

## Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

- 1.1. Novidades nesse capítulo
- 1.2. Um baralho pythônico
- 1.3. Como os métodos especiais são utilizados
- 1.4. Visão geral dos, métodos especiais
- 1.5. Porque len não é um método?
- 1.6. Resumo do capítulo
- 1.7. Para saber mais

2. Uma coleção de sequências

- 2.1. Novidades neste capítulo
- 2.2. Uma visão geral das sequências embutidas
- 2.3. Compreensões de listas e expressões geradoras
- 2.4. Tuplas não são apenas listas imutáveis
- 2.5. Desempacotando sequências e iteráveis
- 2.6. Pattern matching com sequências
- 2.7. Fatiamento
- 2.8. Usando + e \* com sequências
- 2.9. list.sort versus a função embutida sorted
- 2.10. Quando uma lista não é a resposta
- 2.11. Resumo do capítulo
- 2.12. Leitura complementar

3. Dicionários e conjuntos

- 3.1. Novidades nesse capítulo
- 3.2. A sintaxe moderna dos dicts
- 3.3. Pattern matching com mapeamentos
- 3.4. A API padrão dos tipos de mapeamentos
- 3.5. Tratamento automático de chaves ausentes
- 3.6. Variações de dict
- 3.7. Mapeamentos imutáveis
- 3.8. Views de dicionários
- 3.9. Consequências práticas da forma como dict funciona
- 3.10. Teoria dos conjuntos
- 3.11. Consequências práticas da forma de funcionamento dos conjuntos
- 3.12. Operações de conjuntos em views de dict
- 3.13. Resumo do capítulo
- 3.14. Leitura complementar

4. Texto em Unicode versus Bytes

- 4.1. Novidades nesse capítulo
- 4.2. Questões de caracteres
- 4.3. Os fundamentos do

# Python Fluente, Segunda Edição (2023)

Luciano Ramalho

## 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](#). Os arquivos-fonte em formato *AsciiDoc* estão no repositório público <https://github.com/pythonfluente/pythonfluente2e>.

### NOTA

Esta tradução está em construção. No sumário, capítulos ainda não traduzidos estão marcados com o símbolo 🚧 (em obras).

Os links com colchetes [\[assim\]](#) não funcionam porque são referências cruzadas para capítulos ainda não publicados. Eles passarão a funcionar automaticamente quando seus destinos estiverem no ar.

Priorizamos os capítulos 6, 8, 13, 19, 20, e 21. para apoiar o curso *Python Engineer* a ser lançado pela [LINUXtips](#) em 2023. Depois do capítulo 21, os capítulos restantes serão traduzidos em ordem do 1 ao 24.

No momento não precisamos de ajuda para traduzir, mas correções ou sugestões de melhorias são muito bem vindas! Para contribuir, veja os [issues](#) 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](#) por todo o apoio que têm me dado nesse projeto. Quando entreguei o manuscrito para a O'Reilly, negocie 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](#). Com essa mudança contratual, a Editora Novatec não teve interesse em traduzir e publicar a segunda edição.

Felizmente encontrei meu querido amigo Paulo Candido 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 [AsciiDoctor](#) 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.

## Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

- 1.1. Novidades nesse capítulo
- 1.2. Um baralho pythônico
- 1.3. Como os métodos especiais são utilizados
- 1.4. Visão geral dos, métodos especiais
- 1.5. Porque len não é um método?
- 1.6. Resumo do capítulo
- 1.7. Para saber mais

2. Uma coleção de sequências

- 2.1. Novidades neste capítulo
- 2.2. Uma visão geral das sequências embutidas
- 2.3. Compreensões de listas e expressões geradoras
- 2.4. Tuplas não são apenas listas imutáveis
- 2.5. Desempacotando sequências e iteráveis
- 2.6. Pattern matching com sequências
- 2.7. Fatiamento
- 2.8. Usando + e \* com sequências
- 2.9. list.sort versus a função embutida sorted
- 2.10. Quando uma lista não é a resposta
- 2.11. Resumo do capítulo
- 2.12. Leitura complementar

3. Dicionários e conjuntos

- 3.1. Novidades nesse capítulo
- 3.2. A sintaxe moderna dos dicts
- 3.3. Pattern matching com mapeamentos
- 3.4. A API padrão dos tipos de mapeamentos
- 3.5. Tratamento automático de chaves ausentes
- 3.6. Variações de dict
- 3.7. Mapeamentos imutáveis
- 3.8. Views de dicionários
- 3.9. Consequências práticas da forma como dict funciona
- 3.10. Teoria dos conjuntos
- 3.11. Consequências práticas da forma de funcionamento dos conjuntos
- 3.12. Operações de conjuntos em views de dict
- 3.13. Resumo do capítulo
- 3.14. Leitura complementar

4. Texto em Unicode versus Bytes

- 4.1. Novidades nesse capítulo
- 4.2. Questões de caracteres
- 4.3. Os fundamentos do

O copyright desta tradução pertence a mim.

Luciano Ramalho, São Paulo, 13 de março de 2023

# Parte I: Estruturas de dados

## 1. O modelo de dados do 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 criar uma linguagem só um pouco menos teoricamente linda que, por isso mesmo, é uma delícia para programar.

Jim Hugunin, criador do Jython, co-criador do AspectJ, arquiteto do DLR (*Dynamic Language Runtime*) do .Net. "[Story of Jython](#)" (*A História do Jython*) (EN), escrito como prefácio ao [Jython Essentials](#) (EN), de Samuele Pedroni e Noel Rappin (O'Reilly).

Uma das melhores qualidades do Python é sua consistência. Após trabalhar com Python por algum tempo é possível intuir, de uma maneira informada e correta, o funcionamento de recursos que você acabou de conhecer.

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

É possível pensar no modelo de dados como uma descrição do Python na forma de uma 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 uma framework, gastamos um bom tempo programando métodos que são chamados por ela. O mesmo acontece quando nos valemos do Modelo de Dados do Python para criar novas classes. O interpretador do Python invoca métodos especiais para realizar operações básicas sobre os objetos, muitas vezes acionados 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, tais 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](#)" de *A Referência da Linguagem Python* adverte que "*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

## Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de sequências

2.1. Novidades neste capítulo

2.2. Uma visão geral das sequências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando sequências e iteráveis

2.6. Pattern matching com sequências

2.7. Fatiamento

2.8. Usando + e \* com sequências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Consequências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Consequências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

Esse capítulo sofreu poucas alterações desde a primeira edição, pois é uma introdução ao Modelo de Dados do 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 `%`.



DICA

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 se observar 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 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 do 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')
```

Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de sequências

2.1. Novidades neste capítulo

2.2. Uma visão geral das sequências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando sequências e iteráveis

2.6. Pattern matching com sequências

2.7. Fatiamento

2.8. Usando + e \* com sequências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Consequências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Consequências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

Deveríamos criar um método para obter uma carta aleatória? Não é necessário. O 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 do Python.

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

Mas fica melhor:

Como nosso `__getitem__` usa o operador `[]` de `self._cards`, nosso baralho suporta fatiamento automaticamente. Podemos olhar as três primeiras cartas no topo de um novo baralho, e depois pegar apenas 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')
...
```

NOTA

*Reticências nos doctests*

Sempre que possível, extraí as listagens do console do Python usadas neste livro com o `doctest`, para garantir a precisão. Quando a saída era grande 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. Se você estiver tentando rodar esses exemplos no console iterativo, pode simplesmente omitir todos os comentários de doctest.

A iteração muitas vezes é implícita. Se uma coleção não fornecer 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

Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de sequências

2.1. Novidades neste capítulo

2.2. Uma visão geral das sequências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando sequências e iteráveis

2.6. Pattern matching com sequências

2.7. Fatiamento

2.8. Usando + e \* com sequências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Consequências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Consequências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

essa regra, devolvendo 0 para o 2 de paus e 51 para o às 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 uso 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.

NOTA

E 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. Em Capítulo 13 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 para se saber sobre os métodos especiais é que eles foram 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 de uma classe definida pelo usuário, então o 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 como os arrays do NumPy. As coleções de tamanho variável do Python escritas em C incluem uma struct<sup>[1]</sup> chamada PyVarObject, com um campo ob\_size que mantém o número de itens na coleção. Então, se my\_object é uma instância de algum daqueles tipos embutidos, len(my\_object) lê o valor do campo ob\_size, e isso é muito 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 muito mais frequente que invocá-los explicitamente. O único método especial que é chamado frequentemente pelo seu código é \_\_init\_\_, para invocar o método de 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, Seção 17.3.1 no Capítulo 17.

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 de 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 capítulo Capítulo 16. Aqui nosso objetivo é continuar ilustrando o uso dos métodos especiais, através



## Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

- 1.1. Novidades nesse capítulo
- 1.2. Um baralho pythônico
- 1.3. Como os métodos especiais são utilizados
- 1.4. Visão geral dos, métodos especiais
- 1.5. Porque len não é um método?
- 1.6. Resumo do capítulo
- 1.7. Para saber mais

2. Uma coleção de sequências

- 2.1. Novidades neste capítulo
- 2.2. Uma visão geral das sequências embutidas
- 2.3. Compreensões de listas e expressões geradoras
- 2.4. Tuplas não são apenas listas imutáveis
- 2.5. Desempacotando sequências e iteráveis
- 2.6. Pattern matching com sequências
- 2.7. Fatiamento
- 2.8. Usando + e \* com sequências
- 2.9. list.sort versus a função embutida sorted
- 2.10. Quando uma lista não é a resposta
- 2.11. Resumo do capítulo
- 2.12. Leitura complementar

3. Dicionários e conjuntos

- 3.1. Novidades nesse capítulo
- 3.2. A sintaxe moderna dos dicts
- 3.3. Pattern matching com mapeamentos
- 3.4. A API padrão dos tipos de mapeamentos
- 3.5. Tratamento automático de chaves ausentes
- 3.6. Variações de dict
- 3.7. Mapeamentos imutáveis
- 3.8. Views de dicionários
- 3.9. Consequências práticas da forma como dict funciona
- 3.10. Teoria dos conjuntos
- 3.11. Consequências práticas da forma de funcionamento dos conjuntos
- 3.12. Operações de conjuntos em views de dict
- 3.13. Resumo do capítulo
- 3.14. Leitura complementar

4. Texto em Unicode versus Bytes

- 4.1. Novidades nesse capítulo
- 4.2. Questões de caracteres
- 4.3. Os fundamentos do

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](#)).



DICA

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 [Capítulo 17](#).

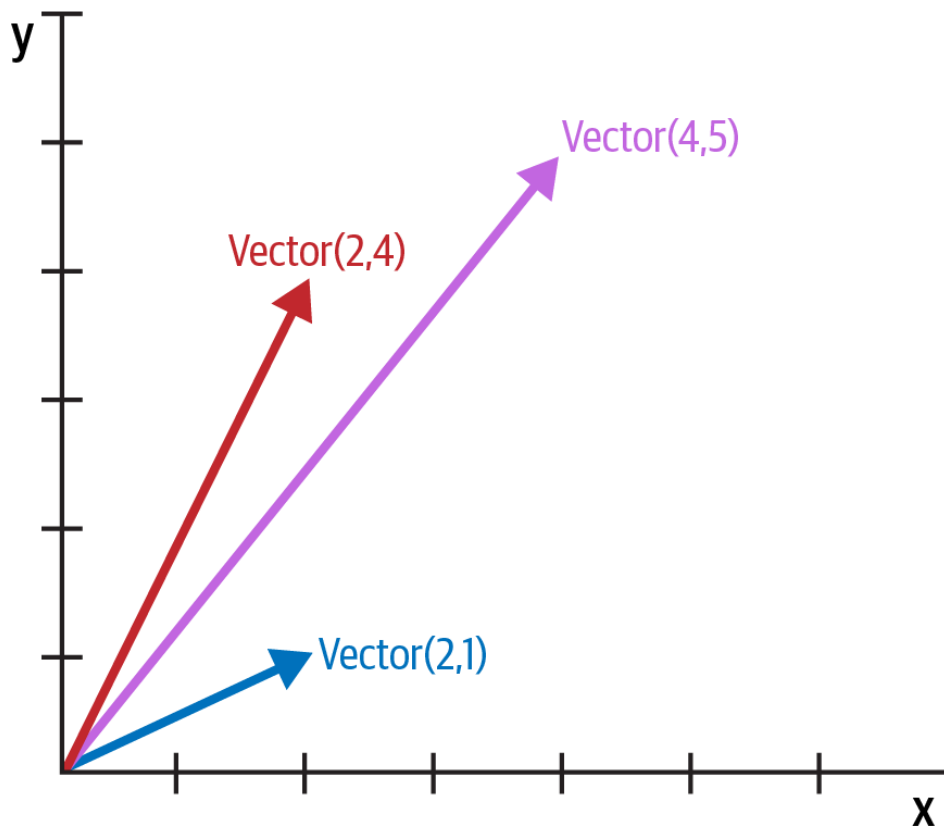


Figura 1. Exemplo de adição de vetores bi-dimensionais;  $\text{Vector}(2, 4) + \text{Vector}(2, 1)$  resulta em  $\text{Vector}(4, 5)$ .

Vamos começar a projetar a API para essa classe escrevendo em 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 `complex`. Então, por consistência, nossa API também usa `abs` para calcular a magnitude de um vetor:

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

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. A simple two-dimensional vector class*

## Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

- 1.1. Novidades nesse capítulo
- 1.2. Um baralho pythônico
- 1.3. Como os métodos especiais são utilizados
- 1.4. Visão geral dos, métodos especiais
- 1.5. Porque len não é um método?
- 1.6. Resumo do capítulo
- 1.7. Para saber mais

2. Uma coleção de sequências

- 2.1. Novidades neste capítulo
- 2.2. Uma visão geral das sequências embutidas
- 2.3. Compreensões de listas e expressões geradoras
- 2.4. Tuplas não são apenas listas imutáveis
- 2.5. Desempacotando sequências e iteráveis
- 2.6. Pattern matching com sequências
- 2.7. Fatiamento
- 2.8. Usando + e \* com sequências
- 2.9. list.sort versus a função embutida sorted
- 2.10. Quando uma lista não é a resposta
- 2.11. Resumo do capítulo
- 2.12. Leitura complementar

3. Dicionários e conjuntos

- 3.1. Novidades nesse capítulo
- 3.2. A sintaxe moderna dos dicts
- 3.3. Pattern matching com mapeamentos
- 3.4. A API padrão dos tipos de mapeamentos
- 3.5. Tratamento automático de chaves ausentes
- 3.6. Variações de dict
- 3.7. Mapeamentos imutáveis
- 3.8. Views de dicionários
- 3.9. Consequências práticas da forma como dict funciona
- 3.10. Teoria dos conjuntos
- 3.11. Consequências práticas da forma de funcionamento dos conjuntos
- 3.12. Operações de conjuntos em views de dict
- 3.13. Resumo do capítulo
- 3.14. Leitura complementar

4. Texto em Unicode versus Bytes

- 4.1. Novidades nesse capítulo
- 4.2. Questões de caracteres
- 4.3. Os fundamentos do

```
"""
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 muito mais sobre esse tópico no capítulo [Capítulo 16](#).



#### AVISO

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 capítulo [Capítulo 16](#).

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

### 1.3.2. Representação de strings

O método especial `__repr__` é chamado pelo `repr` embutido para obter a representação do objeto como string, para inspeção. Sem um `__repr__` personalizado, o console do Python mostraria uma instância de `Vector` como `<Vector object at 0x10e100070>`.

Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de seqüências

2.1. Novidades neste capítulo

2.2. Uma visão geral das seqüências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando seqüências e iteráveis

2.6. Pattern matching com seqüências

2.7. Fatiamento

2.8. Usando + e \* com seqüências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Conseqüências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Conseqüências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

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](#) 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 nosso código espera que os argumentos do construtor 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 pelo método embutido `str()` e usado implicitamente pela função `print`. Ele deve devolver uma string apropriada para ser exibida aos usuários finais.

Algumas vezes a própria string devolvida por `__repr__` é adequada para exibir ao usuário, e você não precisa programar `__str__`, porque a implementação herdada da classe `object` chama `__repr__` como alternativa. O [Exemplo 2](#) é um dos muitos exemplos neste livro com um `__str__` personalizado.



DICA

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?" \(Qual a diferença entre \\_\\_str\\_\\_ e \\_\\_repr\\_\\_ em Python?\)](#) (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 personalizado

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

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

Nossa implementação de `__bool__` é conceitualmente 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"](#) da documentação da *Biblioteca Padrão do Python*.



NOTA

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 a sigla para a mesma expressão em inglês, Abstract Base Classes*). As ABCs e o módulo `collections.abc` são tratados no capítulo [Capítulo 13](#). O objetivo dessa pequena seção é dar uma visão panorâmica das interfaces das coleções mais importantes do Python, mostrando como elas são criadas a partir de métodos especiais.



Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de sequências

2.1. Novidades neste capítulo

2.2. Uma visão geral das sequências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando sequências e iteráveis

2.6. Pattern matching com sequências

2.7. Fatiamento

2.8. Usando + e \* com sequências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Consequências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Consequências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

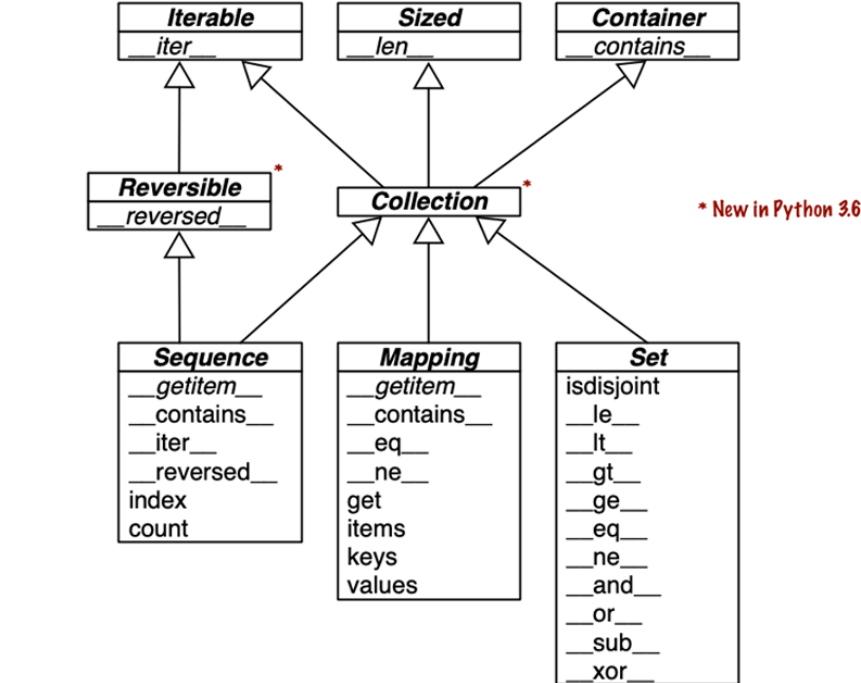


Figura 2. Diagrama de classes UML com os tipos fundamentais de coleções. Métodos como nome em itálico são abstratos, então precisam ser implementados pelas subclasses concretas, tais como list e dict. O restante dos métodos tem implementações concretas, então as subclasses podem herdá-los.

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, e outras formas de iteração
- Sized para suportar a função embutida len
- Container para suportar o operador in

Na verdade, o 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).

NOTA

Desde o Python 3.7, o tipo dict é oficialmente "ordenado", mas isso só que 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 do Python.

### 1.4. Visão geral dos, métodos especiais

O capítulo "Modelo de Dados" de A Referência da Linguagem Python lista mais de 80 nomes de métodos especiais. Mais da metade deles implementa operadores aritméticos, bit a bit, ou de comparação. 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 personalização de classes, \_\_init\_subclass\_\_ (do Python 3.6).

Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de sequências

2.1. Novidades neste capítulo

2.2. Uma visão geral das sequências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando sequências e iteráveis

2.6. Pattern matching com sequências

2.7. Fatiamento

2.8. Usando + e \* com sequências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Consequências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Consequências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

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>
Iteração	<code>__iter__</code> <code>__aiter__</code> <code>__next__</code> <code>__anext__</code> <code>__reversed__</code>
Execução de chamável ou corrotina	<code>__call__</code> <code>__await__</code>
Gerenciamento de contexto	<code>__enter__</code> <code>__exit__</code> <code>__aexit__</code> <code>__aenter__</code>
Criação e destruição de instâncias	<code>__new__</code> <code>__init__</code> <code>__del__</code>
Gerenciamento de atributos	<code>__getattr__</code> <code>__getattribute__</code> <code>__setattr__</code> <code>__delattr__</code> <code>__dir__</code>
Descritores de atributos	<code>__get__</code> <code>__set__</code> <code>__delete__</code> <code>__set_name__</code>
Classes base abstratas	<code>__instancecheck__</code> <code>__subclasscheck__</code>
Metaprogramação de classes	<code>__prepare__</code> <code>__init_subclass__</code> <code>__class_getitem__</code> <code>__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 um operador infixo de multiplicação de matrizes, como veremos no capítulo [Capítulo 16](#).

Tabela 2. Nomes e símbolos de métodos especiais para operadores

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

NOTA

O 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 capítulo [Capítulo 16](#) explica em detalhes os operadores reversos e a atribuição aumentada.

1.5. Porque len não é um método?

Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de seqüências

2.1. Novidades neste capítulo

2.2. Uma visão geral das seqüências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando seqüências e iteráveis

2.6. Pattern matching com seqüências

2.7. Fatiamento

2.8. Usando + e \* com seqüências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Conseqüências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Conseqüências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

Em 2013, fiz essa pergunta a Raymond Hettinger, um dos desenvolvedores principais do Python, e o núcleo de sua resposta era uma citação do ["The Zen of Python" \(O Zen do Python\)](#) (EN): "a praticidade vence a pureza." Em [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 do Python, da mesma forma que `abs`. Mas graças ao método especial `__len__`, também é possível fazer `len` funcionar com nossos objetos personalizados. Isso é um compromisso justo entre a necessidade de objetos embutidos eficientes e a consistência da linguagem. Também de "O Zen do Python": "Casos especiais não são especiais o bastante para quebrar as regras."

NOTA

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, a linguagem ABC—uma ancestral direta do Python, que antecipou muitas das funcionalidades desta última—tinha o operador `#`, que era o equivalente de `len` (se escrevia `#s`). Quando usado como operador infixo, `x#s` contava as ocorrências de `x` em `s`, que em Python obtemos com `s.count(x)`, para qualquer seqüência `s`.

1.6. Resumo do capítulo

Ao implementar métodos especiais, seus objetos podem se comportar como tipos embutidos, permitindo o estilo de programação expressivo que a comunidade considera pythônico.

Uma exigência básica para um objeto 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 seqüências, 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 seqüências. Tirar o máximo proveito dos tipos de seqüências existentes é o assunto do capítulo [Capítulo 2](#). Como implementar suas próprias seqüências será visto na seção [Capítulo 12](#), onde criaremos uma extensão multidimensional da classe `Vector`.

Grças à sobrecarga de operadores, o 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—incluindo operadores reversos e atribuição aumentada—será vista no capítulo [Capítulo 16](#), usando melhorias do exemplo `Vector`.

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

1.7. Para saber mais

O [capítulo "Modelo de Dados"](#) 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.](#) (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](#).

David Beazley tem dois livros tratando do modelo de dados em detalhes, no contexto do Python 3: [Python Essential Reference](#) (EN), 4th ed. (Addison-Wesley), e [Python Cookbook, 3rd ed.](#) (EN) (O'Reilly), com a co-autoria de Brian K. Jones.

O [The Art of the Metaobject Protocol](#) (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 do Python é um exemplo.

Ponto de Vista

Modelo de dados ou modelo de objetos?

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

Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de seqüências

2.1. Novidades neste capítulo

2.2. Uma visão geral das seqüências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando seqüências e iteráveis

2.6. Pattern matching com seqüências

2.7. Fatiamento

2.8. Usando + e \* com seqüências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Conseqüências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Conseqüências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

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 do Python, mas se referem a ele como o "modelo de objetos." Na Wikipedia, a primeira definição de "[modelo de objetos](#)" (EN) é: "as propriedades dos objetos em geral em uma linguagem de programação de computadores específica." É disso que o Modelo de Dados do Python trata. Neste livro, usarei "modelo de dados" porque a documentação prefere este termo ao se referir ao modelo de objetos do Python, e porque esse é o título do [capítulo de A Referência da Linguagem Python](#) mais relevante para nossas discussões.

Métodos de "trouxas"

O *The Original Hacker's Dictionary (Dicionário Hacker Original)* (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."

A comunidade Ruby chama o equivalente dos métodos especiais naquela linguagem de *métodos mágicos*. Muitos integrantes da comunidade Python também adotam esse termo. Eu acredito que os métodos especiais são o contrário de mágica. O Python e o Ruby oferecem a seus usuários um rico protocolo de metajetos integralmente documentado, permitindo que "trouxas" como você e eu possam emular muitas das funcionalidades disponíveis para os desenvolvedores principais que escrevem os interpretadores daquelas linguagens.

Por outro lado, pense no 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 do 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, o Go não tem o conceito de uma interface iterável ou um objeto iterador ao nível do 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 do Go melhorem seu protocolo de metajetos. Em 2021, ele ainda é muito mais limitado do que Python, Ruby, e JavaScript oferecem.

Metaobjetos

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

Um protocolo de metajetos 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 muito mais fácil de implementar em uma linguagem dinâmica como Python, e algumas frameworks fazem exatamente isso. O exemplo mais importante é a [zope.interface](#) (EN), parte da framework sobre a qual o sistema de gerenciamento de conteúdo [Plone](#) é construído.

2. Uma coleção de seqüências

Como vocês 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*). [...] O comando 'FOR' também funciona, de forma geral, em trens.

Leo Geurts, Lambert Meertens, e Steven Pembertonm, *ABC Programmer's Handbook*, p. 8. (Bosko Books)

Antes de criar o 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 "pithônicas": operações genéricas com diferentes tipos de seqüências, tipos tupla e mapeamento embutidos, estrutura [do código] por indentação, tipagem forte sem declaração de variáveis, entre outras. O Python não é assim tão amigável por acidente.

O Python herdou da ABC o tratamento uniforme de seqüências. Strings, listas, seqüências 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 seqüências disponíveis no Python evita que reinventemos a roda, e sua interface comum nos inspira a criar APIs que suportem e se aproveitem de forma apropriada dos tipos de seqüências existentes e futuras.

A maior parte da discussão deste capítulo se aplica às seqüências 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 seqüências de bytes são tratados no [Capítulo 4](#). Além disso, a ideia aqui é falar sobre os tipos de seqüências prontas para usar. A criação de novos tipos de seqüência é o tema do [Capítulo 12](#).

Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de sequências

2.1. Novidades neste capítulo

2.2. Uma visão geral das sequências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando sequências e iteráveis

2.6. Pattern matching com sequências

2.7. Fatiamento

2.8. Usando + e \* com sequências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Consequências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Consequências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

Os principais tópicos cobertos neste capítulo são:

- Compreensão de listas e os fundamentos das expressões geradoras.
- O uso de tuplas como registros versus o uso de tuplas como listas imutáveis
- Desempacotamento de sequências e padrões de sequências.
- Lendo de fatias e escrevendo em fatias
- Tipos especializados de sequências, tais como arrays e filas

2.1. Novidades neste capítulo

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

As outras mudanças não são atualizações e sim 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 [Seção 5.3](#) no [Capítulo 5](#), onde elas são comparadas com `typing.NamedTuple` e `@dataclass`.

**NOTA** 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](#) (EN) no site que complementa o livro, [fluentpython.com](#).

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

Armazenam 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 1](#).

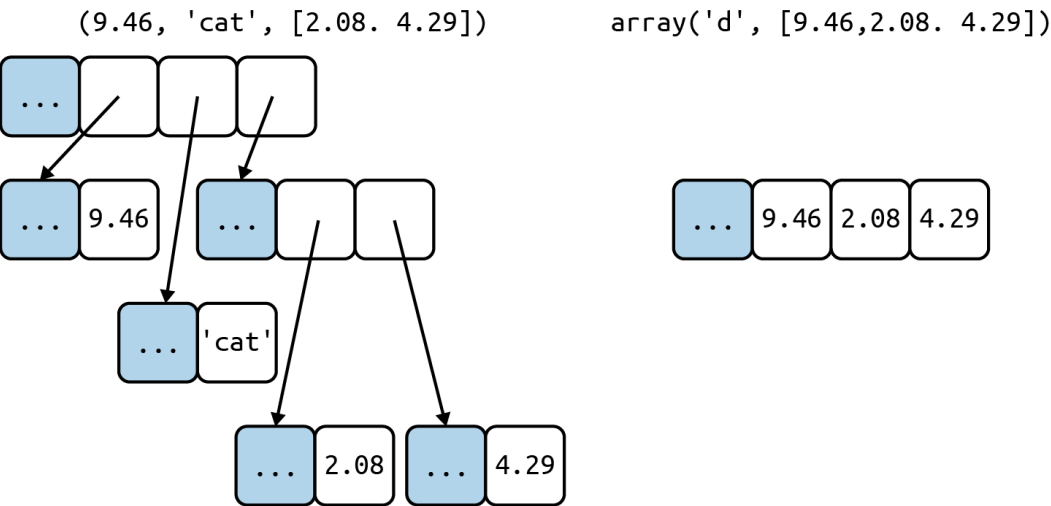


Figura 1. Diagramas de memória simplificados mostrando uma `tuple` e um `array`, cada uma com três itens. As células em cinza representam o cabeçalho de cada objeto Python na memória. A `tuple` tem um array de referências para seus itens. Cada item é um objeto Python separado, possivelmente contendo 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.



Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de sequências

2.1. Novidades neste capítulo

2.2. Uma visão geral das sequências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando sequências e iteráveis

2.6. Pattern matching com sequências

2.7. Fatiamento

2.8. Usando + e \* com sequências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Consequências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Consequências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

Dessa forma, sequências planas são mais compactas, mas estão limitadas a manter valores primitivos como bytes e números inteiros e de ponto flutuante.

NOTA

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 muito 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 2 ajuda a visualizar como as sequências mutáveis herdam todos os métodos das sequências imutáveis e implementam vários métodos adicionais. Os tipos embutidos concretos 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 Capítulo 13. Por serem subclasses virtuais, tuple e list passam nesses testes:

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

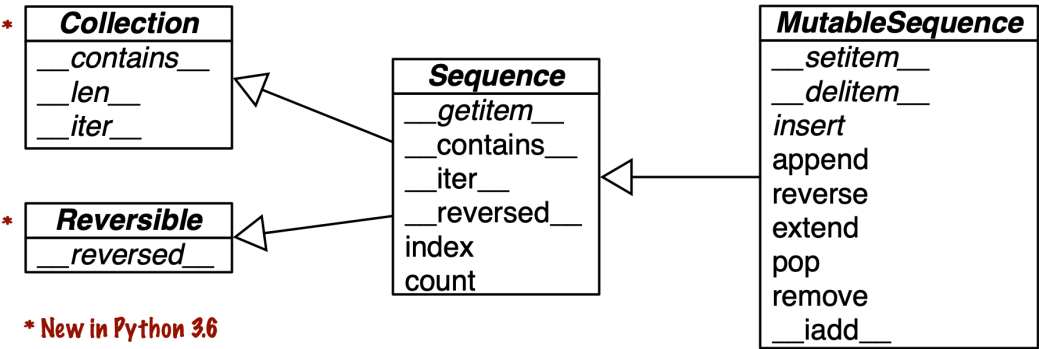


Figura 2. Diagrama de classe UML simplificado para algumas classes de collections.abc (as superclasses estão à esquerda; as setas de herança apontam das subclasses para as superclasses; nomes em itálico indicam classes e métodos abstratos).

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.

Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de sequências

2.1. Novidades neste capítulo

2.2. Uma visão geral das sequências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando sequências e iteráveis

2.6. Pattern matching com sequências

2.7. Fatiamento

2.8. Usando + e \* com sequências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Consequências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Consequências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo


3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do



DICA

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

Aqui está um teste: qual dos dois você acha mais fácil de ler, o [Exemplo 1](#) ou o [Exemplo 2](#)?

Exemplo 1. Cria uma lista de pontos de código Unicode a partir de uma string

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


Exemplo 2. Cria uma lista de pontos de código Unicode a partir de uma string, usando uma listcomp

```
>>> symbols = '$€¥€¤'
>>> 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 1](#). Entretanto, após aprender sobre as listcomps, acho o [Exemplo 2](#) 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 1](#) 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 compreensões 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

Dica de sintaxe

No 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 de bom tom 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 compreensões e expressões geradoras

No Python 3, compreensões de lista, expressões geradoras, e suas irmãs, as compreensões 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 compreensões ou expressões retornarem—diferente das variáveis locais em uma função. A [PEP 572—Assignment Expressions](#) (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 alvo.<sup>[2]</sup>

```
>>> 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):
```

Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de seqüências

2.1. Novidades neste capítulo

2.2. Uma visão geral das seqüências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando seqüências e iteráveis

2.6. Pattern matching com seqüências

2.7. Fatiamento

2.8. Usando + e \* com seqüências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Conseqüências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Conseqüências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

```
File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined

1. x não foi sobrescrito: continua vinculado a 'ABC' .

2. last permanece.

3. c desapareceu; ele só existiu dentro da listcomp.
```

Compreensões de lista criam listas a partir de seqüências ou de qualquer outro tipo iterável, filtrando e transformando os itens. As funções embutidas `filter` e `map` podem fazer o mesmo, mas perde-se alguma 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` do Python.

Considere o [Exemplo 3](#).

Exemplo 3. A mesma lista, criada por uma listcomp e por uma composição de map/filter

```
>>> symbols = '$çŁ¥€¤'
>>> 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 acreditava que `map` e `filter` eram 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](#) no [repositório de código do Python Fluente](#) é um teste de velocidade 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 da multiplicação dos tamanhos dos iteráveis usados. Veja a [Figura 3](#).

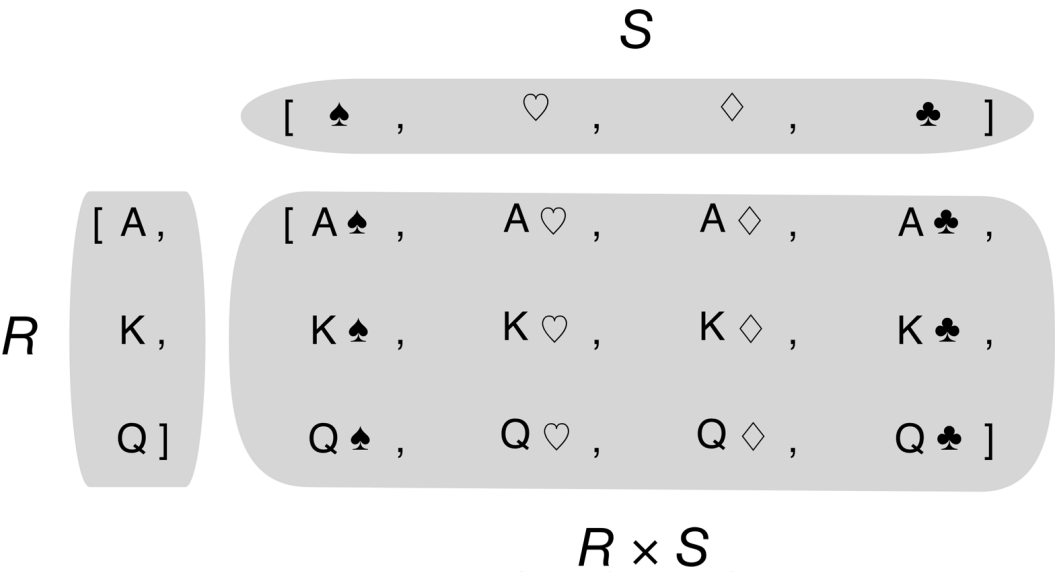


Figura 3. O produto cartesiano de 3 valores de cartas e 4 naipes é uma seqüência de 12 parâmetros.

Por exemplo, imagine que você precisa produzir uma lista de camisetas disponíveis em duas cores e três tamanhos. O [Exemplo 4](#) mostra como produzir tal lista usando uma listcomp. O resultado tem seis itens.

Exemplo 4. Produto cartesiano usando uma compreensão de lista

## Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

- 1.1. Novidades nesse capítulo
- 1.2. Um baralho pythônico
- 1.3. Como os métodos especiais são utilizados
- 1.4. Visão geral dos, métodos especiais
- 1.5. Porque len não é um método?
- 1.6. Resumo do capítulo
- 1.7. Para saber mais

2. Uma coleção de seqüências

- 2.1. Novidades neste capítulo
- 2.2. Uma visão geral das seqüências embutidas
- 2.3. Compreensões de listas e expressões geradoras
- 2.4. Tuplas não são apenas listas imutáveis
- 2.5. Desempacotando seqüências e iteráveis
- 2.6. Pattern matching com seqüências
- 2.7. Fatiamento
- 2.8. Usando + e \* com seqüências
- 2.9. list.sort versus a função embutida sorted
- 2.10. Quando uma lista não é a resposta
- 2.11. Resumo do capítulo
- 2.12. Leitura complementar

3. Dicionários e conjuntos

- 3.1. Novidades nesse capítulo
  - 3.2. A sintaxe moderna dos dicts
  - 3.3. Pattern matching com mapeamentos
  - 3.4. A API padrão dos tipos de mapeamentos
  - 3.5. Tratamento automático de chaves ausentes
  - 3.6. Variações de dict
  - 3.7. Mapeamentos imutáveis
  - 3.8. Views de dicionários
  - 3.9. Conseqüências práticas da forma como dict funciona
  - 3.10. Teoria dos conjuntos
  - 3.11. Conseqüências práticas da forma de funcionamento dos conjuntos
  - 3.12. Operações de conjuntos em views de dict
  - 3.13. Resumo do capítulo
  - 3.14. Leitura complementar
4. Texto em Unicode versus Bytes
- 4.1. Novidades nesse capítulo
  - 4.2. Questões de caracteres
  - 4.3. Os fundamentos do

```
>>> 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` ; adicionar uma quebra de linha listcomp 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 mágicos de um só truque: elas criam listas. Para gerar dados para outros tipos de seqüências, uma genexp é o caminho. A próxima seção é uma pequena incursão às genexps, no contexto de criação de seqüências que não são listas.

### 2.3.4. Expressões geradoras

Para inicializar tuplas, arrays e outros tipos de seqüências, você também poderia começar de 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 5](#) demonstra o uso básico de genexps para criar uma tupla e um array.

*Exemplo 5. 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 [Seção 2.10.1](#).

O [Exemplo 6](#) usa uma genexp com um produto cartesiano para gerar uma relação de camisetas de duas cores em três tamanhos. Diferente do [Exemplo 4](#), 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 6. Produto cartesiano em uma expressão geradora*

## Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

- 1.1. Novidades nesse capítulo
- 1.2. Um baralho pythônico
- 1.3. Como os métodos especiais são utilizados
- 1.4. Visão geral dos métodos especiais
- 1.5. Porque len não é um método?
- 1.6. Resumo do capítulo
- 1.7. Para saber mais

2. Uma coleção de sequências

- 2.1. Novidades neste capítulo
- 2.2. Uma visão geral das sequências embutidas
- 2.3. Compreensões de listas e expressões geradoras
- 2.4. Tuplas não são apenas listas imutáveis
- 2.5. Desempacotando sequências e iteráveis
- 2.6. Pattern matching com sequências
- 2.7. Fatiamento
- 2.8. Usando + e \* com sequências
- 2.9. list.sort versus a função embutida sorted
- 2.10. Quando uma lista não é a resposta
- 2.11. Resumo do capítulo
- 2.12. Leitura complementar

3. Dicionários e conjuntos

- 3.1. Novidades nesse capítulo
- 3.2. A sintaxe moderna dos dicts
- 3.3. Pattern matching com mapeamentos
- 3.4. A API padrão dos tipos de mapeamentos
- 3.5. Tratamento automático de chaves ausentes
- 3.6. Variações de dict
- 3.7. Mapeamentos imutáveis
- 3.8. Views de dicionários
- 3.9. Consequências práticas da forma como dict funciona
- 3.10. Teoria dos conjuntos
- 3.11. Consequências práticas da forma de funcionamento dos conjuntos
- 3.12. Operações de conjuntos em views de dict
- 3.13. Resumo do capítulo
- 3.14. Leitura complementar

4. Texto em Unicode versus Bytes

- 4.1. Novidades nesse capítulo
- 4.2. Questões de caracteres
- 4.3. Os fundamentos do

```
>>> 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 aparece neste exemplo.

### NOTA

O [Capítulo 17](#) 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 do 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 usada 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 ou não ter alguma importância, 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 7](#) 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 7. 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<sup>2</sup>).
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. O loop `for` sabe como recuperar separadamente os itens de uma tupla—isso é chamado "desempacotamento" ("*unpacking*"). Aqui não estamos interessados no segundo item, então o atribuímos a `_`, uma variável descartável, usada apenas para coletar valores que não serão usados.



## Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

- 1.1. Novidades nesse capítulo
- 1.2. Um baralho pythônico
- 1.3. Como os métodos especiais são utilizados
- 1.4. Visão geral dos métodos especiais
- 1.5. Porque len não é um método?
- 1.6. Resumo do capítulo
- 1.7. Para saber mais

2. Uma coleção de sequências

- 2.1. Novidades neste capítulo
- 2.2. Uma visão geral das sequências embutidas
- 2.3. Compreensões de listas e expressões geradoras
- 2.4. Tuplas não são apenas listas imutáveis
- 2.5. Desempacotando sequências e iteráveis
- 2.6. Pattern matching com sequências
- 2.7. Fatiamento
- 2.8. Usando + e \* com sequências
- 2.9. list.sort versus a função embutida sorted
- 2.10. Quando uma lista não é a resposta
- 2.11. Resumo do capítulo
- 2.12. Leitura complementar

3. Dicionários e conjuntos

- 3.1. Novidades nesse capítulo
- 3.2. A sintaxe moderna dos dicts
- 3.3. Pattern matching com mapeamentos
- 3.4. A API padrão dos tipos de mapeamentos
- 3.5. Tratamento automático de chaves ausentes
- 3.6. Variações de dict
- 3.7. Mapeamentos imutáveis
- 3.8. Views de dicionários
- 3.9. Consequências práticas da forma como dict funciona
- 3.10. Teoria dos conjuntos
- 3.11. Consequências práticas da forma de funcionamento dos conjuntos
- 3.12. Operações de conjuntos em views de dict
- 3.13. Resumo do capítulo
- 3.14. Leitura complementar

4. Texto em Unicode versus Bytes

- 4.1. Novidades nesse capítulo
- 4.2. Questões de caracteres
- 4.3. Os fundamentos do



### DICA

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 [Seção 2.6](#). E no console do Python, o resultado do comando anterior é atribuído a `_`—a menos que o resultado seja `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 7](#), atribuímos `('Tokyo', 2003, 32_450, 0.66, 8014)` a `city`, `year`, `pop`, `chg`, `area` em um único comando. E daí o operador `%` atribuiu cada item da tupla `passport` para a posição correspondente da string de formato no argumento `print`. Esses foram dois exemplos de *desempacotamento de tuplas*.



### NOTA

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 \(Desempacotamento Estendido de Iteráveis\)](#).

A seção [Seção 2.5](#) fala muito 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 deveria seguir o exemplo. Isso traz dois benefícios importantes:

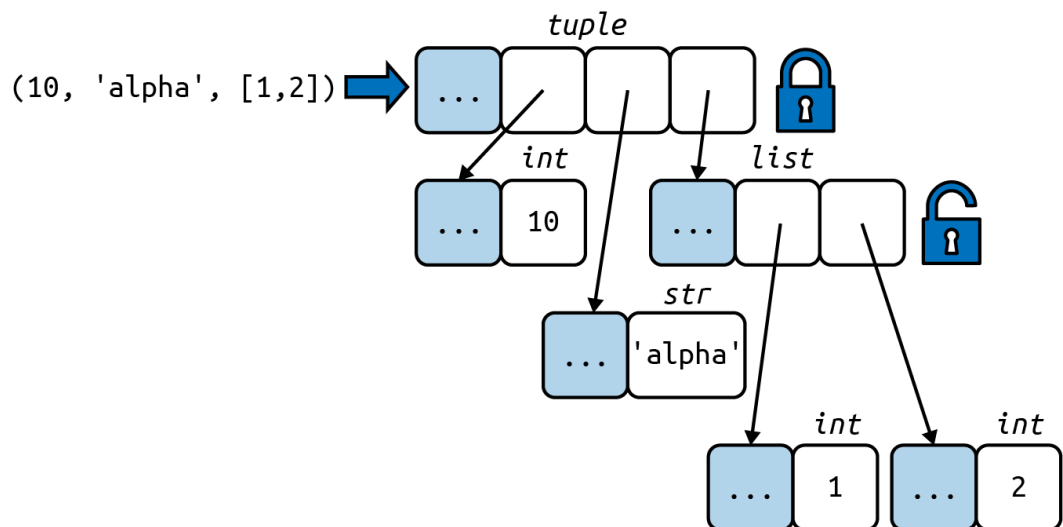
#### Clareza

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 um 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 4](#) representa a disposição inicial da tupla `b` na memória.



*Figura 4. 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, `b` e `a` se tornam diferentes:

```
>>> a = (10, 'alpha', [1, 2])
>>> b = (10, 'alpha', [1, 2])
>>> a == b
True
```

Sumário

Sobre esta tradução

Histórico das traduções

Parte I: Estruturas de dados

1. O modelo de dados do Python

1.1. Novidades nesse capítulo

1.2. Um baralho pythônico

1.3. Como os métodos especiais são utilizados

1.4. Visão geral dos, métodos especiais

1.5. Porque len não é um método?

1.6. Resumo do capítulo

1.7. Para saber mais

2. Uma coleção de sequências

2.1. Novidades neste capítulo

2.2. Uma visão geral das sequências embutidas

2.3. Compreensões de listas e expressões geradoras

2.4. Tuplas não são apenas listas imutáveis

2.5. Desempacotando sequências e iteráveis

2.6. Pattern matching com sequências

2.7. Fatiamento

2.8. Usando + e \* com sequências

2.9. list.sort versus a função embutida sorted

2.10. Quando uma lista não é a resposta

2.11. Resumo do capítulo

2.12. Leitura complementar

3. Dicionários e conjuntos

3.1. Novidades nesse capítulo

3.2. A sintaxe moderna dos dicts

3.3. Pattern matching com mapeamentos

3.4. A API padrão dos tipos de mapeamentos

3.5. Tratamento automático de chaves ausentes

3.6. Variações de dict

3.7. Mapeamentos imutáveis

3.8. Views de dicionários

3.9. Consequências práticas da forma como dict funciona

3.10. Teoria dos conjuntos

3.11. Consequências práticas da forma de funcionamento dos conjuntos

3.12. Operações de conjuntos em views de dict

3.13. Resumo do capítulo

3.14. Leitura complementar

4. Texto em Unicode versus Bytes

4.1. Novidades nesse capítulo

4.2. Questões de caracteres

4.3. Os fundamentos do

```
>>> 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 dos desenvolvedores principais do Python, Raymond Hettinger, em uma resposta à questão ["Are tuples more efficient than lists in Python?" \(As tuplas são mais eficientes que as listas no Python?\)](#) 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 no stack de dados, 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 tem alocadas para si 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, o Python precisa realocar 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 quão 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, isso é só uma otimização; reversed(my\_tuple) funciona sem esse método.

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

	list	tuple	
s.__add__(s2)	●	●	s + s2—concatenação
s.__iadd__(s2)	●		s += s2—concatenação no mesmo lugar
s.append(e)	●		Acrescenta um elemento após o último
s.clear()	●		Apaga todos os itens
s.__contains__(e)	●	●	e in s
s.copy()	●		Cópia rasa da lista