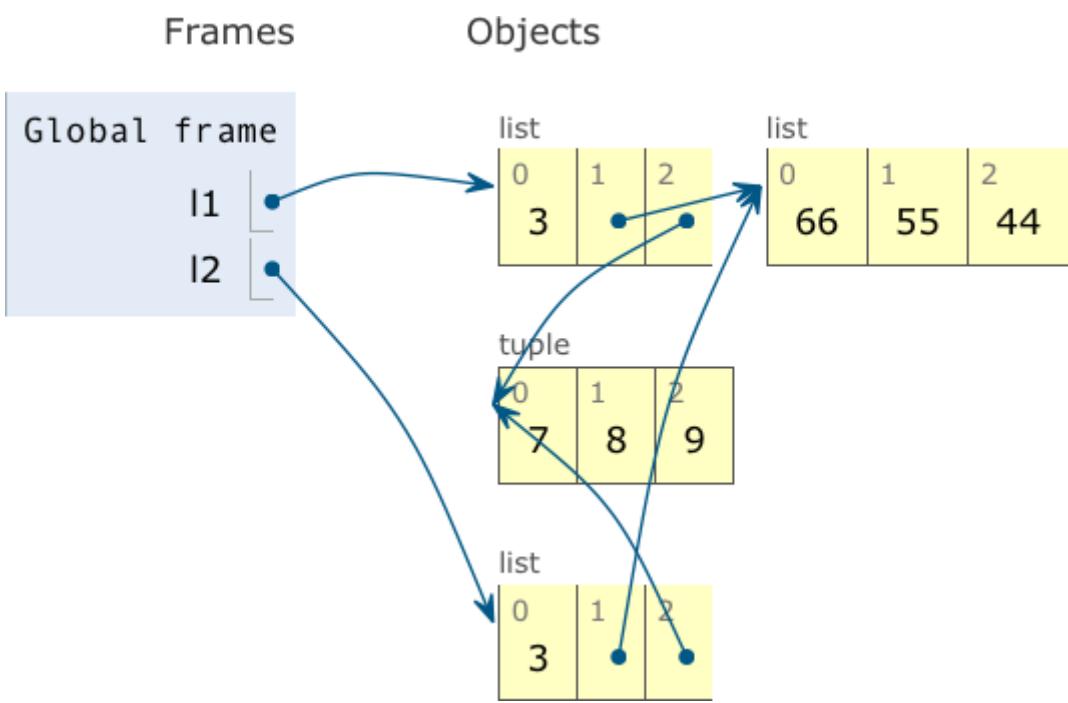


Figura 20. Estado do programa imediatamente após a atribuição `l2 = list(l1)` em Exemplo 91. `l1` e `l2` se referem a listas diferentes, mas as listas compartilham referências para um mesmo objeto interno, a lista [66, 55, 44] e para a tupla (7, 8, 9). (Diagrama gerado pelo Online Python Tutor)

A saída de Exemplo 91 é Exemplo 92, e o estado final dos objetos está representado em Figura 21.

Exemplo 92. Saída de Exemplo 91

```

l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```

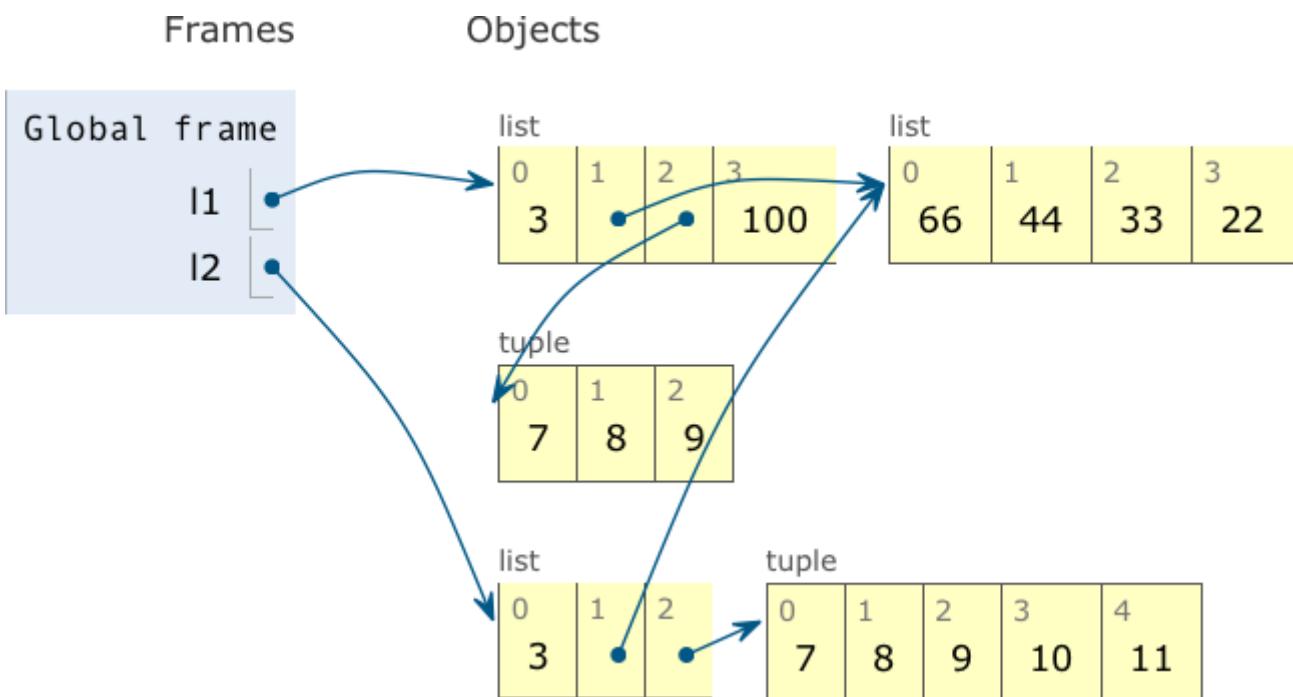


Figura 21. Estado final de l1 e l2: elas ainda compartilham referências para o mesmo objeto lista, que agora contém [66, 44, 33, 22], mas a operação l2[2] += (10, 11) criou uma nova tupla com conteúdo (7, 8, 9, 10, 11), sem relação com a tupla (7, 8, 9) referenciada por l1[2]. (Diagram generated by the Online Python Tutor.)

Já deve estar claro que cópias rasas são fáceis de criar, mas podem ou não ser o que você quer. Nossa próximo tópico é a criação de cópias profundas.

6.4.1. Cópias profundas e cópias rasas

Trabalhar com cópias rasas nem sempre é um problema, mas algumas vezes você vai precisar criar cópias profundas (isto é, cópias que não compartilham referências de objetos internos). O módulo copy oferece as funções `deepcopy` e `copy`, que retornam cópias profundas e rasas de objetos arbitrários.

Para ilustrar o uso de `copy()` e `deepcopy()`, Exemplo 93 define uma classe simples, `Bus`, representando um ônibus escolar que é carregado com passageiros, e então pega ou deixa passageiros ao longo de sua rota.

Exemplo 93. Bus pega ou deixa passageiros

```
class Bus:

    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers)

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
```

```
self.passengers.remove(name)
```

Agora, no Exemplo 94 interativo, vamos criar um objeto bus (bus1) e dois clones—uma cópia rasa (bus2) e uma cópia profunda (bus3)—para ver o que acontece quando bus1 deixa um passageiro.

Exemplo 94. Os efeitos do uso de copy versus deepcopy

```
>>> import copy
>>> bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])
>>> bus2 = copy.copy(bus1)
>>> bus3 = copy.deepcopy(bus1)
>>> id(bus1), id(bus2), id(bus3)
(4301498296, 4301499416, 4301499752) ①
>>> bus1.drop('Bill')
>>> bus2.passengers
['Alice', 'Claire', 'David'] ②
>>> id(bus1.passengers), id(bus2.passengers), id(bus3.passengers)
(4302658568, 4302658568, 4302657800) ③
>>> bus3.passengers
['Alice', 'Bill', 'Claire', 'David'] ④
```

① Usando `copy` e `deepcopy`, criamos três instâncias distintas de `Bus`.

② Após `bus1` deixar '`Bill`', ele também desaparece de `bus2`.

③ A inspeção do atributo dos `passengers` mostra que `bus1` e `bus2` compartilham o mesmo objeto lista, pois `bus2` é uma cópia rasa de `bus1`.

④ `bus3` é uma cópia profunda de `bus1`, então seu atributo `passengers` se refere a outra lista.

Em geral, criar cópias profundas não é uma questão simples. Objetos podem conter referências cíclicas que fariam um algoritmo ingênuo entrar em um loop infinito. A função `deepcopy` memoriza os objetos já copiados, e trata referências cíclicas corretamente. Isso é demonstrado em Exemplo 95.

Exemplo 95. Referências cíclicas: b tem uma referência para a e então é concatenado a a; ainda assim, deepcopy consegue copiar a.

```
>>> a = [10, 20]
>>> b = [a, 30]
>>> a.append(b)
>>> a
[10, 20, [[...], 30]]
>>> from copy import deepcopy
>>> c = deepcopy(a)
>>> c
[10, 20, [[...], 30]]
```

Além disso, algumas vezes uma cópia profunda pode ser profunda demais. Por exemplo, objetos podem ter referências para recursos externos ou para *singletons* (objetos únicos) que não devem ser copiados. Você pode controlar o comportamento de `copy` e de `deepcopy` implementando os métodos especiais `__copy__` e `__deepcopy__`, como descrito em documentação do módulo `copy`.

O compartilhamento de objetos através de apelidos também explica como a passagens de parâmetros funciona em Python, e o problema do uso de tipos mutáveis como parâmetros default. Vamos falar sobre essas questões a seguir.

6.5. Parâmetros de função como referências

O único modo de passagem de parâmetros em Python é a *chamada por compartilhamento (call by sharing)*. É o mesmo modo usado na maioria das linguagens orientadas a objetos, incluindo Javascript, Ruby e Java (em Java isso se aplica aos tipos de referência; tipos primitivos usam a chamada por valor). Chamada por compartilhamento significa que cada parâmetro formal da função recebe uma cópia de cada referência nos argumentos. Em outras palavras, os parâmetros dentro da função se tornam apelidos dos argumentos passados.

O resultado desse esquema é que a função pode modificar qualquer objeto mutável passado a ela como parâmetro, mas não pode mudar a identidade daqueles objetos (isto é, ela não pode substituir integralmente um objeto por outro). O Exemplo 96 mostra uma função simples usando `+=` com um de seus parâmetros. Quando passamos números, listas e tuplas para a função, os argumentos originais são afetados de maneiras diferentes. Veja só:

Exemplo 96. Uma função pode mudar qualquer objeto mutável que receba

```
>>> def f(a, b):
...     a += b
...     return a
...
>>> x = 1
>>> y = 2
>>> f(x, y)
3
>>> x, y ①
(1, 2)
>>> a = [1, 2]
>>> b = [3, 4]
>>> f(a, b)
[1, 2, 3, 4]
>>> a, b ②
([1, 2, 3, 4], [3, 4])
>>> t = (10, 20)
>>> u = (30, 40)
>>> f(t, u) ③
(10, 20, 30, 40)
>>> t, u
((10, 20), (30, 40))
```

① O número `x` não se altera.

② A lista `a` é alterada.

③ A tupla t não se altera.

Outra questão relacionada a parâmetros de função é o uso de valores mutáveis como defaults, discutida a seguir.

6.5.1. Porque evitar tipos mutáveis como default em parâmetros

Parâmetros opcionais com valores default são um ótimo recurso para definição de funções em Python, permitindo que nossas APIs evoluam mantendo a compatibilidade com versões anteriores. Entretanto, evite usar objetos mutáveis como valores default em parâmetros.

Para ilustrar porquê, no Exemplo 97, modificamos o método `__init__` da classe `Bus` do Exemplo 93 para criar `HauntedBus`. Tentamos ser espertos: em vez do valor default `passengers=None`, temos `passengers=[]`, para evitar o if do `__init__` anterior. Essa "esperteza" causa problemas.

Exemplo 97. Uma classe simples ilustrando o perigo de um default mutável

```
class HauntedBus:  
    """A bus model haunted by ghost passengers"""  
  
    def __init__(self, passengers=[]): ①  
        self.passengers = passengers ②  
  
    def pick(self, name):  
        self.passengers.append(name) ③  
  
    def drop(self, name):  
        self.passengers.remove(name)
```

① Quando não passamos o argumento `passengers`, esse parâmetro é vinculado ao objeto lista `default`, que inicialmente está vazia.

② Essa atribuição torna `self.passengers` um apelido de `passengers`, que por sua vez é um apelido para a lista `default`, quando um argumento `passengers` não é passado para a função.

③ Quando os métodos `.remove()` e `.append()` são usados com `self.passengers`, estamos, na verdade, mudando a lista `default`, que é um atributo do objeto-função.

Exemplo 98 mostra o comportamento misterioso de `HauntedBus`.

Exemplo 98. Ônibus assombrados por passageiros fantasmas

```
>>> bus1 = HauntedBus(['Alice', 'Bill']) ①  
>>> bus1.passengers  
['Alice', 'Bill']  
>>> bus1.pick('Charlie')  
>>> bus1.drop('Alice')  
>>> bus1.passengers ②  
['Bill', 'Charlie']  
>>> bus2 = HauntedBus() ③  
>>> bus2.pick('Carrie')
```

```

>>> bus2.passengers
['Carrie']
>>> bus3 = HauntedBus() ④
>>> bus3.passengers ⑤
['Carrie']
>>> bus3.pick('Dave')
>>> bus2.passengers ⑥
['Carrie', 'Dave']
>>> bus2.passengers is bus3.passengers ⑦
True
>>> bus1.passengers ⑧
['Bill', 'Charlie']

```

- ① bus1 começa com uma lista de dois passageiros.
- ② Até aqui, tudo bem: nenhuma surpresa em bus1.
- ③ bus2 começa vazio, então a lista vazia default é vinculada a self.passengers.
- ④ bus3 também começa vazio, e novamente a lista default é atribuída.
- ⑤ A lista default não está mais vazia!
- ⑥ Agora Dave, pego pelo bus3, aparece no bus2.
- ⑦ O problema: bus2.passengers e bus3.passengers se referem à mesma lista.
- ⑧ Mas bus1.passengers é uma lista diferente.

O problema é que instâncias de `HauntedBus` que não recebem uma lista de passageiros inicial acabam todas compartilhando a mesma lista de passageiros entre si.

Este tipo de bug pode ser muito sutil. Como o Exemplo 98 demonstra, quando `HauntedBus` recebe uma lista com passageiros como parâmetro, ele funciona como esperado. Coisas estranhas acontecem somente quando `HauntedBus` começa vazio, pois aí `self.passengers` se torna um apelido para o valor default do parâmetro `passengers`. O problema é que cada valor default é processado quando a função é definida — normalmente quando o módulo é carregado — e os valores default se tornam atributos do objeto-função. Assim, se o valor default é um objeto mutável e você o altera, a alteração vai afetar todas as futuras chamadas da função.

Após executar as linhas do Exemplo 98, você pode inspecionar o objeto `HauntedBus.__init__` e ver os estudantes fantasma assombrando o atributo `__defaults__`:

```

>>> dir(HauntedBus.__init__)
['__annotations__', '__call__', ..., '__defaults__', ...]
>>> HauntedBus.__init__.__defaults__
(['Carrie', 'Dave'])

```

Por fim, podemos verificar que `bus2.passengers` é um apelido vinculado ao primeiro elemento do atributo `HauntedBus.__init__.__defaults__`:

```
>>> HauntedBus.__init__.__defaults__[0] is bus2.passengers
```

True

O problema com defaults mutáveis explica porque `None` é normalmente usado como valor default para parâmetros que podem receber valores mutáveis. No Exemplo 93, `__init__` checa se o argumento `passengers` é `None`. Se for, `self.passengers` é vinculado a uma nova lista vazia. Se `passengers` não for `None`, a implementação correta vincula uma cópia daquele argumento a `self.passengers`. A próxima seção explica porque copiar o argumento é uma boa prática.

6.5.2. Programação defensiva com argumentos mutáveis

Ao escrever uma função que recebe um argumento mutável, você deve considerar com cuidado se o cliente que chama sua função espera que o argumento passado seja modificado.

Por exemplo, se sua função recebe um `dict` e precisa modificá-lo durante seu processamento, esse efeito colateral deve ou não ser visível fora da função? A resposta, na verdade, depende do contexto. É tudo uma questão de alinhar as expectativas do autor da função com as do cliente da função.

O último exemplo com ônibus neste capítulo mostra como o `TwilightBus` viola as expectativas ao compartilhar sua lista de passageiros com seus clientes. Antes de estudar a implementação, veja como a classe `TwilightBus` funciona pela perspectiva de um cliente daquela classe, em Exemplo 99.

Exemplo 99. Passageiros desaparecem quando são deixados por um TwilightBus

```
>>> basketball_team = ['Sue', 'Tina', 'Maya', 'Diana', 'Pat'] ①
>>> bus = TwilightBus(basketball_team) ②
>>> bus.drop('Tina') ③
>>> bus.drop('Pat')
>>> basketball_team ④
['Sue', 'Maya', 'Diana']
```

① `basketball_team` contém o nome de cinco estudantes.

② Um `TwilightBus` é carregado com o time.

③ O bus deixa uma estudante, depois outra.

④ As passageiras desembarcadas desapareceram do time de basquete!

`TwilightBus` viola o "Princípio da Menor Surpresa", uma boa prática do design de interfaces.^[3] Com certeza, é surpreendente que quando o ônibus deixa uma estudante, seu nome seja removido da escalação do time de basquete.

Exemplo 100 é a implementação de `TwilightBus` e uma explicação do problema.

Exemplo 100. Uma classe simples mostrando os perigos de mudar argumentos recebidos

```
class TwilightBus:
    """A bus model that makes passengers vanish"""

    def __init__(self, passengers=None):
```

```

if passengers is None:
    self.passengers = [] ①
else:
    self.passengers = passengers ②

def pick(self, name):
    self.passengers.append(name)

def drop(self, name):
    self.passengers.remove(name) ③

```

- ① Aqui cuidadosamente criamos uma lista vazia quando `passengers` é `None`.
- ② Entretanto, esta atribuição transforma `self.passengers` em um apelido para `passengers`, que por sua vez é um apelido para o argumento passado para `__init__` (i.e. `basketball_team` em Exemplo 99).
- ③ Quando os métodos `.remove()` e `.append()` são usados com `self.passengers`, estamos, na verdade, modificando a lista original recebida como argumento pelo construtor.

O problema aqui é que o ônibus está apelidando a lista passada para o construtor. Ao invés disso, ele deveria manter sua própria lista de passageiros. A solução é simples: em `__init__`, quando o parâmetro `passengers` é fornecido, `self.passengers` deveria ser inicializado com uma cópia daquela lista, como fizemos, de forma correta, em Exemplo 93:

```

def __init__(self, passengers=None):
    if passengers is None:
        self.passengers = []
    else:
        self.passengers = list(passengers) ①

```

- ① Cria uma cópia da lista `passengers`, ou converte o argumento para `list` se ele não for uma lista.

Agora nossa manipulação interna da lista de passageiros não afetará o argumento usado para inicializar o ônibus. E com uma vantagem adicional, essa solução é mais flexível: agora o argumento passado no parâmetro `passengers` pode ser uma tupla ou qualquer outro tipo iterável, como `set` ou mesmo resultados de uma consulta a um banco de dados, pois o construtor de `list` aceita qualquer iterável. Ao criar nossa própria lista, estamos também assegurando que ela suporta os métodos necessários, `.remove()` e `.append()`, operações que usamos nos métodos `.pick()` e `.drop()`.



A menos que um método tenha o objetivo explícito de alterar um objeto recebido como argumento, você deveria pensar bem antes de apelidar tal objeto e simplesmente vinculá-lo a uma variável interna de sua classe. Quando em dúvida, crie uma cópia. Os clientes de sua classe ficarão mais felizes. Claro, criar uma cópia não é grátil: há custos de memória e processamento. Entretanto, uma API que causa bugs sutis é um problema bem maior que uma que seja um pouco mais lenta ou que use mais recursos.

Agora vamos conversar sobre uma das instruções mais incompreendidas em Python: `del`.

6.6. `del` e coleta de lixo

Os objetos nunca são destruídos explicitamente; no entanto, quando eles se tornam inacessíveis, eles podem ser coletados como lixo.

— “Modelo de Dados” capítulo de *A Referência da Linguagem Python*

A primeira surpresa de `del` é não ser uma função, mas uma instrução (*statement*).

Escrevemos `del x` e não `del(x)` — apesar dessa última forma funcionar também, mas apenas porque as expressões `x` e `(x)` em geral tem o mesmo significado em Python.

O segundo aspecto surpreendente é que `del` apaga referências, não objetos. A coleta de lixo pode eliminar um objeto da memória como resultado indireto de `del`, se a variável apagada for a última referência ao objeto. Reassociar uma variável também pode reduzir a zero o número de referências a um objeto, causando sua destruição.

```
>>> a = [1, 2] ①
>>> b = a      ②
>>> del a     ③
>>> b          ④
[1, 2]
>>> b = [3]    ⑤
```

① Cria o objeto `[1, 2]` e vincula `a` a ele.

② Vincula `b` ao mesmo objeto `[1, 2]`.

③ Apaga a referência `a`.

④ `[1, 2]` não é afetado, pois `b` ainda aponta para ele.

⑤ Reassociar `b` a um objeto diferente remove a última referência restante a `[1, 2]`. Agora o coletor de lixo pode descartar aquele objeto.



Existe um método especial `__del__`, mas ele não causa a remoção de uma instância e não deve ser invocado em seu código. O método `__del__` é invocado pelo interpretador Python quando a instância está prestes a ser destruída, para dar a ela a chance de liberar recursos externos. É muito raro ser preciso implementar `__del__` em seu código, mas ainda assim alguns programadores Python perdem tempo codando este método sem necessidade. O uso correto de `__del__` é bastante complexo. Consulte `__del__` [<https://fpy.li/3x>]
no capítulo “Modelo de Dados” em *A Referência da Linguagem Python*.

No CPython, o algoritmo primário de coleta de lixo é a contagem de referências. Essencialmente, cada objeto mantém uma contagem do número de referências apontando para si. Assim que a contagem chega a zero, o objeto é imediatamente destruído: CPython invoca o método `__del__` no objeto (se definido) e daí libera a memória alocada para aquele objeto. No CPython 2.0, um

algoritmo de coleta de lixo geracional foi acrescentado, para detectar grupos de objetos envolvidos em referências cíclicas — grupos que podem ser inacessíveis mesmo que existam referências restantes, quando todas as referências mútuas estão contidas dentro daquele grupo. Outras implementações de Python tem coletores de lixo mais sofisticados, que não se baseiam na contagem de referências, o que significa que o método `__del__` pode não ser chamado imediatamente quando não existem mais referências ao objeto. Veja "PyPy, Garbage Collection, and a Deadlock" [<https://fpy.li/6-7>] (EN) de A. Jesse Jiryu Davis para uma discussão sobre os usos próprios e impróprios de `__del__`.

Para demonstrar o fim da vida de um objeto, Exemplo 101 usa `weakref.finalize` para registrar uma função callback a ser chamada quando o objeto é destruído.

Exemplo 101. Assistindo o fim de um objeto quando não resta nenhuma referência apontando para ele

```
>>> import weakref
>>> s1 = {1, 2, 3}
>>> s2 = s1          ①
>>> def bye():       ②
...     print('...like tears in the rain.')
...
>>> ender = weakref.finalize(s1, bye)  ③
>>> ender.alive  ④
True
>>> del s1
>>> ender.alive  ⑤
True
>>> s2 = 'spam'   ⑥
...like tears in the rain.
>>> ender.alive
False
```

① `s1` e `s2` são apelidos do mesmo conjunto, `{1, 2, 3}`.

② Para essa demonstração, a função `bye` não deve ser um método vinculado ao objeto prestes a ser destruído, nem manter uma referência para o objeto.

③ Registra o callback `bye` no objeto referenciado por `s1`.

④ O atributo `.alive` é `True` antes do objeto `finalize` ser chamado.

⑤ Como vimos, `del` não apaga o objeto, apenas a referência `s1` a ele.

⑥ Reassociar a última referência, `s2`, torna `{1, 2, 3}` inacessível. Ele é destruído, o callback `bye` é invocado, e `ender.alive` se torna `False`.

O ponto principal de Exemplo 101 é mostrar explicitamente que `del` não apaga objetos, mas que objetos podem ser apagados como uma consequência de ficarem inacessíveis após o uso de `del`.

Você pode estar se perguntando porque o objeto `{1, 2, 3}` foi destruído em Exemplo 101. Afinal, a referência `s1` foi passada para a função `finalize`, que precisa tê-la mantida para conseguir monitorar o objeto e invocar o callback. Isso funciona porque `finalize` mantém uma *referência fraca* (*weak reference*) para `{1, 2, 3}`. Referências fracas não aumentam a contagem de referências de um objeto. Assim, uma referência fraca não evita que o objeto alvo seja removido pelo coletor de

lixo. Referências fracas são úteis em cenários de caching, pois não queremos que os objetos "cacheados" sejam mantidos vivos apenas por terem uma referência no cache.



Referências fracas são um tópico muito especializado, então decidi retirá-lo dessa segunda edição. Em vez disso, publiquei a nota "Weak References" em [fluentpython.com](https://fpy.li/weakref) [<https://fpy.li/weakref>].

6.7. Peças que Python prega com imutáveis



Esta seção opcional discute alguns detalhes que, na verdade, não são muito importantes para usuários de Python, e que podem não se aplicar a outras implementações da linguagem ou mesmo a futuras versões de CPython. Entretanto, já vi muita gente tropeçar nesses casos laterais e daí passar a usar o operador `is` de forma incorreta, então acho que vale a pena mencionar esses detalhes.

Fiquei surpreso ao descobrir que, dada uma tupla `t`, a chamada `t[:]` não cria uma cópia, mas devolve uma referência para o mesmo objeto. Da mesma forma, `tuple(t)` também retorna uma referência para a mesma tupla.^[4]

Exemplo 102 demonstra esse fato.

Exemplo 102. Uma tupla construída a partir de outra é, na verdade, exatamente a mesma tupla.

```
>>> t1 = (1, 2, 3)
>>> t2 = tuple(t1)
>>> t2 is t1 ①
True
>>> t3 = t1[:]
>>> t3 is t1 ②
True
```

① `t1` e `t2` estão vinculadas ao mesmo objeto

② Assim como `t3`.

O mesmo comportamento pode ser observado com instâncias de `str`, `bytes` e `frozenset`. Note que `frozenset` não é uma sequência, então `fs[:]` não funciona se `fs` é um `frozenset`. Mas `fs.copy()` tem o mesmo efeito: ele trapaceia e retorna uma referência ao mesmo objeto, e não uma cópia, como mostra Exemplo 103.^[5]

Exemplo 103. Strings literais podem criar objetos compartilhados.

```
>>> t1 = (1, 2, 3)
>>> t3 = (1, 2, 3) ①
>>> t3 is t1 ②
False
>>> s1 = 'ABC'
>>> s2 = 'ABC' ③
```

```
>>> s2 is s1 ④  
True
```

- ① Criando uma nova tupla do zero.
- ② t1 e t3 são iguais, mas não são o mesmo objeto.
- ③ Criando uma segunda str do zero.
- ④ Surpresa: a e b se referem à mesma str!

O compartilhamento de strings literais é uma técnica de otimização chamada *internalização* (*interning*). O CPython usa uma técnica similar com inteiros pequenos, para evitar a duplicação desnecessária de números que aparecem com muita frequência em programas, como 0, 1, -1, etc. Observe que o CPython não internaliza todas as strings e inteiros, e o critério pelo qual ele faz isso é um detalhe de implementação não documentado.



Nunca dependa da internalização de str ou int! Sempre use == em vez de is para verificar a igualdade de strings ou inteiros. A internalização é uma otimização para uso interno do interpretador Python.

Os truques discutidos nessa seção, incluindo o comportamento de `frozenset.copy()`, são mentiras infensivas que economizam memória e tornam o interpretador mais rápido. Não se preocupe, elas não trarão nenhum problema, pois se aplicam apenas a tipos imutáveis. Provavelmente, o melhor uso para esse tipo de detalhe é ganhar apostas contra outros Pythonistas.^[6]

6.8. Resumo do capítulo

Todo objeto em Python tem uma identidade, um tipo e um valor. Apenas o valor do objeto pode mudar ao longo do tempo.^[7]

Se duas variáveis se referem a objetos imutáveis de igual valor (`a == b` is `True`), na prática, dificilmente importa se elas se referem a cópias de mesmo valor ou são apelidos do mesmo objeto, porque o valor de objeto imutável não muda, com uma exceção. A exceção são as tuplas: se ela contém referências para itens mutáveis, então seu valor mudará se o valor de um item mutável for alterado. Na prática, esse cenário não é tão comum. O que nunca muda numa coleção imutável são as identidades dos objetos mantidos ali. A classe `frozenset` não sofre desse problema, porque ela só pode conter elementos hashable, e o valor de um objeto hashable não pode mudar, por definição.

O fato de variáveis conterem referências tem muitas consequências práticas para a programação em Python:

- Uma atribuição simples não cria cópias.
- Uma atribuição composta com `+=` ou `*=` cria novos objetos se a variável à esquerda da atribuição estiver vinculada a um objeto imutável, mas pode modificar um objeto mutável diretamente.
- Atribuir um novo valor a uma variável existente não muda o objeto previamente vinculado à variável. Isso se chama *rebinding* (re-vinculação); a variável passa a se referir a um objeto diferente. Se aquela variável era a última referência ao objeto anterior, aquele objeto será eliminado pela coleta de lixo.

- Parâmetros de função são passados como apelidos, o que significa que a função pode alterar qualquer objeto mutável recebido como argumento. Não há como evitar isso, exceto criando cópias locais ou usando objetos imutáveis (i.e., passando uma tupla em vez de uma lista)
- Usar objetos mutáveis como valores default de parâmetros de função é perigoso, pois se os parâmetros forem modificados pela função, o default muda, afetando chamadas posteriores que usem o default.

Em CPython, um objeto é descartado assim que o número de referências a ele chega a zero. Objetos também podem ser descartados se formarem grupos com referências cíclicas sem nenhuma referência externa ao grupo.

Em algumas situações, pode ser útil manter uma referência para um objeto que não vai, por si só, manter o objeto vivo. Um exemplo é uma classe que queira manter o registro de todas as suas instâncias atuais. Isso pode ser feito com referências fracas, um mecanismo de baixo nível encontrado nas coleções `WeakValueDictionary`, `WeakKeyDictionary`, `WeakSet`, e na função `finalize` do módulo `weakref`.

Leia "Weak References" em fluentpython.com [<https://fpy.li/weakref>] para mais detalhes sobre `weakref`.

6.9. Para saber mais

O capítulo "Modelo de Dados" [<https://fpy.li/2j>] de *A Referência da Linguagem Python* inicia com uma explicação bastante clara sobre identidades e valores de objetos.

Wesley Chun, autor da série *Core Python*, apresentou Understanding Python's Memory Model, Mutability, and Methods [<https://fpy.li/6-8>] (EN) na EuroPython 2011, discutindo não apenas o tema desse capítulo como também o uso de métodos especiais.

Doug Hellmann escreveu os posts "copy — Duplicate Objects" [<https://fpy.li/6-9>] (EN) e "weakref — Garbage-Collectable References to Objects" [<https://fpy.li/6-10>] (EN), cobrindo alguns dos tópicos que acabamos de tratar.

Você pode encontrar mais informações sobre o coletor de lixo geracional do CPython em `gc` — Interface para o coletor de lixo [<https://fpy.li/3y>], que começa com a frase "Este módulo fornece uma interface para o coletor de lixo opcional." O adjetivo "opcional" aqui pode ser surpreendente, mas o capítulo "Modelo de Dados" [<https://fpy.li/2j>] também afirma:

Uma implementação tem permissão para adiar a coleta de lixo ou omiti-la completamente — é um detalhe de implementação como a coleta de lixo é implementada, desde que nenhum objeto que ainda esteja acessível seja coletado.

Pablo Galindo escreveu um texto mais aprofundado sobre o Coletor de Lixo em Python, em "Design of CPython's Garbage Collector" [<https://fpy.li/6-12>] (EN) no *Python Developer's Guide* [<https://fpy.li/6-13>], voltado para contribuidores novos e experientes da implementação CPython.

O coletor de lixo do CPython 3.4 aperfeiçoou o tratamento de objetos contendo um método `__del__`, como descrito em PEP 442—Safe object finalization [<https://fpy.li/6-14>] (EN).

A Wikipedia tem um artigo sobre string interning [<https://fpy.li/6-15>] (EN), que menciona o uso desta técnica em várias linguagens, incluindo Python.

A Wikipedia também tem um artigo sobre "Haddocks' Eyes" [<https://fpy.li/6-16>], a canção de Lewis Carroll que mencionei no início deste capítulo. Os editores da Wikipedia escreveram que a letra é usada em trabalhos de lógica e filosofia "para elaborar o status simbólico do conceito de 'nome': um nome como um marcador de identificação pode ser atribuído a qualquer coisa, incluindo outro nome, introduzindo assim níveis diferentes de simbolização."

Ponto de vista

Tratamento igual para todos os objetos

Aprendi Java antes de conhecer Python. O operador `==` em Java sempre me pareceu equivocado. É mais comum que programadores estejam preocupados com a igualdade que com a identidade. Mas para objetos (não tipos primitivos), o `==` em Java compara referências, não valores dos objetos. Mesmo para algo tão básico quanto comparar strings, Java obriga você a usar o método `equals`. E mesmo assim, há outro problema: se você escrever `a.equals(b)` e `a` for `null`, você causa uma *null pointer exception* (exceção de ponteiro nulo). Os projetistas de Java sentiram necessidade de sobrecarregar `+` para strings; por que não mantiveram essa ideia e sobrecarregaram `==` também?

Python faz melhor. O operador `==` compara valores de objetos; `is` compara referências. E como Python permite sobrecarregar operadores, `==` funciona de forma sensata com todos os objetos na biblioteca padrão, incluindo `None`, que é um objeto de verdade, ao contrário do `null` de Java.

E claro, você pode definir `__eq__` nas suas próprias classes para controlar o que `==` significa para suas instâncias. Se você não sobrecarregar `__eq__`, o método herdado de `object` compara os IDs dos objetos, então a regra básica é que cada instância de uma classe definida pelo usuário é considerada diferente.

Estas são algumas das coisas que me fizeram mudar de Java para Python assim que terminei de ler *The Python Tutorial* em uma tarde de setembro de 1998.

Mutabilidade

Este capítulo não seria necessário se todos os objetos em Python fossem imutáveis. Quando você está lidando com objetos imutáveis, não faz diferença se as variáveis guardam os objetos em si ou referências para objetos compartilhados.

Se `a == b` é verdade, e nenhum dos dois objetos pode mudar, eles podem perfeitamente ser o mesmo objeto. Por isso a internalização de strings é segura. A identidade dos objetos só é importante quando esses objetos podem mudar.

Em programação funcional "pura", todos os dados são imutáveis: concatenar algo a uma coleção, na verdade, cria uma nova coleção. Elixir é uma linguagem funcional prática e fácil de aprender, na qual todos os tipos nativos são imutáveis, incluindo as listas.

Python, por outro lado, não é uma linguagem funcional, muito menos uma linguagem

funcional pura. Instâncias de classes definidas pelo usuário são mutáveis por default em Python — como na maioria das linguagens orientadas a objetos. Ao criar seus próprios objetos, você tem que tomar o cuidado adicional de torná-los imutáveis, se este for um requisito. Cada atributo do objeto precisa ser também imutável, senão você termina criando algo como uma tupla: imutável quanto ao ID do objeto, mas seu valor pode mudar se a tupla contiver um objeto mutável.

Objetos mutáveis também são a razão pela qual programar com threads é tão difícil: threads modificando objetos sem uma sincronização apropriada podem corromper dados. Sincronização excessiva, por outro lado, causa deadlocks. A linguagem e a plataforma Erlang — que inclui Elixir — foi projetada para maximizar o tempo de execução em aplicações distribuídas de alta concorrência, como aplicações de controle de telecomunicações. Naturalmente, eles escolheram tornar os dados imutáveis por default.

Destrução de objetos e coleta de lixo

Não existe em Python uma forma de destruir um objeto diretamente. E essa omissão é uma grande qualidade: se você pudesse destruir um objeto a qualquer momento, o que aconteceria com as referências que apontam para ele?

A coleta de lixo em CPython é feita principalmente por contagem de referências, que é fácil de implementar, mas vulnerável a vazamentos de memória (*memory leaks*) quando existem referências cíclicas. Assim, com a versão 2.0 (de outubro de 2000), um coletor de lixo geracional foi implementado, e ele consegue descartar objetos inatingíveis que foram mantidos vivos por ciclos de referências.

Mas a contagem de referências ainda está lá como mecanismo básico, e ela causa a destruição imediata de objetos com zero referências. Isso significa que, em CPython — pelo menos por hora — é seguro escrever:

```
open('test.txt', 'wt', encoding='utf-8').write('1, 2, 3')
```

Este código é seguro porque a contagem de referências do objeto arquivo será zero após o método `write` retornar, e o arquivo será fechado quando o objeto for descartado. Entretanto, a mesma linha não é segura em Jython ou IronPython, que usam o coletor de lixo dos runtimes de seus ambientes (a Java VM e a .NET CLR, respectivamente), que são mais sofisticados, mas não se baseiam em contagem de referências, e podem demorar mais para destruir o objeto e fechar o arquivo. Em todos os casos, incluindo em CPython, a melhor prática é fechar o arquivo explicitamente, e a forma mais confiável de fazer isso é usando a instrução `with`, que garante o fechamento do arquivo mesmo se acontecerem exceções enquanto ele estiver aberto. Usando `with`, a linha anterior se torna:

```
with open('test.txt', 'wt', encoding='utf-8') as fp:  
    fp.write('1, 2, 3')
```

Se você tiver interesse no assunto de coletores de lixo, você talvez queira ler o artigo de Thomas Perl, "Python Garbage Collector Implementations: CPython, PyPy and GaS"

[<https://fpy.li/6-17>] (EN), onde eu aprendi esses detalhes sobre a segurança de `open().write()` em CPython.

Passagem de parâmetros: chamada por compartilhamento

Uma maneira popular de explicar como a passagem de parâmetros funciona em Python é a frase: "Parâmetros são passados por valor, mas os valores são referências." Isso não está errado, mas causa confusão porque os modos mais comuns de passagem de parâmetros em linguagens tradicionais são *chamada por valor* (a função recebe uma cópia dos argumentos) e *chamada por referência* (a função recebe um ponteiro para o argumento).

Em Python, a função recebe uma cópia dos argumentos, mas os argumentos são sempre referências. Então o valor dos objetos referenciados podem ser alterados pela função, se eles forem mutáveis, mas sua identidade não. Além disso, como a função recebe uma cópia da referência em um argumento, reassociar essa referência no corpo da função não tem qualquer efeito fora da função. Adotei o termo *chamada por compartilhamento* depois de encontrar a definição de *call by sharing* no livro *Programming Language Pragmatics*, 3rd ed., de Michael L. Scott (Morgan Kaufmann), seção "8.3.1: Parameter Modes."

[1] Lynn Andrea Stein é uma aclamada educadora de ciências da computação. Ela atualmente leciona na Olin College of Engineering (EN) [<https://fpy.li/6-1>].

[2] Ao contrário de sequências planas de tipo único, como `str`, `byte` e `array.array`, que não contêm referências e sim seu conteúdo — caracteres, bytes e números — armazenado em um espaço contíguo de memória.

[3] Ver *Principle of least astonishment* [<https://fpy.li/6-5>] (EN).

[4] Isso está claramente documentado. Digite `help(tuple)` no console de Python e leia: "Se o argumento é uma tupla, o valor de retorno é o mesmo objeto." Pensei que sabia tudo sobre tuplas antes de escrever esse livro.

[5] Essa mentirinha inofensiva, do método `copy` não copiar nada, é justificável pela compatibilidade da interface: torna `frozenset` mais compatível com `set`. De qualquer forma, não faz diferença para o usuário final se dois objetos imutáveis idênticos são o mesmo ou são cópias.

[6] Um péssimo uso dessas informações seria perguntar sobre elas quando entrevistando candidatos a emprego ou criando perguntas para exames de "certificação". Há inúmeros fatos mais importantes e úteis para testar conhecimentos de Python.

[7] Na verdade, o tipo de um objeto pode ser modificado, bastando para isso atribuir uma classe diferente ao atributo `__class__` do objeto. Mas isso é uma perversão, e eu me arrependo de ter escrito essa nota de rodapé.

Parte II: Funções como objetos

7. Funções como objetos de primeira classe

Nunca achei que Python tenha sido fortemente influenciado por linguagens funcionais, independente do que outros digam ou pensem. Eu estava mais familiarizado com linguagens imperativas, como o C e o Algol e, apesar de ter tornado as funções objetos de primeira classe, não via Python como uma linguagem funcional.^{[1][2]}

— Guido van Rossum, BDFL de Python

No Python, funções são objetos de primeira classe. Estudiosos de linguagens de programação definem um "objeto de primeira classe" como uma entidade que pode ser:

- Criada durante a execução de um programa;
- Atribuída a uma variável ou a um elemento em uma estrutura de dados;
- Passada como argumento para uma função;
- Devolvida como o resultado de uma função.

Inteiros, strings e dicionários são outros exemplos de objetos de primeira classe no Python—nada de incomum aqui. Tratar funções como objetos de primeira classe é um recurso essencial das linguagens funcionais, como Clojure, Elixir e Haskell. Entretanto, funções de primeira classe são tão úteis que foram adotadas por linguagens muito populares, como Javascript, Go e até Java (desde o JDK 8), nenhuma das quais pretende ser uma "linguagem funcional".

Esse capítulo e quase toda a Parte III do livro exploram as aplicações práticas de se tratar funções como objetos.



O termo "funções de primeira classe" é largamente usado como uma forma abreviada de "funções como objetos de primeira classe". Ele não é ideal, pois sugere a existência de uma "elite" entre as funções. Em Python, todas as funções são de primeira classe.

7.1. Novidades nesse capítulo

A seção "Os nove sabores de objetos invocáveis" (Seção 7.5) se chamava "Sete sabores de objetos invocáveis" na primeira edição deste livro. Os novos invocáveis são corrotinas nativas e geradores assíncronos, introduzidos no Python 3.5 e 3.6, respectivamente. Ambos serão estudados no [ch_async], mas são mencionados aqui ao lado dos outros invocáveis.

A Seção 7.7.1 é nova, e fala de um recurso que surgiu no Python 3.8: parâmetros somente posicionais.

Transferi a discussão sobre acesso a anotações de funções durante a execução para a [runtime_annot_sec]. Quando escrevi a primeira edição, a PEP 484—Type Hints (*Dicas de Tipo*) [<https://fpy.li/pep484>] (EN) ainda estava sendo considerada, e as anotações eram usadas de várias formas diferentes. Desde Python 3.5, anotações precisam estar em conformidade com a PEP 484.

Assim, o melhor lugar para falar delas é na a discussão sobre as dicas de tipo.

A primeira edição desse livro continha seções sobre a introspecção de objetos função, que desciam a detalhes de baixo nível e desviavam do assunto principal do capítulo. Reuni aquelas seções em um post intitulado "Introspection of Function Parameters" (*Introspecção de Parâmetros de Funções*) [<https://fpy.li/7-2>], no fluentpython.com.

Agora vamos ver porque as funções de Python são objetos completos.

7.2. Tratando uma função como um objeto

A sessão de console no Exemplo 104 mostra que funções de Python são objetos. Ali criamos uma função, a chamamos, lemos seu atributo `__doc__` e verificamos que o próprio objeto função é uma instância da classe `function`.

Exemplo 104. Cria e testa uma função, e então lê seu `__doc__` e verifica seu tipo

```
>>> def factorial(n): ①
...     """returns n!
...     return 1 if n < 2 else n * factorial(n - 1)
...
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
>>> factorial.__doc__ ②
'returns n!'
>>> type(factorial) ③
<class 'function'>
```

① Isso é uma sessão do console, então estamos criando uma função "durante a execução".

② `__doc__` é um dos muitos atributos de objetos função.

③ `factorial` é um instância da classe `function`.

O atributo `__doc__` é usado para gerar o texto de ajuda de um objeto. No console de Python, a instrução `help(factorial)` mostrará uma tela como a Figura 22.

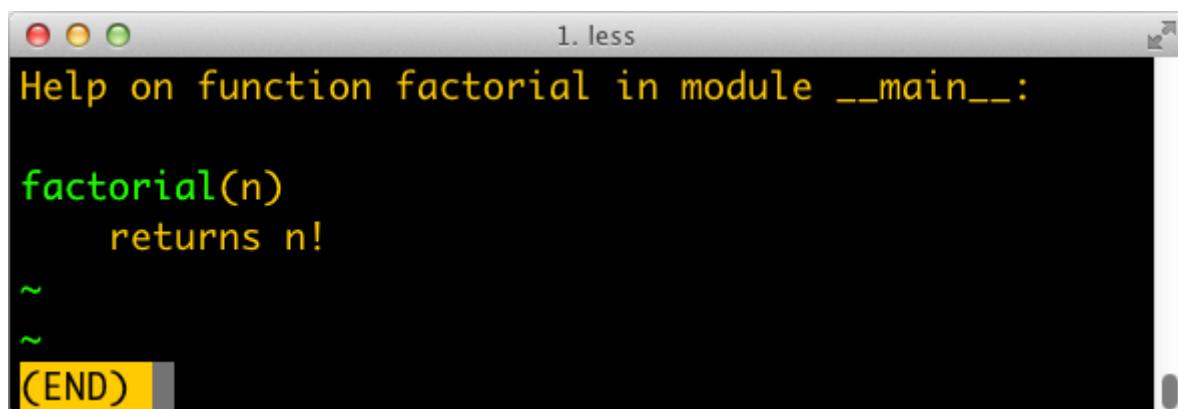


Figura 22. Tela de ajuda para `factorial`; o texto é criado a partir do atributo `__doc__` da função.

O Exemplo 105 mostra a natureza de "primeira classe" de um objeto função. Podemos atribuir tal objeto a uma variável `fact` e invocá-lo por esse nome. Podemos também passar `factorial` como argumento para a função `map` [https://fpy.li/44]. Invocar `map(function, iterable)` devolve um iterável no qual cada item é o resultado de uma chamada ao primeiro argumento (uma função) com elementos sucessivos do segundo argumento (um iterável), `range(11)` no exemplo.

Exemplo 105. Invoca factorial através da variável fact, e passa factorial como argumento para map

```
>>> fact = factorial
>>> fact
<function factorial at 0x...>
>>> fact(5)
120
>>> map(factorial, range(11))
<map object at 0x...>
>>> list(map(factorial, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Ter funções de primeira classe permite programar em um estilo funcional. Um dos marcos da programação funcional [https://fpy.li/45] é o uso de funções de ordem superior, nosso próximo tópico.

7.3. Funções de ordem superior

Uma função que recebe uma função como argumento ou devolve uma função como resultado é uma *função de ordem superior*. Uma dessas funções é `map`, usada no Exemplo 105. Outra é a função embutida `sorted`: o argumento opcional `key` permite fornecer uma função, que será então aplicada na ordenação de cada item, como vimos na Seção 2.9. Por exemplo, para ordenar uma lista de palavras por tamanho, passe a função `len` como `key`, como no Exemplo 106.

Exemplo 106. Ordenando uma lista de palavras por tamanho

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=len)
['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']
>>>
```

Qualquer função com um argumento pode ser usada como chave. Por exemplo, para criar um dicionário de rimas pode ser útil ordenar cada palavra escrita ao contrário. No Exemplo 107, observe que as palavras na lista não são modificadas de forma alguma; apenas suas versões escritas na ordem inversa são utilizadas como critério de ordenação. Por isso as *berries* aparecem juntas.

Exemplo 107. Ordenando uma lista de palavras pela ordem inversa de escrita

```
>>> def reverse(word):
...     return word[::-1]
>>> reverse('testing')
'gnitset'
>>> sorted(fruits, key=reverse)
```

```
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']  
>>>
```

No paradigma funcional de programação, algumas das funções de ordem superior mais conhecidas são `map`, `filter`, `reduce` e `apply`. A função `apply` foi descontinuada no Python 2.3 e removida no Python 3, por não ser mais necessária. Se você precisar chamar uma função com um conjuntos dinâmico de argumentos, pode escrever `fn(*args, **kwargs)` em vez de `apply(fn, args, kwargs)` como fazíamos no século passado.

As funções de ordem superior `map`, `filter` e `reduce` ainda existem, mas temos alternativas melhores para a maioria de seus casos de uso, como mostra a próxima seção.

7.3.1. Substitutos modernos para `map`, `filter`, e `reduce`

Linguagens funcionais normalmente oferecem as funções de ordem superior `map`, `filter`, and `reduce` (algumas vezes com nomes diferentes). As funções `map` e `filter` ainda estão embutidas no Python mas elas não são mais tão importantes desde a introdução das comprehensões de lista e das expressões geradoras. Uma `listcomp` ou uma `genexp` fazem o mesmo que `map` e `filter` combinadas, e são mais legíveis. Considere o Exemplo 108.

Exemplo 108. Listas de fatoriais produzidas com `map` e `filter`, comparadas com alternativas escritas com comprehensões de lista

```
>>> list(map(factorial, range(6))) ①  
[1, 1, 2, 6, 24, 120]  
>>> [factorial(n) for n in range(6)] ②  
[1, 1, 2, 6, 24, 120]  
>>> list(map(factorial, filter(lambda n: n % 2, range(6)))) ③  
[1, 6, 120]  
>>> [factorial(n) for n in range(6) if n % 2] ④  
[1, 6, 120]  
>>>
```

① Cria uma lista de fatoriais de 0! a 5!.

② Mesma operação, com uma compreensão de lista.

③ Lista de fatoriais de números ímpares até 5!, usando `map` e `filter`.

④ A compreensão de lista realiza a mesma tarefa, substituindo `map` e `filter`, e tornando `lambda` desnecessário.

No Python 3, `map` e `filter` devolvem geradores—uma forma de iterador—então sua substituta direta agora é uma expressão geradora (no Python 2, essas funções devolviam listas, então sua alternativa mais próxima era a compreensão de lista).

A função `reduce` foi rebaixada de função embutida, no Python 2, para o módulo `functools` no Python 3. Seu caso de uso mais comum, a somatória, é melhor servido pela função embutida `sum`, disponível desde o Python 2.3 (lançado em 2003). A função `sum` é mais legível e mais eficiente:

Exemplo 109. Soma de inteiros até 99, realizada com reduce e sum

```
>>> from functools import reduce ①
>>> from operator import add ②
>>> reduce(add, range(100)) ③
4950
>>> sum(range(100)) ④
4950
>>>
```

- ① A partir de Python 3.0, reduce deixou de ser uma função embutida.
- ② Importa add para evitar a criação de uma função apenas para somar dois números.
- ③ Soma os inteiros até 99.
- ④ Mesma operação, com sum—não é preciso importar nem chamar reduce e add.



A ideia comum de sum e reduce é aplicar alguma operação sucessivamente a itens em uma série, acumulando os resultados anteriores, reduzindo assim uma série de valores a um único valor.

Outras funções de redução embutidas são all e any:

all(iterable)

Devolve True se não há nenhum elemento falso no iterável; all([]) devolve True.

any(iterable)

Devolve True se qualquer elemento do iterable for verdadeiro; any([]) devolve False.

Dou um explicação mais completa sobre reduce na [multi_hashing], onde um exemplo mais longo, atravessando várias seções, cria um contexto significativo para o uso dessa função. As funções de redução serão resumidas mais à frente no livro, na [iterable_reducing_sec], quando estivermos tratando dos iteráveis. Para usar uma função de ordem superior, às vezes é conveniente criar um pequena função, que será usada apenas uma vez. As funções anônimas existem para isso. Vamos falar delas a seguir.

7.4. Funções anônimas

A palavra reservada lambda cria uma função anônima dentro de uma expressão Python.

Entretanto, a sintaxe simples de Python obriga que o corpo de uma lambda seja uma expressão. Em outras palavras, o corpo não pode conter instruções como while, try, etc. A atribuição com = também é uma instrução, então não pode ocorrer em um lambda. A nova sintaxe da expressão de atribuição, usando :=, pode ser usada. Porém, se você precisar dela, sua lambda provavelmente é muito complicada e difícil de ler, e deveria ser refatorado para uma função nomeada usando def.

O melhor uso das funções anônimas é no contexto de uma lista de argumentos para uma função de ordem superior. Por exemplo, o Exemplo 110 é o exemplo do dicionário de rimas do Exemplo 107 reescrito com lambda, sem definir uma função reverse.

Exemplo 110. Ordenando uma lista de palavras escritas na ordem inversa usando lambda

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=lambda word: word[::-1])
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

Fora do contexto limitado dos argumentos das funções de ordem superior, funções anônimas são pouco úteis no Python. As restrições sintáticas tendem a tornar ilegíveis as lambdas não-triviais. Se uma lambda é difícil de ler, sugiro fortemente seguir o conselho de Fredrik Lundh sobre refatoração.

A receita de Fredrik Lundh para refatoração de lambdas

Se você encontrar um trecho de código difícil de entender por causa de uma lambda, Fredrik Lundh sugere o seguinte procedimento de refatoração:

1. Escreva um comentário explicando o que diabos aquela lambda faz.
2. Estude o comentário por algum tempo, e pense em um nome que traduza sua essência.
3. Converta a lambda para uma declaração def, usando aquele nome.
4. Remova o comentário.

Esse passos são uma citação do *Programação Funcional—COMO FAZER* [<https://fpy.li/46>], leitura obrigatória.

A sintaxe lambda é apenas açúcar sintático: uma expressão lambda cria um objeto função, exatamente como a declaração def. Esse é apenas um dos vários tipos de objetos invocáveis no Python. Na próxima seção revisamos todos eles.

7.5. Os nove sabores de objetos invocáveis

O operador de invocação () pode ser aplicado a outros objetos além de funções. Para determinar se um objeto é invocável, use a função embutida callable(). No Python 3.9, a documentação do modelo de dados [<https://fpy.li/47>] lista nove tipos invocáveis:

Funções definidas pelo usuário

Criadas com instruções def ou expressões lambda.

Funções embutidas

Funções implementadas em C (no CPython), como len ou time.strftime.

Métodos embutidos

Métodos implementados em C, como dict.get.

Métodos

Funções definidas no corpo de uma classe.

Classes

Quando invocada, uma classe executa seu método `__new__` para criar uma instância, e a seguir `__init__`, para inicializá-la. Então a instância é devolvida ao usuário. Como não existe um operador `new` no Python, invocar uma classe é como invocar uma função.^[3]

Instâncias de classe

Se uma classe define um método `__call__`, suas instâncias podem então ser invocadas como funções—esse é o assunto da próxima seção.

Funções geradoras

Funções ou métodos que usam a palavra reservada `yield` em seu corpo. Quando chamadas, devolvem um objeto gerador.

Funções de corrotinas nativas

Funções ou métodos definidos com `async def`. Quando chamados, devolvem um objeto corrotina. Introduzidas no Python 3.5.

Funções geradoras assíncronas

Funções ou métodos definidos com `async def`, contendo `yield` em seu corpo. Quando chamados, devolvem um gerador assíncrono para ser usado com `async for`. Introduzidas no Python 3.6.

Funções geradoras, funções de corrotinas nativas e geradoras assíncronas são diferentes de outros invocáveis: os valores devolvidos tais funções nunca são dados da aplicação, mas objetos que exigem processamento adicional, seja para produzir dados da aplicação, seja para realizar algum trabalho útil. Funções geradoras devolvem iteradores. Ambos são tratados no [ch_generators]. Funções de corrotinas nativas e funções geradoras assíncronas devolvem objetos que só funcionam com a ajuda de um framework de programação assíncrona, tal como `asyncio`. Elas são o assunto do [ch_async].

Dada a variedade dos tipos de invocáveis existentes no Python, a forma mais segura de determinar se um objeto é invocável é usando a função embutida `callable()`:



```
>>> abs, str, 'Ni!'
(<built-in function abs>, <class 'str'>, 'Ni!')
>>> [callable(obj) for obj in (abs, str, 'Ni!')]
[True, True, False]
```

Vamos agora criar instâncias de classes que funcionam como objetos invocáveis.

7.6. Tipos invocáveis definidos pelo usuário

Além das funções serem objetos reais, também é possível fazer com que objetos arbitrários se comportem como funções. Para isso, basta implementar o método de instância `__call__`.

O Exemplo 111 implementa uma classe `BingoCage`. Uma instância é criada a partir de qualquer iterável, e mantém uma `list` interna de itens, em ordem aleatória. Invocar a instância extrai um

item.^[4]

Exemplo 111. bingocall.py: Uma BingoCage faz apenas uma coisa: escolhe itens de uma lista embaralhada

```
import random

class BingoCage:

    def __init__(self, items):
        self._items = list(items) ①
        random.shuffle(self._items) ②

    def pick(self): ③
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage') ④

    def __call__(self): ⑤
        return self.pick()
```

① `__init__` aceita qualquer iterável; criar uma cópia local evita efeitos colaterais inesperados sobre qualquer `list` passada como argumento.

② `shuffle` sempre vai funcionar, pois `self._items` é uma `list`.

③ O método principal.

④ Se `self._items` está vazia, gera uma exceção com uma mensagem apropriada.

⑤ Atalho para `bingo.pick(): bingo()`.

Aqui está uma demonstração simples do Exemplo 111. Observe como uma instância de `bingo` pode ser invocada como uma função, e como a função embutida `callable()` a reconhece como um objeto invocável:

```
>>> bingo = BingoCage(range(3))
>>> bingo.pick()
1
>>> bingo()
0
>>> callable(bingo)
True
```

Uma classe que implementa `__call__` é uma forma fácil de criar objetos similares a funções, com algum estado interno que precisa ser mantido de uma invocação para outra, como os itens restantes na `BingoCage`. Outro bom caso de uso para `__call__` é a implementação de decoradores. Decoradores devem ser invocáveis, e muitas vezes é conveniente "lembrar" algo entre chamadas ao decorador por exemplo, para *memoization* (a manutenção dos resultados de algum processamento complexo e/ou demorado para uso posterior) ou para separar uma implementação complexa entre vários métodos.

A abordagem funcional para a criação de funções com estado interno é através do uso de clausuras (*closures*). Clausuras e decoradores são o assunto do [ch_closure_decorator].

Vamos agora explorar a poderosa sintaxe oferecida pelo Python para declarar parâmetros de funções, e para passar argumentos para elas.

7.7. De parâmetros posicionais a parâmetros somente nomeados

Um dos melhores recursos das funções Python é sua sintaxe muito flexível para declaração e tratamento de parâmetros. Exemplos disso são os usos de * e ** para desempacotar e capturar iteráveis e mapeamentos em argumentos separados na chamada de uma função. Para ver esses recursos em ação, veja o código do Exemplo 112 e os testes mostrando seu uso no Exemplo 113.

Exemplo 112. tag gera elementos HTML; um argumento somente nomeado class_ é usado para passar atributos "class"; o _ é necessário porque class é uma palavra reservada no Python

```
def tag(name, *content, class_=None, **attrs):
    """Generate one or more HTML tags"""
    if class_ is not None:
        attrs['class'] = class_
    attr_pairs = (f' {attr}="{value}"' for attr, value
                  in sorted(attrs.items()))
    attr_str = '.join(attr_pairs)
    if content:
        elements = (f'<{name}{attr_str}>{c}</{name}>'
                    for c in content)
        return '\n'.join(elements)
    else:
        return f'<{name}{attr_str} />'
```

A função tag pode ser invocada de muitas formas, como demonstra o Exemplo 113.

Exemplo 113. Algumas das muitas formas de invocar a função tag do Exemplo 112

```
>>> tag('br') ①
'<br />'
>>> tag('p', 'hello') ②
'<p>hello</p>'
>>> print(tag('p', 'hello', 'world'))
<p>hello</p>
<p>world</p>
>>> tag('p', 'hello', id=33) ③
'<p id="33">hello</p>'
>>> print(tag('p', 'hello', 'world', class_='sidebar')) ④
<p class="sidebar">hello</p>
<p class="sidebar">world</p>
>>> tag(content='testing', name='img') ⑤
'<img content="testing" />'
```

```
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...             'src': 'sunset.jpg', 'class': 'framed'}
>>> tag(**my_tag) ⑥
''
```

- ① Um argumento posicional único produz uma tag vazia com aquele nome.
- ② Quaisquer argumentos após o primeiro serão capturados por *content na forma de uma tuple.
- ③ Argumentos nomeados que não são mencionados explicitamente na assinatura de tag são capturados por **attrs como um dict.
- ④ O parâmetro class_ só pode ser passado como um argumento nomeado.
- ⑤ O primeiro argumento posicional também pode ser passado como argumento nomeado.
- ⑥ Prefixar o dict my_tag com ** passa todos os seus itens como argumentos separados, que são então vinculados aos parâmetros nomeados, com o restante sendo capturado por **attrs. Nesse caso podemos ter uma chave 'class' no dict de argumentos, porque é uma string, e não colide com a palavra reservada class.

Argumentos somente nomeados são um recurso de Python 3. No Exemplo 112, o parâmetro class_ só pode ser passado como um argumento nomeado—ele nunca captura argumentos posicionais não-nomeados. Para especificar argumentos somente nomeados ao definir uma função, eles devem ser nomeados após o argumento prefixado por *. Se você não quer incluir argumentos posicionais variáveis, mas ainda assim deseja incluir argumentos somente nomeados, coloque um * sozinho na assinatura, assim:

```
>>> def f(a, *, b):
...     return a, b
...
>>> f(1, b=2)
(1, 2)
>>> f(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes 1 positional argument but 2 were given
```

Observe que argumentos somente nomeados não precisam ter um valor default: eles podem ser obrigatórios, como o b no exemplo acima.

7.7.1. Parâmetros somente posicionais

Desde Python 3.8, assinaturas de funções definidas pelo usuário podem especificar parâmetros somente posicionais. Esse recurso sempre existiu para funções embutidas, tal como `divmod(a, b)`, que só pode ser chamada com parâmetros posicionais, e não na forma `divmod(a=10, b=4)`.

Para definir uma função que requer parâmetros somente posicionais, use / na lista de parâmetros.

Esse exemplo, de "O que há de novo no Python 3.8" [<https://fpy.li/48>], mostra como emular a função embutida `divmod`:

```
def divmod(a, b, /):
    return (a // b, a % b)
```

Todos os argumentos à esquerda da `/` são somente posicionais. Após a `/`, você pode especificar outros argumentos, que funcionam como da forma usual.



Uma `/` na lista de parâmetros é um erro de sintaxe no Python 3.7 ou anteriores.

Por exemplo, considere a função `tag` do Exemplo 112. Se quisermos que o parâmetro `name` seja somente positional, podemos acrescentar uma `/` após aquele parâmetro na assinatura da função, assim:

```
def tag(name, /, *content, class_=None, **attrs):
    ...
```

Você pode encontrar outros exemplos de parâmetros somente posicionais no já citado "O que há de novo no Python 3.8" [<https://fpy.li/48>] e na PEP 570 [<https://fpy.li/pep570>].

Após esse mergulho nos recursos flexíveis de declaração de argumentos no Python, o resto desse capítulo trata dos pacotes da biblioteca padrão mais úteis para programar em um estilo funcional.

7.8. Pacotes para programação funcional

Apesar de Guido deixar claro que não projetou Python para ser uma linguagem de programação funcional, o estilo de programação funcional pode ser amplamente utilizado, graças a funções de primeira classe, *pattern matching* e o suporte de pacotes como `operator` e `functools`, dos quais falaremos nas próximas duas seções..

7.8.1. O módulo `operator`

Na programação funcional, é muitas vezes conveniente usar um operador aritmético como uma função. Por exemplo, suponha que você queira multiplicar uma sequência de números para calcular fatoriais, mas sem usar recursão. Para calcular a soma, podemos usar `sum`, mas não há uma função equivalente para multiplicação. Você poderia usar `reduce`—como vimos na Seção 7.3.1—mas isso exige um função para multiplicar dois itens da sequência. O Exemplo 114 mostra como resolver esse problema usando `lambda`.

Exemplo 114. Fatorial implementado com `reduce` e uma função anônima

```
from functools import reduce

def factorial(n):
    return reduce(lambda a, b: a*b, range(1, n+1))
```

O módulo `operator` oferece funções equivalentes a dezenas de operadores, para você não precisar escrever funções triviais como `lambda a, b: a*b`. Com ele, podemos reescrever o Exemplo 114 como

no Exemplo 115.

Exemplo 115. Fatorial implementado com reduce e operator.mul

```
from functools import reduce
from operator import mul

def factorial(n):
    return reduce(mul, range(1, n+1))
```

Outro grupo de "lambdas de um só truque" que operator substitui são funções para extrair itens de sequências ou para ler atributos de objetos: itemgetter e attrgetter são fábricas que criam funções customizadas para fazer exatamente isso.

O Exemplo 116 mostra um uso frequente de itemgetter: ordenar uma lista de tuplas pelo valor de um campo. No exemplo, as cidades são exibidas por ordem de código de país (campo 1). Essencialmente, itemgetter(1) cria uma função que, dada uma coleção, devolve o item no índice 1. Isso é mais fácil de escrever e ler que lambda fields: fields[1], que faz a mesma coisa.

Exemplo 116. Demonstração de itemgetter para ordenar uma lista de tuplas (mesmos dados do Exemplo 10)

```
>>> metro_data = [
...     ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
...     ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
...     ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
...     ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
...     ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
... ]
>>>
>>> from operator import itemgetter
>>> for city in sorted(metro_data, key=itemgetter(1)):
...     print(city)
...
('São Paulo', 'BR', 19.649, (-23.547778, -46.635833))
('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889))
('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
('Mexico City', 'MX', 20.142, (19.433333, -99.133333))
('New York-Newark', 'US', 20.104, (40.808611, -74.020386))
```

Se você passar múltiplos argumentos de indice para itemgetter, a função criada por ela vai devolver tuplas com os valores extraídos, algo que pode ser útil para ordenar usando chaves múltiplas:

```
>>> cc_name = itemgetter(1, 0)
>>> for city in metro_data:
...     print(cc_name(city))
...
('JP', 'Tokyo')
```

```

('IN', 'Delhi NCR')
('MX', 'Mexico City')
('US', 'New York-Newark')
('BR', 'São Paulo')
>>>

```

Como `itemgetter` usa o operador `[]`, ela suporta não apenas sequências, mas também mapeamentos e qualquer classe que implemente `__getitem__`.

Uma irmã de `itemgetter` é `attrgetter`, que cria funções para extrair atributos por nome. Se você passar os nomes de vários atributos como argumentos para `attrgetter`, ela vai devolver um tupla de valores. Além disso, se o nome de qualquer argumento contiver um `.` (ponto), `attrgetter` navegará por objetos aninhados para encontrar o atributo. Esses comportamento são apresentados no Exemplo 117. Não é exatamente uma sessão de console curta, pois precisamos criar uma estrutura aninhada para demonstrar o tratamento de atributos com `.` por `attrgetter`.

Exemplo 117. Demonstração de attrgetter para processar uma lista previamente definida de namedtuple chamada metro_data (a mesma lista que aparece no Exemplo 116)

```

>>> from collections import namedtuple
>>> LatLon = namedtuple('LatLon', 'lat lon') ①
>>> Metropolis = namedtuple('Metropolis', 'name cc pop coord') ②
>>> metro_areas = [Metropolis(name, cc, pop, LatLon(lat, lon)) ③
...     for name, cc, pop, (lat, lon) in metro_data]
>>> metro_areas[0]
Metropolis(name='Tokyo', cc='JP', pop=36.933, coord=LatLon(lat=35.689722,
lon=139.691667))
>>> metro_areas[0].coord.lat ④
35.689722
>>> from operator import attrgetter
>>> name_lat = attrgetter('name', 'coord.lat') ⑤
>>>
>>> for city in sorted(metro_areas, key=attrgetter('coord.lat')): ⑥
...     print(name_lat(city)) ⑦
...
('São Paulo', -23.547778)
('Mexico City', 19.433333)
('Delhi NCR', 28.613889)
('Tokyo', 35.689722)
('New York-Newark', 40.808611)

```

① Usa `namedtuple` para definir `LatLon`.

② Também define `Metropolis`.

③ Cria a lista `metro_areas` com instâncias de `Metropolis`; observe o desempacotamento da tupla aninhada para extrair (`lat`, `lon`) e usá-los para criar o `LatLon` do atributo `coord` de `Metropolis`.

④ Obtém a latitude de dentro de `metro_areas[0]`.

⑤ Define um `attrgetter` para obter `name` e o atributo aninhado `coord.lat`.

- ⑥ Usa `attrgetter` novamente para ordenar uma lista de cidades pela latitude.
- ⑦ Usa o `attrgetter` definido antes para exibir apenas o nome e a latitude da cidade.

Abaixo está uma lista parcial das funções definidas em `operator` (filtrei os nomes iniciando com `_` porque são, em sua maioria, detalhes de implementação):

```
>>> [name for name in dir(operator) if not name.startswith('_')]
['abs', 'add', 'and_', 'attrgetter', 'concat', 'contains',
'countOf', 'delitem', 'eq', 'floordiv', 'ge', 'getitem', 'gt',
'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imatmul',
'imod', 'imul', 'index', 'indexOf', 'inv', 'invert', 'ior',
'ipow', 'irshift', 'is_', 'is_not', 'isub', 'itemgetter',
'itruediv', 'ixor', 'le', 'length_hint', 'lshift', 'lt', 'matmul',
'methodcaller', 'mod', 'mul', 'ne', 'neg', 'not_', 'or_', 'pos',
'pow', 'rshift', 'setitem', 'sub', 'truediv', 'truth', 'xor']
```

A maior parte dos 54 nomes listados é mais ou menos evidente. O grupo de nomes formados por um `i` inicial e o nome de um operador—por exemplo `iadd`, `iand`, etc—correspondem aos operadores de atribuição aumentada—por exemplo, `+=`, `&=`, etc. Essas funções mudam seu primeiro argumento no mesmo lugar, se o argumento for mutável; se não, funcionam como seus pares sem o prefixo `i`: simplesmente devolvem o resultado da operação.

Das funções restantes de `operator`, `methodcaller` será a última que veremos. Ela é algo similar a `attrgetter` e `itemgetter`, no sentido de criarem uma função durante a execução. A função criada invoca por nome um método do objeto passado como argumento, como mostra o Exemplo 118.

Exemplo 118. Demonstração de `methodcaller`: o segundo teste mostra a vinculação de argumentos adicionais

```
>>> from operator import methodcaller
>>> s = 'The time has come'
>>> upcase = methodcaller('upper')
>>> upcase(s)
'THE TIME HAS COME'
>>> hyphenate = methodcaller('replace', ' ', '-')
>>> hyphenate(s)
'The-time-has-come'
```

O primeiro teste no Exemplo 118 está ali apenas para mostrar o funcionamento de `methodcaller`; se você precisa usar `str.upper` como uma função, basta chamá-lo na classe `str`, passando uma string como argumento, assim:

```
>>> str.upper(s)
'THE TIME HAS COME'
```

O segundo teste do Exemplo 118 mostra que `methodcaller` pode também executar uma aplicação parcial para fixar alguns argumentos, como faz a função `functools.partial`. Esse é nosso próximo

tópico.

7.8.2. Fixando argumentos com `functools.partial`

O módulo `functools` oferece várias funções de ordem superior. Já vimos `reduce` na Seção 7.3.1. Uma outra é `partial`: dado um invocável, ela produz um novo invocável com alguns dos argumentos do invocável original vinculados a valores pré-determinados. Isso é útil para adaptar uma função que recebe um ou mais argumentos a uma API que requer uma função de callback com menos argumentos. O Exemplo 119 é uma demonstração trivial.

Exemplo 119. Empregando `partial` para usar uma função com dois argumentos onde é necessário um invocável com apenas um argumento

```
>>> from operator import mul
>>> from functools import partial
>>> triple = partial(mul, 3) ①
>>> triple(7) ②
21
>>> list(map(triple, range(1, 10))) ③
[3, 6, 9, 12, 15, 18, 21, 24, 27]
```

① Cria uma nova função `triple` a partir de `mul`, vinculando o primeiro argumento posicional a 3.

② Testa a função.

③ Usa `triple` com `map`; `mul` não funcionaria com `map` nesse exemplo.

Um exemplo mais útil envolve a função `unicode.normalize`, que vimos na Seção 4.7. Se você trabalha com texto em muitas línguas diferentes, pode querer aplicar `unicode.normalize('NFC', s)` a qualquer string `s`, antes de compará-la ou armazená-la. Se você precisa disso com frequência, é conveniente ter uma função `nfc` para executar essa tarefa, como no Exemplo 120.

Exemplo 120. Criando uma função conveniente para normalizar Unicode com `partial`

```
>>> import unicodedata, functools
>>> nfc = functools.partial(unicodedata.normalize, 'NFC')
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> s1 == s2
False
>>> nfc(s1) == nfc(s2)
True
```

`partial` recebe um invocável como primeiro argumento, seguido de um número arbitrário de argumentos posicionais e nomeados para vincular.

O Exemplo 121 mostra o uso de `partial` com a função `tag` (do Exemplo 112), para fixar um argumento posicional e um argumento nomeado.

Exemplo 121. Demonstração de partial aplicada à função tag, do Exemplo 112

```
>>> from tagger import tag
>>> tag
<function tag at 0x10206d1e0> ①
>>> from functools import partial
>>> picture = partial(tag, 'img', class_='pic-frame') ②
>>> picture(src='wumpus.jpeg')
'' ③
>>> picture
functools.partial(<function tag at 0x10206d1e0>, 'img', class_='pic-frame') ④
>>> picture.func ⑤
<function tag at 0x10206d1e0>
>>> picture.args
('img',)
>>> picture.keywords
{'class_': 'pic-frame'}
```

① Importa tag do Exemplo 112 e mostra seu ID.

② Cria a função picture a partir de tag, fixando o primeiro argumento posicional em 'img' e o argumento nomeado class_ em 'pic-frame'.

③ picture funciona como esperado.

④ partial() devolve um objeto functools.partial.^[5]

⑤ Um objeto functools.partial tem atributos que fornecem acesso à função original e aos argumentos fixados.

A função functools.partialmethod faz o mesmo que partial, mas serve para trabalhar com métodos.

O módulo functools também inclui funções de ordem superior para serem usadas como decoradores de função, como cache e singledispatch, entre outras. Essas funções são tratadas no [ch_closure_decorator], que também explica como implementar decoradores customizados.

7.9. Resumo do capítulo

O objetivo deste capítulo foi explorar a natureza das funções como objetos de primeira classe no Python. As principais consequências disso são a possibilidade de atribuir funções a variáveis, passá-las para outras funções, armazená-las em estruturas de dados e acessar os atributos de funções, permitindo que frameworks e ferramentas usem essas informações.

Funções de ordem superior, parte importante da programação funcional, são comuns no Python. As funções embutidas sorted, min e max, além de functools.partial, são exemplos de funções de ordem superior muito usadas na linguagem. O uso de map, filter e reduce já não é tão frequente como costumava ser, graças às comprehensões de lista (e estruturas similares, como as expressões geradoras) e à adição de funções embutidas de redução como sum, all e any.

Desde o Python 3.6, existem nove sabores de invocáveis, como funções simples criadas com lambda ou instâncias de classes que implementam __call__. Geradoras e corrotinas também são invocáveis,

mas seu comportamento é muito diferente dos outros invocáveis. Todos os invocáveis podem ser detectados pela função embutida `callable()`. Invocáveis oferecem uma rica sintaxe para declaração de parâmetros formais, incluindo parâmetros nomeados, parâmetros somente posicionais e anotações.

Por fim, vimos algumas funções do módulo `operator` e `functools.partial`, que facilitam a programação funcional, reduzindo a necessidade de lidar com a sintaxe de `lambda`.

7.10. Leitura complementar

Nos próximos capítulos, continuaremos nossa jornada pela programação com objetos função. O Capítulo 8 é dedicado às dicas de tipo nos parâmetros de função e nos valores devolvidos por elas. O [ch_closure_decorator] mergulha nos decoradores de função—um tipo especial de função de ordem superior—e no mecanismo de clausura (*closure*) que os faz funcionar. O [ch_design_patterns] mostra como as funções de primeira classe podem simplificar alguns padrões clássicos de projetos (*design patterns*) orientados a objetos.

Em *A Referência da Linguagem Python*, a seção "3.2. A hierarquia de tipos padrão" [<https://fpy.li/47>] mostra os novos tipos invocáveis, juntamente com todos os outros tipos embutidos.

O capítulo 7 do *Python Cookbook* [<https://fpy.li/pycook3>] (EN), 3^a ed. (O'Reilly), de David Beazley e Brian K. Jones, é um excelente complemento a esse capítulo, bem como ao [ch_closure_decorator], tratando basicamente dos mesmos conceitos com uma abordagem diferente.

Veja a PEP 3102—Keyword-Only Arguments (*Argumentos somente nomeados*) [<https://fpy.li/pep3102>] (EN) se quiser saber a justificativa e casos de uso desse recurso.

Uma ótima introdução à programação funcional em Python é o "Programação Funcional COMO FAZER" [<https://fpy.li/46>], de A. M. Kuchling. O principal foco daquele texto, entretanto, é o uso de iteradores e geradoras, assunto do [ch_generators].

A questão no StackOverflow, "Python: Why is `functools.partial` necessary?" (*Python: Por que `functools.partial` é necessária?*) [<https://fpy.li/7-12>] (EN), tem uma resposta muito informativa (e engraçada) escrita por Alex Martelli, co-autor do clássico *Python in a Nutshell* (O'Reilly).

Refletindo sobre a pergunta "Seria Python uma linguagem funcional?", criei uma de minhas palestras favoritas, "Beyond Paradigms" ("Para Além dos Paradigmas"), que apresentei na PyCaribbean, na PyBay e na PyConDE. Veja os slides [<https://fpy.li/7-13>] (EN) e o vídeo [<https://fpy.li/7-14>] (EN) da apresentação em Berlim—onde conheci Miroslav Šedivý e Jürgen Gmach, dois dos revisores técnicos desse livro.

Ponto de vista

Python é uma linguagem funcional?

Em algum momento do ano 2000, eu estava participando de uma oficina sobre o framework Zope na Zope Corporation, nos EUA, quando Guido van Rossum entrou na sala (ele não era o instrutor). Na seção de perguntas e respostas que se seguiu, alguém perguntou quais recursos de Python ele tinha trazido de outras linguagens. A resposta de Guido: "Tudo que é bom no

Python foi roubado de outras linguagens."

Shriram Krishnamurthi, professor de Ciência da Computação na Brown University, inicia seu artigo, "Teaching Programming Languages in a Post-Linnaean Age" (*Ensino das Linguagens de Programação em uma Era Pós-Taxonomia-de-Lineu*) [<https://fpy.li/7-15>] (EN), assim:

Os "paradigmas" de linguagens de programação são um legado moribundo e tedioso de uma era passada. Os atuais projetistas de linguagens não tem qualquer respeito por eles, então por que nossos cursos aderem servilmente a tais "paradigmas"?

Nesse artigo, Python é mencionado nominalmente na seguinte passagem:

E como descrever linguagens como Python, Ruby, ou Perl? Seus criadores não tem paciência com as sutilezas dessas nomenclaturas de Lineu; eles pegam emprestados todos os recursos que desejam, criando misturas que desafiam totalmente uma caracterização.

Krishnamurthi argumenta que, ao invés de tentar classificar as linguagens com alguma taxonomia, seria mais útil olhar para elas como agregados de recursos. Suas ideias inspiraram minha palestra "Beyond Paradigms" ("Para Além dos Paradigmas"), mencionada no final da Seção 7.10.

Mesmo se esse não fosse o objetivo de Guido, dotar Python de funções de primeira classe abriu as portas para a programação funcional. Em seu post, "Origins of Python's 'Functional' Features" (*As Origens dos Recursos 'Funcionais' do Python*) [<https://fpy.li/7-1>] (EN), ele conta que `map`, `filter`, e `reduce` foram a primeira motivação para a inclusão do `lambda` ao Python. Todos esses recursos foram adicionados juntos ao Python 1.0 em 1994, por Amrit Prem, de acordo com o Misc/HISTORY [<https://fpy.li/7-17>] (EN) no código-fonte do CPython.

Funções como `map`, `filter` e `reduce` surgiram inicialmente em Lisp, a primeira linguagem funcional. Lisp, entretanto, não limita o que pode ser feito dentro de uma `lambda`, pois tudo em List é uma expressão. Python usa uma sintaxe orientada a instruções (*statement oriented syntax*), na qual as expressões não podem conter instruções, e muitas das estruturas da linguagem são instruções—incluindo `try/catch`, que é o que mais sinto falta quando escrevo uma `lambda`. É o preço a pagar pela sintaxe extremamente legível de Python.^[6] Lisp tem muitas virtudes, mas legibilidade não é uma delas.

Ironicamente, roubar a sintaxe de compreensão de lista de outra linguagem funcional—Haskell—reduziu significativamente a necessidade de usar `map` e `filter`, e também `lambda`.

Além da sintaxe limitada das funções anônimas, o maior obstáculo para uma adoção mais ampla de idiomas de programação funcional no Python é a ausência da eliminação de chamadas de cauda, uma otimização que permite o processamento eficiente de uma função que faz uma chamada recursiva na "cauda" de seu corpo, sem aumentar a pilha de execução a cada recursão. Em outro post, "Tail Recursion Elimination" (*Eliminação de Recursão de*

Cauda) [<https://fpy.li/7-18>] (EN), Guido apresenta várias razões pelas quais tal otimização não é adequada ao Python. O post é uma ótima leitura por seus argumentos técnicos, e mais ainda porque as três primeiras e mais importantes razões dadas são questões de usabilidade. Python não é um prazer de usar, aprender e ensinar por acidente. Guido foi intencional.

Então cá estamos: Python não é, por projeto, uma linguagem funcional—seja lá o quê isso signifique. Python só pega emprestadas algumas boas ideias de linguagens funcionais.

O problema das funções anônimas

Além das restrições sintáticas específicas de Python, funções anônimas tem uma série desvantagem em qualquer linguagem: elas não tem nome.

Estou brincando, mas não muito. Os *stack traces* são mais fáceis de ler quando as funções tem nome. Funções anônimas são um atalho conveniente, nos divertimos programando com elas, mas algumas vezes elas são levadas longe demais—especialmente se a linguagem e o ambiente encorajam o aninhamento profundo de funções anônimas, como faz o Javascript combinado com o Node.js. Ter muitas funções anônimas aninhadas torna a depuração e o tratamento de erros mais difíceis. A programação assíncrona no Python é mais estruturada, talvez pela sintaxe limitada do `lambda` impedir seu abuso e forçar uma abordagem mais explícita. Promessas, futuros e adiados (*deferreds*) são conceitos usados nas APIs assíncronas modernas. Prometo escrever mais sobre programação assíncrona no futuro, mas esse assunto será adiado até o [ch_async].

[1] "Origins of Python's 'Functional' Features" (*As origens dos recursos 'funcionais' de Python*—EN) [<https://fpy.li/7-1>], do blog The History of Python (A História de Python) do próprio Guido.

[2] "Benevolent Dictator For Life." - Ditador Benevolente Vitalício. Veja Guido van Rossum em "Origin of BDFL" (*A Origem do BDFL*) [<https://fpy.li/bdfl>] (EN).

[3] Invocar uma classe normalmente cria uma instância daquela mesma classe, mas outros comportamentos são possíveis, sobrescrevendo o `__new__`. Veremos um exemplo disso na [flexible_new_sec].

[4] Por que criar uma `BingoCage` quando já temos `random.choice`? A função `choice` pode devolver o mesmo item múltiplas vezes, pois o item escolhido não é removido da coleção usada. Invocações de `BingoCage` nunca devolvem um resultado duplicado—desde que a instância tenha sido preenchida com valores únicos.

[5] O código fonte [<https://fpy.li/7-9>] de `functools.py` revela que `functools.partial` é implementada em C e é usada por default. Se ela não estiver disponível, uma implementação em Python puro de `partial` está disponível desde Python 3.4.

[6] Há também o problema da perda de indentação quando colamos trechos de código em fóruns na Web, mas isso é outro assunto.

8. Dicas de tipo em funções

É preciso enfatizar que **Python continuará sendo uma linguagem de tipagem dinâmica, e os autores não tem qualquer intenção de algum dia tornar dicas de tipo obrigatórias, mesmo por mera convenção.**^[1]

— Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa, PEP 484—Type Hints

Dicas de tipo foram a maior mudança na história de Python desde a unificação de tipos e classes [https://fpy.li/descr101] no Python 2.2, lançado em 2001. Entretanto, as dicas de tipo não beneficiam igualmente a todos as pessoas que usam Python. Por isso deverão ser sempre opcionais.

A PEP 484—Type Hints [https://fpy.li/pep484] introduziu a sintaxe e a semântica para declarações explícitas de tipo em argumentos de funções, valores de retorno e variáveis. O objetivo é ajudar ferramentas de desenvolvimento a encontrarem bugs no código-fonte de programas em Python através de análise estática, isto é, sem executar o código através de testes.

Os maiores beneficiários são engenheiros de software profissionais que usam IDEs (*Ambientes de Desenvolvimento Integrados*) e CI (*Integração Contínua*). A análise de custo-benefício que torna as dicas de tipo atrativas para esse grupo não se aplica a todos os usuários de Python.

A base de usuários de Python vai muito além dessa classe de profissionais. Ela inclui cientistas, comerciantes, jornalistas, artistas, inventores, analistas e estudantes de inúmeras áreas, entre outros. Para a maioria dessas pessoas, o custo de aprender dicas de tipo será maior—exceto para a minoria que já conhece outra linguagem com tipos estáticos, subtipos e tipos genéricos. Os benefícios serão menores para muitas pessoas, dada a forma como que elas interagem com Python, o tamanho menor de suas bases de código e de suas equipes—muitas vezes "equipes solo".

A tipagem dinâmica, default de Python, é mais simples e mais expressiva quando programamos para explorar dados e ideias, como é o caso em ciência de dados, computação criativa e para aprender.

Este capítulo se concentra nas dicas de tipo de Python nas assinaturas de função. [ch_more_types] explora as dicas de tipo no contexto de classes e outros recursos do módulo `typing`.

Os tópicos mais importantes aqui são:

- Uma introdução prática à tipagem gradual com Mypy
- As perspectivas complementares da duck typing (*tipagem pato*) e da tipagem nominal
- A revisão das principais categorias de tipos que podem surgir em anotações (cerca de 60% do capítulo)
- Dicas de tipo em parâmetros variádicos (`*args`, `**kwargs`)
- Limitações e desvantagens das dicas de tipo e da tipagem estática.

8.1. Novidades nesse capítulo

Este capítulo é completamente novo. As dicas de tipo apareceram no Python 3.5, após eu ter terminado de escrever a primeira edição de *Python Fluente*.

Dadas as limitações de um sistema de tipagem estática, a melhor ideia da PEP 484 foi propor um *sistema de tipagem gradual*. Vamos começar definindo o que isso significa.

8.2. Sobre tipagem gradual

A PEP 484 introduziu no Python um *sistema de tipagem gradual*. Outras linguagens com sistemas de tipagem gradual são Typescript da Microsoft, Dart (a linguagem do SDK Flutter, criado pelo Google), e Hack (um dialeto de PHP que é compilado para a máquina virtual HHVM do Facebook). O próprio verificador de tipo MyPy começou como uma linguagem: um dialeto de Python de tipagem gradual com seu próprio interpretador. Guido van Rossum convenceu o criador do MyPy, Jukka Lehtosalo, a transformá-lo em uma ferramenta para checar código Python anotado.

Eis uma função com anotações de tipos:

```
def tokenize(s: str) -> list[str]:  
    "Convert a string into a list of tokens."  
    return s.replace('(', ' ( ').replace(')', ' ) ').split()
```

A assinatura informa que a função `tokenize` recebe uma `str` e devolve `list[str]`: uma lista de strings. A utilidade dessa função será explicada no [lis_parser_ex].

Um sistema de tipagem gradual:

É opcional

Por default, o verificador de tipo não deve emitir avisos para código que não tenha dicas de tipo. Em vez disso, o verificador supõe o tipo `Any` quando não consegue determinar o tipo de um objeto. O tipo `Any` é considerado compatível com todos os outros tipos.

Não captura erros de tipagem durante a execução do código

Dicas de tipo são usadas por checadores de tipos, analisadores de código-fonte (*linters*) e IDEs para emitir avisos. Eles não evitam que valores inconsistentes sejam passados para funções ou atribuídos a variáveis durante a execução. Por exemplo, nada impede que alguém chame `tokenize(42)`, apesar da anotação de tipo do argumento `s: str`. A chamada ocorrerá, e teremos um erro de execução no corpo da função.

Não melhora o desempenho

Anotações de tipo fornecem dados que poderiam, em tese, permitir otimizações do bytecode gerado. Mas, até julho de 2021, tais otimizações não ocorrem em nenhum ambiente Python que eu conheça.^[2]

O melhor aspecto de usabilidade da tipagem gradual é que as anotações são sempre opcionais.

Nos sistemas de tipagem estáticos, a maioria das restrições de tipo são fáceis de expressar, muitas

são desajeitadas, muitas são difíceis e algumas são impossíveis. Por exemplo, em julho de 2021, tipos recursivos não tinham suporte—veja as questões #182, Define a JSON type [<https://fpy.li/8-2>] (EN) #731, Support recursive types [<https://fpy.li/8-3>] (EN) do MyPy.

É perfeitamente possível que você escreva um ótimo programa Python, que consiga passar por uma boa cobertura de testes, mas ainda assim não consiga acrescentar dicas de tipo que satisfaçam um verificador de tipagem. Não tem problema; esqueça as dicas de tipo problemáticas e entregue o programa!

Dicas de tipo são opcionais em todos os níveis: você pode criar ou usar pacotes inteiros sem dicas de tipo, pode silenciar o verificador ao importar algum módulo sem dicas de tipo, e você também pode colocar comentários especiais, para que o checador de tipos ignore certas linhas do seu código.



Tentar impor uma cobertura de 100% de dicas de tipo irá provavelmente estimular seu uso de forma impensada, apenas para satisfazer essa métrica. Isso também vai impedir equipes de aproveitarem da melhor forma possível o potencial e a flexibilidade de Python. Código sem dicas de tipo deve ser aceito sem objeções quando anotações tornam o uso de uma API menos amigável ou quando complicam demais seu desenvolvimento.

8.3. Tipagem gradual na prática

Vamos ver como a tipagem gradual funciona na prática, começando com uma função simples e acrescentando gradativamente a ela dicas de tipo, guiados pelo Mypy.



Há muitos checadores de tipos para Python compatíveis com a PEP 484, incluindo o `pytype` [<https://fpy.li/8-4>] do Google, o `Pyright` [<https://fpy.li/8-5>] da Microsoft, o `Pyre` [<https://fpy.li/8-6>] do Facebook — além de verificadores incluídos em IDEs como o PyCharm. Eu escolhi usar o Mypy [<https://fpy.li/mypy>] nos exemplos por ele ser o mais conhecido. Entretanto, algum daqueles outros pode ser mais adequado para alguns projetos ou equipes. O `Pytype`, por exemplo, foi projetado para lidar com bases de código sem nenhuma dica de tipo e ainda assim gerar recomendações úteis. Ele é mais tolerante que o MyPy, e consegue também gerar anotações para o seu código.

Vamos anotar uma função `show_count`, que retorna uma string com um número e uma palavra no singular ou no plural, dependendo do número:

```
>>> show_count(99, 'bird')
'99 birds'
>>> show_count(1, 'bird')
'1 bird'
>>> show_count(0, 'bird')
'no birds'
```

Exemplo 122 mostra o código-fonte de `show_count`, sem anotações.

Exemplo 122. show_count de messages.py sem dicas de tipo.

```
def show_count(count, word):
    if count == 1:
        return f'1 {word}'
    count_str = str(count) if count else 'no'
    return f'{count_str} {word}s'
```

8.3.1. Usando o Mypy

Para começar a checagem de tipos, rodamos o comando `mypy` passando o módulo `messages.py` como parâmetro:

```
…/no_hints/ $ pip install mypy
[muitas mensagens omitidas...]
…/no_hints/ $ mypy messages.py
Success: no issues found in 1 source file
```

Na configuração default, o Mypy não encontra nenhum problema com o Exemplo 122.



Durante a revisão deste capítulo estou usando Mypy 0.910, a versão mais recente no momento (em julho de 2021). A "Introduction" [https://fpy.li/8-7] (EN) do Mypy adverte que ele "é oficialmente software beta. Mudanças ocasionais irão quebrar a compatibilidade com versões mais antigas." O Mypy está gerando pelo menos um relatório diferente daquele que recebi quando escrevi o capítulo, em abril de 2020. E quando você estiver lendo essas linhas, talvez os resultados também sejam diferentes daqueles mostrados aqui.

Se a assinatura de uma função não tem anotações, Mypy a ignora por default—a menos que seja configurado de outra forma.

O Exemplo 123 também inclui testes de unidade do `pytest`. Este é o código de `messages_test.py`:

Exemplo 123. messages_test.py sem dicas de tipo.

```
from pytest import mark

from messages import show_count

@mark.parametrize('qty, expected', [
    (1, '1 part'),
    (2, '2 parts'),
])
def test_show_count(qty, expected):
    got = show_count(qty, 'part')
    assert got == expected

def test_show_count_zero():
```

```
got = show_count(0, 'part')
assert got == 'no parts'
```

Agora vamos acrescentar dicas de tipo, guiados pelo Mypy.

8.3.2. Tornando o Mypy mais rigoroso

A opção de linha de comando `--disallow-untyped-defs` faz o Mypy apontar todas as definições de funções que não tenham dicas de tipo para todos os argumentos e para o valor de retorno.

Usando `--disallow-untyped-defs` com o arquivo de teste produz três erros e uma observação:

```
…/no_hints/ $ mypy --disallow-untyped-defs messages_test.py
messages.py:14: error: Function is missing a type annotation
messages_test.py:10: error: Function is missing a type annotation
messages_test.py:15: error: Function is missing a return type annotation
messages_test.py:15: note: Use "-> None" if function does not return a value
Found 3 errors in 2 files (checked 1 source file)
```

Nas primeiras etapas da tipagem gradual, prefiro usar outra opção:

`--disallow-incomplete-defs`

Com ela, o Mypy não me dá nenhuma nova informação num primeiro momento:

```
…/no_hints/ $ mypy --disallow-incomplete-defs messages_test.py
Success: no issues found in 1 source file
```

Agora acrescento apenas o tipo do retorno a `show_count` em `messages.py`:

```
def show_count(count, word) -> str:
```

Isso é suficiente para fazer o Mypy olhar para o código. Usando a mesma linha de comando anterior para verificar `messages_test.py` fará o Mypy examinar novamente o `messages.py`:

```
…/no_hints/ $ mypy --disallow-incomplete-defs messages_test.py
messages.py:14: error: Function is missing a type annotation
for one or more arguments
Found 1 error in 1 file (checked 1 source file)
```

Agora posso gradualmente acrescentar dicas de tipo, função por função, sem receber avisos sobre as funções onde ainda não adicionei anotações. Essa é uma assinatura completamente anotada que satisfaz o Mypy:

```
def show_count(count: int, word: str) -> str:
```

Em vez de digitar opções de linha de comando como `--disallow-incomplete-defs`, você pode salvar sua configuração favorita da forma descrita na página Mypy configuration file [<https://fpy.li/8-8>] (EN) na documentação do Mypy. Você pode incluir configurações globais e configurações específicas para cada módulo. Aqui está um `mypy.ini` simples, para servir de base:



```
[mypy]
python_version = 3.9
warn_unused_configs = True
disallow_incomplete_defs = True
```

8.3.3. Um valor default para um argumento

A função `show_count` no Exemplo 122 só funciona com substantivos regulares. Se o plural não pode ser composto acrescentando um '`s`', devemos deixar o usuário fornecer a forma plural, assim:

```
>>> show_count(3, 'mouse', 'mice')
'3 mice'
```

Vamos experimentar um pouco de "desenvolvimento orientado a tipos." Primeiro acrescento um teste usando aquele terceiro argumento. Não esqueça de adicionar a dica do tipo de retorno à função de teste, senão o Mypy não vai inspecioná-la.

```
def test_irregular() -> None:
    got = show_count(2, 'child', 'children')
    assert got == '2 children'
```

O Mypy detecta o erro:

```
.../hints_2/ $ mypy messages_test.py
messages_test.py:22: error: Too many arguments for "show_count"
Found 1 error in 1 file (checked 1 source file)
```

Então edito `show_count`, acrescentando o argumento opcional `plural` no Exemplo 124.

Exemplo 124. showcount de hints_2/messages.py com um argumento opcional

```
def show_count(count: int, singular: str, plural: str = '') -> str:
    if count == 1:
        return f'{count} {singular}'
    count_str = str(count) if count else 'no'
    if not plural:
```

```
plural = singular + 's'  
return f'{count_str} {plural}'
```

E agora o Mypy reporta "Success."

Aqui está um erro de digitação que Python não reconhece. Você consegue encontrá-lo?

```
def hex2rgb(color=str) -> tuple[int, int, int]:
```

O relatório de erros do Mypy não é muito útil:



```
colors.py:24: error: Function is missing a type  
annotation for one or more arguments
```

A dica de tipo para o argumento `color` deveria ser `color: str`. Eu escrevi `color=str`, que não é uma anotação: ele determina que o valor default de `color` é `str`.

Pela minha experiência, esse é um erro comum e fácil de passar desapercebido, especialmente em dicas de tipo complexas.

Os seguintes detalhes são considerados um bom estilo para dicas de tipo:

- Sem espaço entre o nome do parâmetro e o `:`; um espaço após o `:`
- Espaços dos dois lados do `=` que precede um valor default de parâmetro

Por outro lado, a PEP 8 diz que não deve haver espaço em torno de `=` se não há nenhuma dica de tipo para aquele parâmetro específico.

Estilo de Código: use flake8 e blue

Em vez de decorar essas regrinhas bobas, use ferramentas como *flake8* [<https://fpy.li/8-9>] e *blue* [<https://fpy.li/8-10>]. O *flake8* informa sobre o estilo do código e várias outras questões, enquanto o *blue* reescreve o código-fonte com base na (maioria) das regras prescritas pela ferramenta de formatação de código *black* [<https://fpy.li/8-11>].

Se o objetivo é impor um estilo de programação "padrão", *blue* é melhor que *black*, porque segue o estilo próprio de Python, de usar aspas simples por default e aspas duplas como alternativa.

```
>>> "I prefer single quotes"  
'I prefer single quotes'
```

No CPython, a preferência por aspas simples está incorporada no `repr()`, entre outros lugares. O módulo `doctest` [<https://fpy.li/doctest>] depende do `repr()` usar aspas simples por default.

Um dos autores do *blue* é Barry Warsaw [https://fpy.li/8-12], co-autor da PEP 8, core developer de Python desde 1994 e membro de Python's Steering Council desde 2019. Daí estamos em ótima companhia quando escolhemos usar aspas simples.

Se você precisar mesmo usar o *black*, use a opção `black -S`. Isso deixará suas aspas intocadas.

8.3.4. Usando None como default

No Exemplo 124, o parâmetro `plural` está anotado como `str`, e o valor default é ''. Assim não há conflito de tipo.

Eu gosto dessa solução, mas em outros contextos `None` é um default melhor. Se o parâmetro opcional requer um tipo mutável, então `None` é o único default sensato, como vimos na Seção 6.5.1.

Com `None` como default para o parâmetro `plural`, a assinatura ficaria assim:

```
from typing import Optional

def show_count(count: int, singular: str, plural: Optional[str] = None) -> str:
```

Vamos destrarchar essa linha:

- `Optional[str]` significa que `plural` pode ser uma `str` ou `None`.
- É obrigatório fornecer explicitamente o valor `default = None`.

Se você não atribuir um valor default a `plural`, o runtime de Python vai tratar o parâmetro como obrigatório. Lembre-se: durante a execução do programa, as dicas de tipo são ignoradas.

Veja que é preciso importar `Optional` do módulo `typing`. Quando importamos tipos, é uma boa prática usar a sintaxe `from typing import X`, para reduzir o tamanho das assinaturas das funções.



`Optional` não é um bom nome, pois aquela anotação não torna o argumento opcional. O que o torna opcional é a atribuição de um valor default ao parâmetro. `Optional[str]` significa apenas: o tipo desse parâmetro pode ser `str` ou `NoneType`. Nas linguagens Haskell e Elm, um tipo parecido se chama `Maybe`.

Agora que tivemos um primeiro contato concreto com a tipagem gradual, vamos examinar o que o conceito de *tipo* significa na prática.

8.4. Tipos são definidos pelas operações possíveis

Há muitas definições do conceito de tipo na literatura. Aqui vamos assumir que tipo é um conjunto de valores e um conjunto de funções que podem ser aplicadas àqueles valores.

— PEP 483—A Teoria das Dicas de Tipo

Na prática, é mais útil considerar o conjunto de operações possíveis como a característica que define um tipo.^[3]

Por exemplo, pensando nas operações possíveis, quais são os tipos válidos para `x` na função a seguir?

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

O tipo do parâmetro `x` pode ser numérico (`int`, `complex`, `Fraction`, `numpy.uint32`, etc.), mas também pode ser uma sequência (`str`, `tuple`, `list`, `array`), uma `numpy.array` N-dimensional, ou qualquer outro tipo que implemente ou herde um método `__mul__` que aceite um inteiro como argumento.

Entretanto, considere a anotação `double` abaixo. Ignore por enquanto a ausência do tipo do retorno, vamos nos concentrar no tipo do parâmetro:

```
from collections import abc

def double(x: abc.Sequence):
    return x * 2
```

Um verificador de tipo irá rejeitar esse código. Se você informar ao Mypy que `x` é do tipo `abc.Sequence`, ele vai marcar `x * 2` como erro, pois a Sequence ABC [https://fpy.li/8-13] não implementa ou herda o método `__mul__`. Durante a execução, o código vai funcionar com sequências concretas como `str`, `tuple`, `list`, `array`, etc., bem como com números, pois durante a execução as dicas de tipo são ignoradas. Mas o verificador de tipo se preocupa apenas com o que estiver explicitamente declarado, e `abc.Sequence` não suporta `__mul__`.

Por essa razão o título dessa seção é "Tipos São Definidos pelas Operações Possíveis." O runtime de Python aceita qualquer objeto como argumento `x` nas duas versões da função `double`. O cálculo de `x * 2` pode funcionar, ou pode causar um `TypeError`, se a operação não for suportada por `x`. Por outro lado, Mypy vai marcar `x * 2` como um erro quando analisar o código-fonte anotado de `double`, pois é uma operação não suportada pelo tipo declarado `x: abc.Sequence`.

Em um sistema de tipagem gradual, acontece uma interação entre duas perspectivas diferentes de tipo:

Duck typing ("tipagem pato")

A perspectiva adotada pelo Smalltalk—a primeira linguagem orientada a objetos—bem como em Python, JavaScript, e Ruby. Objetos tem tipo, mas variáveis (incluindo parâmetros) não tem. Na prática, não importa qual o tipo declarado de um objeto, importam apenas as operações que ele efetivamente suporta. Se eu posso invocar `birdie.quack()` então, nesse contexto, `birdie` é um pato. Por definição, duck typing só é aplicada durante a execução, quando se tenta aplicar operações sobre os objetos. Isso é mais flexível que a *tipagem nominal*, ao preço de permitir mais erros durante a execução.^[4]

Tipagem nominal

É a perspectiva adotada em C++, Java, e C#, e suportada em Python anotado. Objetos e variáveis tem tipos. Mas objetos só existem durante a execução, e o verificador de tipo só se importa com o código-fonte, onde as variáveis (incluindo parâmetros de função) tem anotações com dicas de tipo. Se Duck é uma subclasse de Bird, você pode atribuir uma instância de Duck a um parâmetro anotado como `birdie: Bird`. Mas no corpo da função, o verificador considera a chamada `birdie.quack()` ilegal, pois `birdie` é nominalmente um `Bird`, e aquela classe não fornece o método `.quack()`. Não interessa que o argumento real, durante a execução, é um `Duck`, porque a tipagem nominal é verificada estaticamente. O verificador de tipo não executa qualquer pedaço do programa, ele apenas analisa o código-fonte. Isso é mais rígido que *duck typing*, com a vantagem de capturar alguns bugs durante o desenvolvimento, ou mesmo em tempo real, enquanto o código está sendo digitado em um IDE.

O Exemplo 125 é um exemplo bobo que contrapõe *duck typing* e tipagem nominal, bem como checagem de tipos estática e comportamento durante a execução.^[5]

Exemplo 125. *birds.py*

```
class Bird:  
    pass  
  
class Duck(Bird): ①  
    def quack(self):  
        print('Quack!')  
  
def alert(birdie): ②  
    birdie.quack()  
  
def alert_duck(birdie: Duck) -> None: ③  
    birdie.quack()  
  
def alert_bird(birdie: Bird) -> None: ④  
    birdie.quack()
```

① Duck é uma subclasse de Bird.

② `alert` não tem dicas de tipo, então o verificador a ignora.

③ `alert_duck` aceita um argumento do tipo Duck.

④ `alert_bird` aceita um argumento do tipo Bird.

Verificando *birds.py* com Mypy, encontramos um problema:

```
…/birds/ $ mypy birds.py  
birds.py:16: error: "Bird" has no attribute "quack"  
Found 1 error in 1 file (checked 1 source file)
```

Só de analisar o código fonte, MyPy percebe que `alert_bird` é problemático: a dica de tipo declara o parâmetro `birdie` como do tipo `Bird`, mas o corpo da função chama `birdie.quack()` — e a classe `Bird`

não tem esse método.

Agora vamos tentar usar o módulo `birds` em `daffy.py` no Exemplo 126.

Exemplo 126. daffy.py

```
from birds import *

daffy = Duck()
alert(daffy)      ①
alert_duck(daffy) ②
alert_bird(daffy) ③
```

① Chamada válida, pois `alert` não tem dicas de tipo.

② Chamada válida, pois `alert_duck` recebe um argumento do tipo `Duck` e `daffy` é um `Duck`.

③ Chamada válida, pois `alert_bird` recebe um argumento do tipo `Bird`, e `daffy` também é um `Bird` — a superclasse de `Duck`.

Mypy reporta o mesmo erro em `daffy.py`, sobre a chamada a `quack` na função `alert_bird` definida em `birds.py`:

```
…/birds/ $ mypy daffy.py
birds.py:16: error: "Bird" has no attribute "quack"
Found 1 error in 1 file (checked 1 source file)
```

Mas o Python não vê qualquer problema com `daffy.py` em si: as três chamadas de função estão OK.

Agora, rodando `daffy.py`, o resultado é o seguinte:

```
…/birds/ $ python3 daffy.py
Quack!
Quack!
Quack!
```

Funciona perfeitamente! Viva o duck typing!

Durante a execução do programa, Python não se importa com os tipos declarados. Ele usa apenas duck typing. O Mypy apontou um erro em `alert_bird`, mas a chamada da função com `daffy` funciona corretamente quando executada. À primeira vista isso pode surpreender muitos pythonistas: um verificador de tipo estático muitas vezes encontra erros em código que sabemos que vai funcionar quanto executado.

Entretanto, se daqui a alguns meses você for encarregado de estender o exemplo bobo do pássaro, você agradecerá ao Mypy. Observe esse módulo `woody.py` module, que também usa `birds`, no Exemplo 127.

Exemplo 127. *woody.py*

```
from birds import *

woody = Bird()
alert(woody)
alert_duck(woody)
alert_bird(woody)
```

O Mypy encontra dois erros ao verificar *woody.py*:

```
…/birds/ $ mypy woody.py
birds.py:16: error: "Bird" has no attribute "quack"
woody.py:5: error: Argument 1 to "alert_duck" has incompatible type "Bird";
expected "Duck"
Found 2 errors in 2 files (checked 1 source file)
```

O primeiro erro é em *birds.py*: a chamada a *birdie.quack()* em *alert_bird*, que já vimos antes. O segundo erro é em *woody.py*: *woody* é uma instância de *Bird*, então a chamada *alert_duck(woody)* é inválida, pois aquela função exige um *Duck*. Todo *Duck* é um *Bird*, mas nem todo *Bird* é um *Duck*.

Durante a execução, nenhuma das duas chamadas em *woody.py* funcionariam. A sucessão de falhas é melhor ilustrada em uma sessão no console, através das mensagens de erro, no Exemplo 128.

Exemplo 128. Erros durante a execução e como o Mypy poderia ter ajudado

```
>>> from birds import *
>>> woody = Bird()
>>> alert(woody) ①
Traceback (most recent call last):
...
AttributeError: 'Bird' object has no attribute 'quack'
>>>
>>> alert_duck(woody) ②
Traceback (most recent call last):
...
AttributeError: 'Bird' object has no attribute 'quack'
>>>
>>> alert_bird(woody) ③
Traceback (most recent call last):
...
AttributeError: 'Bird' object has no attribute 'quack'
```

① O Mypy não tinha como detectar esse erro, pois não há dicas de tipo em *alert*.

② O Mypy avisou do problema: Argument 1 to "alert_duck" has incompatible type "Bird"; expected "Duck" (*Argumento 1 para alert_duck é do tipo incompatível "Bird"; argumento esperado era "Duck"*)

- ③ O Mypy está avisando desde o Exemplo 125 que o corpo da função `alert_bird` está errado: "Bird" has no attribute "quack" (*Bird não tem um atributo "quack"*)

Este pequeno experimento mostra que o duck typing é mais fácil para o iniciante e mais flexível, porém permite que operações não suportadas causem erros durante a execução. A tipagem nominal detecta os erros antes da execução, mas algumas vezes rejeita código que seria executado sem erros—como a chamada a `alert_bird(daffy)` no Exemplo 126.

Mesmo que funcione algumas vezes, o nome da função `alert_bird` está incorreto: seu código exige um objeto que suporte o método `.quack()`, que não existe em `Bird`.

Nesse exemplo bobo, as funções tem uma linha apenas. Mas na vida real elas poderiam ser mais longas, e poderiam passar o argumento `birdie` para outras funções, e a origem daquele argumento poderia estar a muitas chamadas de função de distância, tornando difícil localizar a causa do erro durante a execução. O checador de tipos impede que muitos erros como esse aconteçam durante a execução de um programa.



O valor das dicas de tipo é questionável nos pequenos exemplos que cabem em um livro. Os benefícios crescem com o tamanho da base de código. É por essa razão que empresas com milhões de linhas de código em Python—como Dropbox, Google e Facebook—investiram em equipes e ferramentas para promover a adoção global de dicas de tipo internamente, e hoje tem partes significativas e crescentes de sua base de código com anotações de tipo sendo verificadas em suas *pipelines* de integração contínua.

Nessa seção exploramos as relações de tipos e operações no duck typing e na tipagem nominal, começando com a função simples `double()` — que deixamos sem dicas de tipo. Agora vamos dar uma olhada nos tipos mais importantes ao anotar funções.

Vamos ver um bom modo de adicionar dicas de tipo a `double()` quando examinarmos Seção 8.5.10. Mas antes disso, há tipos mais importantes para conhecer.

8.5. Tipos próprios para anotações

Quase todos os tipos em Python podem ser usados em dicas de tipo, mas há restrições e recomendações. Além disso, o módulo `typing` introduziu anotações especiais com uma semântica às vezes surpreendente.

Essa seção trata dos os principais tipos que você pode usar em anotações:

- `typing.Any`
- Tipos e classes simples
- `typing.Optional` e `typing.Union`
- Coleções genéricas, incluindo tuplas e mapeamentos
- Classes base abstratas
- Iteradores genéricos

- Genéricos parametrizados e TypeVar
- typing.Protocols — crucial para *duck typing* estático
- typing.Callable
- typing.NoReturn — um bom modo de encerrar essa lista.

Vamos falar de um de cada vez, começando por um tipo que é estranho, aparentemente inútil, mas de uma importância fundamental.

8.5.1. O tipo Any

A pedra fundamental de qualquer sistema de tipagem gradual é o tipo Any, também conhecido como o *tipo dinâmico*. Quando um verificador de tipo vê um função sem tipo como esta:

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

ele supõe isto:

```
def double(x: Any) -> Any:
    return x * 2
```

Isso significa que o argumento x e o valor de retorno podem ser de qualquer tipo, inclusive de tipos diferentes. Assume-se que Any pode suportar qualquer operação possível.

Compare Any com object. Considere essa assinatura:

```
def double(x: object) -> object:
```

Essa função também aceita argumentos de todos os tipos, porque todos os tipos são *subtipo-de* object.

Entretanto, um verificador de tipo vai rejeitar essa função:

```
def double(x: object) -> object:
    return x * 2
```

O problema é que object não suporta a operação `__mul__`. Veja o que diz o Mypy:

```
.../birds/ $ mypy double_object.py
double_object.py:2: error: Unsupported operand types for * ("object" and "int")
Found 1 error in 1 file (checked 1 source file)
```

Tipos mais gerais tem interfaces mais restritas, isto é, eles suportam menos operações. A classe

`object` implementa menos operações que `abc.Sequence`, que implementa menos operações que `abc.MutableSequence`, que por sua vez implementa menos operações que `list`.

Mas `Any` é um tipo mágico que reside tanto no topo quanto na base da hierarquia de tipos. Ele é simultaneamente o tipo mais geral—então um argumento `n`: `Any` aceita valores de qualquer tipo—e o tipo mais especializado, suportando assim todas as operações possíveis. Pelo menos é assim que o verificador de tipo entende `Any`.

Claro, nenhum tipo consegue suportar qualquer operação possível, então usar `Any` impede o verificador de tipo de cumprir sua missão: detectar operações que podem falhar antes que seu programa levante uma exceção durante a execução.

8.5.1.1. Subtipo-de ou consistente-com

Sistemas tradicionais de tipagem nominal orientados a objetos se baseiam na relação *subtipo-de*. Dada uma classe `T1` e uma subclasse `T2`, então `T2` é *subtipo-de* `T1`.

Observe este código:

```
class T1:  
    ...  
  
class T2(T1):  
    ...  
  
def f1(p: T1) -> None:  
    ...  
  
o2 = T2()  
  
f1(o2) # OK
```

A chamada `f1(o2)` é uma aplicação do Princípio de Substituição de Liskov (*Liskov Substitution Principle*, LSP).

Barbara Liskov^[6] definiu é *subtipo-de* em termos das operações suportadas. Se um objeto do tipo `T2` substitui um objeto do tipo `T1` e o programa continua se comportando de forma correta, então `T2` é *subtipo-de* `T1`.

Seguindo com o código visto acima, essa parte mostra uma violação do LSP:

```
def f2(p: T2) -> None:  
    ...  
  
o1 = T1()  
  
f2(o1) # type error
```

Do ponto de vista das operações suportadas, faz todo sentido: como uma subclasse, T2 herda e precisa suportar todas as operações suportadas por T1. Então uma instância de T2 pode ser usada em qualquer lugar onde se espera uma instância de T1. Mas o contrário não é necessariamente verdadeiro: T2 pode implementar métodos adicionais, então uma instância de T1 não pode ser usada onde se espera uma instância de T2. Este foco nas operações suportadas se reflete no nome _behavioral subtyping (subtipagem comportamental) [<https://fpy.li/8-15>] (EN), também usado para se referir ao LSP.

Em um sistema de tipagem gradual há outra relação, *consistente-com* (*consistent-with*), que se aplica sempre que *subtipo-de* puder ser aplicado, com regras especiais para o tipo Any.

As regras para *consistente-com* são:

1. Dados T1 e um subtipo T2, então T2 é *consistente-com* T1 (substituição de Liskov).
2. Todo tipo é *consistente-com* Any: você pode passar objetos de qualquer tipo em um argumento declarado como Any.
3. Any é *consistente-com* todos os tipos: você sempre pode passar um objeto de tipo Any onde um argumento de outro tipo for esperado.

Considerando as definições anteriores dos objetos o1 e o2, aqui estão alguns exemplos de código válido, ilustrando as regras #2 e #3:

```
def f3(p: Any) -> None:  
    ...  
  
o0 = object()  
o1 = T1()  
o2 = T2()  
  
f3(o0) #  
f3(o1) # tudo certo: regra #2  
f3(o2) #  
  
def f4(): # tipo implícito de retorno: 'Any'  
    ...  
  
o4 = f4() # tipo inferido: 'Any'  
  
f1(o4) #  
f2(o4) # tudo certo: regra #3  
f3(o4) #
```

Todo sistema de tipagem gradual precisa de um tipo coringa como Any



O verbo "inferir" é um sinônimo bonito para "adivinar", quando usado no contexto da análise de tipos. Verificadores de tipo modernos, em Python e outras linguagens, não precisam de anotações de tipo em todo lugar porque conseguem inferir o tipo de muitas expressões. Por exemplo, se eu escrever `x = len(s) * 10`, o

verificador não precisa de uma declaração local explícita para saber que `x` é um `int`, desde que a ferramenta consiga acessar dicas de tipo para `len` em algum lugar.

Agora podemos explorar o restante dos tipos usados em anotações.

8.5.2. Tipos simples e classes

Tipos simples como `int`, `float`, `str`, e `bytes` podem ser usados diretamente em dicas de tipo. Classes concretas da biblioteca padrão, de pacotes externos ou definidas pelo usuário (ex. `FrenchDeck`, `Vector2d`, e `Duck`) também podem ser usadas em dicas de tipo.

Classes base abstratas também são úteis aqui. Voltaremos a elas quando formos estudar os tipos coleção, na Seção 8.5.7.

Para classes, *consistente-com* é definido como *subtipo-de*: uma subclasse é *consistente-com* todas as suas superclasses.

Entretanto, "a praticidade se sobrepõe à pureza", então há uma exceção importante, discutida em seguida.

int é *consistente-com* *complex*

Não há nenhuma relação nominal de subtipo entre os tipo nativos `int`, `float` e `complex`: eles são subclasses diretas de `object`. Mas a PEP 484 decretou [<https://fpy.li/cardxvi>] que `int` é *consistente-com* `float`, e `float` é *consistente-com* `complex`. Na prática, faz sentido: `int` implementa todas as operações que `float` implementa, e `int` implementa operações adicionais para os operadores binários `&`, `|`, `<<`, etc. O resultado final é o seguinte: `int` é *consistente-com* `complex`. Para `i = 3`, `i.real` é `3` e `i.imag` é `0`.

8.5.3. Os tipos Optional e Union

Vimos o tipo especial `Optional` em Seção 8.3.4. Ele resolve o problema de ter `None` como default, como no exemplo daquela seção:

```
from typing import Optional

def show_count(count: int, singular: str, plural: Optional[str] = None) -> str:
```

A sintaxe `Optional[str]` é na verdade um atalho para `Union[str, None]`, que significa que o tipo de `plural` pode ser `str` ou `None`.

Uma sintaxe melhor para Optional e Union em Python 3.10

Desde Python 3.10 é possível escrever `str | bytes` em vez de `Union[str, bytes]`. É menos digitação, e não precisamos importar `Optional` ou `Union` de `typing`. Compare a sintaxe antiga com a nova para a dica de tipo do parâmetro `plural` em `show_count`:

```
plural: Optional[str] = None      # até Python 3.9
plural: str | None = None         # a partir do Python 3.10
```

O operador `|` também funciona com `isinstance` e `issubclass` para declarar o segundo argumento: `isinstance(x, int | str)`. Para saber mais, veja PEP 604—Complementary syntax for Union[] [<https://fpy.li/pep604>] (EN).

A assinatura da função nativa `ord` é um exemplo simples de `Union` - ela aceita `str` or `bytes`, e retorna um `int`:^[7]

```
def ord(c: Union[str, bytes]) -> int: ...
```

Aqui está um exemplo de uma função que aceita uma `str`, mas pode retornar uma `str` ou um `float`:

```
from typing import Union

def parse_token(token: str) -> Union[str, float]:
    try:
        return float(token)
    except ValueError:
        return token
```

Se possível, evite criar funções com o tipo de retorno `Union`, pois isso exige um esforço extra do usuário: para saber o que fazer com o valor recebido da função pode ser necessário verificar o tipo daquele valor durante a execução. Mas a `parse_token` no código acima é um caso de uso razoável no contexto de interpretador de expressões simples.



Na Seção 4.10, vimos funções que aceitam tanto `str` quanto `bytes` como argumento, mas retornam uma `str` se o argumento for `str` ou `bytes`, se o argumento for `bytes`. Nesses casos, o tipo de retorno é determinado pelo tipo da entrada, então `Union` não é uma solução precisa. Para anotar tais funções corretamente, precisamos usar uma variável de tipo—apresentada em Seção 8.5.9—ou sobre carga (*overloading*), que veremos na [overload_sec].

`Union[]` exige pelo menos dois tipos. Tipos `Union` aninhados tem o mesmo efeito que uma `Union` "achatada". Então esta dica de tipo:

```
Union[A, B, Union[C, D, E]]
```

Tem o mesmo efeito desta:

```
Union[A, B, C, D, E]
```

`Union` é mais útil com tipos que não sejam consistentes entre si. Por exemplo: `Union[int, float]` é redundante, pois `int` é *consistente-com* `float`. Se você usar apenas `float` para anotar o parâmetro, ele vai também aceitar valores `int`.

8.5.4. Coleções genéricas

A maioria das coleções em Python são heterogêneas.

Por exemplo, você pode inserir qualquer combinação de tipos diferentes em uma `list`. Entretanto, na prática isso não é muito útil: se você colocar objetos em uma coleção, você certamente vai querer executar alguma operação com eles mais tarde, e normalmente isso significa que eles precisam compartilhar pelo menos um método comum.^[8]

Tipos genéricos podem ser declarados com parâmetros de tipo, para especificar o tipo de item com o qual eles conseguem trabalhar.

Por exemplo, uma `list` pode ser parametrizada para restringir o tipo de elemento ali contido, como se pode ver no Exemplo 129.

Exemplo 129. tokenize com dicas de tipo para Python ≥ 3.9

```
def tokenize(text: str) -> list[str]:  
    return text.upper().split()
```

Em Python ≥ 3.9 , isso significa que `tokenize` retorna uma `list` onde todos os elementos são do tipo `str`.

As anotações `stuff: list` e `stuff: list[Any]` significam a mesma coisa: `stuff` é uma lista de objetos de qualquer tipo.



Se estiver usando Python 3.8 ou anterior, o conceito é o mesmo, mas você precisa de mais código para funcionar, como explicado em [Suporte a tipos de coleção descontinuados](#).

A PEP 585—Type Hinting Generics In Standard Collections [<https://fpy.li/8-16>] (EN) lista as coleções da biblioteca padrão que aceitam dicas de tipo genéricas. A lista a seguir mostra apenas as coleções que usam a forma mais simples de dica de tipo genérica, `container[item]`:

<code>list</code>	<code>collections.deque</code>	<code>abc.Sequence</code>	<code>abc.MutableSequence</code>
<code>set</code>	<code>abc.Container</code>	<code>abc.Set</code>	<code>abc.MutableSet</code>
<code>frozenset</code>	<code>abc.Collection</code>		

Os tipos `tuple` e `mapping` aceitam dicas de tipo mais complexas, como veremos em suas respectivas seções.

No Python 3.10, não há uma boa maneira de anotar `array.array`, levando em consideração o argumento `typecode` do construtor, que determina se o array contém inteiros ou floats. Um problema ainda mais complicado é verificar a faixa dos inteiros, para prevenir `OverflowError`.

durante a execução, ao se adicionar novos elementos. Por exemplo, um array com typecode=B só pode receber valores int de 0 a 255. Até Python 3.11, o sistema de tipagem estática de Python não consegue lidar com esse desafio.

Suporte a tipos de coleção descontinuados

(Você pode pular esse box se usa apenas Python 3.9 ou posterior.)

Em Python 3.7 e 3.8, você precisa importar um `__future__` para fazer a notação `[]` funcionar com as coleções nativas, tal como `list`, como ilustrado no Exemplo 130.

Exemplo 130. tokenize com dicas de tipo para Python ≥ 3.7

```
from __future__ import annotations

def tokenize(text: str) -> list[str]:
    return text.upper().split()
```

O `__future__` não funciona com Python 3.6 ou anterior. O Exemplo 131 mostra como anotar `tokenize` de uma forma que funciona com Python ≥ 3.5.

Exemplo 131. tokenize com dicas de tipo para Python ≥ 3.5

```
from typing import List

def tokenize(text: str) -> List[str]:
    return text.upper().split()
```

Para fornecer um suporte inicial a dicas de tipo genéricas, os autores da PEP 484 criaram dúzias de tipos genéricos no módulo `typing`. A Tabela 15 mostra alguns deles. Para a lista completa, consulte a documentação do módulo `typing` [<https://fpy.li/4a>].

Tabela 15. Alguns tipos de coleção e seus equivalentes nas dicas de tipo

Collection	Type hint equivalent
<code>list</code>	<code>typing.List</code>
<code>set</code>	<code>typing.Set</code>
<code>frozenset</code>	<code>typing.FrozenSet</code>
<code>collections.deque</code>	<code>typing.Deque</code>
<code>collections.abc.MutableSequence</code>	<code>typing.MutableSequence</code>
<code>collections.abc.Sequence</code>	<code>typing.Sequence</code>
<code>collections.abc.Set</code>	<code>typing.AbstractSet</code>
<code>collections.abc.MutableSet</code>	<code>typing.MutableSet</code>

A PEP 585—Type Hinting Generics In Standard Collections [<https://fpy.li/pep585>] deu início a um processo de vários anos para melhorar a usabilidade das dicas de tipo genéricas. Podemos resumir esse processo em quatro etapas:

1. Introduzir `from __future__ import annotations` no Python 3.7 para permitir o uso das classes da biblioteca padrão como genéricos com a notação `list[str]`.
2. Tornar aquele comportamento o default a partir de Python 3.9: `list[str]` agora funciona sem que `future` precise ser importado.
3. Descontinuar (*deprecate*) todos os tipos genéricos do módulo `typing`.^[9] Avisos de descontinuação não serão emitidos pelo interpretador Python, porque os checadores de tipos devem sinalizar os tipos descontinuados quando o programa sendo verificado tiver como alvo Python 3.9 ou posterior.
4. Remover aqueles tipos genéricos redundantes na primeira versão de Python lançada cinco anos após Python 3.9. No ritmo atual, esse deverá ser Python 3.14, também conhecido como Python Pi.

Agora vamos ver como anotar tuplas genéricas.

8.5.5. Tipos tuple

Há três maneiras de anotar o tipo `tuple`, dependendo de como você usa a tupla:

- Tupla como registro
- Tupla como registro com campos nomeados
- Tupla como sequência imutável

8.5.5.1. Tuplas como registros

Se você está usando uma `tuple` como um registro, use o tipo `tuple` nativo e declare os tipos dos campos dentro dos `[]`.

Por exemplo, para anotar uma tupla com nome da cidade, população e país: ('Shanghai', 24.28, 'China'), a dica de tipo é `tuple[str, float, str]`

Observe uma função que recebe um par de coordenadas geográficas e retorna uma Geohash [<https://fpy.li/8-18>], usada assim:

```
>>> shanghai = 31.2304, 121.4737
>>> geohash(shanghai)
'wtw3sjq6q'
```

O Exemplo 132 mostra a definição da função `geohash`, usando o pacote `geolib` do PyPI.

Exemplo 132. coordinates.py com a função geohash

```
from geolib import geohash as gh # type: ignore ①
PRECISION = 9
def geohash(lat_lon: tuple[float, float]) -> str: ②
```

```
    return gh.encode(*lat_lon, PRECISION)
```

① Esse comentário evita que o Mypy avise que o pacote geolib não tem dicas de tipo.

② O parâmetro `lat_lon`, anotado como uma tuple com dois campos float.



Com Python < 3.9, importe e use `typing.Tuple` nas dicas de tipo. Este tipo está descontinuado mas permanecerá na biblioteca padrão pelo menos até 2024.

8.5.5.2. Tuplas como registros com campos nomeados

Para anotar uma tupla com muitos campos, ou tipos específicos de tupla que seu código usa com frequência, recomendo fortemente usar `typing.NamedTuple`, como visto no Capítulo 5. O Exemplo 133 mostra uma variante de Exemplo 132 com `NamedTuple`.

Exemplo 133. coordinates_named.py com NamedTuple, Coordinates e a função geohash

```
from typing import NamedTuple

from geolib import geohash as gh # type: ignore

PRECISION = 9

class Coordinate(NamedTuple):
    lat: float
    lon: float

def geohash(lat_lon: Coordinate) -> str:
    return gh.encode(*lat_lon, PRECISION)
```

Como explicado na Seção 5.2, `typing.NamedTuple` é uma factory de subclasses de `tuple`, então `Coordinate` é *consistente-com* `tuple[float, float]`, mas o inverso não é verdadeiro—afinal, `Coordinate` tem métodos extras adicionados por `NamedTuple`, como `.asdict()`, e também poderia ter métodos definidos pelo usuário.

Na prática, isso significa que é seguro (do ponto de vista do tipo do argumento) passar uma instância de `Coordinate` para a função `display`, definida assim:

```
def display(lat_lon: tuple[float, float]) -> str:
    lat, lon = lat_lon
    ns = 'N' if lat >= 0 else 'S'
    ew = 'E' if lon >= 0 else 'W'
    return f'{abs(lat):0.1f}°{ns}, {abs(lon):0.1f}°{ew}'
```

8.5.5.3. Tuplas como sequências imutáveis

Para anotar tuplas de tamanho desconhecido, usadas como listas imutáveis, você precisa especificar um único tipo, seguido de uma vírgula e ... (isto é o símbolo de reticências de Python,

formado por três caracteres ., não o caractere Unicode U+2026—HORIZONTAL ELLIPSIS).

Por exemplo, `tuple[int, ...]` é uma tupla com itens `int`.

As reticências indicam que qualquer número de elementos ≥ 1 é aceitável. Não há como especificar campos de tipos diferentes para tuplas de tamanho arbitrário.

As anotações `stuff: tuple[Any, ...]` e `stuff: tuple` são equivalentes: `stuff` é uma tupla de tamanho desconhecido contendo objetos de qualquer tipo.

Aqui temos um função `columnize`, que transforma uma sequência em uma tabela de colunas e células, na forma de uma lista de tuplas de tamanho desconhecido. É útil para mostrar os itens em colunas, assim:

```
>>> animals = 'drake fawn heron ibex koala lynx tahr xerus yak zapus'.split()
>>> table = columnize(animals)
>>> table
[('drake', 'koala', 'yak'), ('fawn', 'lynx', 'zapus'), ('heron', 'tahr'),
 ('ibex', 'xerus')]
>>> for row in table:
...     print('.join(f'{word:10}' for word in row))
...
drake      koala      yak
fawn       lynx       zapus
heron      tahr
ibex       xerus
```

O Exemplo 134 mostra a implementação de `columnize`. Observe o tipo do retorno:

```
list[tuple[str, ...]]
```

Exemplo 134. `columnize.py` retorna uma lista de tuplas de strings

```
from collections.abc import Sequence

def columnize(
    sequence: Sequence[str], num_columns: int = 0
) -> list[tuple[str, ...]]:
    if num_columns == 0:
        num_columns = round(len(sequence) ** 0.5)
    num_rows, remainder = divmod(len(sequence), num_columns)
    num_rows += bool(remainder)
    return [tuple(sequence[i::num_rows]) for i in range(num_rows)]
```

8.5.6. Mapeamentos genéricos

Tipos de mapeamento genéricos são anotados como `MappingType[KeyType, ValueType]`. O tipo nativo `dict` e os tipos de mapeamento em `collections` e `collections.abc` aceitam essa notação em Python \geq

3.9. Para versões mais antigas, use `typing.Dict` e outros tipos de mapeamento do módulo `typing`, como discutimos em Suporte a tipos de coleção descontinuados.

O Exemplo 135 mostra um uso na prática de uma função que retorna um índice invertido [<https://fpy.li/8-19>] para permitir a busca de caracteres Unicode pelo nome—uma variação do Exemplo 61 mais adequada para código server-side, que estudaremos no [ch_async].

Dado o início e o final dos códigos de caractere Unicode, `name_index` retorna um `dict[str, set[str]]`, que é um índice invertido mapeando cada palavra para um conjunto de caracteres que tem aquela palavra em seus nomes. Por exemplo, após indexar os caracteres ASCII de 32 a 64, aqui estão os conjuntos de caracteres mapeados para as palavras 'SIGN' e 'DIGIT', e a forma de encontrar o caractere chamado 'DIGIT EIGHT':

```
>>> index = name_index(32, 65)
>>> index['SIGN']
{'$', '>', '=', '+', '<', '%', '#'}
>>> index['DIGIT']
{'8', '5', '6', '2', '3', '0', '1', '4', '7', '9'}
>>> index['DIGIT'] & index['EIGHT']
{'8'}
```

O Exemplo 135 mostra o código fonte de `charindex.py` com a função `name_index`. Além de uma dica de tipo `dict[]`, este exemplo tem três outros aspectos que estão aparecendo pela primeira vez no livro.

Exemplo 135. charindex.py

```
import sys
import re
import unicodedata
from collections.abc import Iterator

RE_WORD = re.compile(r'\w+')
STOP_CODE = sys.maxunicode + 1

def tokenize(text: str) -> Iterator[str]: ①
    """return iterable of uppercased words"""
    for match in RE_WORD.finditer(text):
        yield match.group().upper()

def name_index(start: int = 32, end: int = STOP_CODE) -> dict[str, set[str]]:
    index: dict[str, set[str]] = {} ②
    for char in (chr(i) for i in range(start, end)):
        if name := unicodedata.name(char, ''): ③
            for word in tokenize(name):
                index.setdefault(word, set()).add(char)
    return index
```

- ① tokenize é uma função geradora, assunto do [ch_generators].
- ② A variável local index está anotada. Sem a dica, o Mypy diz: Need type annotation for 'index' (hint: "index: dict[<type>, <type>] = ...").
- ③ Usei o operador morsa (*walrus operator*) := na condição do if. Ele atribui o resultado da chamada a unicodedata.name() a name, e a expressão inteira é calculada a partir daquele resultado. Quando o resultado é '', isso é falso, e o index não é atualizado.^[10]



Ao usar dict como um registro JSON, é comum que todas as chaves sejam do tipo str, com valores de tipos diferentes dependendo das chaves. Isso é tratado na [typeddict_sec], sobre typing.TypedDict.

8.5.7. Classes bases abstratas

Seja conservador no que envia, mas liberal no que aceita.

— lei de Postel, ou o Princípio da Robustez

A Tabela 15 apresenta várias classes abstratas de collections.abc. Idealmente, uma função deveria aceitar argumentos desses tipos abstratos—ou seus equivalentes de typing antes de Python 3.9—e não tipos concretos. Isso dá mais flexibilidade a quem chama a função.

Considere essa assinatura de função:

```
from collections.abc import Mapping

def name2hex(name: str, color_map: Mapping[str, int]) -> str:
```

Usar abc.Mapping permite ao usuário da função fornecer uma instância de dict, defaultdict, ChainMap, uma subclasse de UserDict subclass, ou qualquer outra classe que seja um *subtipo-de* Mapping.

Por outro lado, veja essa assinatura:

```
def name2hex(name: str, color_map: dict[str, int]) -> str:
```

Agora color_map tem que ser um dict ou um de seus subtipos, tal como defaultdict ou OrderedDict. Especificamente, uma subclasse de collections.UserDict não passaria pela verificação de tipo para color_map, apesar de ser a maneira recomendada de criar mapeamentos definidos pelo usuário, como vimos na Seção 3.6.5. O Mypy rejeitaria um UserDict ou uma instância de classe derivada dele, porque UserDict não é uma subclasse de dict; elas são irmãs. Ambas são subclasses abc.MutableMapping.^[11]

Assim, em geral é melhor usar abc.Mapping ou abc.MutableMapping em dicas de tipos de parâmetros, em vez de dict (ou typing.Dict em código antigo). Se a função name2hex não precisar modificar o color_map recebido, a dica de tipo mais precisa para color_map é abc.Mapping. Desse jeito, quem chama não precisa fornecer um objeto que implemente métodos como setdefault, pop, e update, que

fazem parte da interface de `MutableMapping`, mas não de `Mapping`. Isso reflete a segunda parte da lei de Postel: "[seja] liberal no que aceita."

A lei de Postel também nos diz para sermos conservadores no que enviamos. O valor de retorno de uma função é sempre um objeto concreto, então a dica de tipo do valor de saída deve ser um tipo concreto, como no exemplo em Seção 8.5.4 — que usa `list[str]`:

```
def tokenize(text: str) -> list[str]:  
    return text.upper().split()
```

No verbete de `typing.List` [https://fpy.li/4c], a documentação do Python 3.10 diz (NT: tradução abaixo não oficial)

Versão genérica de `list`. Útil para anotar tipos de retorno. Para anotar argumentos é preferível usar um tipo de coleção abstrata, como `Sequence` ou `Iterable`.

Comentários similares aparecem nos verbetes de `typing.Dict` [https://fpy.li/8-21] e `typing.Set` [https://fpy.li/8-22].

Lembre-se que a maioria das ABCs de `collections.abc` e outras classes concretas de `collections`, bem como as coleções nativas, suportam notação de dica de tipo genérica como `collections.deque[str]` desde o Python 3.9. As coleções correspondentes em `typing` só servem para suportar código escrito em Python 3.8 ou anterior. A lista completa de classes que se tornaram genéricas aparece em na seção "Implementation" [https://fpy.li/8-16] da PEP 585—Type Hinting Generics In Standard Collections [https://fpy.li/pep585] (EN).

Para encerrar nossa discussão de ABCs em dicas de tipo, precisamos falar sobre as ABCs `numbers`.

8.5.7.1. A queda da torre numérica

O pacote `numbers` [https://fpy.li/4d] define a assim chamada *torre numérica (numeric tower)* descrita na PEP 3141—A Type Hierarchy for Numbers [https://fpy.li/pep3141] (EN). A torre é uma hierarquia linear de ABCs, com `Number` no topo:

- `Number`
- `Complex`
- `Real`
- `Rational`
- `Integral`

Esses ABCs funcionam perfeitamente para checagem de tipo durante a execução, mas eles não são suportados para checagem de tipo estática. A seção "Numeric Tower" [https://fpy.li/cardxvi] da PEP 484 rejeita as ABCs `numbers` e manda tratar os tipos nativos `complex`, `float`, e `int` como casos especiais, como explicado em `int` é consistente-com `complex`. Voltaremos a essa questão na `[numbers_abc_proto_sec]`, que é dedicada a comparar protocolos e ABCs

Na prática, se você quiser anotar argumentos numéricos para checagem estática de tipos, há algumas opções:

1. Usar um dos tipo concretos, `int`, `float`, ou `complex` — como recomendado pela PEP 488.
2. Declarar um tipo union como `Union[float, Decimal, Fraction]`.
3. Se você quiser evitar a codificação explícita de tipos concretos, usar protocolos numéricos como `SupportsFloat`, tratados na `[runtime_checkable_proto_sec]`.

A Seção 8.5.10 abaixo é um pré-requisito para entender protocolos numéricos.

Antes disso, vamos examinar uma ABC muito útil em dicas de tipo: `Iterable`.

8.5.8. Iterable

A documentação de `typing.List` [<https://fpy.li/4c>] que citei acima recomenda `Sequence` e `Iterable` para dicas de tipo de parâmetros de função.

Esse é um exemplo de argumento `Iterable`, na função `math.fsum` da biblioteca padrão:

```
def fsum(__seq: Iterable[float]) -> float:
```

Arquivos Stub e o Projeto Typeshed



Até Python 3.10, a biblioteca padrão não tem anotações, mas Mypy, PyCharm, etc. conseguem encontrar as dicas de tipo necessárias no projeto Typeshed [<https://fpy.li/8-26>], na forma de *arquivos stub*: arquivos de código-fonte especiais, com a extensão `.pyi`, que contém assinaturas anotadas de métodos e funções, sem a implementação—semelhante a arquivos `.h` na linguagem C.

A assinatura para `math.fsum` está em `/stdlib/2and3/math.pyi` [<https://fpy.li/8-27>]. Os sublinhados iniciais em `__seq` são uma convenção estabelecida na PEP 484 para parâmetros apenas posicionais, como explicado em Seção 8.6.

O Exemplo 136 é outro exemplo do uso de um parâmetro `Iterable`, que produz itens que são `tuple[str, str]`. A função é usada assim:

```
>>> l33t = [('a', '4'), ('e', '3'), ('i', '1'), ('o', '0')]
>>> text = 'mad skilled noob powned leet'
>>> from replacer import zip_replace
>>> zip_replace(text, l33t)
'm4d sk1ll3d n00b p0wn3d l33t'
```

O Exemplo 136 mostra a implementação.

Exemplo 136. replacer.py

```
from collections.abc import Iterable
```

```

FromTo = tuple[str, str] ①

def zip_replace(text: str, changes: Iterable[FromTo]) -> str: ②
    for from_, to in changes:
        text = text.replace(from_, to)
    return text

```

① FromTo é um *apelido de tipo*: atribui tuple[str, str] a FromTo, para tornar a assinatura de zip_replace mais concisa.

② changes tem que ser um Iterable[FromTo]; é o mesmo que escrever Iterable[tuple[str, str]], mas é mais curto e mais fácil de ler.

O TypeAlias Explícito em Python 3.10

PEP 613—Explicit Type Aliases [<https://fpy.li/pep613>] introduziu um tipo especial, o TypeAlias, para tornar as atribuições que criam apelidos de tipos mais visíveis e mais fáceis para os checadores de tipos. A partir de Python 3.10, esta é a forma preferencial de criar um apelidos de tipo.



```

from typing import TypeAlias

FromTo: TypeAlias = tuple[str, str]

```

8.5.8.1. abc.Iterable versus abc.Sequence

Tanto math.fsum quanto replace.zip_replace tem que percorrer todos os argumentos do Iterable para produzir um resultado. Dado um iterável sem fim tal como o gerador itertools.cycle como entrada, essas funções consumiriam toda a memória e derrubariam o processo Python. Apesar desse perigo potencial, é muito comum no Python moderno se oferecer funções que aceitam um Iterable como argumento, mesmo se elas tem que processar a estrutura inteira para obter um resultado. Isso dá a quem chama a função a opção de fornecer um gerador como dado de entrada, em vez de uma sequência pré-construída, podendo economizar bastante memória se o número de itens de entrada for grande.

Por outro lado, a função columnize no Exemplo 134 requer uma Sequence, não um Iterable, pois ela precisa obter o len() do argumento para calcular previamente o número de linhas.

Assim como Sequence, o melhor uso de Iterable é como tipo de argumento. Ele é muito vago como um tipo de saída. Uma função deve ser mais precisa sobre o tipo concreto que retorna.

O tipo Iterator, usado como tipo do retorno no Exemplo 135, está intimamente relacionado a Iterable. Voltaremos a ele em [ch_generators], que trata de geradores e iteradores clássicos.

8.5.9. Genéricos parametrizados e TypeVar

Um genérico parametrizado é um tipo genérico, escrito na forma list[T], onde T é uma variável de tipo que será vinculada a um tipo específico a cada uso. Isso permite que o tipo de um argumento determine o tipo do resultado da função.

O Exemplo 137 define `sample`, uma função que recebe dois argumentos: uma Sequence de elementos de tipo `T` e um `int`. Ela retorna uma `list` de elementos do mesmo tipo `T`, escolhidos aleatoriamente do primeiro argumento.

O Exemplo 137 mostra a implementação.

Exemplo 137. sample.py

```
from collections.abc import Sequence
from random import shuffle
from typing import TypeVar

T = TypeVar('T')

def sample(population: Sequence[T], size: int) -> list[T]:
    if size < 1:
        raise ValueError('size must be >= 1')
    result = list(population)
    shuffle(result)
    return result[:size]
```

Aqui estão dois exemplos que ilustram porque é usei uma variável de tipo em `sample`:

- Se chamada com uma tupla de tipo `tuple[int, ...]` (que é *consistente-com* `Sequence[int]`) então o tipo parametrizado é `int`, e o tipo de retorno é `list[int]`.
- Se chamada com uma `str` (que é *consistente-com* `Sequence[str]`) então o tipo parametrizado é `str`, e o tipo do retorno é `list[str]`.

Por que TypeVar é necessário?

Os autores da PEP 484 queriam introduzir dicas de tipo ao acrescentar o módulo `typing`, sem mudar nada mais na linguagem. Com uma metaprogramação esperta, eles poderiam fazer o operador `[]` funcionar para classes como `Sequence[T]`. Mas o nome da variável `T` dentro dos colchetes precisa ser definido em algum lugar—do contrário o interpretador Python necessitaria de mudanças mais profundas, para suportar a notação de tipos genéricos como um caso especial de `[]`. Por isso o construtor `typing.TypeVar` é necessário: para introduzir o nome da variável no *namespace (espaço de nomes)* corrente. Linguagens como Java, C# e TypeScript não exigem que o nome da variável seja declarado previamente, então elas não precisam de algo equivalente à pseudo-classe `TypeVar` de Python.



Outro exemplo é a função `statistics.mode` da biblioteca padrão, que retorna o ponto de dado mais comum de uma série.

Aqui é uma exemplo de uso da documentação [<https://fpy.li/4e>]:

```
>>> mode([1, 1, 2, 3, 3, 3, 4])
3
```

Sem o uso de TypeVar, mode poderia ter uma assinatura como a apresentada no Exemplo 138.

Exemplo 138. mode_float.py: mode que opera com float e seus subtipos^[12]

```
from collections import Counter
from collections.abc import Iterable

def mode(data: Iterable[float]) -> float:
    pairs = Counter(data).most_common(1)
    if len(pairs) == 0:
        raise ValueError('no mode for empty data')
    return pairs[0][0]
```

Muitos dos usos de mode envolvem valores int ou float, mas Python tem outros tipos numéricos, e é desejável que o tipo de retorno siga o tipo dos elementos do Iterable recebido. Podemos melhorar aquela assinatura usando TypeVar. Vamos começar com uma assinatura parametrizada simples, mas errada.

```
from collections.abc import Iterable
from typing import TypeVar

T = TypeVar('T')

def mode(data: Iterable[T]) -> T:
```

Quando aparece pela primeira vez na assinatura, o parâmetro de tipo T aceita qualquer tipo. Da segunda vez que aparece, só aceitará o mesmo tipo vinculado da primeira vez.

Assim, qualquer iterável é *consistente-com* Iterable[T], incluindo iterável de tipos *unhashable* que collections.Counter não consegue tratar. Precisamos restringir os tipos possíveis de se atribuir a T. Vamos ver maneiras diferentes de fazer isso nas duas seções seguintes.

8.5.9.1. TypeVar restrita

O TypeVar aceita argumentos posicionais adicionais para restringir o tipo parametrizado. Podemos melhorar a assinatura de mode para aceitar um número específico de tipos, assim:

```
from collections.abc import Iterable
from decimal import Decimal
from fractions import Fraction
from typing import TypeVar

NumberT = TypeVar('NumberT', float, Decimal, Fraction)

def mode(data: Iterable[NumberT]) -> NumberT:
```

Está melhor que antes, e era a assinatura de mode em *statistics.pyi* [<https://fpy.li/8-30>], o arquivo stub em typeshed em 25 de maio de 2020.

Entretanto, a documentação em `statistics.mode` [https://fpy.li/8-28] inclui esse exemplo:

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

Na pressa, poderíamos apenas adicionar `str` à definição de `NumberT`:

```
NumberT = TypeVar('NumberT', float, Decimal, Fraction, str)
```

Com certeza funciona, mas `NumberT` estaria muito mal batizado se aceitasse `str`. Mais importante, não podemos ficar listando tipos para sempre, cada vez que percebermos que `mode` pode lidar com outro deles. Podemos fazer com melhor com um outro recurso de `TypeVar`, como veremos a seguir.

8.5.9.2. TypeVar delimitada

Examinando o corpo de `mode` no Exemplo 138, vemos que a classe `Counter` é usada para ordenação. `Counter` é baseada em `dict`, então o tipo do elemento do iterável `data` precisa ser *hashable*.

A princípio, essa assinatura pode parecer que funciona:

```
from collections.abc import Iterable, Hashable

def mode(data: Iterable[Hashable]) -> Hashable:
```

Agora o problema é que o tipo do item retornado é `Hashable`: um ABC que implementa apenas o método `__hash__`. Então o verificador de tipo não vai permitir que façamos nada com o valor retornado, exceto chamar seu método `hash()`. Não é muito útil.

A solução está em outro parâmetro opcional de `TypeVar`: o parâmetro representado pela palavra-chave `bound`. Ele estabelece um limite superior para os tipos aceitos. No Exemplo 139, temos `bound=Hashable`. Isso significa que o tipo do parâmetro pode ser `Hashable` ou qualquer *subtipo-de* `Hashable`.^[12]

Exemplo 139. mode_hashable.py: igual a Exemplo 138, mas com uma assinatura mais flexível

```
from collections import Counter
from collections.abc import Iterable, Hashable
from typing import TypeVar

HashableT = TypeVar('HashableT', bound=Hashable)

def mode(data: Iterable[HashableT]) -> HashableT:
    pairs = Counter(data).most_common(1)
    if len(pairs) == 0:
        raise ValueError('no mode for empty data')
    return pairs[0][0]
```

Em resumo:

- Uma variável de tipo restrita será vinculada um dos tipos nomeados na declaração do TypeVar.
- Uma variável de tipo delimitada será vinculada ao tipo inferido da expressão, desde que o tipo inferido seja *consistente-com* o limite declarado pelo argumento bound= do TypeVar.



É uma pena que a palavra-chave do argumento para declarar uma TypeVar delimitada tenha sido chamado bound=, (*limite*, como substantivo) pois o verbo "bound" (passado do verbo "to bind", *ligar ou vincular*) é muito usado em inglês para descrever a associação de um valor a uma variável. Seria menos confuso se a palavra-chave do argumento fosse boundary=, um substantivo mais comum e explícito que também significa *limite*.

O construtor de typing.TypeVar tem outros parâmetros opcionais (covariant e contravariant) que envolvem conceitos avançados que vamos deixar para o [ch_more_types], [variance_sec].

Agora vamos concluir essa introdução a TypeVar com AnyStr.

8.5.9.3. A variável de tipo pré-definida AnyStr

O módulo typing inclui uma TypeVar pré-definida chamada AnyStr, restrita aos tipos bytes e str. Ele é definido assim:

```
AnyStr = TypeVar('AnyStr', bytes, str)
```

O tipo AnyStr é usado em funções da biblioteca padrão que aceitam tanto bytes quanto str, e retornam valores do tipo recebido.

Agora vamos ver typing.Protocol, um novo recurso de Python 3.8, que permite um uso mais pythônico de dicas de tipo.

8.5.10. Protocolos estáticos



Em programação orientada a objetos, o conceito de um "protocolo" como uma interface informal é tão antigo quanto Smalltalk, e tem sido parte essencial de Python desde o início. Entretanto, no contexto de dicas de tipo, um protocolo é uma subclasse de typing.Protocol, definindo uma interface que um verificador de tipo pode analisar. Os dois tipos de protocolo são tratados no [ch_ifaces_prot_abc]. Aqui apresento apenas uma rápida introdução no contexto de anotações de função.

O tipo Protocol, como descrito em PEP 544—Protocols: Structural subtyping (static duck typing) [<https://fpy.li/pep544>] (EN), é similar às interfaces em Go: um tipo protocolo é definido especificando um ou mais métodos, e o verificador de tipo analisa se aqueles métodos estão implementados onde um tipo daquele protocolo é exigido.

Em Python, uma definição de protocolo é escrita como uma subclasse de typing.Protocol.

Entretanto, classes que *implementam* um protocolo não precisam herdar, registrar ou declarar qualquer relação com a classe que *define* o protocolo. É função do verificador de tipo encontrar os tipos de protocolos disponíveis e verificar sua utilização.

Abaixo temos um problema que pode ser resolvido com a ajuda de Protocol e TypeVar. Suponha que você quisesse criar uma função top(it, n), que retorna os n maiores elementos do iterável it:

```
>>> top([4, 1, 5, 2, 6, 7, 3], 3)
[7, 6, 5]
>>> l = 'mango pear apple kiwi banana'.split()
>>> top(l, 3)
['pear', 'mango', 'kiwi']
>>>
>>> l2 = [(len(s), s) for s in l]
>>> l2
[(5, 'mango'), (4, 'pear'), (5, 'apple'), (4, 'kiwi'), (6, 'banana')]
>>> top(l2, 3)
[(6, 'banana'), (5, 'mango'), (5, 'apple')]
```

A função genérica parametrizada top poderia ser implementada como no Exemplo 140.

Exemplo 140. a função top function com um parâmetro de tipo T indefinido

```
def top(series: Iterable[T], length: int) -> list[T]:
    ordered = sorted(series, reverse=True)
    return ordered[:length]
```

O problema é, como restringir T? Ele não pode ser Any ou object, pois series precisa funcionar com sorted. A sorted nativa na verdade aceita Iterable[Any], mas só porque o parâmetro opcional key recebe uma função que calcula uma chave de ordenação arbitrária para cada elemento. O que acontece se você passar para sorted uma lista de objetos simples, mas não fornecer um argumento key? Vamos tentar:

```
>>> l = [object() for _ in range(4)]
>>> l
[<object object at 0x10fc2fcfa0>, <object object at 0x10fc2fbb0>,
 <object object at 0x10fc2fbcc0>, <object object at 0x10fc2fbcd0>]
>>> sorted(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'object' and 'object'
```

A mensagem de erro mostra que sorted usa o operador < nos elementos do iterável. É só isso? Vamos tentar outro experimento rápido:^[13]

```
>>> class Spam:
...     def __init__(self, n): self.n = n
```

```

...     def __lt__(self, other): return self.n < other.n
...     def __repr__(self): return f'Spam({self.n})'
...
>>> l = [Spam(n) for n in range(5, 0, -1)]
>>> l
[Spam(5), Spam(4), Spam(3), Spam(2), Spam(1)]
>>> sorted(l)
[Spam(1), Spam(2), Spam(3), Spam(4), Spam(5)]

```

Isso confirma a suspeita: eu consigo passar um lista de `Spam` para `sort`, porque `Spam` implementa `__lt__`, o método especial do operador `<`.

Então o parâmetro de tipo `T` no Exemplo 140 deveria ser limitado a tipos que implementam `__lt__`. No Exemplo 139, precisávamos de um parâmetro de tipo que implementava `__hash__`, para poder usar `typing.Hashable` como limite superior do parâmetro de tipo. Mas agora não há um tipo adequado em `typing` ou `abc` para usarmos, então precisamos criar um.

O Exemplo 141 mostra o novo tipo `SupportsLessThan`, um `Protocol`.

Exemplo 141. comparable.py: a definição de um tipo Protocol, SupportsLessThan

```

from typing import Protocol, Any

class SupportsLessThan(Protocol): ①
    def __lt__(self, other: Any) -> bool: ... ②

```

① Um protocolo é uma subclasse de `typing.Protocol`.

② O corpo do protocolo tem uma ou mais definições de método, com `...` no lugar da implementação.

Um tipo `T` é *consistente-com* um protocolo `P` se `T` implementa todos os métodos definido em `P`, com assinaturas de tipo correspondentes.

Dado `SupportsLessThan`, agora podemos definir essa versão funcional de `top` no Exemplo 142.

Exemplo 142. top.py: definição da função top usando uma TypeVar com bound=SupportsLessThan

```

from collections.abc import Iterable
from typing import TypeVar

from comparable import SupportsLessThan

LT = TypeVar('LT', bound=SupportsLessThan)

def top(series: Iterable[LT], length: int) -> list[LT]:
    ordered = sorted(series, reverse=True)
    return ordered[:length]

```

Vamos testar `top`. O Exemplo 143 mostra parte de uma bateria de testes para uso com o `pytest`. Ele

tenta chamar `top` primeiro com um gerador de expressões que produz `tuple[int, str]`, e depois com uma lista de `object`. Com a lista de `object`, esperamos receber uma exceção de `TypeError`.

Exemplo 143. top_test.py: visão parcial da bateria de testes para top

```
from collections.abc import Iterator
from typing import TYPE_CHECKING ①

import pytest

from top import top

# muitas linhas omitidas

def test_top_tuples() -> None:
    fruit = 'mango pear apple kiwi banana'.split()
    series: Iterator[tuple[int, str]] = ( ②
        (len(s), s) for s in fruit)
    length = 3
    expected = [(6, 'banana'), (5, 'mango'), (5, 'apple')]
    result = top(series, length)
    if TYPE_CHECKING: ③
        reveal_type(series) ④
        reveal_type(expected)
        reveal_type(result)
    assert result == expected

# intentional type error
def test_top_objects_error() -> None:
    series = [object() for _ in range(4)]
    if TYPE_CHECKING:
        reveal_type(series)
    with pytest.raises(TypeError) as excinfo:
        top(series, 3) ⑤
    assert "<' not supported" in str(excinfo.value)
```

① A constante `typing.TYPE_CHECKING` é sempre `False` durante a execução do programa, mas os checadores de tipos fingem que ela é `True` quando estão fazendo a verificação.

② Declaração de tipo explícita para a variável `series`, para tornar mais fácil a leitura da saída do Mypy.^[14]

③ Esse `if` evita que as três linhas seguintes sejam executadas durante o teste.

④ `reveal_type()` não pode ser chamada durante a execução, porque não é uma função regular, mas sim um mecanismo de depuração do Mypy - por isso não há `import` para ela. Mypy vai produzir uma mensagem de depuração para cada chamada à pseudo-função `reveal_type()`, mostrando o tipo inferido do argumento.

⑤ Essa linha será marcada pelo Mypy como um erro.

Os testes anteriores são bem sucedidos - mas eles funcionariam de qualquer forma, com ou sem

dicas de tipo em `top.py`. Mais precisamente, se eu verificar aquele arquivo de teste com o Mypy, verei que o `TypeVar` está funcionando como o esperado. Veja a saída do comando `mypy` no Exemplo 144.



Desde o Mypy 0.910 (julho de 2021), em alguns casos a saída de `reveal_type` não mostra precisamente os tipos que declarei, mas mostra tipos compatíveis. Por exemplo, eu não usei `typing.Iterator` e sim `abc.Iterator`. Pode ignorar esse detalhe. O relatório do Mypy ainda é útil. Vou fingir que esse problema do Mypy já foi corrigido quando for discutir os resultados.

Exemplo 144. Saída do mypy top_test.py (linha quebradas para facilitar a leitura)

```
…/comparable/ $ mypy top_test.py
top_test.py:32: note:
    Revealed type is "typing.Iterator[Tuple[builtins.int, builtins.str]]" ①
top_test.py:33: note:
    Revealed type is "builtins.list[Tuple[builtins.int, builtins.str]]"
top_test.py:34: note:
    Revealed type is "builtins.list[Tuple[builtins.int, builtins.str]]" ②
top_test.py:41: note:
    Revealed type is "builtins.list[builtins.object*]" ③
top_test.py:43: error:
    Value of type variable "LT" of "top" cannot be "object" ④
Found 1 error in 1 file (checked 1 source file)
```

- ① Em `test_top_tuples`, `reveal_type(series)` mostra que ele é um `Iterator[tuple[int, str]]` que declarei explicitamente.
- ② `reveal_type(result)` confirma que o tipo produzido pela chamada a `top` é o que eu queria: dado o tipo de `series`, o `result` é `list[tuple[int, str]]`.
- ③ Em `test_top_objects_error`, `reveal_type(series)` mostra que ele é uma `list[object*]`. Mypy põe um `*` após qualquer tipo que tenha sido inferido: não anotei o tipo de `series` nesse teste.
- ④ Mypy marca o erro que esse teste produz intencionalmente: o tipo dos elementos do `Iterable` `series` não pode ser `object` (ele tem que ser do tipo `SupportsLessThan`).

A principal vantagem de um tipo protocolo sobre as ABCs é que uma classe não precisa de nenhuma declaração especial para ser *consistente-com* um tipo protocolo. Isso permite que um protocolo seja criado aproveitando tipos pré-existentes, ou tipos implementados em bases de código que não estão sob nosso controle. Eu não tenho que derivar ou registrar `str`, `tuple`, `float`, `set`, etc. com `SupportsLessThan` para usá-los onde um parâmetro `SupportsLessThan` é esperado. Eles só precisam implementar `__lt__`. E o verificador de tipo ainda será capaz de realizar seu trabalho, porque `SupportsLessThan` está explicitamente declarado como um `Protocol`—diferente dos protocolos implícitos comuns no duck typing, que são invisíveis para o checador de tipos.

A classe especial `Protocol` foi introduzida na PEP 544—Protocols: Structural subtyping (static duck typing) [<https://fpy.li/pep544>]. O Exemplo 142 demonstra porque esse recurso é conhecido como *duck typing estático (static duck typing)*: a solução para anotar o parâmetro `series` de `top` era dizer "O tipo nominal de `series` não importa, desde que ele implemente o método `__lt__`." Em Python, o duck

typing sempre permitiu dizer isso de forma implícita, deixando os checadores de tipos estáticos sem ação. Um verificador de tipo não consegue ler o código fonte em C do CPython, ou executar experimentos no console para descobrir que sorted só requer que seus elementos suportem <.

Agora podemos tornar o duck typing explícito para os verificadores estáticos de tipo. Por isso faz sentido dizer que `typing.Protocol` nos oferece *duck typing estático*.^[15]

Há mais para falar sobre `typing.Protocol`. Vamos voltar a ele na Parte IV, onde [ch_ifaces_prot_abc] compara as abordagens da tipagem estrutural, do duck typing e das ABCs—outro modo de formalizar protocolos. Além disso, a [overload_sec] explica como declarar assinaturas de funções de sobrecarga (*overload*) com `@typing.overload`, e inclui um exemplo bastante extenso usando `typing.Protocol` e uma `TypeVar` delimitada.



O `typing.Protocol` torna possível anotar a função `double` na Seção 8.4 sem perder funcionalidade. O segredo é definir uma classe de protocolo com o método `_mul_`. Convido você a fazer isso como um exercício. A solução está na [typed_double_sec] ([ch_ifaces_prot_abc]).

8.5.11. Callable

Para anotar parâmetros de callback ou objetos *callable* retornados por funções de ordem superior, o módulo `collections.abc` oferece o tipo `Callable`, disponível no módulo `typing` para quem ainda não estiver usando Python 3.9. Um tipo `Callable` é parametrizado assim:

```
Callable[[ParamType1, ParamType2], ReturnType]
```

A lista de parâmetros - `[ParamType1, ParamType2]` — pode ter zero ou mais tipos.

Aqui está um exemplo no contexto de uma função `repl`, parte do interpretador iterativo simples que veremos na [pattern_matching_case_study_sec]:^[16]

```
def repl(input_fn: Callable[[Any], str] = input) -> None:
```

Durante a utilização normal, a função `repl` usa a função `input` nativa de Python para ler expressões inseridas pelo usuário. Entretanto, para testagem automatizada ou para integração com outras fontes de entrada, `repl` aceita um parâmetro `input_fn` opcional: um `Callable` com o mesmo parâmetro e tipo de retorno de `input`.

A função embutida `input` tem a seguinte assinatura no `typeshed`:

```
def input(__prompt: Any = ...) -> str: ...
```

A assinatura de `input` é *consistente-com* esta dica de tipo `Callable`:

```
Callable[[Any], str]
```

Não existe sintaxe para a declarar o tipo de argumentos opcionais ou nomeados. A documentação [https://fpy.li/4f] de `typing.Callable` diz "tais funções são raramente usadas como tipo de callback." Se você precisar de um dica de tipo para acompanhar uma função com assinatura mais flexível, substitua a lista de parâmetros por ... - assim:

```
Callable[..., ReturnType]
```

A interação de parâmetros de tipo genéricos com uma hierarquia de tipos introduz um novo conceito: variância.

8.5.11.1. Variância em tipos callable

Imagine um sistema de controle de temperatura com uma função `update` simples, como mostrada no Exemplo 145. A função `update` chama a função `probe` para obter a temperatura atual, e chama `display` para mostrar a temperatura para o usuário. `probe` e `display` são ambas passadas como argumentos para `update`, por motivos didáticos. O objetivo do exemplo é contrastar duas anotações de `Callable`: uma como tipo de retorno e outra como tipo de parâmetro.

Exemplo 145. Ilustrando a variância.

```
from collections.abc import Callable

def update( ①
    probe: Callable[[], float], ②
    display: Callable[[float], None] ③
) -> None:
    temperature = probe()
    # imagine lots of control code here
    display(temperature)

def probe_ok() -> int: ④
    return 42

def display_wrong(temperature: int) -> None: ⑤
    print(hex(temperature))

update(probe_ok, display_wrong) # type error ⑥

def display_ok(temperature: complex) -> None: ⑦
    print(temperature)

update(probe_ok, display_ok) # OK ⑧
```

① `update` recebe duas funções callable como argumentos.

② `probe` precisa ser uma callable que não recebe nenhum argumento e retorna um `float`

③ `display` recebe um argumento `float` e retorna `None`.

④ `probe_ok` é *consistente-com* `Callable[[], float]` porque retornar um `int` não quebra código que

espera um float.

- ⑤ `display_wrong` não é *consistente-com* `Callable[[float], None]` porque não há garantia que uma função esperando um `int` consiga lidar com um `float`; por exemplo, a função `hex` de Python aceita um `int` mas rejeita um `float`.
- ⑥ O Mypy marca essa linha porque `display_wrong` é incompatível com a dica de tipo no parâmetro `display` em `update`.
- ⑦ `display_ok` é *consistente-com* `Callable[[float], None]` porque uma função que aceita um `complex` também consegue lidar com um argumento `float`.
- ⑧ Mypy está satisfeito com essa linha.

Resumindo, não há problema em fornecer uma função de callback que retorne um `int` quando o código espera uma função callback que retorne um `float`, porque um valor `int` sempre pode ser usado onde um `float` é esperado.

Formalmente, dizemos que `Callable[[], int]` é *subtipo-de* `Callable[[], float]`, assim como `int` é *subtipo-de* `float`. Isso significa que `Callable` é *covariante* no que diz respeito aos tipos de retorno, porque a relação *subtipo-de* dos tipos `int` e `float` aponta na mesma direção que os tipos `Callable` que os usam como tipos de retorno.

Por outro lado, é um erro de tipo fornecer uma função callback que recebe um argumento `int` quando é se espera um callback que possa processar um `float`.

Formalmente, `Callable[[int], None]` não é *subtipo-de* `Callable[[float], None]`. Apesar de `int` ser *subtipo-de* `float`, no `Callable` parametrizado a relação é invertida: `Callable[[float], None]` é *subtipo-de* `Callable[[int], None]`. Assim dizemos que aquele `Callable` é *contravariante* a respeito dos tipos de parâmetros declarados.

A [variance_sec] no [ch_more_types] explica variância em mais detalhes e com exemplos de tipos invariantes, covariantes e contravariantes.

Por hora, saiba que a maioria dos tipos genéricos parametrizados são *invariantes*, portanto mais simples. Por exemplo, se eu declaro `scores: list[float]`, isso me diz exatamente o que posso atribuir a `scores`. Não posso atribuir objetos declarados como `list[int]` ou `list[complex]`:



- Um objeto `list[int]` não é aceitável porque ele não pode conter valores `float` que meu código pode precisar colocar em `scores`.
- Um objeto `list[complex]` não é aceitável porque meu código pode precisar ordenar `scores` para encontrar a mediana, mas `complex` não fornece o método `__lt__`, então `list[complex]` não é ordenável.

Agora chegamos ou último tipo especial que examinaremos nesse capítulo.

8.5.12. NoReturn

Esse é um tipo especial usado apenas para anotar o tipo de retorno de funções que nunca retornam. Normalmente, elas existem para gerar exceções. Há dúzias dessas funções na biblioteca padrão.

Por exemplo, `sys.exit()` levanta `SystemExit` para encerrar o processo Python.

Sua assinatura no `typeshed` é:

```
def exit(__status: object = ...) -> NoReturn: ...
```

O parâmetro `__status__` é apenas posicional, e tem um valor default. Arquivos stub não contém valores default, em vez disso eles usam O tipo de `__status` é `object`, o que significa que pode também ser `None`, assim seria redundante escrever `Optional[object]`.

Na `[ch_class_metaprog]`, o `[checked_class_bottom_ex]` usa `NoReturn` em `__flag_unknown_attrs`, um método projetado para produzir uma mensagem de erro completa e amigável, e então levanta um `AttributeError`.

A última seção desse capítulo épico é sobre parâmetros posicionais e variádicos

8.6. Anotando parâmetros apenas posicionais e variádicos

Lembra da função `tag` do Exemplo 112? Da última vez que vimos sua assinatura foi em Seção 7.7.1:

```
def tag(name, /, *content, class_=None, **attrs):
```

Aqui está `tag`, completamente anotada e ocupando várias linhas - uma convenção comum para assinaturas longas, com quebras de linha como o formatador `blue` [<https://fpy.li/8-10>] faria:

```
from typing import Optional

def tag(
    name: str,
    /,
    *content: str,
    class_: Optional[str] = None,
    **attrs: str,
) -> str:
```

Observe a dica de tipo `*content: str`, para parâmetros posicionais arbitrários; Isso significa que todos aqueles argumentos tem que ser do tipo `str`. O tipo da variável local `content` no corpo da função será `tuple[str, ...]`.

A dica de tipo para argumentos nomeados arbitrários é `**attrs: str` neste exemplo, portanto o tipo de `attrs` dentro da função será `dict[str, str]`. Para uma dica de tipo como `**attrs: float`, o tipo de `attrs` na função seria `dict[str, float]`.

Se for necessário que o parâmetro `attrs` aceite valores de tipos diferentes, é preciso usar uma `Union[]` ou `Any`: `**attrs: Any`.

A notação / para parâmetros puramente posicionais só está disponível com Python ≥ 3.8 . Em Python 3.7 ou anterior, isso é um erro de sintaxe. A convenção da PEP 484 [<https://fpy.li/8-36>] é prefixar o nome cada parâmetro puramente posicional com dois sublinhados. Veja a assinatura de tag novamente, agora em duas linhas, usando a convenção da PEP 484:

```
from typing import Optional

def tag(__name: str, *content: str, class_: Optional[str] = None,
        **attrs: str) -> str:
```

O Mypy entende saber verificar as duas formas de declarar parâmetros puramente posicionais.

Para encerrar esse capítulo, vamos considerar brevemente os limites das dicas de tipo e do sistema de tipagem estática que elas suportam.

8.7. Tipos imperfeitos e testes poderosos

Os mantenedores de grandes bases de código corporativas relatam que muitos bugs são encontrados por checadores de tipos estáticos, e o custo de resolvê-los é menor que se os mesmos bugs fossem descobertos apenas após o código estar rodando em produção. Entretanto, é essencial observar que testes automatizados já eram uma boa prática largamente adotada muito antes da tipagem estática ser introduzida nas empresas que conheço usando Python.

Mesmo em contextos onde ela é mais benéfica, a tipagem estática não pode ser elevada a árbitro final da correção. Não é difícil encontrar:

Falsos Positivos

Ferramentas indicam erros de tipagem em código correto.

Falsos Negativos

Ferramentas não indicam erros em código incorreto.

Além disso, se formos forçados a checar o tipo de tudo, perdemos um pouco do poder expressivo de Python:

- Alguns recursos convenientes não podem ser checados de forma estática: por exemplo, o desempacotamento de argumentos como em `config(**settings)`.
- Recursos avançados como propriedades, descritores, metaclasses e metaprogramação em geral, têm suporte muito deficiente ou estão além da compreensão dos verificadores de tipo.
- Verificadores de tipo ficam obsoletos e/ou incompatíveis após o lançamento de novas versões de Python, rejeitando ou mesmo quebrando ao analisar código com novos recursos da linguagem—às vezes com atrasos de um ano ou até mais.

Restrições comuns de dados não podem ser expressas no sistema de tipo, mesmo restrições simples. Por exemplo, dicas de tipo são incapazes de garantir que "quantidade deve ser um inteiro > 0 " ou que "label deve ser uma string com 6 a 12 letras em ASCII." Em geral, dicas de tipo não são úteis para localizar erros na lógica do negócio subjacente ao código.

Dadas essas ressalvas, dicas de tipo não podem ser o pilar central da qualidade do software, e torná-las obrigatórias sem qualquer exceção só amplificaria os aspectos negativos.

Considere o verificador de tipo estático como uma das ferramentas na infraestrutura moderna de integração de código, ao lado de testadores, analisadores de código (*linters*), etc. O objetivo de uma estrutura de produção de integração de código é reduzir as falhas no software, e testes automatizados podem encontrar muitos bugs que estão fora do alcance de dicas de tipo. Qualquer código que possa ser escrito em Python pode ser testado em Python—com ou sem dicas de tipo.



O título e a conclusão dessa seção foram inspirados pelo artigo "Strong Typing vs. Strong Testing" [<https://fpy.li/8-37>] (EN) de Bruce Eckel, também publicado na antologia *The Best Software Writing I* [<https://fpy.li/8-38>] (EN), editada por Joel Spolsky (Apress). Bruce é um fã de Python, e autor de livros sobre C++, Java, Scala, e Kotlin. Naquele texto, ele conta como foi um defensor da tipagem estática até aprender Python, e conclui: "Se um programa em Python tem testes de unidade adequados, ele poderá ser tão robusto quanto um programa em C++, Java, ou C# com testes de unidade adequados (mas será mais rápido escrever os testes em Python).

Isso encerra nossa cobertura das dicas de tipo em Python por agora. Elas serão também o ponto central do [ch_more_types], que trata de classes genéricas, variância, assinaturas sobrecregadas, coerção de tipos (*type casting*), entre outros tópicos. Até lá, as dicas de tipo aparecerão em várias funções ao longo do livro.

8.8. Resumo do capítulo

Começamos com uma pequena introdução ao conceito de tipagem gradual, depois adotamos uma abordagem prática. É difícil ver como a tipagem gradual funciona sem uma ferramenta que efetivamente leia as dicas de tipo, então desenvolvemos uma função anotada guiados pelos relatórios de erro do Mypy.

Voltando à ideia de tipagem gradual, vimos como ela é um híbrido do duck typing tradicional de Python e da tipagem nominal mais familiar aos usuários de Java, C++ e de outras linguagens de tipagem estática.

A maior parte do capítulo foi dedicada a apresentar os principais grupos de tipos usados em anotações. Muitos dos tipos discutidos estão relacionados a tipos conhecidos de objetos de Python, como coleções, tuplas e callables - estendidos para suportar notação genérica do tipo `Sequence[float]`. Muitos daqueles tipos são substitutos temporários, implementados no módulo `typing` antes que os tipos padrão fossem modificados para suportar genéricos, no Python 3.9.

Alguns desses tipos são entidades especiais: `Any`, `Optional`, `Union`, e `NoReturn` não tem qualquer relação com objetos reais na memória, existem apenas no domínio abstrato do sistema de tipos.

Estudamos genéricos parametrizados e variáveis de tipo, que trazem mais flexibilidade para as dicas de tipo sem sacrificar a segurança da tipagem.

Genéricos parametrizáveis se tornam ainda mais expressivos com o uso de `Protocol`. Como só surgiu no Python 3.8, `Protocol` ainda não é muito usado—mas é muito importante. `Protocol` permite duck typing estático: é a ponte fundamental entre o núcleo de Python, construído sobre duck

typing, e a tipagem nominal que permite a checadores de tipos estáticos encontrarem bugs.

Ao discutir alguns desses tipos, usamos o Mypy para localizar erros de checagem de tipo e tipos inferidos, com a ajuda da função mágica `reveal_type()` do Mypy.

A seção final mostrou como anotar parâmetros exclusivamente posicionais e parâmetros variádicos.

Dicas de tipo são um tópico complexo e em constante evolução. Felizmente são um recurso opcional. Vamos manter Python acessível para a maior base de usuários possível, e parar de defender que todo código Python precisa ter dicas de tipo—como já vi em sermões públicos de evangelistas da tipagem estática.

Nosso BDFL^[17] emérito liderou o movimento de inclusão de dicas de tipo em Python, então é muito justo que esse capítulo comece e termine com suas palavras.

Não gostaria de uma versão de Python na qual eu fosse moralmente obrigado a adicionar dicas de tipo o tempo todo. Realmente acho que dicas de tipo tem seu lugar, mas há muitas ocasiões em que elas não valem a pena, e é maravilhoso que possamos escolher usá-las.^[18]

— Guido van Rossum

8.9. Para saber mais

Bernát Gábor escreveu em seu excelente post, "The state of type hints in Python" [<https://fpy.li/8-41>] (EN):

Dicas de Tipo deveriam ser usadas sempre que valha à pena escrever testes de unidade .

Sou um grande fã de testes, mas também escrevo muito código exploratório. Quando estou explorando, testes e dicas de tipo não ajudam. São um entrave.

Esse post do Gábor é uma das melhores introduções a dicas de tipo em Python que já encontrei, junto com o texto de Geir Arne Hjelle, "Python Type Checking (Guide)" [<https://fpy.li/8-42>] (EN). "Hypermodern Python Chapter 4: Typing" [<https://fpy.li/8-43>] (EN), de Claudio Jolowicz, é uma introdução mas curta que também fala de validação de checagem de tipo durante a execução.

Para uma abordagem mais aprofundada, a documentação do Mypy [<https://fpy.li/8-44>] é a melhor fonte. Ela é útil independente do verificador de tipo que você esteja usando, pois tem páginas de tutorial e de referência sobre tipagem em Python em geral - não apenas sobre o próprio Mypy.

Lá você também encontrará uma conveniente página de referência (ou _cheat sheet) [<https://fpy.li/8-45>] (EN) e uma página muito útil sobre problemas comuns e suas soluções [<https://fpy.li/8-46>] (EN).

A documentação do módulo typing [<https://fpy.li/4a>] é uma boa referência rápida, mas não entra em muitos detalhes.

A PEP 483—The Theory of Type Hints [<https://fpy.li/pep483>] (EN) inclui uma explicação aprofundada sobre variância, usando Callable para ilustrar a contravariância. As referências definitivas são as PEP relacionadas a tipagem. Já existem mais de 20 delas. A audiência alvo das PEPs são os core developers (*desenvolvedores principais da linguagem em si*) e o Steering Council de Python, então elas pressupõe uma grande quantidade de conhecimento prévio, e certamente não são uma leitura leve.

Como já mencionado, o [ch_more_types] cobre outros tópicos sobre tipagem, e a [more_type_hints_further_sec] traz referências adicionais, incluindo a [typing_peps_tbl], com a lista das PEPs sobre tipagem aprovadas ou em discussão até o final de 2021.

"Awesome Python Typing" [<https://fpy.li/8-47>] é uma ótima coleção de links para ferramentas e referências.

Ponto de vista

Apenas Pedale

Esqueça as desconfortáveis bicicletas ultraleves, as malhas brilhantes, os sapatos desajeitados que se prendem a pedais minúsculos, o esforço de quilômetros intermináveis. Em vez disso, faça como você fazia quando era criança - suba na sua bicicleta e descubra o puro prazer de pedalar.

— Grant Petersen, *Just Ride: A Radically Practical Guide to Riding Your Bike* (Apenas Pedale: Um Guia Radicalmente Prático sobre o Uso de sua Bicicleta) (Workman Publishing)

Se programar não é sua profissão principal, mas uma ferramenta útil no seu trabalho ou algo que você faz para aprender, experimentar e se divertir, você provavelmente não precisa de dicas de tipo mais que a maioria dos ciclistas precisa de sapatos com solas rígidas e presilhas metálicas.

Apenas programe.

O Efeito Cognitivo da Tipagem

Eu me preocupo com o efeito que as dicas de tipo terão sobre o estilo de programação em Python.

Concordo que usuários da maioria das APIs se beneficiam de dicas de tipo. Mas Python me atraiu—entre outras razões—porque oferece funções tão poderosas que substituem APIs inteiras, e podemos escrever nós mesmos funções poderosas similares. Considere a função nativa `max()` [<https://fpy.li/8-48>]. Ela é poderosa, mas é fácil de entender. Porém, vou mostrar na [max_overload_sec] que são necessárias 14 linhas de dicas de tipo para anotar corretamente essa função—sem contar a definição de um `typing.Protocol` e algumas definições de `TypeVar` para sustentar aquelas dicas de tipo.

Me inquieta que a adoção estrita de dicas de tipo em bibliotecas desencorajem

programadores de sequer considerarem programar funções assim no futuro.

De acordo com o verbete em inglês na Wikipedia, "relatividade linguística" [<https://fpy.li/8-49>]—ou a hipótese Sapir–Whorf—é um "princípio alegando que a estrutura de uma linguagem afeta a visão de mundo ou a cognição de seus falantes."

A Wikipedia continua:

- A versão *forte* diz que a linguagem *determina* o pensamento, e que categorias linguísticas limitam e determinam as categorias cognitivas.
- A versão *fraca* diz que as categorias linguísticas e seu uso apenas *influenciam* o pensamento e as decisões.

Linguistas em geral concordam que a versão forte é falsa, mas há evidência empírica apoiando a versão fraca.

Não conheço estudos específicos com linguagens de programação, mas na minha experiência, elas tiveram grande impacto sobre a forma como eu abordo problemas. A primeira linguagem de programação que usei profissionalmente foi o Applesoft BASIC, na era dos computadores de 8 bits. Recursão não era diretamente suportada pelo BASIC. Você teria que gerenciar uma pilha na unha para implementar um algoritmo recursivo. Então eu nunca pensava em usar algoritmos ou estruturas de dados recursivos. Eu sabia, em algum nível conceitual, que tais coisas existiam, mas elas não eram parte de meu arsenal de técnicas de resolução de problemas.

Décadas mais tarde, quando aprendi Elixir, gostei de resolver problemas com recursão e usei essa técnica além da conta—até descobrir que muitas das minhas soluções seriam mais simples se que usasse funções existentes nos módulos `Enum` e `Stream` do Elixir. Aprendi que o código idiomático de aplicações em Elixir raramente contém chamadas recursivas explícitas—em vez disso, usam enums e streams que implementam recursão por trás dos panos.

A relatividade linguística pode explicar a ideia recorrente (e também não provada) que aprender linguagens de programação diferentes torna alguém um programador melhor, especialmente quando as linguagens em questão suportam diferentes paradigmas de programação. Praticar com Elixir me tornou mais propenso a aplicar padrões funcionais quando escrevo programas em Python ou Go.

Agora voltando à Terra.

O pacote `requests` provavelmente teria uma API muito diferente se Kenneth Reitz tivesse decidido anotar todas as suas funções (ou tivesse recebido ordens de seu chefe para fazê-lo). Seu objetivo era escrever uma API que fosse fácil de usar, flexível e poderosa. Ele conseguiu, dada a fantástica popularidade de `requests` - em maio de 2020, ela estava em #4 nas PyPI Stats [<https://fpy.li/8-50>], com 2,6 milhões de downloads diários. A #1 era a `urllib3`, uma dependência de `requests`.

Em 2017 os mantenedores de `requests` decidiram [<https://fpy.li/8-51>] não perder seu tempo escrevendo dicas de tipo. Um deles, Cory Benfield, escreveu um email dizendo:

Acho que bibliotecas com APIs 'pythônicas' são as menos propensas a adotar esse sistema de tipagem, pois ele vai adicionar muito pouco valor a elas.

Naquela mensagem, Benfield incluiu esse exemplo extremo de uma tentativa de definição de tipo para o argumento nomeado `files` em `requests.request()` [<https://fpy.li/8-53>]:

```
Optional[  
    Union[  
        Mapping[  
            basestring,  
            Union[  
                Tuple[basestring, Optional[Union[basestring, file]]],  
                Tuple[basestring, Optional[Union[basestring, file]]],  
                Optional[basestring]],  
                Tuple[basestring, Optional[Union[basestring, file]]],  
                Optional[basestring], Optional[Headers]]  
    ]  
,  
    Iterable[  
        Tuple[  
            basestring,  
            Union[  
                Tuple[basestring, Optional[Union[basestring, file]]],  
                Tuple[basestring, Optional[Union[basestring, file]]],  
                Optional[basestring]],  
                Tuple[basestring, Optional[Union[basestring, file]]],  
                Optional[basestring], Optional[Headers]]  
    ]  
]  
]
```

E isso assume essa definição:

```
Headers = Union[  
    Mapping[basestring, basestring],  
    Iterable[Tuple[basestring, basestring]],  
]
```

Você acha que `requests` seria como é se os mantenedores insistissem em ter uma cobertura de dicas de tipo de 100%? `SQLAlchemy` é outro pacote importante que não trabalha muito bem com dicas de tipo.

O que torna essas bibliotecas fabulosas é incorporarem a natureza dinâmica de Python.

Apesar das dicas de tipo trazerem benefícios, há também um preço a ser pago.

Primeiro, há o investimento significativo de aprender como o sistema de tipos funciona. Esse é um preço alto, cobrado uma vez.

Mas há também um custo recorrente, eterno.

Perdemos parte do poder expressivo de Python se insistimos que tudo precisa estar sob a checagem de tipos. Recursos excelentes estão além da capacidade de compreensão dos checadores de tipos, por exemplo o desempacotamento de argumentos: `config(**settings)`.

Se quiser ter uma chamada como `config(**settings)` verificada quanto ao tipo, precisa explicitar cada argumento. Isso me traz lembranças de programas em Turbo Pascal, que escrevi 35 anos atrás.

Bibliotecas que usam metaprogramação são difíceis ou impossíveis de anotar. Claro que a metaprogramação pode ser mal usada, mas isso também é algo que torna muitos pacotes de Python divertidos de usar.

Se dicas de tipo se tornarem obrigatórias sem exceções, por uma decisão superior em grande empresas, aposto que logo veremos pessoas usando geração de código para reduzir a verbosidade em programas Python, uma prática comum com linguagens menos dinâmicas.

Para alguns projetos e contextos, dicas de tipo simplesmente não fazem sentido. Mesmo em contextos onde elas fazer muito sentido, não fazem sentido o tempo todo. Qualquer política razoável sobre o uso de dicas de tipo precisa conter exceções.

Alan Kay, que recebeu o prêmio Turing e foi um dos pioneiros da programação orientada a objetos, certa vez disse:

Algumas pessoas são completamente religiosas no que diz respeito a sistemas de tipo, e como um matemático eu adoro a ideia de sistemas de tipos, mas ninguém até agora inventou um que tenha alcance o suficiente.^[19]

Obrigado, Guido, pela tipagem opcional. Vamos usá-la como foi pensada, e não tentar anotar tudo em conformidade estrita com um estilo de programação que se parece com Java 1.5.

Duck Typing FTW

Duck typing encaixa bem no meu cérebro, e duck typing estático é um bom compromisso, permitindo checagem estática de tipo sem perder muito da flexibilidade que alguns sistemas de tipagem nominal só permitem ao custo de muita complexidade—quando permitem.

Antes da PEP 544, toda essa ideia de dicas de tipo me parecia completamente não-pythônica. Fiquei muito feliz quando vi `typing.Protocol` surgir em Python. Ele traz equilíbrio para a Força.

Generics ou specifics?

De uma perspectiva de Python, o uso do termo "genérico" na tipagem é invertido. Os sentidos comuns do termo "genérico" são "aplicável integralmente a um grupo ou uma classe" ou "sem

uma marca distintiva."

Considere `list` versus `list[str]`. O primeiro tipo é genérico: a lista aceita qualquer objeto. O segundo é específico: só aceita itens do tipo `str`.

Por outro lado, o termo faz sentido em Java. Antes de Java 1.5, todas as coleções de Java (exceto a mágica `array`) eram "specific": só podiam conter referências a `Object`, então era necessário converter os itens que saíam de uma coleção antes que eles pudessem ser usados. Com Java 1.5, as coleções ganharam parâmetros de tipo, e se tornaram "generic."

[1] PEP 484—Type Hints [<https://fpy.li/8-1>], seção *Rationale and Goals*; negritos mantidos do original.

[2] Um compilador JIT ("just-in-time", compiladores que transformam o bytecode gerado pelo interpretador em código da máquina-alvo no momento da execução) como o do PyPy tem informações muito melhores que as dicas de tipo: ele monitora o programa Python durante a execução, detecta os tipos concretos em uso, e gera código de máquina otimizado para aqueles tipos concretos.

[3] Em Python não há sintaxe para controlar o conjunto de possíveis valores de um tipo, exceto para tipos `Enum`. Por exemplo, não é possível, usando dicas de tipo, definir `Quantity` como um número inteiro entre 1 e 10000, ou `AirportCode` como uma sequência de 3 letras. O NumPy oferece `uint8`, `int16`, e outros tipos numéricos ligados à arquitetura do hardware, mas na biblioteca padrão de Python encontramos apenas tipos com conjuntos muitos pequenos de valores (`NoneType`, `bool`) ou conjuntos muito grandes (`float`, `int`, `str`, todas as tuplas possíveis, etc.).

[4] Duck typing é uma forma implícita de *tipagem estrutural*, que as anotações de tipo passaram a suportar explicitamente após a versão 3.8, com a introdução de `typing.Protocol`. Vamos falar disso mais adiante nesse capítulo, em Seção 8.5.10, e com mais detalhes no `[ch_ifaces_prot_abc]`.

[5] Muitas vezes a herança é sobreutilizada e difícil de justificar em exemplos pequenos. Então por favor aceite esse exemplo com animais como uma rápida ilustração de sub-tipagem.

[6] Professora do MIT, designer de linguagens de programação e homenageada com o Turing Award em 2008. Wikipedia: Barbara Liskov [<https://fpy.li/49>].

[7] Para ser mais preciso, `ord` só aceita `str` ou `bytes` com `len(s) == 1`. Mas no momento o sistema de tipagem não consegue expressar essa restrição.

[8] Em ABC - a linguagem que mais influenciou o design inicial de Python - cada lista estava restrita a aceitar valores de um único tipo: o tipo do primeiro item que você colocasse ali.

[9] Uma de minhas contribuições para a documentação do módulo `typing` foi acrescentar dúzias de avisos de descontinuação, enquanto eu reorganizava as entradas abaixo de "Conteúdo do Módulo" [<https://fpy.li/4b>] em subseções, sob a supervisão de Guido van Rossum.

[10] Eu uso `:=` quando faz sentido em alguns exemplos, mas não trato desse operador no livro. Veja PEP 572—Assignment Expressions [<https://fpy.li/pep572>] para entender os detalhes dos operadores de atribuição.

[11] Na verdade, `dict` é uma subclasse virtual de `abc.MutableMapping`. O conceito de subclasse virtual será explicado no `[ch_ifaces_prot_abc]`. Por hora, basta saber que `issubclass(dict, abc.MutableMapping)` é `True`, apesar de `dict` ser implementada em C, herdando apenas de `object`.

[12] Eu contribui com essa solução para `typeshed`, e em 26 de maio de 2020 mode aparecia anotado assim em `statistics.pyi` [<https://fpy.li/8-32>].

[13] É ótimo poder abrir um console iterativo e contar com o duck typing para explorar recursos da linguagem, como acabei de fazer. Sinto muita falta deste tipo de exploração quando uso linguagem que não tem esse recurso.

[14] Sem essa dica de tipo, o Mypy inferiria o tipo de `series` como `Generator[Tuple[builtins.int, builtins.str*], None, None]`, que é prolixo mas *consistente-com* `Iterator[tuple[int, str]]`, como veremos na `[generic_iterable_types_sec]`.

[15] Eu não sei quem inventou a expressão *duck typing estático*, mas ela se tornou mais popular com a linguagem Go, que tem uma semântica de interfaces que é mais parecida com os protocolos de Python que com as interfaces nominais de Java.

[16] REPL significa Read-Eval-Print-Loop (*Ler-Calcular-Imprimir-Recomeçar*), o comportamento básico de interpretadores iterativos.

[17] "Benevolent Dictator For Life." - Ditador Benevolente Vitalício. Veja Guido van Rossum em "Origin of BDFL" [<https://fpy.li/bdfl>].

[18] Do vídeo no Youtube, "Type Hints by Guido van Rossum (March 2015)" [<https://fpy.li/8-39>] (EN). A citação começa em 13'40" [<https://fpy.li/8-40>]. Editei levemente a transcrição para manter a clareza.

[19] Fonte: "A Conversation with Alan Kay" [<https://fpy.li/8-54>].