

Python Fluente, 2ª edição

volume 2: Classes e Protocolos

Luciano Ramalho

Tradução autorizada em português de *Fluent Python, 2nd Edition*

ISBN 978-1-492-05635-5 © 2022 Luciano Ramalho.

Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora dos direitos para publicação e venda desta obra.

© 2025 Luciano Ramalho.

Python Fluente, 2ª edição está publicado sob a licença CC BY-NC-ND 4.0

Atribuição-NãoComercial-SemDerivações 4.0 Internacional [fpy.li/ccby].

O autor mantém uma versão online em PythonFluente.com.

Autor: Luciano Ramalho

Título: Python Fluente, 2ª edição, volume 2: Classes e Protocolos

1ª edição: 2015

2ª edição: 2022

Revisão: pyfl2-vol2-pb-2026-03-02.pdf

Tradução da 2ª edição: Paulo Candido de Oliveira Filho

Ilustração de capa: Thiago Castor (xilogravura "Calango")

Design da capa: Luciano Ramalho, Zander Catta Preta @ Z•Edições

Design do miolo: Luciano Ramalho, com AsciiDoctor

Ficha catalográfica: Edison Luís dos Santos

Publisher: Heinar Maracy @ Z•Edições

R135p Ramalho, Luciano.

Python Fluente, 2ª edição, volume 2: Classes e Protocolos /
Luciano Ramalho - São Paulo, SP - Z.Edições, 2025.

364 p.; il.; 17 cm x 24 cm

ISBN: 978-65-989778-3-2

1.Informática. 2.Linguagem de Programação. 3.Python.
4.Metaprogramação.

I.Título II.Classes e Protocolos III.RAMALHO, Luciano.

CDU: 004.438

CDD: 005.133

Para Marta, com todo o meu amor.

Sumário

Parte II.b: Funções como objetos	1
9. Decoradores e Clausuras	3
9.1. Novidades neste capítulo.....	4
9.2. Introdução aos decoradores	4
9.3. Quando Python executa decoradores.....	6
9.4. Decoradores de registro	8
9.5. Regras de escopo de variáveis	9
9.6. Clausuras	13
9.7. A instrução <code>nonlocal</code>	17
9.8. Implementando um decorador simples.....	19
9.9. Decoradores na biblioteca padrão.....	23
9.10. Decoradores parametrizados	34
9.11. Resumo do capítulo.....	42
9.12. Para saber mais	43
10. Padrões de projetos com funções de primeira classe.....	49
10.1. Novidades neste capítulo.....	50
10.2. Estudo de caso: refatorando Estratégia	50
10.3. Estratégia com decorador de registro	63
10.4. O padrão Comando	64
10.5. Resumo do capítulo.....	67
10.6. Para saber mais	67
Parte III: Classes e Protocolos	73
11. Um objeto pythônico	75
11.1. Novidades neste capítulo.....	76
11.2. Representações de objetos	76
11.3. A volta da classe <code>Vector</code>	77
11.4. Um construtor alternativo	80
11.5. <code>classmethod</code> versus <code>staticmethod</code>	81
11.6. Exibição formatada.....	83
11.7. Um <code>Vector2d</code> <i>hashable</i>	88

11.8. Suportando o casamento de padrões posicionais	91
11.9. Listagem completa Vector2d, versão 3	92
11.10. Atributos privados e "protegidos" no Python.....	97
11.11. Economizando memória com <code>__slots__</code>	99
11.12. Sobrescrevendo atributos de classe	105
11.13. Resumo do capítulo	108
11.14. Para saber mais	110
12. Métodos especiais para sequências	117
12.1. Novidades neste capítulo	117
12.2. Vector: uma sequência definida pelo usuário	118
12.3. Vector versão #1: compatível com Vector2d	119
12.4. Protocolos e a tipagem pato	122
12.5. Vector versão #2: sequência fatiável	124
12.6. Vector versão #3: atributos dinâmicos.....	129
12.7. Vector versão #4: o <i>hash</i> e um <code>==</code> mais rápido.....	134
12.8. Vector versão #5: formatação.....	141
12.9. Resumo do capítulo	150
12.10. Para saber mais	151
13. Interfaces, protocolos, e ABCs	159
13.1. O mapa de tipagem.....	160
13.2. Novidades neste capítulo	161
13.3. Dois tipos de protocolos.....	162
13.4. Programando patos	164
13.5. Tipagem ganso	172
13.6. Protocolos estáticos.....	199
13.7. Resumo do capítulo	217
13.8. Para saber mais	218
14. Herança: para o bem ou para o mal.....	227
14.1. Novidades neste capítulo	228
14.2. A função <code>super()</code>	228
14.3. Problemas com subclasses de tipos embutidos	231
14.4. Herança múltipla e a Ordem de Resolução de Métodos	235

14.5. Classes <i>mixin</i>	242
14.6. Herança múltipla no mundo real	245
14.7. Lidando com a herança	253
14.8. Resumo do capítulo	259
14.9. Para saber mais	260
15. Mais dicas de tipo	267
15.1. Novidades neste capítulo	267
15.2. Assinaturas sobrecarregadas	267
15.3. A anotação <code>TypedDict</code>	276
15.4. Coerção de tipo (<i>type casting</i>)	286
15.5. Lendo dicas de tipo durante a execução	289
15.6. Implementando uma classe genérica	295
15.7. Variância	298
15.8. Implementando um protocolo estático genérico	307
15.9. Resumo do capítulo	310
15.10. Para saber mais	311
16. Sobrecarga de operadores	321
16.1. Novidades neste capítulo	322
16.2. Introdução à sobrecarga de operadores	323
16.3. Operadores unários	323
16.4. Sobrecarregando <code>+</code> para adição em <code>Vector</code>	327
16.5. Sobrecarregando <code>*</code> para multiplicação por escalar	334
16.6. Usando <code>@</code> como operador infixo	337
16.7. Resumindo os operadores aritméticos	339
16.8. Operadores de comparação rica	340
16.9. Operadores de atribuição aumentada	344
16.10. Resumo do capítulo	350
16.11. Para saber mais	352

Parte II.b: Funções como objetos

Esta versão impressa do *Python Fluente*, 2ª Edição foi publicada em três volumes, com oito capítulos por volume.

A Parte II ficou dividida entre os Volumes 1 e 2.



A Parte II.a está no Volume 1, e também na Web:

«Capítulo 7—Funções como objetos de primeira classe» [fpy.li/7]

«Capítulo 8—Padrões de projeto com funções» [fpy.li/8]

Os capítulos 9 e 10 estão neste Volume 2.

Capítulo 9. Decoradores e Clausuras

Houve uma certa quantidade de reclamações sobre a escolha do nome "decorador" para esse recurso. A mais frequente foi sobre o nome não ser consistente com seu uso no livro da GoF.^[1] O nome decorator provavelmente se origina de seu uso no âmbito dos compiladores—uma árvore sintática é percorrida e anotada.

— PEP 318—Decorators for Functions and Methods

Decoradores de função nos permitem "marcar" funções no código-fonte, para aprimorar de alguma forma seu comportamento. É um mecanismo muito poderoso. Por exemplo, o decorador `@functools.cache` armazena um mapeamento de argumentos para resultados, e depois usa esse mapeamento para evitar computar novamente o resultado quando a função é chamada com argumentos já vistos. Isso pode acelerar muito uma aplicação.

Para dominar esse recurso, é preciso antes entender clausuras (*closures*)—nome dado à estrutura onde uma função captura variáveis presentes no escopo onde a função é definida, necessárias para a execução da função futuramente.^[2]

A palavra reservada mais obscura de Python é `nonlocal`, introduzida no Python 3.0. É perfeitamente possível ter uma vida produtiva e lucrativa programando em Python sem jamais usá-la, seguindo uma dieta estrita de orientação a objetos centrada em classes. Entretanto, caso queira implementar seus próprios decoradores de função, precisa entender clausuras, e então a necessidade de `nonlocal` fica evidente.

Além de sua aplicação aos decoradores, clausuras também são essenciais para qualquer tipo de programação utilizando *callbacks*, e para codar em um estilo funcional quando isso fizer sentido.

O objetivo final deste capítulo é explicar exatamente como funcionam os decoradores de função, desde simples decoradores de registro até os complicados decoradores parametrizados. Mas antes de chegar a esse objetivo, precisamos tratar de:

- Como Python analisa a sintaxe de decoradores
- Como Python decide se uma variável é local
- Por que clausuras existem e como elas funcionam
- Qual problema é resolvido por `nonlocal`

Após criar essa base, chegaremos aos decoradores:

- Como implementar um decorador bem comportado
- Decoradores poderosos da biblioteca padrão: `@cache` e `@singledispatch`
- Como implementar um decorador parametrizado

9.1. Novidades neste capítulo

Nesta edição, apresento o decorador de *caching* `functools.cache` do Python 3.9 antes do `functools.lru_cache`, que é mais antigo. A Seção 9.9.2 apresenta também o uso de `lru_cache` sem argumentos, uma novidade do Python 3.8. Expandi a Seção 9.9.3 para incluir dicas de tipo, a sintaxe recomendada para usar `functools.singledispatch` desde o Python 3.7. A Seção 9.10 agora inclui o Exemplo 27, que usa uma classe e não uma clausura para implementar um decorador.

Começamos com a introdução aos decoradores mais suave que consegui imaginar.

9.2. Introdução aos decoradores

Um decorador é um invocável que recebe outra função como um argumento (a função decorada).

Um decorador pode executar algum processamento com a função decorada, e pode devolver a mesma função ou substituí-la por outra função ou objeto invocável.^[3]

Em outras palavras, supondo a existência de uma função decoradora chamada `decorate`, este código:


```
@decorate
def target():
    print('running target()')
```

tem o mesmo efeito de:

```
def target():
    print('running target()')

target = decorate(target)
```

O resultado final é o mesmo: após a execução de qualquer um destes exemplos, o nome `target` está vinculado a qualquer que seja a função devolvida por `decorate(target)`—que tanto pode ser a função inicialmente chamada `target` quanto uma outra função diferente.

Para confirmar que a função decorada é substituída, veja a sessão de console no Exemplo 1.

Exemplo 1. Um decorador normalmente substitui uma função por outra, diferente

```
>>> def deco(func):
...     def inner():
...         print('running inner()')
...     return inner ①
...
>>> @deco
... def target(): ②
...     print('running target()')
...
>>> target() ③
running inner()
>>> target ④
<function deco.<locals>.inner at 0x10063b598>
```

① `deco` devolve seu objeto função `inner`.

② `target` é decorada por `deco`.

- ③ Invocar a target decorada causa, na verdade, a execução de `inner`.
- ④ A inspeção revela que `target` é agora uma referência a `inner`.

Estritamente falando, decoradores são apenas açúcar sintático. Como vimos, é sempre possível chamar um decorador como um invocável normal, passando outra função como parâmetro. Algumas vezes isso inclusive é conveniente, especialmente quando estamos fazendo *metaprogramação*—mudando o comportamento de um programa durante a execução.

Três fatos essenciais sobre decoradores:

1. Um decorador é uma função ou outro invocável.
2. Um decorador pode, opcionalmente, substituir a função decorada por outra.
3. Decoradores são executados assim que um módulo é carregado.

Vamos agora nos concentrar nesse terceiro ponto.

9.3. Quando Python executa decoradores

Uma característica fundamental dos decoradores é serem executados logo após a função decorada ser definida. Isso normalmente acontece no momento da importação (*import time*), ou seja, quando um módulo é carregado pelo Python. Observe *registration.py* no Exemplo 2.

Exemplo 2. O módulo `registration.py`

```
registry = [] ①

def register(func): ②
    print(f'running register({func})') ③
    registry.append(func) ④
    return func ⑤

@register ⑥
def f1():
    print('running f1()')

@register
def f2():
```

```

    print('running f2()')

def f3(): ⑦
    print('running f3()')

def main(): ⑧
    print('running main()')
    print('registry ->', registry)
    f1()
    f2()
    f3()

if __name__ == '__main__':
    main() ⑨

```

- ① registry vai armazenar referências para funções decoradas por @register.
- ② register recebe uma função como argumento.
- ③ Exibe a função que está sendo decorada, para demonstração.
- ④ Insere func em registry.
- ⑤ É obrigatório devolver uma função; aqui devolvemos a mesma função recebida como argumento.
- ⑥ f1 e f2 são decoradas por @register.
- ⑦ f3 não é decorada.
- ⑧ main exibe registry, depois chama f1(), f2(), e f3().
- ⑨ main() só é invocada se *registration.py* for executado como um script.

O resultado da execução de *registration.py* é assim:

```

$ python3 registration.py
running register(<function f1 at 0x100631bf8>)
running register(<function f2 at 0x100631c80>)
running main()
registry -> [<function f1 at 0x100631bf8>, <function f2 at 0x100631c80>]
running f1()
running f2()
running f3()

```

Observe que `register` roda (duas vezes) antes de qualquer outra função no módulo. Quando `register` é chamada, ela recebe o objeto função a ser decorado como argumento—por exemplo, `<function f1 at 0x100631bf8>`.

Após o carregamento do módulo, a lista `registry` contém referências para as duas funções decoradas: `f1` e `f2`. Essas funções, bem como `f3`, são executadas apenas quando chamadas explicitamente por `main`.

Se `registration.py` for importado (e não executado como um script), a saída é essa:

```
>>> import registration
running register(<function f1 at 0x10063b1e0>)
running register(<function f2 at 0x10063b268>)
```

Nesse momento, se você inspecionar `registry`, verá isso:

```
>>> registration.registry
[<function f1 at 0x10063b1e0>, <function f2 at 0x10063b268>]
```

O ponto central do Exemplo 2 é enfatizar que decoradores de função são executados assim que o módulo é importado, mas as funções decoradas só rodam quando são invocadas explicitamente. Isso ressalta a diferença entre o *momento da importação* e o *momento da execução* na operação de um módulo em Python.

9.4. Decoradores de registro

Considerando a forma como decoradores são normalmente usados em código do mundo real, o Exemplo 2 é incomum por dois motivos:

- A função do decorador é definida no mesmo módulo das funções decoradas. Tipicamente, um decorador é definido em um módulo de uma biblioteca e aplicado a funções de outros módulos de bibliotecas ou aplicações.
- O decorador `register` devolve a mesma função recebida como argumento. Na prática, a maior parte dos decoradores define e devolve uma função interna.

Apesar do decorador `register` no Exemplo 2 devolver a função decorada inalterada, ele não é inútil. Decoradores parecidos são usados por muitos

frameworks Python para adicionar funções a um registro central—por exemplo, um registro mapeando padrões de URLs para funções que geram respostas HTTP. Tais decoradores de registro podem ou não modificar as funções decoradas.

Vamos ver um decorador de registro em ação na Seção 10.3 (Capítulo 10).

A maioria dos decoradores modifica a função decorada. Eles normalmente fazem isso definindo e devolvendo uma função interna para substituir a função decorada. Código que usa funções internas quase sempre depende de clausuras para operar corretamente. Para entender as clausuras, precisamos dar um passo atrás e revisar como o escopo de variáveis funciona no Python.

9.5. Regras de escopo de variáveis

No Exemplo 3, definimos e testamos uma função que lê duas variáveis: uma variável local `a`—definida como parâmetro de função—e a variável `b`, que não é definida em lugar algum na função.

Exemplo 3. Função lendo uma variável local e uma variável global

```
>>> def f1(a):
...     print(a)
...     print(b)
...
>>> f1(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f1
NameError: global name 'b' is not defined
```

O erro obtido não é surpreendente. Continuando do Exemplo 3, se atribuirmos um valor a um `b` global e então chamarmos `f1`, funciona:

```
>>> b = 6
>>> f1(3)
3
6
```

Agora vamos ver um exemplo que pode ser surpreendente.

Leia com atenção a função `f2`, no Exemplo 4. As primeiras duas linhas são as mesmas da `f1` do Exemplo 3, e então ela faz uma atribuição a `b`. Mas para com um erro no segundo `print`, antes da atribuição ser executada.

Exemplo 4. A variável `b` é local, porque um valor é atribuído a ela no corpo da função

```
>>> b = 6
>>> def f2(a):
...     print(a)
...     print(b)
...     b = 9
...
>>> f2(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f2
UnboundLocalError: local variable 'b' referenced before assignment
```

Observe que a saída começa com 3, provando que o comando `print(a)` foi executado. Mas o segundo, `print(b)`, nunca roda. Quando vi isso pela primeira vez, me espantei. Achei que o 6 seria exibido, pois há uma variável global `b`, e a atribuição para a variável local `b` ocorre após `print(b)`.

Mas quando Python compila o corpo da função, ele decide que `b` é uma variável local, por ser atribuída dentro da função. O bytecode gerado reflete essa decisão. O código tentará acessar `b` no escopo local. Mais tarde, quando a chamada `f2(3)` acontece, o corpo de `f2` obtém e exibe o valor da variável local `a`, mas ao tentar obter o valor da variável local `b`, descobre que `b` não está vinculado a nada.

Isso não é um bug, mas uma escolha de projeto: Python não exige que você declare variáveis, mas assume que uma variável que recebe uma atribuição no corpo de uma função é uma variável local. Isso é muito melhor que o comportamento de JavaScript, que também não requer declarações de variáveis, mas se você esquecer de declarar uma variável como local (com `var`), pode acabar alterando uma variável global por acidente.

Se queremos que o interpretador trate `b` como uma variável global e também atribuir um novo valor a ela dentro da função, usamos a declaração `global`:

```
>>> b = 6
>>> def f3(a):
...     global b
...     print(a)
...     print(b)
...     b = 9
...
>>> f3(3)
3
6
>>> b
9
```

Nos exemplos anteriores, vimos dois escopos em ação:

O escopo global do módulo

Composto por nomes atribuídos a valores fora de qualquer bloco de classe ou função.

O escopo local da função `f3`

Composto por nomes atribuídos a valores como parâmetros, ou diretamente no corpo da função.

Há um outro escopo onde variáveis podem existir, chamado *nonlocal* (não-local). Ele ocorre quando há funções aninhadas; vamos tratar disso em breve.

Agora que vimos como o escopo de variáveis funciona no Python, podemos enfrentar as clausuras na Seção 9.6. Se tiver curiosidade sobre as diferenças no bytecode das funções no Exemplo 3 e no Exemplo 4, veja o quadro a seguir.

Comparando bytecodes

O módulo `dis` descompila o bytecode de funções. Leia no Exemplo 5 e no Exemplo 6 os bytecodes de `f1` e `f2`, do Exemplo 3 e do Exemplo 4, respectivamente.

Exemplo 5. Bytecode da função f1 do Exemplo 3

```
>>> from dis import dis
>>> dis(f1)
 2          0 LOAD_GLOBAL          0 (print) ①
          3 LOAD_FAST              0 (a) ②
          6 CALL_FUNCTION          1 (1 positional, 0 keyword pair)
          9 POP_TOP

 3          10 LOAD_GLOBAL         0 (print)
          13 LOAD_GLOBAL          1 (b) ③
          16 CALL_FUNCTION          1 (1 positional, 0 keyword pair)
          19 POP_TOP
          20 LOAD_CONST           0 (None)
          23 RETURN_VALUE
```

① Carrega o nome global print.

② Carrega o nome local a.

③ Carrega o nome global b.

Compare o bytecode de f1, visto no Exemplo 5 acima, com o bytecode de f2 no Exemplo 6.

Exemplo 6. Bytecode da função f2 do Exemplo 4

```
>>> dis(f2)
 2          0 LOAD_GLOBAL          0 (print)
          3 LOAD_FAST              0 (a)
          6 CALL_FUNCTION          1 (1 positional, 0 keyword pair)
          9 POP_TOP

 3          10 LOAD_GLOBAL         0 (print)
          13 LOAD_FAST              1 (b) ①
          16 CALL_FUNCTION          1 (1 positional, 0 keyword pair)
          19 POP_TOP

 4          20 LOAD_CONST           1 (9)
          23 STORE_FAST              1 (b)
          26 LOAD_CONST           0 (None)
          29 RETURN_VALUE
```


- ① Carrega o nome *local* b. Isso mostra que o compilador considera b uma variável local, mesmo com uma atribuição a b ocorrendo mais tarde, porque a natureza da variável—se ela é ou não local—não pode mudar no corpo da função.

A máquina virtual (VM) do CPython que executa o bytecode é uma máquina de pilha (*stack machine*), então as operações LOAD e POP se referem à pilha. A descrição mais detalhada dos opcodes de Python está além da finalidade desse livro, mas eles estão documentados com o módulo, em "dis—Disassembler de bytecode de Python" [fpy.li/5x].

9.6. Clausuras

Na blogosfera, as clausuras (*closures*) são às vezes confundidas com funções anônimas. A confusão se deve à história paralela destes conceitos: definir funções dentro de outras funções se torna mais comum e conveniente quando existem funções anônimas. E clausuras só fazem sentido a partir do momento em que você tem funções aninhadas. Daí que muitos aprendem as duas ideias ao mesmo tempo.

Na verdade, uma clausura é uma função—vamos chamá-la de *f*—com um escopo estendido, incorporando variáveis acessadas no corpo de *f* que não são variáveis globais nem variáveis locais de *f*. Tais variáveis devem vir do escopo local de uma função que engloba *f*.

Não interessa se a função é anônima ou não; o que importa é que ela pode acessar variáveis não-globais definidas fora de seu corpo.

É um conceito difícil de entender, melhor ilustrado por um exemplo.

Imagine uma função *avg*, para calcular a média de uma série de valores que cresce continuamente; por exemplo, o preço diário de um produto ao longo de toda a sua história. A cada dia, um novo preço é acrescentado, e a média é computada levando em conta todos os preços até então.

Começando do zero, `avg` poderia ser usada assim:

```
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

Como `avg` é definida, e onde fica o histórico com os valores anteriores?

Para começar, o Exemplo 7 mostra uma implementação baseada em uma classe.

Exemplo 7. `average_oo.py`: uma classe para calcular uma média cumulativa

```
class Averager():

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total / len(self.series)
```

A classe `Averager` cria instâncias invocáveis:

```
>>> avg = Averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

O Exemplo 8, a seguir, é uma implementação funcional, usando a função de ordem superior `make_averager`.

Exemplo 8. average.py: uma função de ordem superior para a calcular uma média cumulativa

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total / len(series)

    return averager
```

Quando invocada, `make_averager` devolve um objeto função `averager`. Cada vez que um `averager` é invocado, ele insere o argumento recebido na série, e calcula a média atual, como mostra o Exemplo 9.

Exemplo 9. Testando o Exemplo 8

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(15)
12.0
```

Note as semelhanças entre os dois exemplos: chamamos `Averager()` ou `make_averager()` para obter um objeto invocável `avg`, que atualizará a série histórica e calculará a média atual. No Exemplo 7, `avg` é uma instância de `Averager`, no Exemplo 8 é a função interna `averager`. Nos dois casos, basta chamar `avg(n)` para incluir `n` na série e obter a média atualizada.

É óbvio onde o `avg` da classe `Averager` armazena o histórico: no atributo de instância `self.series`. Mas onde a função `avg` no Exemplo 8 encontra a `series`?

Observe que `series` é uma variável local de `make_averager`, pois a atribuição `series = []` acontece no corpo daquela função. Mas quando `avg(10)` é chamada, `make_averager` já retornou, e seu escopo local não existe mais.

Dentro de `averager`, `series` é uma *variável livre*: uma variável que é mencionada mas não é um parâmetro, e não tem uma atribuição no escopo local. Esse termo técnico é essencial para entender uma clausura. Veja a Figura 1.

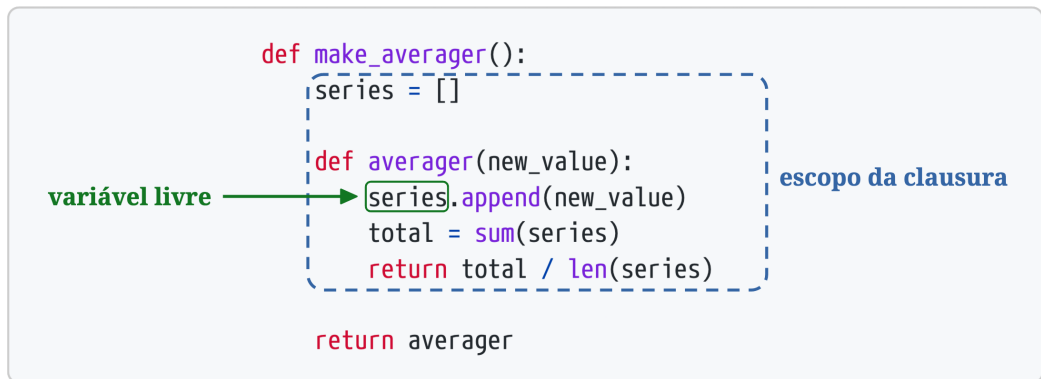


Figura 1. A clausura de `averager` estende o escopo daquela função para incluir a vinculação da variável livre `series`.

Podemos inspecionar o objeto `averager` para ver como Python armazena os nomes das variáveis locais e variáveis livres no atributo `__code__`, que representa o corpo compilado da função. O Exemplo 10 demonstra isso.

Exemplo 10. Inspecionando a função criada por `make_averager` no Exemplo 8

```
>>> avg.__code__.co_varnames
('new_value', 'total')
>>> avg.__code__.co_freevars
('series',)
```

O valor de `series` é armazenado no atributo `__closure__` da função `avg`. Cada item em `avg.__closure__` corresponde a um nome em `__code__`, e tem um atributo chamado `cell_contents`, com o valor vinculado. Confira o Exemplo 11:

Exemplo 11. Continuando do Exemplo 9

```
>>> avg.__closure__
(<cell at 0x107a44f78: list object at 0x107a91a48>,)
>>> avg.__closure__[0].cell_contents
[10, 11, 12]
```

Resumindo: uma clausura é uma função que retém os vínculos das variáveis livres que existem quando a função é definida, de forma que elas possam ser usadas mais tarde, quando a função for invocada, mas o escopo de sua definição não puder mais ser acessado.

Note que a única situação na qual uma função pode ter de lidar com variáveis externas não-globais é quando ela estiver aninhada dentro de outra função, e aquelas variáveis sejam parte do escopo local da função externa.

9.7. A instrução `nonlocal`

A implementação anterior de `make_averager` funciona, mas é ineficiente. No Exemplo 8, armazenamos todos os valores na série histórica e calculamos sua soma cada vez que `averager` é invocada. Uma implementação melhor armazenaria apenas o total e a contagem de itens até aquele momento, e calcularia a média com esses dois números.

O Exemplo 12 é uma implementação errada, apenas para ilustrar. Consegue ver onde o código quebra?

Exemplo 12. Função de ordem superior incorreta para calcular uma média cumulativa sem armazenar todo o histórico

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        count += 1
        total += new_value
        return total / count

    return averager
```

Ao testar o Exemplo 12, o resultado é um erro:

```
>>> avg = make_averager()
>>> avg(10)
Traceback (most recent call last):
...
UnboundLocalError: local variable 'count' referenced before assignment
>>>
```

O problema é que a instrução `count += 1` significa o mesmo que `count = count + 1`, quando `count` é um número ou qualquer tipo imutável. Então, estamos realmente atribuindo um valor a `count` no corpo de `averager`, e isso a torna uma variável local. O mesmo problema afeta a variável `total`.

Não tivemos esse problema no Exemplo 8, porque nunca atribuímos nada ao nome `series`; apenas chamamos `series.append` e invocamos `sum` e `len` nele. Nos valemos, então, do fato de listas serem mutáveis. Mas com tipos imutáveis, como números, strings, tuplas, etc., só é possível ler, nunca atualizar. Se você reatribuir, como em `count += 1`, estará implicitamente criando uma variável local `count`. Ela não será mais uma variável livre, e seu valor não será atualizado na clausura.

A palavra reservada `nonlocal` foi introduzida no Python 3 para contornar esse problema. Ela permite declarar uma variável livre, mesmo quando a variável é atribuída dentro da função. Se um novo valor é atribuído a uma variável `nonlocal`, o valor armazenado na clausura é atualizado. O Exemplo 13 é a implementação correta da versão otimizada de `make_averager`.

Exemplo 13. Calcula uma média cumulativa sem armazenar todo o histórico

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```

Após estudar o `nonlocal`, podemos resumir como a consulta de variáveis funciona no Python.

9.7.1. A lógica do acesso a variáveis

Quando uma função é definida, o compilador de bytecode de Python determina como encontrar uma variável `x` que aparece na função, baseado nas seguintes regras.^[4]

- Se há uma declaração `global x`, então `x` está vinculada à variável global `x` do módulo.^[5]
- Se há uma declaração `nonlocal x`, então `x` está vinculada à variável local `x` na função circundante mais próxima de onde `x` for definida.
- Se `x` é um parâmetro ou tem um valor atribuído a si no corpo da função, então `x` é uma variável local.
- Se `x` é referenciada mas não atribuída, e não é um parâmetro:
 - `x` será procurada nos escopos locais do corpos das funções circundantes (os escopos não-locais).
 - Se `x` não for encontrada nos escopos circundantes, será lida do escopo global do módulo.
 - Se `x` não for encontrada no escopo global, será lida de `__builtins__.__dict__`.

Tendo visto as clausuras de Python, podemos agora implementar decoradores com funções aninhadas.

9.8. Implementando um decorador simples

O Exemplo 14 é um decorador que cronometra cada invocação da função decorada e exibe o tempo decorrido, os argumentos passados, e o resultado da chamada.

Exemplo 14. clockdeco0.py: decorador que exibe o tempo de execução da função

```
import time

def clock(func):
    def clocked(*args): ①
        t0 = time.perf_counter()
        result = func(*args) ②
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print(f'[{elapsed:0.8f}s] {name}({arg_str}) -> {result!r}')
        return result
    return clocked ③
```

- ① Define a função interna `clocked` para aceitar qualquer número de argumentos posicionais.
- ② Essa linha só funciona porque a clausura de `clocked` engloba a variável livre `func`.
- ③ Devolve a função interna para substituir a função decorada.

O Exemplo 15 demonstra o uso do decorador `clock`.

Exemplo 15. Usando o decorador `clock`

```
import time
from clockdeco0 import clock

@clock
def snooze(seconds):
    time.sleep(seconds)

@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

if __name__ == '__main__':
    print('*' * 40, 'Calling snooze(.123)')
    snooze(.123)
```



```
print('*' * 40, 'Calling factorial(6)')
print('6! =', factorial(6))
```

O resultado da execução do Exemplo 15 é o seguinte:

```
$ python3 clockdeco_demo.py
***** Calling snooze(.123)
[0.12363791s] snooze(0.123) -> None
***** Calling factorial(6)
[0.00000095s] factorial(1) -> 1
[0.00002408s] factorial(2) -> 2
[0.00003934s] factorial(3) -> 6
[0.00005221s] factorial(4) -> 24
[0.00006390s] factorial(5) -> 120
[0.00008297s] factorial(6) -> 720
6! = 720
```

9.8.1. Como isso funciona

Lembre-se de que esse código:

```
@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)
```

na verdade faz isso:

```
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

factorial = clock(factorial)
```

Então, nos dois exemplos, `clock` recebe a função `factorial` como seu argumento `func` (veja o Exemplo 14). Ela então cria e devolve a função `clocked`, que o interpretador Python atribui a `factorial` (no primeiro exemplo, por baixo dos panos). De fato, se você importar o módulo `clockdeco_demo` e verificar o `__name__` de `factorial`, verá isso:

```
>>> import clockdeco_demo
>>> clockdeco_demo.factorial.__name__
'clogged'
```

O nome `factorial` agora é uma referência para a função `clogged`. Daqui por diante, cada vez que `factorial(n)` for invocada, `clogged(n)` será executada. Essencialmente, `clogged` faz o seguinte:

1. Registra o tempo inicial `t0`.
2. Chama a função `factorial` original, salvando o resultado.
3. Computa o tempo decorrido.
4. Formata e exibe os dados coletados.
5. Devolve o resultado salvo no passo 2.

Esse é o comportamento típico de um decorador: ele substitui a função decorada com uma nova função que aceita os mesmos argumentos e (normalmente) devolve o que quer que a função decorada deveria devolver, enquanto realiza também algum processamento adicional.



Em *Padrões de Projetos*, de Gamma et al., a descrição curta do padrão decorador começa assim: "Atribui dinamicamente responsabilidades adicionais a um objeto." Decoradores de função se encaixam nessa descrição. Mas, no nível da implementação, os decoradores de Python guardam pouca semelhança com o decorador clássico descrito no *Padrões de Projetos* original. No *Ponto de vista* escrevi mais sobre esse assunto.

O decorador `clock` implementado no Exemplo 14 tem alguns defeitos: ele não aceita argumentos nomeados, e encobre o `__name__` e o `__doc__` da função decorada.

O Exemplo 16 usa o decorador `functools.wraps` para copiar os atributos relevantes de `func` para `clogged`. Nesta nova versão, os argumentos nomeados também são tratados corretamente.

Exemplo 16. *clockdeco.py*: um decorador clock melhorado

```
import time
import functools

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.perf_counter()
        result = func(*args, **kwargs)
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_lst = [repr(arg) for arg in args]
        arg_lst.extend(f'{k}={v!r}' for k, v in kwargs.items())
        arg_str = ', '.join(arg_lst)
        print(f'[elapsed:{0.8f}s] {name}({arg_str}) -> {result!r}')
        return result
    return clocked
```

O `functools.wraps` é apenas um dos decoradores prontos para uso da biblioteca padrão. Na próxima seção, veremos o decorador mais útil oferecido por `functools`: `cache`.

9.9. Decoradores na biblioteca padrão

Python tem três funções embutidas projetadas para decorar métodos: `property`, `classmethod` e `staticmethod`. Vamos discutir `property` na «Seção 22.4» [fpy.li/8k] (vol.3) e os outros na Seção 11.5.

No Exemplo 16 vimos outro decorador importante: `functools.wraps`, um auxiliar na criação de decoradores bem comportados. Três dos decoradores mais interessantes da biblioteca padrão são `cache`, `lru_cache` e `singledispatch`—todos do módulo `functools`. Falaremos deles a seguir.

9.9.1. Memoização com `functools.cache`

O decorador `functools.cache` implementa *memoização*:^[6] uma técnica de otimização que armazena os resultados de invocações de uma função

dispendiosa em um *cache*, evitando repetir o processamento para argumentos previamente utilizados.

Uma boa demonstração é aplicar `@cache` à função recursiva, e dolorosamente lenta, que gera o *enésimo* número da sequência de Fibonacci, como mostra o Exemplo 17.

Exemplo 17. Algoritmo recursivo e ridiculamente dispendioso para calcular o enésimo número na série de Fibonacci

```
from clockdeco import clock

@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 2) + fibonacci(n - 1)

if __name__ == '__main__':
    print(fibonacci(6))
```

Aqui está o resultado da execução de *fibonacci_demo.py*. Exceto pela última linha, toda a saída é produzida pelo decorador `clock`:

```
$ python3 fibo_demo.py
[0.00000042s] fibonacci(0) -> 0
[0.00000049s] fibonacci(1) -> 1
[0.00006115s] fibonacci(2) -> 1
[0.00000031s] fibonacci(1) -> 1
[0.00000035s] fibonacci(0) -> 0
[0.00000030s] fibonacci(1) -> 1
[0.00001084s] fibonacci(2) -> 1
[0.00002074s] fibonacci(3) -> 2
[0.00009189s] fibonacci(4) -> 3
[0.00000029s] fibonacci(1) -> 1
[0.00000027s] fibonacci(0) -> 0
[0.00000029s] fibonacci(1) -> 1
[0.00000959s] fibonacci(2) -> 1
[0.00001905s] fibonacci(3) -> 2
```

```
[0.00000026s] fibonacci(0) -> 0
[0.00000029s] fibonacci(1) -> 1
[0.00000997s] fibonacci(2) -> 1
[0.00000028s] fibonacci(1) -> 1
[0.00000030s] fibonacci(0) -> 0
[0.00000031s] fibonacci(1) -> 1
[0.00001019s] fibonacci(2) -> 1
[0.00001967s] fibonacci(3) -> 2
[0.00003876s] fibonacci(4) -> 3
[0.00006670s] fibonacci(5) -> 5
[0.00016852s] fibonacci(6) -> 8
8
```

O desperdício é óbvio: `fibonacci(1)` é chamada oito vezes, `fibonacci(2)` cinco vezes, etc. Mas acrescentar apenas duas linhas, para usar cache, melhora muito o desempenho. Veja o Exemplo 18.

Exemplo 18. Implementação mais rápida, usando caching

```
import functools

from clockdeco import clock

@functools.cache ①
@clock ②
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 2) + fibonacci(n - 1)

if __name__ == '__main__':
    print(fibonacci(6))
```

- ① Essa linha funciona com Python 3.9 ou posterior. Veja a Seção 9.9.2 para uma alternativa que suporta versões anteriores de Python.
- ② Este é um exemplo de decoradores empilhados: `@cache` é aplicado à função devolvida por `@clock`.



Para entender os decoradores empilhados (*stacked decorators*), lembre-se de que `@` é açúcar sintático para aplicar a função decoradora à função abaixo dela. Se houver mais de um decorador, eles se comportam como invocações aninhadas.

Este código...

```
@alpha
@beta
def my_fn():
    ...
```

...faz o mesmo que este:

```
my_fn = alpha(beta(my_fn))
```

Em outras palavras, o decorador beta é aplicado primeiro, e a função devolvida por ele é então passada para alpha.

Usando o cache no Exemplo 18, a função `fibonacci` é chamada apenas uma vez para cada valor de `n`:

```
$ python3 fibo_demo_lru.py
[0.00000043s] fibonacci(0) -> 0
[0.00000054s] fibonacci(1) -> 1
[0.00006179s] fibonacci(2) -> 1
[0.00000070s] fibonacci(3) -> 2
[0.00007366s] fibonacci(4) -> 3
[0.00000057s] fibonacci(5) -> 5
[0.00008479s] fibonacci(6) -> 8
8
```

Em outro teste, para calcular `fibonacci(30)`, o Exemplo 18 fez as 31 chamadas necessárias em 0,00017s (tempo total), enquanto o Exemplo 17 sem cache, demorou 12,09s em um notebook Intel Core i7, porque chamou `fibonacci(1)` 832.040 vezes, num total de 2.692.537 chamadas!

Todos os argumentos recebidos pela função decorada devem ser *hashable*, pois o *cache* usa um dict para armazenar os resultados, e as chaves são formadas pelos argumentos posicionais e nomeados usados nas chamadas.

Além de tornar viáveis esses algoritmos recursivos tolos, @cache brilha de verdade em aplicações que precisam buscar informações de APIs remotas.



O `functools.cache` pode consumir toda a memória disponível, se houver um número muito grande de itens no cache. Eu o considero mais adequado para scripts rápidos de linha de comando. Para processos de longa duração, recomendo usar `functools.lru_cache` com um parâmetro `maxsize` adequado, como explicado na próxima seção.

9.9.2. Usando o `functools.lru_cache`

O decorador `functools.cache` é, na realidade, um mero invólucro em torno da antiga função `functools.lru_cache`, que é mais flexível e também compatível com versões anteriores ao Python 3.9.

A maior vantagem de `@lru_cache` é a possibilidade de limitar seu uso de memória através do parâmetro `maxsize`, que tem um default bastante pequeno: 128. Isso significa que o cache pode armazenar no máximo 128 resultados.

LRU é a sigla de *Least Recently Used* ("Usado Menos Recentemente"). Significa que registros que há algum tempo não são lidos, são descartados para dar lugar a novos itens.

Desde o Python 3.8, `lru_cache` pode ser aplicado de duas formas. Esta é a forma mais simples:

```
@lru_cache
def função_dispendiosa(a, b):
    ...
```

A outra forma é invocá-lo como uma função, com `()` (funciona desde o Python 3.2):

```
@lru_cache()
def função_dispendiosa(a, b):
    ...
```

Nos dois casos, os parâmetros default seriam utilizados. São eles:

maxsize=128

Estabelece o número máximo de registros a serem armazenados. Após o cache estar cheio, o registro menos recentemente usado é descartado, para dar lugar a cada novo item. Para um desempenho ótimo, `maxsize` deve ser uma potência de 2. Se você passar `maxsize=None`, a lógica LRU é desabilitada e o cache funciona mais rápido, mas os itens nunca são descartados, podendo levar a um consumo excessivo de memória. É assim que o `@functools.cache` funciona.

typed=False

Determina se os resultados de diferentes tipos de argumentos devem ser armazenados separadamente. Por exemplo, na configuração default, argumentos inteiros e de ponto flutuante considerados iguais são armazenados apenas uma vez. Assim, haverá apenas uma entrada para as chamadas `f(1)` e `f(1.0)`. Se `typed=True`, aqueles argumentos produziram registros diferentes, possivelmente armazenando resultados distintos.

Eis um exemplo de invocação de `@lru_cache` com parâmetros diferentes dos defaults:

```
@lru_cache(maxsize=2**20, typed=True)
def costly_function(a, b):
    ...
```

Vamos agora examinar outro decorador poderoso: `functools singledispatch`.

9.9.3. Funções genéricas com despacho único

Imagine que estamos criando uma ferramenta para depurar aplicações Web. Queremos gerar código HTML para tipos diferentes de objetos Python.

Poderíamos começar com uma função como essa:


```
import html

def htmlize(obj):
    content = html.escape(repr(obj))
    return f'<pre>{content}</pre>'
```

Isso funcionará para qualquer tipo de objeto, mas agora queremos estender a função para gerar HTML específico para determinados tipos. Alguns exemplos seriam:

str

Substituir os caracteres de mudança de linha na string por '
\n' e usar tags <p> em vez de <pre>.

int

Mostrar o número em formato decimal e hexadecimal (com um caso especial para bool).

list

Gerar uma lista em HTML, formatando cada item de acordo com seu tipo.

float e Decimal

Mostrar o valor como de costume, mas também na forma de fração (por que não?).

O comportamento que desejamos aparece no Exemplo 19.

Exemplo 19. htmlize() gera HTML adaptado para diferentes tipos de objetos

```
>>> htmlize({1, 2, 3}) ①
'<pre>{1, 2, 3}</pre>'
>>> htmlize(abs)
'<pre>&lt;built-in function abs&gt;</pre>'
>>> htmlize('Heimlich & Co.\n- a game') ②
'<p>Heimlich &amp; Co.<br/>\n- a game</p>'
>>> htmlize(42) ③
'<pre>42 (0x2a)</pre>'
>>> print(htmlize(['alpha', 66, {3, 2, 1}])) ④
<ul>
```

```

<li><p>alpha</p></li>
<li><pre>66 (0x42)</pre></li>
<li><pre>{1, 2, 3}</pre></li>
</ul>
>>> htmlize(True) ⑤
'<pre>True</pre>'
>>> htmlize(fractions.Fraction(2, 3)) ⑥
'<pre>2/3</pre>'
>>> htmlize(2/3) ⑦
'<pre>0.6666666666666666 (2/3)</pre>'
>>> htmlize(decimal.Decimal('0.02380952'))
'<pre>0.02380952 (1/42)</pre>'

```

- ① A função original é registrada para object, então ela serve para capturar e tratar todos os tipos de argumentos que não foram capturados pelas outras implementações.
- ② Objetos str também passam por escape de HTML, mas são cercados por `<p></p>`, com quebras de linha `
` inseridas antes de cada `'\n'`.
- ③ Um int é exibido nos formatos decimal e hexadecimal, dentro de um bloco `<pre></pre>`.
- ④ Cada item na lista é formatado de acordo com seu tipo, e a sequência inteira é apresentada como uma lista HTML.
- ⑤ Apesar de ser um subtipo de int, bool recebe um tratamento especial.
- ⑥ Mostra Fraction como uma fração.
- ⑦ Mostra float e Decimal com a fração equivalente aproximada.

Como não temos no Python a sobrecarga de métodos ao estilo de Java, não podemos simplesmente criar variações de `htmlize` com assinaturas diferentes para cada tipo de dado que queremos tratar de forma distinta. Uma solução possível em Python seria transformar `htmlize` em uma função de despacho, com uma cadeia de `if/elif/...` ou `match/case/...` chamando funções especializadas como `htmlize_str`, `htmlize_int`, etc. Isso não é extensível pelos usuários de nosso módulo, e é desajeitado: com o tempo, a função `htmlize` ficaria muito longa, e o acoplamento entre ela e as funções especializadas seria forte demais.

O decorador `functools singledispatch` permite que diferentes módulos contribuam para a solução geral, e que você forneça facilmente funções

especializadas, mesmo para tipos pertencentes a pacotes externos que não possam ser editados. Se você decorar uma função simples com `@singledispatch`, ela se torna o ponto de entrada para uma *função genérica*: um grupo de funções que executam a mesma operação de formas diferentes, dependendo do tipo do primeiro argumento. Este é o significado do termo *single dispatch* (despacho único). Se mais argumentos fossem usados para selecionar a função específica, teríamos despacho múltiplo (*multiple dispatch*), um recurso nativo em linguagens como Common Lisp, Julia e C#.

O Exemplo 20 mostra como implementar o despacho único.



`functools.singledispatch` existe desde o Python 3.4, mas só passou a suportar a sintaxe com dicas de tipo no Python 3.7. As últimas duas funções no Exemplo 20 ilustram a sintaxe que funciona em todas as versões de Python desde a 3.4.

Exemplo 20. `@singledispatch` cria uma `@htmlize.register` customizada, para juntar várias funções em uma função genérica

```
from functools import singledispatch
from collections import abc
import fractions
import decimal
import html
import numbers

@singledispatch ①
def htmlize(obj: object) -> str:
    content = html.escape(repr(obj))
    return f'<pre>{content}</pre>'

@htmlize.register ②
def _(text: str) -> str: ③
    content = html.escape(text).replace('\n', '<br/>\n')
    return f'<p>{content}</p>'

@htmlize.register ④
def _(seq: abc.Sequence) -> str:
    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
    return '<ul>\n<li>' + inner + '</li>\n</ul>'
```

```

@htmlize.register ⑤
def _(n: numbers.Integral) -> str:
    return f'<pre>{n} (0x{n:x})</pre>'

@htmlize.register ⑥
def _(n: bool) -> str:
    return f'<pre>{n}</pre>'

@htmlize.register(fractions.Fraction) ⑦
def _(x) -> str:
    frac = fractions.Fraction(x)
    return f'<pre>{frac.numerator}/{frac.denominator}</pre>'

@htmlize.register(decimal.Decimal) ⑧
@htmlize.register(float)
def _(x) -> str:
    frac = fractions.Fraction(x).limit_denominator()
    return f'<pre>{x} ({frac.numerator}/{frac.denominator})</pre>'

```

- ① @singledispatch marca a função base, que trata o tipo object.
- ② Cada função especializada é decorada com @«base».register.
- ③ O tipo do primeiro argumento passado durante a execução determina quando essa definição de função em particular será utilizada. O nome das funções especializadas é irrelevante; _ é uma boa escolha para deixar isso claro.^[7]
- ④ Registra uma nova função para cada tipo que precisa de tratamento especial, com uma dica de tipo correspondente no primeiro parâmetro.
- ⑤ As ABCs em numbers são úteis para uso em conjunto com singledispatch.^[8]
- ⑥ bool é um *subtipo-de* numbers.Integral, mas a lógica de singledispatch busca a implementação com o tipo correspondente mais específico, independente da ordem na qual eles aparecem no código.
- ⑦ Se você não quiser ou não puder adicionar dicas de tipo à função decorada, você pode passar o tipo para o decorador @«base».register. Essa sintaxe funciona em Python 3.4 ou posterior.
- ⑧ O decorador @«base».register devolve a função sem decoração, então é possível empilhá-los para registrar dois ou mais tipos na mesma implementação.^[9]

Sempre que possível, registre as funções especializadas para tratar ABCs (classes abstratas), como `numbers.Integral` e `abc.MutableSequence`, ao invés das implementações concretas como `int` e `list`. Isso permite ao seu código suportar uma variedade maior de tipos compatíveis. Por exemplo, uma extensão de Python pode fornecer alternativas para o tipo `int` com número fixo de bits como subclasses de `numbers.Integral`.^[10]



Usar ABCs ou `typing.Protocol` com `@singledispatch` permite que seu código suporte classes existentes ou futuras que sejam subclasses reais ou virtuais daquelas ABCs, ou que implementem aqueles protocolos. O uso de ABCs e o conceito de uma subclasse virtual são assuntos do Capítulo 13.

Uma qualidade notável do mecanismo de `singledispatch` é que você pode registrar funções especializadas em qualquer lugar do sistema, em qualquer módulo. Se mais tarde você adicionar um módulo com um novo tipo definido pelo usuário, é fácil acrescentar uma nova função específica para tratar aquele tipo. É possível também escrever funções customizadas para classes que você não escreveu e não pode modificar.

O `singledispatch` foi uma adição muito bem pensada à biblioteca padrão, e oferece outras facilidades que não cabem neste livro. Uma boa referência é a *PEP 443—Single-dispatch generic functions* [[fpy.li/pep443](https://www.python.org/peps/pep-0443/)] mas ela não menciona o uso de dicas de tipo, que foram criadas depois. A documentação do módulo `functools` foi melhorada e oferece um tratamento mais atualizado, com vários exemplos na seção referente ao `singledispatch` [[fpy.li/5y](https://docs.python.org/3/library/functools.html#functools.singledispatch)].



O `@singledispatch` não foi criado para trazer para Python a sobrecarga de métodos no estilo de Java. Uma classe única com muitas variações sobrecarregadas de um método é melhor que uma única função com uma longa sequência de blocos `if/elif/elif/elif`. Mas as duas soluções concentram responsabilidade demais em uma única unidade de código—uma classe ou a função. A vantagem de `@singledispatch` é seu suporte à extensão modular: cada módulo pode registrar uma função especializada para cada tipo suportado. Em um caso de uso realista, as implementações das funções genéricas não estariam todas no mesmo módulo, como ocorre no Exemplo 20.

Vimos decoradores recebendo argumentos, como `@lru_cache(maxsize=1024)` e o `htmlize.register(float)` criado por `@singledispatch` no Exemplo 20. A próxima seção mostra como criar decoradores com parâmetros.

9.10. Decoradores parametrizados

Ao analisar um decorador no código-fonte, Python passa a função decorada como primeiro argumento para a função do decorador. Mas como fazemos um decorador aceitar outros argumentos? A resposta é: criar uma fábrica de decoradores que recebe aqueles argumentos e devolve um decorador, que é então aplicado à função a ser decorada. Complicado? Sim. Mas vamos começar com um exemplo baseado no decorador mais simples que vimos: `register` no Exemplo 21.

Exemplo 21. O módulo `registration.py` resumido, do Exemplo 2, repetido aqui por conveniência

```
registry = []

def register(func):
    print(f'running register({func})')
    registry.append(func)
    return func

@register
def f1():
    print('running f1()')

print('running main()')
print('registry ->', registry)
f1()
```

9.10.1. Um decorador de registro parametrizado

Para tornar mais fácil a habilitação ou desabilitação do registro executado por `register`, faremos esse último aceitar um parâmetro opcional `active` que, se `False`, não registra a função decorada. Conceitualmente, a nova função `register` não é um decorador, mas uma fábrica de decoradores. Quando chamada, ela devolve o decorador que será realmente aplicado à função alvo.

Exemplo 22. Para aceitar parâmetros, o novo decorador register precisa ser invocado como uma função

```
registry = set() ①

def register(active=True): ②
    def decorate(func): ③
        print('running register'
              f'(active={active})->decorate({func})')
        if active: ④
            registry.add(func)
        else:
            registry.discard(func) ⑤

        return func ⑥
    return decorate ⑦

@register(active=False) ⑧
def f1():
    print('running f1()')

@register() ⑨
def f2():
    print('running f2()')

def f3():
    print('running f3()')
```

- ① registry é agora um set, tornando mais rápido acrescentar ou remover funções.
- ② register recebe um argumento nomeado opcional.
- ③ A função interna decorate é o verdadeiro decorador; observe como ela aceita uma função como argumento.
- ④ Registra func apenas se o argumento active (obtido da clausura) for True.
- ⑤ Se active é falso, remove a função (sem efeito se a função não está no set).
- ⑥ Como decorate é um decorador, tem que devolver uma função.
- ⑦ register é nossa fábrica de decoradores, então devolve decorate.

- ⑧ A fábrica `@register` precisa ser invocada como uma função, com os parâmetros desejados.
- ⑨ Mesmo se nenhum parâmetro for passado, ainda assim `register` deve ser invocada como uma função: `@register()` para criar e devolver o verdadeiro decorador, `decorate`.

O ponto central aqui é que `register()` devolve `decorate`, que então é aplicado à função decorada.

O código do Exemplo 22 está em um módulo *registration_param.py*. Se o importarmos, veremos o seguinte:

```
>>> import registration_param
running register(active=False)->decorate(<function f1 at 0x10063c1e0>)
running register(active=True)->decorate(<function f2 at 0x10063c268>)
>>> registration_param.registry
[<function f2 at 0x10063c268>]
```

Veja como apenas a função `f2` aparece no `registry`; `f1` não aparece porque `active=False` foi passado para a fábrica de decoradores `register`, então o `decorate` aplicado a `f1` não adiciona essa função a `registry`.

Se, ao invés de usar a sintaxe `@`, usarmos `register` como uma função regular, a sintaxe necessária para decorar uma função `f` seria `register()(f)`, para inserir `f` ao `registry`, ou `register(active=False)(f)`, para não inseri-la (ou removê-la). Veja o Exemplo 23 para uma demonstração da adição e remoção de funções do `registry`.

Exemplo 23. Usando o módulo `registration_param` listado no Exemplo 22

```
>>> from registration_param import *
running register(active=False)->decorate(<function f1 at 0x10073c1e0>)
running register(active=True)->decorate(<function f2 at 0x10073c268>)
>>> registry ①
{<function f2 at 0x10073c268>}
>>> register()(f3) ②
running register(active=True)->decorate(<function f3 at 0x10073c158>)
<function f3 at 0x10073c158>
>>> registry ③
```



```
{<function f3 at 0x10073c158>, <function f2 at 0x10073c268>}
>>> register(active=False)(f2) ④
running register(active=False)->decorate(<function f2 at 0x10073c268>)
<function f2 at 0x10073c268>
>>> registry ⑤
{<function f3 at 0x10073c158>}
```

- ① Quando o módulo é importado, f2 é inserida no registry.
- ② A expressão register() devolve decorate, que então é aplicado a f3.
- ③ A linha anterior adicionou f3 ao registry.
- ④ Essa chamada remove f2 do registry.
- ⑤ Confirma que apenas f3 permanece no registry.

O funcionamento de decoradores parametrizados é bastante complexo, e esse que acabamos de discutir é mais simples que a maioria. Decoradores parametrizados em geral substituem a função decorada, e sua construção exige um nível adicional de aninhamento. Vamos agora explorar a arquitetura de uma dessas pirâmides de funções.

9.10.2. Um decorador parametrizado de cronometragem

Nesta seção vamos revisitar o decorador clock, acrescentando um recurso: os usuários podem passar uma string para formatar o relatório sobre a função cronometrada. Veja o Exemplo 24.



Para simplificar, o Exemplo 24 está baseado na implementação inicial de clock no Exemplo 14, e não na versão melhorada do Exemplo 16 que usa `@functools.wraps`, evitando mais um nível de aninhamento de funções.

Exemplo 24. Módulo clockdeco_param.py: o decorador clock parametrizado

```
import time

DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}' ①

def clock(fmt=DEFAULT_FMT): ②
    def decorate(func): ③
        def clocked(*_args): ④
            t0 = time.perf_counter()
            _result = func(*_args) ⑤
            elapsed = time.perf_counter() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args) ⑥
            result = repr(_result) ⑦
            print(fmt.format(**locals())) ⑧
            return _result ⑨
        return clocked ⑩
    return decorate ⑪

if __name__ == '__main__':

    @clock() ⑫
    def snooze(seconds):
        time.sleep(seconds)

    for i in range(3):
        snooze(.123)
```

- ① Formato padrão da saída citando variáveis locais da função clocked.
- ② clock é a nossa fábrica de decoradores parametrizados.
- ③ decorate é o verdadeiro decorador.
- ④ clocked envolve a função decorada.
- ⑤ _result é o resultado real da função decorada.
- ⑥ _args armazena os verdadeiros argumentos de clocked, enquanto args é a str usada para exibição.
- ⑦ result é a str que representa _result, para uso no formato.

- ⑧ Usar `**locals()` aqui permite que qualquer variável local de `clocked` seja referenciada em `fmt`.^[11]
- ⑨ `clocked` vai substituir a função decorada, então ela deve devolver o mesmo que aquela função devolve.
- ⑩ `decorate` devolve `clocked`.
- ⑪ `clock` devolve `decorate`.
- ⑫ Nesse auto-teste, `clock()` é chamado sem argumentos, então o decorador aplicado usará o formato default, `str`.

Se você rodar o Exemplo 24 no console, o resultado é o seguinte:

```
$ python3 clockdeco_param.py
[0.12412500s] snooze(0.123) -> None
[0.12411904s] snooze(0.123) -> None
[0.12410498s] snooze(0.123) -> None
```

Para exercitar a nova funcionalidade, veremos mais dois módulos que usam o `clockdeco_param`, o Exemplo 25 e o Exemplo 26, e as saídas que eles produzem.

Exemplo 25. `clockdeco_param_demo1.py`

```
import time
from clockdeco_param import clock

@clock('{name}: {elapsed}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

Saída do Exemplo 25:

```
$ python3 clockdeco_param_demo1.py
snooze: 0.12414693832397461s
snooze: 0.1241159439086914s
snooze: 0.12412118911743164s
```

Exemplo 26. clockdeco_param_demo2.py

```
import time
from clockdeco_param import clock

@clock('{name}({args}) dt={elapsed:0.3f}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

Saída do Exemplo 26:

```
$ python3 clockdeco_param_demo2.py
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
```



Lennart Regebro—um dos revisores técnicos da primeira edição—argumenta que seria melhor programar decoradores como classes implementando `__call__`, e não como funções, como os exemplos neste capítulo. Concordo que aquela abordagem é melhor para decoradores não-triviais. Mas para explicar a ideia básica desse recurso da linguagem, funções são mais fáceis de entender. Para conhecer técnicas mais robustas de criação de decoradores, veja as referências na Seção 9.12, especialmente o blog de Graham Dumpleton e o módulo `wrapt`.

Agora veremos um exemplo no estilo recomendado por Regebro e Dumpleton.

9.10.3. Um decorador de cronometragem em forma de classe

Como um último exemplo, o Exemplo 27 mostra a implementação de um decorador parametrizado `clock`, programado como uma classe com `__call__`. Compare o Exemplo 24 com o Exemplo 27. Qual você prefere?

Exemplo 27. Módulo `clockdeco_cls.py`: decorador parametrizado `clock`, implementado como uma classe

```
import time

DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'

class clock: ①

    def __init__(self, fmt=DEFAULT_FMT): ②
        self.fmt = fmt

    def __call__(self, func): ③
        def clocked(*_args):
            t0 = time.perf_counter()
            _result = func(*_args) ④
            elapsed = time.perf_counter() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args)
            result = repr(_result)
            print(self.fmt.format(**locals()))
            return _result
        return clocked
```

- ① Ao invés de uma função externa `clock`, a classe `clock` é nossa fábrica de decoradores parametrizados. Escrevi `clock` com `c` minúsculo para deixar claro que essa implementação substitui exatamente a função `clock` no Exemplo 24.
- ② O argumento passado em `clock(my_format)` é atribuído ao parâmetro `fmt` aqui. O construtor da classe devolve uma instância de `clock`, com `my_format` armazenado em `self.fmt`.
- ③ `__call__` torna a instância de `clock` invocável. Quando chamada, a instância substitui a função decorada com `clocked`.
- ④ `clocked` envolve a função decorada.

Isso encerra nossa exploração dos decoradores de função. Veremos os decoradores de classe no «Capítulo 24» [fpy.li/24] (vol.3).

9.11. Resumo do capítulo

Percorremos um terreno difícil neste capítulo. Tentei tornar a jornada tão suave quanto possível, mas entramos nos domínios da meta-programação, onde nada é simples.

Partimos de um decorador simples `@register`, sem uma função interna, e terminamos com um `@clock()` parametrizado envolvendo dois níveis de funções aninhadas.

Decoradores de registro, apesar de serem essencialmente simples, têm aplicações reais nos frameworks Python. Vamos aplicar a ideia de registro em uma implementação do padrão de projeto Estratégia, no Capítulo 10.

Entender como os decoradores realmente funcionam exigiu falar da diferença entre *momento de importação* e *momento de execução*. Então mergulhamos no escopo de variáveis, clausuras e a nova declaração `nonlocal`. Dominar as clausuras e `nonlocal` é valioso não apenas para criar decoradores, mas também para escrever programas orientados a eventos para GUIs ou E/S assíncrona com *callbacks*, e para adotar um estilo funcional quando fizer sentido.

Decoradores parametrizados quase sempre implicam em pelo menos dois níveis de funções aninhadas, talvez mais se você quiser usar `@functools.wraps`, e produzir um decorador com um suporte melhor a técnicas mais avançadas. Uma dessas técnicas é o empilhamento de decoradores, que vimos no Exemplo 18. Para decoradores mais sofisticados, uma implementação baseada em classes pode ser mais fácil de ler e manter.

Como exemplos de decoradores parametrizados na biblioteca padrão, visitamos os poderosos `@cache` e `@singledispatch`, do módulo `functools`.

9.12. Para saber mais

O item #26 do livro *Effective Python*, 2nd ed. [fpy.li/effectpy] (Addison-Wesley), de Brett Slatkin, trata das melhores práticas para decoradores de função, e recomenda sempre usar `functools.wraps`—que vimos no Exemplo 16. Como eu queria manter o código o mais simples possível, não segui o excelente conselho de Slatkin em todos os exemplos.

Graham Dumpleton tem, em seu blog, uma série de posts abrangentes [fpy.li/9-5] sobre técnicas para implementar decoradores bem comportados, começando com *How you implemented your Python decorator is wrong* (A forma como você implementou seu decorador em Python está errada) [fpy.li/9-6]. Seus conhecimentos sobre o tema também aparecem no módulo `wrapt` [fpy.li/9-7], que ele escreveu para simplificar a implementação de decoradores e invólucros (*wrappers*) dinâmicos de função, que suportam introspecção e se comportam de forma correta quando decorados novamente, quando aplicados a métodos e quando usados como descritores de atributos (o «Capítulo 23» [fpy.li/23] (vol.3) é sobre descritores).

Metaprogramming [fpy.li/9-8], o capítulo 9 do *Python Cookbook*, 3ª ed. de David Beazley e Brian K. Jones (O'Reilly), tem várias receitas ilustrando desde decoradores elementares até alguns muito sofisticados, incluindo um que pode ser invocado como um decorador regular ou como uma fábrica de decoradores, por exemplo, `@clock` ou `@clock()`. É a *Recipe 9.6. Defining a Decorator That Takes an Optional Argument* (Definindo um Decorador Que Recebe um Argumento Opcional) daquele livro de receitas.

Michele Simionato criou *decorator* [fpy.li/9-9], um pacote para "simplificar o uso de decoradores para o programador comum, e popularizar os decoradores através da apresentação de vários exemplos não-triviais", de acordo com sua documentação.

Criada quando os decoradores ainda eram um recurso novo no Python, a página wiki *Python Decorator Library* [fpy.li/9-10] tem dezenas de exemplos. Como começou há muitos anos, algumas das técnicas apresentadas foram suplantadas, mas ela ainda é uma excelente fonte de inspiração.

Closures in Python [fpy.li/9-11] é um post curto de Fredrik Lundh, explicando a terminologia das clausuras.

A *PEP 3104—Access to Names in Outer Scopes* [fpy.li/9-12] (Acesso a Nomes em Escopos Externos) descreve a introdução da declaração `nonlocal`. Ela também inclui uma excelente revisão de como essa questão foi resolvida em outras linguagens dinâmicas (Perl, Ruby, JavaScript, etc.) e os prós e contras das opções de design disponíveis para Python.

Em um nível mais teórico, a *PEP 227—Statically Nested Scopes* [fpy.li/9-13] (Escopos estaticamente Aninhados_) documenta a introdução do escopo léxico como um opção no Python 2.1 e como padrão no Python 2.2, explicando a justificativa e as opções de design para a implementação de clausuras no Python.

A *PEP 443* [fpy.li/9-14] traz a justificativa e uma descrição detalhada do mecanismo de funções genéricas de despacho único. Um post de Guido van Rossum de março de 2005 *Five-Minute Multimethods in Python* [fpy.li/9-15] (Multi-métodos de cinco minutos em Python), mostra os passos para uma implementação de funções genéricas (também chamadas multi-métodos) usando decoradores. O código de multi-métodos de Guido é interessante, mas é apenas um exemplo didático. Para conhecer uma implementação de funções genéricas de despacho múltiplo moderna e pronta para uso em produção, veja a *Reg* [fpy.li/9-16] de Martijn Faassen—autor de *Morepath* [fpy.li/9-17], um framework Web guiado por modelos e orientado a REST.

Ponto de vista

Escopo dinâmico versus escopo léxico

O projetista de qualquer linguagem que contenha funções de primeira classe se depara com essa questão: sendo um objeto de primeira classe, uma função é definida dentro de um determinado escopo, mas pode ser invocada em outros escopos. O problema é: como avaliar as variáveis livres? A solução mais simples de implementar chama-se "escopo dinâmico". Isso significa que variáveis livres são avaliadas olhando para dentro do ambiente onde a função é invocada.

Se Python tivesse escopo dinâmico e não tivesse clausuras, poderíamos improvisar `avg` (similar ao Exemplo 8) desta forma:


```

>>> ### esta não é uma sessão real de Python! ###
>>> avg = make_averager()
>>> series = [] ①
>>> avg(10)
10.0
>>> avg(11) ②
10.5
>>> avg(12)
11.0
>>> series = [1] ③
>>> avg(5)
3.0

```

- ① Antes de usar `avg`, precisamos definir por nós mesmos `series = []`, então precisamos saber que `averager` (dentro de `make_averager`) se refere a uma lista chamada `series`.
- ② Por trás da cortina, `series` acumula os valores cuja média será calculada.
- ③ Quando `series = [1]` é executada, a lista anterior é perdida. Isso poderia ocorrer por acidente, ao computar duas médias cumulativas independentes ao mesmo tempo.

O ideal é que funções sejam opacas, sua implementação invisível para os usuários. Mas com escopo dinâmico, se a função usa variáveis livres, o programador precisa saber do funcionamento interno da função, para poder preparar um ambiente onde ela execute corretamente. Após anos lutando com a linguagem de preparação de documentos LaTeX, o excelente livro *Practical LaTeX* (LaTeX Prático), de George Grätzer (Springer), me ensinou que as variáveis no LaTeX usam escopo dinâmico. Por isso me confundiam tanto!

O Lisp do Emacs também usa escopo dinâmico, pelo menos como default. Veja *Dynamic Binding* [fpy.li/9-18] (Vinculação Dinâmica) no manual do Emacs para uma breve explicação.

O escopo dinâmico é mais fácil de implementar, e essa foi provavelmente a razão de John McCarthy ter tomado esse caminho quando criou o Lisp, a primeira linguagem a ter funções de primeira classe. O texto de Paul

Graham, *The Roots of Lisp* [fpy.li/9-19] (As Raízes do Lisp) é uma explicação acessível do artigo original de John McCarthy sobre a linguagem Lisp, *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I* [fpy.li/9-20] (Funções Recursivas de Expressões Simbólicas e Sua Computação por Máquina). O artigo de McCarthy é uma obra prima no nível da Nona Sinfonia de Beethoven. Paul Graham o traduziu do jargão matemático para um inglês mais compreensível e código executável.

O comentário de Paul Graham explica como o escopo dinâmico é complexo. Citando *The Roots of Lisp*:

É um testemunho eloquente dos perigos do escopo dinâmico, que mesmo o primeiro exemplo de funções de ordem superior em Lisp estivesse errado por causa dele. Talvez, em 1960, McCarthy não estivesse inteiramente ciente das implicações do escopo dinâmico, que continuou presente nas implementações de Lisp por um tempo surpreendentemente longo—até Sussman e Steele desenvolverem o Scheme, em 1975. O escopo léxico não complica demais a definição de eval, mas pode tornar mais difícil escrever compiladores.

Atualmente, o escopo léxico é o padrão: variáveis livres são avaliadas considerando o ambiente onde a função foi definida. O escopo léxico complica a implementação de linguagens com funções de primeira classe, pois requer o suporte a clausuras. Por outro lado, o escopo léxico torna o código-fonte mais fácil de ler. A maioria das linguagens inventadas desde o Algol tem escopo léxico. Uma exceção notável é o JavaScript, onde a variável especial `this` é confusa, pois pode ter escopo léxico ou dinâmico, dependendo da forma como o código for escrito [fpy.li/9-21] (EN).

Por muitos anos, o `lambda` de Python não implementava clausuras, contribuindo para a má fama deste recurso entre os fãs da programação funcional na blogosfera. Isso foi resolvido no Python 2.2 (de dezembro de 2001), mas a blogosfera nunca perdoa. Desde então, `lambda` é triste apenas devido à sua sintaxe limitada.

Os decoradores de Python e o padrão de projeto Decorator

Os decoradores de função de Python se encaixam na descrição geral dos decoradores de Gamma et al. em *Padrões de Projeto*: "Acrescenta responsabilidades adicionais a um objeto de forma dinâmica. Decoradores fornecem uma alternativa flexível à criação de subclasses para estender funcionalidade."

Ao nível da implementação, os decoradores de Python não lembram o padrão de projeto decorador clássico, mas é possível fazer uma analogia. No padrão de projeto, Decorador e Componente são classes abstratas. Uma instância de um decorador concreto envolve uma instância de um componente concreto para adicionar comportamentos a ela. Citando *Padrões de Projeto*:

O decorador se adapta à interface do componente decorado, assim sua presença é transparente para os clientes do componente. O decorador encaminha requisições para o componente e pode executar ações adicionais (tal como desenhar uma borda) antes ou depois do encaminhamento. A transparência permite aninhar decoradores de forma recursiva, possibilitando assim um número ilimitado de responsabilidades adicionais. (p. 175 da edição em inglês)

No Python, a função decoradora faz o papel de uma subclasse concreta de Decorador, e a função interna que ela devolve é uma instância do decorador. A função devolvida envolve a função a ser decorada, que é análoga ao componente no padrão de projeto. A função devolvida é transparente, pois se adapta à interface do componente (ao aceitar os mesmos argumentos). Adaptando a última frase da citação: "A transparência permite empilhar decoradores, possibilitando assim um número ilimitado de comportamentos adicionais".

Veja que não estou sugerindo que decoradores de função devam ser usados para implementar o padrão decorador em programas Python. Pode até ser feito em situações específicas, mas em geral o padrão decorador é melhor implementado com classes representando o decorador e os componentes que ela vai envolver.

[1] GoF se refere ao livro *Design Patterns* (traduzido no Brasil como Padrões de Projeto). Seus autores ficaram conhecidos como a *Gang of Four* (Gangue dos Quatro).

[2] NT: Adotamos a tradução "clausura" para "closure". O termo em inglês é pronunciado "clôujure", e o nome da linguagem Clojure brinca com esse fato. Alguns autores usam "fechamento", mas esta é uma tradução de "closure" no contexto da teoria dos conjuntos que não tem relação com "closure" na teoria de linguagens de programação. Gosto da palavra clausura por uma analogia cultural: em alguns conventos, a clausura é um espaço fechado onde freiras vivem em isolamento. Suas memórias são seu único vínculo com o exterior, mas elas retratam o mundo do passado. Em programação, uma clausura é um espaço isolado onde a função tem acesso a variáveis que existiam quando a própria função foi criada, variáveis de um escopo que não existe mais, preservadas apenas na memória clausura.

[3] Se você substituir "função" por "classe" na sentença anterior, o resultado é uma descrição resumida do papel de um decorador de classe, assunto que veremos no «Capítulo 24» [fpy.li/24] (vol.3).

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

[5] Python não tem um escopo global de programa, apenas escopos globais de módulos.

[6] Esclarecendo, isso não é um erro de ortografia: *memoization* [fpy.li/9-2] é um termo da ciência da computação vagamente relacionado a "memorização", mas não idêntico.

[7] Infelizmente, o Mypy 0.770 reclama quando vê múltiplas funções com o mesmo nome.

[8] Apesar do alerta em «Seção 8.5.7.1» [fpy.li/8g] (vol.1), as ABCs de *numbers* não foram descontinuadas, e você as encontra em código de Python 3.

[9] Talvez algum dia seja possível expressar isso com um único `@htmlize.register` sem parâmetros, e uma dica de tipo usando `Union`. Mas quando tentei, Python gerou um `TypeError` com uma mensagem dizendo que `Union` não é uma classe. Então, apesar da *sintaxe* da PEP 484 ser suportada, a *semântica* ainda não chegou lá.

[10] NumPy, por exemplo, implementa vários tipos de números inteiros e de ponto flutuante [fpy.li/9-3] (EN) em formatos voltados para a arquitetura da máquina.

[11] O revisor técnico Miroslav Šedivý observou: "Isso também quer dizer que analisadores de código-fonte (*linters*) vão reclamar de variáveis não utilizadas, pois eles tendem a ignorar o uso de `locals()`." Sim, esse é mais um exemplo de como ferramentas estáticas de checagem desencorajam o uso dos recursos dinâmicos de Python que me atraíram (e a incontáveis outros programadores) quando adotei a linguagem. Para deixar o *linter* feliz, eu poderia escrever o nome de cada variável duas vezes na chamada: `fmt.format(elapsed=elapsed, name=name, args=args, result=result)`. Prefiro não fazer isso. Se você usa ferramentas estáticas de checagem, é importante saber quando ignorá-las.

Capítulo 10. Padrões de projetos com funções de primeira classe

Conformidade a padrões não é medida de virtude.^[1]

— Ralph Johnson, co-autor do clássico "Padrões de Projetos"

Em engenharia de software, um *padrão de projeto* [fpy.li/5z] é uma receita genérica para solucionar um problema de design comum. Não é preciso conhecer padrões de projeto para acompanhar esse capítulo, vou explicar os padrões usados nos exemplos.

O uso de padrões de projeto em programação foi popularizado pelo livro seminal *Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos* (Addison-Wesley), de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides—também conhecidos como *the Gang of Four* (A Gangue dos Quatro) ou pela sigla *GoF*. O livro é um catálogo de 23 padrões, apresentados como arranjos de classes e exemplificados com código em C++, mas considerados úteis também em outras linguagens orientadas a objetos.

Apesar dos padrões de projeto serem independentes da linguagem, isso não significa que todo padrão se aplica a todas as linguagens. Por exemplo, o «Capítulo 17» [fpy.li/17] (vol.3) vai mostrar que não faz sentido implementar a receita do padrão Iterador (*Iterator*) [fpy.li/10-2] em Python, pois esse padrão está embutido na linguagem e pronto para ser usado, na forma de geradores—que não precisam de classes para funcionar, e exigem menos código que a receita do livro clássico.

Na introdução da obra, os autores reconhecem que a linguagem usada na implementação determina quais padrões são relevantes:

A escolha da linguagem de programação é importante, pois ela influencia nosso ponto de vista. Nossos padrões supõe uma linguagem com recursos equivalentes aos de Smalltalk e do C++—e essa escolha determina o que pode e o que não pode ser facilmente implementado. Se tivéssemos presumido uma linguagem procedural, poderíamos ter incluído padrões de projetos chamados "Herança", "Encapsulamento" e "Polimorfismo". Da mesma forma,

alguns de nossos padrões são suportados diretamente por linguagens orientadas a objetos menos conhecidas. CLOS, por exemplo, tem múltiplos métodos, reduzindo a necessidade de um padrão como o Visitante.^[2]

Em sua apresentação de 1996, *Design Patterns in Dynamic Languages* [fpy.li/norvigdp] (Padrões de Projetos em Linguagens Dinâmicas), Peter Norvig afirma que 16 dos 23 padrões do livro original se tornam "invisíveis ou mais simples" em uma linguagem dinâmica (slide 9). Ele se refere às linguagens Lisp e Dylan, mas vários recursos dinâmicos citados também existem em Python. Em especial, em uma linguagem que oferece funções de primeira classe, Norvig sugere repensar os padrões clássicos conhecidos como *Strategy* (Estratégia), *Command* (Comando), *Template Method* (Método Gabarito) e *Visitor* (Visitante).

O objetivo desse capítulo é mostrar como, em certos casos, funções podem realizar o mesmo trabalho de classes, com menos código e mais clareza. Vamos refatorar uma implementação de Estratégia usando funções como objetos, removendo muito código redundante. Vamos também discutir uma abordagem similar para simplificar o padrão Comando.

10.1. Novidades neste capítulo

Movi este capítulo para o final da Parte II, para poder então aplicar o decorador de registro na Seção 10.3, e também usar dicas de tipo nos exemplos. A maior parte das dicas de tipo usadas nesse capítulo são simples, e ajudam na legibilidade.

10.2. Estudo de caso: refatorando Estratégia

Estratégia é um bom exemplo de um padrão de projeto que pode ser mais simples em Python, usando funções como objetos de primeira classe. Na próxima seção vamos descrever e implementar Estratégia usando a estrutura "clássica" descrita em *Padrões de Projetos*. Se você estiver familiarizado com o padrão original, pode pular direto para Seção 10.2.2, onde refatoramos o código usando funções, eliminando várias linhas.

10.2.1. Estratégia clássica

O diagrama de classes UML na Figura 1 retrata um arranjo de classes exemplificando o padrão Estratégia.

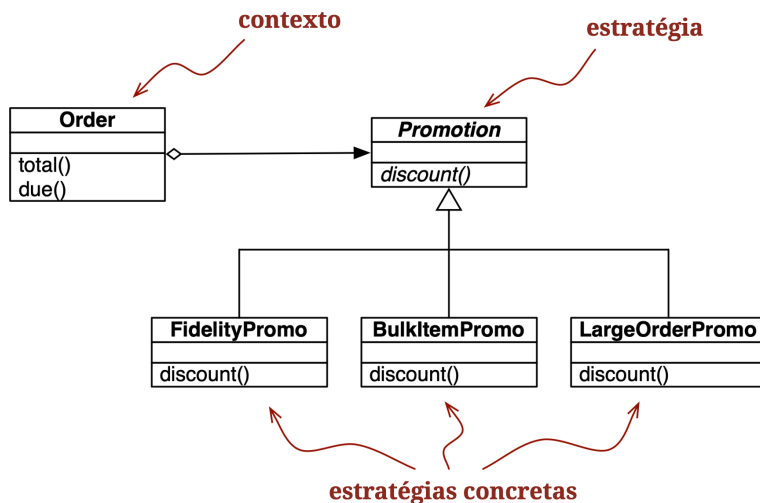


Figura 1. Diagrama de classes UML para calcular descontos em pedidos, com o padrão de projeto Estratégia.

O padrão Estratégia é resumido assim em *Padrões de Projetos*:

Define uma família de algoritmos, encapsula cada um deles, e os torna intercambiáveis. Estratégia permite que o algoritmo varie de forma independente dos clientes que o usam.

Um exemplo claro de Estratégia, aplicado ao domínio do ecommerce, é o cálculo de descontos em pedidos de acordo com os atributos do cliente ou pela inspeção dos itens do pedido.

Considere uma loja online com as seguintes regras para descontos:

- Clientes com 1.000 ou mais pontos de fidelidade recebem um desconto global de 5% por pedido.
- Um desconto de 10% é aplicado a cada item com 20 ou mais unidades no mesmo pedido.
- Pedidos com 10 ou mais itens diferentes têm um desconto global de 7%.

Para simplificar, vamos assumir que apenas um desconto pode ser aplicado a cada pedido.

O diagrama de classes UML para o padrão Estratégia aparece na Figura 1. Seus participantes são:

Contexto (*Context*)

Oferece um serviço delegando parte do processamento para componentes intercambiáveis, que implementam algoritmos alternativos. Neste exemplo, o contexto é uma classe `Order`, configurada para aplicar um desconto promocional de acordo com um algoritmo entre vários possíveis.

Estratégia (*Strategy*)

A interface comum dos componentes que implementam diferentes algoritmos. No nosso exemplo, esse papel cabe a uma classe abstrata chamada `Promotion`.

Estratégia concreta (*Concrete strategy*)

Cada uma das subclasses concretas de Estratégia. `FidelityPromo`, `BulkPromo`, e `LargeOrderPromo` são as três estratégias concretas implementadas.

O código no Exemplo 1 segue o modelo da Figura 1. Como descrito em *Padrões de Projetos*, a estratégia concreta é escolhida pelo cliente da classe de contexto. No nosso exemplo, antes de instanciar um pedido, o sistema deveria, de alguma forma, selecionar a estratégia de desconto promocional e passá-la para o construtor de `Order`. A seleção da estratégia está fora do escopo do padrão.

Exemplo 1. Implementação da classe `Order` com estratégias de desconto intercambiáveis

```
from abc import ABC, abstractmethod
from collections.abc import Sequence
from decimal import Decimal
from typing import NamedTuple, Optional

class Customer(NamedTuple):
    name: str
    fidelity: int
```



```

class LineItem(NamedTuple):
    product: str
    quantity: int
    price: Decimal

    def total(self) -> Decimal:
        return self.price * self.quantity

class Order(NamedTuple): # the Context
    customer: Customer
    cart: Sequence[LineItem]
    promotion: Optional['Promotion'] = None

    def total(self) -> Decimal:
        totals = (item.total() for item in self.cart)
        return sum(totals, start=Decimal(0))

    def due(self) -> Decimal:
        if self.promotion is None:
            discount = Decimal(0)
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'

class Promotion(ABC): # the Strategy: an abstract base class
    @abstractmethod
    def discount(self, order: Order) -> Decimal:
        """Return discount as a positive dollar amount"""

class FidelityPromo(Promotion): # first Concrete Strategy
    """5% discount for customers with 1000 or more fidelity points"""

    def discount(self, order: Order) -> Decimal:
        rate = Decimal('0.05')
        if order.customer.fidelity >= 1000:
            return order.total() * rate
        return Decimal(0)

```

```

class BulkItemPromo(Promotion): # second Concrete Strategy
    """10% discount for each LineItem with 20 or more units"""

    def discount(self, order: Order) -> Decimal:
        discount = Decimal(0)
        for item in order.cart:
            if item.quantity >= 20:
                discount += item.total() * Decimal('0.1')
        return discount

class LargeOrderPromo(Promotion): # third Concrete Strategy
    """7% discount for orders with 10 or more distinct items"""

    def discount(self, order: Order) -> Decimal:
        distinct_items = {item.product for item in order.cart}
        if len(distinct_items) >= 10:
            return order.total() * Decimal('0.07')
        return Decimal(0)

```

Observe que no Exemplo 1, programei `Promotion` como uma classe base abstrata (ABC), para usar o decorador `@abstractmethod` e deixar o padrão mais explícito.

O Exemplo 2 apresenta os doctests usados para demonstrar e verificar a operação de um módulo implementando as regras descritas anteriormente.

Exemplo 2. Usos da classe `Order` com a aplicação de diferentes promoções

```

>>> joe = Customer('John Doe', 0) ①
>>> ann = Customer('Ann Smith', 1100)
>>> cart = (LineItem('banana', 4, Decimal('.5')), ②
...         LineItem('apple', 10, Decimal('1.5')),
...         LineItem('watermelon', 5, Decimal(5)))
>>> Order(joe, cart, FidelityPromo()) ③
<Order total: 42.00 due: 42.00>
>>> Order(ann, cart, FidelityPromo()) ④
<Order total: 42.00 due: 39.90>
>>> banana_cart = (LineItem('banana', 30, Decimal('.5')), ⑤
...                LineItem('apple', 10, Decimal('1.5')))
>>> Order(joe, banana_cart, BulkItemPromo()) ⑥

```

```

<Order total: 30.00 due: 28.50>
>>> long_cart = tuple(LineItem(str(sku), 1, Decimal(1))) ⑦
...               for sku in range(10))
>>> Order(joe, long_cart, LargeOrderPromo()) ⑧
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, LargeOrderPromo())
<Order total: 42.00 due: 42.00>

```

- ① Dois clientes: joe tem 0 pontos de fidelidade, ann tem 1.100.
- ② Um carrinho de compras com três itens.
- ③ A promoção FidelityPromo não dá qualquer desconto para joe.
- ④ ann recebe um desconto de 5% porque tem pelo menos 1.000 pontos.
- ⑤ O banana_cart contém 30 unidade do produto "banana" e 10 maçãs.
- ⑥ Graças à BulkItemPromo, joe recebe um desconto de \$1,50 no preço das bananas.
- ⑦ O long_cart tem 10 itens diferentes, cada um custando \$1,00.
- ⑧ joe recebe um desconto de 7% no pedido total, por causa da LargerOrderPromo.

O Exemplo 1 funciona, mas podemos implementar a mesma funcionalidade com menos linhas de código usando funções como objetos. Vejamos como.

10.2.2. Estratégia baseada em funções

Cada estratégia concreta no Exemplo 1 é uma classe com um método: `discount`. Além disso, as instâncias de estratégia não tem estado (nenhum atributo de instância). Você poderia dizer que elas se parecem muito com funções simples, e estaria certa. O Exemplo 3 é uma refatoração do Exemplo 1, trocando as estratégias concretas por funções simples e removendo a ABC Promo. São necessários apenas alguns pequenos ajustes na classe `Order`.^[3]

Exemplo 3. A classe `Order` com as estratégias implementadas como funções

```

from collections.abc import Sequence
from dataclasses import dataclass
from decimal import Decimal
from typing import Optional, Callable, NamedTuple

```

```

class Customer(NamedTuple):
    name: str
    fidelity: int

class LineItem(NamedTuple):
    product: str
    quantity: int
    price: Decimal

    def total(self):
        return self.price * self.quantity

@dataclass(frozen=True)
class Order: # the Context
    customer: Customer
    cart: Sequence[LineItem]
    promotion: Optional[Callable[['Order'], Decimal]] = None ①

    def total(self) -> Decimal:
        totals = (item.total() for item in self.cart)
        return sum(totals, start=Decimal(0))

    def due(self) -> Decimal:
        if self.promotion is None:
            discount = Decimal(0)
        else:
            discount = self.promotion(self) ②
        return self.total() - discount

    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'

③

def fidelity_promo(order: Order) -> Decimal: ④
    """5% discount for customers with 1000 or more fidelity points"""
    if order.customer.fidelity >= 1000:
        return order.total() * Decimal('0.05')
    return Decimal(0)

```

```
def bulk_item_promo(order: Order) -> Decimal:
    """10% discount for each LineItem with 20 or more units"""
    discount = Decimal(0)
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * Decimal('0.1')
    return discount

def large_order_promo(order: Order) -> Decimal:
    """7% discount for orders with 10 or more distinct items"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * Decimal('0.07')
    return Decimal(0)
```

- ① Essa dica de tipo diz: `promotion` pode ser `None`, ou pode ser um invocável que recebe uma `Order` como argumento e devolve um `Decimal`.
- ② Para calcular o desconto, invocamos `self.promotion`, passando `self` como argumento. Veja a razão disso logo abaixo.
- ③ Nenhuma classe abstrata.
- ④ Cada estratégia é uma função.



Por que `self.promotion(self)`?

Na classe `Order`, `promotion` não é um método. É um atributo de instância que por acaso é invocável. Então a primeira parte da expressão, `self.promotion`, busca aquele invocável. Mas, ao invocá-lo, precisamos fornecer uma instância de `Order`, que neste caso é `self`. Por isso `self` aparece duas vezes na expressão.

A «Seção 23.4» [fpy.li/8e] (vol.3) vai explicar o mecanismo que vincula automaticamente métodos a instâncias. Mas isso não se aplica a `promotion`, pois este atributo não é um método.

O código no Exemplo 3 é mais curto que o do Exemplo 1. Usar a nova `Order` é também um pouco mais simples, como mostram os doctests no Exemplo 4.

Exemplo 4. Amostra do uso da classe `Order` com as promoções como funções

```
>>> joe = Customer('John Doe', 0) ①
>>> ann = Customer('Ann Smith', 1100)
>>> cart = [LineItem('banana', 4, Decimal('.5')),
...         LineItem('apple', 10, Decimal('1.5')),
...         LineItem('watermelon', 5, Decimal(5))]
>>> Order(joe, cart, fidelity_promo) ②
<Order total: 42.00 due: 42.00>
>>> Order(ann, cart, fidelity_promo)
<Order total: 42.00 due: 39.90>
>>> banana_cart = [LineItem('banana', 30, Decimal('.5')),
...                LineItem('apple', 10, Decimal('1.5'))]
>>> Order(joe, banana_cart, bulk_item_promo) ③
<Order total: 30.00 due: 28.50>
>>> long_cart = [LineItem(str(item_code), 1, Decimal(1))
...              for item_code in range(10)]
>>> Order(joe, long_cart, large_order_promo)
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, large_order_promo)
<Order total: 42.00 due: 42.00>
```

- ① Mesmos dispositivos de teste do Exemplo 1.
- ② Para aplicar uma estratégia de desconto a uma `Order`, passamos a função de promoção como argumento.
- ③ Uma função de promoção diferente é usada aqui e no teste seguinte.

Note que não precisamos criar uma nova instância de `Promotion` a cada novo pedido: as funções já estão prontas para usar.

É interessante notar que no *Padrões de Projetos*, os autores sugerem que: "Objetos Estratégia muitas vezes são bons "peso mosca" (*flyweight*)".^[4] O padrão *Peso Mosca* é definido em outra parte do livro assim: "Um *peso mosca* é um objeto compartilhado que pode ser usado em múltiplos contextos simultaneamente."^[5] O compartilhamento é recomendado para reduzir o custo da criação de um novo objeto concreto de estratégia, quando a mesma estratégia é aplicada repetidamente a cada novo contexto—no nosso exemplo, a cada nova instância de `Order`. Se uma loja que recebe 100.000 pedidos por dia, cada estratégia concreta será instanciada milhares de vezes. Então, para reduzir o

custo de processamento do padrão Estratégia, os autores recomendam a aplicação de mais um padrão. Enquanto isso, o número de linhas e o custo de manutenção de seu código vai aumentando.

Um caso de uso mais espinhoso, com estratégias concretas complexas mantendo estados internos, pode exigir a combinação de todas as partes dos padrões de projeto Estratégia e Peso Mosca. Muitas vezes, porém, estratégias concretas não têm estado interno; elas lidam apenas com dados vindos do contexto. Neste caso, não tenha dúvida, use as boas e velhas funções ao invés de escrever classes de um só método implementando uma interface de um só método declarada em outra classe diferente. Uma função pesa menos que uma instância de uma classe definida pelo usuário, e não há necessidade do Peso Mosca, pois cada função da estratégia é criada apenas uma vez por processo Python, quando o módulo é carregado. Uma função também é um "objeto compartilhado que pode ser usado em múltiplos contextos simultaneamente".

Uma vez implementado o padrão Estratégia com funções, outras possibilidades nos ocorrem. Suponha que você queira criar uma "meta-estratégia", que seleciona o melhor desconto disponível para uma dada `Order`. Nas próximas seções vamos estudar as refatorações adicionais para implementar esse requisito, usando abordagens que se valem de funções e módulos vistos como objetos.

10.2.3. Escolhendo a melhor estratégia: abordagem simples

Dados os mesmos clientes e carrinhos de compras dos testes no Exemplo 4, vamos agora acrescentar três testes adicionais ao Exemplo 5.

Exemplo 5. A função `best_promo` aplica todos os descontos e devolve o maior

```
>>> Order(joe, long_cart, best_promo) ①
<Order total: 10.00 due: 9.30>
>>> Order(joe, banana_cart, best_promo) ②
<Order total: 30.00 due: 28.50>
>>> Order(ann, cart, best_promo) ③
<Order total: 42.00 due: 39.90>
```

① `best_promo` selecionou a `larger_order_promo` para o cliente `joe`.

② Aqui `joe` recebeu o desconto de `bulk_item_promo`, por comprar muitas bananas.

- ③ Neste caso `best_promo` deu à cliente fiel `ann` o desconto de fidelidade: `fidelity_promo`.

A implementação de `best_promo` é simples. Veja o Exemplo 6.

Exemplo 6. `best_promo` encontra o desconto máximo em uma lista de funções

```
promos = [fidelity_promo, bulk_item_promo, large_order_promo] ①

def best_promo(order: Order) -> Decimal: ②
    """Compute the best discount available"""
    return max(promo(order) for promo in promos) ③
```

- ① `promos`: lista de estratégias implementadas como funções.
- ② `best_promo` recebe uma instância de `Order` como argumento, como as outras funções *_`promo`.
- ③ Usando uma expressão geradora, aplicamos cada uma das funções de `promos` a `order`, e devolvemos o maior desconto encontrado.

O Exemplo 6 é bem direto: `promos` é uma `list` de funções. Quando você se acostuma à ideia de funções como objetos de primeira classe, o próximo passo é notar como pode ser útil construir estruturas de dados contendo funções.

Apesar do Exemplo 6 funcionar e ser fácil de ler, há alguma duplicação que poderia levar a um bug sutil: para adicionar uma nova estratégia, precisamos escrever a função e lembrar de incluí-la na lista `promos`. De outra forma a nova promoção só funcionará quando passada explicitamente como argumento para `Order`, e não será considerada por `best_promotion`.

Vamos examinar algumas soluções para essa questão.

10.2.4. Encontrando estratégias em um módulo

Módulos também são objetos de primeira classe no Python, e a biblioteca padrão oferece várias funções para lidar com eles. A função embutida `globals` é descrita assim na documentação de Python:

globals()

Devolve um dicionário representando a tabela de nomes do escopo global. Isso é sempre o dicionário do módulo atual (dentro de uma função, é o módulo onde ela foi definida, não o módulo onde é invocada).

O Exemplo 7 é uma forma um tanto *hacker* de usar `globals` para ajudar `best_promo` a encontrar automaticamente outras funções `*_promo` disponíveis.

Exemplo 7. A lista `promos` é construída a partir da introspecção do espaço de nomes global do módulo

```
from decimal import Decimal
from strategy import Order
from strategy import (
    fidelity_promo, bulk_item_promo, large_order_promo ①
)

promos = [promo for name, promo in globals().items() ②
          if name.endswith('_promo') and ③
          name != 'best_promo' ④
]

def best_promo(order: Order) -> Decimal: ⑤
    """Compute the best discount available"""
    return max(promo(order) for promo in promos)
```

- ① Importa as funções de promoções, para que fiquem disponíveis no espaço de nomes global.^[6]
- ② Itera sobre cada item no dict devolvido por `globals()`.
- ③ Seleciona apenas aqueles valores onde o nome termina com o sufixo `_promo` e...
- ④ ...filtra e remove a própria `best_promo`, para evitar uma recursão infinita quando `best_promo` for invocada.
- ⑤ Nenhuma mudança em `best_promo`.

Outra forma de coletar as promoções disponíveis seria criar um módulo e colocar nele todas as funções de estratégia, exceto `best_promo`.

No Exemplo 8, a única mudança significativa é que a lista de funções de estratégia é criada pela introspecção de um módulo separado chamado `promotions`. Veja que o Exemplo 8 depende da importação do módulo `promotions` bem como de funções de introspecção de alto nível do módulo `inspect` da biblioteca padrão.

Exemplo 8. A lista `promos` é construída a partir da introspecção de um novo módulo, `promotions`

```
from decimal import Decimal
from inspect import getmembers, isfunction

from strategy import Order
import promotions

promos = [func for _, func in getmembers(promotions, isfunction)]

def best_promo(order: Order) -> Decimal:
    """Compute the best discount available"""
    return max(promo(order) for promo in promos)
```

A função `inspect.getmembers` devolve os atributos de um objeto—neste caso, o módulo `promotions`—opcionalmente filtrados por um predicado (uma função booleana). Usamos `inspect.isfunction` para obter apenas as funções.

O Exemplo 8 funciona independente dos nomes dados às funções; o que importa é que o módulo `promotions` contém apenas funções que, dado um pedido, calculam os descontos. Claro, isso é uma suposição implícita do código. Se alguém criasse uma função com uma assinatura diferente no módulo `promotions`, `best_promo` geraria um erro ao tentar aplicá-la a um pedido.

Poderíamos acrescentar testes mais estritos para filtrar as funções, por exemplo inspecionando seus argumentos. O ponto principal do Exemplo 8 não é oferecer uma solução completa, mas enfatizar um uso possível da introspecção de módulo.

Uma alternativa mais explícita para coletar dinamicamente as funções de desconto promocional seria usar um decorador simples. É nosso próximo tópico.

10.3. Estratégia com decorador de registro

Lembre-se que nossa principal objeção ao Exemplo 6 foi a repetição dos nomes das funções em suas definições e na lista `promos`, usada pela função `best_promo` para determinar o maior desconto aplicável. A repetição é problemática porque alguém pode acrescentar uma nova função de estratégia promocional e esquecer de adicioná-la manualmente à lista `promos`—caso em que `best_promo` vai ignorar a nova estratégia, introduzindo um bug silencioso. O Exemplo 9 resolve esse problema com a técnica vista na Seção 9.4.

Exemplo 9. A lista `promos` é preenchida pelo decorador `promotion`

```
Promotion = Callable[[Order], Decimal]

promos: list[Promotion] = [] ①

def promotion(promo: Promotion) -> Promotion: ②
    promos.append(promo)
    return promo

def best_promo(order: Order) -> Decimal:
    """Compute the best discount available"""
    return max(promo(order) for promo in promos) ③

@promotion ④
def fidelity(order: Order) -> Decimal:
    """5% discount for customers with 1000 or more fidelity points"""
    if order.customer.fidelity >= 1000:
        return order.total() * Decimal('0.05')
    return Decimal(0)

@promotion
def bulk_item(order: Order) -> Decimal:
    """10% discount for each LineItem with 20 or more units"""
    discount = Decimal(0)
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * Decimal('0.1')
    return discount
```

```
@promotion
def large_order(order: Order) -> Decimal:
    """7% discount for orders with 10 or more distinct items"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * Decimal('0.07')
    return Decimal(0)
```

- ① A lista `promos` é global no módulo, e começa vazia.
- ② `promotion` é um decorador de registro: ele devolve a função `promo` inalterada, após inserí-la na lista `promos`.
- ③ Nenhuma mudança é necessária em `best_promo`, pois ela se baseia na lista `promos`.
- ④ Qualquer função decorada com `@promotion` será adicionada a `promos`.

Essa solução tem várias vantagens sobre aquelas apresentadas anteriormente:

- As funções de estratégia de promoção não precisam usar nomes especiais—não há necessidade do sufixo `_promo`.
- O decorador `@promotion` realça o propósito da função decorada, e também torna mais fácil desabilitar temporariamente uma promoção: basta transformar a linha do decorador em comentário.
- Estratégias de desconto promocional podem ser definidas em outros módulos, em qualquer lugar do sistema, desde que o decorador `@promotion` seja aplicado a elas.

Na próxima seção vamos discutir Comando (*Command*)—outro padrão de projeto que é algumas vezes implementado via classes de um só método, quando funções simples seriam suficientes.

10.4. O padrão Comando

Comando é outro padrão de projeto que pode ser simplificado com o uso de funções passadas como argumentos. A Figura 2 mostra o arranjo das classes nesse padrão.

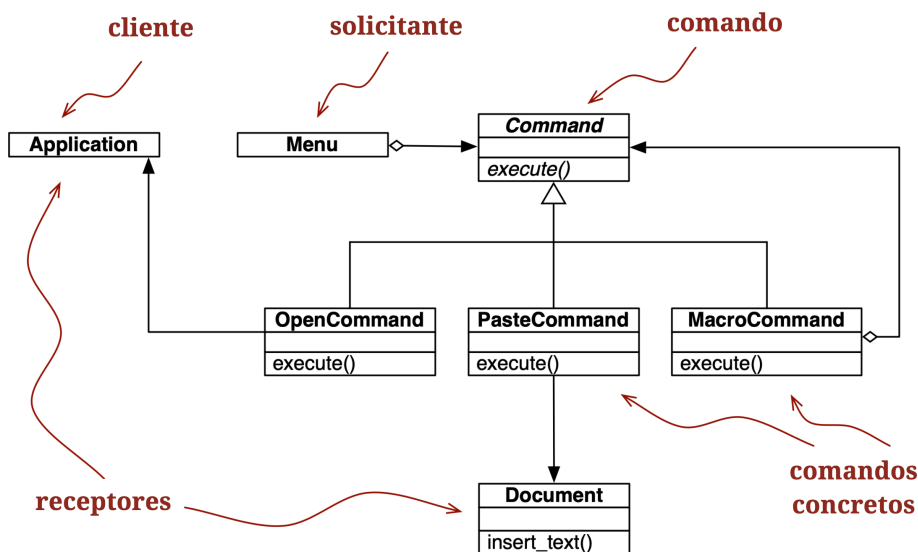


Figura 2. Diagrama de classes UML para um editor de texto controlado por menus, implementado com o padrão de projeto Comando. Cada comando pode ter um receptor diferente: o objeto que implementa a ação. Para PasteCommand, o receptor é Document. Para OpenCommand, o receptor é a aplicação.

O objetivo de Comando é desacoplar um objeto que invoca uma operação (o *invoker* ou solicitante) do objeto fornecedor que implementa aquela operação (o *receiver* ou receptor). No exemplo em *Padrões de Projetos*, cada solicitante é um item de menu em uma aplicação gráfica, e os receptores são o documento sendo editado ou a própria aplicação.

A ideia é colocar um objeto Command entre os dois, implementando uma interface com um único método, `execute`, que chama algum método no receptor para executar a operação desejada. Assim, o solicitante não precisa conhecer a interface do receptor, e receptores diferentes podem ser adaptados com diferentes subclasses de Command. O solicitante é configurado com um comando concreto, e o opera chamando seu método `execute`. Observe na Figura 2 que MacroCommand pode armazenar um sequência de comandos; seu método `execute()` chama o mesmo método em cada comando armazenado.

Citando *Padrões de Projetos*, "Comandos são um substituto orientado a objetos para *callbacks*." A pergunta é: precisamos de um substituto orientado a objetos para *callbacks*? Algumas vezes sim, mas nem sempre. Em vez de dar ao solicitante uma instância de Command, podemos dar a ele uma função. Em vez de invocar `command.execute()`, o solicitante pode invocar `command()` diretamente.

O `MacroCommand` pode ser uma classe que implementa `__call__`. Instâncias de `MacroCommand` seriam invocáveis, cada uma contendo uma lista de comandos para invocação futura, como implementado no Exemplo 10.

Exemplo 10. Cada instância de `MacroCommand` tem uma lista interna de comandos

```
class MacroCommand:
    """A command that executes a list of commands"""

    def __init__(self, commands):
        self.commands = list(commands) ①

    def __call__(self):
        for command in self.commands: ②
            command()
```

- ① Criar uma nova lista com os itens do argumento `commands` garante que ela seja iterável e mantém uma cópia local de referências a comandos em cada instância de `MacroCommand`.
- ② Quando uma instância de `MacroCommand` é invocada, cada comando em `self.commands` é chamado em sequência.

Usos mais avançados do padrão Comando—para implementar "desfazer", por exemplo—podem exigir mais que uma simples função de *callback*. Mesmo assim, Python oferece algumas alternativas que merecem ser consideradas:

- Uma instância invocável como `MacroCommand` no Exemplo 10 pode manter qualquer estado que seja necessário, e oferecer outros métodos além de `__call__`.
- Uma clausura pode ser usada para armazenar algum estado interno em uma função entre invocações.

Isso encerra nossa revisão do padrão Comando usando funções de primeira classe. Por alto, a abordagem aqui foi similar à que aplicamos a Estratégia: substituir por funções as instâncias de uma classe participante que implementava uma interface de método único. Afinal, todo invocável de Python implementa uma interface de método único, e esse método se chama `__call__`.

10.5. Resumo do capítulo

Como apontou Peter Norvig alguns anos após o surgimento do clássico *Padrões de Projetos*, "16 dos 23 padrões têm implementações qualitativamente mais simples em Lisp ou Dylan que em C++, pelo menos para alguns usos de cada padrão". Python compartilha alguns dos recursos dinâmicos das linguagens Lisp e Dylan, especialmente funções de primeira classe, nosso foco neste capítulo.

Na mesma palestra citada no início deste capítulo, refletindo sobre o 20º aniversário de *Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos*, Ralph Johnson afirmou que um dos defeitos do livro é: "Excesso de ênfase nos padrões como linhas de chegada, em vez de como etapas em um processo de design".^[7] Neste capítulo usamos o padrão Estratégia como ponto de partida: uma solução que funcionava, mas que simplificamos usando funções de primeira classe.

Em muitos casos, funções ou objetos invocáveis oferecem um caminho mais natural para implementar *callbacks* em Python que a imitação dos padrões Estratégia ou Comando como descritos pela Gangue dos Quatro em *Padrões de Projetos*. A refatoração de Estratégia e a discussão de Comando nesse capítulo são exemplos de uma ideia mais geral: algumas vezes você pode encontrar um padrão de projeto ou uma API que exigem que seus componentes implementem uma interface com um único método, e aquele método tem um nome que soa muito genérico, como "executar", "rodar" ou "fazer". Tais padrões ou APIs podem frequentemente ser implementados em Python com menos código repetitivo, usando funções como objetos de primeira classe.

10.6. Para saber mais

A Receita 8.21. *Implementing the Visitor Pattern* (Implementando o Padrão Visitante) no *Python Cookbook 3rd ed*, mostra uma implementação elegante do padrão Visitante, na qual uma classe `NodeVisitor` trata métodos como objetos de primeira classe.

Sobre o tópico mais geral de padrões de projetos, a oferta de leituras para o programador Python não é tão numerosa quando aquela disponível para as comunidades de outras linguagens.

Expert Python Programming, de Tarek Ziadé (Packt), é um dos melhores livros sobre Python em nível intermediário, apresenta vários padrões clássicos com uma abordagem pythônica e no seu último capítulo, *Learning Python Design Patterns*, de Gennadiy Zlobin (Packt), é o único livro inteiramente dedicado a padrões em Python que encontrei. Mas esta obra de 100 páginas cobre apenas 8 dos 23 padrões de projeto originais.

Alex Martelli já apresentou várias palestras sobre padrões de projetos em Python. Há um vídeo de sua apresentação na EuroPython [fpy.li/10-5] e um conjunto de slides em seu site pessoal [fpy.li/10-6]. Ao longo dos anos, encontrei vários conjuntos de slides e vídeos, então vale a pena pesquisar o nome dele e com palavras "Python Design Patterns".

Há muitos livros sobre padrões de projetos com ênfase em Java. Meu preferido é *Head First Design Patterns* (Use a Cabeça: Padrões de Projeto), 2ª ed., de Eric Freeman e Elisabeth Robson (O'Reilly). Eles explicam 16 dos 23 padrões clássicos. Se você gosta do estilo amalucado da série *Head First* e precisa de uma introdução a esse tópico, vai adorar esse livro. A segunda edição foi atualizada para incorporar o uso de funções de primeira classe em Java, tornando alguns dos exemplos mais próximos do modo como escreveríamos em Python.

Para um olhar moderno sobre padrões, do ponto de vista de uma linguagem dinâmica com tipagem pato (*duck typing*) e funções de primeira classe, *Design Patterns in Ruby* ("Padrões de Projetos em Ruby") de Russ Olsen (Addison-Wesley) traz muitas ideias aplicáveis também ao Python. Apesar de suas muitas diferenças sintáticas, no nível semântico Python e Ruby estão mais próximos entre si que de Java ou do C++.

No slides de *Design Patterns in Dynamic Languages* [fpy.li/norvigdp] (Padrões de Projetos em Linguagens Dinâmicas), Peter Norvig mostra como funções de primeira classe e outros recursos dinâmicos tornam vários dos padrões de projeto originais mais simples ou mesmo desnecessários.

A "Introdução" do *Padrões de Projetos* original, de Gamma et al. já vale o preço do livro—mas até que o catálogo de 23 padrões, que inclui desde receitas muito importantes até algumas raramente úteis. Alguns princípios de projetos de software muito conhecidos, como "Programa para uma interface, não para uma implementação" e "Prefira a composição de objetos à herança de classe", são citações daquela introdução.

A ideia de padrões de projetos se originou com o arquiteto Christopher Alexander et al., e foi apresentada no livro *A Pattern Language* ("Uma Linguagem de Padrões") (Oxford University Press). A ideia de Alexander é criar um vocabulário padronizado, permitindo que equipes compartilhem decisões comuns em projetos de edificações. M. J. Dominus wrote "*Design Patterns*" Aren't [fpy.li/10-7] (Padrões de Projetos Não São), uma curiosa apresentação de slides acompanhada de um texto argumentando que a visão original de Alexander sobre os padrões é mais profunda e mais humanista, e também se aplica à engenharia de software.

Ponto de vista

Padrões para quem precisa de padrões

Python tem funções de primeira classe e tipos de primeira classe, e Norvig afirma que esses recursos afetam 10 dos 23 padrões (slide 10 de *Design Patterns in Dynamic Languages* [fpy.li/norvigdp]). Na Seção 9.9.3, vimos que Python também tem funções genéricas de despacho único, uma forma limitada dos multi-métodos do CLOS, que Gamma et al. sugerem como uma maneira mais simples de implementar o padrão clássico Visitante (*Visitor*). Norvig, por outro lado, diz (no slide 10) que os multi-métodos simplificam o padrão Construtor (*Builder*). Ligar padrões de projetos a recursos de linguagens não é uma ciência exata.

Em cursos a redor do mundo todo, padrões de projetos são frequentemente ensinados usando exemplos em Java. Ouvi mais de um estudante dizer que eles foram levados a crer que os padrões de projeto originais são úteis qualquer que seja a linguagem usada na implementação. A verdade é que os 23 padrões "clássicos" de *Padrões de Projetos* se aplicam muito bem ao Java, apesar de terem sido apresentados principalmente no contexto do C++ (no livro, há alguns exemplos em Smalltalk). Mas isso não significa que todos aqueles padrões podem ser aplicados de forma igualmente satisfatória a qualquer linguagem. Os autores dizem explicitamente, logo no início de seu livro, que "alguns de nossos padrões são suportados diretamente por linguagens orientadas a objetos menos conhecidas" (a citação completa apareceu na primeira página deste capítulo).

Agora que Python está se tornando cada vez mais popular no ambiente acadêmico, podemos esperar que novos livros sobre padrões de projetos sejam escritos com foco nesta linguagem. Além disso, o Java 8 introduziu referências a métodos e funções anônimas, e esses recursos muito esperados devem incentivar o surgimento de novas abordagens aos padrões em Java—reconhecendo que, à medida que as linguagens evoluem, também é preciso evoluir nosso entendimento sobre quando e como aplicar os padrões de projetos clássicos.

O chamado da natureza

Enquanto trabalhávamos juntos para dar os toques finais a este livro, o revisor técnico Leonardo Rochael pensou:

Se funções têm um método `__call__`, e métodos também são invocáveis, será que os métodos `__call__` também tem um método `__call__`?

Não sei se a descoberta do Leo é útil, mas com certeza é curiosa:

```
>>> def turtle():
...     return 'eggs'
...
>>> turtle()
'eggs'
>>> turtle.__call__()
'eggs'
>>> turtle.__call__.__call__()
'eggs'
>>> turtle.__call__.__call__.__call__()
'eggs'
>>> turtle.__call__.__call__.__call__.__call__()
'eggs'
>>> turtle.__call__.__call__.__call__.__call__.__call__()
'eggs'
>>> turtle.__call__.__call__.__call__.__call__.__call__.__call__()
'eggs'
```

Turtles all the way down! [fpy.li/10-8]^[8]

[1] De um slide na palestra *Root Cause Analysis of Some Faults in Design Patterns* (Análise das Causas Básicas de Alguns Defeitos em Padrões de Projetos), apresentada por Ralph Johnson no IME/CCSL da Universidade de São Paulo, em 15 de novembro de 2014.

[2] *Visitor*, Citado da página 4 da edição em inglês de *Padrões de Projeto*.

[3] Precisei reimplementar `Order` com `@dataclass` devido a um bug no Mypy. Você pode ignorar esse detalhe, pois essa classe funciona também com `NamedTuple`, exatamente como no Exemplo 1. Quando `Order` é uma `NamedTuple`, o Mypy 0.910 encerra com erro ao checar a dica de tipo para `promotion`. Tentei acrescentar `# type ignore` àquela linha específica, mas o erro persistia. Entretanto, se `Order` for criada com `@dataclass`, o Mypy trata corretamente a mesma dica de tipo. O Issue #9397 [fpy.li/10-3] não havia sido resolvido em 19 de julho de 2021, quando essa nota foi escrita. Espero que o problema tenha sido solucionado quando você estiver lendo isso. NT: Aparentemente foi resolvido. O Issue #9397 gerou o Issue #12629 [fpy.li/62], fechado com indicação de solucionado em agosto de 2022, o último comentário indicando que a opção de linha de comando `--enable-recursive-aliases` do Mypy evita os erros relatados).

[4] veja a página 323 da edição em inglês de *Padrões de Projetos*.

[5] *Ibid.*, p. 196.

[6] Tanto o `flake8` quanto o VS Code reclamam que esses nomes são importados mas não são usados. Por definição, ferramentas de análise estática não conseguem lidar com a natureza dinâmica de Python. Se seguirmos todos os conselhos dessas ferramentas, logo estaremos escrevendo programas austeros e prolixos similares aos de Java, mas com a sintaxe de Python.

[7] *Root Cause Analysis of Some Faults in Design Patterns* (Análise das Causas Básicas de Alguns Defeitos em Padrões de Projetos), palestra apresentada por Johnson no IME/CCSL da Universidade de São Paulo, em 15 de novembro de 2014.

[8] NT: Literalmente: "Tartarugas até lá embaixo". Esta é uma forma poética de falar sobre regressão infinita, em alusão ao mito de que a Terra se apoia sobre uma tartaruga gigante, que se apoia sobre outra tartaruga gigante, que se apoia sobre outra tartaruga gigante, que se apoia sobre outra tartaruga gigante, que se apoia sobre outra tartaruga gigante, que se apoia sobre outra tartaruga gigante, que se apoia sobre outra tartaruga gigante...

Parte III: Classes e Protocolos

Capítulo 11. Um objeto pythônico

Para uma biblioteca ou framework, ser pythônica significa tornar tão fácil e tão natural quanto possível que um programador Python descubra como realizar uma tarefa.^[1]

— Martijn Faassen, criador de frameworks Python e JavaScript

Graças ao Modelo de Dados de Python, nossos tipos definidos pelo usuário podem se comportar de forma tão natural quanto os tipos embutidos. E isso pode ser realizado sem herança, no espírito do *duck typing*: implemente os métodos necessários e seus objetos se comportarão da forma esperada.

Nos capítulos anteriores, estudamos o comportamento de vários objetos embutidos. Vamos agora criar classes definidas pelo usuário que se portam como objetos Python nativos. As classes na sua aplicação provavelmente não precisam implementar tantos métodos especiais quanto os exemplos nesse capítulo. Mas se você estiver escrevendo uma biblioteca ou um framework, os programadores que usarão suas classes talvez esperem que elas se comportem como as classes fornecidas pelo Python. Satisfazer tal expectativa é um dos jeitos de ser "pythônico".

Esse capítulo começa onde o «Capítulo 1» [fpy.li/1] (vol.1) terminou, mostrando como implementar vários métodos especiais comumente vistos em objetos Python de diferentes tipos.

Veremos como:

- Suportar as funções embutidas que convertem objetos para outros tipos (por exemplo, `repr()`, `bytes()`, `complex()`, etc.)
- Implementar um construtor alternativo como um método da classe
- Estender a mini-linguagem de formatação usada pelas f-strings, pela função embutida `format()` e pelo método `str.format()`
- Fornecer acesso a atributos apenas para leitura
- Tornar um objetos *hashable*, para uso em conjuntos e como chaves de dict
- Economizar memória com `__slots__`

Vamos fazer tudo isso enquanto desenvolvemos `Vector2d`, um tipo simples de vetor euclidiano bi-dimensional. No Capítulo 12, o mesmo código servirá de base para uma classe de vetor N-dimensional.

A evolução do exemplo incluirá dois tópicos conceituais importantes:

- Como e quando usar os decoradores `@classmethod` e `@staticmethod`
- Atributos privados e protegidos no Python: uso, convenções e limitações

11.1. Novidades neste capítulo

Acrescentei uma nova epígrafe e também algumas palavras ao segundo parágrafo do capítulo, para falar do conceito de "pythônico"—que na primeira edição era mencionado só no final do livro.

Atualizei a Seção 11.6 para mencionar as f-strings, introduzidas no Python 3.6. É uma mudança pequena, pois as f-strings suportam a mesma mini-linguagem de formatação que a função embutida `format()` e o método `str.format()`, então quaisquer métodos `__format__` implementados antes vão funcionar também com as f-strings.

O resto do capítulo quase não mudou—os métodos especiais são praticamente os mesmos desde o Python 3.0, e a maioria existe desde o Python 2.2.

Vamos começar pelos métodos de representação de objetos.

11.2. Representações de objetos

Todas as linguagens orientadas a objetos têm pelo menos uma forma padrão de se obter uma representação de qualquer objeto como uma string. Python tem duas formas:

`repr()`

Devolve uma string representando o objeto como o desenvolvedor quer vê-lo. É o que aparece quando o console de Python ou um depurador mostram um objeto.

str()

Devolve uma string representando o objeto de uma forma amigável para o usuário final. É o que aparece quando se passa um objeto como argumento para `print()`.

Os métodos especiais `__repr__` e `__str__` suportam `repr()` e `str()`, como vimos no «Capítulo 1» [fpy.li/1] (vol.1).

Existem mais dois métodos especiais para gerar representações alternativas de objetos, `__bytes__` e `__format__`. O método `__bytes__` é análogo a `__str__`: ele é chamado por `bytes()` para obter um objeto representado como uma sequência de bytes. Já `__format__` é usado por f-strings, pela função embutida `format()` e pelo método `str.format()`. Todos eles chamam `obj.format(fmt_spec)` para gerar uma string exibindo o objeto conforme códigos de formatação especiais. Vamos tratar de `__bytes__` na próxima seção e de `__format__` logo depois.



Se você está vindo de Python 2, lembre-se de que no Python 3 `__repr__`, `__str__` e `__format__` devem sempre devolver strings Unicode (tipo `str`). Apenas `__bytes__` deveria devolver uma sequência de bytes (tipo `bytes`).

11.3. A volta da classe Vector

Para demonstrar os vários métodos usados para gerar representações de objetos, vamos criar uma classe `Vector2d`, similar à que vimos no «Capítulo 1» [fpy.li/1] (vol.1). O Exemplo 1 ilustra o comportamento básico que esperamos de uma instância de `Vector2d`.

Exemplo 1. Instâncias de `Vector2d` têm várias representações

```
>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y) ①
3.0 4.0
>>> x, y = v1 ②
>>> x, y
(3.0, 4.0)
>>> v1 ③
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1)) ④
```

```

>>> v1 == v1_clone ⑤
True
>>> print(v1) ⑥
(3.0, 4.0)
>>> octets = bytes(v1) ⑦
>>> octets
b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\
x00\\x10@'
>>> abs(v1) ⑧
5.0
>>> bool(v1), bool(Vector2d(0, 0)) ⑨
(True, False)

```

- ① Os componentes de um `Vector2d` podem ser acessados diretamente como atributos (não é preciso invocar métodos *getter*).
- ② Um `Vector2d` pode ser desempacotado para uma tupla de variáveis.
- ③ O `repr` de um `Vector2d` imita o código-fonte usado para construir a instância.
- ④ Usar `eval` aqui mostra que o `repr` de um `Vector2d` é uma representação fiel da chamada a seu construtor.^[2]
- ⑤ `Vector2d` suporta a comparação com `==` (muito útil para testes).
- ⑥ `print` chama `str`, que no caso de `Vector2d` exibe um par ordenado.
- ⑦ `bytes` usa o método `__bytes__` para produzir uma representação binária.
- ⑧ `abs` usa o método `__abs__` para devolver a magnitude do `Vector2d`.
- ⑨ `bool` usa o método `__bool__` para devolver `False` se o `Vector2d` tiver magnitude zero, caso contrário esse método devolve `True`.

A classe `Vector2d` do Exemplo 1 é implementada em `vector2d_v0.py` (no Exemplo 2). O código está baseado no «Exemplo 2 do Capítulo 1» [fpy.li/8w] (vol.1), exceto pelos métodos para os operadores `+` e `*`, que veremos mais tarde no Capítulo 16. Vamos acrescentar o método para `==`, pois ele facilita escrever testes. Nesse ponto, `Vector2d` usa vários métodos especiais para oferecer operações que um pythonista espera encontrar em um objeto bem projetado.

Exemplo 2. vector2d_v0.py: todos os métodos até aqui são métodos especiais

```
from array import array
import math

class Vector2d:
    typecode = 'd' ①

    def __init__(self, x, y):
        self.x = float(x) ②
        self.y = float(y)

    def __iter__(self):
        return (i for i in (self.x, self.y)) ③

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self) ④

    def __str__(self):
        return str(tuple(self)) ⑤

    def __bytes__(self):
        return (self.typecode.encode('ascii') + ⑥
                bytes(array(self.typecode, self))) ⑦

    def __eq__(self, other):
        return tuple(self) == tuple(other) ⑧

    def __abs__(self):
        return math.hypot(self.x, self.y) ⑨

    def __bool__(self):
        return bool(abs(self)) ⑩
```

① typecode é um atributo de classe, usado na conversão de instâncias de Vector2d de/para bytes.

② Converter x e y para float em __init__ captura erros mais rápido, algo útil quando Vector2d é chamado com argumentos não numéricos.

- ③ `__iter__` torna um `Vector2d` iterável; é isso que faz o desempacotamento funcionar (por exemplo, `x, y = my_vector`). Usamos uma expressão geradora para produzir os dois componentes, um após outro.^[3]
- ④ O `__repr__` cria uma string interpolando os componentes com `{!r}`, para obter seus `repr`; como `Vector2d` é iterável, `*self` alimenta `format` com os componentes `x` e `y`.
- ⑤ Como `Vector2d` é iterável, é fácil criar uma tuple para exibição como um par ordenado.
- ⑥ Para gerar bytes, convertemos o `typecode` para bytes e concatenamos...
- ⑦ ...bytes convertidos a partir de um array criado iterando sobre a instância.
- ⑧ Para comparar facilmente todos os componentes, criamos tuplas a partir dos operandos. Isso funciona para operandos que sejam instâncias de `Vector2d`, mas tem problemas. Veja o alerta abaixo.
- ⑨ A magnitude é o comprimento da hipotenusa do triângulo retângulo com os catetos formados pelos componentes `x` e `y`.
- ⑩ `__bool__` usa `abs(self)` para computar a magnitude, então a converte para `bool`; assim, `0.0` se torna `False`, qualquer valor diferente de zero é `True`.



O método `__eq__` no Exemplo 2 funciona para operandos `Vector2d`, mas também devolve `True` ao comparar instâncias de `Vector2d` a outros iteráveis contendo os mesmos valores numéricos (por exemplo, `Vector(3, 4) == [3, 4]`). Isso pode ser considerado uma característica ou um bug. Essa discussão terá que esperar até o Capítulo 16, onde falamos de sobrecarga de operadores.

Temos um conjunto bastante completo de métodos básicos, mas ainda precisamos de uma maneira de reconstruir um `Vector2d` a partir da representação binária produzida por `bytes()`.

11.4. Um construtor alternativo

Já que podemos exportar um `Vector2d` na forma de bytes, naturalmente precisamos de um método para importar um `Vector2d` de uma sequência binária. Procurando na biblioteca padrão por algo similar, descobrimos que `array.array`

tem um método de classe chamado `.frombytes`, adequado a nossos propósitos—já o vimos na «Seção 2.10.1» [fpy.li/7v] (vol.1). Adotamos o mesmo nome e usamos sua funcionalidade em um método de classe para `Vector2d` em `vector2d_v1.py` (no Exemplo 3).

Exemplo 3. Parte de `vector2d_v1.py`: esse trecho mostra apenas o método de classe `frombytes`, acrescentado à definição de `Vector2d` em `vector2d_v0.py` (no Exemplo 2)

```
@classmethod ①
def frombytes(cls, octets): ②
    typecode = chr(octets[0]) ③
    memv = memoryview(octets[1:]).cast(typecode) ④
    return cls(*memv) ⑤
```

- ① O decorador `classmethod` modifica um método para que ele possa ser chamado diretamente em uma classe.
- ② Nenhum argumento `self`; em vez disso, a própria classe é passada como primeiro argumento—por convenção chamado `cls`.
- ③ Lê o `typecode` do primeiro byte.
- ④ Cria uma `memoryview` a partir da sequência binária `octets`, e usa o `typecode` para convertê-la.^[4]
- ⑤ Desempacota a `memoryview` resultante da conversão no par de argumentos necessários para o construtor.

Acabei de usar um decorador `classmethod`, e ele é muito específico do Python. Vamos então falar um pouco disso.

11.5. `classmethod` versus `staticmethod`

O decorador `classmethod` não é mencionado no tutorial de Python, nem tampouco o `staticmethod`. Quem OO com Java pode se perguntar porque Python tem esses dois decoradores, e não apenas `staticmethod`.

Vamos começar com `classmethod`. O Exemplo 3 mostra seu uso: definir um método que opera na classe, e não em suas instâncias. O `classmethod` muda a forma como o método é chamado, então recebe a própria classe como primeiro argumento, em vez de uma instância. Seu uso mais comum é em construtores

alternativos, como `frombytes` no Exemplo 3. Observe como a última linha de `frombytes` o argumento `cls`, invocando-o para criar uma nova instância: `cls(*memv)`.

O decorador `staticmethod`, por outro lado, muda um método para que ele não receba um argumento automaticamente. Essencialmente, um método estático é apenas uma função simples que por acaso mora no corpo de uma classe, em vez de ser definida no nível do módulo. O Exemplo 4 compara a operação de `classmethod` e `staticmethod`.

Exemplo 4. Comparando o comportamento de `classmethod` e `staticmethod`

```
>>> class Demo:
...     @classmethod
...     def klassmeth(*args):
...         return args ①
...     @staticmethod
...     def statmeth(*args):
...         return args ②
...
>>> Demo.klassmeth() ③
(<class '__main__.Demo'>,)
>>> Demo.klassmeth('spam')
(<class '__main__.Demo'>, 'spam')
>>> Demo.statmeth() ④
()
>>> Demo.statmeth('spam')
('spam',)
```

- ① `klassmeth` apenas devolve todos os argumentos posicionais.
- ② `statmeth` faz o mesmo.
- ③ Não importa como ele seja invocado, `Demo.klassmeth` recebe sempre a classe `Demo` como primeiro argumento.
- ④ `Demo.statmeth` se comporta exatamente como uma boa e velha função.



O decorador `classmethod` é obviamente útil mas, em minha experiência, bons casos de uso para `staticmethod` são raros. Talvez a função seja intimamente relacionada a classe, mesmo sem nunca usá-la em seu corpo. Daí você pode querer que ela

fique próxima no código-fonte. Mesmo assim, definir a função logo antes ou logo depois da classe, no mesmo módulo, é perto o suficiente na maioria dos casos.^[5]

Agora que vimos para que serve o `classmethod` (e que o `staticmethod` não é muito útil), vamos voltar para a questão da representação de objetos e entender como gerar uma saída formatada.

11.6. Exibição formatada

As f-strings, a função embutida `format()` e o método `str.format()` delegam a lógica da formatação para cada tipo, chamando seu método `__format__(fmt_spec)`. A string `fmt_spec` especifica a formatação desejada. Esta especificação é:

- O segundo argumento em `format(my_obj, fmt_spec)`, ou
- O que aparece após os dois pontos (`:`) em um campo de substituição delimitado por `{}` dentro de uma f-string ou na string `s` em `s.format()`

Por exemplo:

```
>>> brl = 1 / 4.82 # BRL to USD currency conversion rate
>>> brl
0.20746887966804978
>>> format(brl, '0.4f') ①
'0.2075'
>>> '1 BRL = {rate:0.2f} USD'.format(rate=brl) ②
'1 BRL = 0.21 USD'
>>> f'1 USD = {1 / brl:0.2f} BRL' ③
'1 USD = 4.82 BRL'
```

- ① A especificação de formato é `'0.4f'`.
- ② A especificação de formato é `'0.2f'`. O `rate` no campo de substituição não é parte da especificação de formato. Ele determina qual argumento nomeado de `.format()` entra naquele campo de substituição.
- ③ Novamente, a especificação é `'0.2f'`. A expressão `1 / brl` não é parte dela.

O segundo e o terceiro comentário apontam um fato importante: uma string de formatação tal como `'{0.mass:5.3e}'` usa duas notações separadas. O `'0.mass'` à esquerda dos dois pontos é a parte `field_name` da sintaxe de campo de substituição, e pode ser uma expressão arbitrária em uma f-string. O `'5.3e'` após os dois pontos é a especificação do formato. A notação usada na especificação de formato é chamada Mini-Linguagem de Especificação de Formato [fpy.li/63].



Se f-strings, `format()` e `str.format()` são novidades para você, minha experiência como professor me informa que é melhor estudar primeiro a função embutida `format()`, que usa apenas a Mini-Linguagem de Especificação de Formato [fpy.li/63]. Após pegar o jeito dela, leia «Literais de string formatados» [fpy.li/64] e «Sintaxe das string de formato» [fpy.li/65], para aprender sobre a notação de campo de substituição (`{:}`), usada em f-strings e no método `str.format()` (incluindo os marcadores de conversão `!s`, `!r`, e `!a`). F-strings não tornam o método `str.format()` obsoleto: na maioria dos casos f-strings resolvem o problema, mas algumas vezes é melhor especificar a string de formatação em outro arquivo (diferente de onde ela será utilizada).

Alguns tipos embutidos têm seus próprios códigos de apresentação na Mini-Linguagem de Especificação de Formato. Por exemplo—entre muitos outros códigos—o tipo `int` suporta `b` e `x`, para saídas em base 2 e base 16, respectivamente, enquanto `float` implementa `f`, para uma exibição de ponto fixo, e `%`, para exibir porcentagens:

```
>>> format(42, 'b')
'101010'
>>> format(2 / 3, '.1%')
'66.7%'
```

A Mini-Linguagem de Especificação de Formato é extensível, porque cada classe interpreta o argumento `fmt_spec` como quiser. Por exemplo, as classes no módulo `datetime` usam em seus métodos `__format__` os mesmos códigos de formatação das funções `strftime()`, que são mais antigas. Veja abaixo alguns exemplos de uso da função `format()` e do método `str.format()`:


```
>>> from datetime import datetime
>>> now = datetime.now()
>>> format(now, '%H:%M:%S')
'18:49:05'
>>> "It's now {:%I:%M %p}".format(now)
"It's now 06:49 PM"
```

Se a classe não implementar `__format__`, o método herdado de `object` devolve `str(my_object)`. Como `Vector2d` tem um `__str__`, isso funciona:

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
```

Entretanto, se você passar um especificador de formato, `object.__format__` gera um `TypeError`:

```
>>> format(v1, '.3f')
Traceback (most recent call last):
...
TypeError: non-empty format string passed to object.__format__
```

Vamos corrigir isso implementando nossa própria mini-linguagem de formatação. O primeiro passo será presumir que o especificador de formato fornecido pelo usuário tem por objetivo formatar cada componente `float` do vetor. Esse é o resultado esperado:

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

O Exemplo 5 implementa `__format__` para produzir as formatações vistas acima.

Exemplo 5. O método `Vector2d.__format__`, versão #1

```
# inside the Vector2d class

def __format__(self, fmt_spec=''):
    components = (format(c, fmt_spec) for c in self) ①
    return '({}, {})'.format(*components) ②
```

- ① Usa a função embutida `format` para aplicar o `fmt_spec` a cada componente do vetor, criando um iterável de strings formatadas.
- ② Insere as strings formatadas no gabarito `'(x, y)'`.

Agora vamos acrescentar um código de formatação customizado à nossa mini-linguagem: se o especificador de formato terminar com `'p'`, vamos exibir o vetor em coordenadas polares: $\langle r, \theta \rangle$, onde r é a magnitude e θ (theta) é o ângulo em radianos. O restante do especificador de formato (o que quer que venha antes do `'p'`) será usado como antes.



Ao escolher a letra para um código customizado de formato, evitei sobrescrever códigos usados por outros tipos. Na Mini-Linguagem de Especificação de Formato [fpy.li/63] vemos que inteiros usam os códigos `'bcdxXn'`, floats usam `'eEfFgGn%'` e strings usam `'s'`. Então escolhi `'p'` para coordenadas polares. Como cada classe interpreta esses códigos de forma independente, reutilizar uma letra em um formato customizado para um novo tipo não é um erro, mas pode ser confuso para os usuários.

Para gerar coordenadas polares, já temos o método `__abs__` para a magnitude. Vamos então escrever um método `angle` simples, usando a função `math.atan2()`, para obter o ângulo. Eis o código:

```
# inside the Vector2d class

def angle(self):
    return math.atan2(self.y, self.x)
```

Com isso, podemos agora aperfeiçoar nosso `__format__` para gerar coordenadas polares. Veja o Exemplo 6.

Exemplo 6. O método `Vector2d.__format__`, versão #2, agora com coordenadas polares

```
def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('p'): ①
        fmt_spec = fmt_spec[:-1] ②
        coords = (abs(self), self.angle()) ③
        outer_fmt = '<{}, {}>' ④
    else:
        coords = self ⑤
        outer_fmt = '({}, {})' ⑥
    components = (format(c, fmt_spec) for c in coords) ⑦
    return outer_fmt.format(*components) ⑧
```

- ① O formato termina com 'p': usar coordenadas polares.
- ② Remove o sufixo 'p' de `fmt_spec`.
- ③ Cria uma tuple de coordenadas polares: (magnitude, angle).
- ④ Configura o formato externo com colchetes angulares < >.
- ⑤ Caso contrário, usa os componentes x, y de `self` para coordenadas retangulares.
- ⑥ Configura o formato externo com parênteses.
- ⑦ Gera um iterável cujos componentes são strings formatadas.
- ⑧ Insere as strings formatadas no formato externo.

Com o Exemplo 6, obtemos resultados como esses:

```
>>> format(Vector2d(1, 1), 'p')
'<1.4142135623730951, 0.7853981633974483>'
>>> format(Vector2d(1, 1), '.3ep')
'<1.414e+00, 7.854e-01>'
>>> format(Vector2d(1, 1), '0.5fp')
'<1.41421, 0.78540>'
```

Como mostrou essa seção, não é difícil estender a Mini-Linguagem de Especificação de Formato para suportar tipos definidos pelo usuário.

Vamos agora passar a um assunto que vai além das aparências: tornar nosso `Vector2d` *hashable*, para podermos colocar vetores em conjuntos ou usá-los como chaves em um dict.

11.7. Um `Vector2d` *hashable*

Da forma como ele está definido até agora, as instâncias de nosso `Vector2d` não são *hashable*, então não podemos colocá-las em um set:

```
>>> v1 = Vector2d(3, 4)
>>> hash(v1)
Traceback (most recent call last):
...
TypeError: unhashable type: 'Vector2d'
>>> set([v1])
Traceback (most recent call last):
...
TypeError: unhashable type: 'Vector2d'
```

Para tornar um `Vector2d` *hashable*, precisamos implementar `__hash__` (`__eq__` também é necessário; já codamos esse método). Além disso, precisamos tornar imutáveis as instâncias do vetor, como vimos na «Seção 3.4.1» [fpy.li/8t] (vol.1).

Nesse momento, qualquer um pode fazer `v1.x = 7`. Nada no código proíbe modificar um `Vector2d`. Mas o comportamento que queremos é o seguinte:

```
>>> v1.x, v1.y
(3.0, 4.0)
>>> v1.x = 7
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

Faremos isso transformando os componentes `x` e `y` em propriedades apenas para leitura no Exemplo 7.

Exemplo 7. `vector2d_v3.py`: só as mudanças necessárias para tornar `Vector2d` imutável aparecem aqui; a listagem completa está no Exemplo 11

```
class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x) ①
        self.__y = float(y)

    @property ②
    def x(self): ③
        return self.__x ④

    @property ⑤
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y)) ⑥

    # remaining methods: same as previous Vector2d
```

- ① Usa exatamente dois sublinhados como prefixo (com zero ou um sublinhado como sufixo), para tornar um atributo privado.^[6]
- ② O decorador `@property` marca o método *getter* de uma propriedade.
- ③ O método *getter* tem nome da propriedade pública que ele expõe: `x`.
- ④ Apenas devolve `self.__x`.
- ⑤ Repete a mesma fórmula para a propriedade `y`.
- ⑥ Todos os métodos que apenas leem os componentes `x` e `y` podem continuar lendo as propriedades públicas através de `self.x` e `self.y` em vez de usar os atributos privados. Por isso omiti o resto da classe.



`Vector.x` e `Vector.y` são exemplos de propriedades apenas para leitura. Propriedades para leitura/escrita serão tratadas no «Capítulo 22» [fpy.li/22] (vol.3), onde mergulhamos mais fundo no decorador `@property`.

Agora que nossos vetores estão razoavelmente protegidos contra mutação acidental, podemos implementar o método `__hash__`. Ele deve devolver um `int` e, idealmente, levar em consideração os hashes dos atributos do objeto usados também no método `__eq__`, pois objetos que são considerados iguais ao serem comparados devem ter o mesmo *hash*. A documentação [fpy.li/66] do método especial `__hash__` sugere computar o *hash* de uma tupla com os componentes, e é isso que fazemos no Exemplo 8.

Exemplo 8. vector2d_v3.py: implementação de hash

```
# inside class Vector2d:

def __hash__(self):
    return hash((self.x, self.y))
```

Com o acréscimo do método `__hash__`, temos agora vetores *hashable*:

```
>>> v1 = Vector2d(3, 4)
>>> v2 = Vector2d(3.1, 4.2)
>>> hash(v1), hash(v2)
(1079245023883434373, 1994163070182233067)
>>> {v1, v2}
{Vector2d(3.1, 4.2), Vector2d(3.0, 4.0)}
```



Não é estritamente necessário implementar propriedades ou proteger de alguma forma os atributos de instância para criar um tipo *hashable*. Só é necessário implementar corretamente `__hash__` e `__eq__`. Mas, como o valor de um objeto *hashable* nunca deve mudar, é um bom motivo para aprender a criar propriedades apenas para leitura (*read only*).

Se você criar um tipo com que tem um valor numérico escalar, você pode implementar os métodos `__int__` e `__float__`, invocados pelos construtores `int()` e `float()`, que são usados, em alguns contextos, para conversão de tipo. Há também o método `__complex__`, para suportar o construtor embutido `complex()`. Talvez `Vector2d` pudesse oferecer o `__complex__`, mas deixo isso como um exercício para vocês.

11.8. Suportando o casamento de padrões posicionais

Até aqui, instâncias de `Vector2d` são compatíveis com o casamento de padrões com instâncias de classe—vistos na «Seção 5.8.2» [fpy.li/8a] (vol.1).

No Exemplo 9, os padrões nomeados funcionam como esperado.

Exemplo 9. Padrões nomeados para sujeitos `Vector2d`—requer Python 3.10

```
def keyword_pattern_demo(v: Vector2d) -> None:
    match v:
        case Vector2d(x=0, y=0):
            print(f'{v!r} is null')
        case Vector2d(x=0):
            print(f'{v!r} is vertical')
        case Vector2d(y=0):
            print(f'{v!r} is horizontal')
        case Vector2d(x=x, y=y) if x==y:
            print(f'{v!r} is diagonal')
        case _:
            print(f'{v!r} is awesome')
```

Entretanto, se tentamos usar um padrão posicional, como esse:

```
case Vector2d(_, 0):
    print(f'{v!r} is horizontal')
```

o resultado é esse:

```
TypeError: Vector2d() accepts 0 positional sub-patterns (1 given)
```

Para resolver, criamos um atributo de classe `__match_args__`:

```
class Vector2d:
    __match_args__ = ('x', 'y')
```

O atributo `__match_args__` tem os nomes dos atributos de instância na ordem em que eles serão usados no casamento de padrões posicionais.

Agora podemos escrever menos código ao criar padrões para casar com sujeitos `Vector2d`, como no Exemplo 10.

Exemplo 10. Padrões posicionais para sujeitos `Vector2d`—requer Python 3.10

```
def positional_pattern_demo(v: Vector2d) -> None:
    match v:
        case Vector2d(0, 0):
            print(f'{v!r} is null')
        case Vector2d(0):
            print(f'{v!r} is vertical')
        case Vector2d(_, 0):
            print(f'{v!r} is horizontal')
        case Vector2d(x, y) if x==y:
            print(f'{v!r} is diagonal')
        case _:
            print(f'{v!r} is awesome')
```

O atributo de classe `__match_args__` não precisa incluir todos os atributos públicos de instância. Em especial, se o `__init__` da classe tem argumentos obrigatórios e opcionais, que são depois vinculados a atributos de instância, pode ser razoável nomear apenas os argumentos obrigatórios em `__match_args__`, omitindo os opcionais.

Agora vamos revisar tudo o que programamos até aqui no `Vector2d`.

11.9. Listagem completa `Vector2d`, versão 3

Já estamos trabalhando no `Vector2d` há algum tempo, mostrando apenas trechos isolados. O Exemplo 11 é uma listagem completa e consolidada de `vector2d_v3.py`, incluindo os doctests que usei durante o desenvolvimento.

Exemplo 11. `vector2d_v3.py`: o módulo completo

```
"""
A two-dimensional vector class

>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y)
3.0 4.0
>>> x, y = v1
>>> x, y
(3.0, 4.0)
>>> v1
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1))
>>> v1 == v1_clone
True
>>> print(v1)
(3.0, 4.0)
>>> octets = bytes(v1)
>>> octets
b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
>>> abs(v1)
5.0
>>> bool(v1), bool(Vector2d(0, 0))
(True, False)

Test of `frombytes()` class method:

>>> v1_clone = Vector2d.frombytes(bytes(v1))
>>> v1_clone
Vector2d(3.0, 4.0)
>>> v1 == v1_clone
True

Tests of `format()` with Cartesian coordinates:

>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
```

```
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

Tests of the ``.angle()`` method::

```
>>> Vector2d(0, 0).angle()
0.0
>>> Vector2d(1, 0).angle()
0.0
>>> epsilon = 10**-8
>>> abs(Vector2d(0, 1).angle() - math.pi/2) < epsilon
True
>>> abs(Vector2d(1, 1).angle() - math.pi/4) < epsilon
True
```

Tests of ``.format()`` with polar coordinates:

```
>>> format(Vector2d(1, 1), 'p') # doctest:+ELLIPSIS
'<1.414213..., 0.785398...>'
>>> format(Vector2d(1, 1), '.3ep')
'<1.414e+00, 7.854e-01>'
>>> format(Vector2d(1, 1), '0.5fp')
'<1.41421, 0.78540>'
```

Tests of ``.x`` and ``.y`` read-only properties:

```
>>> v1.x, v1.y
(3.0, 4.0)
>>> v1.x = 123
Traceback (most recent call last):
...
AttributeError: property 'x' of 'Vector2d' object has no setter
```

Tests of hashing:

```
>>> v1 = Vector2d(3, 4)
>>> v2 = Vector2d(3.1, 4.2)
>>> len({v1, v2})
2
```

```
"""
```

```
from array import array
import math
```

```
class Vector2d:
```

```
    __match_args__ = ('x', 'y')
```

```
    typecode = 'd'
```

```
    def __init__(self, x, y):
        self.__x = float(x)
        self.__y = float(y)
```

```
    @property
    def x(self):
        return self.__x
```

```
    @property
    def y(self):
        return self.__y
```

```
    def __iter__(self):
        return (i for i in (self.x, self.y))
```

```
    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)
```

```
    def __str__(self):
        return str(tuple(self))
```

```
    def __bytes__(self):
        return (self.typecode.encode('ascii') +
                bytes(array(self.typecode, self)))
```

```
    def __eq__(self, other):
        return tuple(self) == tuple(other)
```

```
    def __hash__(self):
        return hash((self.x, self.y))
```

```

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

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

def angle(self):
    return math.atan2(self.y, self.x)

def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('p'):
        fmt_spec = fmt_spec[:-1]
        coords = (abs(self), self.angle())
        outer_fmt = '<{}, {}>'
    else:
        coords = self
        outer_fmt = '({}, {})'
    components = (format(c, fmt_spec) for c in coords)
    return outer_fmt.format(*components)

@classmethod
def frombytes(cls, octets):
    typecode = chr(octets[0])
    memv = memoryview(octets[1:]).cast(typecode)
    return cls(*memv)

```

Recordando, nessa seção e nas anteriores vimos alguns dos métodos especiais essenciais que você pode querer implementar para oferecer um objeto completo.



Você deve implementar os métodos especiais que forem úteis em sua aplicação. Os usuários finais não se importam se os objetos que da aplicação são pythônicos ou não.

Por outro lado, se suas classes são parte de uma biblioteca para ser usada por outros programadores Python, você não tem como adivinhar como eles vão usar seus objetos. E os usuários de sua biblioteca estarão esperando os comportamentos pythônicos que descrevemos aqui.

Como programado no Exemplo 11, `Vector2d` é um exemplo didático com uma longa lista de métodos especiais relacionados à representação de objetos, não um modelo para qualquer classe que você vai escrever.

Na próxima seção, deixamos o `Vector2d` de lado por um tempo para discutir o design e as limitações do mecanismo de atributos privados no Python—o prefixo de duplo sublinhado em `self.__x`.

11.10. Atributos privados e "protegidos" no Python

Em Python, não há como criar variáveis privadas como as criadas com o modificador `private` em Java. O que temos no Python é um mecanismo simples para prevenir que um atributo "privado" em uma subclasse seja acidentalmente sobrescrito.

Considere o seguinte cenário: alguém escreveu uma classe chamada `Dog`, que usa um atributo de instância `mood` internamente, sem expô-lo. Você precisa criar a uma subclasse `Beagle` de `Dog`. Se você criar seu próprio atributo de instância `mood`, sem saber da colisão de nomes, vai afetar o atributo `mood` usado pelos métodos herdados de `Dog`. Isso seria bem complicado de depurar.

Para prevenir esse tipo de problema, se você nomear o atributo de instância no formato `__mood` (dois sublinhados iniciais e zero ou no máximo um sublinhado no final), Python armazena o nome no `__dict__` da instância, prefixado com um sublinhado seguido do nome da classe. Na classe `Dog`, por exemplo, `__mood` se torna `_Dog__mood` e em `Beagle` ele será `_Beagle__mood`. Este mecanismo do Python é conhecido pela encantadora alcunha de *name mangling* (desfiguração de nome).

O Exemplo 12 mostra o resultado na classe `Vector2d` do Exemplo 7.

Exemplo 12. Nomes de atributos privados são "desfigurados" no dict

```
>>> v1 = Vector2d(3, 4)
>>> v1.__dict__
{'_Vector2d__y': 4.0, '_Vector2d__x': 3.0}
>>> v1._Vector2d__x
3.0
```

A desfiguração do nome oferece proteção, não segurança: evita acessos acidentais, mas não intencionais. A Figura 1 mostra um dispositivo de proteção.

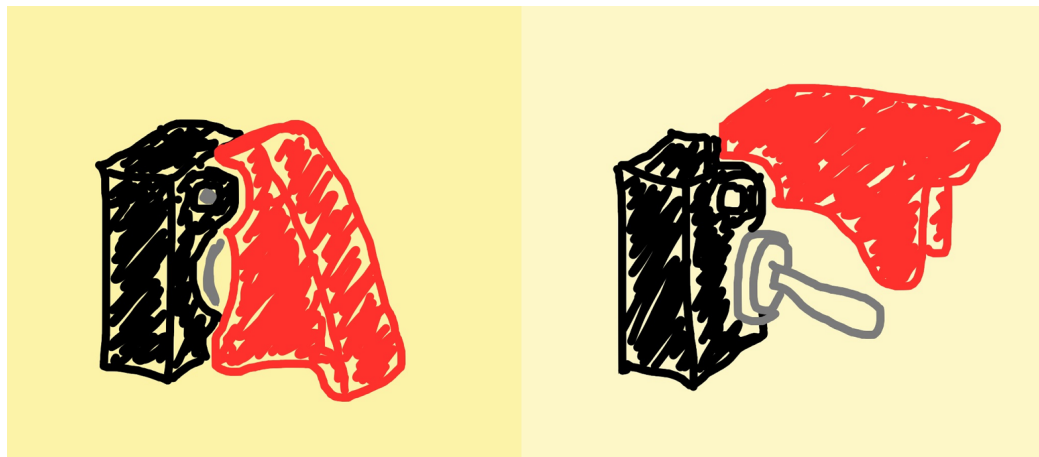


Figura 1. A capa sobre um interruptor é um dispositivo de proteção, não de segurança: previne acidentes, não sabotagem

Qualquer um que saiba como os nomes privados são modificados pode ler o atributo privado diretamente, como mostra a última linha do Exemplo 12. Este conhecimento é útil para depuração e serialização. Também pode ser usado para atribuir um valor a um componente privado de um `Vector2d`, escrevendo `v1._Vector2d__x = 7`. Mas se você estiver fazendo isso num código em produção, não poderá reclamar se alguma coisa explodir.

A funcionalidade de desfiguração de nomes não é amada por todos os pythonistas, nem tampouco a aparência estranha de nomes escritos como `self.__x`. Muitos preferem evitar essa sintaxe e usar apenas um sublinhado no prefixo para "proteger" atributos por convenção: `self._x`. Críticos da desfiguração automática com o sublinhado duplo dizem que preocupações com modificações acidentais a atributos devem ser tratadas através de convenções de nomenclatura. O criador do *pip*, *virtualenv* e outros projetos importantes, Ian Bicking, escreveu:

Nunca, de forma alguma, use dois sublinhados como prefixo. Isso é irritantemente privado. Se colisão de nomes for uma preocupação, use desfiguração explícita de nomes em seu lugar (por exemplo, `_MyThing_blahblah`). Isso é essencialmente a mesma coisa que o sublinhado duplo, mas é transparente enquanto o sublinhado duplo é obscuro.^[7]

O prefixo de sublinhado único não tem nenhum significado especial para o interpretador Python, quando usado em nomes de atributo. Mas essa é uma convenção muito presente entre programadores Python: tais atributos não devem ser acessados de fora da classe.^[8] É fácil respeitar a privacidade de um objeto que marca seus atributos com um único `_`, da mesma forma que é fácil respeitar a convenção de tratar como constantes as variáveis com nomes inteiramente em maiúsculas.

Atributos com um único `_` como prefixo são chamados "protegidos" em algumas partes da documentação de Python, por exemplo na documentação do módulo *gettext* [fpy.li/68]. A prática de "proteger" atributos por convenção com a forma `self._x` é muito difundida, mas chamar isso de atributo "protegido" não é tão comum. Alguns até falam em atributo "privado" nesses casos.

Concluindo: os componentes de `Vector2d` são "privados" e nossas instâncias de `Vector2d` são "imutáveis"—com aspas irônicas—pois não há como tornar uns realmente privados e outras realmente imutáveis.^[9]

Vamos agora voltar à nossa classe `Vector2d`. Na próxima seção trataremos de um atributo especial (não um método) que reduz o uso de memória das instâncias, sem afetar muito sua interface pública: `__slots__`.

11.11. Economizando memória com `__slots__`

Por default, Python armazena os atributos de cada instância em um `dict` chamado `__dict__`. Como vimos em «Seção 3.9» [fpy.li/82] (vol.1), um `dict` ocupa um espaço significativo de memória, mesmo com as otimizações mencionadas naquela seção. Mas se você definir um atributo de classe chamado `__slots__` com uma sequência de nomes de atributos, Python usará um modelo alternativo de armazenamento para os atributos de instância: os atributos nomeados em `__slots__` serão armazenados em um array de referências oculto, que usa menos memória que um `dict`. Vamos ver como isso funciona através de alguns exemplos simples, começando pelo Exemplo 13.

Exemplo 13. A classe Pixel usa __slots__

```
>>> class Pixel:
...     __slots__ = ('x', 'y') ①
...
>>> p = Pixel() ②
>>> p.__dict__ ③
Traceback (most recent call last):
...
AttributeError: 'Pixel' object has no attribute '__dict__'
>>> p.x = 10 ④
>>> p.y = 20
>>> p.color = 'red' ⑤
Traceback (most recent call last):
...
AttributeError: 'Pixel' object has no attribute 'color'
```

- ① `__slots__` deve estar presente quando a classe é criada; acrescentá-lo ou modificá-lo posteriormente não tem efeito. Os nomes de atributos podem estar em uma tuple ou em uma list. Prefiro usar uma tuple, para deixar claro que não faz sentido modificá-la.
- ② Cria uma instância de `Pixel` para testar, pois os efeitos de `__slots__` são vistos nas instâncias.
- ③ Primeiro efeito: instâncias de `Pixel` não têm um `__dict__`.
- ④ Define normalmente os atributos `p.x` e `p.y`.
- ⑤ Segundo efeito: tentar definir um atributo não listado em `__slots__` gera um `AttributeError`.

Até aqui, tudo bem. Agora vamos criar uma subclasse de `Pixel`, no Exemplo 14, para ver o lado contraintuitivo de `__slots__`.

Exemplo 14. `OpenPixel` é uma subclasse de `Pixel`

```
>>> class OpenPixel(Pixel): ①
...     pass
...
>>> op = OpenPixel()
>>> op.__dict__ ②
{}
```



```

>>> op.x = 8 ③
>>> op.__dict__ ④
{}
>>> op.x ⑤
8
>>> op.color = 'green' ⑥
>>> op.__dict__ ⑦
{'color': 'green'}

```

- ① OpenPixel não declara qualquer atributo próprio.
- ② Surpresa: instâncias de OpenPixel têm um `__dict__`.
- ③ Se você definir o atributo `x` (nomeado no `__slots__` da classe base `Pixel`)...
- ④ ...ele não será armazenado no `__dict__` da instância...
- ⑤ ...mas sim no array oculto de referências na instância.
- ⑥ Se você definir um atributo não nomeado no `__slots__`...
- ⑦ ...ele será armazenado no `__dict__` da instância.

O Exemplo 14 mostra que o efeito de `__slots__` é herdado apenas parcialmente por uma subclasse. Para se assegurar que instâncias de uma subclasse não tenham o `__dict__`, é preciso declarar `__slots__` novamente na subclasse.

Se você declarar `__slots__ = ()` (uma tupla vazia), as instâncias da subclasse não terão um `__dict__` e só aceitarão atributos nomeados no `__slots__` da classe base.

Se você quiser que uma subclasse tenha atributos adicionais, basta nomeá-los em `__slots__`, como mostra o Exemplo 15.

Exemplo 15. The ColorPixel, another subclass of Pixel

```

>>> class ColorPixel(Pixel):
...     __slots__ = ('color',) ①
>>> cp = ColorPixel()
>>> cp.__dict__ ②
Traceback (most recent call last):
...
AttributeError: 'ColorPixel' object has no attribute '__dict__'
>>> cp.x = 2
>>> cp.color = 'blue' ③

```

```
>>> cp.flavor = 'banana'
Traceback (most recent call last):
...
AttributeError: 'ColorPixel' object has no attribute 'flavor'
```

- ① Em resumo, o `__slots__` da superclasse é adicionado ao `__slots__` da classe atual. Não esqueça que tuplas com um único elemento devem ter uma vírgula no final.
- ② Instâncias de `ColorPixel` não tem um `__dict__`.
- ③ Você pode definir atributos declarados no `__slots__` dessa classe e nos de suas superclasses, mas nenhum outro.

Curiosamente, também é possível colocar o nome `'__dict__'` em `__slots__`. Neste caso, as instâncias vão manter os atributos nomeados em `__slots__` num array de referências da instância, mas também vão aceitar atributos criados dinamicamente, que serão armazenados `__dict__`, como de costume. Isso é necessário para usar o decorador `@cached_property`, tratado na «Seção 22.3.5» [fpy.li/7x] (vol.3).

Naturalmente, incluir `'__dict__'` em `__slots__` pode anular a economia de memória, dependendo do número de atributos estáticos e dinâmicos em cada instância, e de como eles são usados. Otimização descuidada é pior que otimização prematura: aumenta a complexidade sem trazer benefícios.

Outro atributo de instância especial que você pode querer incluir é `__weakref__`, necessário para que objetos suportem referências fracas (mencionadas brevemente na «Seção 6.6» [fpy.li/86] (vol.1)). Esse atributo existe por default em instâncias de classes definidas pelo usuário. Entretanto, se a classe define `__slots__`, e é necessário que as instâncias possam ser alvo de referências fracas, então é preciso incluir `__weakref__` entre os atributos nomeados em `__slots__`.

Vejamos agora o efeito de `__slots__` em `Vector2d`.

11.11.1. Uma medida simples da economia gerada por `__slots__`

Exemplo 16 mostra a implementação de `__slots__` em `Vector2d`.

Exemplo 16. `vector2d_v3_slots.py`: o atributo `__slots__` é a única adição a `Vector2d`

```
class Vector2d:
    __match_args__ = ('x', 'y') ①
    __slots__ = ('_x', '_y') ②

    typecode = 'd'
    # methods are the same as previous version
```

- ① `__match_args__` lista os nomes dos atributos públicos, para casamento de padrões posicionais.
- ② `__slots__`, por outro lado, lista os nomes dos atributos de instância, que neste caso são atributos privados.

Para medir a economia de memória, escrevi o script `mem_test.py`. Ele recebe, como argumento de linha de comando, o nome de um módulo com uma variante da classe `Vector2d`, e usa uma compreensão de lista para criar uma `list` com 10.000.000 de instâncias de `Vector2d`. Na primeira execução, vista no Exemplo 17, usei `vector2d_v3.Vector2d` (do Exemplo 7); na segunda execução usei a versão com `__slots__` do Exemplo 16.

Exemplo 17. `mem_test.py` cria 10 milhões de instâncias de `Vector2d`, usando a classe definida no módulo nomeado

```
$ time python3 mem_test.py vector2d_v3
Selected Vector2d type: vector2d_v3.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      6,983,680
  Final RAM usage:  1,666,535,424

real    0m11.990s
user    0m10.861s
sys 0m0.978s
$ time python3 mem_test.py vector2d_v3_slots
Selected Vector2d type: vector2d_v3_slots.Vector2d
Creating 10,000,000 Vector2d instances
```

```
Initial RAM usage:    6,995,968
Final RAM usage:      577,839,104

real    0m8.381s
user    0m8.006s
sys    0m0.352s
```

Como revela o Exemplo 17, o uso de RAM do script cresce para 1,55 GB quando o `__dict__` de instância é usado em cada uma das 10 milhões de instâncias de `Vector2d`, mas isso se reduz a 551 MB quando `Vector2d` tem um atributo `__slots__`. A versão com `__slots__` também é mais rápida. O script *mem_test.py* neste teste lida basicamente com o carregamento do módulo, a medição da memória utilizada e a formatação de resultados. O código-fonte pode ser encontrado no repositório *fluentpython/example-code-2e* [fpy.li/11-11].



Se você precisa manipular milhões de objetos com dados numéricos, deveria na verdade estar usando os arrays da NumPy (veja a «Seção 2.10.3» [fpy.li/8h] (vol.1)), que são eficientes no de uso de memória, e também tem funções para processamento numérico extremamente otimizadas, muitas das quais operam sobre o array inteiro em paralelo. Projetei a classe `Vector2d` apenas como um contexto para a discussão de métodos especiais, pois sempre que possível tento evitar exemplos vagos com `Foo` e `Bar`.

11.11.2. Resumindo os problemas com `__slots__`

O atributo de classe `__slots__` pode proporcionar uma economia significativa de memória se usado corretamente, mas existem algumas ressalvas:

- É preciso lembrar de redeclarar `__slots__` em cada subclasse, para evitar que suas instâncias tenham um `__dict__`.
- Instâncias só poderão ter os atributos listados em `__slots__`, a menos que `__dict__` seja incluído em `__slots__` (mas isso pode anular a economia de memória).
- Classe que usam `__slots__` não podem usar o decorador `@cached_property`, a menos que nomeiem `__dict__` explicitamente em `__slots__`.

- Instâncias não podem ser alvo de referências fracas, a menos que `__weakref__` seja incluído em `__slots__`.

O último tópico do capítulo trata de sobrescrever de um atributo de classe em instâncias e subclasses.

11.12. Sobrescrevendo atributos de classe

Um recurso característico de Python é a forma como atributos de classe podem ser usados como valores default para atributos de instância. `Vector2d` contém o atributo de classe `typecode`. No método `__bytes__` eu o acesso como `self.typecode` de propósito. As instâncias de `Vector2d` são criadas sem um atributo `typecode` próprio, então a expressão `self.typecode` vai, por default, ler atributo de classe `Vector2d.typecode`.

Agora, quando atribuímos valor a um atributo na instância—por exemplo, um atributo `typecode` na instância—o atributo de classe com o mesmo nome não é alterado. Mas daí em diante, sempre a expressão `self.typecode` aparecer, o `typecode` da instância será usado, na prática escondendo o atributo de classe de mesmo nome. Isso abre a possibilidade de customizar uma instância individual com um `typecode` diferente do padrão definido na classe `Vector2d`.

O `Vector2d.typecode` default é `'d'`: isso significa que cada componente do vetor será representado como um número de ponto flutuante de precisão dupla e 8 bytes de tamanho quando for exportado para bytes. Se definirmos o `typecode` de uma instância `Vector2d` como `'f'` antes da exportação, cada componente será exportado como um número de ponto flutuante de precisão simples e 4 bytes de tamanho. O Exemplo 18 demonstra isso.



Estamos falando de criar um novo atributo em uma instância, por isso o Exemplo 18 usa a implementação de `Vector2d` sem `__slots__`, como aparece no Exemplo 11.

Exemplo 18. Customizando uma instância para redefinir o atributo typecode, sobrescrevendo o valor do typecode da classe Vector2d

```
>>> from vector2d_v3 import Vector2d
>>> v1 = Vector2d(1.1, 2.2)
>>> dumpd = bytes(v1)
>>> dumpd
b'd\x9a\x99\x99\x99\x99\x99\xf1?\x9a\x99\x99\x99\x99\x01@'
>>> len(dumpd) ①
17
>>> v1.typecode = 'f' ②
>>> dumpf = bytes(v1)
>>> dumpf
b'f\xcd\xcc\x8c?\xcd\xcc\x0c@'
>>> len(dumpf) ③
9
>>> Vector2d.typecode ④
'd'
```

- ① A representação default em bytes tem 17 bytes de comprimento.
- ② Define typecode como 'f' na instância v1.
- ③ Agora bytes tem 9 bytes de comprimento.
- ④ Vector2d.typecode não foi modificado; apenas a instância v1 usa o typecode 'f'.

Isso deixa claro porque a exportação para bytes de um Vector2d tem um prefixo typecode: queríamos suportar a exportação de vetores com números de diferentes precisões.

Para modificar um atributo de classe, é preciso redefini-lo diretamente na classe, e não através de uma instância. Poderíamos modificar o typecode default para todas as instâncias (que não tenham seu próprio typecode) assim:

```
>>> Vector2d.typecode = 'f'
```

Porém, no Python, há uma maneira idiomática de obter um efeito mais permanente, e de ser mais explícito sobre a modificação. Como atributos de classe são públicos, eles são herdados por subclasses. Então é uma prática

comum fazer a subclasse customizar um atributo da classe. As views baseadas em classes do Django usam amplamente essa técnica. O Exemplo 19 mostra como se faz.

Exemplo 19. O ShortVector2d é uma subclasse de Vector2d, que apenas sobrescreve o typecode default

```
>>> from vector2d_v3 import Vector2d
>>> class ShortVector2d(Vector2d): ①
...     typecode = 'f'
...
>>> sv = ShortVector2d(1/11, 1/27) ②
>>> sv
ShortVector2d(0.09090909090909091, 0.037037037037037035) ③
>>> len(bytes(sv)) ④
9
```

- ① Cria ShortVector2d como uma subclasse de Vector2d apenas para sobrescrever o atributo de classe typecode.
- ② Cria sv, uma instância de ShortVector2d, para demonstração.
- ③ Verifica o repr de sv.
- ④ Verifica que a quantidade de bytes exportados é 9, e não 17 como antes.

Esse exemplo também explica porque não atribui a constante 'Vector2d' ao class_name no método __repr__, optando por obter o nome da classe através de type(self).__name__:

```
# inside class Vector2d:

def __repr__(self):
    class_name = type(self).__name__
    return '{}({!r}, {!r})'.format(class_name, *self)
```

Se eu tivesse escrito o class_name explicitamente, subclasses de Vector2d como ShortVector2d teriam que sobrescrever __repr__ só para mudar o class_name. Lendo o nome do type da instância, tornei __repr__ mais seguro para ser herdado.

Aqui termina nossa conversa sobre a criação de uma classe simples, que aproveita modelo de dados de Python para se adaptar bem ao restante da linguagem: oferecendo diferentes representações do objeto, fornecendo um código de formatação customizado, expondo atributos somente para leitura e suportando `hash()` para se integrar a conjuntos e mapeamentos.

11.13. Resumo do capítulo

O objetivo desse capítulo foi demonstrar o uso dos métodos especiais e as convenções na criação de uma classe pythônica bem comportada.

Será `vector2d_v3.py` (do Exemplo 11) mais pythônica que `vector2d_v0.py` (do Exemplo 2)? A classe `Vector2d` em `vector2d_v3.py` com certeza utiliza mais recursos de Python. Mas, decidir qual das duas implementações de `Vector2d` é mais adequada, depende do contexto onde a classe será usada. No *Zen of Python*, Tim Peter escreveu:

Simples é melhor que complexo.

Um objeto deve ser tão simples quanto seus requisitos exigem—e não um desfile de recursos da linguagem. Se o código é parte de uma aplicação, deve se concentrar no que é necessário para atender os usuários finais, e nada mais. Se o código for parte de uma biblioteca para uso por outros programadores, então é razoável oferecer comportamentos esperados por pythonistas, implementados através de métodos especiais. Por exemplo, `__eq__` pode não ser um requisito do negócio, mas torna a classe mais fácil de testar.

Ao expandir o código do `Vector2d` meu objetivo foi criar um contexto para a discussão dos métodos especiais e outras convenções de programação em Python. Os exemplos neste capítulo demonstraram vários dos métodos especiais mencionados no «Capítulo 1» [fpy.li/1] (vol.1):

- Métodos de representação de strings e bytes: `__repr__`, `__str__`, `__format__` e `__bytes__`
- Métodos para reduzir um objeto a um número: `__abs__`, `__bool__` e `__hash__`
- O operador `__eq__`, para facilitar testes e permitir *hashing* (juntamente com `__hash__`)

Quando suportamos a conversão para bytes, também implementamos um construtor alternativo, `Vector2d.frombytes()`, que nos deu motivo para falar dos decoradores `@classmethod` (muito conveniente) e `@staticmethod` (não tão útil: funções a nível do módulo são mais simples). O método `frombytes` foi inspirado pelo método de mesmo nome na classe `array.array`.

Vimos que a Mini-Linguagem de Especificação de Formato [fpy.li/63] é extensível, ao implementarmos um método `__format__` que analisa uma especificação de formato passada para a função embutida `format(obj, fmt_spec)`, ou dentro de campos de substituição `f'{expr:fmt_spec}'` em f-strings, ou ainda strings usadas como alvo do método `str.format()`.

Para preparar que instâncias de `Vector2d` sejam *hashable*, fizemos um esforço para torná-las imutáveis, ao menos prevenindo modificações acidentais, programando os atributos `x` e `y` como privados, e expondo-os como propriedades para leitura apenas. Então implementamos `__hash__` usando a técnica recomendada, aplicando o operador `^` (xor) aos *hashes* dos atributos da instância.

Discutimos a seguir a economia de memória e as ressalvas de se declarar um atributo `__slots__` em `Vector2d`. Como o uso de `__slots__` tem efeitos colaterais, ele só faz real sentido quando é preciso processar um número muito grande de instâncias—pense em milhões de instâncias, não apenas milhares. Em muitos destes casos, usar a *pandas* [fpy.li/pandas] pode ser a melhor opção.

O último tópico tratado foi a sobrescrita de um atributo de classe acessado através das instâncias (por exemplo, `self.typecode`). Fizemos isso primeiro criando um atributo de instância, depois criando uma subclasse e sobrescrevendo o atributo no nível da classe.

Por todo o capítulo, aponte como escolhas de design nos exemplos foram baseadas no estudo das APIs dos objetos padrão de Python. Se esse capítulo pode ser resumido em uma só frase, seria essa:

Para criar objetos pythônicos, observe como se comportam os objetos do Python.

— Antigo provérbio chinês

11.14. Para saber mais

Este capítulo tratou de vários dos métodos especiais do modelo de dados, então naturalmente as referências primárias são as mesmas do «Capítulo 1» [fpy.li/1] (vol.1), onde tivemos uma ideia geral do mesmo tópico. Por conveniência, vou repetir aquelas indicações aqui:

Modelo de Dados, em A Referência da Linguagem Python

A maioria dos métodos usados neste capítulo estão documentados em «Customização básica» [fpy.li/69].

***Python in a Nutshell*, 3rd ed. (Martelli, Ravenscroft & Holden)**

Trata com profundidade dos métodos especiais .

***Python Cookbook*, 3rd ed. (Beazley & Jones)**

Práticas modernas de Python demonstradas através de receitas. Especialmente o Capítulo 8, *Classes and Objects*, que traz várias receitas relacionadas às discussões deste capítulo.

***Python Essential Reference*, 4th ed. (Beazley)**

Trata do modelo de dados em detalhes, apesar de falar apenas de Python 2.6 e do 3.0 (na quarta edição). Todos os conceitos fundamentais são os mesmos, e a maior parte das APIs do Modelo de Dados não mudou nada desde Python 2.2, quando aconteceu a unificação dos tipos embutidos e classes definidas pelo usuário.

Em 2015—o ano que terminei a primeira edição de *Python Fluente*—Hynek Schlawack começou a desenvolver o pacote `attrs`. Da documentação de `attrs`:

attrs é um pacote Python que vai trazer de volta a alegria de criar classes, liberando você do tedioso trabalho de implementar protocolos de objeto (também conhecidos como métodos dunder)

Mencionei `attrs` como uma alternativa mais poderosa ao `@dataclass` na «Seção 5.10» [fpy.li/85] (vol.1). As fábricas de classes de dados do «Capítulo 5» [fpy.li/5] (vol.1), assim como `attrs`, automaticamente equipam suas classes com vários métodos especiais. Mas saber como programar métodos especiais ainda é essencial para entender o que aqueles pacotes fazem, para decidir se você

realmente precisa deles e para sobrescrever os métodos que eles geram, quando necessário.

Vimos neste capítulo todos os métodos especiais relacionados à representação de objetos, exceto `__index__` e `__fspath__`. Discutiremos `__index__` no Capítulo 12, na Seção 12.5.2. Não vou tratar de `__fspath__`. Para aprender sobre esse método, veja a *PEP 519—Adding a file system path protocol* [fpy.li/pep519] (Adicionando um protocolo de caminho de sistema de arquivos).

A necessidade de diferentes strings de representação para objetos apareceu primeiro em Smalltalk. O artigo *How to Display an Object as a String: printString and displayString* [fpy.li/11-13] (Como Exibir um Objeto como uma String: `printString` e `displayString`), de Bobby Woolf, discute a implementação dos métodos `printString` e `displayString` na linguagem Smalltalk em 1996. Foi lá que encontrei as descrições "como o desenvolvedor quer vê-lo" para a `repr()` e "como o usuário quer vê-lo" para a `str()`, na Seção 11.2.

Ponto de Vista

Proteção versus segurança em atributos privados

O Perl não tem nenhum amor por privacidade forçada. Ele preferiria que você não entrasse em sua sala de estar [apenas] por não ter sido convidado, e não porque ele tem uma espingarda.

— Larry Wall, criador da linguagem Perl

Python e Perl estão em polos opostos em vários aspectos, mas Guido e Larry parecem concordar sobre a privacidade de objetos.

Ensinando Python para muitos programadores Java ao longo do anos, percebi que muitos têm uma fé excessiva nas garantias de "privacidade" oferecidas pelo Java. Na verdade, os modificadores `private` e `protected` de Java normalmente fornecem defesas apenas contra acidentes (isto é, proteção). Eles só oferecem segurança contra ataques mal-intencionados se a aplicação for especialmente configurada e implantada sob um `SecurityManager` [fpy.li/11-15] de Java, e isso raramente acontece na prática, mesmo em instalações corporativas preocupadas com segurança.

Para provar meu argumento, considere a classe Java a seguir.

Exemplo 20. Confidential.java: uma classe Java com um campo privado chamado secret

```
public class Confidential {  
  
    private String secret = "";  
  
    public Confidential(String text) {  
        this.secret = text.toUpperCase();  
    }  
}
```

No Exemplo 20, armazeno o text no campo secret após convertê-lo todo para caixa alta, para deixar óbvio que o argumento text passado para o construtor sofre uma transformação antes de ser armazenado.

A verdadeira demonstração consiste em rodar *expose.py* com Jython. Este script usa introspecção (*reflection* ou reflexão no jargão de Java) para acessar o valor de um campo privado. O código aparece no Exemplo 21.

Exemplo 21. expose.py: código em Jython para ler o conteúdo de um campo privado em outra classe

```
#!/usr/bin/env jython  
# NOTE: Jython is still Python 2.7 in late2020  
  
import Confidential  
  
message = Confidential('top secret text')  
secret_field = Confidential.getDeclaredField('secret')  
secret_field.setAccessible(True) # break the lock!  
print 'message.secret = ', secret_field.get(message)
```

Executando o Exemplo 21, o resultado é esse:

```
$ jython expose.py  
message.secret = TOP SECRET TEXT
```

A string 'TOP SECRET TEXT' foi lida do campo privado `secret` da classe `Confidential`.

Não há magia aqui: *expose.py* usa a API de reflexão de Java para obter uma referência para o campo privado chamado 'secret', e então chama `secret_field.setAccessible(True)` para tornar acessível seu conteúdo. A mesma coisa pode ser feita com código Java, claro (mas exige mais que o triplo de linhas; veja o arquivo *Expose.java* [fpy.li/11-16] no repositório de código [fpy.li/code] deste livro.

A chamada `.setAccessible(True)` só falhará se o script Jython ou o programa principal em Java (por exemplo, `Expose.class`) estiverem rodando sob a supervisão de um `SecurityManager` [fpy.li/11-15]. Mas, no mundo real, aplicações Java raramente são implantadas com um `SecurityManager`—com a exceção das *applets* Java, quando elas ainda eram suportadas pelos navegadores.

Meu ponto: também em Java, na prática os modificadores de controle de acesso oferecem proteção mas não segurança. Então relaxe e aprecie o poder dado a você pelo Python. E use esse poder com responsabilidade.

Propriedades ajudam a reduzir custos iniciais

Nas primeiras versões de `Vector2d`, os atributos `x` e `y` eram públicos, como são, por default, todos os atributos de instância e classe no Python. Naturalmente, os usuários de vetores precisam acessar seus componentes. Apesar de nossos vetores serem iteráveis e poderem ser desempacotados em um par de variáveis, também é desejável poder escrever `my_vector.x` e `my_vector.y` para obter cada componente.

Quando surge a necessidade de proteger os atributos `x` e `y`, implementamos propriedades, mas nada muda no restante do código ou na interface pública de `Vector2d`, como se verifica através dos doctests. Continuamos podendo acessar `my_vector.x` e `my_vector.y`.

Isso mostra que podemos sempre iniciar o desenvolvimento de nossas classes da maneira mais simples possível, com atributos públicos, pois quando (ou se) for preciso impor restrições depois, com *getters* e *setters*, eles podem ser implementados usando propriedades, sem mudar nada no

código que já interage com nossos objetos através dos nomes que eram, inicialmente, simples atributos públicos como `x` e `y` em nosso exemplo.

Essa abordagem é o oposto daquilo que é encorajado pela linguagem Java: um programador Java não pode começar com atributos públicos simples e apenas mais tarde, se necessário, implementar propriedades, porque elas não existem naquela linguagem. Portanto, escrever *getters* e *setters* é a regra em Java—mesmo quando esses métodos não fazem nada de útil—porque a API não pode evoluir de atributos públicos simples para *getters* e *setters* sem quebrar todo o código que já use aqueles atributos.

Além disso, como Martelli, Ravenscroft e Holden observam no Python in a Nutshell 3rd ed. [fpy.li/pynut3], digitar chamadas a *getters* e *setters* por toda parte é patético. Você é obrigado a escrever coisas como:

```
>>> my_object.set_foo(my_object.get_foo() + 1)
```

Apenas para fazer isso:

```
>>> my_object.foo += 1
```

Ward Cunningham, inventor do wiki e um pioneiro da Programação Extrema (*Extreme Programming*), recomenda perguntar: "Qual a coisa mais simples que tem alguma chance de funcionar?" A ideia é se concentrar no objetivo.^[10] Implementar *setters* e *getters* desde o início é um desvio em relação ao objetivo. Em Python, podemos simplesmente usar atributos públicos, sabendo que podemos transformá-los mais tarde em propriedades, se essa necessidade surgir.

[1] Do post no blog de Faassen intitulado *What is Pythonic?* (O que é Pythônico?) [fpy.li/11-1]

[2] Usei `eval` para clonar o objeto apenas para demonstrar a sintaxe da string gerada por `repr`; para clonar uma instância, a função `copy.copy` é mais segura e rápida.

[3] Essa linha também poderia ser escrita assim: `yield self.x`; `yield self.y`. Terei mais a dizer sobre o método especial `__iter__`, sobre expressões geradoras e sobre a palavra reservada `yield` no «Capítulo 17» [fpy.li/17] (vol.3).

[4] Tivemos uma pequena introdução a `memoryview` e explicamos seu método `.cast` na «Seção 2.10.2» [fpy.li/8d] (vol.1).

[5] Leonardo Rochaël, um dos revisores técnicos deste livro, discorda de minha opinião desabonadora sobre o `staticmethod`, e recomenda como contra-argumento o post de blog *The Definitive Guide on How to Use Static, Class or Abstract Methods in Python* [fpy.li/11-2] (O Guia Definitivo sobre Como Usar Métodos Estáticos, de Classe ou Abstratos em Python), de Julien Danjou. O post de Danjou é muito bom; recomendo sua leitura. Mas não foi suficiente para mudar meu ponto de vista sobre `staticmethod`. Você terá que decidir por conta própria se vale ou não a pena usar `staticmethod`.

[6] Os prós e contras dos atributos privados são assunto da Seção 11.10, mais adiante.

[7] Do *Paste Style Guide* [fpy.li/11-8] (Guia de Estilo do Paste).

[8] Em módulos, um único `_` no início de um nome de nível superior tem sim um efeito: se você escrever `from mymod import *`, os nomes com um prefixo `_` não são importados de `mymod`. Entretanto, ainda é possível escrever `from mymod import _privatefunc`. Isso é explicado no *Tutorial de Python*, seção 6.1., "Mais sobre módulos" [fpy.li/67].

[9] Se você acha este estado de coisas deprimente e desejaria que Python fosse mais parecido com o Java nesse aspecto, nem leia minha discussão sobre a força relativa do modificador `private` de Java no Ponto de Vista.

[10] Veja *Simplest Thing that Could Possibly Work: A Conversation with Ward Cunningham, Part V* [fpy.li/11-14] (A Coisa Mais Simples que Poderia Funcionar: Uma Conversa com Ward Cunningham, Parte V).

Capítulo 12. Métodos especiais para sequências

Não queira saber se aquilo é um pato: veja se ele grasna-como-um pato, anda-como-um pato, etc., etc., dependendo de qual subconjunto de comportamentos de pato você precisa usar em seu jogo de palavras.
(comp.lang.python, Jul. 26, 2000)

— Alex Martelli

Neste capítulo, vamos criar uma classe `Vector`, para representar um vetor multidimensional—um avanço significativo sobre o `Vector2D` bidimensional do Capítulo 11. `Vector` vai se comportar como uma sequência plana imutável como outras que existem em Python. Seus elementos serão números de ponto flutuante, e ao final do capítulo a classe suportará o seguinte:

- O protocolo de sequência básico: `__len__` e `__getitem__`
- Representação abreviada de instâncias com muitos itens
- Suporte adequado a fatiamento, produzindo novas instâncias de `Vector`
- *Hashing* agregado, considerando cada elemento contido na sequência
- Um extensão customizada da linguagem de formatação

Também vamos implementar, com `__getattr__`, o acesso dinâmico a atributos, como forma de substituir as propriedades apenas para leitura que usamos no `Vector2d`—apesar disso não ser comum em sequências.

Além disso, teremos uma discussão conceitual sobre a ideia de protocolos como interfaces informais. Vamos discutir a relação entre protocolos e a tipagem pato (*duck typing*), e as implicações práticas disso na criação de seus próprios tipos.

12.1. Novidades neste capítulo

Não fiz grandes mudanças neste capítulo. Há uma breve discussão nova sobre o `typing.Protocol` em um quadro de dicas, no final da Seção 12.4.

Na Seção 12.5.2, a implementação do `__getitem__` no Exemplo 6 está mais concisa e robusta que o exemplo na primeira edição, graças ao *duck typing* e ao `operator.index`. Essa mudança foi replicada para as implementações seguintes de `Vector` aqui e no Capítulo 16.

Vamos começar.

12.2. Vector: uma sequência definida pelo usuário

Nossa estratégia na implementação de `Vector` será usar composição, não herança. Vamos armazenar os componentes em um array de números de ponto flutuante, e implementar os métodos necessários para que nossa classe `Vector` se comporte como uma sequência plana imutável.

Mas antes de implementar os métodos de sequência, vamos desenvolver uma implementação básica de `Vector` compatível com nossa classe `Vector2d`, vista anteriormente—exceto onde tal compatibilidade não fizer sentido.

Aplicações de vetores além de três dimensões

Quem precisa de vetores com 1.000 dimensões? Vetores N-dimensionais (com valores grandes de N) são bastante utilizados em recuperação de informação, onde documentos e consultas textuais são representados como vetores, com uma dimensão para cada palavra. Isso se chama Modelo vetorial [fpy.li/6b]. Nesse modelo, a métrica fundamental de relevância é a *similaridade de cosseno*—o cosseno do ângulo entre um vetor que representa a consulta e um vetor representando um documento. Conforme o ângulo diminui, o valor do cosseno aumenta, indicando a relevância do documento para aquela consulta: cosseno próximo de 1 significa alta relevância; próximo de 0 indica baixa relevância.

Dito isto, a classe `Vector` nesse capítulo é um exemplo didático. O objetivo é apenas demonstrar alguns métodos especiais de Python no contexto de um tipo sequência, sem grandes conceitos matemáticos.

A NumPy e a SciPy são as ferramentas que você precisa para fazer cálculos

vetoriais em aplicações reais. O pacote *gensim* [fpy.li/12-2] do PyPi, de Radim Řehůřek, implementa a modelagem de espaço vetorial para processamento de linguagem natural e recuperação de informação, usando a NumPy e a SciPy.

12.3. Vector versão #1: compatível com Vector2d

A primeira versão de `Vector` deve ser tão compatível quanto possível com nossa classe `Vector2d` desenvolvida anteriormente.

Entretanto, pela própria natureza das classes, o construtor de `Vector` não é compatível com o construtor de `Vector2d`. Poderíamos fazer `Vector(3, 4)` e `Vector(3, 4, 5)` funcionarem, recebendo argumentos arbitrários com `*args` em `__init__`. Mas a melhor prática para um construtor de sequências é receber os dados através de um argumento iterável, como fazem todos os tipos embutidos de sequências. O Exemplo 1 mostra algumas maneiras de instanciar objetos do nosso novo `Vector`.

Exemplo 1. Testes de `Vector.__init__` e `Vector.__repr__`

```
>>> Vector([3.1, 4.2])
Vector([3.1, 4.2])
>>> Vector((3, 4, 5))
Vector([3.0, 4.0, 5.0])
>>> Vector(range(10))
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
```

Exceto pela nova assinatura do construtor, verifiquei que todos os testes realizados com `Vector2d` (por exemplo, `Vector2d(3, 4)`) passam e produzem os mesmos resultados com um `Vector` de dois componentes, como `Vector([3, 4])`.



Quando um `Vector` tem mais de seis componentes, a string produzida por `repr()` é abreviada com `...`, como visto na última linha do Exemplo 1. Isso é fundamental para qualquer tipo de coleção que possa conter um número grande de itens, pois `repr` é usado na depuração—e você não quer que um único objeto grande ocupe milhares de linhas em seu console ou

arquivo de log. Use o módulo `reprlib` para produzir representações de tamanho limitado, como no Exemplo 2. O módulo `reprlib` se chamava `repr` no Python 2.7.

O Exemplo 2 é a primeira versão de `Vector` baseada no Exemplo 2 e Exemplo 3 do Capítulo 11.

Exemplo 2. `vector_v1.py`: baseado em `vector2d_v1.py`

```
from array import array
import reprlib
import math

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components) ①

    def __iter__(self):
        return iter(self._components) ②

    def __repr__(self):
        components = reprlib.repr(self._components) ③
        components = components[components.find('['):-1] ④
        return f'Vector({components})'

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (self.typecode.encode('ascii') +
                bytes(self._components)) ⑤

    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __abs__(self):
        return math.hypot(*self) ⑥

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

```

@classmethod
def frombytes(cls, octets):
    typecode = chr(octets[0])
    memv = memoryview(octets[1:]).cast(typecode)
    return cls(memv) ⑦

```

- ① O atributo de instância "protegido" `self._components` vai manter um array com os componentes do Vector.
- ② Para permitir iteração, devolvemos um itereador sobre `self._components`; a função `iter()` é assunto do «Capítulo 17» [fpy.li/17] (vol.3), juntamente com o método `__iter__`.
- ③ Usa `reprlib.repr()` para obter uma representação de tamanho limitado de `self._components` (por exemplo, `array('d', [0.0, 1.0, 2.0, 3.0, 4.0, ...])`).
- ④ Remove o prefixo `array('d', e o)` final, antes de inserir a string em uma chamada ao construtor de Vector.
- ⑤ Cria um objeto bytes diretamente de `self._components`.
- ⑥ Desde o Python 3.8, `math.hypot` aceita pontos N-dimensionais. Já usei a seguinte expressão antes: `math.sqrt(sum(x * x for x in self))`.
- ⑦ A única mudança necessária no `frombytes` anterior é na última linha: passamos a `memoryview` diretamente para o construtor, sem desempacotá-la com `*`, como fazíamos antes.

O uso de `reprlib.repr` merece uma explicação. Essa função produz representações seguras de estruturas grandes ou recursivas, limitando o tamanho da string devolvida e indicando a abreviação com `'...'`. Eu queria que o `repr` de um Vector se parecesse com `Vector([3.0, 4.0, 5.0])` e não com `Vector(array('d', [3.0, 4.0, 5.0]))`, porque a existência de um array dentro de um Vector é um detalhe de implementação. Como essas chamadas ao construtor criam objetos Vector idênticos, preferi a sintaxe mais simples, usando um argumento `list`.

Ao escrever o `__repr__`, eu poderia construir uma string para exibir `components` com este código: `reprlib.repr(list(self._components))`. Mas isto teria um custo adicional, pois eu estaria copiando cada item de `self._components` para uma `list` só para usar a `list` no `repr`. Em vez disso, decidi aplicar `reprlib.repr` diretamente

no array `self._components`, e então remover os caracteres fora dos `[]`. É isso o que faz a segunda linha do `__repr__` no Exemplo 2.



Por seu papel na depuração, chamar `repr()` em um objeto não deveria nunca gerar uma exceção. Se alguma coisa der errado dentro de sua implementação de `__repr__`, você deve lidar com o problema e fazer o melhor possível para produzir uma saída aproveitável, que dê ao usuário uma chance de identificar o objeto receptor (`self`).

Observe que os métodos `__str__`, `__eq__`, e `__bool__` são idênticos a suas versões em `Vector2d`, e apenas um caractere mudou em `frombytes` (retirei um `*` na última linha). Esta é uma das vantagens de fazer o `Vector2d` original iterável.

Poderíamos criar `Vector` como uma subclasse de `Vector2d`, mas escolhi não fazer assim por duas razões. Em primeiro lugar, os construtores são incompatíveis, o que torna relação de super/subclasse desaconselhável, por violar o princípio de substituição de Liskov [fpy.li/6c]. Seria possível contornar isso como um tratamento engenhoso dos argumentos em `__init__`, mas a segunda razão é mais importante: eu queria que `Vector` fosse um exemplo independente de uma classe que implementa o protocolo de sequência. É o que faremos a seguir, após uma discussão sobre o termo *protocolo*.

12.4. Protocolos e a tipagem pato

Desde o primeiro capítulo vimos que não é necessário herdar de qualquer classe específica para criar um tipo sequência completamente funcional em Python; basta implementar os métodos que satisfazem o protocolo de sequência. Mas de que tipo de protocolo estamos falando?

No contexto da programação orientada a objetos, um protocolo é uma interface informal, definida apenas na documentação (e não no código). Por exemplo, o protocolo de sequência no Python implica apenas nos métodos `__len__` e `__getitem__`. Qualquer classe `Spam`, que implemente esses métodos com a assinatura e a semântica padrão, pode ser usada em qualquer lugar onde uma sequência é esperada. É irrelevante se `Spam` é uma subclasse dessa ou daquela outra classe; tudo o que importa é que ela fornece os métodos necessários. Vimos isso no «Exemplo 1 do Capítulo 1» [fpy.li/8x] (vol.1), reproduzido no Exemplo 3.

Exemplo 3. Código do «Exemplo 1 do Capítulo 1» [fpy.li/8x] (vol.1), repetido aqui

```
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 classe `FrenchDeck`, no Exemplo 3, pode tirar proveito de muitas facilidades de Python por implementar o protocolo de sequência, mesmo que isso não esteja declarado em qualquer ponto do código. Um programador Python experiente vai olhar para ela e entender que aquilo *é* uma sequência, mesmo sendo apenas uma subclasse de `object`. Dizemos que ela *é* uma sequência porque ela *se comporta* como uma sequência. Esta abordagem ficou conhecida como *duck typing* (literalmente "tipagem pato"), após o post de Alex Martelli citado no início deste capítulo.

Como protocolos são informais e não obrigatórios, muitas vezes é possível resolver nosso problema implementando apenas parte de um protocolo, se exatamente como a classe será utilizada. Por exemplo, apenas `__getitem__` é necessário para suportar iteração; não é preciso implementar `__len__`.



Com a *PEP 544—Protocols: Structural subtyping (static duck typing)* [fpy.li/pep544] (Protocolos: sub-tipagem estrutural (tipagem pato estática)), o Python 3.8 suporta *classes protocolo*: subclasses de `typing.Protocol`, que estudamos na «Seção 8.5.10» [fpy.li/8m] (vol.1). Este novo uso da palavra "protocolo" no Python tem um significado parecido, mas não idêntico. Quando

preciso diferenciá-los, escrevo "protocolo estático" para me referir a um protocolos formalizado por uma classe subclasse de `typing.Protocol`, e "protocolo dinâmico" para me referir ao sentido tradicional. Uma diferença fundamental é que uma implementação de um protocolo estático precisa oferecer todos os métodos definidos na classe protocolo. A Seção 13.3 apresentará muito mais detalhes.

Vamos agora implementar o protocolo de sequência em `Vector`, primeiro sem suporte adequado ao fatiamento, que acrescentaremos mais tarde.

12.5. Vector versão #2: sequência fatiável

Como vimos no exemplo da classe `FrenchDeck`, suportar o protocolo de sequência é muito fácil se você puder delegar para um atributo sequência em seu objeto, como nosso `array self._components`. Esses `__len__` e `__getitem__` de uma linha são um bom começo:

```
class Vector:
    # muitas linhas omitidas...

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

    def __getitem__(self, index):
        return self._components[index]
```

Com tais acréscimos, as seguintes operações funcionam:

```
>>> v1 = Vector([3, 4, 5])
>>> len(v1)
3
>>> v1[0], v1[-1]
(3.0, 5.0)
>>> v7 = Vector(range(7))
>>> v7[1:4]
array('d', [1.0, 2.0, 3.0])
```


Como se vê, até o fatiamento é suportado—mas não muito bem. Seria melhor se uma fatia de um `Vector` fosse também uma instância de `Vector`, e não um `array`. A classe `FrenchDeck` do primeiro capítulo tem o mesmo problema: quando fatiamos, obtemos uma `list`. No caso de `Vector`, perdemos muita funcionalidade quando o fatiamento devolve um simples `array`.

Considere os tipos sequência embutidos: cada um deles, ao ser fatiado, produz uma nova instância de seu próprio tipo, e não de um outro tipo.

Para fazer `Vector` produzir fatias como instâncias de `Vector`, não podemos simplesmente delegar o fatiamento para `array`. Precisamos analisar os argumentos recebidos em `__getitem__` e fazer a coisa certa.

Vejamos agora como Python transforma a sintaxe `my_seq[1:3]` em argumentos para `my_seq.__getitem__(...)`.

12.5.1. Como funciona o fatiamento

Uma demonstração vale mais que mil palavras, então veja o Exemplo 4.

Exemplo 4. Examinando o comportamento de `__getitem__` e fatias

```
>>> class MySeq:
...     def __getitem__(self, index):
...         return index ①
...
>>> s = MySeq()
>>> s[1] ②
1
>>> s[1:4] ③
slice(1, 4, None)
>>> s[1:4:2] ④
slice(1, 4, 2)
>>> s[1:4:2, 9] ⑤
(slice(1, 4, 2), 9)
>>> s[1:4:2, 7:9] ⑥
(slice(1, 4, 2), slice(7, 9, None))
```

- ① Para essa demonstração, o método `__getitem__` simplesmente devolve o que for passado a ele.

- ② Um único índice, nada de novo.
- ③ A notação 1:4 se torna `slice(1, 4, None)`.
- ④ `slice(1, 4, 2)` significa comece em 1, pare em 4, ande de 2 em 2.
- ⑤ Surpresa: a presença de vírgulas dentro do `[]` significa que `__getitem__` recebe uma tupla.
- ⑥ A tupla pode inclusive conter vários objetos `slice`.

Vamos agora olhar mais de perto a própria classe `slice`, no Exemplo 5.

Exemplo 5. Inspeccionando os atributos da classe `slice`

```
>>> slice ①
<class 'slice'>
>>> dir(slice) ②
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__le__', '__lt__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'indices', 'start', 'step', 'stop']
```

- ① `slice` é um tipo embutido (que já vimos antes na «Seção 2.7.2» [fpy.li/8p] (vol.1)).
- ② Inspeccionando uma `slice` descobrimos os atributos de dados `start`, `stop`, e `step`, e um método `indices`.

No Exemplo 5, a chamada `dir(slice)` revela um atributo `indices`, um método pouco conhecido mas muito interessante. Eis o que diz `help(slice.indices)`:

`S.indices(len) -> (start, stop, stride)`

Supondo uma sequência de tamanho `len`, calcula os índices `start` (*início*) e `stop` (*fim*), e a extensão do `stride` (*passo*) da fatia estendida descrita por `S`. Índices fora dos limites são recortados, exatamente como acontece em uma fatia normal.

Em outras palavras, o método `indices` expõe a lógica complexa implementada nas sequências embutidas, para tratar índices inexistentes ou negativos e fatias maiores que a sequência original. Esse método produz tuplas "normalizadas" com os inteiros não-negativos `start`, `stop`, e `stride` ajustados para uma sequência de um dado tamanho.

Aqui estão dois exemplos. Imagine que estamos lidando com uma sequência de `len == 5`, por exemplo `'ABCDE'`. Neste casos, passamos o valor 5 para `indices`:

```
>>> slice(None, 10, 2).indices(5) ①  
(0, 5, 2)  
>>> slice(-3, None, None).indices(5) ②  
(2, 5, 1)
```

① `'ABCDE'[:10:2]` é o mesmo que `'ABCDE'[0:5:2]`.

② `'ABCDE'[-3:]` é o mesmo que `'ABCDE'[2:5:1]`.

No código de nosso `Vector` não vamos precisar do método `slice.indices()`, pois quando recebermos uma fatia como argumento vamos delegar seu tratamento para o array interno `_components`. Mas quando você não puder contar com os serviços de uma sequência subjacente, esse método poupa o trabalho de implementar uma lógica sutil.

Agora que sabemos como tratar fatias, vamos ver a implementação aperfeiçoada de `Vector.__getitem__`.

12.5.2. Um `__getitem__` que devolve fatias

O Exemplo 6 lista os dois métodos necessários para fazer `Vector` se comportar como uma sequência: `__len__` e `__getitem__` (com o último implementado para tratar corretamente o fatiamento).

Exemplo 6. Parte de `vector_v2.py`: métodos `__len__` e `__getitem__` adicionados à classe `Vector`, de `vector_v1.py` (no Exemplo 2)

```
def __len__(self):  
    return len(self._components)
```

```
def __getitem__(self, key):
    if isinstance(key, slice): ①
        cls = type(self) ②
        return cls(self._components[key]) ③
    index = operator.index(key) ④
    return self._components[index] ⑤
```

- ① Se o argumento `key` é uma `slice`...
- ② ...obtem a classe da instância (isto é, `Vector`) e...
- ③ ...invoca a classe para criar outra instância de `Vector` a partir de uma fatia do array `_components`.
- ④ Se podemos obter um `index` de `key`...
- ⑤ ...devolve o item específico de `_components`.

A função `operator.index()` chama o método especial `__index__`. A função e o método especial foram definidos na *PEP 357—Allowing Any Object to be Used for Slicing* [fpy.li/pep357] (Permitir que Qualquer Objeto seja Usado para Fatiamento), proposta por Travis Oliphant, para permitir que qualquer um dos numerosos tipos de inteiros na NumPy fossem usados como argumentos de índices e fatias. A diferença essencial entre `operator.index()` e `int()` é que a primeira foi projetada para o propósito específico de obter índices. Por exemplo, `int(3.14)` devolve 3, mas `operator.index(3.14)` gera um `TypeError`, porque não faz sentido tentar usar um `float` como índice de um array.



O uso excessivo de `isinstance` pode ser um sinal de design orientado a objetos ruim, mas tratar fatias em `__getitem__` é um caso de uso justificável. Na primeira edição, também usei um teste `isinstance` com `key`, para checar se esse argumento era um inteiro. O uso de `operator.index` evita esse teste, e gera um `TypeError` com uma mensagem muito informativa, se não for possível obter o `index` a partir de `key`. Observe a última mensagem de erro no Exemplo 7, abaixo.

Após a adição do código do Exemplo 6 à classe `Vector` class, temos o comportamento apropriado para fatiamento, como demonstra o Exemplo 7.

Exemplo 7. Testes do `Vector.__getitem__` aperfeiçoado, do Exemplo 6

```
>>> v7 = Vector(range(7))
>>> v7[-1] ①
6.0
>>> v7[1:4] ②
Vector([1.0, 2.0, 3.0])
>>> v7[-1:] ③
Vector([6.0])
>>> v7[1,2] ④
Traceback (most recent call last):
...
TypeError: 'tuple' object cannot be interpreted as an integer
```

- ① Um índice inteiro recupera apenas o valor de um componente, um float.
- ② Uma fatia como índice cria um novo `Vector`.
- ③ Um fatia de `len == 1` também cria um `Vector`.
- ④ `Vector` não suporta indexação multidimensional, então tuplas de índices ou de fatias geram um erro.

12.6. Vector versão #3: atributos dinâmicos

Ao evoluir `Vector2d` para `Vector`, perdemos a habilidade de acessar os componentes do vetor por nome (por exemplo, `v.x`, `v.y`). Agora estamos trabalhando com vetores que podem ter um número grande de componentes. Ainda assim, pode ser conveniente acessar os primeiros componentes usando letras como atalhos, como `v.z` em vez de `v[2]`.

Esta é a sintaxe alternativa que queremos oferecer para a leitura dos quatro primeiros componentes de um vetor:

```
>>> v = Vector(range(10))
>>> v.x
0.0
>>> v.y, v.z, v.t
(1.0, 2.0, 3.0)
```

No Vector2d, oferecemos acesso somente para leitura a x e y através do decorador @property (veja o Exemplo 7 do Capítulo 11). Poderíamos incluir quatro propriedades no Vector, mas isso seria tedioso. O método especial `__getattr__` é uma opção melhor.

O método `__getattr__` é invocado só quando a busca por um atributo falha. Simplificando, dada a expressão `my_obj.x`, Python verifica se a instância de `my_obj` tem um atributo chamado x; em caso negativo, a busca passa para a classe (`my_obj.__class__`) e depois sobe pelo diagrama de herança.^[1] Se por fim o atributo x não for encontrado, o método `__getattr__`, definido na classe de `my_obj`, é chamado com `self` e o nome do atributo como uma string, por exemplo, 'x'.

O Exemplo 8 lista nosso método `__getattr__`. Ele basicamente verifica se o atributo desejado é uma das letras x y z t. Em caso positivo, devolve o componente correspondente do vetor.

Exemplo 8. Parte de vector_v3.py: método `__getattr__` acrescentado à classe Vector

```
__match_args__ = ('x', 'y', 'z', 't') ①

def __getattr__(self, name):
    cls = type(self) ②
    try:
        pos = cls.__match_args__.index(name) ③
    except ValueError: ④
        pos = -1
    if 0 <= pos < len(self._components): ⑤
        return self._components[pos]
    msg = f'{cls.__name__!r} object has no attribute {name!r}' ⑥
    raise AttributeError(msg)
```

- ① Define `__match_args__` para permitir casamento de padrões posicionais sobre os atributos dinâmicos suportados por `__getattr__`.^[2]
- ② Obtém a classe de Vector, para uso posterior.
- ③ Tenta obter a posição de name em `__match_args__`.
- ④ `.index(name)` gera um `ValueError` quando name não é encontrado; define pos como -1. (Eu preferiria usar algo como `str.find` aqui, mas tuple não implementa esse método.)

- ⑤ Se pos está dentro da faixa de componentes disponíveis, devolve aquele componente.
- ⑥ Se chegamos até aqui, gera um `AttributeError` com uma mensagem de erro padrão.

Não é difícil implementar `__getattr__`, mas neste caso não é o suficiente. Observe a interação bizarra no Exemplo 9.

Exemplo 9. Comportamento inapropriado: realizar uma atribuição a `v.x` não gera um erro, mas introduz uma inconsistência

```
>>> v = Vector(range(5))
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0])
>>> v.x ①
0.0
>>> v.x = 10 ②
>>> v.x ③
10
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0]) ④
```

- ① Acessa o elemento `v[0]` como `v.x`.
- ② Atribui um novo valor a `v.x`. Isso deveria gerar uma exceção.
- ③ Ler `v.x` obtém o novo valor, 10.
- ④ Entretanto, os componentes do vetor não mudam.

Você consegue explicar o que está acontecendo? Em especial, por que no passo ③, `v.x` devolve 10, se este valor não está presente no array de componentes do vetor? Se você não souber responder de imediato, estude a explicação de `__getattr__` que aparece logo antes do Exemplo 8. A razão é um sutil, mas é um fundamento importante para entender técnicas que veremos mais tarde no livro.

Após pensar um pouco sobre essa questão, veja a seguir a explicação para o que aconteceu.

A inconsistência no Exemplo 9 ocorre devido à forma como `__getattr__` funciona: Python só chama esse método como último recurso, quando o objeto não contém o atributo nomeado. Entretanto, após atribuirmos `v.x = 10`, o objeto `v` agora contém um atributo `x`, e então `__getattr__` não será mais invocado para obter `v.x`: o interpretador vai apenas devolver o valor `10`, que agora está vinculado a `v.x`. Por outro lado, nossa implementação de `__getattr__` obtém os valores dos "atributos virtuais" listados em `__match_args__` acessando apenas `self._components`, ignorando qualquer outro atributo da instância.

Para evitar essa inconsistência, precisamos mudar a lógica de definição de atributos em nossa classe `Vector`.

Como você se lembra, nos nossos últimos exemplos de `Vector2d` no Capítulo 11, tentar atribuir valores aos atributos de instância `.x` ou `.y` gerava um `AttributeError`. Em `Vector`, queremos produzir a mesma exceção em resposta a tentativas de atribuição a qualquer nome de atributo com um única letra minúscula, para evitar confusão. Para fazer isso, implementaremos `__setattr__`, como listado no Exemplo 10.

Exemplo 10. Parte de `vector_v3.py`: o método `__setattr__` na classe `Vector`

```
def __setattr__(self, name, value):
    cls = type(self)
    if len(name) == 1: ①
        if name in cls.__match_args__: ②
            error = 'readonly attribute {attr_name!r}'
        elif name.islower(): ③
            error = "can't set attributes 'a' to 'z' in {cls_name!r}"
        else:
            error = '' ④
        if error: ⑤
            msg = error.format(cls_name=cls.__name__, attr_name=name)
            raise AttributeError(msg)
    super().__setattr__(name, value) ⑥
```

- ① Tratamento especial para nomes de atributos com uma única letra.
- ② Se `name` está em `__match_args__`, configura uma mensagem de erro específica.
- ③ Se `name` é uma letra minúscula, configura a mensagem de erro sobre nomes de uma letra.

- ④ Caso contrário, configura uma mensagem de erro vazia.
- ⑤ Se existir uma mensagem de erro não-vazia, gera um `AttributeError`.
- ⑥ Caso default: chama `__setattr__` na superclasse para seguir o comportamento padrão.



A função `super()` fornece uma maneira de acessar dinamicamente métodos de superclasses, uma necessidade em uma linguagem dinâmica que suporta herança múltipla, como Python. Ela é usada para delegar alguma tarefa de um método em uma subclasse para um método adequado em uma superclasse, como visto no Exemplo 10. Falaremos mais sobre `super` na Seção 14.4.

Ao escolher a mensagem de erro para mostrar com `AttributeError`, primeiro eu verifiquei o comportamento do tipo embutido `complex`, pois ele é imutável e tem um par de atributos de dados `real` e `imag`. Tentar mudar qualquer um dos dois em uma instância de `complex` gera um `AttributeError` com a mensagem "can't set attribute" ("não é possível setar o atributo"). Por outro lado, a tentativa de modificar um atributo protegido por uma propriedade, como fizemos no Seção 11.7, produz a mensagem "read-only attribute" ("atributo apenas para leitura"). Eu me inspirei em ambas as frases para definir a string error em `__setitem__`, mas fui mais explícito sobre os atributos proibidos.

Note que não estamos proibindo a modificação de todos os atributos, apenas daqueles com nomes formados por uma letra minúscula, para evitar conflitos com os atributos suportados apenas para leitura: `x`, `y`, `z`, e `t`.



Sabendo que declarar `__slots__` no nível da classe impede a definição de novos atributos de instância, é tentador usar esse recurso em vez de implementar `__setattr__` como fizemos. Entretanto, por todas as ressalvas discutidas na Seção 11.11.2, usar `__slots__` apenas para prevenir a criação de atributos de instância não é recomendado. `__slots__` deve ser usado apenas para economizar memória, e apenas quando isso for um problema real.

Mesmo não suportando escrita nos componentes de `Vector`, aqui está uma lição importante deste exemplo: muitas vezes, quando você implementa `__getattr__`, é necessário também escrever o `__setattr__`, para evitar comportamentos inconsistentes em seus objetos.

Para permitir a modificação de componentes, poderíamos implementar `__setitem__`, para permitir `v[0] = 1.1`, e/ou `__setattr__`, para fazer `v.x = 1.1` funcionar. Mas `Vector` permanecerá imutável, pois queremos torná-lo *hashable*, na próxima seção.

12.7. Vector versão #4: o *hash* e um `==` mais rápido

Vamos novamente implementar um método `__hash__`. Juntamente com o `__eq__` existente, isso tornará as instâncias de `Vector` *hashable*.

O `__hash__` do `Vector2d` (no Exemplo 8 do Capítulo 11) computava o *hash* de uma *tuple* construída com os dois componentes, `self.x` e `self.y`. Agora podemos estar lidando com milhares de componentes, então criar uma *tuple* pode ser caro demais. Em vez disso, vou aplicar sucessivamente o operador \wedge (xor) aos *hashes* de todos os componentes, assim: `v[0] ^ v[1] ^ v[2]`. É para isso que serve a função `functools.reduce`. Anteriormente afirmei que `reduce` não é mais tão popular quanto antes,^[3] mas computar o *hash* de todos os componentes do vetor é um bom caso de uso para ela. A Figura 1 ilustra a ideia geral da função `reduce`.

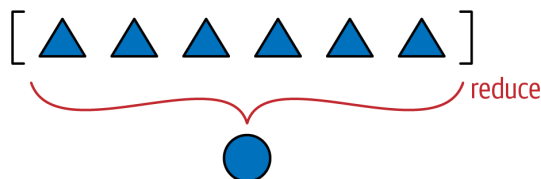


Figura 1. Funções de redução—`reduce`, `sum`, `any`, `all`—produzem um único resultado agregando valores de uma sequência ou de qualquer objeto iterável finito.

Até aqui vimos que `functools.reduce()` pode ser substituída por `sum()`. Vamos agora explicar exatamente como ela funciona. A ideia chave é reduzir uma série de valores a um valor único. O primeiro argumento de `reduce()` é uma função com dois argumentos, o segundo argumento é um iterável. Vamos dizer que temos uma função `fn`, que recebe dois argumentos, e uma lista `lst`. Quando

chamamos `reduce(fn, lst)`, `fn` será aplicada ao primeiro par de elementos de `lst`—`fn(lst[0], lst[1])`—produzindo um primeiro resultado, `r1`. Então `fn` é aplicada a `r1` e ao próximo elemento—`fn(r1, lst[2])`—produzindo um segundo resultado, `r2`. Agora `fn(r2, lst[3])` é chamada para produzir `r3` ... e assim por diante, até o último elemento, quando finalmente um único elemento, `rN`, é produzido e devolvido.

Veja como `reduce` pode ser usada para computar $5!$ (o fatorial de 5):

```
>>> 2 * 3 * 4 * 5 # resultado esperado: 5! == 120
120
>>> import functools
>>> functools.reduce(lambda a,b: a*b, range(1, 6))
120
```

Voltando a nosso problema de *hash*, o Exemplo 11 demonstra a ideia da computação de um xor agregado, fazendo isso de três formas diferente: com um laço `for` e com dois modos diferentes de usar `reduce`.

Exemplo 11. Três maneiras de calcular o xor acumulado de inteiros de 0 a 5

```
>>> n = 0
>>> for i in range(1, 6): ①
...     n ^= i
...
>>> n
1
>>> import functools
>>> functools.reduce(lambda a, b: a^b, range(6)) ②
1
>>> import operator
>>> functools.reduce(operator.xor, range(6)) ③
1
```

- ① xor agregado com um laço `for` e uma variável de acumulação.
- ② `functools.reduce` usando uma função anônima.
- ③ `functools.reduce` substituindo a `lambda` customizada por `operator.xor`.

Das alternativas apresentadas no Exemplo 11, a última é minha favorita, e o laço for vem a seguir. Qual sua preferida?

Como visto na «Seção 7.8.1» [fpy.li/8j] (vol.1), operator oferece a funcionalidade de todos os operadores infixos de Python em formato de função, diminuindo a necessidade do uso de lambda.

Para escrever `Vector.__hash__` no meu estilo preferido precisamos importar os módulos `functools` e `operator`. O Exemplo 12 apresenta as mudanças relevantes.

Exemplo 12. Parte de `vector_v4.py`: duas importações e o método `__hash__` adicionados à classe `Vector` de `vector_v3.py`

```
from array import array
import reprlib
import math
import functools ①
import operator ②

class Vector:
    typecode = 'd'

    # many lines omitted in book listing...

    def __eq__(self, other): ③
        return tuple(self) == tuple(other)

    def __hash__(self):
        hashes = (hash(x) for x in self._components) ④
        return functools.reduce(operator.xor, hashes, 0) ⑤

    # more lines omitted...
```

- ① Importa `functools` para usar `reduce`.
- ② Importa `operator` para usar `xor`.
- ③ Não há mudanças em `__eq__`; listei-o aqui porque é uma boa prática manter `__eq__` e `__hash__` próximos no código-fonte, pois eles precisam trabalhar juntos.

- ④ Cria uma expressão geradora para computar sob demanda o *hash* de cada componente.
- ⑤ Alimenta *reduce* com hashes e a função *xor*, para computar o código *hash* agregado; o terceiro argumento, *0*, é o inicializador (veja o aviso a seguir).



Ao usar *reduce*, é uma boa prática fornecer o terceiro argumento, *reduce(function, iterable, initializer)*, para prevenir a seguinte exceção: `TypeError: reduce() of empty sequence with no initial value` ("reduce() de uma sequência vazia sem valor inicial", uma mensagem bem escrita: explica o problema e diz como resolvê-lo). O *initializer* é o valor devolvido se a sequência for vazia e é usado como primeiro argumento no laço de redução, e portanto deve ser o elemento neutro da operação. Assim, o *initializer* para *+*, *|*, *^* (*xor*) deve ser *0*, mas para *** e *&* deve ser *1*.

Da forma como está implementado, o método `__hash__` no Exemplo 12 é um exemplo perfeito de uma do padrão *map-reduce* (mapear e reduzir). Veja a (Figura 2).

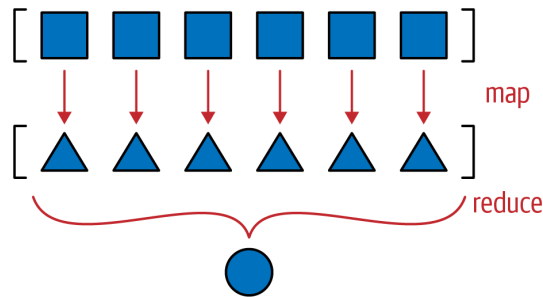


Figura 2. *Map-reduce*: *map* aplica uma função a cada item, gerando uma nova série, *reduce* computa o agregado.

A etapa de mapeamento produz um *hash* para cada componente, e a etapa de redução agrega todos os *hashes* com o operador *xor*. Se usarmos a função *map* em vez de uma *genexp*, a etapa de mapeamento fica ainda mais visível:

```
def __hash__(self):
    hashes = map(hash, self._components)
    return functools.reduce(operator.xor, hashes)
```



A solução com `map` era menos eficiente no Python 2, onde a função `map` criava uma nova `list` com os resultados. Mas no Python 3, `map` é preguiçosa (*lazy*): ela cria um gerador que produz os resultados sob demanda, e assim economiza memória—exatamente como a expressão geradora que usamos no método `__hash__` do Exemplo 8.

E enquanto estamos falando de funções de redução, podemos substituir nossa implementação apressada de `__eq__` com uma outra, menos custosa em termos de processamento e uso de memória, pelo menos para vetores grandes. Como visto no Exemplo 2 do Capítulo 11, temos esta implementação bastante concisa de `__eq__`:

```
def __eq__(self, other):  
    return tuple(self) == tuple(other)
```

Isso funciona com `Vector2d` e com `Vector`—e até considera `Vector([1, 2])` igual a `(1, 2)`, o que pode ser um problema, mas por ora vamos ignorar esta questão.^[4] Mas para instâncias de `Vector`, que podem ter milhares de componentes, esse método é muito ineficiente. Ele cria duas tuplas copiando todo o conteúdo dos operandos, apenas para usar o `__eq__` do tipo `tuple`. Para `Vector2d` (com apenas dois componentes), é um bom atalho. Mas não para grandes vetores multidimensionais. Uma forma melhor de comparar um `Vector` com outro `Vector` ou iterável seria o código do Exemplo 13.

Exemplo 13. A implementação de `Vector.__eq__` usando `zip` em um laço `for`, para uma comparação mais eficiente

```
def __eq__(self, other):  
    if len(self) != len(other): ①  
        return False  
    for a, b in zip(self, other): ②  
        if a != b: ③  
            return False  
    return True ④
```

- ① Objetos de tamanho diferentes não são iguais. Teste necessário porque `zip` retorna quando termina o iterável menor.

- ② zip produz um gerador de tuplas criadas a partir dos itens em cada argumento iterável.
- ③ Sai assim que dois componentes sejam diferentes, devolvendo False.
- ④ Caso contrário, os objetos são iguais.



O nome da função zip vem de zíper, pois o fecho de roupas funciona engatando pares de dentes a partir de duas abas paralelas, uma boa analogia visual para o que faz zip(esquerda, direita). Nenhuma relação com arquivos comprimidos.

Por padrão, zip encerra silenciosamente a geração de tuplas assim que um de seus argumentos é consumido até o fim, ainda que sobre itens em outros argumentos. Escrevi na primeira edição deste livro que este comportamento violava o princípio *fail fast* (falhar logo) do Python, e que zip deveria gerar um `ValueError` se os iteráveis não forem todos do mesmo tamanho, como acontece quando se desempacota um iterável para uma tupla de variáveis de tamanho diferente.

No Python 3.10, zip passou a aceitar o argumento nomeado opcional `strict=True`, que faz o que eu imaginava. Mas atenção: para preservar a compatibilidade, o default é `strict=False`, portanto o comportamento padrão ainda é parar sem avisar assim que um dos argumentos é consumido. Veja a caixa O fantástico zip logo adiante para saber mais sobre zip.

O Exemplo 13 é eficiente, mas a função `all` pode produzir o mesmo resultado do laço `for` em uma linha: se todas as comparações entre componentes correspondentes forem `True`, o resultado é `True`. Assim que uma comparação é `False`, `all` devolve `False`. Confira o Exemplo 14: novamente, comparamos os `len` para não invocar `zip` se os vetores têm tamanhos diferentes.

Exemplo 14. `Vector.__eq__` com zip e `all`: mesma lógica do Exemplo 13

```
def __eq__(self, other):  
    return (len(self) == len(other) and  
            all(a == b for a, b in zip(self, other)))
```

O fantástico zip

Ter um laço for que itera sobre itens sem perder tempo com variáveis de índice é muito bom e evita muitos bugs, mas exige algumas funções utilitárias especiais. Uma delas é a função embutida zip, que facilita a iteração em paralelo sobre dois ou mais iteráveis, devolvendo tuplas que você pode desempacotar em variáveis, uma para cada item nas entradas paralelas. Veja o Exemplo 15.

Exemplo 15. A função embutida zip trabalhando

```
>>> zip(range(3), 'ABC') ①
<zip object at 0x10063ae48>
>>> list(zip(range(3), 'ABC')) ②
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(zip(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3])) ③
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2)]
>>> from itertools import zip_longest ④
>>> list(zip_longest(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3],
fillvalue=-1))
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2), (-1, -1, 3.3)]
```

- ① zip devolve um gerador que produz tuplas sob demanda.
- ② Cria uma list apenas para exibição; normalmente iteramos sobre o gerador.
- ③ zip para sem aviso quando um dos iteráveis é esgotado.
- ④ A função `itertools.zip_longest` se comporta de forma diferente: ela usa um `fillvalue` opcional (por default `None`) para preencher os valores ausentes, e assim consegue gerar tuplas até que o último iterável seja esgotado.

A função zip pode também ser usada para transpor uma matriz, representada como iteráveis aninhados. Por exemplo:

```
>>> a = [(1, 2, 3),
...      (4, 5, 6)]
>>> list(zip(*a))
```



```
[(1, 4), (2, 5), (3, 6)]
>>> b = [(1, 2),
...      (3, 4),
...      (5, 6)]
>>> list(zip(*b))
[(1, 3, 5), (2, 4, 6)]
```

Se você quiser entender `zip`, passe algum tempo considerando como esses exemplos funcionam.

A função embutida `enumerate` é outra função geradora usada com frequência em laços `for`, para evitar manipulação direta de variáveis índice. Quem não estiver familiarizado com `enumerate` deve estudar a seção dedicada a ela na documentação das Funções embutidas [fpy.li/6d]. Voltaremos a falar sobre `zip` e `enumerate`, bem como várias outras funções geradores na biblioteca padrão, na «Seção 17.9» [fpy.li/8q] (vol.3).

Vamos encerrar esse capítulo trazendo de volta o método `__format__` do `Vector2d` para o `Vector`.

12.8. Vector versão #5: formatação

O método `__format__` de `Vector` será parecido com o mesmo método em `Vector2d`, mas em vez de fornecer uma exibição customizada em coordenadas polares, `Vector` usará coordenadas esféricas—também conhecidas como coordenadas "hiperesféricas", pois agora suportamos n dimensões, e esferas com mais de 3 dimensões são "hiperesferas".^[5] Por este motivo, mudaremos o sufixo do formato customizado de 'p' para 'h'.



Como vimos na Seção 11.6, ao estender a «Minilinguagem de especificação de formato» [fpy.li/63] é bom evitar os códigos de formato usados pelos tipos embutidos. Nossa minilinguagem estendida também usa os códigos de formato dos números de ponto flutuante ('eEfFgGn%') com seus significados originais. Inteiros usam 'bdxXn' e strings usam 's'. Escolhi 'p' para as coordenadas polares de `Vector2d`. O código 'h' para coordenadas hiperesféricas é uma boa opção.

Por exemplo, dado um objeto `Vector` em um espaço 4D (`len(v) == 4`), o código `'h'` irá produzir uma linha como `<3.2, 15.0, 45.0, 30.0>`, onde 3.2 é a magnitude (`abs(v)`), e os demais números são os componentes angulares que uma matemática chamaria de Φ_1, Φ_2, Φ_3 .

Aqui estão algumas amostras do formato de coordenadas esféricas em 4D, retiradas dos doctests de `vector_v5.py` (veja o Exemplo 16):

```
>>> format(Vector([-1, -1, -1, -1]), 'h')
'<2.0, 2.0943951023931957, 2.186276035465284, 3.9269908169872414>'
>>> format(Vector([2, 2, 2, 2]), '.3eh')
'<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
>>> format(Vector([0, 1, 0, 0]), '0.5fh')
'<1.00000, 1.57080, 0.00000, 0.00000>'
```

Antes de podermos implementar as pequenas mudanças necessárias em `__format__`, precisamos escrever um par de métodos de apoio: `angle(n)`, para computar uma das coordenadas angulares (por exemplo, Φ_1), e `angles()`, para devolver um iterável com todas as coordenadas angulares. Não vou descrever a matemática aqui; se você tiver curiosidade, a página *n*-sphere [fpy.li/nsphere] da Wikipedia apresenta as fórmulas que usei para calcular coordenadas esféricas a partir das coordenadas cartesianas no array de componentes de `Vector`.

O Exemplo 16 é a listagem completa de `vector_v5.py`, consolidando tudo que implementamos desde a Seção 12.3, e acrescentando a formatação customizada

Exemplo 16. `vector_v5.py`: a classe `Vector` com métodos para suportar `__format__`

```
"""
A multidimensional `Vector` class, take 5

A `Vector` is built from an iterable of numbers::

>>> Vector([3.1, 4.2])
Vector([3.1, 4.2])
>>> Vector((3, 4, 5))
Vector([3.0, 4.0, 5.0])
>>> Vector(range(10))
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
```

Tests with two dimensions (same results as ``vector2d_v1.py``)::

```
>>> v1 = Vector([3, 4])
>>> x, y = v1
>>> x, y
(3.0, 4.0)
>>> v1
Vector([3.0, 4.0])
>>> v1_clone = eval(repr(v1))
>>> v1 == v1_clone
True
>>> print(v1)
(3.0, 4.0)
>>> octets = bytes(v1)
>>> octets
b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00@\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
>>> abs(v1)
5.0
>>> bool(v1), bool(Vector([0, 0]))
(True, False)
```

Test of ``.frombytes()`` class method:

```
>>> v1_clone = Vector.frombytes(bytes(v1))
>>> v1_clone
Vector([3.0, 4.0])
>>> v1 == v1_clone
True
```

Tests with three dimensions::

```
>>> v1 = Vector([3, 4, 5])
>>> x, y, z = v1
>>> x, y, z
(3.0, 4.0, 5.0)
>>> v1
Vector([3.0, 4.0, 5.0])
>>> v1_clone = eval(repr(v1))
>>> v1 == v1_clone
```

```

True
>>> print(v1)
(3.0, 4.0, 5.0)
>>> abs(v1) # doctest:+ELLIPSIS
7.071067811...
>>> bool(v1), bool(Vector([0, 0, 0]))
(True, False)

```

Tests with many dimensions::

```

>>> v7 = Vector(range(7))
>>> v7
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
>>> abs(v7) # doctest:+ELLIPSIS
9.53939201...

```

Test of ``.__bytes__`` and ``.frombytes()`` methods::

```

>>> v1 = Vector([3, 4, 5])
>>> v1_clone = Vector.frombytes(bytes(v1))
>>> v1_clone
Vector([3.0, 4.0, 5.0])
>>> v1 == v1_clone
True

```

Tests of sequence behavior::

```

>>> v1 = Vector([3, 4, 5])
>>> len(v1)
3
>>> v1[0], v1[len(v1)-1], v1[-1]
(3.0, 5.0, 5.0)

```

Test of slicing::

```

>>> v7 = Vector(range(7))
>>> v7[-1]
6.0
>>> v7[1:4]

```

```

Vector([1.0, 2.0, 3.0])
>>> v7[-1:]
Vector([6.0])
>>> v7[1,2]
Traceback (most recent call last):
...
TypeError: 'tuple' object cannot be interpreted as an integer

```

Tests of dynamic attribute access::

```

>>> v7 = Vector(range(10))
>>> v7.x
0.0
>>> v7.y, v7.z, v7.t
(1.0, 2.0, 3.0)

```

Dynamic attribute lookup failures::

```

>>> v7.k
Traceback (most recent call last):
...
AttributeError: 'Vector' object has no attribute 'k'
>>> v3 = Vector(range(3))
>>> v3.t
Traceback (most recent call last):
...
AttributeError: 'Vector' object has no attribute 't'
>>> v3.spam
Traceback (most recent call last):
...
AttributeError: 'Vector' object has no attribute 'spam'

```

Tests of hashing::

```

>>> v1 = Vector([3, 4])
>>> v2 = Vector([3.1, 4.2])
>>> v3 = Vector([3, 4, 5])
>>> v6 = Vector(range(6))
>>> hash(v1), hash(v3), hash(v6)
(7, 2, 1)

```

Most hash codes of non-integers vary from a 32-bit to 64-bit CPython build::

```
>>> import sys
>>> hash(v2) == (384307168202284039 if sys.maxsize > 2**32 else
357915986)
True
```

Tests of `'format()'` with Cartesian coordinates in 2D::

```
>>> v1 = Vector([3, 4])
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

Tests of `'format()'` with Cartesian coordinates in 3D and 7D::

```
>>> v3 = Vector([3, 4, 5])
>>> format(v3)
'(3.0, 4.0, 5.0)'
>>> format(Vector(range(7)))
'(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0)'
```

Tests of `'format()'` with spherical coordinates in 2D, 3D and 4D::

```
>>> format(Vector([1, 1]), 'h') # doctest:+ELLIPSIS
'<1.414213..., 0.785398...>'
>>> format(Vector([1, 1]), '.3eh')
'<1.414e+00, 7.854e-01>'
>>> format(Vector([1, 1]), '0.5fh')
'<1.41421, 0.78540>'
>>> format(Vector([1, 1, 1]), 'h') # doctest:+ELLIPSIS
'<1.73205..., 0.95531..., 0.78539...>'
>>> format(Vector([2, 2, 2]), '.3eh')
'<3.464e+00, 9.553e-01, 7.854e-01>'
>>> format(Vector([0, 0, 0]), '0.5fh')
```

```
'<0.00000, 0.00000, 0.00000>'
>>> format(Vector([-1, -1, -1, -1]), 'h') # doctest:+ELLIPSIS
'<2.0, 2.09439..., 2.18627..., 3.92699...>'
>>> format(Vector([2, 2, 2, 2]), '.3eh')
'<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
>>> format(Vector([0, 1, 0, 0]), '0.5fh')
'<1.00000, 1.57080, 0.00000, 0.00000>'
"""
```

```
from array import array
import reprlib
import math
import functools
import operator
import itertools ①
```

```
class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __iter__(self):
        return iter(self._components)

    def __repr__(self):
        components = reprlib.repr(self._components)
        components = components[components.find('['):-1]
        return f'Vector({components})'

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (self.typecode.encode('ascii') +
                bytes(self._components))

    def __eq__(self, other):
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))

    def __hash__(self):
```

```

hashes = (hash(x) for x in self)
return functools.reduce(operator.xor, hashes, 0)

def __abs__(self):
    return math.hypot(*self)

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

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

def __getitem__(self, key):
    if isinstance(key, slice):
        cls = type(self)
        return cls(self._components[key])
    index = operator.index(key)
    return self._components[index]

__match_args__ = ('x', 'y', 'z', 't')

def __getattr__(self, name):
    cls = type(self)
    try:
        pos = cls.__match_args__.index(name)
    except ValueError:
        pos = -1
    if 0 <= pos < len(self._components):
        return self._components[pos]
    msg = f'{cls.__name__!r} object has no attribute {name!r}'
    raise AttributeError(msg)

def angle(self, n): ②
    r = math.hypot(*self[n:])
    a = math.atan2(r, self[n-1])
    if (n == len(self) - 1) and (self[-1] < 0):
        return math.pi * 2 - a
    else:
        return a

def angles(self): ③
    return (self.angle(n) for n in range(1, len(self)))

```



```

def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('h'): # hyperspherical coordinates
        fmt_spec = fmt_spec[:-1]
        coords = itertools.chain([abs(self)],
                                self.angles()) ④

        outer_fmt = '<{}>' ⑤
    else:
        coords = self
        outer_fmt = '({})' ⑥
    components = (format(c, fmt_spec) for c in coords) ⑦
    return outer_fmt.format(', '.join(components)) ⑧

@classmethod
def frombytes(cls, octets):
    typecode = chr(octets[0])
    memv = memoryview(octets[1:]).cast(typecode)
    return cls(memv)

```

- ① Importa `itertools` para usar a função `chain` em `__format__`.
- ② Computa uma das coordenadas angulares, conforme o artigo *n-sphere* [fpy.li/nsphere] na Wikipedia.
- ③ Cria uma expressão geradora para computar sob demanda todas as coordenadas angulares.
- ④ Produz uma *genexp* usando `itertools.chain`, para iterar de forma contínua sobre a magnitude e as coordenadas angulares.
- ⑤ Configura uma coordenada esférica para exibição delimitada por `< e >`.
- ⑥ Configura uma coordenda cartesiana para exibição entre parênteses.
- ⑦ Cria uma *genexp* para formatar cada componente.
- ⑧ Insere componentes separados por vírgulas nos delimitadores.



Usamos intensivamente expressões geradoras em `__format__`, `angle`, e `angles`, mas nosso foco aqui é fornecer um `__format__` para levar `Vector` ao mesmo nível de implementação de `Vector2d`. Quando tratarmos de geradores, no «Capítulo 17» [fpy.li/17] (vol.3), vamos usar parte do código de `Vector` nos exemplos, e lá os geradores serão explicados em detalhes.

Completamos a missão deste capítulo. Aperfeiçoaremos a classe `Vector` com operadores infixos no Capítulo 16. Nosso objetivo aqui foi explorar técnicas para programação de métodos especiais que são úteis em uma grande variedade de classes que implementam coleções de valores.

12.9. Resumo do capítulo

A classe `Vector`, o exemplo que desenvolvemos nesse capítulo, foi projetada para ser compatível com `Vector2d`, exceto pelo uso de uma assinatura de construtor diferente, aceitando um único argumento iterável, como fazem todos os tipos embutidos de sequências. O fato de `Vector` se comportar como uma sequência apenas por implementar `__getitem__` e `__len__` deu margem a uma discussão sobre protocolos, as interfaces informais usadas em linguagens com tipagem pato.

A seguir vimos como a sintaxe `my_seq[a:b:c]` funciona por baixo dos panos, criando um objeto `slice(a, b, c)` e entregando esse objeto a `__getitem__`. Armados com esse conhecimento, fizemos `Vector` responder corretamente ao fatiamento, devolvendo novas instâncias de `Vector`, como se espera de qualquer sequência pythônica.

O próximo passo foi fornecer acesso somente para leitura aos primeiros componentes de `Vector`, usando uma notação do tipo `my_vec.x`. Fizemos isso implementando `__getattr__`. Ao suportar esta forma de acessar atributos, podemos induzir o usuário tentar alterar aqueles componentes usando a forma `my_vec.x = 7`, revelando um possível bug. Consertamos o problema implementando também `__setattr__`, para barrar a atribuição de valores a atributos com nomes de uma letra. Após escrever um `__getattr__`, é comum surgir a necessidade de adicionar também `__setattr__`, para evitar comportamentos surpreendentes.

Implementar a função `__hash__` nos deu um contexto perfeito para usar `functools.reduce`, pois precisávamos aplicar o operador `xor (^)` sucessivamente aos *hashes* de todos os componentes de `Vector`, para produzir um código de *hash* agregado para o `Vector` como um todo. Após aplicar `reduce` em `__hash__`, usamos a função de redução embutida `all`, para criar um método `__eq__` eficiente.

O último aperfeiçoamento em `Vector` foi reimplementar o método `__format__` de `Vector2d`, para suportar coordenadas esféricas como alternativa às coordenadas cartesianas default. Usamos alguma matemática e vários geradores para programar `__format__` e suas funções auxiliares, mas esses são detalhes de implementação. Voltaremos aos geradores no «Capítulo 17» [fpy.li/17] (vol.3). O objetivo daquela última seção foi suportar um formato customizado, cumprindo assim a promessa de um `Vector` capaz de fazer tudo que um `Vector2d` faz, e algo mais.

Como fizemos no Capítulo 11, muitas vezes aqui examinamos como os objetos padrão de Python se comportam, para emulá-los e dar a `Vector` uma funcionalidade "pythônica".

No Capítulo 16 vamos implemenar vários operadores infixos em `Vector`. A matemática será mais simples que o método `angle()` de `Vector`, mas explorar como os operadores infixos funcionam no Python é uma grande lição sobre design orientado a objetos. Mas antes de chegar à sobrecarga de operadores em uma classe, vamos estudar a organização de várias classes com interfaces e herança, os assuntos do Capítulo 13 e do Capítulo 14.

12.10. Para saber mais

A maioria dos métodos especiais tratados no exemplo `Vector` também apareceram no exemplo `Vector2d`, no Capítulo 11, então as referências na Seção 11.14 são relevantes aqui também.

A poderosa função de ordem superior *reduce* também é conhecida como *fold* (dobrar), *accumulate* (acumular), *aggregate* (agregar), *compress* (comprimir), e *inject* (injetar). Para mais informações, veja o artigo *Fold (higher-order function)* [fpy.li/12-5] (Dobrar (função de ordem superior)), que apresenta aplicações daquela função, com ênfase em programação funcional com estruturas de dados recursivas. O artigo também inclui uma tabela mostrando funções similares a *fold* em dezenas de linguagens de programação.

Em *What's New in Python 2.5* [fpy.li/12-6] (Novidades no Python 2.5) há uma pequena explicação sobre o método `__index__`, projetado para suportar métodos `__getitem__`, como vimos na Seção 12.5.2. A *PEP 357—Allowing Any Object to be Used for Slicing* [fpy.li/pep357] (Permitir que Qualquer Objeto seja Usado para

Fatiamento) detalha a necessidade daquele método especial na perspectiva do mantenedor de uma extensão em C—Travis Oliphant, o principal criador da NumPy. As muitas contribuições de Oliphant tornaram Python uma das mais importantes linguagem para computação científica, favorecendo sua ampla adoção em aplicações de aprendizagem de máquina.

Ponto de vista

Protocolos como interfaces informais

Protocolos não são uma invenção de Python. Os criadores de Smalltalk, que também cunharam a expressão "orientado a objetos", usavam "protocolo" como um sinônimo para aquilo que hoje chamamos de interfaces. Alguns ambientes de programação Smalltalk permitiam que os programadores marcassem um grupo de métodos como um protocolo, mas tal marcação era só para documentação e navegação pelo código, e não era usada pela linguagem. Por isso acredito que "interface informal" é uma explicação curta razoável para "protocolo" quando falo para uma audiência mais familiar com interfaces formais, que são checadas por um compilador.

Protocolos bem estabelecidos ou consagrados evoluem naturalmente em qualquer linguagem que usa tipagem dinâmica (isto é, quando a checagem de tipos acontece durante a execução), porque não há informação estática de tipo em assinaturas de métodos e em variáveis. Ruby é outra importante linguagem orientada a objetos que tem tipagem dinâmica e usa protocolos.

Na documentação de Python, muitas vezes podemos perceber que um protocolo está sendo discutido pelo uso de palavras como *"a file like object"* ("um objeto semelhante a um arquivo"). Esta é uma forma abreviada de dizer "algo que se comporta como um arquivo, implementando as partes da interface de arquivo relevantes no presente contexto".

Você poderia achar que implementar apenas parte de um protocolo é um desleixo, mas isso tem a vantagem de manter as coisas simples. A Seção 3.3 [fpy.li/6e] do capítulo "Modelo de Dados" na documentação de Python sugere:

Ao implementar uma classe que emula qualquer tipo embutido, é importante que a emulação seja implementada apenas na medida em que faça sentido para o objeto que está sendo modelado. Por exemplo, algumas sequências podem funcionar bem com a recuperação de elementos individuais, mas extrair uma fatia pode não fazer sentido.

Quando não precisamos escrever métodos inúteis apenas para cumprir o contrato de uma interface excessivamente detalhista e satisfazer o compilador, fica mais fácil seguir o princípio KISS [fpy.li/6f].

Por outro lado, se quiser usar um checador de tipos para checar suas implementações de protocolos, então uma definição mais estrita de "protocolo" é necessária. É isso que `typing.Protocol` possibilita.

Terei mais a dizer sobre protocolos e interfaces no Capítulo 13, onde esses conceitos são o assunto principal.

De onde vieram os patos

Creio que a comunidade Ruby, mais que qualquer outra, ajudou a popularizar o termo *duck typing*, ao pregar para as massas de convertidos do Java. Mas a expressão já era usada nas discussões de Python muito antes de Ruby ou Python se tornarem "populares". De acordo com a Wikipedia, um dos primeiros exemplos de uso da analogia do pato, no contexto da programação orientada a objetos, foi uma mensagem para Python-list [fpy.li/12-11], escrita por Alex Martelli e datada de 26 de julho de 2000: "polymorphism (was Re: Type checking in python?)" (*polimorfismo (era Re: Verificação de tipo em python?)*) [fpy.li/12-9]. Foi dali que veio a citação no início desse capítulo. Se você tiver curiosidade sobre as origens literárias do termo "duck typing", e a aplicação desse conceito de orientação a objetos em muitas linguagens, veja a página Duck typing [fpy.li/6g] na Wikipedia.

Um `__format__` seguro, com usabilidade aperfeiçoada

Ao implementar `__format__`, não tomei qualquer precaução a respeito de instâncias de `Vector` com um número muito grande de componentes, como fizemos no `__repr__` usando `reprlib`. A justificativa é que `repr()` é usado para depuração e registro de logs, então precisa sempre gerar uma saída

minimamente aproveitável, enquanto `__format__` é usado para exibir resultados para usuários finais, que presumivelmente desejam ver o `Vector` inteiro. Se isso for inconveniente, então seria bom implementar uma nova extensão à minilinguagem de especificação de formato.

O que eu faria: por default, qualquer `Vector` formatado mostraria um número razoável mas limitado de componentes, digamos uns 30. Se existirem mais elementos que isso, o comportamento default seria similar ao de `repr lib`: cortar o excesso e exibir `' ... '`. Entretanto, se o especificador de formato terminar com um código especial `*`, significando "all" (*todos*), então a limitação de tamanho seria desabilitada. Assim, um usuário que desconhece o problema de exibição de vetores muito grandes não será penalizado. Mas se a limitação não for desejada, a presença das `' ... '` pode levar o usuário a consultar a documentação e descobrir o uso do `*` como opção de formatação.

A busca por uma soma pythônica

Na *python-list* [fpy.li/12-11], há uma thread de abril de 2003 intitulada *Pythonic Way to Sum n-th List Element?* [fpy.li/12-12] (A forma pythônica de somar o n-ésimo elemento em listas).

Não há uma resposta única para a "O que é pythônico?", da mesma forma que não há uma resposta única para "O que é belo?"

Mas talvez esta troca de ideias traga alguma luz.

O autor original, Guy Middleton, pediu melhorias para a solução abaixo, afirmando não gostar de usar `lambda`. Adaptei o código apresentado aqui: em 2003, `reduce` era uma função embutida, mas no Python 3 precisamos importá-la; também substituí os nomes `x` e `y` por `my_list` e `sub` (para sub-lista), e usei `ac` como variável acumuladora para o `reduce`.

No caso específico, Middleton quer somar o segundo item de cada lista de uma série de listas. Este foi o código que ele enviou para iniciar a discussão:

```
>>> from functools import reduce
>>> my_list = [[1, 2, 3], [30, 50, 70], [9, 8, 7]]
>>> reduce(lambda ac, n: ac+n, [sub[1] for sub in my_list])
60
```

Esse código usa várias peculiaridades de Python: `lambda`, `reduce` e uma compreensão de lista. Ele provavelmente ficaria em último lugar em um concurso de popularidade, pois ofende quem odeia `lambda` e também aqueles que desprezam as compreensões de lista.

Se você vai usar `lambda`, provavelmente não há razão para usar uma compreensão de lista—exceto para filtrar com `if`, que não é o caso aqui.

Aqui está uma solução minha que ofenderá todo mundo, exceto os fanáticos por `lambda`:

```
>>> reduce(lambda ac, sub: ac + sub[1], my_list, 0)
60
```

Não participei da discussão original, e não usaria este código porque também não gosto muito de `lambda`, principalmente em casos obscuros como este. Apenas quis mostrar aqui um exemplo sem uma compreensão de lista.

A primeira resposta veio de Fernando Perez, criador do IPython e do Jupyter Notebook, mostrando como a NumPy suporta arrays n -dimensionais e fatiamento n -dimensional:

```
>>> import numpy as np
>>> my_array = np.array(my_list)
>>> np.sum(my_array[:, 1])
60
```

A solução de Perez é boa, mas obviamente requer a NumPy.

Guy Middleton elogiou esta próxima solução, de Paul Rubin e Skip Montanaro:

```
>>> import operator
>>> reduce(operator.add, [sub[1] for sub in my_list], 0)
60
```

Então Evan Simpson perguntou, "O que há de errado em fazer assim?":

```
>>> ac = 0
>>> for sub in my_list:
...     ac += sub[1]
...
>>> ac
60
```

Muitos concordaram que este código era bastante pythônico. Alex Martelli chegou a escrever que Guido provavelmente resolveria o problema desta maneira. Gosto do código de Evan Simpson, mas também gosto do comentário de David Eppstein sobre ele:

Se você quer a soma de uma lista de itens, deveria escrever algo como "a soma de uma lista de itens", não como "faça um laço sobre esses itens, mantenha uma variável ac, execute uma série de somas". Por que temos linguagens de alto nível, senão para expressar nossas intenções em um nível mais alto e deixar a linguagem se preocupar com as operações de baixo nível necessárias para executá-las?

E daí Alex Martelli voltou para sugerir:

Fazemos somas com tanta frequência que eu não me importaria de forma alguma se Python a tornasse uma função embutida. Mas `reduce(operator.add, ...)` não é mesmo uma boa maneira de expressar isso, na minha opinião (e vejam que, como um antigo APLista^[6] e um apreciador da FP^[7], eu deveria gostar daquilo, mas não gosto).

Martelli então sugere uma função `sum()`, que ele mesmo programa e propõe para Python. Ela se torna uma função embutida no Python 2.3, lançado apenas três meses após aquela conversa na lista. E a sintaxe preferida de Alex se torna a regra:

```
>>> sum([sub[1] for sub in my_list])
60
```

No final do ano seguinte (novembro de 2004), Python 2.4 foi lançado e incluía expressões geradoras, fornecendo o que agora é, na minha opinião, a resposta mais pythônica para a pergunta original de Guy Middleton:

```
>>> sum(sub[1] for sub in my_list)
60
```

Isso não só é mais legível que `reduce`, também evita a armadilha da sequência vazia: `sum([])` é `0`, simples assim.

Na mesma conversa, Alex Martelli sugeriu que a função embutida `reduce` de Python 2 trazia mais problemas que soluções, porque encorajava idiomas de programação difíceis de explicar. Ele foi bastante convincente: a função foi rebaixada para o módulo `functools` no Python 3.

Ainda assim, `functools.reduce` tem seus usos. Ela resolveu o problema de nosso `Vector.__hash__` de uma forma que eu chamaria de pythônica.

[1] A pesquisa de atributos é mais complicada que isso; veremos todos os detalhes sinistros na Parte V: Metaprogramação (vol.3). Por ora, esta explicação simplificada nos serve.

[2] Apesar de `__match_args__` existir para suportar casamento de padrões desde o Python 3.10, é inofensivo definir este atributo em versões anteriores da linguagem. Na primeira edição chamei este atributo de `shortcut_names`. Com o novo nome, ele cumpre dois papéis: suportar padrões posicionais em instruções `case` e guardar os nomes dos atributos dinâmicos suportados por uma lógica especial em `__getattr__` e `__setattr__`.

[3] `sum`, `any`, e `all` cobrem a maioria dos casos de uso comuns de `reduce`. Veja a discussão na «Seção 7.3.1» [fpy.li/8b] (vol.1).

[4] Vamos considerar seriamente o caso de `Vector([1, 2]) == (1, 2)` na Seção 16.2.

[5] O website Wolfram Mathworld tem um artigo sobre hypersphere (*hiperesfera*) [fpy.li/12-4]; na Wikipedia, "hypersphere" redireciona para a página *n*-sphere [fpy.li/nsphere]

[6] NT: Aqui Martelli refere-se à linguagem APL [fpy.li/6h]

[7] NT: E aqui à linguagem FP [fpy.li/6j]

Capítulo 13. Interfaces, protocolos, e ABCs

Programe mirando uma interface, não uma implementação.^[1]

— Gamma, Helm, Johnson, Vlissides, First Principle of Object-Oriented Design

A programação orientada a objetos tem tudo a ver com interfaces. A melhor forma de entender um tipo em Python é conhecer os métodos que aquele tipo oferece—sua interface—como vimos na «Seção 8.4» [fpy.li/8s] (vol.1). Desde o Python 3.8, temos quatro maneiras de definir e usar interfaces. Elas estão ilustradas no *Mapa de Sistemas de Tipagem* (Figura 1).

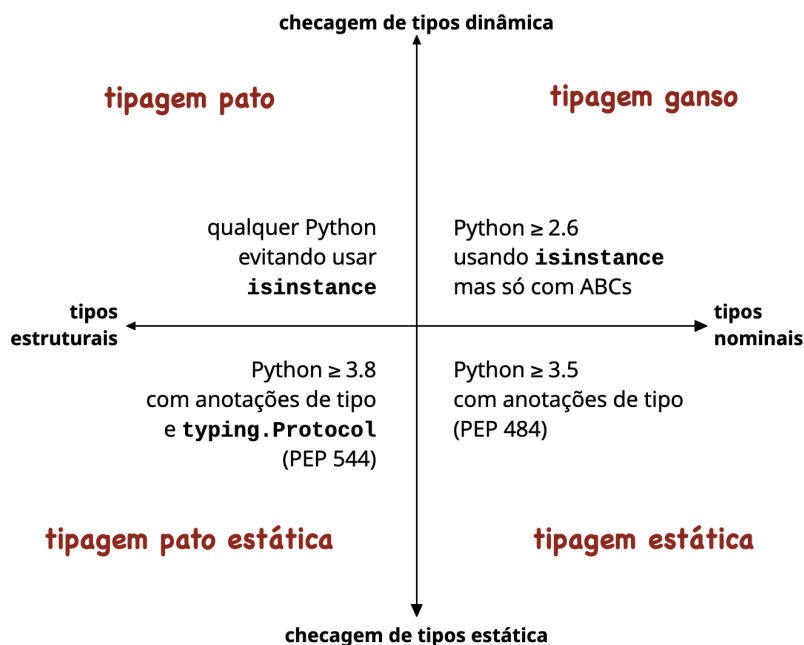


Figura 1. Na metade superior, checagens de tipo dinâmicas (em tempo de execução) usando só o interpretador Python; a metade inferior requer um checador estático externo como o *Mypy*, ou um IDE como o *PyCharm*. Os quadrantes da esquerda se referem à tipagem baseada na estrutura do objeto—isto é, os métodos oferecidos pelo objeto, independente de sua classe ou superclasses; os quadrantes da direita dependem de tipos explicitamente nomeados no código: a classe do objeto, ou suas superclasses.

Podemos as quatro abordagens assim:

Tipagem pato (*duck typing*)

O tratamento padrão para tipos em Python desde o início. Estamos estudando tipagem pato desde o primeiro capítulo do volume 1.

Tipagem ganso (*goose typing*)

A abordagem suportada pelas classes base abstratas (ABCs, *sigla em inglês para Abstract Base Classes*) desde Python 2.6, que depende de checar objetos contra ABCs durante a execução. A tipagem ganso é um dos principais temas deste capítulo.

Tipagem estática

A abordagem tradicional das linguagens de tipos estáticos como C e Java; suportada desde o Python 3.5 pelo módulo `typing`, e aplicada por checadores de tipos externos compatíveis com a PEP 484—Type Hints [fpy.li/pep484]. Este não é o foco deste capítulo. A maior parte do «Capítulo 8» [fpy.li/8] (vol.1) e do Capítulo 15 mais adiante são sobre tipagem estática.

Tipagem pato estática (*static duck typing*)

Uma abordagem popularizada pela linguagem Go; suportada por subclasses de `typing.Protocol`—lançada no Python 3.8 e também aplicada com o suporte de checadores de tipos externos. Tratamos desse tema pela primeira vez na «Seção 8.5.10» [fpy.li/8m] (vol.1), e continuamos neste capítulo.

13.1. O mapa de tipagem

As quatro abordagens retratadas na Figura 1 são complementares: elas têm diferentes prós e contras. Não faz sentido descartar qualquer uma delas.

Cada uma dessas quatro abordagens depende de interfaces para funcionar, mas a tipagem estática pode ser implementada de forma limitada usando apenas tipos concretos em vez de abstrações de interfaces como protocolos e classes base abstratas. Este capítulo é sobre tipagem pato, tipagem ganso, e tipagem pato estática—disciplinas de tipagem com foco em interfaces.

O capítulo está dividido em quatro seções principais, tratando de três dos quatro quadrantes no Mapa de Sistemas de Tipagem. (Figura 1):

- A Seção 13.3 compara duas formas de tipagem estrutural com protocolos—o lado esquerdo do Mapa.
- A Seção 13.4 se aprofunda na tipagem `pato`, que já é familiar para quem programa em Python. Vamos ver como fazê-la mais segura, preservando sua melhor qualidade: a flexibilidade.
- A Seção 13.5 explica o uso de ABCs para uma checagem de tipo mais estrita durante a execução do código. É a seção mais longa, não por ser a mais importante, mas porque há mais seções sobre tipagem `pato`, tipagem `pato` estática e tipagem estática em outras partes do livro.
- A Seção 13.6 cobre o uso, a implementação e o design de subclasses de `typing.Protocol`—para checagem de tipo estática e durante a execução.

13.2. Novidades neste capítulo

Editei profundamente este capítulo, e ele ficou cerca de 24% mais longo que o capítulo correspondente (o capítulo 11) na primeira edição de *Python Fluente*. Apesar de algumas seções e muitos parágrafos serem idênticos, há muito conteúdo novo. Estes são os principais acréscimos e modificações:

- A introdução do capítulo e o Mapa de Sistemas de Tipagem (Figura 1) são novos. Essa é a chave da maior parte do conteúdo novo—e de todos os outros capítulos relacionados à tipagem em Python ≥ 3.8 .
- A Seção 13.3 compara protocolos dinâmicos e estáticos.
- Atualizei a Seção 13.4.3 e dei um título que destaca sua importância: Programação defensiva e "falhe logo"
- A Seção 13.6 é toda nova. Ela se apoia na apresentação inicial na «Seção 8.5.10» [fpy.li/8m] (vol.1).
- Atualizei os diagramas de classe de `collections.abc` para incluir a ABC `Collection`, introduzida no Python 3.6.

Na primeira edição de *Python Fluente* escrevi uma seção encorajando o uso das ABCs do módulo `numbers` para tipagem ganso. Na Seção 13.6.8 explico por que, atualmente, é melhor usar protocolos numéricos estáticos do módulo `typing` como `SupportsFloat` se você planeja usar checadores de tipos estáticos, ou checagem durante a execução no estilo da tipagem ganso.

13.3. Dois tipos de protocolos

A palavra *protocolo* tem significados diferentes na ciência da computação, dependendo do contexto. Um protocolo de rede como o HTTP especifica comandos que um cliente pode enviar para um servidor, como GET, PUT e HEAD.

Vimos na Seção 12.4 que um protocolo especifica métodos que um objeto precisa oferecer para cumprir um papel.

O exemplo FrenchDeck no «Capítulo 1» [fpy.li/1] (vol.1) demonstra um protocolo, o protocolo de sequência: os métodos que permitem a um objeto Python se comportar como uma sequência.

Implementar um protocolo completo pode exigir muitos métodos, mas muitas vezes não há problema em implementar apenas parte dele. Considere a classe Vowels no Exemplo 1.

Exemplo 1. Implementação parcial do protocolo de sequência usando __getitem__

```
>>> class Vowels:
...     def __getitem__(self, i):
...         return 'AEIOU'[i]
...
>>> v = Vowels()
>>> v[0]
'A'
>>> v[-1]
'U'
>>> for c in v: print(c)
...
A
E
I
O
U
>>> 'E' in v
True
>>> 'Z' in v
False
```

Implementar `__getitem__` é o suficiente para obter itens pelo índice, e também para permitir iteração e o operador `in`. O método `__getitem__` é o método essencial do protocolo de sequência.

Veja a seção Protocolo de Sequência [fpy.li/6k] do Manual de referência da API Python/C:

```
int PySequence_Check(PyObject *o)
```

Retorna 1 se o objeto oferecer o protocolo de sequência, caso contrário retorna 0. Note que esta função retorna 1 para classes Python com um método `__getitem__`, a menos que sejam subclasses de `dict` [...]

Esperamos que uma sequência também suporte `len()`, através da implementação de `__len__`. `Vowels` não tem um método `__len__`, mas ainda assim se comporta como uma sequência em alguns contextos. E isso pode ser o suficiente para nossos propósitos. Por isso gosto de dizer que um protocolo é uma "interface informal." Também é assim que protocolos são entendidos em Smalltalk, o primeiro ambiente de programação orientado a objetos a usar esse termo.

Exceto em páginas sobre programação de redes, a maioria dos usos da palavra "protocolo" na documentação de Python se refere a essas interfaces informais.

Agora, com a adoção da *PEP 544—Protocols: Structural subtyping (static duck typing)* [fpy.li/pep544] no Python 3.8, a palavra "protocolo" ganhou um novo sentido em Python—um sentido próximo, mas diferente. Como vimos na «Seção 8.5.10» [fpy.li/8m] (vol.1), a PEP 544 nos permite criar subclasses de `typing.Protocol` para definir um ou mais métodos que uma classe deve implementar (ou herdar) para satisfazer um checador de tipos estático.

Quando precisar ser específico, vou adotar os seguintes termos:

Protocolo dinâmico

Os protocolos informais que Python sempre teve. Protocolos dinâmicos são implícitos, definidos por convenção e descritos na documentação. Os protocolos dinâmicos mais importantes de Python são implementados no próprio interpretador, e documentados no capítulo Modelo de Dados [fpy.li/2j] em *A Referência da Linguagem Python*.

Protocolo estático

Um protocolo como definido pela *PEP 544—Protocols...* [fpy.li/pep544], a partir de Python 3.8. Um protocolo estático é declarado explicitamente como uma subclasse de `typing.Protocol`.

Há duas diferenças fundamentais entre eles:

- Um objeto pode implementar apenas parte de um protocolo dinâmico e ainda assim ser útil; mas para satisfazer um protocolo estático, o objeto precisa oferecer todos os métodos declarados na classe do protocolo, mesmo que seu programa não precise de todos eles.
- Protocolos estáticos podem ser inspecionados por checadores de tipos estáticos, protocolos dinâmicos não.

Os dois tipos de protocolo têm uma característica essencial: uma classe nunca precisa declarar que suporta um protocolo pelo nome (por herança).

Antes dos protocolos estáticos, Python já oferecia outra forma de definir uma interface explícita no código: uma classe base abstrata (ABC). O restante deste capítulo trata de protocolos dinâmicos e estáticos, bem como das ABCs.

13.4. Programando patos

Vamos começar nossa discussão de protocolos dinâmicos com os dois mais importantes em Python: o protocolo de sequência e o iterável. O interpretador faz grandes esforços para lidar com objetos que fornecem mesmo uma implementação mínima desses protocolos, como explicado na próxima seção.

13.4.1. Python curte sequências

A filosofia do Modelo de Dados de Python é cooperar o máximo possível com os protocolos dinâmicos essenciais. Quando se trata de sequências, Python faz de tudo para lidar até com implementações mais rudimentares.

A Figura 2 mostra como a interface `Sequence` está formalizada como uma ABC. O interpretador Python e as sequências embutidas como `list`, `str`, etc., não dependem de forma alguma daquela ABC. Só estou usando a figura para descrever o que uma `Sequence` completa deve oferecer.

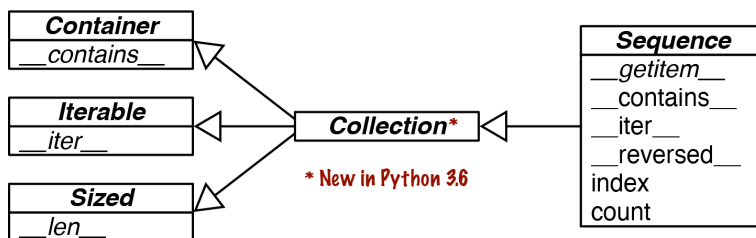


Figura 2. Diagrama de classe UML para a ABC Sequence e classes abstratas relacionadas de collections.abc. As setas de herança apontam de uma subclasse para suas superclasses. Nomes em *itálico* são métodos abstratos. Antes de Python 3.6, não existia uma ABC Collection—Sequence era uma subclasse direta de Container, Iterable e Sized.



A maior parte das ABCs no módulo collections.abc existe para formalizar interfaces que já eram implementadas por objetos nativos e implicitamente suportadas pelo interpretador, muito antes daquele módulo existir. As ABCs são úteis como pontos de partida para novas classes, e para permitir checagem de tipo explícita durante a execução (tipagem ganso), bem como para servirem de dicas de tipo para checadores de tipos estáticos.

Estudando a Figura 2, vemos que uma subclasse concreta de Sequence deve implementar `__getitem__` e `__len__` (de Sized). Todos os outros métodos Sequence são concretos, então as subclasses podem herdar suas implementações ou fornecer versões melhores.

Agora, lembre-se da classe Vowels no Exemplo 1. Ela não herda de abc.Sequence e implementa apenas `__getitem__`.

As instâncias de Vowels são iteráveis porque, na falta de um `__iter__`, Python tenta iterar invocando `__getitem__` com índices inteiros começando em 0. Da mesma forma que Python é esperto o suficiente para iterar sobre instâncias de Vowels, ele também consegue fazer o operador `in` funcionar mesmo quando o método `__contains__` não existe: ele faz uma busca sequencial para verificar se o item está presente.

Em resumo, dada a importância das sequências como estruturas de dados, Python consegue fazer a iteração e o operador `in` funcionarem invocando `__getitem__` quando `__iter__` e `__contains__` não estão presentes.

O FrenchDeck original do «Capítulo 1» [fpy.li/1] (vol.1) também não é subclasse de `abc.Sequence`, mas ele implementa os dois métodos do protocolo de sequência: `__getitem__` e `__len__`. Veja o Exemplo 2.

Exemplo 2. Um baralho como uma sequência de cartas, como «Exemplo 1 do Capítulo 1» [fpy.li/8x] (vol.1)

```
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]
```

Muitos dos exemplos no «Capítulo 1» [fpy.li/1] (vol.1) funcionam por causa do tratamento especial que Python dá a estruturas vagamente semelhantes a uma sequência. O protocolo iterável em Python representa uma forma extrema de tipagem pato: o interpretador tenta dois métodos diferentes para iterar sobre objetos.

Para deixar mais claro, os comportamentos que descrevi nessa seção estão implementados no próprio interpretador, na maioria dos casos em C. Eles não dependem dos métodos da ABC Sequence. Por exemplo, os métodos concretos `__iter__` e `__contains__` na classe `Sequence` emulam comportamentos internos do interpretador Python. Se tiver curiosidade, veja o código-fonte destes métodos em *Lib/_collections_abc.py* [fpy.li/13-3].

Agora vamos estudar um exemplo que demonstra por que checadores de tipos estáticos não têm como lidar com protocolos dinâmicos.

13.4.2. Monkey patching: implementando um protocolo em runtime

Monkey patching é o ato de remendar (*patch*) dinamicamente um programa durante a execução do código (*runtime*), para acrescentar funcionalidade ou corrigir bugs. Por exemplo, a biblioteca de rede *gevent* [www.gevent.org/api/gevent.monkey.html] faz "monkey patch" em partes da biblioteca padrão de Python, para permitir concorrência sem threads ou `async/await`.^[2] O monkey patch não lê nem altera o código-fonte do programa, apenas os objetos na memória que representam as partes do programa, como módulos, classes e funções.

Vamos fazer *monkey patch* na classe `FrenchDeck` do Exemplo 2 para superar uma grande limitação: ela não pode ser embaralhada. Anos atrás, quando escrevi pela primeira vez o exemplo `FrenchDeck`, implementei um método `shuffle`. Depois tive uma sacada pythônica: se um `FrenchDeck` funciona como uma sequência, não precisa ter um método `shuffle`, pois já existe a função `random.shuffle`, que "embaralha a sequência x internamente" conforme a documentação oficial [fpy.li/6m].

A função `random.shuffle` é usada assim:

```
>>> from random import shuffle
>>> l = list(range(10))
>>> shuffle(l)
>>> l
[5, 2, 9, 7, 8, 3, 1, 4, 0, 6]
```



Ao adotar protocolos estabelecidos, aumenta muito suas chances de aproveitar o código já existente na biblioteca padrão e em bibliotecas de terceiros, graças à tipagem pato.

Entretanto, se tentamos usar `shuffle` com uma instância de `FrenchDeck` ocorre uma exceção, como visto no Exemplo 3.

Exemplo 3. random.shuffle não funciona com FrenchDeck

```
>>> from random import shuffle
>>> from frenchdeck import FrenchDeck
>>> deck = FrenchDeck()
>>> shuffle(deck)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../random.py", line 265, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'FrenchDeck' object does not support item assignment
```

A mensagem de erro é clara: "o objeto 'FrenchDeck' não suporta a atribuição de itens". O problema é que `shuffle` opera internamente, trocando os itens de lugar dentro da coleção, mas `FrenchDeck` só implementa o protocolo de sequência imutável. Para ser uma sequência mutável, `FrenchDeck` precisa oferecer um método `__setitem__`.

Como Python é dinâmico, podemos consertar isso durante a execução, até mesmo no console interativo. O Exemplo 4 mostra como fazer isso.

Exemplo 4. "Monkey patching" o FrenchDeck para torná-lo mutável e compatível com `random.shuffle` (continuação do Exemplo 3)

```
>>> def set_card(deck, position, card): ①
...     deck._cards[position] = card
...
>>> FrenchDeck.__setitem__ = set_card ②
>>> shuffle(deck) ③
>>> deck[:5]
[Card(rank='3', suit='hearts'), Card(rank='4', suit='diamonds'), Card
(rank='4',
suit='clubs'), Card(rank='7', suit='hearts'), Card(rank='9', suit='
spades')]
```

- ① Cria uma função que recebe `deck`, `position`, e `card` como argumentos.
- ② Atribui aquela função a um atributo chamado `__setitem__` na classe `FrenchDeck`.
- ③ `deck` agora pode ser embaralhado, pois acrescentei o método necessário do protocolo de sequência mutável.

A assinatura do método especial `__setitem__` está definida na Referência da Linguagem Python em Emulando tipos contêineres [fpy.li/6n]. Aqui nomeei os argumentos `deck`, `position`, `card`—e não `self`, `key`, `value` como na referência da linguagem—para mostrar que todo método Python começa sua vida como uma

função comum, e nomear o primeiro argumento `self` é só uma convenção. Fugir da convenção é OK em uma sessão no console onde o código é descartável, mas em um arquivo de código-fonte de Python é muito melhor usar `self`, `key`, e `value`, seguindo a documentação.

O truque é que `set_card` pressupõe que o deck tem um atributo chamado `_cards`, e seu valor deve ser uma sequência mutável. A função `set_cards` é então anexada à classe `FrenchDeck` como o método especial `__setitem__`. Isso é um exemplo de *monkey patching*: modificar uma classe ou módulo durante a execução, sem tocar no código-fonte. O "monkey patching" é poderoso, mas o código que executa a modificação fica muito intimamente acoplado ao programa sendo modificado, muitas vezes trabalhando com atributos privados e não-documentados.

Além de ser um exemplo de monkey patching, o Exemplo 4 enfatiza a natureza dinâmica dos protocolos na tipagem pato: `random.shuffle` não se importa com a classe do argumento, ela só precisa que o objeto implemente métodos do protocolo de sequência mutável. Não importa sequer se o objeto "nasceu" com os métodos necessários ou se eles foram de alguma forma adquiridos depois.

Quando bem aplicada, tipagem pato não é loucamente insegura ou difícil de depurar e manter. A próxima seção mostra alguns padrões de programação úteis para detectar protocolos dinâmicos sem recorrer a checagens explícitas.

13.4.3. Programação defensiva e "falhe logo"

Programação defensiva é como direção defensiva: um conjunto de práticas para melhorar a segurança, mesmo na presença de programadores (ou motoristas) descuidados.

Muitos bugs não podem ser encontrados exceto durante a execução—mesmo nas principais linguagens de tipagem estática.^[3] Em uma linguagem de tipagem dinâmica, "falhe logo" (*fail fast*) é um ótimo conselho para codar programas mais seguros e mais fáceis de manter. Falhar logo significa assegurar que os erros em tempo de execução sejam detectados o mais cedo possível. Por exemplo, rejeitando argumentos inválidos no início do corpo de uma função.

Exemplo prático: quando você escreve código que aceita uma sequência de itens para processar internamente como uma `list`, não valide o argumento só por checagem de tipo. Em vez disso, receba o argumento e construa imediatamente

uma `list` a partir dele. Um exemplo desse padrão de programação é o método `__init__` no Exemplo 11, que veremos mais à frente nesse capítulo:

```
def __init__(self, iterable):  
    self._balls = list(iterable)
```

Desta forma você torna seu código mais flexível, pois o construtor de `list()` processa qualquer iterável que caiba na memória. Se o argumento não for iterável, a chamada vai falhar logo com uma exceção de `TypeError` bastante clara, no exato momento em que o objeto for inicializado. Se quiser ser mais explícito, pode colocar a chamada a `list()` em um `try/except`, para adequar a mensagem de erro—mas eu escreveria este código extra apenas em uma API externa, pois a falha já estaria bem visível para os mantenedores que conhecem a base de código. De toda forma, a chamada errônea vai aparecer perto do final do `traceback`, tornando-a fácil de corrigir. Se você não barrar o argumento inválido no construtor da classe, o programa vai quebrar mais tarde, quando algum outro método da classe precisar usar a variável `self._balls` e ela não for uma `list`. Então a causa do problema estará mais distante e será mais difícil de encontrar.

Naturalmente, seria ruim passar o argumento para `list()` se os dados não devem ser copiados, ou por seu tamanho ou porque quem chama a função, espera que os itens sejam modificados internamente, como no caso de `random.shuffle`. Neste caso, uma checagem durante a execução como `isinstance(x, abc.MutableSequence)` seria a melhor opção: a abordagem da tipagem ganso.

Se tiver receio de consumir um gerador infinito—algo que não é um problema muito comum—pode começar chamando `len()` com o argumento. Isso rejeitaria iteradores, mas lidaria de forma segura com tuplas, arrays e outras classes existentes ou futuras que implementem a interface `Sequence` completa. Chamar `len()` normalmente não custa muito, e um argumento inválido vai gerar um erro na hora.

Por outro lado, se qualquer iterável for aceitável, chame `iter(x)` assim que possível, para obter um iterador, como veremos na «Seção 17.3» [fpy.li/89] (vol.3). E novamente, se `x` não for iterável, isso falhará logo com uma exceção fácil de depurar.

Nos casos que acabei de descrever, uma dica de tipo poderia apontar alguns problemas mais cedo, mas não todos os problemas. Lembre-se de que o tipo `Any` é *consistente-com* qualquer outro tipo. Inferência de tipo pode fazer com que uma variável seja marcada com o tipo `Any`. Quando isso acontece, o checador de tipos se torna inútil. Além disso, dicas de tipo não são aplicadas durante a execução. Falhar logo é a última linha de defesa.

Código defensivo usando tipagem pato também pode incluir lógica para lidar com tipos diferentes sem usar testes com `isinstance()` e `hasattr()`.

Um exemplo é como poderíamos imitar como `collections.namedtuple` [fpy.li/13-8] lida com o argumento `field_names`: ele aceita uma única string com identificadores separados por espaços ou vírgulas, ou uma sequência de identificadores. O Exemplo 5 mostra como eu faria isso usando tipagem pato.

Exemplo 5. Tipagem pato para lidar com uma string ou um iterável de strings

```
try: ①
    field_names = field_names.replace(',', ' ').split() ②
except AttributeError: ③
    pass ④
field_names = tuple(field_names) ⑤
if not all(s.isidentifier() for s in field_names): ⑥
    raise ValueError('field_names must all be valid identifiers')
```

- ① Supõe que é uma string.
- ② Converte vírgulas em espaços e divide o resultado em uma lista de nomes.
- ③ Perdão, `field_names` não grana como uma `str`: não tem `.replace`, ou tem um `.replace` que devolve algo que não funciona com `.split`
- ④ Se um `AttributeError` aconteceu, então `field_names` não é uma `str`. Supomos que já é um iterável de nomes.
- ⑤ Para ter certeza de que é um iterável e para manter nossa própria cópia, criamos uma tupla com o que temos. Uma `tuple` é mais compacta que uma lista, e também impede que meu código troque os nomes por acidente.
- ⑥ Usamos `str.isidentifier` para garantir que todos os nomes são válidos.

O passo ② do Exemplo 5 é uma aplicação de EAFP ou Princípio de Hopper.^[4] Em vez de testar se `field_names` é uma string, invocamos métodos como se fosse uma string, e se não der certo, tratamos a exceção. Não pedimos licença: fazemos o que temos que fazer e pedimos perdão se for necessário.

O Exemplo 5 mostra uma situação em que a tipagem pato é mais expressiva que dicas de tipo estáticas. Não há como escrever uma dica de tipo que diga "o argumento `field_names` deve ser uma string de identificadores separados por espaços ou vírgulas." Esta é a parte relevante da assinatura de `namedtuple` no `typeshed` (veja o código-fonte completo em `stdlib/3/collections/init.pyi` [fpy.li/13-9]):

```
def namedtuple(
    typename: str,
    field_names: Union[str, Iterable[str]],
    *,
    # outros parâmetros omitidos
```

Como se vê, `field_names` está anotado como `Union[str, Iterable[str]]`, que ajuda em parte, mas não é suficiente para descrever a estrutura interna da string.

Após revisar protocolos dinâmicos, passamos para uma forma mais explícita de checagem de tipo durante a execução: tipagem ganso.

13.5. Tipagem ganso

Uma classe abstrata representa uma interface.^[5]

— Bjarne Stroustrup, criador do C++

Python não tem uma palavra-chave `interface`. Usamos classes base abstratas (ABCs) para definir interfaces úteis para checagem explícita de tipo durante a execução, e também para anotações compatíveis com checadores de tipos estáticos.

O verbete classe base abstrata [fpy.li/6p] no Glossário da Documentação de Python tem uma boa explicação do valor dessas estruturas para linguagens que usam tipagem pato:

Classes base abstratas complementam a tipagem pato, fornecendo uma maneira de definir interfaces quando outras técnicas, como `hasattr()`, seriam desajeitadas ou sutilmente erradas (por exemplo, com métodos mágicos). ABCs introduzem subclasses virtuais, classes que não herdam de uma classe mas ainda são reconhecidas por `isinstance()` e `issubclass()`; veja a documentação do módulo `abc`.

A tipagem ganso é uma abordagem à checagem de tipo durante a execução que se apoia nas ABCs. Vou deixar que Alex Martelli explique, no texto *Pássaros aquáticos e as ABCs*.



Sou muito grato a meus amigos Alex Martelli e Anna Ravenscroft. Mostrei a eles a primeira lista de tópicos do *Python Fluente* na OSCON 2013, e eles me encorajaram a submeter à O'Reilly para publicação. Depois os dois contribuíram com revisões técnicas minuciosas. Alex já era a pessoa mais citada nesse livro quando se ofereceu para escrever este ensaio.

Pássaros aquáticos e as ABCs

por Alex Martelli

Fui creditado na Wikipedia [[fpy.li/13-11](https://en.wikipedia.org/wiki/Python_(programming_language)#Python_2.0)] por ajudar a popularizar o meme útil e frase de efeito "*duck typing*" (isto é, ignorar o tipo declarado de um objeto, e em vez disso se dedicar a assegurar que o objeto implementa os nomes, assinaturas e semântica dos métodos necessários para o uso pretendido).

Em Python, isso essencialmente significa evitar o uso de `isinstance` para checar o tipo do objeto (sem nem mencionar a abordagem ainda pior de checar, por exemplo, se `type(foo) is bar`—que é corretamente considerado um anátema, pois inibe até as formas mais simples de herança!).

No geral, a abordagem da tipagem pato continua muito útil em inúmeros contextos—mas em muitos outros, uma nova abordagem muitas vezes preferível evoluiu ao longo do tempo. E aqui começa nossa história...

Em gerações recentes, a taxonomia de gênero e espécies (incluindo, mas não limitada à família de pássaros aquáticos conhecida como Anatidae) foi guiada principalmente pela *fenética*—uma abordagem centrada nas similaridades de morfologia e comportamento... principalmente traços *observáveis*. A analogia com "*duck typing*" era evidente.

Entretanto, a evolução paralela muitas vezes pode produzir características similares, tanto morfológicas quanto comportamentais, em espécies sem qualquer relação de parentesco, que apenas calharam de evoluir em nichos ecológicos similares, porém separados. "Similaridades acidentais" parecidas acontecem também em programação—por exemplo, considere um exemplo clássico de programação orientada a objetos:^[6]

```
class Artist:
    def draw(self): ...

class Gunslinger:
    def draw(self): ...

class Lottery:
    def draw(self): ...
```

Obviamente, a mera existência de um método chamado `draw`, sem parâmetros, não é suficiente para garantir que dois objetos `x` e `y`, que aceitem as invocações `x.draw()` e `y.draw()`, são de qualquer forma intercambiáveis ou abstratamente equivalentes—nada pode ser inferido sobre a similaridade da semântica resultante de tais chamadas. Na verdade, é necessário um programador consciente para, de alguma forma, *assegurar* afirmativamente que tal equivalência é verdadeira em algum nível.

Em biologia (e outras disciplinas), este problema levou à emergência (e, em muitas facetas, à dominância) de uma abordagem alternativa à *fenética*, conhecida como *cladística* — que baseia as escolhas taxonômicas em características herdadas de ancestrais comuns em vez daquelas que evoluíram de forma independente (o sequenciamento de DNA cada vez mais barato e rápido vem tornando a cladística bastante prática em mais casos).

Por exemplo, os *Chloephaga*, gênero de gansos sul-americanos (antes classificados como próximos a outros gansos) e as *tadornas* (gênero de patos sul-americanos) estão agora agrupados juntos na subfamília *Tadornidae* (sugerindo que eles são mais próximos entre si do que de qualquer outro *Anatidae*, pois compartilham um ancestral comum mais próximo). Além disso, a análise de DNA mostrou que o *Asarcornis* (pato da floresta ou pato de asas brancas) não é tão próximo do *Cairina moschata* (pato-do-mato), esse último uma *tadorna*, como as similaridades corporais e comportamentais sugeriram por tanto tempo—então o pato da floresta foi reclassificado em um gênero próprio, inteiramente fora da subfamília!

Isso importa? Depende do contexto! Para o propósito de decidir como cozinhar uma ave após caçá-la, por exemplo, características observáveis específicas (mas nem todas—a plumagem, por exemplo, é de mínima importância nesse contexto), especialmente textura e sabor (a boa e velha fenética), podem ser mais relevantes que a cladística. Mas para outros problemas, tal como a suscetibilidade a diferentes patógenos (se quiser criar aves aquáticas em cativeiro, ou preservá-las na natureza), a proximidade do DNA pode ser mais importante.

Então, a partir dessa analogia aproximada com as revoluções taxonômicas no mundo das aves aquáticas, estou recomendando suplementar (não substituir inteiramente—em determinados contextos ela ainda servirá) o bom e velho *duck typing* por... *goose typing* (tipagem ganso)!

Goose typing significa o seguinte: `isinstance(obj, cls)` agora é plenamente aceitável... desde que `cls` seja uma classe base abstrata—em outras palavras, a metaclasses de `cls` é `abc.ABCMeta`.

Você vai encontrar muitas classes abstratas prontas em `collections.abc` (e outras no módulo `numbers` da biblioteca padrão do Python)^[7]

Dentre as muitas vantagens conceituais das ABCs sobre classes concretas (e.g., a prescrição de Scott Meyer “toda classe não-final (não-folha) deveria ser abstrata”; veja o Item 33 [fpy.li/13-12] de seu livro, *More Effective C++*, Addison-Wesley), as ABCs de Python acrescentam uma grande vantagem prática: o método de classe `register`, que permite ao código da aplicação “declarar” que determinada classe é uma subclasse “virtual” de uma ABC (para este propósito, a classe registrada precisa cumprir os requisitos de

nome de métodos e assinatura da ABC e, mais importante, o contrato semântico subjacente—mas não precisa ter sido desenvolvida com qualquer conhecimento da ABC, e especificamente não precisa herdar dela!). Isso é um longo caminho andado na direção de quebrar a rigidez e o acoplamento forte que torna herança algo para ser usado com mais cautela que aquela tipicamente praticada pela maioria dos programadores orientados a objetos.

Em algumas ocasiões você sequer precisa registrar uma classe para que uma ABC a reconheça como uma subclasse!

Esse é o caso das ABCs cuja essência se resume em alguns métodos especiais. Por exemplo:^[8]

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
```

Como se vê, `abc.Sized` reconhece `Struggle` como uma subclasse, sem necessidade de registro, já que implementar o método especial chamado `__len__` é o suficiente (o método deve ser implementado com a sintaxe e semântica corretas—deve poder ser chamado sem argumentos e retornar um inteiro não-negativo indicando o "comprimento" do objeto; mas qualquer código que implemente um método com nome especial, como `__len__`, com uma sintaxe e uma semântica arbitrárias e incompatíveis tem problemas bem maiores que estes).

Então, aqui está minha mensagem de despedida: sempre que você estiver implementando uma classe que incorpore quaisquer dos conceitos representados nas ABCs de `number`, `collections.abc` ou em outro framework que estiver usando, assegure-se (caso necessário) de ser uma subclasse ou de registrar sua classe com a ABC correspondente. No início de seu programa que utiliza uma biblioteca ou framework que define classes que omitiram esse passo, registre você mesmo as classes. Daí, quando precisar checar se um argumento é, por exemplo, "uma sequência", verifique se:

```
isinstance(the_arg, collections.abc.Sequence)
```

E não defina ABCs customizadas (ou metaclasses) em código de produção. Se você sentir uma forte necessidade de fazer isso, aposto que é um caso da síndrome de "todos os problemas se parecem com um prego" em alguém que acabou de ganhar um novo martelo brilhante—você (e os futuros mantenedores de seu código) serão mais felizes se limitando a código simples e direto, e evitando tais profundezas. *Valê!*

Em resumo, tipagem ganso implica:

- Criar subclasses de ABCs, para tornar explícito que você está implementando uma interface previamente definida.
- Checagem de tipo durante a execução usando as ABCs em vez de classes concretas como segundo argumento para `isinstance` e `issubclass`.

Alex também aponta que herdar de uma ABC é mais que implementar os métodos necessários: é também uma declaração de intenções clara da parte do desenvolvedor. A intenção também pode ficar explícita através do registro de uma subclasse virtual.



Detalhes sobre o uso de `register` são tratados na Seção 13.5.6, mais adiante. Por hora, aqui está um pequeno exemplo: dada a classe `FrenchDeck`, se eu quiser que ela passe em uma checagem como `issubclass(FrenchDeck, Sequence)`, posso torná-la uma *subclasse virtual* da ABC `Sequence` assim:

```
from collections.abc import Sequence
Sequence.register(FrenchDeck)
```

O uso de `isinstance` e `issubclass` se torna mais aceitável se você está checando ABCs em vez de classes concretas. Se usadas com classes concretas, checagens de tipo limitam o polimorfismo—um recurso essencial da programação orientada a objetos. Mas com ABCs esses testes são mais flexíveis. Afinal, se um componente não implementa uma ABC sendo uma subclasse—mas implementa os métodos

necessários—ele sempre pode ser registrado posteriormente e passar naquelas checagens de tipo explícitas.

Entretanto, mesmo com ABCs, você deve se precaver contra o uso excessivo de checagens com `isinstance`, pois isso pode ser sintoma de um design ruim.

Normalmente, não é bom ter uma série de `if/elif/elif` com checagens de `isinstance` executando ações diferentes, dependendo do tipo de objeto: neste caso você deveria estar usando polimorfismo—isto é, projetando suas classes para permitir ao interpretador invocar os métodos corretos, em vez de codificar diretamente a lógica de despacho em blocos `if/elif/elif`.

Por outro lado, não há problema em executar uma checagem com `isinstance` contra uma ABC se você quer garantir um contrato de API: "Cara, você precisa implementar isso se quiser me chamar," como costuma dizer o revisor técnico Lennart Regebro. Isso é especialmente útil em sistemas com arquiteturas modulares extensíveis por plug-ins. Fora dos frameworks, tipagem pato muitas vezes é mais simples e flexível que checagens de tipo explícitas.

Por fim, em seu ensaio Alex reforça mais de uma vez a necessidade de limitar a criação de ABCs. Uso excessivo de ABCs imporia cerimônia a uma linguagem que se tornou popular por ser prática e pragmática. Durante o processo de revisão do *Python Fluente*, Alex colocou num e-mail:

ABCs servem para encapsular conceitos muito genéricos, abstrações introduzidas por um framework—coisa como "uma sequência" e "um número exato". [Os leitores] quase certamente não precisam escrever alguma nova ABC, apenas usar as já existentes de forma correta, para obter 99% dos benefícios sem qualquer risco sério de design mal-feito.

Agora vamos ver a tipagem ganso na prática.

13.5.1. Criando uma subclasse de uma ABC

Seguindo o conselho de Martelli, vamos aproveitar uma ABC existente, `collections.MutableSequence`, antes de ousar inventar uma nova. No Exemplo 6, `FrenchDeck2` é explicitamente declarada como subclasse de `collections.MutableSequence`.

Exemplo 6. frenchdeck2.py: FrenchDeck2, uma subclasse de collections.MutableSequence

```
from collections import namedtuple, abc

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

class FrenchDeck2(abc.MutableSequence):
    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]

    def __setitem__(self, position, value): ①
        self._cards[position] = value

    def __delitem__(self, position): ②
        del self._cards[position]

    def insert(self, position, value): ③
        self._cards.insert(position, value)
```

- ① `__setitem__` é tudo que precisamos para possibilitar o embaralhamento...
- ② ...mas uma subclasse de `MutableSequence` é forçada a implementar `__delitem__`, um método abstrato daquela ABC.
- ③ Também precisamos implementar `insert`, o terceiro método abstrato de `MutableSequence`.

Python não verifica a implementação de métodos abstratos durante a importação (quando o módulo *frenchdeck2.py* é carregado na memória e compilado), mas apenas durante a execução, quando tentamos de fato instanciar `FrenchDeck2`. Ali, se deixamos de implementar qualquer um dos métodos abstratos, recebemos

uma exceção de `TypeError` com uma mensagem como *Can't instantiate abstract class FrenchDeck2 with abstract methods __delitem__, insert* (Impossível instanciar a classe abstrata `FrenchDeck2` com os métodos abstratos `__delitem__`, `insert`). Por isso precisamos implementar `__delitem__` e `insert`, mesmo se nossos exemplos usando `FrenchDeck2` não precisem desses comportamentos: a ABC `MutableSequence` os exige.

Como mostra a Figura 3, nem todos os métodos das ABCs `Sequence` e `MutableSequence` são abstratos.

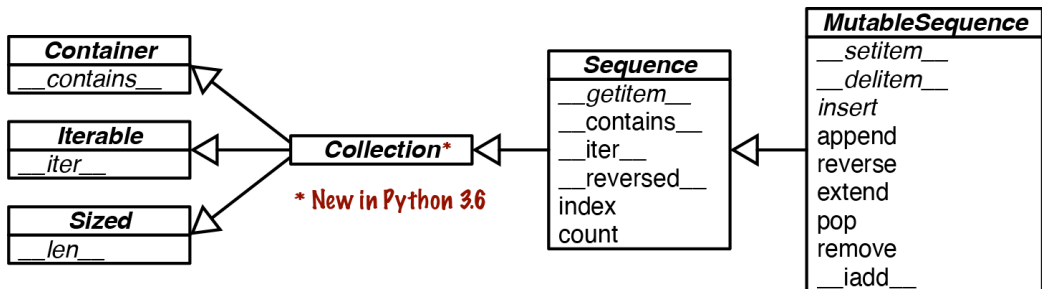


Figura 3. Diagrama de classe UML para a ABC `MutableSequence` e suas superclasses em `collections.abc` (as setas de herança apontam das subclasses para as ancestrais; nomes em *itálico* são classes e métodos abstratos).

Para escrever `FrenchDeck2` como uma subclasse de `MutableSequence`, tive que pagar o preço de implementar `__delitem__` e `insert`, desnecessários em meus exemplos. Em troca, `FrenchDeck2` herda cinco métodos concretos de `Sequence`: `__contains__`, `__iter__`, `__reversed__`, `index`, e `count`. De `MutableSequence`, ela recebe outros seis métodos: `append`, `reverse`, `extend`, `pop`, `remove`, e `__iadd__`— que suporta o operador `+=` para concatenação direta.

Os métodos concretos em cada ABC de `collections.abc` são implementados nos termos da interface pública da classe, então funcionam sem qualquer conhecimento da estrutura interna das instâncias.



Como programador de uma subclasse concreta, você pode sobrescrever os métodos concretos herdados das ABCs com implementações mais eficientes. Por exemplo, `__contains__` funciona executando uma busca sequencial, mas se a sua classe de sequência mantém os itens ordenados, você pode escrever um `__contains__` que executa uma busca binária usando a

função `bisect` [fpy.li/13-13] da biblioteca padrão. Veja *Managing Ordered Sequences with Bisect* [fpy.li/bisect] em *fluentpython.com* para conhecer mais sobre esta função.

Para usar bem as ABCs, você precisa saber o que está disponível. Vamos então revisar as ABCs de `collections` a seguir.

13.5.2. ABCs na biblioteca padrão

Desde Python 2.6, a biblioteca padrão oferece várias ABCs. A maioria está definida no módulo `collections.abc`, mas há outras nos pacotes `io` e `numbers`, por exemplo. A Figura 4 é um diagrama de classe resumido (sem os nomes dos atributos) das 17 ABCs definidas em `collections.abc`.

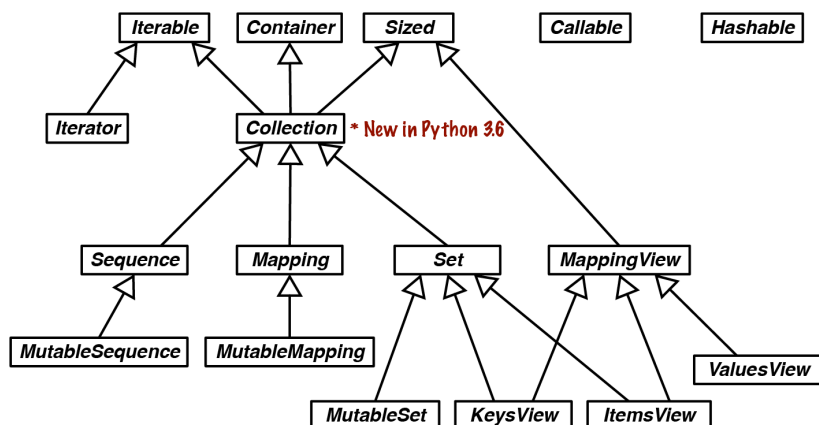


Figura 4. Diagrama de classes UML para as ABCs em `collections.abc`.



Há dois módulos chamados `abc` na biblioteca padrão. Aqui estamos falando sobre o `collections.abc`. Para reduzir o tempo de carregamento, desde o Python 3.4 aquele módulo é implementado fora do pacote `collections` — em `Lib/_collections_abc.py` [fpy.li/13-14] — então é importado separado de `collections`. O outro módulo `abc` é apenas `abc` (i.e., `Lib/abc.py` [fpy.li/13-15]), onde a classe `abc.ABC` é definida. Toda ABC depende do módulo `abc`, mas não precisamos importá-lo nós mesmos, exceto para criar uma nova ABC.

A documentação de `collections.abc` inclui uma ótima «tabela» [fpy.li/13-16] resumindo as ABCs, suas relações e seus métodos abstratos e concretos (chamados "métodos mixin"). Há muita herança múltipla acontecendo na Figura 4. Vamos dedicar a maior parte do Capítulo 14 à herança múltipla, mas por hora é suficiente dizer que isso normalmente não causa problemas no caso das ABCs.^[9] Vamos rever os grupos na Figura 4:

Iterable, Container, Sized

Toda coleção deveria herdar destas ABCs ou implementar protocolos compatíveis. `Iterable` define `__iter__` para suportar iteração, `Container` define `__contains__` para o operador `in`, e `Sized` define `__len__` para `len()`.

Collection

Essa ABC não tem nenhum método próprio, mas foi acrescentada no Python 3.6 para facilitar a criação de subclasses de `Iterable`, `Container`, e `Sized`.

Sequence, Mapping, Set

Esses são os principais tipos de coleções imutáveis, e cada um tem uma subclasse mutável. Um diagrama detalhado de `MutableSequence` é apresentado na Figura 3; para `MutableMapping` e `MutableSet`, veja os diagramas UML no «Capítulo 3» [fpy.li/3] (vol.1).

MappingView

No Python 3, os objetos devolvidos pelos métodos de mapeamentos `.items()`, `.keys()`, e `.values()` implementam as interfaces definidas em `ItemsView`, `KeysView`, e `ValuesView`, respectivamente. Os dois primeiros também implementam a rica interface de `Set`, com todos os operadores que vimos na «Seção 3.11.1» [fpy.li/8n] (vol.1).

Iterator

Observe que `iterator` é subclasse de `Iterable`. Discutiremos este detalhe em «Capítulo 17» [fpy.li/17] (vol.3).

Callable, Hashable

Estas não são coleções, mas `collections.abc` foi o primeiro pacote a definir ABCs na biblioteca padrão, e estas duas foram incluídas por serem importantes. Elas suportam a checagem de tipos de objetos que precisam ser invocáveis ou *hashable*.

Para a detecção de invocável, a função embutida `callable(obj)` é mais conveniente que `isinstance(obj, Callable)`.

Se `isinstance(obj, Hashable)` devolver `False`, pode ter certeza de que `obj` não é *hashable*. Mas se ela devolver `True`, pode ser um falso positivo. Isso é explicado no box seguinte.

`isinstance` com `Hashable` e `Iterable` pode te enganar

É fácil interpretar errado os resultados de testes usando `isinstance` e `issubclass` com as ABCs `Hashable` e `Iterable`. Quando `isinstance(obj, Hashable)` devolve `True`, significa apenas que a classe de `obj` implementa ou herda `__hash__`. Mas se `obj` é uma tupla contendo itens *unhashable*, então `obj` não é *hashable*, apesar do resultado positivo da checagem com `isinstance`. O revisor técnico Jürgen Gmach mostrou que a tipagem `pato` oferece o modo mais preciso de determinar se uma instância é *hashable*: chamar `hash(obj)`. Essa chamada vai levantar um `TypeError` se `obj` não for *hashable*.

Por outro lado, mesmo quando `isinstance(obj, Iterable)` retorna `False`, o Python ainda pode ser capaz de iterar sobre `obj` usando `__getitem__` com índices baseados em 0, como vimos em «Capítulo 1» [fpy.li/1] (vol.1) e na Seção 13.4.1. A documentação de `collections.abc.Iterable` [fpy.li/6q] afirma:

A única maneira confiável de determinar se um objeto é iterável é chamar `iter(obj)`.

Após vermos algumas das ABCs existentes, vamos praticar tipagem `ganso` implementando uma ABC do zero, e a colocando em uso. O objetivo aqui não é encorajar todo mundo a criar ABCs a torto e a direito, mas mostrar como ler o código-fonte das ABCs encontradas na biblioteca padrão e em outros pacotes.

13.5.3. Definindo e usando uma ABC

Escrevi esta advertência no capítulo "Interfaces" da primeira edição deste livro:

ABCs, como os descritores e as metaclasses, são ferramentas para criar frameworks. Assim, só uma pequena minoria dos desenvolvedores Python tem a oportunidade de criar ABCs sem impor limitações pouco razoáveis e trabalho desnecessário a seus colegas programadores.

Agora ABCs têm mais casos de uso potenciais, em dicas de tipo para permitir tipagem estática. Como discutido na «Seção 8.5.7» [fpy.li/8r] (vol.1), usar ABCs em vez de tipos concretos em dicas de tipos de argumentos de função dá mais flexibilidade a quem chama a função.

Para justificar a criação de uma ABC, precisamos pensar em um contexto para usá-la como um ponto de extensão em um framework. Então aqui está nosso contexto: imagine que você precisa exibir publicidade em um site ou em uma app de celular, em ordem aleatória, mas sem repetir um anúncio antes que o inventário completo de anúncios tenha sido exibido. Agora vamos presumir que estamos desenvolvendo um gerenciador de publicidade. Um dos requisitos é permitir o uso de classes de escolha aleatória não repetida fornecidas pelo usuário.^[10] Para deixar claro aos usuários do framework de anúncios o que se espera de um componente de "escolha aleatória não repetida", vamos definir uma ABC.

Na bibliografia sobre estruturas de dados, *stack* (pilha) e *queue* (fila) descrevem interfaces abstratas em termos dos arranjos físicos dos objetos. Vamos seguir o mesmo caminho e usar uma metáfora do mundo real para batizar nossa ABC: gaiolas de bingo e sorteadores de loteria são máquinas projetadas para escolher aleatoriamente itens de um conjunto finito, sem repetir, até o conjunto ser esgotado. Vamos chamar a ABC de Tombola, seguindo o nome italiano do bingo, e do recipiente giratório que mistura os números.

A ABC Tombola tem quatro métodos. Os dois métodos abstratos são:

.load(...)

Coloca itens na coleção.

.pick()

Remove e devolve um item aleatório da coleção.

Os métodos concretos são:

.loaded()

Devolve True se existir pelo menos um item na coleção.

.inspect()

Devolve uma tuple construída a partir dos itens atualmente na coleção, sem modificar o conteúdo (a ordem interna não é preservada).

A Figura 5 mostra a ABC Tombola e três implementações concretas. Vale notar que *registered* (registrada) e *virtual subclass* (subclasse virtual) não são termos da UML padrão, mas representam uma relação de classe específica de Python, como veremos na Seção 13.5.6.

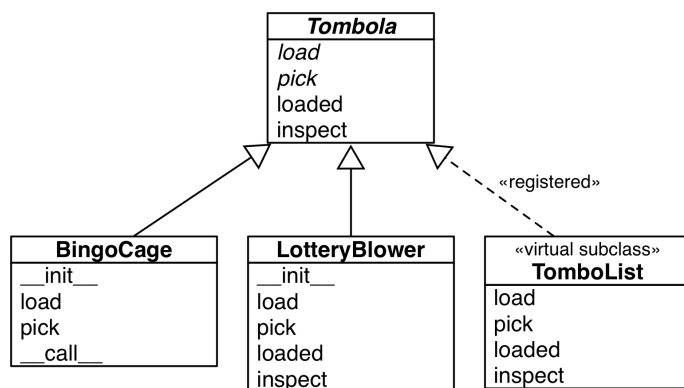


Figura 5. Diagrama UML para uma ABC e três subclasses. O nome da ABC Tombola e de seus métodos abstratos estão escritos em *itálico*, segundo as convenções da UML. A seta tracejada indica que TomboList implementa a interface Tombola, e também está registrada como subclasse virtual daquela ABC.

O Exemplo 7 mostra a definição da ABC Tombola.

Exemplo 7. tombola.py: ABC com dois métodos abstratos e dois métodos concretos.

```
import abc

class Tombola(abc.ABC): ①

    @abc.abstractmethod
    def load(self, iterable): ②
        """Add items from an iterable."""

    @abc.abstractmethod
    def pick(self): ③
        """Remove item at random, returning it.

        Must raise LookupError when the instance is empty.
        """

    def loaded(self): ④
        """True if there's at least 1 item, otherwise False."""
        return bool(self.inspect()) ⑤

    def inspect(self):
        """Return a sorted tuple with the items currently inside."""
        items = []
        while True: ⑥
            try:
                items.append(self.pick())
            except LookupError:
                break
        self.load(items) ⑦
        return tuple(items)
```

- ① Para definir uma ABC, crie uma subclasse de `abc.ABC`.
- ② Um método abstrato é marcado com o decorador `@abstractmethod`, e muitas vezes seu corpo é vazio, exceto por uma docstring.^[11]
- ③ A docstring instrui os implementadores a levantarem `LookupError` se não existirem itens para escolher.

- ④ Uma ABC pode incluir métodos concretos.
- ⑤ Métodos concretos em uma ABC devem depender apenas da interface definida pela ABC (isto é, outros métodos concretos ou abstratos ou propriedades da ABC).
- ⑥ Não sabemos como as subclasses concretas vão armazenar os itens, mas podemos escrever o resultado de `inspect` esvaziando a `Tombola` com chamadas sucessivas a `.pick()`...
- ⑦ ...e então usando `.load(...)` para colocar tudo de volta.



Um método abstrato pode ter uma implementação. Mas mesmo que tenha, as subclasses ainda são obrigadas a sobrescrevê-lo, mas poderão invocar o método abstrato com `super()`, acrescentando funcionalidade em vez de implementar do zero. Veja os detalhes do uso de `@abstractmethod` na documentação do módulo `abc` [fpy.li/6r].

O código do método `.inspect()` é ridículo mas funciona. Ele serve para mostrar que podemos usar `.pick()` e `.load(...)` para inspecionar o que está dentro de `Tombola`, puxando e devolvendo os itens—sem saber como eles são realmente armazenados. O objetivo deste exemplo é ressaltar que não há problema em oferecer métodos concretos em ABCs, desde que eles dependam apenas de outros métodos na interface. Conhecendo suas estruturas de dados internas, as subclasses concretas de `Tombola` podem sobrescrever `.inspect()` com uma implementação mais eficiente, mas não são obrigadas a fazer isso.

O método `.loaded()` no Exemplo 7 tem uma linha, mas é custoso: ele chama `.inspect()` para criar a tuple apenas para aplicar `bool()` nela. Funciona, mas subclasses concretas podem fazer bem melhor, como veremos.

Observe que nossa implementação tortuosa de `.inspect()` exige a captura de um `LookupError` lançado por `self.pick()`. O fato de `self.pick()` poder disparar um `LookupError` também é parte de sua interface, mas não há como tornar isso explícito em Python, exceto na documentação (veja a docstring para o método abstrato `pick` no Exemplo 7).

Escolhi a exceção `LookupError` por sua posição na hierarquia de exceções em relação a `IndexError` e `KeyError`, as exceções mais comuns de ocorrerem nas estruturas de dados usadas para implementar uma `Tombola` concreta. Dessa forma, as implementações podem lançar `LookupError`, `IndexError`, `KeyError`, ou uma subclasse customizada de `LookupError` para atender à interface. Veja o Exemplo 8.

Exemplo 8. Parte da hierarquia de classes de exceção.

```
BaseException
├── GeneratorExit
├── KeyboardInterrupt
├── SystemExit
├── Exception
│   ├── ArithmeticError
│   │   ├── FloatingPointError
│   │   ├── OverflowError
│   │   └── ZeroDivisionError
│   ├── AssertionError
│   ├── AttributeError
│   ├── BufferError
│   ├── EOFError
│   ├── ImportError
│   ├── LookupError      ①
│   │   ├── IndexError   ②
│   │   └── KeyError     ③
│   ├── MemoryError
│   ... etc.
```

- ① `LookupError` é a exceção que tratamos em `Tombola.inspect`.
- ② `IndexError` é a subclasse de `LookupError` gerada quando tentamos acessar um item em uma sequência com um índice além da última posição.
- ③ `KeyError` ocorre quando usamos uma chave inexistente para acessar um item em um mapeamento (`dict` etc.).

Agora temos nossa própria ABC `Tombola`. Para observar a checagem da interface feita por uma ABC, vamos tentar enganar `Tombola` com uma implementação defeituosa no Exemplo 9.

Exemplo 9. Uma Tombola falsa não passa despercebida

```
>>> from tombola import Tombola
>>> class Fake(Tombola): ①
...     def pick(self):
...         return 13
...
>>> Fake ②
<class '__main__.Fake'>
>>> f = Fake() ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Fake with abstract method
load
```

- ① Declara Fake como subclasse de Tombola.
- ② A classe é criada, nenhum erro até agora.
- ③ Um `TypeError` é sinalizado quando tentamos instanciar Fake. A mensagem é bastante clara: Fake é considerada abstrata porque deixou de implementar `load`, um dos métodos abstratos declarados na ABC Tombola.

Então definimos nossa primeira ABC, e a usamos para validar uma classe. Logo vamos criar uma subclasse de Tombola, mas primeiro temos que falar sobre algumas regras para a programação de ABCs.

13.5.4. Detalhes da sintaxe das ABCs

A forma padrão de declarar uma ABC é criar uma subclasse de `abc.ABC` ou de alguma outra ABC.

Além da classe base ABC e do decorador `@abstractmethod`, o módulo `abc` define os decoradores `@abstractclassmethod`, `@abstractstaticmethod`, e `@abstractproperty`. Entretanto, os três últimos foram descontinuados no Python 3.3, quando se tornou possível empilhar decoradores sobre `@abstractmethod`, tornando os outros redundantes.

Por exemplo, a maneira preferível de declarar um método de classe abstrato é:

```
class MyABC(abc.ABC):
    @classmethod
    @abc.abstractmethod
    def an_abstract_classmethod(cls, ...):
        pass
```



A ordem dos decoradores de função empilhados importa, e no caso de `@abstractmethod`, a documentação é explícita:

Quando `@abstractmethod` é aplicado em combinação com outros descritores de método, ele deve ser aplicado como o decorador mais interno...^[12]

Em outras palavras, nenhum outro decorador pode aparecer entre `@abstractmethod` e a instrução `def`.

Agora que abordamos essas questões de sintaxe das ABCs, vamos colocar `Tombola` em uso, implementando duas subclasses concretas.

13.5.5. Criando uma subclasse de `Tombola`

Dada a ABC `Tombola`, vamos desenvolver duas subclasses concretas que satisfazem a interface. Essas classes estão ilustradas na Figura 5, junto com a subclasse virtual que será discutida na seção seguinte.

A classe `BingoCage` no Exemplo 10 é uma variação do «Exemplo 8 do Capítulo 7» [fpy.li/8y] (vol.1) usando um randomizador melhor. `BingoCage` implementa os métodos abstratos obrigatórios `load` e `pick`.

Exemplo 10. `bingo.py`: `BingoCage` é uma subclasse concreta de `Tombola`

```
import random

from tombola import Tombola
```

```

class BingoCage(Tombola): ①

    def __init__(self, items):
        self._randomizer = random.SystemRandom() ②
        self._items = []
        self.load(items) ③

    def load(self, items):
        self._items.extend(items)
        self._randomizer.shuffle(self._items) ④

    def pick(self): ⑤
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')

    def __call__(self): ⑥
        self.pick()

```

- ① Essa classe `BingoCage` estende `Tombola` explicitamente.
- ② `random.SystemRandom()` implementa a API `random` sobre a função `os.urandom()`, que fornece bytes aleatórios "adequados para uso em criptografia", segundo a documentação do módulo `os` [fpy.li/6t].
- ③ Delega o carregamento inicial para o método `.load()`
- ④ Em vez da função `random.shuffle()` normal, usamos o método `.shuffle()` de nossa instância de `SystemRandom`.
- ⑤ `pick` é implementado como no «Exemplo 8 do Capítulo 7» [fpy.li/8y] (vol.1).
- ⑥ `__call__` também é do «Exemplo 8 do Capítulo 7» [fpy.li/8y] (vol.1). Ele não é necessário para satisfazer a interface de `Tombola`, mas é comum que subclasses tenham mais métodos.

`BingoCage` herda o custoso método `loaded` e o tolo `inspect` de `Tombola`. Ambos poderiam ser sobrescritos com métodos de uma linha mais rápidos, como no Exemplo 11. A questão é: podemos decidir apenas herdar os métodos concretos de uma ABC. Os métodos herdados de `Tombola` não são tão rápidos quanto poderiam ser na `BingoCage` concreta, mas fornecem os resultados esperados para qualquer subclasse de `Tombola` que implemente `pick` e `load` corretamente.

O Exemplo 11 mostra uma implementação muito diferente, mas também válida, da interface de Tombola. Em vez de misturar as "bolas" e tirar a última, LottoBlower tira um item de uma posição aleatória..

Exemplo 11. lotto.py: LottoBlower é uma subclasse concreta que sobrecarrega os métodos inspect e loaded de Tombola

```
import random

from tombola import Tombola

class LottoBlower(Tombola):

    def __init__(self, iterable):
        self._balls = list(iterable) ①

    def load(self, iterable):
        self._balls.extend(iterable)

    def pick(self):
        try:
            position = random.randrange(len(self._balls)) ②
        except ValueError:
            raise LookupError('pick from empty LottoBlower')
        return self._balls.pop(position) ③

    def loaded(self): ④
        return bool(self._balls)

    def inspect(self): ⑤
        return tuple(self._balls)
```

- ① O construtor aceita qualquer iterável: o argumento é usado para construir uma lista.
- ② A função `random.randrange(...)` levanta um `ValueError` se a faixa de valores estiver vazia, então capturamos esse erro e trocamos por `LookupError`, para ser compatível com `Tombola`.
- ③ Caso contrário, o item selecionado aleatoriamente é retirado de `self._balls`.

- ④ Sobrescreve `loaded` para evitar a chamada a `inspect` (como `Tombola.loaded` faz no Exemplo 7). Podemos fazer isso mais rápido acessando `self._balls` diretamente—não precisamos criar uma nova tupla.
- ⑤ Sobrescreve `inspect` com uma linha de código.

O Exemplo 11 ilustra um idioma que vale a pena mencionar: em `__init__`, `self._balls` armazena `list(iterable)`, e não apenas uma referência para `iterable` (isto é, nós não apenas atribuímos `self._balls = iterable`, apelidando o argumento). Como mencionado na Seção 13.4.3, isso torna a `LottoBlower` flexível, pois o argumento `iterable` pode ser de qualquer tipo iterável. Ao mesmo tempo, garantimos que os itens serão armazenados em uma `list`, de onde podemos retirar itens com `.pop()`. E mesmo quando recebemos uma lista no argumento `iterable`, `list(iterable)` produz uma cópia, o que é uma boa prática, considerando que vamos remover itens da lista, e o cliente pode não estar esperando que a lista passada seja modificada.^[13]

Chegamos agora à característica dinâmica da tipagem ganso: declarar subclasses virtuais com o método `register`.

13.5.6. Uma subclasse virtual de uma ABC

Uma característica essencial da tipagem ganso—e uma razão pela qual ela merece um nome de ave aquática—é a habilidade de registrar uma classe como uma *subclasse virtual* de uma ABC, mesmo se a classe não herde da ABC. Ao fazer isso, prometemos que a classe implementa fielmente a interface definida na ABC—e Python vai acreditar em nós sem checar. Se mentirmos, vamos enfrentar exceções de tempo de execução.

Isso é feito invocando um método de classe `register` da ABC. A subclasse registrada será reconhecida por `issubclass`, mas herdará qualquer método ou atributo da ABC.



Subclasses virtuais não herdam da ABC na qual se registram, e sua conformidade com a interface da ABC nunca é checada, nem quando são instanciadas. E mais, neste momento checadores de tipos estáticos não conseguem tratar subclasses virtuais. Mais detalhes em [Mypy issue 2922—ABCMeta.register support](https://mypy.readthedocs.io/en/latest/faq.html#abc-meta-register-support) [fpy.li/13-22].

O método `register` é normalmente invocado como uma função comum (veja a Seção 13.5.7), mas também pode ser usado como decorador. No Exemplo 12, usamos a sintaxe de decorador e implementamos `TomboList`, uma subclasse virtual de `Tombola`, ilustrada na Figura 6.

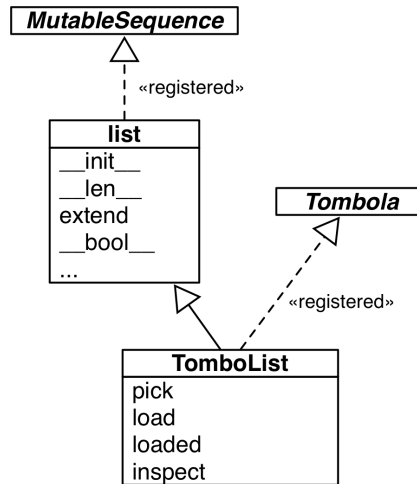


Figura 6. Diagrama de classe UML para `TomboList`, subclasse real de `list` e subclasse virtual de `Tombola`.

Exemplo 12. `tombolist.py`: a classe `TomboList` é uma subclasse virtual de `Tombola`

```

from random import randrange

from tombola import Tombola

@Tombola.register ①
class TomboList(list): ②

    def pick(self):
        if self: ③
            position = randrange(len(self))
            return self.pop(position) ④
        else:
            raise LookupError('pop from empty TomboList')

    load = list.extend ⑤

    def loaded(self):
        return bool(self) ⑥
  
```

```
def inspect(self):  
    return tuple(self)
```

```
# Tombola.register(TombolList) ⑦
```

- ① `TombolList` é registrada como subclasse virtual de `Tombola`.
- ② `TombolList` estende `list`.
- ③ `TombolList` herda seu comportamento booleano de `list`, devolvendo `True` se a lista não estiver vazia.
- ④ Nosso `pick` invoca `self.pop`, herdado de `list`, passando um índice aleatório para um item.
- ⑤ `TombolList.load` é o mesmo que `list.extend`.
- ⑥ `loaded` delega para `bool`.^[14]
- ⑦ É sempre possível invocar `register` dessa forma, e é útil fazer assim quando você precisa registrar uma classe cujo código você não mantém, mas que implementa a interface.

Note que, por causa do registro, as funções `issubclass` e `isinstance` agem como se `TombolList` fosse uma subclasse de `Tombola`:

```
>>> from tombola import Tombola  
>>> from tombolist import TombolList  
>>> issubclass(TombolList, Tombola)  
True  
>>> t = TombolList(range(100))  
>>> isinstance(t, Tombola)  
True
```

Entretanto, a herança é guiada por um atributo de classe especial chamado `__mro__`—sigla de *Method Resolution Order* (Ordem de Resolução de Métodos). Esse atributo lista a classe e suas superclasses na ordem que Python segue para procurar métodos.^[15] Se você inspecionar o `__mro__` de `TombolList`, verá que ele lista apenas as superclasses "reais"—`list` e `object`:

```
>>> TombolList.__mro__  
(<class 'tombolist.TombolList'>, <class 'list'>, <class 'object'>)
```

Tombola não está em `TombolList.__mro__`, então `TombolList` não herda nenhum método de `Tombola`.

Isso conclui nosso estudo de caso da ABC `Tombola`. Na próxima seção, vamos falar sobre como a função `register` das ABCs é usada na vida real.

13.5.7. O uso de `register` na prática

No Exemplo 12, usamos `Tombola.register` como um decorador de classe. Antes de Python 3.3, `register` não podia ser usado dessa forma—ele tinha que ser invocado como uma função normal após a definição da classe, como sugerido pelo comentário no final do Exemplo 12. Mas `register` continua sendo usado como uma função para registrar classes definidas em outro lugar. Por exemplo, no código-fonte [fpy.li/13-24] do módulo `collections.abc`, os tipos nativos `tuple`, `str`, `range`, e `memoryview` são registrados como subclasses virtuais de `Sequence` assim:

```
Sequence.register(tuple)  
Sequence.register(str)  
Sequence.register(range)  
Sequence.register(memoryview)
```

Vários outros tipos nativos estão registrados com as ABCs em `_collections_abc.py`. Esses registros ocorrem apenas quando aquele módulo é importado, o que não causa problema, pois você terá mesmo que importar o módulo para obter as ABCs. Por exemplo, você precisa importar `MutableMapping` de `collections.abc` para checar algo como `isinstance(my_dict, MutableMapping)`.

Criar uma subclasse de uma ABC ou se registrar com uma ABC são duas maneiras explícitas de fazer nossas classes passarem checagens com `issubclass` e `isinstance` (que também se apoia em `issubclass`). Mas algumas ABCs também suportam tipagem estrutural. A próxima seção explica isso.

13.5.8. Tipagem estrutural com ABCs

As ABCs são usadas principalmente com tipagem nominal.

Quando uma classe `Sub` herda explicitamente de `UmaABC`, ou está registrada com `UmaABC`, o nome de `UmaABC` fica ligado ao da classe `Sub`—é assim que, durante a execução, `issubclass(UmaABC, Sub)` devolve `True`.

Em contraste, a tipagem estrutural diz respeito a olhar para a estrutura da interface pública de um objeto para determinar seu tipo: um objeto é *consistente-com* um tipo se implementa os métodos definidos no tipo.^[16] A tipagem pato estática e a tipagem pato dinâmica são duas abordagens à tipagem estrutural.

Acontece que algumas ABCs também suportam tipagem estrutural. Em seu ensaio *Pássaros aquáticos e as ABCs*, Alex mostra que uma classe pode ser reconhecida como subclasse de uma ABC mesmo sem registro. Aqui está novamente o exemplo dele, com um teste adicional usando `issubclass`:

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
>>> issubclass(Struggle, abc.Sized)
True
```

A classe `Struggle` é considerada uma subclasse de `abc.Sized` pela função `issubclass` (e, consequentemente, também por `isinstance`) porque `abc.Sized` implementa um método de classe especial chamado `__subclasshook__`.

O `__subclasshook__` de `Sized` verifica se o argumento classe tem um atributo chamado `__len__`. Se tiver, então a classe é considerada uma subclasse virtual de `Sized`. Veja o Exemplo 13.

```
class Sized(metaclass=ABCMeta):

    __slots__ = ()

    @abstractmethod
    def __len__(self):
        return 0

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Sized:
            if any("__len__" in B.__dict__ for B in C.__mro__): ①
                return True ②
            return NotImplemented ③
```

- ① Se há um atributo chamado `__len__` no `__dict__` de qualquer classe listada em `C.__mro__` (isto é, `C` e suas superclasses)...
- ② ...devolve `True`, sinalizando que `C` é uma subclasse virtual de `Sized`.
- ③ Caso contrário devolve `NotImplemented`, para permitir que a checagem de subclasse continue.



Se você tiver interesse nos detalhes da checagem de subclasse, estude o código-fonte do método `ABCMeta.__subclasscheck__` no Python 3.6: *Lib/abc.py* [fpy.li/13-26]. Saiba que é complicado: lá há muitos ifs e duas chamadas recursivas. No Python 3.7, Ivan Levkivskyi e Inada Naoki reescreveram em C a maior parte da lógica do módulo `abc`, para melhorar o desempenho. Veja Python issue #31333 [fpy.li/13-27]. A implementação atual de `ABCMeta.__subclasscheck__` simplesmente chama `abc_subclasscheck`. O código-fonte em C relevante está em `_cpython/Modules/_abc.c#L605` [fpy.li/13-28].

É assim que `__subclasshook__` permite às ABCs suportarem a tipagem estrutural. Você pode formalizar uma interface com uma ABC, pode fazer checagens `isinstance` com aquela ABC, e ainda ter uma classe sem qualquer relação de herança aprovada por uma checagem de `issubclass` porque ela implementa um

certo método (ou porque ela faz o necessário para convencer o `__subclasshook__` da ABC aprová-la como subclasse virtual).

É uma boa ideia implementar `__subclasshook__` em nossas próprias ABCs? Provavelmente não. Todas as implementações de `__subclasshook__` que vi no código-fonte de Python estão em ABCs como `Sized`, que declara apenas um método especial, e elas simplesmente verificam a presença do nome daquele método especial. Dado seu status "especial", é quase certeza que qualquer método chamado `__len__` faz o que se espera. Mas mesmo no reino dos métodos especiais e ABCs fundamentais, pode ser arriscado fazer tais suposições. Por exemplo, mapeamentos implementam `__len__`, `__getitem__`, e `__iter__`, mas corretamente não são considerados subtipos de `Sequence`, pois não podemos recuperar itens usando índices a partir de zero ou obter fatias. Por isso a classe `abc.Sequence` [fpy.li/13-29] não implementa `__subclasshook__`.

Para ABCs que você ou eu podemos escrever, um `__subclasshook__` seria ainda menos confiável. Não estou preparado para acreditar que qualquer classe chamada `Spam` que implemente ou herde `load`, `pick`, `inspect`, e `loaded` vai necessariamente se comportar como uma `Tombola`. É melhor deixar o programador afirmar isso, fazendo de `Spam` uma subclasse de `Tombola`, ou registrando a classe com `Tombola.register(Spam)`. Claro, o seu `__subclasshook__` poderia também verificar assinaturas de métodos e outras características, mas não creio que valha o esforço.

13.6. Protocolos estáticos



Vimos algo sobre protocolos estáticos na «Seção 8.5.10» [fpy.li/8m] (vol.1). Pensei em deixar toda a discussão sobre protocolos para este capítulo, mas decidi que a apresentação inicial de dicas de tipo em funções precisava incluir protocolos, pois a tipagem pato é uma parte essencial de Python, e a checagem de tipos estática sem protocolos não consegue lidar muito bem com muitas APIs pythônicas.

Vamos encerrar este capítulo ilustrando os protocolos estáticos com dois exemplos simples, e uma discussão sobre as ABCs numéricas e protocolos. Começaremos mostrando como um protocolo estático possibilita anotar e checar tipos na função `double()`, que vimos antes na «Seção 8.4» [fpy.li/8s] (vol.1).

13.6.1. A função `double` tipada

Quando apresento Python para programadores mais habituados à tipagem estática, um de meus exemplos é esta função `double` capaz de lidar com uma variedade de tipos:

```
>>> def double(x):
...     return x * 2
...
>>> double(1.5)
3.0
>>> double('A')
'AA'
>>> double([10, 20, 30])
[10, 20, 30, 10, 20, 30]
>>> from fractions import Fraction
>>> double(Fraction(2, 5))
Fraction(4, 5)
```

Antes da introdução dos protocolos estáticos, não havia uma forma prática de acrescentar dicas de tipo a `double` sem limitar seus usos possíveis.^[17]

Graças à tipagem `pato`, `double` funciona mesmo com tipos inventados depois, tal como a classe `Vector` aprimorada que veremos na Seção 16.5:

```
>>> from vector_v7 import Vector
>>> double(Vector([11.0, 12.0, 13.0]))
Vector([22.0, 24.0, 26.0])
```

A implementação inicial de dicas de tipo no Python era um sistema de tipos nominal: o nome de um tipo em uma anotação tinha que corresponder ao nome do tipo do argumento real—ou com o nome de uma de suas superclasses. Como é impossível nomear todos os tipos que implementam um protocolo (suportando as operações requeridas), a tipagem `pato` não podia ser descrita por dicas de tipo antes do Python 3.8.

Agora, com `typing.Protocol`, podemos informar ao Mypy que `double` recebe um argumento `x` que suporta `x * 2`. O Exemplo 14 mostra como.

Exemplo 14. `double_protocol.py`: a definição de `double` usando um `Protocol`.

```
from typing import TypeVar, Protocol

T = TypeVar('T') ①

class Repeatable(Protocol):
    def __mul__(self: T, repeat_count: int) -> T: ... ②

RT = TypeVar('RT', bound=Repeatable) ③

def double(x: RT) -> RT: ④
    return x * 2
```

- ① Vamos usar esse `T` na assinatura de `__mul__`.
- ② `__mul__` é a essência do protocolo `Repeatable`. O parâmetro `self` normalmente não é anotado—presume-se que seu tipo seja a classe. Aqui usamos `T` para assegurar que o tipo do resultado é o mesmo tipo de `self`. Além disso observe que decidi limitar `repeat_count` ao tipo `int` neste protocolo.
- ③ A variável de tipo `RT` é vinculada pelo protocolo `Repeatable`: o checador de tipos vai exigir que o tipo efetivo implemente `Repeatable`.
- ④ Agora o checador de tipos pode checar que o parâmetro `x` é um objeto que pode ser multiplicado por um inteiro, e que o valor retornado tem o mesmo tipo que `x`.

Este exemplo mostra por que o subtítulo da PEP 544 [fpy.li/pep544] é *static duck typing* (tipagem pato estática). O tipo nominal do argumento concreto `x` passado a `double`, é irrelevante, desde que `grasne`—ou seja, desde que implemente `__mul__`.

13.6.2. Protocolos estáticos checados durante a execução

No Mapa de Tipagem (Figura 1), `typing.Protocol` aparece na área de checagem estática—a metade inferior do diagrama. Entretanto, ao definir uma subclasse de `typing.Protocol`, você pode usar o decorador `@runtime_checkable` para fazer aquele protocolo aceitar checagens com `isinstance`/`issubclass` durante a execução. Isso funciona porque `typing.Protocol` é uma ABC, assim suporta o `__subclasshook__` que vimos na Seção 13.5.8.

No Python 3.9, o módulo `typing` inclui sete protocolos prontos para uso que são verificáveis durante a execução. Aqui estão dois deles, citados diretamente da documentação de `typing` [fpy.li/gv]:

`class typing.SupportsComplex`

Um ABC com um método abstrato *complex*.

`class typing.SupportsFloat`

Um ABC com um método abstrato *float*.

Estes protocolos foram projetados para checar a "convertibilidade" de tipos numéricos: se um objeto `n` implementa `__complex__`, então deveria ser possível obter um `complex` invocando `complex(n)`, pois o método especial `__complex__` existe para suportar a função embutida `complex()`.

Exemplo 15 mostra o código-fonte [fpy.li/13-31] do protocolo `typing.SupportsComplex`.

Exemplo 15. código-fonte do protocolo `typing.SupportsComplex`

```
@runtime_checkable
class SupportsComplex(Protocol):
    """An ABC with one abstract method __complex__."""
    __slots__ = ()

    @abstractmethod
    def __complex__(self) -> complex:
        pass
```

A chave é o método abstrato `__complex__`.^[18] Durante a checagem de tipo estática, um objeto será considerado *consistente-com* o protocolo `SupportsComplex` se implementar um método `__complex__` que recebe apenas `self` e retorna um `complex`.

Graças ao decorador de classe `@runtime_checkable`, aplicado a `SupportsComplex`, aquele protocolo também pode ser utilizado em checagens com `isinstance` no Exemplo 16.

Exemplo 16. Usando SupportsComplex durante a execução

```
>>> from typing import SupportsComplex
>>> import numpy as np
>>> c64 = np.complex64(3+4j) ①
>>> isinstance(c64, complex) ②
False
>>> isinstance(c64, SupportsComplex) ③
True
>>> c = complex(c64) ④
>>> c
(3+4j)
>>> isinstance(c, SupportsComplex) ⑤
False
>>> complex(c)
(3+4j)
```

- ① `complex64` é um dos cinco tipos de números complexos fornecidos pelo NumPy.
- ② Nenhum dos tipos complexos da NumPy é subclasse do `complex` embutido.
- ③ Mas os tipos complexos de NumPy implementam `__complex__`, então cumprem o protocolo `SupportsComplex`.
- ④ Portanto, você pode criar objetos `complex` a partir deles.
- ⑤ O tipo `complex` embutido não implementa `__complex__`, mas `complex(c)` funciona sem problemas se `c` for uma instância de `complex`.

Como consequência deste último ponto, se você quiser testar se um objeto `c` é um `complex` ou `SupportsComplex`, você deve passar uma tupla de tipos como segundo argumento para `isinstance`, assim:

```
isinstance(c, (complex, SupportsComplex))
```

Uma outra alternativa seria usar a `ABC Complex`, definida no módulo `numbers`. O tipo embutido `complex` e os tipos `complex64` e `complex128` da NumPy são todos registrados como subclasses virtuais de `numbers.Complex`, então o código a seguir funciona:

```
>>> import numbers
>>> isinstance(c, numbers.Complex)
True
>>> isinstance(c64, numbers.Complex)
True
```

Na primeira edição de *Python Fluente* eu recomendava o uso das ABCs de `numbers`, mas agora esse não é mais um bom conselho, pois aquelas ABCs não são reconhecidas pelos checadores de tipos estáticos, como veremos na Seção 13.6.8.

Nesta seção eu queria demonstrar que um protocolo verificável durante a execução funciona com `isinstance`, mas na verdade esse exemplo não é um caso de uso particularmente bom de `isinstance`, como a barra lateral Confie no pato explica.



Se você estiver usando o `Mypy`, há uma vantagem nas checagens explícitas com `isinstance`: quando você escreve uma instrução `if` onde a condição é `isinstance(n, MyType)`, então o `Mypy` infere que dentro do bloco `if`, o tipo do objeto `n` é *consistente-com* `MyType`.

Confie no pato

Durante a execução, muitas vezes a tipagem pato é a melhor abordagem para checagem de tipos: em vez de chamar `isinstance` ou `hasattr`, apenas tente realizar as operações que você precisa com o objeto, e trate as exceções conforme necessário. Segue um exemplo concreto.

Continuando a discussão anterior, dado um objeto `n` que preciso usar como número complexo, essa seria uma abordagem:

```
if isinstance(n, (complex, SupportsComplex)):
    # código que precisa converter `n` para `complex`
else:
    raise TypeError('n must be convertible to complex')
```


A abordagem da tipagem ganso seria usar a `ABC numbers.Complex`:

```
if isinstance(n, numbers.Complex):
    # código que assume que 'n' é instância de 'Complex'
else:
    raise TypeError('n must be an instance of Complex')
```

Mas eu prefiro aproveitar a tipagem pato e pedir perdão em vez de permissão (Princípio de Hopper):

```
try:
    c = complex(n)
except TypeError as exc:
    raise TypeError('n must be convertible to complex') from exc
```

Mas se o único tratamento que você vai dar para o `TypeError` é levantar `TypeError`, eu escreveria só isso:

```
c = complex(n)
```

Neste último caso, se `n` não é de um tipo aceitável, o Python levantará uma exceção com uma mensagem bem clara. Por exemplo, se `n` é uma `tuple`, esse é o resultado:

```
TypeError: complex() first argument must be a string or a number,
not 'tuple'
```

Em português: "O primeiro argumento de `complex()` deve ser uma string ou um número, não 'tuple'".

A abordagem da tipagem pato é simples e correta neste caso.

Agora que vimos como usar protocolos estáticos durante a execução com tipos pré-existentes como `complex` e `numpy.complex64`, precisamos discutir as limitações de protocolos verificáveis durante a execução.

13.6.3. Limitações das checagens de protocolo durante a execução

Vimos que dicas de tipo são geralmente ignoradas durante a execução, e isso também afeta o uso de checagens com `isinstance` ou `issubclass` com protocolos estáticos.

Por exemplo, qualquer classe com um método `__float__` é considerada—durante a execução—uma subclasse virtual de `SupportsFloat`, mesmo se seu método `__float__` não devolver um `float`.

Veja essa sessão no console:

```
>>> import sys
>>> sys.version
'3.9.5 (v3.9.5:0a7dcdb13, May 3 2021, 13:17:02) \n[Clang 6.0 (clang-600.0.57)]'
>>> c = 3+4j
>>> c.__float__
<method-wrapper '__float__' of complex object at 0x10a16c590>
>>> c.__float__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float
```

Em Python 3.9, o tipo `complex` tem um método `__float__`, mas ele existe apenas para gerar `TypeError` com uma mensagem de erro explícita. Se aquele método `__float__` tivesse anotações, o tipo de retorno seria `NoReturn`—que vimos na «Seção 8.5.12» [fpy.li/8f] (vol.1).

Mas incluir dicas de tipo em `complex.__float__` no *typeshed* não resolveria esse problema, porque o interpretador Python em geral ignora dicas de tipo—e também não acessa os arquivos de anotações de tipo do *typeshed*.

Continuando da sessão anterior de Python 3.9:

```
>>> from typing import SupportsFloat
>>> c = 3+4j
>>> isinstance(c, SupportsFloat)
True
```

```
>>> isinstance(complex, SupportsFloat)
True
```

Então temos resultados enganosos: as checagens durante a execução usando `SupportsFloat` sugerem que você pode converter um `complex` para `float`, mas na verdade isso gera um erro de tipo.



O problema específico com o tipo `complex` foi resolvido no Python 3.10, com a remoção do método `complex.__float__`.

Mas o problema geral persiste: checagens com `isinstance` / `issubclass` só olham para a presença ou ausência de métodos, sem checar sequer suas assinaturas, muito menos suas anotações de tipo. E isso não vai mudar tão cedo, porque este tipo de checagem de tipos durante a execução traria um custo de processamento inaceitável.^[19]

Agora vamos implementar um protocolo estático em uma classe concreta.

13.6.4. Suportando um protocolo estático

Lembra da classe `Vector2d`, que desenvolvemos no Capítulo 11? Dado que um número `complex` e uma instância de `Vector2d` contém um par de números de ponto flutuante, faz sentido suportar a conversão de `Vector2d` para `complex`.

O Exemplo 17 mostra a implementação do método `__complex__`, para melhorar a última versão de `Vector2d`, vista no Exemplo 11 do Capítulo 11. Para deixar o serviço completo, podemos suportar a operação inversa, com um método de classe `fromcomplex`, que constrói um `Vector2d` a partir de um `complex`.

Exemplo 17. `vector2d_v4.py`: métodos para conversão de e para `complex`

```
def __complex__(self):
    return complex(self.x, self.y)

@classmethod
def fromcomplex(cls, n):
    return cls(n.real, n.imag) ①
```

- ① Presume que `n` tem atributos `.real` e `.imag`. Veremos uma implementação melhor no Exemplo 18.

Dado o código acima, e o método `__abs__` que o `Vector2d` já tinha em Exemplo 11 do Capítulo 11, temos o seguinte:

```
>>> from typing import SupportsComplex, SupportsAbs
>>> from vector2d_v4 import Vector2d
>>> v = Vector2d(3, 4)
>>> isinstance(v, SupportsComplex)
True
>>> isinstance(v, SupportsAbs)
True
>>> complex(v)
(3+4j)
>>> abs(v)
5.0
>>> Vector2d.fromcomplex(3+4j)
Vector2d(3.0, 4.0)
```

Para checagem de tipos durante a execução, o Exemplo 17 serve bem, mas para uma cobertura estática e relatório de erros melhores com o Mypy, os métodos `__abs__`, `__complex__`, e `fromcomplex` deveriam receber dicas de tipo, como mostrado no Exemplo 18.

Exemplo 18. `vector2d_v5.py`: acrescentando anotações aos métodos mencionados

```
def __abs__(self) -> float: ①
    return math.hypot(self.x, self.y)

def __complex__(self) -> complex: ②
    return complex(self.x, self.y)

@classmethod
def fromcomplex(cls, n: complex | SupportsComplex) -> Vector2d: ③
    c = complex(n) ④
    return cls(c.real, c.imag)
```

- ① A anotação de resultado `float` é necessária, senão o Mypy infere `Any`, e não checa o corpo do método.

- ② Mesmo sem a anotação, o Mypy inferiu que isto devolve um `complex`. A anotação evita um aviso, dependendo da configuração do Mypy.
- ③ Aqui `SupportsComplex` garante que `n` é conversível.
- ④ Esta conversão explícita é necessária, pois um tipo *consistente-com* `SupportsComplex` não necessariamente tem os atributos `.real` e `.img`, que usamos na linha seguinte. A própria classe `Vector2d` não tem estes atributos, mas implementa `__complex__`.

O tipo de retorno de `fromcomplex` pode ser `Vector2d` se a linha `from __future__ import annotations` aparecer no início do módulo. Aquela importação faz as dicas de tipo serem armazenadas como strings, sem serem processadas durante a importação, quando as definições de função são tratadas. Sem o `__future__ import of annotations`, `Vector2d` é uma referência inválida neste momento (a classe não está inteiramente definida ainda) e deveria ser escrita como uma string: `'Vector2d'`, como se fosse uma referência adiantada. Essa importação de `__future__` foi introduzida na PEP 563—Postponed Evaluation of Annotations [fpy.li/pep563], implementada no Python 3.7. Aquele comportamento estava marcado para se tornar default no 3.10, mas a mudança foi adiada para uma versão futura.^[20] Quando isso acontecer, a importação será redundante mas inofensiva.

Agora vamos criar—e depois estender—um novo protocolo estático.

13.6.5. Projetando um protocolo estático

Quando estudamos tipagem ganso, vimos a ABC `Tombola` na Seção 13.5.3. Aqui vamos ver como definir uma interface similar usando um protocolo estático.

A ABC `Tombola` especifica dois métodos: `pick` e `load`. Poderíamos também definir um protocolo estático com esses dois métodos, mas aprendi com a comunidade Go que protocolos de apenas um método tornam a tipagem pato estática mais útil e flexível. A biblioteca padrão do Go tem inúmeras interfaces, como `Reader`, uma interface para E/S que requer apenas um método `read`. Depois, se você concluir que um protocolo mais complexo é necessário, pode combinar dois ou mais protocolos para definir um novo.

Usar um componente que escolhe itens aleatoriamente pode ou não exigir o recarregamento do componente, mas ele certamente precisa de um método para sortear um item, então escolhi o método `pick` para o protocolo mínimo `RandomPicker`. O código do protocolo está no Exemplo 19, e seu uso é demonstrado por testes no Exemplo 20.

Exemplo 19. `randompick.py`: definição de `RandomPicker`

```
from typing import Protocol, runtime_checkable, Any

@runtime_checkable
class RandomPicker(Protocol):
    def pick(self) -> Any: ...
```



O método `pick` retorna `Any`. Na Seção 15.8 veremos como tornar `RandomPicker` um tipo genérico, com um parâmetro que permite aos usuários do protocolo especificarem o tipo de retorno do método `pick`.

Exemplo 20. `randompick_test.py`: `RandomPicker` em uso

```
import random
from typing import Any, Iterable, TYPE_CHECKING

from randompick import RandomPicker ①

class SimplePicker: ②
    def __init__(self, items: Iterable) -> None:
        self._items = list(items)
        random.shuffle(self._items)

    def pick(self) -> Any: ③
        return self._items.pop()

def test_isinstance() -> None: ④
    popper: RandomPicker = SimplePicker([1]) ⑤
    assert isinstance(popper, RandomPicker) ⑥
```

```
def test_item_type() -> None: ⑦
    items = [1, 2]
    popper = SimplePicker(items)
    item = popper.pick()
    assert item in items
    if TYPE_CHECKING:
        reveal_type(item) ⑧
    assert isinstance(item, int)
```

- ① Não é necessário importar um protocolo estático para definir uma classe que o implementa; aqui eu importei `RandomPicker` apenas para usá-lo em `test_isinstance` mais tarde.
- ② `SimplePicker` implementa `RandomPicker`, mas não é uma subclasse dele. Isso é a tipagem pato estática em ação.
- ③ `Any` é o tipo de retorno default, então essa anotação não é estritamente necessária, mas deixa mais claro que estamos implementando o protocolo `RandomPicker`, como definido em Exemplo 19.
- ④ Não esqueça de acrescentar dicas `-> None` aos seus testes, se quiser que o Mypy olhe para eles.
- ⑤ Acrescentei uma dica de tipo para a variável `popper`, para mostrar que o Mypy entende que o `SimplePicker` é *consistente-com*.
- ⑥ Teste provando que uma instância de `SimplePicker` também é uma instância de `RandomPicker`. Isso funciona por causa do decorador `@runtime_checkable` aplicado a `RandomPicker`, e porque o `SimplePicker` tem um método `pick`, como exigido.
- ⑦ Este teste invoca o método `pick` de `SimplePicker`, verifica que ele retorna um dos itens dados a `SimplePicker`, e então realiza testes estáticos e de execução sobre o item obtido.
- ⑧ Esta linha gera uma observação no relatório do Mypy.

Como vimos no «Exemplo 22 do Capítulo 8» [fpy.li/92] (vol.1), `reveal_type` é uma função "mágica" reconhecida pelo Mypy. Ela não é importada, e só conseguimos invocá-la de dentro de blocos `if` cuja condição é `typing.TYPE_CHECKING`, uma constante que é sempre considerada `True` pelos checadores de tipos estáticos, mas é `False` durante a execução do programa.

Os dois testes no Exemplo 20 passam. O Mypy também não encontra erro naquele código, e mostra o resultado de `reveal_type` sobre o item retornado por `pick`:

```
$ mypy randompick_test.py
randompick_test.py:24: note: Revealed type is 'Any'
```

Tendo criado nosso primeiro protocolo, vamos estudar algumas recomendações sobre essa prática.

13.6.6. Melhores práticas no desenvolvimento de protocolos

Após 10 anos de experiência com tipagem estática em Go, está claro que protocolos estreitos são mais úteis—muitas vezes tais protocolos têm um único método, raramente mais que um par de métodos. Martin Fowler descreve uma boa ideia para se ter em mente ao desenvolver protocolos: a *Role Interface* [fpy.li/13-33], (interface papel—no sentido de incorporar uma personagem). A ideia é que um protocolo deve ser definido em termos de um papel que um objeto pode desempenhar, e não em termos de uma classe específica.

Além disso, é comum ver um protocolo definido próximo a uma função que o usa para anotar um argumento, em vez de forçar os clientes da função a importar uma definição de interface de alguma biblioteca central. Isso facilita a criação de novos tipos compatíveis com aquela função, favorecendo a extensibilidade e facilitando testes com *mocks* (simulacros).

As duas práticas, protocolos estreitos e protocolos em código cliente, evitam um acoplamento muito forte, em acordo com o Princípio da Segregação de Interface [fpy.li/6v], que podemos resumir como "Clientes não devem ser forçados a depender de interfaces que não usam."

A página *Contributing to typeshed* [fpy.li/13-35] (Colaborando com o typeshed) recomenda a seguinte convenção de nomenclatura para protocolos estáticos:

- Use nomes simples para protocolos que representam um conceito claro (e.g., `Iterator`, `Container`).
- Use `SupportsX` para protocolos que oferecem métodos que podem ser chamados (e.g., `SupportsInt`, `SupportsRead`, `SupportsReadSeek`).^[21]

- Use `HasX` para protocolos que têm atributos de dados que podem ser lidos ou escritos, ou métodos *getter/setter* (e.g., `HasItems`, `HasFileNo`).

A biblioteca padrão do Go tem uma convenção de nomenclatura que eu gosto: para protocolos de método único, se o nome do método é um verbo, acrescente o sufixo adequado (em inglês, "-er" ou "-or", em geral) para torná-lo um substantivo. Por exemplo, em vez de `SupportsRead`, temos `Reader`. Outros exemplos incluem `Formatter`, `Animator`, e `Scanner`. Para se inspirar, veja "Go (Golang) Standard Library Interfaces (Selected)" [fpy.li/13-36] de Asuka Kenji.

Uma boa razão para se criar protocolos minimalistas é que eles servem de base para protocolos mais complexos, quando necessário. Veremos a seguir que não é difícil criar um protocolo derivado com um método adicional.

13.6.7. Estendendo um protocolo

Como mencionei na seção anterior, os desenvolvedores Go defendem que, na dúvida, melhor escolher o minimalismo ao definir interfaces—o nome usado para protocolos estáticos naquela linguagem. Muitas das interfaces Go mais usadas têm um único método.

Quando a prática revela que um protocolo com mais métodos seria útil, em vez de adicionar métodos ao protocolo original, é melhor derivar dali um novo protocolo. Estender um protocolo estático em Python tem algumas ressalvas, como mostra o Exemplo 21.

Exemplo 21. `randompickload.py`: estendendo `RandomPicker`

```
from typing import Protocol, runtime_checkable
from randompick import RandomPicker

@runtime_checkable ①
class LoadableRandomPicker(RandomPicker, Protocol): ②
    def load(self, Iterable) -> None: ... ③
```

- ① Se você quer que o protocolo derivado possa ser checado durante a execução, precisa aplicar o decorador `@runtime_checkable` novamente—pois os comportamentos definidos em decoradores de classes não são herdados.^[22]

- ② Todo protocolo deve nomear explicitamente `typing.Protocol` como uma de suas classes base, além do protocolo que estamos estendendo. Isto é diferente da forma como herança funciona de modo geral.^[23]
- ③ De volta à programação orientada a objetos "normal": só precisamos declarar o método novo no protocolo derivado. A declaração do método `pick` é herdada de `RandomPicker`.

Isto conclui o último exemplo sobre definir e usar um protocolo estático neste capítulo. Para encerrar, vamos olhar as ABCs numéricas e sua possível substituição por protocolos numéricos.

13.6.8. As ABCs de `numbers` e os novos protocolos numéricos

Como vimos na «Seção 8.5.7.1» [fpy.li/8g] (vol.1), as ABCs no pacote `numbers` da biblioteca padrão funcionam bem para checagem de tipos durante a execução.

Se você precisa checar um inteiro, pode usar `isinstance(x, numbers.Integral)` para aceitar `int`, `bool` (que é subclasse de `int`) ou outros tipos inteiros oferecidos por bibliotecas externas que registram seus tipos como subclasses virtuais das ABCs de `numbers`. Por exemplo, a NumPy tem 21 tipos inteiros [fpy.li/13-39]—bem como diversos tipos de ponto flutuante registrados como `numbers.Real`, e números complexos com várias amplitudes de bits, registrados como `numbers.Complex`.



Vale notar que `decimal.Decimal` não é registrado como uma subclasse virtual de `numbers.Real`. A razão é que, se você precisa da precisão de `Decimal`, então vai querer evitar mistura acidental de números decimais com números de ponto flutuante (que são menos precisos).

Infelizmente, a torre numérica não foi projetada para checagem de tipo estática. A ABC raiz—`numbers.Number`—não tem métodos, então se você declarar `x: Number`, o Mypy não vai deixar você fazer operações aritméticas ou chamar qualquer método com `x`.

Quando as ABCs de `numbers` não servem, quais as opções? Um bom lugar para procurar soluções de tipagem é no projeto *typeshed*. Como parte da biblioteca padrão de Python, o módulo `statistics` tem um arquivo `stub` correspondente no *typeshed* com dicas de tipo, o `statistics.pyi` [fpy.li/13-40].

Lá você encontrará as seguintes definições, que são usadas para anotar diversas funções:

```
_Number = Union[float, Decimal, Fraction]
_NumberT = TypeVar('_NumberT', float, Decimal, Fraction)
```

Essa abordagem está correta, mas é limitada. Ela não suporta tipos numéricos fora da biblioteca padrão, que as ABCs de `numbers` suportam durante a execução—quando tipos numéricos são registrados como subclasses virtuais.

A tendência atual é recomendar os protocolos numéricos fornecidos pelo módulo `typing`, como `SupportsFloat`, que discutimos na Seção 13.6.2.

Infelizmente, durante a execução os protocolos numéricos podem deixar você na mão. Como mencionado na Seção 13.6.3, o tipo `complex` no Python 3.9 implementa `__float__`, mas o método existe apenas para lançar uma `TypeError` com uma mensagem explícita: "can't convert complex to float" (não é possível converter `complex` para `float`). Por alguma razão, ele também implementa `__int__`. A presença destes métodos faz `isinstance` produzir resultados enganosos no Python 3.9. No Python 3.10, os métodos de `complex` que geravam `TypeError` incondicionalmente foram removidos.^[24]

Por outro lado, os tipos complexos da NumPy implementam métodos `__float__` e `__int__` que funcionam, emitindo apenas um aviso quando cada um deles é usado pela primeira vez:

```
>>> import numpy as np
>>> cd = np.cdouble(3+4j)
>>> cd
(3+4j)
>>> float(cd)
<stdin>:1: ComplexWarning: Casting complex values to real
discards the imaginary part
3.0
```

O problema oposto também acontece: os tipos embutidos `complex`, `float`, e `int`, bem como `numpy.float16` e `numpy.uint8`, não têm um método `__complex__`, então `isinstance(x, SupportsComplex)` retorna `False` para eles.^[25] Os tipos complexos da

NumPy, tal como `np.complex64`, implementam `__complex__` para conversão em um `complex` embutido.

Entretanto, na prática, o construtor embutido `complex()` trabalha com instâncias de todos esses tipos sem erros ou avisos.

```
>>> import numpy as np
>>> from typing import SupportsComplex
>>> sample = [1+0j, np.complex64(1+0j), 1.0, np.float16(1.0), 1, np.
uint8(1)]
>>> [isinstance(x, SupportsComplex) for x in sample]
[False, True, False, False, False, False]
>>> [complex(x) for x in sample]
[(1+0j), (1+0j), (1+0j), (1+0j), (1+0j), (1+0j)]
```

Isso mostra que checagens de `SupportsComplex` com `isinstance` sugerem que todas aquelas conversões para `complex` falhariam, mas elas funcionam. Na lista de discussão *typing-sig*, Guido van Rossum indicou que o `complex` embutido aceita um único argumento, e por isso as conversões funcionam.

Por outro lado, o `Mypy` aceita argumentos de todos esses seis tipos em uma chamada à função `to_complex()`, definida assim:

```
def to_complex(n: SupportsComplex) -> complex:
    return complex(n)
```

No momento em que escrevo isso, a `NumPy` não tem dicas de tipo, então seus tipos numéricos são todos `Any`.^[26] Por outro lado, o `Mypy` de alguma maneira "sabe" que o `int` e o `float` embutidos podem ser convertidos para `complex`, apesar de, no *typeshed*, apenas a classe embutida `complex` ter o método `__complex__`.^[27]

Concluindo, apesar da expectativa de que a checagem de tipos numéricos não seria difícil, a situação atual é a seguinte: as dicas de tipo da PEP 484 desprezam [fpy.li/cardxvi] a torre numérica e recomendam implicitamente que os checadores de tipos tratem como casos especiais as relações de tipo entre os `complex`, `float`, e `int` embutidos. O `Mypy` faz isso, e também, pragmaticamente, aceita que `int` e `float` são *consistentes-com* `SupportsComplex`, apesar deles não implementarem `__complex__`.



Só encontrei resultados inesperados usando checagens com `isinstance` em conjunto com os protocolos numéricos `Supports*` quando fiz experiências de conversão de `ou` para `complex`. Se você não usa números complexos, pode confiar naqueles protocolos em vez das ABCs de `numbers`.

As principais lições dessa seção são:

- As ABCs de `numbers` são boas para checagem de tipos durante a execução, mas não servem para tipagem estática.
- Os protocolos numéricos estáticos `SupportsComplex`, `SupportsFloat`, etc. funcionam bem para tipagem estática, mas são pouco confiáveis para checagem de tipos durante a execução se números complexos estiverem envolvidos.

Estamos agora prontos para a revisão dos temas deste capítulo.

13.7. Resumo do capítulo

O Mapa de Tipagem (Figura 1) é a chave para entender este capítulo. Após uma breve introdução às quatro abordagens da tipagem, comparamos protocolos dinâmicos e estáticos, que suportam tipagem `pato` e tipagem `pato` estática, respectivamente. Os dois tipos de protocolo compartilham uma característica essencial: nunca é exigido de uma classe que ela declare explicitamente o suporte a qualquer protocolo. Uma classe suporta um protocolo apenas implementando os métodos necessários.

A próxima parada foi a Seção 13.4, onde exploramos os esforços que o interpretador Python faz para que os protocolos dinâmicos de sequência e iterável funcionem, incluindo a implementação parcial de ambos. Então vimos como fazer uma classe implementar um protocolo durante a execução, através da adição de métodos via *monkey patching*. A seção sobre tipagem `pato` terminou com sugestões de programação defensiva, incluindo a detecção de tipos estruturais sem checagens explícitas com `isinstance` ou `hasattr`, usando `try/except` e falhando logo.

Após Alex Martelli introduzir a tipagem `ganso` em *Pássaros aquáticos e as ABCs*, vimos como criar subclasses de ABCs existentes, examinamos algumas ABCs

importantes da biblioteca padrão, e criamos uma ABC do zero, que então implementamos por herança e por registro. Finalizamos aquela seção vendo como o método especial `__subclasshook__` permite que ABCs suportem a tipagem estrutural, pelo reconhecimento de classes não-relacionadas, mas que fornecem os métodos exigidos pela interface declarada na ABC.

Retomamos o estudo da tipagem pato estática na Seção 13.6, que iniciamos na «Seção 8.5.10» [fpy.li/8m] (vol.1). Vimos como o decorador `@runtime_checkable` também aproveita o `__subclasshook__` para suportar tipagem estrutural durante a execução—mesmo que o melhor uso dos protocolos estáticos seja com checadores de tipos estáticos, que podem levar em consideração as dicas de tipo, tornando a tipagem estrutural mais confiável. Então falamos sobre o projeto e a codificação de um protocolo estático e como estendê-lo. O capítulo terminou com a triste história do abandono da torre numérica e das limitações da alternativa proposta: os protocolos numéricos estáticos, tal como `SupportsFloat` e outros adicionados ao módulo `typing` no Python 3.8.

A mensagem principal deste capítulo é que temos quatro maneiras complementares de programar com interfaces no Python moderno, cada uma com diferentes vantagens e deficiências. Você encontrará casos de uso adequados para cada esquema de tipagem em qualquer base de código moderna de tamanho significativo. Rejeitar qualquer destas abordagens tornará seu trabalho como programador Python mais difícil e limitado.

Dito isso, Python ganhou sua enorme popularidade enquanto suportava apenas tipagem pato. Outras linguagens populares que aproveitam o poder e a simplicidade da tipagem pato são JavaScript, PHP e Ruby, e outras, menos populares mas muito influentes, como Lisp, Smalltalk, Erlang, Elixir e Clojure.

13.8. Para saber mais

Para uma rápida revisão dos prós e contras da tipagem, bem como da importância de `typing.Protocol` para a saúde de bases de código checadas estaticamente, recomendo fortemente o post de Glyph Lefkowitz *I Want A New Duck: typing.Protocol and the future of duck typing* [fpy.li/13-42] (Quero um novo pato: `typing.Protocol` e o futuro da tipagem pato). Também aprendi bastante em seu post *Interfaces and Protocols* [fpy.li/13-43], comparando `typing.Protocol` com `zope.interface`—um mecanismo mais antigo para definir interfaces em sistemas

que suportam plug-in fracamente acoplados, usado no *Plone CMS* [fpy.li/13-44], no framework Web na *Pyramid* [fpy.li/13-45], e no framework de programação assíncrona *Twisted* [fpy.li/13-46], um projeto fundado por Glyph.^[28]

Bons livros sobre Python têm—quase que por definição—uma ótima cobertura de tipagem pato. Dois de meus livros favoritos de Python tiveram atualizações lançadas após a primeira edição de *Python Fluente: The Quick Python Book*, 3rd ed., (Manning), de Naomi Ceder; e *Python in a Nutshell*, 3rd ed., de Alex Martelli, Anna Ravenscroft, e Steve Holden (O'Reilly).

Para uma discussão sobre os prós e contras da tipagem dinâmica, veja a entrevista de Guido van Rossum com Bill Venners em *Contracts in Python: A Conversation with Guido van Rossum, Part IV* [fpy.li/13-47] (Contratos em Python: uma conversa com Guido van Rossum). O post *Dynamic Typing* [fpy.li/13-48], de Martin Fowler, traz uma avaliação perspicaz e equilibrada deste debate. Ele também escreveu *Role Interface* [fpy.li/13-33] (interface papel), que mencionei na Seção 13.6.6. Apesar de não ser sobre tipagem pato, aquele post é altamente relevante para o projeto de protocolos em Python, pois ele contrasta as interfaces papel estreitas com as interfaces públicas bem mais abrangentes de classes em geral.

A documentação do Mypy é, muitas vezes, a melhor fonte de informação sobre qualquer tema relacionado a tipagem estática em Python, incluindo à tipagem pato estática, tratada em *Protocols and structural subtyping* [fpy.li/13-50].

As demais referências são sobre tipagem ganso.

O *Python Cookbook*, 3rd ed. de Beazley & Jones (O'Reilly) tem uma seção sobre como definir uma ABC (Recipe 8.12). O livro foi escrito antes de Python 3.4, então eles não usam a atual sintaxe preferida para declarar ABCs, criar uma subclasse de `abc.ABC` (em vez disso, eles usam a palavra-chave `metaclass`, da qual só vamos precisar mesmo no «Capítulo 24» [fpy.li/24] (vol.3)). Tirando esse pequeno detalhe, a receita cobre os principais recursos das ABCs muito bem.

The Python Standard Library by Example de Doug Hellmann (Addison-Wesley), tem um capítulo sobre o módulo `abc`. Ele também está disponível na Web, em *PyMOTW—Python Module of the Week* [fpy.li/13-51]. Hellmann usa a declaração de ABC no estilo antigo: `PluginBase(metaclass=abc.ABCMeta)` em vez de `PluginBase(abc.ABC)`, suportada desde o Python 3.4.

Quando usamos ABCs, herança múltipla não é apenas comum, mas praticamente inevitável, pois cada uma das ABCs fundamentais de coleções (*Sequence*, *Mapping*, *Set*) estende *Collection*, que por sua vez estende múltiplas ABCs (veja Figura 4). Assim, o Capítulo 14 é um complemento importante ao presente capítulo.

A *PEP 3119—Introducing Abstract Base Classes* [fpy.li/13-52] apresenta a justificativa para as ABCs. A *PEP 3141—A Type Hierarchy for Numbers* [fpy.li/13-53] apresenta as ABCs do módulo *numbers* [fpy.li/13-54], mas a discussão no *Mypy issue #3186* [fpy.li/13-55] intitulado "int is not a Number?" (int não é um número?) inclui alguns argumentos sobre por que a torre numérica não serve para checagem estática de tipo. Alex Waygood escreveu uma «resposta abrangente» [fpy.li/13-56] no *StackOverflow*, discutindo formas de anotar tipos numéricos.

Vou continuar monitorando o *Mypy issue #3186* [fpy.li/13-55] para os próximos capítulos dessa saga, na esperança de um final feliz que torne a tipagem estática e a tipagem ganso compatíveis, como deveriam ser.

Ponto de vista

A Jornada MVP da tipagem estática no Python

Trabalhei na Thoughtworks, empresa pioneira em metodologias ágeis de engenharia de software. A Thoughtworks muitas vezes ajuda os clientes a criar e implantar um MVP: *Minimum Viable Product* (Produto Mínimo Viável), "uma versão simples de um produto, oferecida para os usuários com o objetivo de validar hipóteses centrais do negócio," conforme a definição de Paulo Caroli em *Lean Inception* [fpy.li/13-58], um artigo no «blog coletivo» [fpy.li/13-59] editado por Martin Fowler.

Guido van Rossum e os outros mantenedores que projetaram e implementaram a tipagem estática têm seguido a estratégia do MVP desde 2006. Primeiro, a *PEP 3107—Function Annotations* [fpy.li/pep3107] foi implementada no Python 3.0 com uma semântica bastante limitada: apenas uma sintaxe para anexar anotações a parâmetros e resultados de funções, armazenadas no objeto função. Isso foi feito para explicitamente permitir experimentação e receber feedback—os principais benefícios de um MVP.

Oito anos depois, a *PEP 484—Type Hints* [fpy.li/pep484] foi proposta e aprovada. Sua implementação, no Python 3.5, não exigiu mudanças na linguagem ou na biblioteca padrão—exceto a adição do módulo `typing`, do qual nenhuma outra parte da biblioteca padrão dependia.

A PEP 484 suportava apenas tipos nominais com genéricos—similar ao Java—mas com a checagem estática sendo executada por ferramentas externas. Recursos importantes não existiam, como anotações de variáveis, tipos embutidos genéricos, e protocolos.

Apesar destas limitações, este MVP de tipagem foi bem sucedido o suficiente para atrair investimento e adoção por parte de empresas com enormes bases de código em Python, como a Dropbox, o Google e o Facebook, bem como apoio de IDEs profissionais como o PyCharm [fpy.li/13-60], o Wing [fpy.li/13-61], e o VS Code [fpy.li/13-62].

A *PEP 526—Syntax for Variable Annotations* [fpy.li/pep526] foi o primeiro passo evolutivo que exigiu mudanças no interpretador, no Python 3.6. Mais mudanças no interpretador foram feitas na versão 3.7 para suportar a *PEP 563—Postponed Evaluation of Annotations* [fpy.li/pep563] e a *PEP 560—Core support for typing module and generic types* [fpy.li/pep560], que permitiram que coleções embutidas e da biblioteca padrão aceitem dicas de tipo genéricas "de fábrica" no Python 3.9, graças à *PEP 585—Type Hinting Generics In Standard Collections* [fpy.li/pep585].

Durante todos esses anos, alguns usuários de Python—including eu—ficamos desapontados com os tipos estáticos. Após aprender Go, a falta de tipagem pato estática em Python era incompreensível, pois a tipagem pato sempre foi uma característica marcante desta linguagem.

Mas essa é a natureza dos MVPs: eles podem não satisfazer todos os usuários em potencial, mas exigem menos esforço de implementação, e guiam o desenvolvimento posterior com o feedback do uso em situações reais.

Se há uma coisa que todos aprendemos com Python 3, é que progresso incremental é mais seguro que lançamentos estrondosos. Estou contente que não tivemos que esperar pelo Python 4—se é que existirá—para tornar Python mais atrativo para grandes empresas, onde os benefícios da tipagem estática superam a complexidade adicional.

Abordagens à tipagem em linguagens populares

A Figura 7 é uma variação do Mapa de Tipagem (Figura 1) com algumas linguagens conhecidas que suportam cada um dos modos de tipagem.

TypeScript e Python ≥ 3.8 são as únicas linguagens em minha pequena amostra que suportam todas as quatro abordagens.

Go é claramente uma linguagem de tipos estáticos na tradição do Pascal, mas ela foi a pioneira da tipagem pato estática—pelo menos entre as linguagens mais usadas hoje. Também coloquei Go no quadrante da tipagem ganso por causa de sua sintaxe especial para checagem de tipo (*type assertion*), que permite tratar tipos nominais dinamicamente durante a execução.

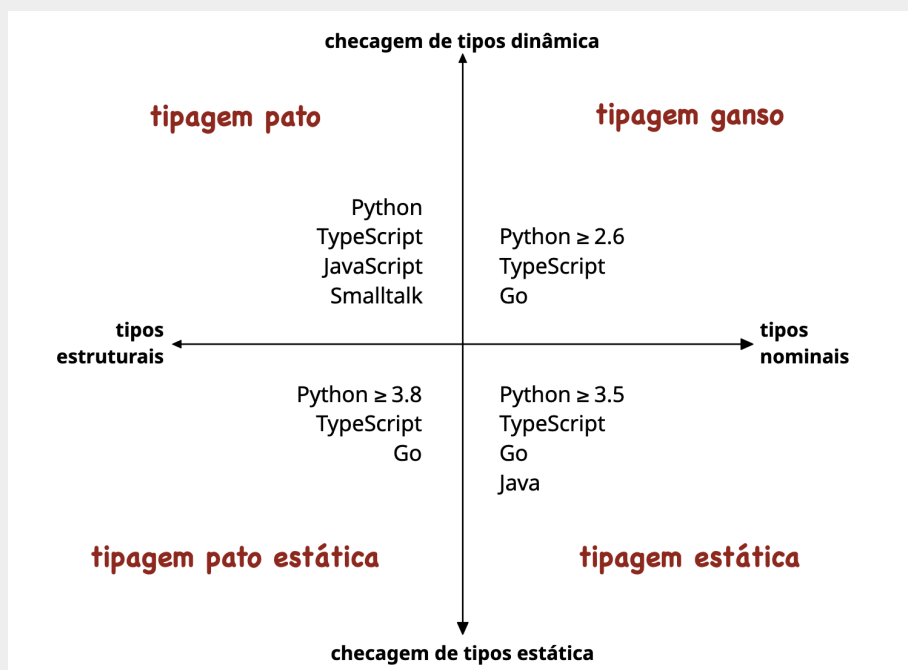


Figura 7. Quatro abordagens para checagem de tipos e algumas linguagens que as usam.

No ano 2000, só existiam linguagens populares nos quadrantes diametralmente opostos da tipagem pato e da tipagem estática. Não conheço nenhuma linguagem que suportava tipagem pato estática ou tipagem ganso 20 anos atrás, mas pode ser que existam. O fato de cada um dos quatro quadrantes ter pelo menos três linguagens populares sugere que muita gente vê benefícios em cada uma das quatro abordagens à tipagem.

Monkey patching

Monkey patching tem uma reputação ruim. Se usado com exagero, pode gerar sistemas difíceis de entender e manter. O remendo (*patch*) dinâmico está normalmente fortemente acoplado ao seu alvo, tornando-se quebradiço quando o código evolui. Outro problema é que duas bibliotecas que aplicam remendos deste tipo durante a execução podem pisar nos pés uma da outra, com a segunda biblioteca a rodar destruindo os remendos da primeira.

Mas o monkey patching pode também ser útil, por exemplo, para fazer uma classe implementar um protocolo durante a execução. O design pattern *Adapter* resolve o mesmo problema de modo mais verboso implementando toda uma nova classe.

É fácil usar monkey patching em código Python, mas há limitações. Ao contrário de Ruby e JavaScript, Python não permite mudar o comportamento dos tipos embutidos durante a execução. Na verdade, considero isto uma vantagem, pois dá a certeza de que um objeto `str` terá sempre os mesmos métodos. Esta limitação reduz a chance de bibliotecas aplicarem correções conflitantes quando importadas em seu projeto.

Metáforas e idiomatismos em interfaces

Uma metáfora promove o entendimento tornando restrições e acessos visíveis. Esse é o valor das palavras *stack* (pilha) e *queue* (fila) para descrever estruturas de dados fundamentais: elas tornam claras as operações permitidas, isto é, como os itens podem ser adicionados ou removidos. Por outro lado, Alan Cooper et al. escrevem em *About Face, the Essentials of Interaction Design*, 4th ed. (Wiley):

Fidelidade estrita a metáforas liga interfaces de forma desnecessariamente firme aos mecanismos do mundo físico.

Os autores estão falando de interface de usuário, mas a advertência se aplica também a APIs. Eles admitem que quando "cai no nosso colo" uma metáfora "verdadeiramente apropriada", podemos usá-la (escreveram "cai no nosso colo" porque é tão difícil encontrar metáforas adequadas que ninguém deveria perder tempo tentando encontrá-las). Acredito que a imagem da máquina de bingo que usei nesse capítulo é apropriada.

About Face é, disparado, o melhor livro sobre design de UI que já li—e eu li uns tantos. Abandonar as metáforas como paradigmas de design, adotando em seu lugar "interfaces idiomáticas", foi a lição mais valiosa que aprendi com o trabalho de Cooper.

Em *About Face*, Cooper não lida com APIs, mas quanto mais penso em suas ideias, mais vejo como se aplicam ao Python. Cada protocolo fundamental da linguagem é o que Cooper chama de *idiom* (idiomatismo).^[29]

Uma vez que aprendemos o que é uma "sequência", podemos aplicar este conhecimento em diferentes contextos. Este é o tema principal de *Python Fluente*: ressaltar os idiomatismos fundamentais da linguagem, para que o seu código seja conciso, eficaz e legível para um pythonista fluente.

- [1] Design Patterns: Elements of Reusable Object-Oriented Software, Introduction, p. 18.
- [2] O artigo "Monkey patch" [fpy.li/13-4] na Wikipedia tem um exemplo engraçado em Python.
- [3] Por isso a necessidade de testes automatizados.
- [4] A pioneira da computação Grace Hopper dizia que, para inovar em uma burocracia, é mais fácil pedir perdão do que permissão ("*It's Easier to Ask Forgiveness than Permission*" ou *EAFP*).
- [5] No original: "An abstract class represents an interface", Bjarne Stroustrup, *The Design and Evolution of C++* (Addison-Wesley, 1994), p. 278.
- [6] NT: O exemplo citado por Martelli é intraduzível. Ele joga com três significados diferentes do verbo "to draw": artista desenha; o pistoleiro saca (a arma); a loteria sorteia um número.
- [7] Você também pode, claro, definir suas próprias ABCs—mas eu não recomendaria esse caminho a ninguém, exceto aos mais avançados pythonistas, da mesma forma que os desencorajaria de definir suas próprias metaclasses customizadas... e mesmo para os ditos "mais avançados pythonistas", aqueles que exibem o domínio de todos os recantos por mais obscuros da linguagem, essas não são ferramentas de uso frequente. Este tipo de "metaprogramação profunda", se alguma vez for apropriada, o será no contexto dos autores de frameworks abrangentes, projetados para serem estendidos de forma independente por inúmeras equipes de desenvolvimento diferentes... menos que 1% dos "mais avançados pythonistas" precisará disso alguma vez na vida!—A.M
- [8] NT: Outro exemplo intraduzível. A frase "class struggle" é uma referência bem humorada ao conceito marxista da "luta de classes".
- [9] Herança múltipla foi *considerada nociva* e excluída do Java, exceto para interfaces: Interfaces Java podem estender múltiplas interfaces, e classes Java podem implementar múltiplas interfaces.
- [10] Talvez o cliente precise auditar o randomizador ou a agência queira fornecer um randomizador "viciado". Nunca se sabe...
- [11] Antes das ABCs existirem, métodos abstratos levantariam um `NotImplementedError` para sinalizar que as subclasses eram responsáveis por suas implementações. No Smalltalk-80, o corpo dos métodos abstratos invocaria `subclassResponsibility`, um método herdado de `object` que gerava um erro com a mensagem "Minha subclasse deveria ter sobrescrito uma de minhas mensagens."
- [12] O verbete `@abc.abstractmethod` [fpy.li/6s] na documentação do módulo `abc` [docs.python.org/pt-br/dev/library/abc.html].
- [13] A «Seção 6.5.2» [fpy.li/8z] (vol.1) trata do problema de apelidamento que acabamos de evitar aqui.
- [14] O truque usado com `load()` não funciona com `loaded()`, pois o tipo `list` não implementa `__bool__`, o método que eu teria de vincular a `loaded`. O `bool()` nativo não precisa de `__bool__` para funcionar, porque pode também usar `__len__`. Veja 4.1. Teste do Valor Verdade [fpy.li/2g] na documentação de Python.
- [15] Há toda uma explicação sobre o atributo de classe `__mro__` na Seção 14.4. Por agora, essas informações básicas são o suficiente.
- [16] O conceito de consistência de tipo é explicado na «Seção 8.5.1.1» [fpy.li/83] (vol.1).

[17] Concorde que `double()` não é muito útil, exceto como um exemplo. Mas a biblioteca padrão de Python tem muitas funções que não poderiam ser anotadas de modo apropriado antes dos protocolos estáticos serem adicionados, no Python 3.8. Ajudei a corrigir alguns bugs no *typeshed* acrescentando dicas de tipo com protocolos. Por exemplo, no *pull request* onde consertei *Should Mypy warn about potential invalid arguments to max?* [fpy.li/shed4051] (Deveria o Mypy avisar sobre argumentos potencialmente inválidos passados a `max`?) defini um protocolo `_SupportsLessThan`, que usei para melhorar as anotações de `max`, `min`, `sorted`, e `list.sort`.

[18] O atributo `__slots__` é irrelevante para nossa discussão aqui—é uma otimização sobre a qual falamos na Seção 11.11.

[19] Agradeço a Ivan Levkivskiy, co-autor da «PEP 544 (sobre protocolos)» [fpy.li/pep544], por apontar que checagem de tipo não é apenas uma questão de checar se o tipo de `x` é `T`: é sobre determinar que o tipo de `x` é *consistente-com* `T`, o que pode ser custoso. Não é de se espantar que o Mypy leve alguns segundos para fazer uma checagem de tipos, mesmo em scripts Python curtos.

[20] Leia a decisão [fpy.li/13-32] do Python Steering Council na lista *python-dev*.

[21] Qualquer método pode ser chamado, então essa recomendação não diz muito. Talvez "forneça um ou dois métodos"? De qualquer forma, é uma recomendação, não uma regra absoluta.

[22] Para detalhes e justificativa, veja a seção sobre `@runtime_checkable` [fpy.li/13-37] na PEP 544—Protocols: Structural subtyping (static duck typing).

[23] Novamente, leia *Merging and extending protocols* [fpy.li/13-38] na PEP 544 para os detalhes e justificativa.

[24] ver Issue #41974—Remove `complex.__float__`, `complex.__floordiv__`, etc [fpy.li/13-41].

[25] Eu não testei todas as outras variantes de *float* e *integer* que a NumPy oferece.

[26] Os tipos numéricos da NumPy são todos registrados com as ABCs apropriadas de `numbers`, que o Mypy ignora.

[27] Isso é uma mentira bem intencionada da parte do *typeshed*: a partir de Python 3.9, o tipo embutido `complex` na verdade não tem mais um método `__complex__`.

[28] Agradeço ao revisor técnico Jürgen Gmach por ter recomendado o post *Interfaces and Protocols*.

[29] Neste contexto, a tradução correta de *idiom* não é "idioma", mas sim "idiomatismo" que é uma "construção ou locução peculiar a uma língua". Cooper defende que uma GUI é uma linguagem com locuções peculiares, como menus e caixas de diálogo.

Capítulo 14. Herança: para o bem ou para o mal

[...] precisávamos de toda uma teoria melhor sobre herança (e ainda precisamos). Por exemplo, herança e instanciação (que é um tipo de herança) confundem a pragmática (fatorar o código para economizar espaço) quanto a semântica (usada para tarefas demais, como: especialização, generalização, especiação, etc.).^[1]

— Alan Kay, Os Primórdios de Smalltalk

Este capítulo é sobre herança e criação de subclasses. Vou presumir um entendimento básico destes conceitos, que você pode ter aprendido lendo «O Tutorial do Python» [fpy.li/6w], ou trabalhando com outra linguagem orientada a objetos, tal como Java, C# ou C++. Vamos nos concentrar em quatro características de Python:

- A função `super()`
- Armadilhas na criação de subclasses de tipos embutidos
- Herança múltipla e a ordem de resolução de métodos
- Classes `mixins`

Herança múltipla acontece quando uma classe tem duas ou mais superclasses. Ela existe em C++, mas não em Java ou C#. Muitos consideram que a herança múltipla não vale os problemas que causa. Ela foi deliberadamente deixada de fora de Java, após supostamente ser usada em excesso nos primeiros anos de C++.

Este capítulo apresenta a herança múltipla para aqueles que nunca a usaram, e oferece dicas para lidar com herança simples ou múltipla, quando necessário.

Em 2021, quando escrevo essas linhas, há uma forte reação contra o uso excessivo de herança em geral—não apenas herança múltipla—porque superclasses e subclasses são fortemente acopladas, ou seja, interdependentes. Esse acoplamento forte significa que modificações em uma classe podem ter efeitos inesperados e de longo alcance em suas subclasses, tornando os sistemas frágeis e difíceis de entender.

Entretanto, ainda temos que dar manutenção a sistemas existentes, que podem ter hierarquias de classe complexas, ou trabalhar com frameworks que nos obrigam a usar herança—algumas vezes até herança múltipla.

Vou ilustrar as aplicações práticas da herança múltipla com a biblioteca padrão, o framework Django e o toolkit para programação de interface gráfica Tkinter.

14.1. Novidades neste capítulo

Não há nenhum recurso novo no Python relacionado ao tema deste capítulo, mas fiz inúmeras modificações baseadas nos comentários dos revisores técnicos da segunda edição, especialmente Leonardo Rocha e Caleb Hattingh.

Escrevi uma nova seção de abertura, tratando especificamente da função embutida `super()`, e mudei os exemplos na Seção 14.4, para explorar mais profundamente a forma como `super()` suporta a herança múltipla cooperativa.

A Seção 14.5 também é nova. Reorganizei a Seção 14.6, apresentando exemplos mais simples de *mixin* na biblioteca padrão, antes de apresentar o exemplos com o Django e a hierarquia complicada do Tkinter.

Como o próprio título sugere, as ressalvas à herança sempre foram um dos temas principais desse capítulo. Mas como cada vez mais desenvolvedores consideram essa técnica problemática, acrescentei alguns parágrafos sobre como evitar a herança no final da Seção 14.8 e da Seção 14.9.

Vamos começar com uma revisão da mal compreendida função `super()`.

14.2. A função `super()`

O uso consistente da função embutida `super()` é essencial na criação de programas orientados a objetos fáceis de manter em Python.

Quando uma subclasse sobrescreve um método de uma superclasse, o novo método normalmente precisa invocar o método correspondente na superclasse. Aqui está o modo recomendado de fazer isso, tirado de um exemplo da documentação do módulo *collections*, na seção `OrderedDict: Exemplos e Receitas` [fpy.li/6x].^[2]


```
class LastUpdatedOrderedDict(OrderedDict):
    """Armazena itens mantendo por ordem de atualização."""

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

Para executar sua tarefa, `LastUpdatedOrderedDict` sobrescreve `__setitem__` para:

1. Usar `super().__setitem__`, invocando aquele método na superclasse e permitindo que ele insira ou atualize o par chave/valor.
2. Invocar `self.move_to_end`, para garantir que a key atualizada esteja na última posição.

Invocar um `__init__` herdado é particularmente importante, para permitir que a superclasse execute sua parte na inicialização da instância.



Se você aprendeu programação orientada a objetos com Java, deve se lembrar de que, naquela linguagem, um método construtor invoca automaticamente o construtor sem argumentos da superclasse. Python não faz isso. Acostume-se a escrever o seguinte código padrão:

```
def __init__(self, a, b) :
    super().__init__(a, b)
    ... # mais código para inicializar a instância
```

Você pode já ter visto código que não usa `super()`, e em vez disso invoca o método na superclasse diretamente, assim:

```
class NotRecommended(OrderedDict):
    """Isto é um contra-exemplo!"""

    def __setitem__(self, key, value):
        OrderedDict.__setitem__(self, key, value)
        self.move_to_end(key)
```

Esta alternativa até funciona nesse caso em particular, mas não é recomendada por duas razões. Primeiro, codifica a superclasse explicitamente. O nome `OrderedDict` aparece na declaração `class` e também dentro de `__setitem__`. Se, no futuro, alguém modificar a declaração `class` para mudar a classe base ou adicionar outra, pode se esquecer de atualizar o corpo de `__setitem__`, introduzindo um bug.

A segunda razão é que `super` implementa lógica para tratar hierarquias de classe com herança múltipla. Voltaremos a isso na Seção 14.4. Para concluir essa recapitulação de `super`, é bom rever como essa função era invocada no Python 2. Sem os parâmetros default, a assinatura de `super` é mais reveladora:

```
class LastUpdatedOrderedDict(OrderedDict):
    """Funciona igual em Python 2 e Python 3"""

    def __setitem__(self, key, value):
        super(LastUpdatedOrderedDict, self).__setitem__(key, value)
        self.move_to_end(key)
```

Os dois parâmetros de `super` agora são opcionais. O compilador de bytecode de Python 3 fornece os argumentos examinando o contexto quando `super()` é invocado dentro de um método. Os parâmetros são:

type

O início do caminho para a superclasse que implementa o método desejado. Por default, é a classe onde está o método que invoca `super()`.

object_or_type

O objeto (ao invocar métodos de instância) ou classe (ao invocar métodos de classe) que será o receptor da chamada ao método.^[3] Por default, é `self` se a chamada `super()` acontece no corpo de um método de instância.

A chamada a `super()` devolve um objeto proxy dinâmico que encontra um método (tal como `__setitem__` no exemplo) em uma superclasse do argumento `type` e o vincula a `object_or_type`, de modo que não precisamos passar explicitamente o receptor (`self`) quando invocamos o método.

No Python 3, ainda é permitido passar explicitamente o primeiro e o segundo argumentos a `super()`.^[4] Mas eles são necessários apenas em casos especiais, para testes, ou depuração, ou para contornar algum comportamento indesejado em uma superclasse.

Vamos agora discutir armadilhas na criação de subclasses de tipos embutidos.

14.3. Problemas com subclasses de tipos embutidos

Nas primeiras versões do Python não era possível criar subclasses de tipos embutidos como `list` ou `dict`. Desde o Python 2.2 isso é possível, mas há uma limitação importante: o código em C dos tipos embutidos normalmente não invoca os métodos sobrescritos por classes definidas pelo usuário. Há uma boa descrição curta do problema na documentação do PyPy, na seção *Differences between PyPy and CPython* (Diferenças entre o PyPy e o CPython), em *Subclasses of built-in types* [fpy.li/pypydif] (Subclasses de tipos embutidos):

Oficialmente, o CPython não tem nenhuma regra sobre exatamente quando um método sobrescrito de subclasses de tipos embutidos é ou não invocado implicitamente. Como uma aproximação, esses métodos nunca são chamados por outros métodos embutidos do mesmo objeto. Por exemplo, um `__getitem__` sobrescrito em uma subclasse de `dict` nunca será invocado pelo método `get()` do tipo embutido.

Concretamente, isto significa que `meu_dict['x']` e `meu_dict.get('x')` podem produzir resultados diferentes, mesmo no caso mais simples quando a chave 'x' existe, supondo que `meu_dict` é uma instância de uma subclasse de `dict` criada por você.

O Exemplo 1 ilustra o problema.

Exemplo 1. Nosso `__setitem__` sobrescrito é ignorado pelos métodos `__init__` e `__update__` to tipo embutido dict

```
>>> class DoppelDict(dict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2) ①
...
>>> dd = DoppelDict(one=1) ②
>>> dd
{'one': 1}
>>> dd['two'] = 2 ③
>>> dd
{'one': 1, 'two': [2, 2]}
>>> dd.update(three=3) ④
>>> dd
{'three': 3, 'one': 1, 'two': [2, 2]}
```

- ① `DoppelDict.__setitem__` duplica os valores ao armazená-los (por nenhuma razão, apenas para termos um efeito visível). Ele funciona delegando para a superclasse.
- ② O método `__init__`, herdado de `dict`, claramente ignora que `__setitem__` foi sobrescrito: o valor de 'one' não foi duplicado.
- ③ O operador `[]` invoca nosso `__setitem__` e funciona como esperado: 'two' está mapeado para o valor duplicado `[2, 2]`.
- ④ O método `update` de `dict` também não usa nossa versão de `__setitem__`: o valor de 'three' não foi duplicado.

Este comportamento dos tipos embutidos viola uma regra básica da programação orientada a objetos: a busca por métodos deveria sempre começar pela classe do receptor (`self`), mesmo quando a invocação ocorre dentro de um método implementado na superclasse. Isso é o que se chama *late binding* (vinculação tardia), que Alan Kay—um dos criadores de Smalltalk—considera ser uma característica essencial da programação orientada a objetos: em qualquer chamada na forma `x.method()`, o método exato a ser chamado deve ser determinado durante a execução, baseado na classe do receptor `x`.^[5] Este triste estado de coisas contribui para os problemas que vimos na «Seção 3.5.3» [fpy.li/88] (vol.1).

O problema não está limitado a chamadas dentro de uma instância—saber se `self.get()` invoca `self.__getitem__()`. Também acontece com métodos sobrescritos de outras classes que deveriam ser chamados por métodos embutidos. O Exemplo 2 foi adaptado da documentação do PyPy [fpy.li/14-5].

Exemplo 2. O `__getitem__` de `AnswerDict` é ignorado por `dict.update`

```
>>> class AnswerDict(dict):
...     def __getitem__(self, key): ①
...         return 42
...
>>> ad = AnswerDict(a='foo') ②
>>> ad['a'] ③
42
>>> d = {}
>>> d.update(ad) ④
>>> d['a'] ⑤
'foo'
>>> d
{'a': 'foo'}
```

- ① `AnswerDict.__getitem__` sempre devolve 42, independente da chave.
- ② `ad` é um `AnswerDict` carregado com o par chave-valor ('a', 'foo').
- ③ `ad['a']` devolve 42, como esperado.
- ④ `d` é uma instância direta de `dict`, que atualizamos com `ad`.
- ⑤ O método `dict.update` ignora nosso `AnswerDict.__getitem__`.



Criar subclasses diretamente de tipos embutidos, como `dict`, `list` ou `str`, é um processo propenso ao erro, pois os métodos embutidos quase sempre ignoram métodos sobrescritos pelo usuário. Em vez de criar subclasses de tipos embutidos, derive suas classes do módulo `collections` [fpy.li/2w], usando as classes `UserDict`, `UserList`, e `UserString`, que foram projetadas para serem fáceis de estender.

Herdando de `collections.UserDict` em vez de `dict`, os problemas expostos no Exemplo 1 e no Exemplo 2 desaparecem. Veja o Exemplo 3.

Exemplo 3. DoppelDict2 e AnswerDict2 funcionam como esperado, porque estendem UserDict e não dict

```
>>> import collections
>>>
>>> class DoppelDict2(collections.UserDict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2)
...
>>> dd = DoppelDict2(one=1)
>>> dd
{'one': [1, 1]}
>>> dd['two'] = 2
>>> dd
{'two': [2, 2], 'one': [1, 1]}
>>> dd.update(three=3)
>>> dd
{'two': [2, 2], 'three': [3, 3], 'one': [1, 1]}
>>>
>>> class AnswerDict2(collections.UserDict):
...     def __getitem__(self, key):
...         return 42
...
>>> ad = AnswerDict2(a='foo')
>>> ad['a']
42
>>> d = {}
>>> d.update(ad)
>>> d['a']
42
>>> d
{'a': 42}
```

Como um experimento, para medir o trabalho extra necessário para criar uma subclasse de um tipo embutido, reescrevi a classe StrKeyDict do «Exemplo 9 do Capítulo 3» [fpy.li/93] (vol.1), para torná-la uma subclasse de dict em vez de UserDict. Para fazê-la passar pelo mesmo banco de testes, tive que implementar `__init__`, `get`, e `update`, pois as versões herdadas de dict se recusaram a cooperar com os métodos sobrescritos `__missing__`, `__contains__` e `__setitem__`. A subclasse de UserDict no «Exemplo 9 do Capítulo 3» [fpy.li/93] (vol.1) tem 16 linhas, enquanto a subclasse experimental de dict acabou com 33 linhas.^[6]

Para deixar claro: esta seção tratou de um problema que se aplica apenas à delegação a métodos dentro do código em C dos tipos embutidos, e afeta apenas classes derivadas diretamente daqueles tipos. Se você criar uma subclasse de uma classe escrita em Python, tal como `UserDict` ou `MutableMapping`, não vai encontrar este problema.^[7]

Vamos agora examinar uma questão que aparece na herança múltipla: se uma classe tem duas superclasses, como Python decide qual atributo usar quando invocamos `super().attr`, mas ambas as superclasses têm um atributo com este nome?

14.4. Herança múltipla e a Ordem de Resolução de Métodos

Qualquer linguagem que implemente herança múltipla precisa lidar com o potencial conflito de nomes, quando superclasses contêm métodos com nomes iguais. Este é o chamado "problema do losango" (*diamond problem*), ilustrado na Figura 1 e no Exemplo 4, onde da hierarquia começa na classe base *Root* (raiz) e termina na classe *Leaf* (folha).^[8]

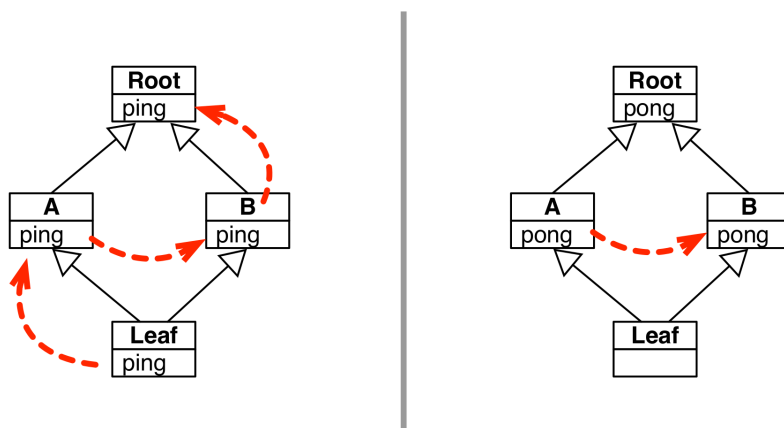


Figura 1. Esquerda: Sequência de ativação para a chamada `leaf1.ping()`. Direita: Sequência de ativação para a chamada `leaf1.pong()`.

Exemplo 4. diamond.py: classes Leaf, A, B, Root formam o grafo na Figura 1

```
class Root: ①
    def ping(self):
        print(f'{self}.ping() in Root')

    def pong(self):
        print(f'{self}.pong() in Root')

    def __repr__(self):
        cls_name = type(self).__name__
        return f'<instance of {cls_name}>'

class A(Root): ②
    def ping(self):
        print(f'{self}.ping() in A')
        super().ping()

    def pong(self):
        print(f'{self}.pong() in A')
        super().pong()

class B(Root): ③
    def ping(self):
        print(f'{self}.ping() in B')
        super().ping()

    def pong(self):
        print(f'{self}.pong() in B')

class Leaf(A, B): ④
    def ping(self):
        print(f'{self}.ping() in Leaf')
        super().ping()
```

① Root fornece ping, pong, e `__repr__` (para facilitar a leitura da saída).

② Os métodos ping e pong na classe A chamam `super()`.

- ③ Apenas o método ping na classe B invoca super().
- ④ A classe Leaf implementa apenas ping, e invoca super().

Vejamos agora o efeito da invocação dos métodos ping e pong em uma instância de Leaf (Exemplo 5).

Exemplo 5. Doctests para chamadas a ping e pong em um objeto Leaf

```
>>> leaf1 = Leaf() ①
>>> leaf1.ping() ②
<instance of Leaf>.ping() in Leaf
<instance of Leaf>.ping() in A
<instance of Leaf>.ping() in B
<instance of Leaf>.ping() in Root

>>> leaf1.pong() ③
<instance of Leaf>.pong() in A
<instance of Leaf>.pong() in B
```

- ① leaf1 é uma instância de Leaf.
- ② Chamar leaf1.ping() ativa os métodos ping em Leaf, A, B, e Root, porque os métodos ping nas três primeiras classes chamam super().ping().
- ③ Chamar leaf1.pong() ativa pong em A através da herança, que por sua vez invoca super.pong(), ativando B.pong.

As sequências de ativação que aparecem no Exemplo 5 e na Figura 1 são determinadas por dois fatores:

- A ordem de resolução de métodos da classe Leaf.
- O uso de super() em cada método.

A ordem de resolução de métodos é conhecida pela sigla MRO (*Method Resolution Order*). Em Python, todas as classes têm um atributo chamado `__mro__`, que armazena uma tupla de referências a superclasses, na ordem de resolução dos métodos, indo desde a classe corrente até a classe object.^[9]

Para a classe Leaf, o `__mro__` é o seguinte:

```
>>> Leaf.__mro__ # doctest:+NORMALIZE_WHITESPACE
(<class 'diamond1.Leaf'>, <class 'diamond1.A'>, <class 'diamond1.B'>,
 <class 'diamond1.Root'>, <class 'object'>)
```



Na Figura 1, pode parecer que a MRO descreve uma busca em largura [fpy.li/6y], mas isso é apenas uma coincidência para esta hierarquia de classes simples. A MRO é computada por um algoritmo da literatura de computação, chamado C3. Seu uso no Python está detalhado no artigo *The Python 2.3 Method Resolution Order* [fpy.li/14-10] (A Ordem de Resolução de Métodos no Python 2.3), de Michele Simionato. É um texto difícil, mas Simionato escreve: "...a menos que você use herança múltipla intensivamente, e mantenha hierarquias não-triviais, não é necessário entender o algoritmo C3, e você pode facilmente ignorar este artigo."

A MRO determina apenas a ordem de ativação, mas se um método específico será ou não ativado em cada uma das classes vai depender de cada implementação chamar ou não `super()`.

Considere o experimento com o método `pong`. A classe `Leaf` não sobrescreve aquele método, então a chamada `leaf1.pong()` ativa a implementação na próxima classe listada em `Leaf.__mro__`: a classe `A`. O método `A.pong` invoca `super().pong()`. A classe `B` class é e próxima na MRO, portanto `B.pong` é ativado. Mas aquele método não invoca `super().pong()`, então a sequência de ativação termina ali.

Além do grafo de herança, a MRO também considera a ordem na qual as superclasses aparecem na declaração da uma subclasse. Considerando o programa *diamond.py* (no Exemplo 4), se a classe `Leaf` fosse declarada como `Leaf(B, A)`, daí a classe `B` apareceria antes de `A` em `Leaf.__mro__`. Isso afetaria a ordem de ativação dos métodos `ping`, e também faria `leaf1.pong()` ativar `B.pong` através da herança, mas `A.pong` e `Root.pong` não seriam invocados, porque `B.pong` não invoca `super()`.

Quando um método invoca `super()`, ele é um método cooperativo. Métodos cooperativos permitem a herança múltipla cooperativa. Esses termos são

intencionais: para funcionar, a herança múltipla no Python exige a cooperação ativa dos métodos envolvidos invocando `super()`. Na classe B, ping coopera, mas pong não.



Um método não-cooperativo pode ser a causa de bugs sutis. Muitos programadores, lendo o Exemplo 4, poderiam esperar que, quando o método A.pong invoca `super().pong()`, isso acabaria por ativar `Root.pong`. Mas se B.pong for ativado antes, ele deixa a bola cair. Por isso, recomenda-se que um método subscrito m de uma classe não-base invoque `super().m()`.

Métodos cooperativos devem ter assinaturas compatíveis, porque nunca se sabe se A.ping será chamado antes ou depois de B.ping. A sequência de ativação depende da ordem de A e B na declaração de cada subclasse que herda de ambos.

Python é uma linguagem dinâmica, então a interação de `super()` com a MRO também é dinâmica. O Exemplo 6 mostra um resultado surpreendente desse comportamento dinâmico.

Exemplo 6. diamond2.py: classes para demonstrar a natureza dinâmica de `super()`

```
from diamond import A ①

class U(): ②
    def ping(self):
        print(f'{self}.ping() in U')
        super().ping() ③

class LeafUA(U, A): ④
    def ping(self):
        print(f'{self}.ping() in LeafUA')
        super().ping()
```

- ① A classe A vem de *diamond.py* (no Exemplo 4).
- ② A classe U não tem relação com A ou Root do módulo diamond.
- ③ O que `super().ping()` faz? Resposta: depende. Continue lendo.
- ④ LeafUA é subclasse de U e A, nessa ordem.

Se você criar uma instância de `U` e tentar chamar `ping`, ocorre um erro:

```
>>> u = U()
>>> u.ping()
Traceback (most recent call last):
...
AttributeError: 'super' object has no attribute 'ping'
```

O objeto `'super'` devolvido por `super()` não tem um atributo `'ping'`, porque a MRO de `U` tem duas classes: `U` e `object`, e esta última não tem um atributo chamado `'ping'`.

Entretanto, o método `U.ping` não é completamente inútil. Veja isso:

```
>>> leaf2 = LeafUA()
>>> leaf2.ping()
<instance of LeafUA>.ping() in LeafUA
<instance of LeafUA>.ping() in U
<instance of LeafUA>.ping() in A
<instance of LeafUA>.ping() in Root
>>> LeafUA.__mro__ # doctest:+NORMALIZE_WHITESPACE
(<class 'diamond2.LeafUA'>, <class 'diamond2.U'>,
 <class 'diamond.A'>, <class 'diamond.Root'>, <class 'object'>)
```

A chamada `super().ping()` em `LeafUA` ativa `U.ping`, que também coopera chamando `super().ping()`, ativando `A.ping` e, por fim, `Root.ping`.

Observe que as classes base de `LeafUA` são (`U`, `A`), nesta ordem. Se em vez disso as bases fossem (`A`, `U`), daí `leaf2.ping()` nunca chegaria a `U.ping`, porque o `super().ping()` em `A.ping` ativaria `Root.ping`, e esse último não invoca `super()`.

Em um programa real, uma classe como `U` poderia ser uma classe *mixin*: uma classe projetada para ser usada ao lado outras classes em herança múltipla, fornecendo funcionalidade adicional. Vamos estudar *mixins* na Seção 14.5.

Para concluir essa discussão sobre a MRO, a Figura 2 ilustra parte do complexo grafo de herança múltipla do toolkit de interface gráfica Tkinter, da biblioteca padrão de Python.

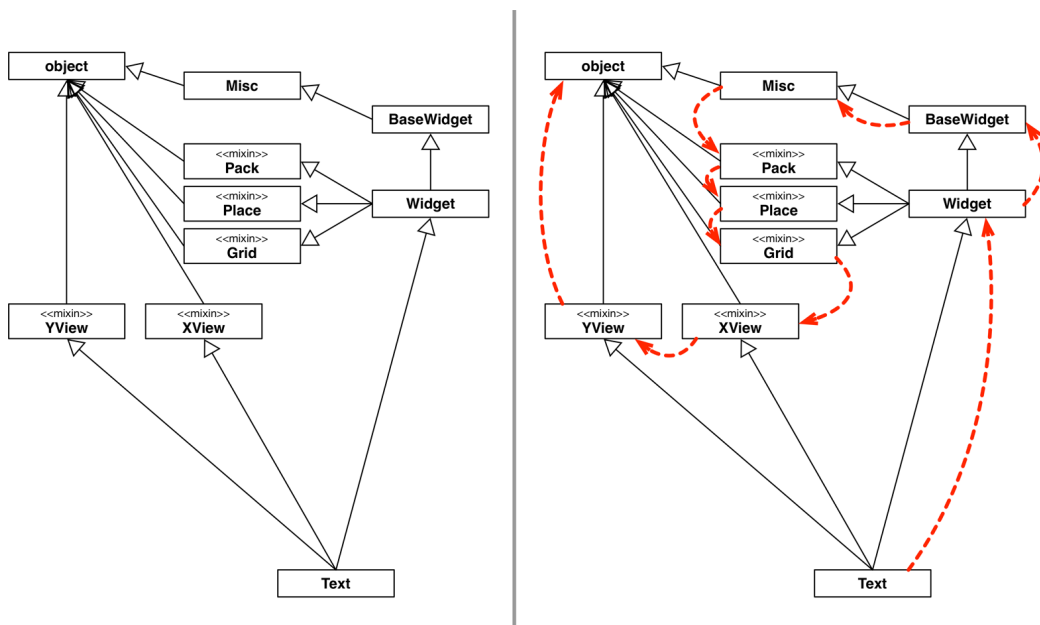


Figura 2. Esquerda: diagrama UML da classe e das superclasses do componente Text do Tkinter. Direita: O longo e sinuoso caminho de Text.__mro__, desenhado com as setas pontilhadas.

Para estudar a figura, comece pela classe Text, na parte inferior. A classe Text implementa um componente de texto completo, editável e com múltiplas linhas. Ele sozinho fornece muita funcionalidade, mas também herda muitos métodos de outras classes. A imagem à esquerda mostra um diagrama de classe UML simples. À direita, a mesma imagem é decorada com setas mostrando a MRO, como listada no Exemplo 7 com a ajuda de uma função de conveniência print_mro.

Exemplo 7. MRO de tkinter.Text

```
>>> def print_mro(cls):
...     print(' ', '.join(c.__name__ for c in cls.__mro__))
>>> import tkinter
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```

Vamos agora falar sobre mixins.

14.5. Classes *mixin*

Uma *classe mixin* é feita para ser herdada com pelo menos uma outra classe, em um arranjo de herança múltipla. Uma *mixin* não é feita para ser a única classe base de uma classe concreta, pois não fornece toda a funcionalidade para um objeto concreto, apenas adicionando ou customizando o comportamento de classes filhas ou irmãs.



Classes *mixin* são uma convenção sem qualquer suporte explícito no Python e no C++. Ruby permite a definição explícita e o uso de módulos que funcionam como *mixins*—coleções de métodos que podem ser incluídas para adicionar funcionalidade a uma classe. C#, PHP, e Rust implementam *traits* (*traços* ou *aspectos*), que são também uma forma explícita de *mixin*.

Vamos ver um exemplo simples e útil de uma classe *mixin*.

14.5.1. Mapeamentos maiúsculos

O Exemplo 8 mostra a `UpperCaseMixin`, uma classe criada para fornecer acesso indiferente a maiúsculas/minúsculas para mapeamentos com chaves do tipo `string`, convertendo todas as chaves para maiúsculas quando elas são adicionadas ou consultadas.

Exemplo 8. uppermixin.py: UpperCaseMixin suporta mapeamentos indiferentes a maiúsculas/minúsculas

```
import collections

def _upper(key): ①
    try:
        return key.upper()
    except AttributeError:
        return key

class UpperCaseMixin: ②
    def __setitem__(self, key, item):
        super().__setitem__(_upper(key), item)
```

```

def __getitem__(self, key):
    return super().__getitem__(_upper(key))

def get(self, key, default=None):
    return super().get(_upper(key), default)

def __contains__(self, key):
    return super().__contains__(_upper(key))

```

- ① Esta função auxiliar recebe uma *key* de qualquer tipo e tenta devolver `key.upper()`; se isto falha, devolve a *key* inalterada.
- ② A *mixin* implementa quatro métodos essenciais de mapeamentos, sempre chamando `super()` após tentar converter a chave em maiúsculas.

Como todos os métodos de `UpperCaseMixin` chamam `super()`, esta *mixin* depende de uma classe irmã que implemente ou herde métodos com a mesma assinatura. Para dar sua contribuição, uma *mixin* normalmente precisa aparecer antes de outras classes na MRO de uma subclasse. Na prática, isto significa que *mixins* devem aparecer primeiro na tupla de classes base em uma declaração de classe. O Exemplo 9 apresenta dois exemplos.

Exemplo 9. uppermixin.py: duas classes que usam UpperCaseMixin

```

class UpperDict(UpperCaseMixin, collections.UserDict): ①
    pass

class UpperCounter(UpperCaseMixin, collections.Counter): ②
    """Specialized 'Counter' that uppercases string keys""" ③

```

- ① `UpperDict` não precisa implementar nenhum método, mas `UpperCaseMixin` tem ser a primeira classe base, caso contrário os métodos chamados seriam os de `UserDict`.
- ② `UpperCaseMixin` também funciona com `Counter`.
- ③ Em vez de `pass`, é melhor fornecer uma *docstring* para satisfazer a sintaxe da instrução `class`, que precisa ter um corpo.

Aqui estão alguns doctests de `UpperDict`, do módulo *uppermix.py* [fpy.li/14-11]:

```
>>> d = UpperDict([('a', 'letter A'), (2, 'digit two')])
>>> list(d.keys())
['A', 2]
>>> d['b'] = 'letter B'
>>> 'b' in d
True
>>> d['a'], d.get('B')
('letter A', 'letter B')
>>> list(d.keys())
['A', 2, 'B']
```

E uma rápida demonstração de `UpperCounter`:

```
>>> c = UpperCounter('BaNaNA')
>>> c.most_common()
[('A', 3), ('N', 2), ('B', 1)]
```

`UpperDict` e `UpperCounter` parecem quase mágicas, mas tive que estudar cuidadosamente o código de `UserDict` e `Counter` para fazer `UpperCaseMixin` trabalhar com eles. Por exemplo, minha primeira versão de `UpperCaseMixin` não incluía o método `get`. Aquela versão funcionava com `UserDict`, mas não com `Counter`. A classe `UserDict` herda `get` de `collections.abc.Mapping`, e aquele `get` invoca `__getitem__`, que implementei. Mas as chaves não eram transformadas em maiúsculas quando uma `UpperCounter` era carregada no `__init__`. Isso acontecia porque `Counter.__init__` usa `Counter.update`, que por sua vez recorre ao método `get` herdado de `dict`. Entretanto, o método `get` na classe `dict` não invoca `__getitem__`.

Esta é a essência do problema discutido na «Seção 3.5.3» [fpy.li/88] (vol.1). É também uma clara demonstração da natureza frágil e quebradiça de programas que se apoiam no acoplamento forte da herança, mesmo nessa pequena escala.

A próxima seção apresenta vários exemplos de herança múltipla, muitas vezes usando classes `mixin`.

14.6. Herança múltipla no mundo real

No livro *Design Patterns* (Padrões de Projetos),^[10] quase todo o código está em C++. O único exemplo de herança múltipla é o padrão *Adapter* (Adaptador). Em Python a herança múltipla também não é regra, mas há exemplos importantes, que comentarei nessa seção.

14.6.1. ABCs também são *mixins*

Na biblioteca padrão de Python, o uso mais visível de herança múltipla é o pacote `collections.abc`. Nenhuma controvérsia aqui: afinal, até o Java suporta herança múltipla de interfaces, e ABCs são declarações de interface que podem, opcionalmente, fornecer implementações concretas de métodos.^[11]

A documentação oficial do pacote `collections.abc` [fpy.li/6z] chama de *mixin methods* (métodos mixin) os métodos concretos implementados nas ABCs de coleções. As ABCs que oferecem métodos mixin cumprem dois papéis: elas são definições de interfaces e também classes mixin. Por exemplo, a «implementação» [fpy.li/14-14] de `collections.UserDict` aproveita vários métodos mixin fornecidos por `collections.abc.MutableMapping`.

14.6.2. `ThreadingMixIn` e `ForkingMixIn`

O pacote `http.server` [fpy.li/72] inclui as classes `HTTPServer` e `ThreadingHTTPServer`. Esta última foi adicionada ao Python 3.7. A documentação de `ThreadingHTTPServer` diz (nossa tradução):

Esta classe é idêntica a HTTPServer, mas trata requisições com threads, usando a ThreadingMixIn. Isso é útil para lidar com navegadores Web que abrem sockets prematuramente, situação na qual o HTTPServer esperaria indefinidamente.

As duas linhas abaixo são o «código-fonte completo» [fpy.li/14-16] da classe `ThreadingHTTPServer` no Python 3.10:

```
class ThreadingHTTPServer(socketserver.ThreadingMixIn, HTTPServer):
    daemon_threads = True
```

O «código-fonte» [fpy.li/14-17] de `socketserver.ThreadingMixIn` tem 38 linhas, incluindo os comentários e as docstrings. O Exemplo 10 apresenta um resumo de sua implementação.

Exemplo 10. Parte de Lib/socketserver.py no Python 3.10

```
class ThreadingMixIn:
    """Mixin class to handle each request in a new thread."""

    # 8 linhas omitidas aqui

    def process_request_thread(self, request, client_address): ①
        ... # 6 linhas omitidas aqui

    def process_request(self, request, client_address): ②
        ... # 8 linhas omitidas aqui

    def server_close(self): ③
        super().server_close()
        self._threads.join()
```

- ① `process_request_thread` não invoca `super()` porque é um método novo, não sobrescreve um método herdado. Sua implementação invoca três métodos de instância que `HTTPServer` implementa ou herda.
- ② Isto sobrescreve o método `process_request`, que `HTTPServer` herda de `socketserver.BaseServer`, iniciando uma thread e delegando o trabalho real para a `process_request_thread` que roda naquela thread. O método não invoca `super()`.
- ③ `server_close` invoca `super().server_close()` para parar de receber requisições, e então espera que as threads iniciadas por `process_request` terminem sua execução.

A documentação do módulo `socketserver` [fpy.li/73] apresenta a `ThreadingMixIn` e a `ForkingMixIn`. Esta última classe foi projetada para suportar servidores concorrentes baseados em `os.fork()` [fpy.li/74], uma API para iniciar processos filhos, disponível em sistemas derivados do Unix, compatíveis com a norma «POSIX» [fpy.li/7c].

14.6.3. Mixins de views genéricas no Django



Não é necessário conhecer Django para acompanhar essa seção. Uso uma pequena parte do framework como um exemplo prático de herança múltipla, e tentarei fornecer todo o pano de fundo necessário (supondo que você tenha alguma experiência com desenvolvimento Web no lado servidor, com qualquer linguagem ou framework).

No Django, uma view é um objeto invocável que recebe um argumento `request`—um objeto representando uma requisição HTTP—e devolve um objeto representando uma resposta HTTP. Nosso interesse aqui são as diferentes respostas. Elas podem ser tão simples quanto um redirecionamento, sem nenhum conteúdo em seu corpo, ou tão complexas quanto uma página de catálogo de uma loja online, renderizada a partir de um template HTML que exibe múltiplas mercadorias, com botões de compra e links para páginas com detalhes.

Originalmente, o Django oferecia uma série de funções, chamadas *generic views* (views genéricas), que implementavam alguns casos de uso comuns. Por exemplo, muitos sites precisam exibir resultados de busca que incluem dados de vários itens, com listagens ocupando múltiplas páginas, cada resultado contendo também um link para uma página de informações detalhadas sobre aquele item. No Django, uma view de lista e uma view de detalhes são feitas para funcionarem juntas, resolvendo esse problema: uma view de lista renderiza resultados de busca, e uma view de detalhes produz uma página para cada item individual.

Entretanto, as views genéricas originais eram funções, então não eram extensíveis. Se quiséssemos algo similar mas não exatamente igual a uma view de lista genérica, era preciso começar do zero.

O conceito de views baseadas em classes foi introduzido no Django 1.3, juntamente com um conjunto de classes de views genéricas divididas em classes base, mixins e classes concretas prontas para o uso. No Django 3.2, as classes base e as mixins estão no módulo base do pacote `django.views.generic`, ilustrado na Figura 3. No topo do diagrama vemos duas classes que se encarregam de responsabilidades muito diferentes: `View` e `TemplateResponseMixin`.

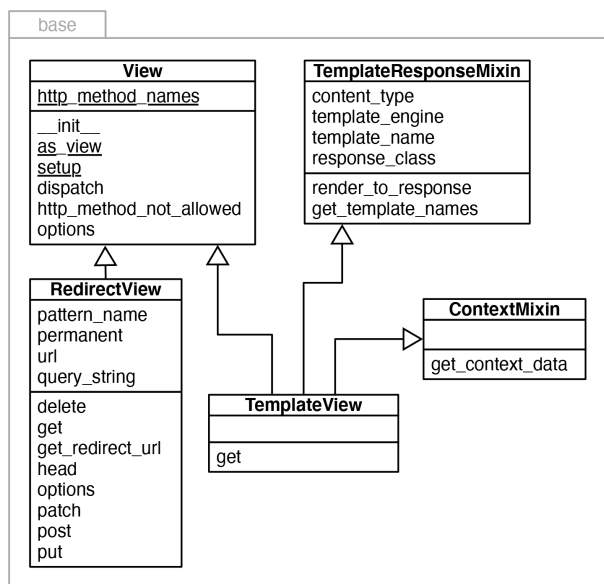


Figura 3. Diagrama de classes UML do módulo `django.views.generic.base`.



Um ótimo recurso para estudar essas classes é o site *Classy Class-Based Views* [fpy.li/14-21], onde você pode navegar facilmente pelo diagrama das classes, ver todos os métodos em cada classe (métodos herdados, sobrescritos e adicionados), consultar sua documentação e estudar seu «código-fonte no GitHub» [fpy.li/14-22].

`View` é a classe base de todas as views (ela poderia ser uma ABC), e oferece funcionalidade essencial como o método `dispatch`, que delega para métodos de tratamento de requisições (*request handling*) como `get`, `head`, `post`, etc., implementados por subclasses concretas para tratar os diversos verbos HTTP.^[12] A classe `RedirectView` herda de `View` e implementa `get`, `head`, `post`, etc.

Espera-se que as subclasses concretas de `View` implementem os métodos de tratamento, então por que aqueles métodos não são parte da interface de `View`? A razão: subclasses são livres para implementar apenas os métodos de tratamento que querem suportar. Uma `TemplateView` é usada apenas para exibir conteúdo, então ela implementa apenas `get`. Se uma requisição HTTP POST é enviada para uma `TemplateView`, o método herdado `View.dispatch` verifica que não há um método de tratamento para `post`, e produz uma resposta HTTP 405 Method Not Allowed.^[13]

A `TemplateResponseMixin` fornece funcionalidade que interessa apenas às views que precisam usar um template. Uma `RedirectView`, por exemplo, não tem conteúdo, então não precisa de um template e não herda dessa mixin. `TemplateResponseMixin` fornece comportamentos para `TemplateView` e outras views que renderizam templates, tal como `ListView`, `DetailView`, etc., definidas nos subpacotes de `django.views.generic`. A Figura 4 mostra o módulo `django.views.generic.list` e parte do módulo base.

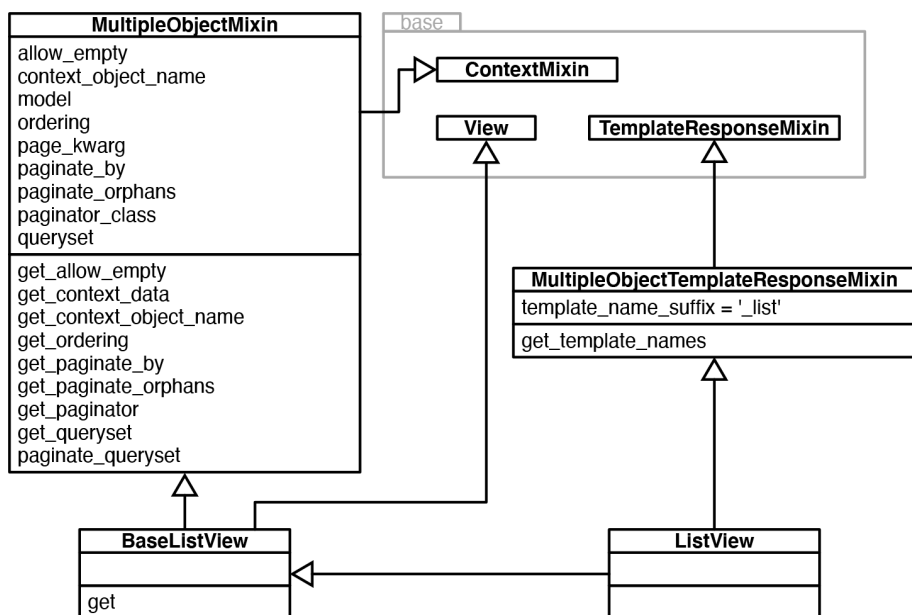


Figura 4. Diagrama de classe UML do o módulo `django.views.generic.list`. Aqui as três classes do módulo base aparecem recolhidas (veja a Figura 3). A classe `ListView` não tem métodos ou atributos: é uma classe agregada.

Para usuários do Django, a classe mais importante na Figura 4 é `ListView`, uma classe agregada sem qualquer código (seu corpo é apenas uma docstring). Quando instanciada, uma `ListView` tem um atributo de instância `object_list`, através do qual o código do template pode iterar para montar o conteúdo da página, normalmente o resultado de uma consulta a um banco de dados, composto de múltiplos objetos. Toda a funcionalidade relacionada com a geração deste iterável de objetos vem da `MultipleObjectMixin`. Esta mixin também oferece uma lógica complexa de paginação—para exibir parte dos resultados em uma página e links para mais páginas.

Suponha que você queira criar uma view que não vai renderizar um template, mas sim produzir uma lista de objetos em formato JSON. Para isso existe `BaseListView`. Ela oferece um ponto inicial de extensão fácil de usar, unindo a funcionalidade de `View` e de `MultipleObjectMixin`, mas sem a complexidade do mecanismo de templates.

A API de views baseadas em classes do Django é um exemplo melhor de herança múltipla que o Tkinter. É mais fácil entender suas classes mixin: cada uma tem um propósito bem definido, e seus nomes terminam com o sufixo `...Mixin`.

Views baseadas em classes não são universalmente aceitas por usuários do Django. Muitos as usam de forma limitada, como caixas opacas. Mas quando é necessário criar algo novo, muitos programadores Django continuam criando funções monolíticas de views, para abarcar todas aquelas responsabilidades, ao invés de tentar reutilizar as views base e as mixins.

Demora um certo tempo para aprender a usar as views baseadas em classes e a forma de estendê-las para suprir as necessidades específicas de uma aplicação, mas considero que vale a pena estudá-las. Elas eliminam muito código repetitivo, facilitam o reuso de soluções, e melhoram até a comunicação das equipes—por exemplo, pela definição de nomes padronizados para os templates e para as variáveis passadas para contextos de templates. Views baseadas em classes são views do Django "on rails"^[14].

14.6.4. Herança múltipla no Tkinter

Um exemplo extremo de herança múltipla na biblioteca padrão de Python é o toolkit de interface gráfica «Tkinter» [fpy.li/76]. No momento em que escrevo essa seção, o Tkinter já tem 25 anos de idade. Não é um exemplo das melhores práticas atuais. Mas mostra como a herança múltipla era usada quando os programadores ainda não conheciam suas desvantagens. E vai nos servir de contra-exemplo, quando tratarmos de algumas boas práticas, na próxima seção.

Usei parte da hierarquia de componentes do Tkinter para ilustrar a MRO na Figura 2. A Figura 5 mostra todas as classes de componentes no pacote base `tkinter` (há mais componentes gráficos no subpacote `tkinter.ttk` [fpy.li/77]).

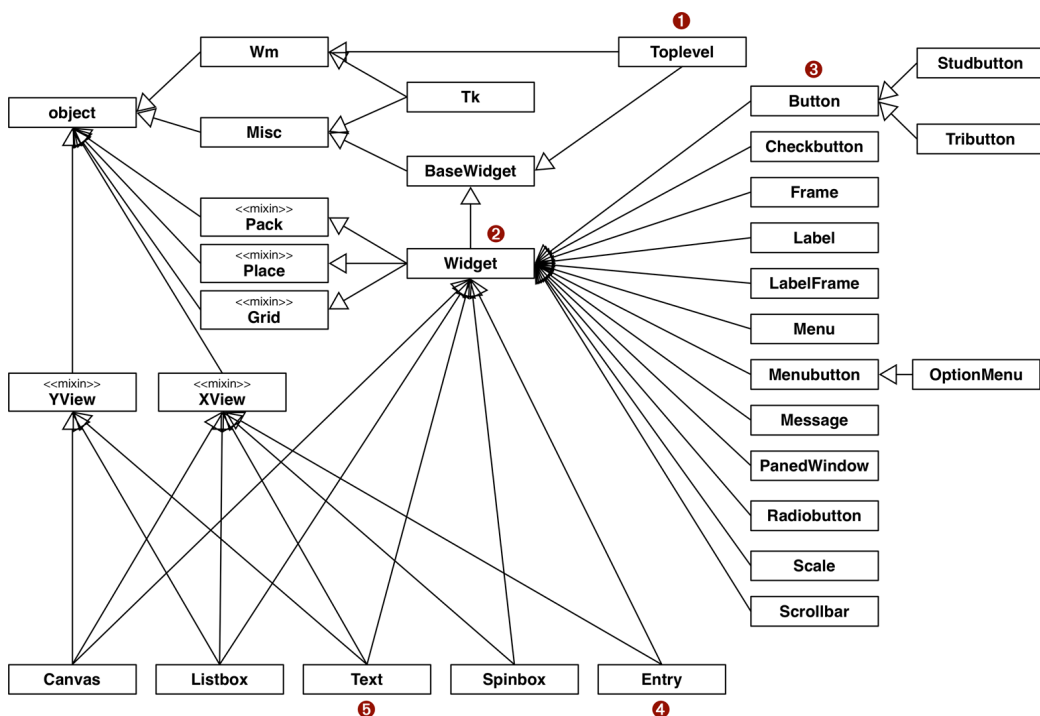


Figura 5. Diagrama de classes resumido da hierarquia de classes de interface gráfica do Tkinter; classes marcadas com «mixin» existem para oferecer metodos concretos a outras classes, por herança múltipla.

Considere as seguintes classes na Figura 5:

- ① Toplevel: A classe de uma janela principal em um aplicação Tkinter.
- ② Widget: A superclasse de todos os objetos visíveis que podem ser colocados em uma janela.
- ③ Button: Um componente de botão simples.
- ④ Entry: Um campo de texto editável de uma única linha.
- ⑤ Text: Um campo de texto editável de múltiplas linhas.

Aqui estão as MROs dessas classes, como exibidas pela função `print_mro` do Exemplo 7:

```
>>> import tkinter
>>> print_mro(tkinter.Toplevel)
Toplevel, BaseWidget, Misc, Wm, object
>>> print_mro(tkinter.Widget)
Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Button)
Button, Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Entry)
Entry, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, object
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```



Pelos padrões atuais, a hierarquia de classes do Tkinter é profunda demais. Poucas partes da biblioteca padrão de Python tem mais que três ou quatro níveis de classes concretas, e o mesmo pode ser dito da biblioteca de classes de Java. Entretanto, é interessante observar que algumas das hierarquias mais profundas da biblioteca de classes de Java são precisamente os pacotes relacionados à programação de interfaces gráficas: `java.awt` [fpy.li/14-26] e `javax.swing` [fpy.li/14-27]. O «Squeak» [fpy.li/14-28], uma versão moderna e aberta de Smalltalk, inclui o poderoso e inovador toolkit de interface gráfica Morphic, também com uma hierarquia de classes profunda. Na minha experiência, é nos toolkits de interface gráfica que a herança é mais útil.

Observe como essas classes se relacionam com outras:

- `Toplevel` é a única classe gráfica que não herda de `Widget`, porque ela é a janela primária e não se comporta como um componente; por exemplo, ela não pode ser fixada a uma janela ou moldura (*frame*). `Toplevel` herda de `Wm`, que fornece funções de acesso direto ao gerenciador de janelas do ambiente gráfico do sistema operacional, para tarefas como definir o título da janela e configurar suas bordas.

- Widget herda diretamente de BaseWidget e de Pack, Place, e Grid. As últimas três classes são gerenciadores de geometria: são responsáveis por organizar componentes dentro de uma janela ou moldura. Cada uma delas encapsula uma estratégia de layout e uma API de colocação de componentes diferente.
- Button, como a maioria dos componentes, descende diretamente apenas de Widget, mas indiretamente de Misc, que fornece dezenas de métodos para todos os componentes.
- Entry é subclasse de Widget e XView, que suporta rolagem horizontal.
- Text é subclasse de Widget, XView e YView (para rolagem vertical).

Vamos agora discutir algumas boas práticas de herança múltipla e examinar como o Tkinter se comporta.

14.7. Lidando com a herança

Aquilo que Alan Kay escreveu na epígrafe continua sendo verdade: ainda não existe um teoria geral sobre herança que guie os programadores. O que temos são regras gerais, padrões de projetos, "melhores práticas", acrônimos perspicazes, tabus, etc. Alguns desses nos dão orientações úteis, mas nenhum deles é universalmente aceito ou sempre aplicável.

É fácil criar projetos frágeis e incompreensíveis usando herança, mesmo sem herança múltipla. Como não temos uma teoria abrangente, aqui estão algumas dicas para evitar diagramas de classes parecidos com um prato de espaguete.

14.7.1. Prefira a composição de objetos à herança de classes

O título desta seção é o segundo princípio do design orientado a objetos, do livro *Padrões de Projetos*,^[15] e é o melhor conselho que posso oferecer aqui. Uma vez que você se sinta confortável com a herança, é fácil usá-la em excesso. Colocar objetos em uma hierarquia elegante apela para nosso senso de ordem; programadores fazem isso por pura diversão.

Preferir a composição leva a designs mais flexíveis. Por exemplo, no caso da classe `tkinter.Widget`, em vez de herdar os métodos de todos os gerenciadores de geometria, instâncias do componente poderiam manter uma referência para um gerenciador de geometria, e invocar seus métodos. Afinal, um `Widget` não deveria

"ser" um gerenciador de geometria, mas poderia usar os serviços de um deles por delegação. E daí você poderia adicionar um novo gerenciador de geometria sem afetar a hierarquia de classes do componente e sem se preocupar com colisões de nomes.

Mesmo com herança simples, este princípio aumenta a flexibilidade, porque a subclasses são uma forma de acoplamento forte, e árvores de herança muito altas tendem a ser quebradiças.

A composição e a delegação podem substituir o uso de mixins para tornar comportamentos disponíveis para diferentes classes, mas não podem substituir o uso de herança de interfaces para definir uma hierarquia de tipos.

14.7.2. Entenda o motivo de usar herança em cada caso

Ao lidarmos com herança múltipla, é útil ter claras as razões pelas quais subclasses são criadas em cada caso específico. As principais razões são:

- Herança de interface cria um subtipo, implicando em uma relação *é-um*. A melhor forma de fazer isso é usando ABCs.
- Herança de implementação evita duplicação de código pela reutilização. Mixins podem ajudar nisso.

Na prática, frequentemente as duas razões coexistem, mas quando você puder tornar a intenção clara, faça isso. Herança para reutilização de código é um detalhe de implementação, e muitas vezes pode ser substituída por composição e delegação. Por outro lado, herança de interfaces é o fundamento de qualquer framework. Idealmente, a herança de interfaces deveria usar apenas ABCs como classes base.

14.7.3. Torne a interface explícita com ABCs

No Python moderno, se uma classe tem por objetivo definir uma interface, ela deveria ser explicitamente uma ABC ou uma subclasse de `typing.Protocol`. Uma ABC deveria ser subclasse apenas de `abc.ABC` ou de outras ABCs. A herança múltipla de ABCs não é problemática.

14.7.4. Use mixins explícitas para reutilizar código

Se uma classe é projetada para fornecer implementações de métodos para reutilização por múltiplas subclasses não relacionadas, sem implicar em uma relação do tipo *é-uma*, ele deveria ser uma classe mixin explícita. No Python, não há uma maneira formal de declarar uma classe como mixin. Por isso, recomendo que seus nomes incluam o sufixo `Mixin`. Conceitualmente, uma mixin não define um novo tipo; ela simplesmente empacota métodos para reutilização. Uma mixin não deveria nunca ser instanciada, e classes concretas não devem herdar apenas de uma mixin. Cada mixin deveria fornecer um único comportamento específico, implementando poucos métodos intimamente relacionados. Mixins devem evitar manter qualquer estado interno; isto é, uma classe mixin não deve ter atributos de instância.

14.7.5. Ofereça classes agregadas aos usuários

Uma classe construída principalmente herdando de mixins, sem adicionar estrutura ou comportamento próprios, é chamada de classe agregada.^[16]

— Grady Booch et al., *Object-Oriented Analysis and Design with Applications*

Se alguma combinação de ABCs ou mixins for especialmente útil para o código cliente, ofereça uma classe que una essas funcionalidades de uma forma sensata.

Por exemplo, aqui está o «código-fonte» [fpy.li/14-29] completo da classe `ListView` do Django, do canto inferior direito da Figura 4:

```
class ListView(MultipleObjectTemplateResponseMixin, BaseListView):
    """
    Render some list of objects, set by `self.model` or `self.queryset`.
    `self.queryset` can actually be any iterable of items, not just a
    queryset.
    """
```

O corpo de `ListView` é vazio^[17], mas a classe fornece um serviço útil: ela une uma mixin e uma classe base que devem ser usadas em conjunto.

Outro exemplo é `tkinter.Widget` [fpy.li/14-30], que tem quatro classes base e nenhum método ou atributo próprios—apenas uma docstring. Graças à classe agregada `Widget`, podemos criar um novo componente com as mixins necessárias, sem precisar descobrir em que ordem elas devem ser declaradas para funcionarem como desejado.

Note que classes agregadas não precisam ser inteiramente vazias (mas frequentemente são).

14.7.6. Só crie subclasses de classes feitas para serem herdadas

Em um comentário sobre esse capítulo, o revisor técnico Leonardo Rochael sugeriu o alerta abaixo.



Criar subclasses e sobrescrever métodos de qualquer classe complexa é um processo muito suscetível a erros, porque os métodos da superclasse podem ignorar inesperadamente métodos sobrescritos na subclasse. Sempre que possível, evite sobrescrever métodos, ou pelo menos limite-se a criar subclasses de classes projetadas para serem facilmente estendidas, e apenas daquelas formas pelas quais a classe foi desenhada para ser estendida.

É um ótimo conselho, mas como descobrimos se uma classe foi projetada para ser estendida?

A primeira resposta é a documentação (algumas vezes na forma de docstrings ou até de comentários no código). Por exemplo, o pacote `socketserver` [fpy.li/78] de Python é descrito como "um framework para servidores de rede". Sua classe `BaseServer` [fpy.li/79] foi projetada para a criação de subclasses, como o próprio nome sugere. E mais importante, a documentação e a «docstring no código-fonte da classe» [fpy.li/14-33] informa explicitamente quais de seus métodos foram criados para serem sobrescritos por subclasses.

No Python ≥ 3.8 uma nova forma de tornar tais restrições de projeto explícitas foi oferecida pela *PEP 591—Adding a final qualifier to typing* [fpy.li/pep591] (Acrescentando um qualificador "final" à tipagem). A PEP introduz um decorador `@final` [fpy.li/7a], que pode ser aplicado a classes ou a métodos individuais, para

que IDEs ou checadores de tipos possam detectar tentativas de criar subclasses de classes ou de sobrescrever métodos que não foram projetados para serem herdadas ou sobrescritos.^[18]

14.7.7. Evite criar subclasses de classes concretas

Criar subclasses de classes concretas é mais perigoso que criar subclasses de ABCs e mixins, pois instâncias de classes concretas normalmente têm um estado interno, que pode ser corrompido quando sobrescrevemos métodos que interferem naquele estado. Mesmo se nossos métodos cooperarem chamando `super()`, e o estado interno seja protegido através da sintaxe `__x`, restarão ainda inúmeras formas pelas quais sobrescrever um método pode introduzir bugs.

No texto *Pássaros aquáticos e as ABCs* (Seção 13.5), Alex Martelli cita *More Effective C++*, de Scott Meyer, que diz: "toda classe não-final (não-folha) deveria ser abstrata". Em outras palavras, Meyer recomenda que subclasses deveriam ser criadas apenas a partir de classes abstratas.

Se você precisar usar subclasses para reutilização de código, então o código a ser reutilizado deve estar em métodos mixin de ABCs, ou em classes mixin explicitamente nomeadas.

Vamos agora analisar o Tkinter do ponto de vista destas recomendações.

14.7.8. Tkinter: o bom, o mau e o feio

A maioria dos conselhos da seção anterior não são seguidos pelo Tkinter, com a notável exceção de oferecer classes agregadas (Seção 14.7.5). E mesmo assim, este não é um grande exemplo, pois a composição provavelmente funcionaria melhor para integrar os gerenciadores de geometria a Widget, como discutido na Seção 14.7.1.

Mas lembre-se de que o Tkinter é parte da biblioteca padrão desde o Python 1.1, lançado em 1994. O Tkinter é uma fachada em Python para o toolkit de GUI Tk, escrito na linguagem Tcl. O combo Tcl/Tk não é, na origem, orientado a objetos, então a API Tk é basicamente um imenso catálogo de funções. Entretanto, o toolkit é conceitualmente orientado a objetos, apesar de não usar classes na implementação original em Tcl.

A docstring de `tkinter.Widget` começa com as palavras "Internal class" (Classe interna). Isto sugere que `Widget` deveria provavelmente ser uma ABC. Apesar da classe `Widget` não ter métodos próprios, ela define uma interface. Sua mensagem é: "Você pode contar que todos os componentes do Tkinter vão oferecer os métodos básicos de componente (`__init__`, `destroy`, e dezenas de funções da API Tk), além dos métodos de todos os três gerenciadores de geometria". Vamos combinar que essa não é uma boa definição de interface (é abrangente demais), mas ainda assim é uma interface, e `Widget` a "define" como a união das interfaces de suas superclasses.

A classe `Tk`, que encapsula a lógica da aplicação gráfica, herda de `Wm` e `Misc`, nenhuma das quais é abstrata ou mixin (`Wm` não é uma mixin típica, porque `TopLevel` é subclasse apenas dela). O nome da classe `Misc` é, por si só, um mau sinal. `Misc` tem mais de 100 métodos, e todos os componentes herdam dela. Por que é necessário que cada um dos componentes tenham métodos para tratamento do clipboard, seleção de texto, gerenciamento de timer e coisas assim? Não é possível colar algo em um botão ou selecionar texto de uma barra de rolagem. `Misc` deveria ser dividida em várias classes mixin especializadas, e nem todos os componentes deveriam herdar de todas aquelas mixins.

Para ser justo, como usuário do Tkinter você não precisa, de forma alguma, entender ou usar herança múltipla. Ela é um detalhe de implementação, oculto atrás das classes de componentes que serão instanciadas ou usadas como base para subclasses em seu código. Mas você sofrerá as consequências da herança múltipla excessiva quando digitar `dir(tkinter.Button)` e tentar encontrar um método específico em meio aos 214 atributos listados. E terá que enfrentar a complexidade, caso decida implementar um novo componente Tk.



Apesar de ter problemas, o Tkinter é estável, flexível, e fornece um visual moderno se você usar o pacote `tkinter.ttk` e seus componentes tematizados. Além disso, alguns dos componentes originais, como `Canvas` e `Text`, são incrivelmente poderosos. Em poucas horas é possível transformar um objeto `Canvas` em uma aplicação de desenho razoavelmente completa. Se você se interessa pela programação de interfaces gráficas, com certeza vale a pena estudar o Tkinter e o Tcl/Tk.

Aqui termina nossa viagem através do labirinto da herança.

14.8. Resumo do capítulo

Este capítulo começou com uma revisão da função `super()` no contexto de herança simples. Daí discutimos o problema da criação de subclasses de tipos embutidos: seus métodos nativos, implementados em C, não invocam os métodos sobrescritos em subclasses, exceto em uns poucos casos especiais. É por isso que, quando precisamos de tipos `list`, `dict`, ou `str` customizados, é mais fácil criar subclasses de `UserList`, `UserDict`, ou `UserString` (todos definidos no módulo `'collections [fpy.li/2w]'`), que encapsulam os tipos embutidos correspondentes e delegam operações para aqueles—três exemplos a favor da composição sobre a herança na biblioteca padrão. Se o comportamento desejado for muito diferente daquilo que os tipos embutidos oferecem, pode ser mais fácil criar uma subclasse da ABC apropriada em `collections.abc [fpy.li/6z]`, e escrever sua própria implementação.

O restante do capítulo foi dedicado à faca de dois gumes da herança múltipla. Primeiro vimos como a ordem de resolução de métodos, definida no atributo de classe `__mro__`, trata o problema de conflitos potenciais de nomes em métodos herdados. Também examinamos como a função embutida `super()` se comporta em hierarquias com herança múltipla, e como ela às vezes tem um comportamento surpreendente. O comportamento de `super()` foi projetado para suportar classes `mixin`, que estudamos usando o exemplo simples de `UpperCaseMixin` (para mapeamentos indiferentes a maiúsculas/minúsculas).

Exploramos como a herança múltipla e os métodos `mixin` são usados nas ABCs de Python, bem como na construção de servidores HTTP com `mixins` baseados em `threads` e `forks` de `socketserver`. Usos mais complexos de herança múltipla foram exemplificados com as `views` baseadas em classes do Django e com o toolkit de interface gráfica Tkinter. Apesar do Tkinter não ser um exemplo das melhores práticas modernas, é um exemplo de hierarquias de classe complexas que podemos encontrar em sistemas legados.

Encerrando o capítulo, apresentamos sete recomendações para lidar com herança, e aplicamos alguns daqueles conselhos em um comentário sobre a hierarquia de classes do Tkinter.

Rejeitar a herança—mesmo a herança simples—é uma tendência moderna. Go é uma das mais bem sucedidas linguagens criadas no século 21. Ela não inclui um

elemento chamado "classe", mas você pode construir tipos que são estruturas (*structs*) de campos encapsulados, e associar métodos a essas estruturas. Em Go é possível definir interfaces, que são checadas pelo compilador usando tipagem estrutural, também conhecida como tipagem pato estática—algo muito similar ao que temos com os tipos protocolo desde o Python 3.8. Essa linguagem também tem uma sintaxe especial para a criação de tipos e interfaces por composição, mas não há suporte a herança—nem entre interfaces.

Então talvez o melhor conselho sobre herança seja: evite-a se puder. Mas, frequentemente, não temos essa opção: os frameworks que usamos nos impõe suas escolhas de design.

14.9. Para saber mais

No que diz respeito à legibilidade, composição feita adequadamente é superior a herança. Como é mais frequente ler o código que escrevê-lo, como regra geral evite subclasses, mas em especial não misture os vários tipos de herança e não crie subclasses para compartilhar código.

— Hynek Schlawack, *Subclassing in Python Redux*

Durante a revisão final desse livro, o revisor técnico Jürgen Gmach recomendou o post *Subclassing in Python Redux* [fpy.li/14-37] (O ressurgimento das subclasses em Python), de Hynek Schlawack—a fonte da citação acima. Schlawack é o autor do popular pacote *attrs*, e foi um dos principais contribuidores do framework de programação assíncrona *Twisted*, um projeto criado por Glyph Lefkowitz em 2002. De acordo com Schlawack, após algum tempo os desenvolvedores perceberam que haviam criado subclasses em excesso no projeto. O post é longo, e cita outros posts e palestras importantes. Muito recomendado.

Naquela mesma conclusão, Hynek Schlawack escreve: "Não esqueça que, na maioria dos casos, tudo o que você precisa é de uma função." Concordo, e é precisamente por essa razão que *Python Fluente* trata em detalhes das funções, antes de falar de classes e herança. Meu objetivo foi mostrar o quanto você pode alcançar com funções se valendo das classes na biblioteca padrão, antes de criar suas próprias classes.

A criação de subclasses de tipos embutidos, a função `super`, e recursos avançados como descritores e metaclasses, foram todos introduzidos no artigo *Unifying types and classes in Python 2.2* [fpy.li/descr101] (Unificando tipos e classes em Python 2.2), de Guido van Rossum. Desde então, nada realmente importante mudou nesses recursos. Python 2.2 foi uma proeza fantástica de evolução da linguagem, adicionando vários novos recursos poderosos em um todo coerente, sem quebrar a compatibilidade com versões anteriores. Os novos recursos eram 100% opcionais. Para usá-los, bastava programar explicitamente uma subclasse de direta ou indireta de `object`, para criar uma assim chamada *new-style class* (classe no novo estilo). No Python 3, todas as classes são subclasses de `object`.

O *Python Cookbook*, 3ª ed., de David Beazley e Brian K. Jones (O'Reilly) inclui várias receitas mostrando o uso de `super()` e de classes *mixin*. Você pode começar pela esclarecedora seção 8.7. *Calling a Method on a Parent Class* [fpy.li/14-38] (Invocando um método em uma superclasse), e seguir as referências internas a partir dali.

O post *Python's super() considered super!* [fpy.li/14-39] (O `super()` de Python é mesmo *super!*!], de Raymond Hettinger, explica o funcionamento de `super` e a herança múltipla de uma perspectiva positiva. Ele foi escrito em resposta a *_Python's Super is nifty, but you can't use it (Previously: Python's Super Considered Harmful)* [fpy.li/14-40] (O `super` de Python é bacana, mas você não deve usá-lo (Antes: `super` de Python considerado nocivo)), de James Knight. A resposta de Martijn Pieters a *How to use super() with one argument?* [fpy.li/14-41] (Como usar `super()` com um só argumento?) inclui uma explicação concisa e aprofundada de `super`, incluindo sua relação com descritores, um conceito que estudaremos apenas no «Capítulo 23» [fpy.li/23] (vol.3). Assim é `super`: simples de usar nos casos básicos, mas também uma ferramenta poderosa e complexa, que alcança alguns dos recursos dinâmicos mais avançados de Python, raramente encontrados em outras linguagens.

Apesar dos títulos daqueles posts, o problema não é exatamente a função embutida `super`—que no Python 3 ficou mais fácil de usar do que era no Python 2. A questão real é a herança múltipla, algo inerentemente complicado e traiçoeiro. Michele Simionato vai além da crítica, e de fato oferece uma solução em seu *Setting Multiple Inheritance Straight* [fpy.li/14-42] (Colocando a herança múltipla em seu devido lugar): ele implementa *traits* ("traços"), uma forma explícita de *mixin* originada na linguagem Self. Simionato escreveu, em seu blog, uma longa série de posts sobre herança múltipla em Python, incluindo *The wonders of*

cooperative inheritance, or using super in Python 3 [fpy.li/14-43] (As maravilhas da herança cooperativa, ou usando super em Python 3); *Mixins considered harmful, part 1* [fpy.li/14-44] (Mixins consideradas nocivas, parte 1) e *part 2* [fpy.li/14-45]; e *Things to Know About Python Super, part 1* [fpy.li/14-46] (O que você precisa saber sobre o super de Python), *part 2* [fpy.li/14-47], e *part 3* [fpy.li/14-48]. Os posts mais antigos usam a sintaxe de super de Python 2, mas ainda são relevantes.

Li a primeira edição do *Object-Oriented Analysis and Design*, 3ª ed., de Grady Booch et al., e o recomendo fortemente como uma introdução geral ao pensamento orientado a objetos, independente da linguagem de programação. É um dos raros livros que trata da herança múltipla sem preconceitos.

Hoje, mais que nunca, aconselha-se evitar a herança. Então cá estão duas referências sobre como fazer isso. Brandon Rhodes escreveu *The Composition Over Inheritance Principle* [fpy.li/14-49] (O princípio da composição antes da herança), parte de seu excelente guia *Python Design Patterns* [fpy.li/14-50] (Padrões de Projetos no Python). Augie Fackler e Nathaniel Manista apresentaram *The End Of Object Inheritance & The Beginning Of A New Modularity* [fpy.li/14-51] (O Fim da Herança de Objetos & O Início de Uma Nova Modularidade) na PyCon 2013. Fackler e Manista falam sobre organizar sistemas em torno de interfaces e das funções que lidam com os objetos que implementam aquelas interfaces, evitando o acoplamento forte e os pontos de falha de classes e da herança. É o modo de pensar da comunidade Go, aplicado ao Python.

Soapbox

Pense nas classes realmente necessárias

[...] começamos a defender a ideia de herança como uma maneira de permitir que iniciantes pudessem construir [algo] a partir de frameworks que só poderiam ser projetadas por especialistas^[19].

— Alan Kay, The Early History of Smalltalk ("Os Primórdios de Smalltalk")

A imensa maioria dos programadores escreve aplicações, não frameworks. Mesmo aqueles que escrevem frameworks provavelmente passam boa parte de seu tempo escrevendo aplicações. Quando escrevemos aplicações, normalmente não precisamos criar hierarquias de classes. No máximo

escrevemos classes que são subclasses de ABCs ou de outras classes oferecidas pelo framework. Como desenvolvedores de aplicações, é muito raro precisarmos escrever uma classe que funcionará como superclasse de outra. Quase sempre as classes que escrevemos são classes folha: classes concretas sem subclasses.

Se, trabalhando como desenvolvedor de aplicações, você se pegar criando hierarquias de classe de múltiplos níveis, aposto que está vivendo uma destas situações:

- Você está reinventando a roda. Procure um framework ou biblioteca que forneça componentes que você possa reutilizar em sua aplicação.
- Você está usando um framework mal projetado. Procure uma alternativa.
- Você está complicando demais. Lembre-se do *Princípio KISS*.
- Você ficou entediado programando aplicações e decidiu criar um novo framework. Parabéns e boa sorte!

Também é possível que todas as alternativas acima descrevam situação: você ficou entediado e decidiu reinventar a roda, escrevendo seu próprio framework mal projetado e excessivamente complexo, e está sendo forçado a programar classe após classe para resolver problemas triviais. Espero que você esteja se divertindo, ou pelo menos que esteja sendo pago para fazer isso.

Tipos embutidos mal-comportados: bug ou *feature*?

Os tipos embutidos `dict`, `list`, e `str` são blocos básicos essenciais do próprio Python, então precisam ser rápidos—qualquer problema de desempenho ali teria severos impactos em praticamente todo o resto. É por isso que o CPython adotou atalhos que fazem com que muitos métodos embutidos escritos em C se comportem mal, ao não cooperarem com os métodos sobrescritos por subclasses em Python.

Uma solução para este dilema seria oferecer duas implementações para cada um desses tipos: uma "interno", otimizada para uso pelo interpretador, e uma externa, facilmente extensível.

Mas veja só, isso nós já temos: `UserDict`, `UserList`, e `UserString` não são tão rápidos quanto seus equivalentes embutidos, mas são fáceis de estender. A abordagem pragmática tomada pelo CPython significa que também podemos usar, em nossas próprias aplicações, as implementações altamente otimizadas mas difíceis de estender. E isso faz sentido, considerando que não é tão frequente precisarmos de um mapeamento, uma lista ou uma string customizados, mas usamos `dict`, `list`, e `str` diariamente. Só precisamos estar cientes dos compromissos envolvidos.

Herança através das linguagens

Alan Kay criou o termo "orientado a objetos", e Smalltalk tinha apenas herança simples, apesar de existirem versões com diferentes formas de suporte a herança múltipla, incluindo os dialetos modernos de Smalltalk, Squeak e Pharo, que suportam *traits* ("traços")—um dispositivo de linguagem que pode substituir classes mixin, mas evita alguns dos problemas da herança múltipla.

A primeira linguagem popular a implementar herança múltipla foi o C++, e esse recurso foi abusado o suficiente para que o Java—criado para ser um substituto do C++—fosse projetado sem suporte a herança múltipla de implementação (isto é, sem classes mixin). Quer dizer, isso até o Java 8 (e Kotlin) permitir métodos default, que que aproximam suas interfaces do conceito de classes abstratas que temos em Python e C++.

Outras linguagens que suportam *traits* são versões recentes de PHP e Groovy, bem como Rust, Scala, e Raku—a linguagem antes conhecida como Perl 6.^[20] Então podemos dizer que *traits* estão na moda em 2021.

Ruby traz uma perspectiva original para a herança múltipla: não a suporta, mas introduz mixins como um recurso explícito da linguagem. Uma classe Ruby pode incluir um módulo em seu corpo, e aí os métodos definidos no módulo se tornam parte da implementação da classe. Essa é uma forma "pura" de mixin, sem herança envolvida, e está claro que uma mixin em Ruby não influencia o tipo da classe onde que a utiliza. Isto oferece os benefícios das mixins, evitando muitos de seus problemas mais comuns.

Duas novas linguagens orientadas a objetos que estão recebendo muita atenção limitam severamente a herança: Go e Julia. Ambas giram em torno de programar "objetos" implementando "métodos", e suportam «polimorfismo» [fpy.li/7b], mas evitam o termo "classe".

Go não tem nenhum tipo de herança, mas oferece sintaxe para facilitar a composição em suas interfaces e structs. Julia tem uma hierarquia de tipos, mas subtipos não podem herdar estrutura, só comportamentos, e só é permitido criar subtipos de tipos abstratos. Além disso, os métodos de Julia são implementados com despacho múltiplo—uma forma mais avançada do mecanismo de despacho único que vimos na Seção 9.9.3.

[1] Alan Kay, "The Early History of Smalltalk" (*Os Primórdios de Smalltalk*), na SIGPLAN Not. 28, 3 (março de 1993), 69–95. Também disponível online [fpy.li/14-1]. Agradeço ao meu amigo Christiano Anderson por compartilhar essa referência quando eu estava escrevendo este capítulo.

[2] A docstring original estava errada, reportei no *issue #141721* [fpy.li/7e], enviei PR, e traduzi aqui.

[3] Adotamos o termo "receptor" como tradução para *receiver*, que é o objeto *x* vinculado um método *m* no momento da chamada *x.m()*.

[4] Também é possível passar apenas o primeiro argumento, mas isso não é útil e pode logo ser descontinuado, com as bênçãos de Guido van Rossum, o próprio criador de *super()*. Veja a discussão em *Is it time to deprecate unbound super methods?* [fpy.li/14-4] (Está na hora de descontinuar métodos "super" não vinculados?).

[5] É interessante observar que o C++ diferencia métodos virtuais e não-virtuais. Métodos virtuais têm vinculação tardia, enquanto os métodos não-virtuais são vinculados na compilação. Apesar de todos os métodos que podemos escrever em Python serem de vinculação tardia, como um método virtual, objetos embutidos escritos em C parecem ter métodos não-virtuais por default, pelo menos no CPython.

[6] Se você tiver curiosidade, o experimento está no arquivo *14-inheritance/strkeydict_dictsab.py* [fpy.li/14-7] do repositório *fluentpython/example-code-2e* [fpy.li/code].

[7] Aliás, nesse mesmo tópico, o PyPy se comporta mais "corretamente" que o CPython, às custas de introduzir uma pequena incompatibilidade. Veja os detalhes em "Differences between PyPy and CPython" (*Diferenças entre o PyPy e o CPython*) [fpy.li/14-5].

[8] Adotamos a convenção dos computólogos e desenhamos árvores de cabeça para baixo: a raiz no topo, as folhas na base.

[9] Classes também têm um método *.mro()*, mas este é um recurso avançado de programação de metaclasses, mencionado na «Seção 24.2» [fpy.li/7s] (vol.3). Durante o uso normal de uma classe, apenas o conteúdo do atributo *__mro__* importa.

[10] Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides, *Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos* (Bookman).

[11] Como já mencionado, o Java 8 permite que interfaces também forneçam implementações de métodos. Esse novo recurso é chamado *Default Methods* [fpy.li/14-12] (Métodos Default) no Tutorial oficial de Java.

[12] Os programadores Django sabem que o método de classe `as_view` é a parte mais visível da interface View, mas isso não é relevante para nós aqui.

[13] Se você gosta de padrões de projetos, note que o mecanismo de despacho do Django é uma variação dinâmica do padrão *Template Method* [fpy.li/75] (Método Template). Ele é dinâmico porque a classe View não obriga subclasses a implementarem todos os métodos de tratamento, mas `dispatch` verifica, durante a execução, se um método de tratamento concreto está disponível para cada requisição específica.

[14] NT: Literalmente "nos trilhos", mas claramente uma referência ao popular framework *Ruby on Rails*

[15] Esse princípio aparece na página 20 da introdução, na edição em inglês.

[16] Grady Booch et al., "Object-Oriented Analysis and Design with Applications" (*Análise e Projeto Orientados a Objetos, com Aplicações*), 3ª ed. (Addison-Wesley), p. 109.

[17] NT: a doutrina diz "Renderiza alguma lista de objetos, definida por `self.model` ou `self.queryset`. `self.queryset` na pode ser qualquer iterável de itens, não apenas um `queryset`."

[18] A PEP 591 também introduz uma anotação Final [fpy.li/7d] para variáveis e atributos que não devem ser reatribuídos ou sobrescritos.

[19] Alan Kay, *The Early History of Smalltalk* (Os Promórdios de Smalltalk), na SIGPLAN Not. 28, 3 (março de 1993), 69–95. Também disponível online [fpy.li/14-1]. Agradeço a meu amigo Cristiano Anderson, que compartilhou esta referência quando eu estava escrevendo esse capítulo)

[20] Meu amigo e revisor técnico Leonardo Rochael explica isso melhor do que eu poderia: "A existência continuada junto com o persistente adiamento da chegada do Perl 6 estava drenando a força de vontade da evolução do próprio Perl. Agora o Perl continua a ser desenvolvido como uma linguagem separada (está na versão 5.34), sem a ameaça de ser descontinuada pela linguagem antes conhecida como Perl 6."

Capítulo 15. Mais dicas de tipo

Aprendi uma dura lição: para programas pequenos, a tipagem dinâmica é ótima. Para programas grandes precisamos de uma abordagem mais disciplinada. E ajuda se a linguagem der a você aquela disciplina, ao invés de dizer "Bem, faça o que quiser".^[1]

— Guido van Rossum, um fã do Monty Python

Este capítulo é uma continuação do «Capítulo 8» [fpy.li/8] (vol.1), e fala mais sobre o sistema de tipagem gradual de Python. Os tópicos principais são:

- Assinaturas de funções sobrecarregadas
- `typing.TypedDict`: dando dicas de tipos para dicts usados como registros
- Coerção de tipo
- Acesso a dicas de tipo durante a execução
- Tipos genéricos
 - Declarando uma classe genérica
 - Variância: tipos invariantes, covariantes e contravariantes
 - Protocolos estáticos genéricos

15.1. Novidades neste capítulo

Este capítulo é inteiramente novo, escrito para essa segunda edição de *Python Fluente*. Vamos começar com sobrecargas.

15.2. Assinaturas sobrecarregadas

No Python, as funções podem aceitar diferentes combinações de argumentos.

O decorador `@typing.overload` permite anotar tais combinações. Isto é particularmente importante quando o tipo devolvido pela função depende do tipo de dois ou mais parâmetros.

Considere a função embutida `sum`. Esse é o texto de `help(sum)`, traduzido:

```
>>> help(sum)
sum(iterable, /, start=0)
    Devolve a soma de um valor 'start' (default: 0) mais a soma dos
    números de um iterável

    Quando o iterável é vazio, devolve o valor inicial ('start').
    Esta função é direcionada especificamente para uso com valores
    numéricos e pode rejeitar tipos não-numéricos.
```

A função embutida `sum` é escrita em C, mas o *typeshed* tem dicas de tipos sobrecarregadas para ela, em *builtins.pyi* [fpy.li/15-2]:

```
@overload
def sum(__iterable: Iterable[_T]) -> Union[_T, int]: ...
@overload
def sum(__iterable: Iterable[_T], start: _S) -> Union[_T, _S]: ...
```

Primeiro, vamos olhar a sintaxe geral das sobrecargas. Esse acima é todo o código sobre `sum` que encontrei no arquivo *stub* (*.pyi*). A implementação fica em um arquivo diferente. As reticências (...) não tem qualquer função além de cumprir a exigência sintática para um corpo de função, em vez usar de `pass`. Assim os arquivos *.pyi* são arquivos Python válidos. Como mencionado na «Seção 8.6» [fpy.li/7t] (vol.1), os dois sublinhados prefixando `__iterable` são a convenção da PEP 484 para argumentos apenas posicionais, que é checada pelo Mypy. Isso significa que você pode invocar `sum(my_list)`, mas não `sum(__iterable = my_list)`.

O chegador de tipos tenta fazer a correspondência entre os argumentos dados com cada assinatura sobrecarregada, em ordem. A chamada `sum(range(100), 1000)` não casa com a primeira sobrecarga, pois aquela assinatura tem apenas um parâmetro. Mas casa com a segunda.

Você pode também usar `@overload` em um módulo Python (*.py*) normal, colocando as assinaturas sobrecarregadas logo antes da assinatura real da função e de sua implementação. O Exemplo 1 mostra como a função `sum` apareceria anotada e implementada em um módulo Python.

Exemplo 1. mysum.py: definição da função sum com assinaturaas sobrecarregadas

```
import functools
import operator
from collections.abc import Iterable
from typing import overload, Union, TypeVar

T = TypeVar('T')
S = TypeVar('S') ①

@overload
def sum(it: Iterable[T]) -> Union[T, int]: ... ②
@overload
def sum(it: Iterable[T], /, start: S) -> Union[T, S]: ... ③
def sum(it, /, start=0): ④
    return functools.reduce(operator.add, it, start)
```

- ① Precisamos deste segundo TypeVar na segunda assinatura.
- ② Essa assinatura é para o caso simples: `sum(my_iterable)`. O tipo do resultado pode ser `T`—o tipo dos elementos que `my_iterable` produz—ou pode ser `int`, se o iterável for vazio, pois o valor default do parâmetro `start` é `0`.
- ③ Quando `start` é dado, ele pode ser de qualquer tipo `S`, então o tipo do resultado é `Union[T, S]`. É por isso que precisamos de `S`. Se `T` fosse reutilizado aqui, então o tipo de `start` teria que ser do mesmo tipo dos elementos de `Iterable[T]`.
- ④ A assinatura da implementação real da função não tem dicas de tipo.

São muitas linhas para anotar uma função de uma única linha. Sinto muito, mas pelo menos a função do exemplo não é `foo`.

Se quiser aprender sobre `@overload` lendo código, o *typeshed* tem centenas de exemplos. Quando escrevo esse capítulo, o arquivo stub `[fpy.li/15-3]` do *typeshed* para as funções embutidas de Python tem 186 sobrecargas—mais que qualquer outro na biblioteca padrão.



Aproveite a tipagem gradual

Tentar produzir código 100% anotado pode levar a dicas de tipo que acrescentam muito ruído e pouco valor agregado.

Refatoração para simplificar as dicas de tipo pode levar a APIs inconvenientes para quem vai usar. Algumas vezes é melhor ser pragmático, e deixar parte do código sem dicas de tipo.

As APIs convenientes e práticas que consideramos pythônicas são muitas vezes difíceis de anotar. Na próxima seção veremos um exemplo: são necessárias seis sobrecargas para anotar adequadamente a função embutida `max`, que é muito flexível nos parâmetros que aceita.

15.2.1. Sobrecarga máxima

É difícil acrescentar dicas de tipo a funções que usam os poderosos recursos dinâmicos de Python.

Quando estudava o `typeshed`, encontrei o relatório de bug #4051 [[fpy.li/shed4051](https://github.com/python/typeshed/issues/4051)]: Mypy não avisou que é proibido passar `None` como um dos argumentos para a função embutida `max()`, ou passar um iterável que em algum momento produz `None`. Nos dois casos, você recebe uma exceção como a seguinte durante a execução:

```
TypeError: '>' not supported between instances of 'int' and 'NoneType'
```

Tradução: `>` não é suportado entre instâncias de `'int'` e `'NoneType'`.

A documentação de `max` começa com a seguinte sentença:

Devolve o maior item em um iterável ou o maior de dois ou mais argumentos.

Para mim, essa é uma descrição bastante intuitiva.

Mas se eu for anotar uma função descrita nesses termos, tenho que decidir como declarar "um iterável" ou "dois ou mais argumentos". A realidade é ainda mais complicada, porque `max` também pode receber dois argumentos opcionais: `key` e `default`.

Escrevi `max` em Python para evidenciar a relação entre o funcionamento da função e as anotações sobrecarregadas (a função embutida original é escrita em C); veja o Exemplo 2.

Exemplo 2. mymax.py: Versão da função max em Python

```
# imports and definitions omitted, see next listing

MISSING = object()
EMPTY_MSG = 'max() arg is an empty sequence'

# overloaded type hints omitted, see next listing

def max(first, *args, key=None, default=MISSING):
    if args:
        series = args
        candidate = first
    else:
        series = iter(first)
        try:
            candidate = next(series)
        except StopIteration:
            if default is not MISSING:
                return default
            raise ValueError(EMPTY_MSG) from None
    if key is None:
        for current in series:
            if candidate < current:
                candidate = current
    else:
        candidate_key = key(candidate)
        for current in series:
            current_key = key(current)
            if candidate_key < current_key:
                candidate = current
                candidate_key = current_key
    return candidate
```

O foco deste exemplo não é a lógica de max, então não vou explicar a implementação, exceto para falar sobre MISSING. A constante MISSING é uma instância única de object, usada como sentinela. É o valor default para o argumento nomeado default=. A escolha de MISSING em vez de None como valor default para o argumento nomeado default permite que a função max detecte estas duas situações diferentes:

1. O usuário passou None como argumento default.
2. O usuário não passou o argumento default (neste caso seu valor fica sendo MISSING).

Quando first é um iterável vazio...

1. Se o usuário não forneceu um argumento para default=, então ele é MISSING, e max gera um ValueError.
2. Se usuário forneceu um valor para default=, incluindo None, e então max devolve o valor de default.

Para consertar o issue #4051 [fpy.li/shed4051], escrevi o código no Exemplo 3.^[2]

Exemplo 3. mymax.py: início do módulo, com importações, definições e sobrecargas

```
from collections.abc import Callable, Iterable
from typing import Protocol, Any, TypeVar, overload, Union

class SupportsLessThan(Protocol):
    def __lt__(self, other: Any) -> bool: ...

T = TypeVar('T')
LT = TypeVar('LT', bound=SupportsLessThan)
DT = TypeVar('DT')

MISSING = object()
EMPTY_MSG = 'max() arg is an empty sequence'

@overload
def max(__arg1: LT, __arg2: LT, *args: LT, key: None = ...) -> LT:
    ...
@overload
def max(__arg1: T, __arg2: T, *args: T, key: Callable[[T], LT]) -> T:
    ...
@overload
def max(__iterable: Iterable[LT], *, key: None = ...) -> LT:
    ...
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT]) -> T:
    ...
@overload
```

```
def max(__iterable: Iterable[LT], *, key: None = ...,
        default: DT) -> Union[LT, DT]:
    ...
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT],
        default: DT) -> Union[T, DT]:
    ...
```

Minha implementação de `max` em Python tem mais ou menos o mesmo tamanho daquelas importações e declarações de tipo. Graças à tipagem pato, meu código não tem nenhuma checagem usando `isinstance`, e fornece a mesma checagem de erro daquelas dicas de tipo—mas apenas durante a execução, claro.

Uma vantagem importante de `@overload` é declarar o tipo devolvido da forma mais precisa possível, de acordo com os tipos dos argumentos recebidos. Veremos esta vantagem a seguir, estudando as sobrecargas de `max`, em grupos de duas ou três por vez.

15.2.1.1. Argumentos implementando `SupportsLessThan`, sem `key` ou `default`

```
@overload
def max(__arg1: LT, __arg2: LT, *_args: LT, key: None = ...) -> LT:
    ...
# ... lines omitted ...
@overload
def max(__iterable: Iterable[LT], *, key: None = ...) -> LT:
    ...
```

Nestes casos, as entradas são ou argumentos separados do tipo `LT` que implementam `SupportsLessThan`, ou um `Iterable` de itens desse tipo. O tipo devolvido por `max` é do mesmo tipo dos argumentos ou itens reais, como vimos na «Seção 8.5.9.2» [fpy.li/7w] (vol.1).

Amostras de chamadas que casam com essas sobrecargas:

```
max(1, 2, -3) # returns 2
max(['Go', 'Python', 'Rust']) # returns 'Rust'
```

15.2.1.2. Argumento key fornecido, mas default não

```
@overload
def max(__arg1: T, __arg2: T, *_args: T, key: Callable[[T], LT]) -> T:
    ...
# ... lines omitted ...
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT]) -> T:
    ...
```

As entradas podem ser item separados de qualquer tipo `T` ou um único `Iterable[T]`, e `key` deve ser um invocável que recebe um argumento do mesmo tipo `T`, e devolve um valor que implementa `SupportsLessThan`. O tipo devolvido por `max` é o mesmo dos argumentos reais.

Amostras de chamadas que casam com essas sobrecargas:

```
max(1, 2, -3, key=abs) # returns -3
max(['Go', 'Python', 'Rust'], key=len) # returns 'Python'
```

15.2.1.3. Argumento default fornecido, key não

```
@overload
def max(__iterable: Iterable[LT], *, key: None = ...,
        default: DT) -> Union[LT, DT]:
    ...
```

A entrada é um iterável de itens do tipo `LT` que implemente `SupportsLessThan`. O argumento `default=` é o valor devolvido quando `Iterable` é vazio. Assim, o tipo devolvido por `max` deve ser uma `Union` do tipo `LT` e do tipo do argumento `default`.

Amostras de chamadas que casam com essas sobrecargas:

```
max([1, 2, -3], default=0) # returns 2
max([], default=None) # returns None
```

15.2.1.4. Argumentos key e default fornecidos

```
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT],
        default: DT) -> Union[T, DT]:
    ...
```

As entradas são:

- Um Iterable de itens de qualquer tipo T
- Invocável que recebe um argumento do tipo T e devolve um valor do tipo LT, que implementa SupportsLessThan
- Um valor default de qualquer tipo DT

O tipo devolvido por `max` deve ser uma Union do tipo T e do tipo do argumento default:

```
max([1, 2, -3], key=abs, default=None) # returns -3
max([], key=abs, default=None) # returns None
```

15.2.2. Lições da sobrecarga de max

Dicas de tipo permitem ao Mypy marcar uma chamada como `max([None, None])` com essa mensagem de erro:

```
mymax_demo.py:109: error: Value of type variable "_LT" of "max"
cannot be "None"
```

Por outro lado, escrever tantas linhas para suportar o checador de tipos pode desencorajar a criação de funções convenientes e flexíveis como `max`. Se eu precisasse reinventar também a função `min`, poderia refatorar e reutilizar a maior parte da implementação de `max`. Mas teria que copiar e colar todas as declarações de sobrecarga—apesar delas serem idênticas para `min`, exceto pelo nome da função.

Meu amigo João S. O. Bueno—um dos desenvolvedores Python mais talentosos que conheço—escreveu o seguinte tweet [fpy.li/15-4]:

Apesar de ser difícil expressar a assinatura de `max`—ela se encaixa muito facilmente em nossa estrutura mental. Considero a expressividade das marcas de anotação muito limitadas, se comparadas à de Python.

Vamos agora examinar o elemento de tipagem `TypedDict`. Ele não é tão útil quanto imaginei inicialmente, mas tem seus usos. Experimentar com `TypedDict` demonstra as limitações da tipagem estática para lidar com estruturas dinâmicas, como dados em formato JSON.

15.3. A anotação `TypedDict`



É tentador usar `TypedDict` para se proteger contra erros ao tratar estruturas de dados dinâmicas como as respostas da API JSON. Mas os exemplos aqui deixam claro que o tratamento correto de JSON precisa acontecer durante a execução, e não com checagem estática de tipo. Para checar estruturas similares a JSON usando dicas de tipo durante a execução, dê uma olhada no pacote *pydantic* [fpy.li/15-5] no PyPI.

Algumas vezes os dicionários de Python são usados como registros, as chaves interpretadas como nomes de campos e os valores como valores dos campos de diferentes tipos. Considere, por exemplo, um registro descrevendo um livro, em JSON ou Python:

```
{"isbn": "0134757599",  
 "title": "Refactoring, 2e",  
 "authors": ["Martin Fowler", "Kent Beck"],  
 "pagecount": 478}
```

Antes de Python 3.8, não havia uma boa maneira de anotar um registro como esse, pois os tipos de mapeamento que vimos na «Seção 8.5.6» [fpy.li/8c] (vol.1) limitam os valores a um mesmo tipo.

Aqui estão duas tentativas ruins de anotar um registro como o objeto JSON acima:

`dict[str, Any]`

As chaves são `str` mas os valores podem ser de qualquer tipo.

`dict[str, str|int|list[str]]`

Difícil de ler, e não preserva a relação entre os nomes dos campos e seus respectivos tipos: `title` deve ser uma `str`, ele não pode ser um `int` ou uma `List[str]`.

A PEP 589—*TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys* [fpy.li/pep589] (*TypedDict*: dicas de tipo para dicionários com um conjunto fixo de chaves) resolve este problema. O Exemplo 4 mostra um *TypedDict* simples.

Exemplo 4. books.py: a definição de BookDict

```
from typing import TypedDict

class BookDict(TypedDict):
    isbn: str
    title: str
    authors: list[str]
    pagecount: int
```

À primeira vista, `typing.TypedDict` pode parecer uma fábrica de classes de dados, similar a `typing.NamedTuple`—tratada no «Capítulo 5» [fpy.li/5] (vol.1).

A similaridade sintática é enganosa. *TypedDict* é muito diferente. Ele existe apenas para orientar um checador de tipos, e não tem qualquer efeito durante a execução.

TypedDict fornece duas coisas:

- Uma sintaxe similar à de classe para anotar um `dict` com dicas de tipo para os valores de cada campo identificado por um chaves.
- Um construtor que informa que o checador de tipos deve esperar um `dict` com chaves e valores como especificados.

Durante a execução, um construtor de TypedDict como BookDict é um placebo: ele tem o mesmo efeito de uma chamada ao construtor de dict com os mesmos argumentos.

O fato de BookDict criar um dict simples também significa que:

- Os "campos" na definição da pseudoclasse não criam atributos de instância.
- Não é possível escrever inicializadores com valores default para os "campos".
- Não é permitido definir métodos.

Vamos explorar o comportamento de um BookDict durante a execução (no Exemplo 5).

Exemplo 5. Usando um BookDict, mas não exatamente como planejado

```
>>> from books import BookDict
>>> pp = BookDict(title='Programming Pearls', ①
...               authors='Jon Bentley', ②
...               isbn='0201657880',
...               pagecount=256)
>>> pp ③
{'title': 'Programming Pearls', 'authors': 'Jon Bentley', 'isbn':
'0201657880',
 'pagecount': 256}
>>> type(pp)
<class 'dict'>
>>> pp.title ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'title'
>>> pp['title']
'Programming Pearls'
>>> BookDict.__annotations__ ⑤
{'isbn': <class 'str'>, 'title': <class 'str'>, 'authors':
typing.List[str],
 'pagecount': <class 'int'>}
```

- ① É possível invocar BookDict como um construtor de dict, com argumentos nomeados, ou passando um argumento dict—incluindo um literal dict.

- ② Ops... esqueci que `authors` deve ser uma lista. Mas não há checagem de tipos estáticos durante a execução.
- ③ O resultado da chamada a `BookDict` é um dict simples...
- ④ ...assim não é possível ler os campos usando a notação `objeto.campo`.
- ⑤ As dicas de tipo estão em `BookDict.__annotations__`, e não em `pp`.

Sem um checador de tipos, `TypedDict` é tão útil quanto comentários em um programa: pode ajudar a documentar o código, mas só isso. As fábricas de classes do «Capítulo 5» [fpy.li/5] (vol.1), por outro lado, são úteis mesmo se você não usar um checador de tipos, porque durante a execução elas geram uma classe customizada que pode ser instanciada. Elas também fornecem vários métodos ou funções úteis, listadas na «Seção 5.2.1» [fpy.li/8v] (vol.1).

O Exemplo 6 cria um `BookDict` válido e tenta executar algumas operações com ele. A seguir, o Exemplo 7 mostra como `TypedDict` permite que o Mypy encontre erros.

Exemplo 6. `demo_books.py`: operações legais e ilegais em um `BookDict`

```
from books import BookDict
from typing import TYPE_CHECKING

def demo() -> None: ①
    book = BookDict( ②
        isbn='0134757599',
        title='Refactoring, 2e',
        authors=['Martin Fowler', 'Kent Beck'],
        pagecount=478
    )
    authors = book['authors'] ③
    if TYPE_CHECKING: ④
        reveal_type(authors) ⑤
    authors = 'Bob' ⑥
    book['weight'] = 4.2
    del book['title']

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

- ① Lembre-se de adicionar o tipo devolvido, assim o Mypy não ignora a função.
- ② Este é um BookDict válido: todas as chaves estão presentes, com valores do tipo correto.
- ③ O Mypy vai inferir o tipo de authors a partir da anotação na chave 'authors' em BookDict.
- ④ `typing.TYPE_CHECKING` só é `True` quando os tipos no programa estão sendo checados. Durante a execução ele é sempre falso.
- ⑤ O `if` anterior evita que `reveal_type(authors)` seja chamado durante a execução. `reveal_type` não é uma função de Python disponível durante a execução, mas sim um instrumento de depuração fornecido pelo Mypy. Por isso não há um `import` para ela. Veja sua saída no Exemplo 7.
- ⑥ As últimas três linhas da função `demo` são ilegais. Elas vão disparar mensagens de erro no Exemplo 7.

Verificando a tipagem em *demo_books.py*, do Exemplo 6, obtemos o Exemplo 7.

Exemplo 7. Verificando os tipos em demo_books.py

```
.../typeddict/ $ mypy demo_books.py
demo_books.py:13: note: Revealed type is
                    'built-ins.list[built-ins.str]' ①
demo_books.py:14: error: Incompatible types in assignment
                    (expression has type "str",
                    variable has type "List[str]") ②
demo_books.py:15: error: TypedDict "BookDict" has no key 'weight' ③
demo_books.py:16: error: Key 'title' of TypedDict "BookDict"
                    cannot be deleted ④
Found 3 errors in 1 file (checked 1 source file)
```

- ① Esta observação é o resultado de `reveal_type(authors)`.
- ② O tipo da variável `authors` foi inferido a partir do tipo da expressão que a inicializou, `book['authors']`. Você não pode atribuir uma `str` para uma variável do tipo `List[str]`. Checadores de tipo em geral não permitem que o tipo de uma variável mude.^[3]
- ③ Não é permitido atribuir a uma chave que não é parte da definição de `BookDict`.

④ Não se pode apagar uma chave que é parte da definição de BookDict.

Vejamos agora BookDict sendo usado em assinaturas de função, para checar o tipo em chamadas de função.

Imagine que você precisa gerar XML a partir de registros de livros como esse:

```
<BOOK>
  <ISBN>0134757599</ISBN>
  <TITLE>Refactoring, 2e</TITLE>
  <AUTHOR>Martin Fowler</AUTHOR>
  <AUTHOR>Kent Beck</AUTHOR>
  <PAGECOUNT>478</PAGECOUNT>
</BOOK>
```

Se você estivesse escrevendo o código em MicroPython, para ser integrado a um pequeno microcontrolador, poderia escrever uma função parecida com o Exemplo 8.^[4]

Exemplo 8. books.py: a função to_xml

```
AUTHOR_ELEMENT = '<AUTHOR>{</AUTHOR>'

def to_xml(book: BookDict) -> str: ①
    elements: list[str] = [] ②
    for key, value in book.items():
        if isinstance(value, list): ③
            elements.extend(
                AUTHOR_ELEMENT.format(n) for n in value) ④
        else:
            tag = key.upper()
            elements.append(f'<{tag}>{value}</{tag}>')
    xml = '\n\t'.join(elements)
    return f'<BOOK>\n\t{xml}\n</BOOK>'
```

① O principal objetivo do exemplo: usar BookDict em uma assinatura de função.

② Se a coleção começa vazia, o Mypy não tem como inferir o tipo dos elementos. Por isso a anotação de tipo é necessária aqui.^[5]

- ③ O Mypy entende testes com `isinstance`, e trata `value` como uma `list` neste bloco.
- ④ Quando usei `key == 'authors'` como condição do `if` que guarda este bloco, o Mypy encontrou um erro nessa linha: `"object" has no attribute "__iter__"` ("*object*" não tem um atributo `"__iter__"`), porque inferiu o tipo de `value` devolvido por `book.items()` como `object`, que não suporta o método `__iter__` exigido pela expressão geradora. O teste com `isinstance` funciona porque garante que `value` é uma `list` neste bloco.

O Exemplo 9 mostra uma função que interpreta uma `str` JSON e devolve um `BookDict`.

Exemplo 9. books_any.py: a função from_json

```
def from_json(data: str) -> BookDict:
    whatever = json.loads(data) ①
    return whatever ②
```

- ① O tipo devolvido por `json.loads()` é `Any`.^[6]
- ② Posso devolver `whatever`—de tipo `Any`—porque `Any` é *consistente-com* todos os tipos, incluindo o tipo declarado do valor devolvido, `BookDict`.

É muito importante de ter em mente segundo ponto do Exemplo 9: o Mypy não vai apontar qualquer problema neste código, mas durante a execução o valor em `whatever` pode não se adequar à estrutura de `BookDict`—pode até não ser um `dict`!

Se você rodar o Mypy com `--disallow-any-expr`, ele vai reclamar sobre as duas linhas no corpo de `from_json`:

```
.../typeddict/ $ mypy books_any.py --disallow-any-expr
books_any.py:30: error: Expression has type "Any"
books_any.py:31: error: Expression has type "Any"
Found 2 errors in 1 file (checked 1 source file)
```

As linhas 30 e 31 mencionadas no trecho acima são o corpo da função `from_json`. Podemos silenciar o erro de tipo acrescentando uma dica de tipo à inicialização da variável `whatever`, como no Exemplo 10.

Exemplo 10. books.py: a função from_json com uma anotação de variável

```
def from_json(data: str) -> BookDict:
    whatever: BookDict = json.loads(data) ①
    return whatever ②
```

- ① --disallow-any-expr não gera erros quando uma expressão de tipo Any é imediatamente atribuída a uma variável com uma dica de tipo.
- ② Agora whatever é do tipo BookDict, o tipo declarado do valor devolvido.



Não se deixe enganar por uma falsa sensação de tipagem segura com o Exemplo 10! Olhando o código estático, o checador de tipos não tem como prever se `json.loads()` irá devolver qualquer coisa parecida com um `BookDict`. Apenas a validação durante a execução pode garantir isso.

A checagem de tipos estática é incapaz de prevenir erros cm código inerentemente dinâmico, como `json.loads()`, que cria objetos Python de tipos diferentes durante a execução. O Exemplo 11, o Exemplo 12 e o Exemplo 13 demonstram isso.

Exemplo 11. demo_not_book.py: from_json devolve um BookDict inválido, e to_xml o aceita

```
from books import to_xml, from_json
from typing import TYPE_CHECKING

def demo() -> None:
    NOT_BOOK_JSON = """
        {"title": "Andromeda Strain",
         "flavor": "pistachio",
         "authors": true}
    """

    not_book = from_json(NOT_BOOK_JSON) ①
    if TYPE_CHECKING: ②
        reveal_type(not_book)
        reveal_type(not_book['authors'])

    print(not_book) ③
```

```

    print(not_book['flavor']) ④

    xml = to_xml(not_book) ⑤
    print(xml) ⑥

if __name__ == '__main__':
    demo()

```

- ① Essa linha não produz um BookDict válido—veja o conteúdo de NOT_BOOK_JSON.
- ② Vamos deixar o Mypy revelar alguns tipos.
- ③ Isso não deve causar problemas: print consegue lidar com object e com qualquer outro tipo.
- ④ BookDict não tem uma chave 'flavor', mas o fonte JSON tem...o que acontecerá?
- ⑤ Lembre-se da assinatura: to_xml(book: BookDict) -> str:
- ⑥ Como será a saída em XML?

Agora checamos *demo_not_book.py* com o Mypy:

Exemplo 12. Relatório do Mypy para demo_not_book.py, reformatado por legibilidade

```

.../typeddict/ $ mypy demo_not_book.py
demo_not_book.py:12: note: Revealed type is
    'TypedDict('books.BookDict', {'isbn': built-ins.str,
                                'title': built-ins.str,
                                'authors': built-ins.list[built-
ins.str],
                                'pagecount': built-ins.int})' ①
demo_not_book.py:13: note: Revealed type is 'built-ins.list[built-
ins.str]' ②
demo_not_book.py:16: error: TypedDict "BookDict" has no key 'flavor' ③
Found 1 error in 1 file (checked 1 source file)

```

- ① O tipo revelado é o tipo estático, não o conteúdo de not_book durante a execução.

- ② De novo, este é o tipo estático de `not_book['authors']`, como definido em `BookDict`. Não o tipo durante a execução.
- ③ Este erro é para a linha `print(not_book['flavor'])`: esta chave não existe no tipo estático.

Agora vamos executar *demo_not_book.py*, mostrando o resultado no Exemplo 13.

Exemplo 13. Resultado da execução de demo_not_book.py

```
.../typeddict/ $ python3 demo_not_book.py
{'title': 'Andromeda Strain', 'flavor': 'pistachio', 'authors': True} ①
pistachio ②
<BOOK> ③
    <TITLE>Andromeda Strain</TITLE>
    <FLAVOR>pistachio</FLAVOR>
    <AUTHORS>True</AUTHORS>
</BOOK>
```

- ① Isso não é um `BookDict` de verdade.
- ② O valor de `not_book['flavor']`.
- ③ `to_xml` recebe um argumento `BookDict`, mas não há qualquer checagem durante a execução: entra lixo, sai lixo.

O Exemplo 13 mostra que *demo_not_book.py* devolve bobagens, mas não há qualquer erro durante a execução. Usar um `TypedDict` ao tratar dados em formato JSON não resultou em uma tipagem segura.

Olhando o código de `to_xml` no Exemplo 8 do ponto de vista da tipagem pato, o argumento `book` deve fornecer um método `.items()` que devolve um iterável de tuplas na forma `(chave, valor)`, onde:

- `chave` deve ter um método `.upper()`
- `valor` pode ser qualquer coisa.

A conclusão desta demonstração: quando estamos lidando com dados de estrutura dinâmica, tal como JSON ou XML, `TypedDict` não é, de forma alguma, um substituto para a validação de dados durante a execução. Para isso, use o *pydantic* [fpy.li/15-5].

TypedDict tem mais recursos, incluindo suporte a chaves opcionais, uma forma limitada de herança e uma sintaxe de declaração alternativa. Para saber mais sobre ele, estude a *PEP 589—TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys* [fpy.li/pep589] (TypedDict: dicas de tipo para dicionários com um conjunto fixo de chaves).

Vamos agora voltar nossa atenção para uma função que é melhor evitar, mas que algumas vezes é inevitável: `typing.cast`.

15.4. Coerção de tipo (*type casting*)

Nenhum sistema de tipos é perfeito, nem tampouco os checadores estáticos de tipo, as dicas de tipo no projeto *typeshed* ou as dicas de tipo em pacotes de terceiros, quando existem. A função especial `typing.cast()` é uma forma de lidar com defeitos ou incorreções nas dicas de tipo em código que não podemos consertar. A documentação do Mypy 0.930 [fpy.li/15-14] explica (tradução nossa):

Coerções são usadas para silenciar avisos espúrios do checador de tipos, e ajudam o checador quando ele não consegue entender o que está acontecendo.

Durante a execução, `typing.cast` não faz absolutamente nada. Esta é sua implementação [fpy.li/15-15]:

```
def cast(typ, val):
    """Cast a value to a type.
    This returns the value unchanged. To the type checker this
    signals that the return value has the designated type, but at
    runtime we intentionally don't check anything (we want this
    to be as fast as possible).
    """
    return val
```

A docstring diz: "Coage um valor para um tipo. Isto devolve o valor inalterado. Para o checador de tipos, isto sinaliza que o valor devolvido tem o tipo designado, mas na execução não fazemos nenhuma checagem (queremos que isto seja tão rápido quanto possível)".

A PEP 484 exige que os checadores de tipos "acreditem cegamente" em `cast`. A seção "Casts" (*Coerções*) da PEP 484 [fpy.li/15-16] mostra um exemplo onde o checador precisa da orientação de `cast`:

```
from typing import cast

def find_first_str(a: list[object]) -> str:
    index = next(i for i, x in enumerate(a) if isinstance(x, str))
    # We only get here if there's at least one string
    return cast(str, a[index])
```

A chamada `next()` na expressão geradora vai devolver o índice de um item `str` ou levantar `StopIteration`. Assim, `find_first_str` vai sempre devolver uma `str` se não for gerada uma exceção, e `str` é o tipo declarado do valor devolvido.

Mas se a última linha for apenas `return a[index]`, o Mypy inferiria o tipo devolvido como `object`, porque o argumento `a` é declarado como `list[object]`. Então `cast()` é necessário para orientar o Mypy.^[7]

Aqui está outro exemplo com `cast`, desta vez para corrigir uma dica de tipo desatualizada na biblioteca padrão de Python. No «Exemplo 12 do Capítulo 21» [fpy.li/94] (vol.3), criei um objeto `asyncio.Server`, e queria obter o endereço onde o servidor está ouvindo (aceitando conexões). Escrevi esta linha de código:

```
addr = server.sockets[0].getsockname()
```

Mas o Mypy informou o seguinte erro:

```
Value of type "Optional[List[socket]]" is not indexable
```

A dica de tipo para `Server.sockets` no *typeshed*, em maio de 2021, é válida para Python 3.6, onde o atributo `sockets` podia ser `None`. Mas no Python 3.7, `sockets` se tornou uma propriedade, com um *getter* que sempre devolve uma `list`—que pode ser vazia, se o servidor não tiver um *socket*. E desde o Python 3.8, esse *getter* devolve uma `tuple` (usada como uma sequência imutável).

Já que não posso consertar o *typeshed* nesse instante,^[8] acrescentei um `cast`, assim:

```
from asyncio.trsock import TransportSocket
from typing import cast

# ... muitas linhas omitidas ...

socket_list = cast(tuple[TransportSocket, ...], server.sockets)
addr = socket_list[0].getsockname()
```

Usar `cast` nesse caso exigiu algumas horas para entender o problema e ler o código-fonte de `asyncio`, para encontrar o tipo correto para `sockets`: a classe `TransportSocket` do módulo não-documentado `asyncio.trsock`. Também precisei adicionar duas instruções `import` e mais uma linha de código para melhorar a legibilidade.^[9] Mas agora o código está mais seguro.

A leitora atenta pode ter notado que `sockets[0]` poderia gerar um `IndexError` se `sockets` estiver vazio. Entretanto, até onde entendo o `asyncio`, isso não pode acontecer no «Exemplo 12 do Capítulo 21» [fpy.li/94] (vol.3), pois no momento em que leio o atributo `sockets`, o `server` já está pronto para aceitar conexões, portanto o atributo não estará vazio. E, de qualquer forma, `IndexError` ocorre durante a execução. O `Mypy` não consegue localizar esse problema nem mesmo em um caso trivial como `print([][0])`.



Não se acostume a usar `cast` para silenciar o `Mypy` toda hora, porque normalmente o `Mypy` está certo quando aponta um erro. Se você estiver aplicando `cast` com frequência, isso é um *code smell* [fpy.li/15-20] (cheiro no código). Sua equipe pode estar fazendo um mau uso das dicas de tipo, ou sua base de código pode ter dependências de baixa qualidade.

Apesar de suas desvantagens, há usos válidos para `cast`. Eis algo que Guido van Rossum escreveu sobre isso:

O que há de errado com uma eventual chamada a `cast()` ou um comentário
`# type: ignore ?[10]`

É insensato banir inteiramente o uso de `cast`, principalmente porque as alternativas para contornar esses problemas são piores:

- `# type: ignore` é menos informativo.^[11]
- Usar `Any` é contagioso: já que `Any` é *consistente-com* todos os tipos, seu abuso pode produzir efeitos em cascata através da inferência de tipo, minando a capacidade do checador de tipos para detectar erros em outras partes do código.

Claro, nem todos os contratempos de tipagem podem ser resolvidos com `cast`. Algumas vezes precisamos de `# type: ignore`, do `Any` aqui ou ali, ou mesmo deixar uma função sem dicas de tipo.

A seguir, vamos falar sobre o uso de anotações durante a execução.

15.5. Lendo dicas de tipo durante a execução

Durante a importação, Python lê as dicas de tipo em funções, classes e módulos, e as armazena em atributos chamados `__annotations__`. Considere, por exemplo, a função `clip` no Exemplo 14.^[12]

Exemplo 14. `clipannot.py`: a assinatura anotada da função `clip`

```
def clip(text: str, max_len: int = 80) -> str:
```

As dicas de tipo são armazenadas em um dict no atributo `__annotations__` da função:

```
>>> from clip_annot import clip
>>> clip.__annotations__
{'text': <class 'str'>, 'max_len': <class 'int'>, 'return': <class 'str'>}
```

A chave `'return'` está mapeada para a dica do tipo devolvido após o símbolo `->` no Exemplo 14.

Note que as anotações são avaliadas pelo interpretador no momento da importação, ao mesmo tempo em que os valores default dos parâmetros são avaliados. Por isso os valores nas anotações são as classes Python `str` e `int`, e não as strings `'str'` e `'int'`. A avaliação das anotações no momento da importação é o padrão desde o Python 3.10, mas isso pode mudar se a PEP 563 [fpy.li/pep563] ou a PEP 649 [fpy.li/pep649] se tornarem o comportamento padrão.

15.5.1. Problemas com anotações durante a execução

O aumento do uso de dicas de tipo gerou dois problemas:

- Importar módulos usa mais CPU e memória quando são há dicas de tipo.
- Referências a tipos ainda não definidos exigem o uso de strings em vez dos tipos reais.

As duas questões são relevantes. A primeira pelo que acabamos de ver: anotações são avaliadas pelo interpretador durante a importação e armazenadas no atributo `__annotations__`. Quando uma empresa tem milhares de servidores importante arquivos Python, o custo pode ser significativo, mesmo considerando que a importação de cada módulo só acontece no início do processo.

Vamos nos concentrar agora no segundo problema.

Armazenar anotações como string é necessário algumas vezes, por causa do problema da referência futura (*forward reference*): quando uma dica de tipo precisa se referir a uma classe definida mais adiante no mesmo módulo. Entretanto uma manifestação comum desse problema no código-fonte não se parece de forma alguma com uma referência futura: quando um método devolve um novo objeto da mesma classe. Já que o objeto classe não está definido até Python terminar a avaliação do corpo da classe, as dicas de tipo precisam usar o nome da classe como string. Eis um exemplo:

```
class Rectangle:
    # ... lines omitted ...
    def stretch(self, factor: float) -> 'Rectangle':
        return Rectangle(width=self.width * factor)
```

Escrever dicas de tipo com referências futuras como strings é a prática padrão e exigida no Python 3.10. Os checadores de tipos estáticos foram projetados desde o início para lidar com esse problema.

Mas durante a execução, se você escrever código para ler a anotação `return de stretch`, vai receber a string `'Rectangle'` em vez de uma referência ao tipo real, a classe `Rectangle`. E aí seu código precisa descobrir o que aquela string significa.

O módulo `typing` inclui três funções e uma classe categorizadas como Auxiliares de introspecção [fpy.li/7h], sendo `typing.get_type_hints` a mais importante delas. Parte de sua documentação afirma:

`get_type_hints(obj, globals=None, locals=None, include_extras=False)`

[...] Isso é muitas vezes igual a `obj.__annotations__`. Além disso, referências futuras codificadas como strings literais são tratadas por sua avaliação nos espaços de nomes `globals` e `locals`. [...]



Desde o Python 3.10, a nova função `inspect.get_annotations(...)` [fpy.li/15-25] deve ser usada, em vez de `get_type_hints`. Entretanto, alguns leitores podem ainda não estar trabalhando com Python 3.10, então usarei `get_type_hints` nos exemplos, pois essa função está disponível desde a inclusão do módulo `typing`, no Python 3.5.

A *PEP 563—Postponed Evaluation of Annotations* [fpy.li/pep563] (Avaliação Adiada de Anotações) foi aprovada para tornar desnecessário escrever anotações como strings, e para reduzir o custo das dicas de tipo durante a execução. A ideia principal está descrita nessas duas sentenças do *Abstract* [fpy.li/15-26]:

Esta PEP propõe modificar as anotações de funções e de variáveis, de forma que elas não mais sejam avaliadas no momento da definição da função. Em vez disso, elas são preservadas em `__annotations__` na forma de strings.

A partir de Python 3.7, é assim que anotações são tratadas em qualquer módulo que comece com a seguinte instrução `import`:

```
from __future__ import annotations
```

Para demonstrar seu efeito, coloquei a mesma função `clip` do Exemplo 14 em um módulo `clip_annot_post.py` com aquela linha de importação `__future__` no início.

No console, esse é o resultado de importar aquele módulo e ler as anotações de `clip`:

```
>>> from clip_annot_post import clip
>>> clip.__annotations__
{'text': 'str', 'max_len': 'int', 'return': 'str'}
```

Como se vê, todas as dicas de tipo são agora strings simples, apesar de não terem sido escritas como strings na definição de `clip` (no Exemplo 14).

A função `typing.get_type_hints` consegue resolver muitas dicas de tipo, incluindo essas de `clip`:

```
>>> from clip_annot_post import clip
>>> from typing import get_type_hints
>>> get_type_hints(clip)
{'text': <class 'str'>, 'max_len': <class 'int'>, 'return': <class 'str'>}
```

A chamada a `get_type_hints` nos dá os tipos reais—mesmo em alguns casos onde a dica de tipo original foi escrita como uma string. Esta é a maneira recomendada de ler dicas de tipo durante a execução.

O comportamento prescrito na PEP 563 estava previsto para se tornar o default no Python 3.10, tornando a importação com `__future__` desnecessária. Entretanto, os mantenedores da *FastAPI* e do *pydantic* avisaram de que tal mudança quebraria seu código, que se baseia em dicas de tipo durante a execução e não podem usar `get_type_hints` de forma confiável.

Na discussão que se seguiu na lista de e-mail `python-dev`, Łukasz Langa—autor da PEP 563—descreveu algumas limitações daquela função:

[...] a verdade é que `typing.get_type_hints()` tem limites que tornam seu uso geral custoso durante a execução e, mais importante, insuficiente para resolver todos os tipos. O exemplo mais comum se refere a contextos não-

globais nos quais tipos são gerados (isto é, classes aninhadas, classes dentro de funções, etc.). Mas um dos principais exemplos de referências futuras, classes com métodos aceitando ou devolvendo objetos de seu próprio tipo, também não é tratado de forma apropriada por `typing.get_type_hints()` se um gerador de classes for usado. Há alguns truques que podemos usar para ligar os pontos mas, de uma forma geral, isso não é bom.^[13]

O Steering Council de Python decidiu adiar a elevação da PEP 563 a comportamento padrão até Python 3.11 ou posterior, dando mais tempo aos desenvolvedores para criar uma solução para os problemas que a PEP 563 tentou resolver, sem quebrar o uso disseminado das dicas de tipo durante a execução. A PEP 649—*Deferred Evaluation Of Annotations Using Descriptors* [fpy.li/pep649] (Avaliação adiada de anotações usando descritores) está sendo considerada como uma possível solução, mas algum outro acordo ainda pode ser alcançado.

Resumindo: ler dicas de tipo durante a execução não é 100% confiável no Python 3.10 e provavelmente mudará em alguma futura versão.



Empresas usando Python em escala muito grande desejam os benefícios da tipagem estática, mas não querem pagar o preço da avaliação de dicas de tipo no momento da importação. A checagem estática acontece nas estações de trabalho dos desenvolvedores e em servidores de integração contínua dedicados, mas o carregamento de módulos acontece com uma frequência e um volume muito maiores, em servidores de produção, e este custo não é desprezível em grande escala.

Isto cria uma tensão na comunidade Python, entre aqueles que querem as dicas de tipo armazenadas apenas como strings—para reduzir os custos de carregamento—versus aqueles que também querem usar as dicas de tipo durante a execução, como os criadores e os usuários do *pydantic* e da *FastAPI*, para quem seria mais fácil acessar diretamente os tipos, ao invés de precisarem analisar strings nas anotações, uma tarefa complicada.

15.5.2. Lidando com o problema

Dada a instabilidade da situação atual, se você precisar ler anotações durante a execução, recomendo o seguinte:

- Evite ler `__annotations__` diretamente; em vez disso, use `inspect.get_annotations` (desde o Python 3.10) ou `typing.get_type_hints` (desde o Python 3.5).
- Escreva uma função customizada própria, como um invólucro para `inspect.get_annotations` ou `typing.get_type_hints`, e faça o restante de sua base de código chamar aquela função, de forma que mudanças futuras fiquem restritas a um único local.

Para demonstrar esse segundo ponto, aqui estão as primeiras linhas da classe `Checked`, que estudaremos no «Exemplo 5 do Capítulo 24» [fpy.li/95] (vol.3):

```
class Checked:
    @classmethod
    def _fields(cls) -> dict[str, type]:
        return get_type_hints(cls)
```

O método de `Checked._fields` evita que outras partes do módulo dependam diretamente de `typing.get_type_hints`. Se `get_type_hints` mudar no futuro, exigindo lógica adicional, ou se eu quiser substituí-la por `inspect.get_annotations`, a mudança estará limitada a `Checked._fields` e não afetará o restante do programa.



Dadas as discussões atuais e as mudanças propostas para a inspeção de dicas de tipo durante a execução, a página da documentação oficial «Boas práticas para anotações» [fpy.li/7j] é leitura obrigatória, e deverá ser atualizada até o lançamento do Python 3.11. Aquele *how-to* foi escrito por Larry Hastings, autor da PEP 649—*Deferred Evaluation Of Annotations Using Descriptors* [fpy.li/pep649] (Avaliação Adiada de Anotações Usando Descritores), uma alternativa para tratar os problemas da PEP 563—*Postponed Evaluation of Annotations* [fpy.li/pep563] (Avaliação Adiada de Anotações).

As demais seções deste capítulo cobrem tipos genéricos, começando pela forma de definir uma classe genérica, que pode ser parametrizada por seus usuários.

15.6. Implementando uma classe genérica

No Exemplo 7 do Capítulo 13, definimos a ABC Tombola: uma interface para classes que funcionam como um recipiente para sorteio de bingo. A classe `LottoBlower` (Exemplo 11 do Capítulo 13) é uma implementação concreta. Vamos agora estudar uma versão genérica de `LottoBlower`, usada da forma que aparece no Exemplo 15.

Exemplo 15. `generic_lotto_demo.py`: usando uma classe genérica de sorteio de bingo

```
from generic_lotto import LottoBlower

machine = LottoBlower[int](range(1, 11)) ①

first = machine.pick() ②
remain = machine.inspect() ③
```

- ① Para instanciar uma classe genérica, passamos a ela um parâmetro de tipo concreto, como `int` aqui.
- ② O Mypy irá inferir corretamente que `first` é um `int`...
- ③ ... e que `remain` é uma tuple de inteiros.

Além disso, o Mypy aponta violações do tipo parametrizado com mensagens úteis, como ilustrado no Exemplo 16.

Exemplo 16. `generic_lotto_errors.py`: erros apontados pelo Mypy

```
from generic_lotto import LottoBlower

machine = LottoBlower[int]([1, .2])
## error: List item 1 has incompatible type "float"; ①
##         expected "int"

machine = LottoBlower[int](range(1, 11))

machine.load('ABC')
```

```

## error: Argument 1 to "load" of "LottoBlower" ②
##     has incompatible type "str";
##     expected "Iterable[int]"
## note: Following member(s) of "str" have conflicts:
## note:     Expected:
## note:         def __iter__(self) -> Iterator[int]
## note:     Got:
## note:         def __iter__(self) -> Iterator[str]

```

- ① Na instanciação de `LottoBlower[int]`, o Mypy marca o float.
- ② Na chamada `.load('ABC')`, o Mypy explica porque uma `str` não serve: `str.__iter__` devolve um `Iterator[str]`, mas `LottoBlower[int]` exige um `Iterator[int]`.

O Exemplo 17 é a implementação.

Exemplo 17. generic_lotto.py: uma classe genérica de sorteador de bingo

```

import random

from collections.abc import Iterable
from typing import TypeVar, Generic

from tombola import Tombola

T = TypeVar('T')

class LottoBlower(Tombola, Generic[T]): ①

    def __init__(self, items: Iterable[T]) -> None: ②
        self._balls = list[T](items)

    def load(self, items: Iterable[T]) -> None: ③
        self._balls.extend(items)

    def pick(self) -> T: ④
        try:
            position = random.randrange(len(self._balls))
        except ValueError:
            raise LookupError('pick from empty LottoBlower')
        return self._balls.pop(position)

```

```
def loaded(self) -> bool: ⑤
    return bool(self._balls)

def inspect(self) -> tuple[T, ...]: ⑥
    return tuple(self._balls)
```

- ① Declarações de classes genéricas muitas vezes usam herança múltipla, porque precisamos incluir a superclasse `Generic` para declarar os parâmetros de tipo formais—neste caso, `T`.
- ② O argumento `items` em `__init__` é do tipo `Iterable[T]`, que se torna `Iterable[int]` quando uma instância é declarada como `LottoBlower[int]`.
- ③ O método `load` é igualmente anotado.
- ④ O tipo do valor devolvido `T` agora se torna `int` em um `LottoBlower[int]`.
- ⑤ Nenhuma variável de tipo aqui.
- ⑥ Por fim, `T` define o tipo dos itens na `tuple` devolvida.



A seção *User-defined generic types* [fpy.li/7k] (Tipos genéricos definidos pelo usuário), na documentação do módulo `typing`, é curta, inclui bons exemplos e fornece alguns detalhes que não menciono aqui.

Agora que vimos como implementar um classe genérica, vamos definir a terminologia para falar sobre tipos genéricos.

15.6.1. Jargão básico para tipos genéricos

Aqui estão algumas definições que encontrei estudando genéricos.^[14]

Tipo genérico

Um tipo declarado com uma ou mais variáveis de tipo.

Exemplos: `LottoBlower[T]`, `abc.Mapping[KT, VT]`

Parâmetro de tipo formal

As variáveis de tipo que aparecem em um declaração de tipo genérica.

Exemplo: `KT` e `VT` no último exemplo: `abc.Mapping[KT, VT]`

Tipo parametrizado

Um tipo declarado com os parâmetros de tipo reais.

Exemplos: `LottoBlower[int]`, `abc.Mapping[str, float]`

Parâmetro de tipo real

Os tipos reais passados como parâmetros quando um tipo parametrizado é declarado.

Exemplo: o `int` em `LottoBlower[int]`

O próximo tópico é sobre como tornar os tipos genéricos mais flexíveis, introduzindo os conceitos de covariância, contravariância e invariância.

15.7. Variância

Dependendo de sua experiência com genéricos em outras linguagens, esta pode ser a seção mais difícil do livro. O conceito de variância é abstrato, e uma apresentação rigorosa faria essa seção se parecer com páginas de um livro de matemática.



Na prática, a variância é mais relevante para autores de bibliotecas que querem suportar novos tipos de coleções genéricas ou fornecer uma API baseada em *callbacks*. Mesmo nestes casos, é possível evitar muita complexidade suportando apenas coleções invariantes—que é o que temos hoje na biblioteca padrão. Então, em uma primeira leitura você pode pular toda esta seção, ou ler apenas as partes sobre tipos invariantes.

Já vimos o conceito de *variância* na «Seção 8.5.11.1» [fpy.li/7y] (vol.1), aplicado a tipos genéricos `Callable` parametrizados. Aqui vamos expandir o conceito para abarcar tipos genéricos de coleções, usando uma analogia do "mundo real" para tornar mais concreto esse conceito abstrato.

Imagine uma cantina escolar que tenha como regra que apenas máquinas servindo sucos podem ser instaladas.^[15] Máquinas de bebida genéricas não são permitidas, pois podem servir refrigerantes, que foram banidos pela direção da escola.^[16]

15.7.1. Uma máquina de bebida invariante

Vamos tentar modelar o cenário da cantina com uma classe genérica `BeverageDispenser`, que pode ser parametrizada com o tipo de bebida.. Veja o Exemplo 18.

Exemplo 18. `invariant.py`: definições de tipo e função `install`

```
from typing import TypeVar, Generic

class Beverage: ①
    """Any beverage."""

class Juice(Beverage):
    """Any fruit juice."""

class OrangeJuice(Juice):
    """Delicious juice from Brazilian oranges."""

T = TypeVar('T') ②

class BeverageDispenser(Generic[T]): ③
    """A dispenser parameterized on the beverage type."""
    def __init__(self, beverage: T) -> None:
        self.beverage = beverage

    def dispense(self) -> T:
        return self.beverage

def install(dispenser: BeverageDispenser[Juice]) -> None: ④
    """Install a fruit juice dispenser."""
```

- ① `Beverage`, `Juice`, e `OrangeJuice` formam uma hierarquia de tipos.
- ② Uma declaração `TypeVar` simples.
- ③ `BeverageDispenser` é parametrizada pelo tipo de bebida.
- ④ `install` é uma função global do módulo. Sua dica de tipo faz valer a regra de que apenas máquinas de suco são aceitáveis.

Dadas as definições no Exemplo 18, o seguinte código é válido:

```
juice_dispenser = BeverageDispenser(Juice())
install(juice_dispenser)
```

Entretanto, isto não é válido:

```
beverage_dispenser = BeverageDispenser(Beverage())
install(beverage_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[Beverage]"
##           expected "BeverageDispenser[Juice]"
```

Uma máquina que serve qualquer Beverage não é aceitável, pois a cantina exige uma máquina especializada em Juice.

De forma um tanto surpreendente, este código também é inválido:

```
orange_juice_dispenser = BeverageDispenser(OrangeJuice())
install(orange_juice_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[OrangeJuice]"
##           expected "BeverageDispenser[Juice]"
```

Uma máquina especializada em OrangeJuice também não é permitida. Apenas BeverageDispenser[Juice] serve. No jargão da tipagem, dizemos que BeverageDispenser[Generic[T]] é invariante quando BeverageDispenser[OrangeJuice] não é compatível com BeverageDispenser[Juice]—apesar do fato de OrangeJuice ser um *subtipo-de* Juice.

Os tipos de coleções mutáveis de Python—tal como list e set—são invariantes. A classe LottoBlower do Exemplo 17 também é invariante.

15.7.2. Uma máquina de bebida covariante

Se quisermos ser mais flexíveis, e modelar as máquinas de bebida como uma classe genérica que aceite alguma bebida e também seus subtipos, precisamos tornar a classe covariante. O Exemplo 19 mostra como declararíamos BeverageDispenser.

Exemplo 19. covariant.py: definições de tipo e função install

```
T_co = TypeVar('T_co', covariant=True) ①

class BeverageDispenser(Generic[T_co]): ②
    def __init__(self, beverage: T_co) -> None:
        self.beverage = beverage

    def dispense(self) -> T_co:
        return self.beverage

def install(dispenser: BeverageDispenser[Juice]) -> None: ③
    """Install a fruit juice dispenser."""
```

- ① Define `covariant=True` ao declarar a variável de tipo; `_co` é o sufixo convencional para parâmetros de tipo covariantes no *typed*.
- ② Usa `T_co` para parametrizar a classe especial `Generic`.
- ③ As dicas de tipo para `install` são as mesmas do Exemplo 18.

O código abaixo funciona porque tanto a máquina de Juice quanto a de OrangeJuice são válidas em uma `BeverageDispenser` covariante:

```
juice_dispenser = BeverageDispenser(Juice())
install(juice_dispenser)
orange_juice_dispenser = BeverageDispenser(OrangeJuice())
install(orange_juice_dispenser)
```

mas uma máquina de uma `Beverage` arbitrária não é aceitável:

```
beverage_dispenser = BeverageDispenser(Beverage())
install(beverage_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[Beverage]"
##         expected "BeverageDispenser[Juice]"
```

Isso é uma covariância: a relação de subtipo das máquinas parametrizadas varia na mesma direção da relação de subtipo dos parâmetros de tipo.

15.7.3. Uma lata de lixo contravariante

Vamos agora modelar a regra da cantina para a instalação de uma lata de lixo. Vamos supor que a comida e a bebida são servidas em recipientes biodegradáveis, e as sobras e utensílios descartáveis também são biodegradáveis. As latas de lixo devem ser adequadas para resíduos biodegradáveis.



Neste exemplo didático, vamos fazer algumas suposições e classificar o lixo em uma hierarquia simplificada:

- **Refuse** (*Resíduo*) é o tipo mais geral de lixo. Todo lixo é resíduo.
- **Biodegradable** (*Biodegradável*) é um tipo de lixo decomposto por microrganismos ao longo do tempo. Parte do **Refuse** não é **Biodegradable**.
- **Compostable** (*Compostável*) é um tipo específico de lixo **Biodegradable** que pode ser transformado de em fertilizante orgânico, em um processo de compostagem. Na nossa definição, nem todo lixo **Biodegradable** é **Compostable**.

Para modelar a regra descrevendo uma lata de lixo aceitável na cantina, precisamos introduzir o conceito de "contravariância" através de um exemplo, apresentado no Exemplo 20.

Exemplo 20. contravariant.py: definições de tipo e a função install

```
from typing import TypeVar, Generic

class Refuse: ①
    """Any refuse."""

class Biodegradable(Refuse):
    """Biodegradable refuse."""

class Compostable(Biodegradable):
    """Compostable refuse."""

T_contra = TypeVar('T_contra', contravariant=True) ②
```

```
class TrashCan(Generic[T_contra]): ③
    def put(self, refuse: T_contra) -> None:
        """Store trash until dumped."""

    def deploy(trash_can: TrashCan[Biodegradable]): ④
        """Deploy a trash can for biodegradable refuse."""
```

- ① Uma hierarquia de tipos para resíduos: Refuse é o tipo mais geral, Compostable o mais específico.
- ② T_contra é o nome convencional para uma variável de tipo contravariante.
- ③ TrashCan é contravariante ao tipo de resíduo.
- ④ A função deploy exige uma lata de lixo compatível com TrashCan[Biodegradable].

Dadas essas definições, os seguintes tipos de lata de lixo são aceitáveis:

```
bio_can: TrashCan[Biodegradable] = TrashCan()
deploy(bio_can)

trash_can: TrashCan[Refuse] = TrashCan()
deploy(trash_can)
```

A função deploy aceita uma TrashCan[Refuse], pois ela pode receber qualquer tipo de resíduo, incluindo Biodegradable. Entretanto, uma TrashCan[Compostable] não serve, pois ela não pode receber Biodegradable:

```
compost_can: TrashCan[Compostable] = TrashCan()
deploy(compost_can)
## mypy: Argument 1 to "deploy" has
## incompatible type "TrashCan[Compostable]"
##         expected "TrashCan[Biodegradable]"
```

Vamos resumir os conceitos vistos até aqui.

15.7.4. Revisão da variância

A variância é uma propriedade sutil. As próximas seções recapitulam o conceito de tipos invariantes, covariantes e contravariantes, e fornecem algumas regras gerais para pensar sobre eles.

15.7.4.1. Tipos invariantes

Um tipo genérico L é invariante quando não há nenhuma relação de supertipo ou subtipo entre dois tipos parametrizados, independente da relação que possa existir entre os parâmetros concretos. Em outras palavras, se L é invariante, então $L[A]$ não é supertipo ou subtipo de $L[B]$. Eles são inconsistentes em ambos os sentidos.

As coleções mutáveis da biblioteca padrão de Python são invariantes. O tipo `list` é um bom exemplo: `list[int]` não é *consistente-com* `list[float]`, e vice-versa.

Em geral, se um parâmetro de tipo formal aparece em dicas de tipo de argumentos a métodos, e o mesmo parâmetro aparece nos tipos devolvidos pelo método, aquele parâmetro deve ser invariante, para garantir a segurança de tipo na atualização e leitura da coleção.

Por exemplo, aqui está parte das dicas de tipo para o tipo embutido `list` no *typeshed* [fpy.li/15-30]:

```
class list(MutableSequence[_T], Generic[_T]):
    @overload
    def __init__(self) -> None: ...
    @overload
    def __init__(self, iterable: Iterable[_T]) -> None: ...
    # ... lines omitted ...
    def append(self, __object: _T) -> None: ...
    def extend(self, __iterable: Iterable[_T]) -> None: ...
    def pop(self, __index: int = ...) -> _T: ...
    # etc...
```

Veja que `_T` aparece entre os parâmetros de `__init__`, `append` e `extend`, e como tipo devolvido por `pop`. Não há como tornar segura a tipagem dessa classe se ela for covariante ou contravariante em `_T`.

15.7.4.2. Tipos covariantes

Considere dois tipos A e B, onde B é *consistente-com* A, e nenhum deles é Any. Alguns autores usam os símbolos <: e :> para indicar relações de tipos como essas:

A :> B

A é um *supertipo-de* ou igual a B.

B <: A

B é um *subtipo-de* ou igual a A.

Dado A :> B, um tipo genérico C é covariante quando C[A] :> C[B].

Observe que a direção da seta no símbolo :> é a mesma nos dois casos em que A está à esquerda de B. Tipos genéricos covariantes seguem a relação de subtipo do tipo real dos parâmetros.

Contêineres imutáveis podem ser covariantes. Por exemplo, é assim que a classe `typing.FrozenSet` está documentada [fpy.li/7m] como covariante com uma variável de tipo usando o nome convencional `T_co`:

```
class FrozenSet(frozenset, AbstractSet[T_co]):
```

Aplicando a notação :> a tipos parametrizados, temos:

```
float :> int  
frozenset[float] :> frozenset[int]
```

Iteradores são outro exemplo de genéricos covariantes: eles não são coleções apenas para leitura como um `frozenset`, mas apenas produzem itens sob demanda. Qualquer código que espere um `abc.Iterator[float]` que produz números de ponto flutuante pode usar com segurança um `abc.Iterator[int]` que produz inteiros. Tipos `Callable` são covariantes no tipo devolvido pela mesma razão.

15.7.4.3. Tipos contravariantes

Dado $A \rightarrow B$, um tipo genérico K é contravariante se $K[A] < K[B]$.

Tipos genéricos contravariantes revertem a relação de subtipo dos tipos reais dos parâmetros.

A classe `TrashCan` exemplifica isso:

```
Refuse > Biodegradable
TrashCan[Refuse] < TrashCan[Biodegradable]
```

Um contêiner contravariante normalmente é uma estrutura de dados só para escrita, também conhecida como "coletor" (*sink*). Não há exemplos de coleções deste tipo na biblioteca padrão, mas existem alguns tipos com parâmetros de tipo contravariantes.

`Callable[[ParamType, ...], ReturnType]` é contravariante nos tipos dos parâmetros, mas covariante no `ReturnType`, como vimos na «Seção 8.5.11.1» [fpy.li/7y] (vol.1). Além disso, `Generator` [fpy.li/15-32], `Coroutine` [fpy.li/typepecoro], e `AsyncGenerator` [fpy.li/15-33] têm um parâmetro de tipo contravariante. O tipo `Generator` está descrito na «Seção 17.13.3» [fpy.li/87] (vol.3); `Coroutine` e `AsyncGenerator` são descritos no «Capítulo 21» [fpy.li/21] (vol.3).

Para efeito da presente discussão sobre variância, o ponto principal é que parâmetros formais contravariantes definem o tipo dos argumentos usados para invocar ou enviar dados para o objeto, enquanto parâmetros formais covariantes definem os tipos de saídas produzidos pelo objeto—o tipo devolvido por uma função ou produzido por um gerador. Os significados precisos de "enviar" e "produzir" são definidos na «Seção 17.13» [fpy.li/7z] (vol.3).

A partir destas observações sobre saídas covariantes e entradas contravariantes podemos derivar algumas orientações úteis.

15.7.4.4. Regras gerais de variância

Por fim, aqui estão algumas regras gerais a considerar quando estamos pensando sobre variância:

1. Se um parâmetro de tipo formal define um tipo para dados que saem de um objeto, ele pode ser covariante.
2. Se um parâmetro de tipo formal define um tipo para dados que entram em um objeto, ele pode ser contravariante.
3. Se um parâmetro de tipo formal define um tipo para dados que saem de um objeto e o mesmo parâmetro define um tipo para dados que entram em um objeto, ele deve ser invariante.
4. Na dúvida, use parâmetros de tipo formais invariantes. Não haverá prejuízo se no futuro precisar usar parâmetros de tipo covariantes ou contravariantes, pois nestes casos a tipagem ficará mais tolerante e não quebrará códigos existentes.

`Callable[[ParamType, ...], ReturnType]` demonstra as regras 1 e 2: O `ReturnType` é covariante, e cada `ParamType` é contravariante.

Por default, `TypeVar` cria parâmetros formais invariantes, e é assim que as coleções mutáveis na biblioteca padrão são anotadas. Veremos mais exemplos de variância na «Seção 17.13.3» [fpy.li/87] (vol.3).

A seguir, veremos como usar um protocolo estático genérico.

15.8. Implementando um protocolo estático genérico

A biblioteca padrão de Python 3.10 fornece alguns protocolos estáticos genéricos. Um deles é `SupportsAbs`, implementado assim no módulo `typing` [fpy.li/15-34]:

```
@runtime_checkable
class SupportsAbs(Protocol[T_co]):
    """An ABC with one abstract method __abs__ that is covariant in its
       return type."""
    __slots__ = ()

    @abstractmethod
    def __abs__(self) -> T_co:
        pass
```

T_co é declarado de acordo com a convenção de nomenclatura:

```
T_co = TypeVar('T_co', covariant=True)
```

Graças a SupportsAbs, o Mypy considera válido o seguinte código, como visto no Exemplo 21.

Exemplo 21. abs_demo.py: uso do protocolo genérico SupportsAbs

```
import math
from typing import NamedTuple, SupportsAbs

class Vector2d(NamedTuple):
    x: float
    y: float

    def __abs__(self) -> float: ①
        return math.hypot(self.x, self.y)

def is_unit(v: SupportsAbs[float]) -> bool: ②
    """'True' if the magnitude of 'v' is close to 1."""
    return math.isclose(abs(v), 1.0) ③

assert issubclass(Vector2d, SupportsAbs) ④

v0 = Vector2d(0, 1) ⑤
sqrt2 = math.sqrt(2)
v1 = Vector2d(sqrt2 / 2, sqrt2 / 2)
v2 = Vector2d(1, 1)
v3 = complex(.5, math.sqrt(3) / 2)
v4 = 1 ⑥

assert is_unit(v0)
assert is_unit(v1)
assert not is_unit(v2)
assert is_unit(v3)
assert is_unit(v4)

print('OK')
```


- ① Definir `__abs__` torna `Vector2d` *consistente-com* `SupportsAbs`.
- ② Parametrizar `SupportsAbs` com `float` assegura...
- ③ ...que o Mypy aceite `abs(v)` como primeiro argumento para `math.isclose`.
- ④ Graças a `@runtime_checkable` na definição de `SupportsAbs`, essa é uma asserção válida durante a execução.
- ⑤ Todo o restante do código passa pelas checagens do Mypy e pelas asserções durante a execução.
- ⑥ O tipo `int` também é *consistente-com* `SupportsAbs`. De acordo com o *typeshed* [[fpy.li/15-35](https://typeshed.dev/15-35)], `int.__abs__` devolve um `int`, o que é *consistente-com* o parametro de tipo `float` declarado na dica de tipo `is_unit` para o argumento `v`.

De forma similar, podemos escrever uma versão genérica do protocolo `RandomPicker`, apresentado no Exemplo 19 do Capítulo 13, que foi definido com um único método `pick` devolvendo `Any`.

O Exemplo 22 mostra como criar um `RandomPicker` genérico, covariante no tipo devolvido por `pick`.

Exemplo 22. generic_randompick.py: definição do RandomPicker genérico

```
from typing import Protocol, runtime_checkable, TypeVar

T_co = TypeVar('T_co', covariant=True) ①

@runtime_checkable
class RandomPicker(Protocol[T_co]): ②
    def pick(self) -> T_co: ... ③
```

- ① Declara `T_co` como covariante.
- ② Isso torna `RandomPicker` genérico, com um parâmetro de tipo formal covariante.
- ③ Usa `T_co` como tipo do valor devolvido.

O protocolo genérico `RandomPicker` pode ser covariante porque seu único parâmetro formal é usado em um tipo de saída.

Com isso, podemos dizer que temos mais um capítulo.

15.9. Resumo do capítulo

Começamos com um exemplo simples de uso de `@overload`, seguido por um exemplo mais complexo, que estudamos em detalhes: as assinaturas sobrecarregadas exigidas para anotar corretamente a função embutida `max`.

A seguir veio o tipo especial `typing.TypedDict`. Escolhi tratar dele aqui e não no «Capítulo 5» [fpy.li/5] (vol.1), onde vimos `typing.NamedTuple`, porque `TypedDict` parece mas não é uma fábrica de classes; ele é meramente uma forma de acrescentar dicas de tipo a uma variável ou a um argumento que exige um `dict` com um conjunto específico de chaves do tipo `string`, e tipos específicos para cada chave—algo que acontece quando usamos um `dict` como registro, muitas vezes no contexto do tratamento de dados JSON. Aquela seção foi um pouco mais longa porque usar `TypedDict` pode levar a um falso sentimento de segurança, e eu queria mostrar como as checagens durante a execução e o tratamento de erros são inevitáveis quando tentamos criar registros estruturados estaticamente a partir de mapeamentos, que são dinâmicos por natureza.

Então falamos sobre `typing.cast`, uma função criada para nos permitir orientar o checador de tipos. É importante considerar cuidadosamente quando usar `cast`, porque seu uso excessivo atrapalha o checador de tipos.

O acesso a dicas de tipo durante a execução veio em seguida. O ponto principal era usar `typing.get_type_hints` em vez de ler o atributo `__annotations__` diretamente. Entretanto, aquela função pode não ser confiável para algumas anotações, e vimos que os mantenedores de Python ainda estão discutindo uma forma de tornar as dicas de tipo usáveis durante a execução, e ao mesmo tempo reduzir seu impacto sobre o uso de CPU e memória.

A última seção foi sobre genéricos, começando com a classe genérica `LottoBlower`—que mais tarde aprendemos ser uma classe genérica invariante. Aquele exemplo foi seguido pelas definições de quatro termos básicos: tipo genérico, parâmetro de tipo formal, tipo parametrizado e parâmetro de tipo real.

Continuamos pelo grande tópico da variância, usando máquinas bebidas e latas de lixo para uma cantina como exemplos da "vida real" para tipos genéricos invariantes, covariantes e contravariantes. Então revisamos, formalizamos e aplicamos aqueles conceitos a exemplos na biblioteca padrão de Python.

Por fim, vimos como é definido um protocolo estático genérico, primeiro considerando o protocolo `typing.SupportsAbs`, e então aplicando a mesma ideia ao exemplo do `RandomPicker`, tornando-o mais rigoroso que o protocolo original do Capítulo 13.



O sistema de tipos de Python é um campo imenso e em rápida expansão. Este capítulo não é abrangente. Escolhi me concentrar em tópicos que são amplamente aplicáveis, ou particularmente complexos, ou conceitualmente importantes, e que assim provavelmente se manterão relevantes por mais tempo.

15.10. Para saber mais

O sistema de tipagem estática de Python já era complexo quando foi originalmente projetado, e tem se tornado mais complexo a cada ano. A Tabela 1 lista todas as PEPs que encontrei até maio de 2021. Seria necessário um livro inteiro para cobrir tudo.

*Tabela 1. PEPs sobre dicas de tipo, com links nos títulos. PEPs com números marcados com * são importantes o suficiente para serem mencionadas no parágrafo de abertura da documentação de `typing` [fpy.li/4a]. Pontos de interrogação na coluna Python indica PEPs em discussão ou ainda não implementadas; "n/a" aparece em PEPs informacionais sem relação com uma versão específica de Python. Dados coletados em maio de 2021.*

PEP	título	Python	ano
3107	<i>Function Annotations</i> [fpy.li/pep3107] (Anotações de Função)	3.0	2006
483*	<i>The Theory of Type Hints</i> [fpy.li/pep483] (A Teoria das Dicas de Tipo)	n/a	2014
484*	<i>Type Hints</i> [fpy.li/pep484] (Dicas de Tipo)	3.5	2014
482	<i>Literature Overview for Type Hints</i> [fpy.li/pep482] (Revisão da Literatura sobre Dicas de Tipo)	n/a	2015
526*	<i>Syntax for Variable Annotations</i> [fpy.li/pep526] (Sintaxe para Anotações de Variáveis)	3.6	2016

PEP	título	Python	ano
544*	<i>Protocols: Structural subtyping (static duck typing)</i> [fpy.li/pep544] (Protocolos: subtipagem estrutural (duck typing estático))	3.8	2017
557	<i>Data Classes</i> [fpy.li/pep557] (Classes de Dados)	3.7	2017
560	<i>Core support for typing module and generic types</i> [fpy.li/pep560] (Suporte nativo para tipagem de módulos e tipos genéricos)	3.7	2017
561	<i>Distributing and Packaging Type Information</i> [fpy.li/pep561] (Distribuindo e Empacotando Informação de Tipo)	3.7	2017
563	<i>Postponed Evaluation of Annotations</i> [fpy.li/pep563] (Avaliação Adiada de Anotações)	3.7	2017
586*	<i>Literal Types</i> [fpy.li/pep586] (Tipos Literais)	3.8	2018
585	<i>Type Hinting Generics In Standard Collections</i> [fpy.li/pep585] (Dicas de Tipo para Genéricos nas Coleções Padrão)	3.9	2019
589*	<i>TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys</i> [fpy.li/pep589] (TypedDict: Dicas de Tipo para Dicionários com um Conjunto Fixo de Chaves)	3.8	2019
591*	<i>Adding a final qualifier to typing</i> [fpy.li/pep591] (Acrescentando um qualificador final à tipagem)	3.8	2019
593	<i>Flexible function and variable annotations</i> [fpy.li/pep593] (Anotações flexíveis para funções e variáveis)	?	2019
604	<i>Allow writing union types as X Y</i> [fpy.li/pep604] (Permitir a definição de tipos de união como X Y)	3.10	2019
612	<i>Parameter Specification Variables</i> [fpy.li/pep612] (Variáveis de Especificação de Parâmetros)	3.10	2019
613	<i>Explicit Type Aliases</i> [fpy.li/pep613] (Aliases de Tipo Explícitos)	3.10	2020
645	<i>Allow writing optional types as x?</i> [fpy.li/pep645] (Permitir a definição de tipos opcionais como x?)	?	2020

PEP	título	Python	ano
646	<i>Variadic Generics</i> [fpy.li/pep646] (Genéricos Variádicos)	?	2020
647	<i>User-Defined Type Guards</i> [fpy.li/pep647] (Guardas de Tipos Definidos pelo Usuário)	3.10	2021
649	<i>Deferred Evaluation Of Annotations Using Descriptors</i> [fpy.li/pep649] (Avaliação Adiada de Anotações Usando Descritores)	?	2021
655	<i>Marking individual TypedDict items as required or potentially-missing</i> [fpy.li/pep655] (Marcando itens individuais de TypedDict como obrigatórios ou potencialmente ausentes)	?	2021

A documentação oficial de Python mal consegue acompanhar tudo aquilo, então a documentação do Mypy [fpy.li/mypy] é uma referência essencial. *Robust Python*, de Patrick Viafore (O’Reilly), é o primeiro livro com um tratamento abrangente do sistema de tipagem estática de Python que conheço, publicado em agosto de 2021. Você pode estar lendo o segundo livro sobre o assunto nesse exato instante.

O tópico sutil da variância tem sua própria seção na PEP 484 [fpy.li/15-37], e também é abordado na página *Generics* [fpy.li/15-38] do Mypy, bem como em sua inestimável página *Common Issues* [fpy.li/15-39] (Problemas Comuns).

A PEP 362—*Function Signature Object* [fpy.li/pep362] (O objeto assinatura de função) vale a pena ler se você pretende usar o módulo `inspect`, que complementa a função `typing.get_type_hints`.

Se tiver interesse na história de Python, saiba que Guido van Rossum publicou *Adding Optional Static Typing to Python* [fpy.li/15-40] (Acrescentando tipagem estática opcional ao Python).

Python 3 Types in the Wild: A Tale of Two Type Systems [fpy.li/15-41] (Os tipos de Python 3 na natureza: um conto de dois sistemas de tipo) é um artigo científico de Ingkarat Rak-amnouykit e outros, do Rensselaer Polytechnic Institute e do IBM TJ Watson Research Center. O artigo avalia o uso de dicas de tipo em projetos de código aberto no GitHub, mostrando que a maioria dos projetos não as usa, e também que a maioria dos projetos que têm dicas de tipo aparentemente não usa um checador de tipos. Achei particularmente interessante a discussão das

semânticas diferentes do Mypy e do *pytype* do Google, onde os autores concluem que eles são "essencialmente dois sistemas de tipos diferentes."

Dois artigos fundamentais sobre tipagem gradual são *Pluggable Type Systems* [fpy.li/15-42] (Sistemas de tipo conectáveis), de Gilad Bracha, e *Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages* [fpy.li/15-43] (Tipagem Estática Quando Possível, Tipagem Dinâmica Quando Necessário: O Fim da Guerra Fria Entre Linguagens de Programação), de Eric Meijer e Peter Drayton.^[17]

Apreendi muito lendo as partes relevantes de alguns livros sobre outras linguagens que implementam algumas das mesmas ideias:

- *Atomic Kotlin* [fpy.li/15-44], de Bruce Eckel e Svetlana Isakova (Mindview)
- *Effective Java*, 3rd ed., [fpy.li/15-45], de Joshua Bloch (Addison-Wesley)
- *Programming with Types: TypeScript Examples* [fpy.li/15-46], de Vlad Riscutia (Manning)
- *Programming TypeScript* [fpy.li/15-47], de Boris Cherny (O'Reilly)
- *The Dart Programming Language* [fpy.li/15-48] de Gilad Bracha (Addison-Wesley).^[18]

Para algumas visões críticas sobre os sistemas de tipagem, recomendo os posts de Victor Youdaiken *Bad ideas in type theory* [fpy.li/15-49] (Ideias ruins em teoria dos tipos) e *Types considered harmful II* [fpy.li/15-50] (Tipos considerados nocivos II).

Por fim, me surpreendi ao encontrar *Generics Considered Harmful* [fpy.li/15-51] (Genéricos Considerados Nocivos), de Ken Arnold, um desenvolvedor principal de Java desde o início, bem como co-autor das primeiras quatro edições do livro oficial *The Java Programming Language* (Addison-Wesley)—com James Gosling, o principal criador de Java.

Infelizmente, as críticas de Arnold também se aplicam ao sistema de tipagem estática de Python. Quando leio as muitas regras e casos especiais das PEPs de tipagem, sou constantemente lembrado dessa passagem do post de Arnold:

O que nos traz ao problema que sempre cito para o C++: a "exceção de enésima ordem à regra de exceção".

É mais ou menos assim: "Você pode fazer x, exceto no caso y, a menos que y faça z, caso em que você pode se..."

Felizmente, Python tem uma vantagem crítica sobre o Java e o C++: um sistema de tipagem opcional. Podemos silenciar os checadores de tipos e omitir as dicas de tipo quando se tornam muito inconvenientes.

Ponto de Vista

As tocas de coelho da tipagem

Quando usamos um checador de tipos, algumas vezes somos obrigados a descobrir e importar classes que não precisávamos conhecer, e que nosso código não precisa usar—exceto para escrever dicas de tipo. Tais classes não são documentadas, provavelmente porque são consideradas detalhes de implementação pelos autores dos pacotes. Aqui estão dois exemplos da biblioteca padrão.

Tive que vasculhar a imensa documentação do `asyncio`, e depois navegar o código-fonte de vários módulos daquele pacote para descobrir a classe não-documentada `TransportSocket` no módulo igualmente não documentado `asyncio.trsock` só para usar `cast()` no exemplo do `server.sockets`, na Seção 15.4. Usar `socket.socket` em vez de `TransportSocket` seria incorreto, pois esse último não é subtipo do primeiro, como explicitado em uma docstring [[fpy.li/15-52](#)] no código-fonte.

Caí em uma toca de coelho similar quando acrescentei dicas de tipo ao «Exemplo 13 do Capítulo 19» [[fpy.li/96](#)] (vol.3), uma demonstração simples de multiprocessing. Aquele exemplo usa objetos `SimpleQueue`, obtidos invocando `multiprocessing.SimpleQueue()`. Entretanto, não pude usar aquele nome em uma dica de tipo, porque `multiprocessing.SimpleQueue` não é uma classe! É um método vinculado da classe não documentada `multiprocessing.BaseContext`, que cria e devolve uma instância da classe `SimpleQueue`, definida no módulo não-documentado `multiprocessing.queues`.

Em cada um desses casos, tive que gastar algumas horas até encontrar a classe não-documentada correta para importar, só para escrever uma única dica de tipo. Esse tipo de pesquisa é parte do trabalho quando você está escrevendo um livro. Mas se eu estivesse criando o código para uma aplicação, provavelmente evitaria tais caças ao tesouro por causa de uma única linha, e simplesmente colocaria `# type: ignore`. Algumas vezes essa é a única solução com custo-benefício positivo.

Notação de variância em outras linguagens

A variância é um tópico complicado, e a sintaxe das dicas de tipo de Python deixa a desejar. Esta citação direta da PEP 484 evidencia isso:

Covariância ou contravariância não são propriedades de uma variável de tipo, mas sim uma propriedade da classe genérica definida usando essa variável.^[19]

Se esse é o caso, por que a covariância e a contravariância são declaradas com `TypeVar` e não na classe genérica?

Os autores da PEP 484 optaram por introduzir dicas de tipo sem fazer qualquer modificação no interpretador. Em Python, todo identificador aparece pela primeira vez no código-fonte de um módulo através de uma atribuição, ou uma instrução especial como `import`, `class`, ou `def`. Por isso tiveram que criar `TypeVar` para declarar uma variável de tipo através de uma atribuição:

```
T = TypeVar('T')
```

Para não mexer no *parser*, reutilizaram o operador `[]` na sintaxe `Klass[T]` para genéricos—em vez da notação `Klass<T>` usada em outras linguagens populares, incluindo C#, Java, Kotlin e TypeScript. Estas linguagens não exigem que variáveis de tipo sejam declaradas antes de serem usadas.

Além disso, a sintaxe do Kotlin e do C# torna claro se um parâmetro de tipo é covariante, contravariante ou invariante exatamente onde isso faz sentido: na declaração de classe ou interface.

Em Kotlin, poderíamos declarar a `BeverageDispenser` assim:

```
class BeverageDispenser<out T> {  
    // etc...  
}
```

O modificador `out` no parâmetro de tipo formal significa que `T` é um tipo de *output* (saída), e portanto `BeverageDispenser` é covariante. Você provavelmente consegue adivinhar como `TrashCan` seria declarada:

```
class TrashCan<in T> {  
    // etc...  
}
```

Dado `T` como um parâmetro de tipo formal de *input* (entrada), então `TrashCan` é contravariante. Se nem `in` nem `out` aparecem, então a classe é invariante naquele parâmetro.

É fácil lembrar das regras gerais de variância (Seção 15.7.4.4) quando `out` e `in` são usados nos parâmetros de tipo formais. Isso sugere uma convenção melhor para nomear de variáveis de tipo covariantes e contravariantes:

```
T_out = TypeVar('T_out', covariant=True)  
T_in = TypeVar('T_in', contravariant=True)
```

Aí poderíamos definir as classes assim:

```
class BeverageDispenser(Generic[T_out]):  
    ...  
  
class TrashCan(Generic[T_in]):  
    ...
```

Será tarde demais para adotar `T_out` e `T_in` em vez de `T_co` e `T_contra` que foram sugeridos na PEP 484?

[1] De um vídeo no YouTube da *A Language Creators' Conversation: Guido van Rossum, James Gosling, Larry Wall & Anders Hejlsberg* (Uma Conversa entre Criadores de Linguagens: Guido van Rossum, James Gosling, Larry Wall & Anders Hejlsberg), transmitido em 2 de abril de 2019. A citação (editada por brevidade) começa em 1:32:05 [fpy.li/15-1]. Produzi e publiquei a transcrição completa em github.com/fluentpython/language-creators.

[2] Agradeço a Jelle Zijlstra—um mantenedor do *typeshed*—que me ensinou várias coisas, incluindo como reduzir minhas nove sobrecargas originais para "apenas" seis.

[3] Em maio de 2020, o *pytype* ainda permite isso. Mas seu FAQ [fpy.li/15-6] diz que tal operação será proibida no futuro. Veja a pergunta *Why didn't pytype catch that I changed the type of an annotated variable?* (Por que o *pytype* não avisou quando eu mudei o tipo de uma variável anotada?) no FAQ [fpy.li/15-6] do *pytype*.

[4] Prefiro usar o pacote *lxml* [fpy.li/15-8] para gerar e interpretar XML: ele é fácil de começar a usar, completo e rápido. Infelizmente, nem o *lxml* nem o *ElementTree* [fpy.li/7f] do próprio Python cabem na RAM limitada de meu microcontrolador hipotético.

[5] A documentação do *Mypy* discute isso na seção *Types of empty collections* [fpy.li/15-11] (Tipos de coleções vazias) da página *Common issues and solutions* [fpy.li/15-10] (Problemas comuns e soluções).

[6] Brett Cannon, Guido van Rossum e outros vem discutindo como escrever dicas de tipo para *json.loads()* desde 2016, em *Mypy issue #182: Define a JSON type (Definir um tipo JSON)* [fpy.li/15-12].

[7] O uso de *enumerate* no exemplo serve para confundir intencionalmente o checador de tipos. Uma implementação mais simples, produzindo strings diretamente, sem passar pelo índice de *enumerate*, seria corretamente analisada pelo *Mypy*, e o *cast()* não seria necessário.

[8] Relatei o problema em *issue #5535* [fpy.li/15-17] no *typeshed*, "Dica de tipo errada para o atributo *sockets* em *asyncio.base_events.Server* sockets attribute.", e ele foi rapidamente resolvido por Sebastian Rittau. Mas decidi manter o exemplo, pois ele ilustra um caso de uso comum para *cast*, e o *cast* que escrevi é inofensivo.

[9] Na realidade, inicialmente coloquei um comentário *# type: ignore* às linhas com *server.sockets[0]* porque, após pesquisar um pouco, encontrei linhas similares na documentação [fpy.li/7g] do *asyncio* e em um caso de teste [fpy.li/15-19], e aí comecei a suspeitar que o problema não estava no meu código.

[10] Mensagem de 18 de maio de 2020 [fpy.li/15-21] para a lista de e-mail *typing-sig*.

[11] A sintaxe *# type: ignore[code]* permite especificar qual erro do *Mypy* está sendo silenciado, mas os códigos nem sempre são fáceis de interpretar. Veja a página *Error codes* [fpy.li/15-22] na documentação do *Mypy*.

[12] Não vou entrar nos detalhes da implementação de *clip*, mas se você tiver curiosidade, pode ler o módulo completo em *clip_annot.py* [fpy.li/15-23].

[13] Mensagem *PEP 563 in light of PEP 649* [fpy.li/15-27] (PEP 563 à luz da PEP 649), publicado em 16 de abril de 2021.

[14] Os termos são do livro clássico de Joshua Bloch, *Effective Java*, 3rd ed. As traduções, definições e exemplos são meus.

[15] A primeira vez que vi a analogia da cantina para variância foi no prefácio de Erik Meijer para o livro *The Dart Programming Language* ("A Linguagem de Programação Dart"), de Gilad Bracha (Addison-Wesley).

[16] Muito melhor que banir livros!

[17] O leitor de notas de rodapé se lembrará que dei o crédito a Erik Meijer pela analogia da cantina

para explicar variância.

[18] Esse livro foi escrito para o Dart 1. Há mudanças significativas no Dart 2, inclusive no sistema de tipos. Mesmo assim, Bracha é um pesquisador importante na área de design de linguagens de programação, e achei o livro valioso por sua perspectiva sobre o design do Dart.

[19] Veja o último parágrafo da seção *Covariance and Contravariance* [fpy.li/15-37] (Covariância e Contravariância) na PEP 484.

Capítulo 16. Sobrecarga de operadores

Certas coisas me deixam meio dividido, como a sobrecarga de operadores. Deixei a sobrecarga de operadores de fora em uma decisão bastante pessoal, pois tinha visto gente demais abusar [deste recurso] no C++.^[1]

— James Gosling, Criador de Java

Em Python, podemos calcular juros compostos usando uma fórmula escrita assim:

```
interest = principal * ((1 + rate) ** periods - 1)
```

Operadores que aparecem entre operandos, como + em `1 + rate`, são *operadores infixos*. No Python, operadores infixos podem lidar com qualquer tipo arbitrário. Assim, se você está trabalhando com dinheiro de verdade, pode armazenar `principal`, `rate`, e `periods` como números exatos—instâncias da classe `decimal.Decimal` de Python. A mesma fórmula vai funcionar como escrita, calculando um resultado exato.

Mas em Java, se você mudar de `float` para `BigDecimal`, para obter resultados exatos, não é mais possível usar operadores infixos, porque naquela linguagem eles só funcionam com tipos primitivos como `float` ou `long`. Veja a mesma fórmula escrita em Java para funcionar com números `BigDecimal`:

```
BigDecimal interest = principal.multiply(BigDecimal.ONE.add(rate)
                                .pow(periods).subtract(BigDecimal.ONE));
```

Está claro que operadores infixos tornam as fórmulas mais legíveis. A sobrecarga de operadores é necessária para suportar a notação infixa de operadores com tipos definidos pelo usuário ou em extensões compiladas, como os arrays da NumPy.

Oferecer a sobrecarga de operadores em uma linguagem de alto nível e fácil de usar foi talvez uma das principais razões do grande sucesso de Python na ciência de dados, incluindo as aplicações científicas e financeiras.

Na Seção 1.3.1» [fpy.li/84] (vol.1), vimos algumas implementações triviais de operadores em uma classe básica `Vector`. Escrevi os métodos `__add__` e `__mul__` no «Exemplo 2 do Capítulo 1» [fpy.li/8w] (vol.1) para demonstrar como os métodos especiais suportam a sobrecarga de operadores, mas deixei passar alguns problemas sutis naquelas implementações. Além disso, no Exemplo 2 do Capítulo 11 notamos que o método `Vector2d.__eq__` considera `True` a seguinte expressão: `Vector(3, 4) == [3, 4]`. Tal resultado pode fazer sentido ou não. Neste capítulo vamos cuidar destes problemas, e falaremos também de:

- Como um método de operador infixo deve indicar que não consegue tratar um operando
- Tipagem pato e tipagem ganso para lidar com operandos de tipos diferentes
- O comportamento especial dos operadores de comparação rica (`==`, `>`, `<=`, etc.)
- O tratamento padrão de operadores de atribuição aumentada, como `+=`, e como sobrecarregá-los

16.1. Novidades neste capítulo

A tipagem ganso é uma parte fundamental de Python, mas as ABCs `numbers` não são suportadas na tipagem estática. Então, mudei o Exemplo 11 para usar tipagem pato, em vez de uma checagem explícita usando `isinstance` contra `numbers.Real`.^[2]

Na primeira edição do *Python Fluente*, tratei do operador de multiplicação de matrizes `@` como uma mudança futura, pois o Python 3.5 ainda estava em desenvolvimento. Agora o `@` está integrado ao fluxo do capítulo na Seção 16.6. Aproveitei a tipagem ganso para tornar a implementação de `__matmul__` mais segura na primeira edição, sem comprometer sua flexibilidade.

A Seção 16.11 agora inclui algumas novas referências—incluindo um post do blog de Guido van Rossum. Também inclui menções a duas bibliotecas que demonstram usos interessantes da sobrecarga de operadores em contextos não numéricos: `pathlib` e `Scapy`.

16.2. Introdução à sobrecarga de operadores

A sobrecarga de operadores permite que objetos definidos pelo usuário suportem operadores infixos como `+` e `|`, ou com operadores unários como `-` e `~`. De forma geral, em Python a notação de invocação de função (`f()`), o acesso a atributos (`p.x`) e o acesso a itens e o fatiamento (`v[0]`) também são operadores, mas este capítulo trata dos operadores unários e infixos.

A sobrecarga de operadores tem má reputação em certos círculos. É um recurso que pode ser abusado, resultando em programadores confusos, bugs, e gargalos de desempenho inesperados. Mas se bem utilizada, possibilita APIs agradáveis de usar e código legível. Python alcança um bom equilíbrio entre flexibilidade, usabilidade e segurança, pela imposição de algumas limitações:

- Não é permitido modificar o significado dos operadores para os tipos embutidos.
- Não é permitido criar novos operadores, apenas sobrecarregar os existentes.
- Alguns poucos operadores não podem ser sobrecarregados: `is`, `and`, `or` e `not` (mas os operadores `==`, `&`, `|`, e `~` podem).

No Capítulo 12, na classe `Vector`, já apresentamos um operador infixos `==`, suportado pelo método `__eq__`. Neste capítulo, vamos melhorar a implementação de `__eq__` para lidar melhor com operandos de outros tipos além de `Vector`. Entretanto, os operadores de comparação rica (`==`, `!=`, `>`, `<`, `>=`, `<=`) são casos especiais de sobrecarga de operadores, então começaremos sobrecarregando quatro operadores aritméticos em `Vector`: os operadores unários `-` e `+`, seguido pelos infixos `+` e `*`.

Vamos começar pelo tópico mais fácil: operadores unários.

16.3. Operadores unários

Na Referência da Linguagem Python, a seção Operações aritméticas unárias e bit a bit [fpy.li/7n], cita três operadores unários, listados abaixo com os seus métodos especiais:

- implementado por `__neg__`

Negativo aritmético unário. Se x é 42 então $-x == -42$.

+ implementado por `__pos__`

Positivo aritmético unário. Em geral, $x == +x$, mas há alguns poucos casos em que isto não ocorre. Veja: Quando x e $+x$ não são iguais (ao final desta seção).

\sim implementado por `__invert__`

Negação binária, ou inversão bit a bit de um inteiro, definida como $\sim x == -(x+1)$. Se x é 2 então $\sim x == -3$, porque a representação binária de 2 é 0010 e -3 é 1101. Veja Complemento para dois [fpy.li/7p] na Wikipédia para entender esta representação de inteiros com sinal.

O capítulo Modelo de Dados na Referência da Linguagem Python_ também inclui a função embutida `abs()` como um operador unário. O método especial associado é `__abs__`, como já vimos.

É fácil suportar operadores unários. Basta implementar o método especial apropriado, que receberá apenas um argumento: `self`. Use a lógica que fizer sentido na sua classe, mas respeite a regra geral dos operadores: sempre devolva um novo objeto. Em outras palavras, não modifique o receptor (`self`), mas crie e devolva uma nova instância do tipo adequado.

No caso de `-` e `+`, o resultado será provavelmente uma instância da mesma classe de `self`. Para o `+` unário, se o receptor for imutável você deveria devolver `self`; caso contrário, devolva uma cópia de `self`. Para `abs()`, o resultado deve ser um número escalar.^[3]

Já no caso de \sim , é difícil determinar o que seria um resultado razoável se você não estiver lidando com bits de um número inteiro. No pacote de análise de dados *pandas* [fpy.li/pandas], o `til` nega condições booleanas de filtragem; veja exemplos na documentação do *pandas*, em *Boolean indexing* [fpy.li/16-4] (indexação booleana).

Como prometido acima, vamos implementar vários novos operadores na classe *Vector*, do Capítulo 12. O Exemplo 1 mostra o método `__abs__`, que já estava no Exemplo 16 do Capítulo 12, e os novos métodos `__neg__` e `__pos__` para operadores unários.

Exemplo 1. vector_v6.py: operadores unários - e + implementados.

```
def __abs__(self):
    return math.hypot(*self)

def __neg__(self):
    return Vector(-x for x in self) ①

def __pos__(self):
    return Vector(self) ②
```

① Para computar $-v$, cria um novo `Vector` com a negação de cada componente de `self`.

② Para computar $+v$, cria um novo `Vector` com cada componente de `self`.

Lembre-se de que instâncias de `Vector` são iteráveis, e o `Vector.__init__` recebe um argumento iterável, por isso as implementações de `__neg__` e `__pos__` ficaram tão simples.

Não vamos implementar `__invert__`. Se um usuário tentar escrever $\sim v$ para uma instância de `Vector`, Python vai gerar um `TypeError` com uma mensagem clara: “bad operand type for unary \sim : 'Vector'” (operando inválido para o \sim unário: 'Vector').

O quadro a seguir trata de uma curiosidade que algum dia poderá ajudar você a ganhar uma aposta sobre o $+$ unário.

Quando x e $+x$ não são iguais

Todo mundo espera que $x == +x$, e isso é verdade no Python quase todo o tempo, mas encontrei dois casos na biblioteca padrão onde $x != +x$.

O primeiro caso envolve a classe `decimal.Decimal`. Você pode obter $x != +x$ se x é uma instância de `Decimal`, criada em um dado contexto aritmético e $+x$ for então calculada em um contexto com definições diferentes. Por exemplo, x é calculado em um contexto com uma determinada precisão, mas a precisão do contexto é modificada e daí $+x$ é avaliado. Veja o Exemplo 2.

Exemplo 3. O + unário produz um novo `Counter` sem as contagens negativas ou zero

```
>>> ct = Counter('abracadabra')
>>> ct
Counter({'a': 5, 'r': 2, 'b': 2, 'd': 1, 'c': 1})
>>> ct['r'] = -3
>>> ct['d'] = 0
>>> ct
Counter({'a': 5, 'b': 2, 'c': 1, 'd': 0, 'r': -3})
>>> +ct
Counter({'a': 5, 'b': 2, 'c': 1})
```

Como visto, +ct devolve um contador onde todas as contagens são maiores que zero.

Agora voltamos à nossa programação normal.

16.4. Sobrecarregando + para adição em Vector

A classe Vector é um tipo sequência, e a seção Emulando tipos contêineres [fpy.li/6n] da documentação oficial do Python diz que sequências devem suportar o operador + para concatenação e * para repetição. Entretanto, aqui vamos implementar + e * como operações matemáticas de vetores, algo um pouco mais complicado porém mais útil para um tipo Vector.



Usuários que desejem concatenar ou repetir instâncias de Vector podem convertê-las para tuplas ou listas, aplicar o operador e convertê-las de volta—graças ao fato de Vector ser iterável e poder ser criado a partir de um iterável:

```
>>> v_concat = Vector(list(v1) + list(v2))
>>> v_repeat = Vector(tuple(v1) * 5)
```

Somar dois vetores euclidianos resulta em um novo vetor cujos componentes são as somas dos componentes correspondentes dos operandos. Ilustrando:

```
>>> v1 = Vector([3, 4, 5])
>>> v2 = Vector([6, 7, 8])
>>> v1 + v2
Vector([9.0, 11.0, 13.0])
>>> v1 + v2 == Vector([3 + 6, 4 + 7, 5 + 8])
True
```

E o que acontece se tentarmos somar duas instâncias de `Vector` de tamanhos diferentes? Poderíamos gerar um erro, mas considerando as aplicações práticas (tal como recuperação de informação), é melhor preencher o `Vector` menor com zeros. Esse é o resultado que queremos:

```
>>> v1 = Vector([3, 4, 5, 6])
>>> v3 = Vector([1, 2])
>>> v1 + v3
Vector([4.0, 6.0, 5.0, 6.0])
```

Dados esses requisitos básicos, podemos implementar `__add__`:

Exemplo 4. Método `Vector.__add__`, versão #1

```
# dentro da classe Vector

def __add__(self, other):
    pairs = itertools.zip_longest(self, other, fillvalue=0.0) ①
    return Vector(a + b for a, b in pairs) ②
```

- ① `pairs` é um gerador que produz tuplas (a, b) , onde a vem de `self` e b de `other`. Se `self` e `other` tiverem tamanhos diferentes, `fillvalue` fornece os valores que faltam no o iterável mais curto.
- ② Um novo `Vector` é criado a partir de uma expressão geradora, produzindo uma soma para cada (a, b) de `pairs`.

Note que `__add__` cria um novo `Vector`, sem modificar `self` ou `other`.



Métodos especiais implementando operadores unários ou infixos não devem nunca modificar o valor dos operandos. Espera-se que expressões com tais operandos produzam resultados criando novos objetos. Só operadores de atribuição aumentada podem modificar o primeiro operando (`self`), quando ele é mutável, como discutido na Seção 16.9.

O Exemplo 4 permite somar um `Vector` a um `Vector2d` ou uma tupla:

Exemplo 5. A versão #1 de `Vector.__add__` aceita objetos que não são `Vector`

```
>>> v1 = Vector([3, 4, 5])
>>> v1 + (10, 20, 30)
Vector([13.0, 24.0, 35.0])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v1 + v2d
Vector([4.0, 6.0, 5.0])
```

Os dois usos de `+` no Exemplo 5 funcionam porque `__add__` usa `zip_longest(...)`, capaz de consumir qualquer iterável, e a expressão geradora que cria um novo `Vector` simplesmente efetua a operação `a + b` com os pares produzidos por `zip_longest(...)`, então qualquer iterável que produza números compatíveis com `float` servirá. Entretanto, se trocarmos a ordem dos operandos, a soma de tipos diferentes falha. Veja o Exemplo 6.

Exemplo 6. O Exemplo 4 falha quando o operando à esquerda não é `Vector`

```
>>> v1 = Vector([3, 4, 5])
>>> (10, 20, 30) + v1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "Vector") to tuple
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v2d + v1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Vector2d' and 'Vector'
```

Para suportar operações envolvendo objetos de tipos diferentes, Python implementa um mecanismo especial de despacho para os métodos especiais de operadores infixos.

Dada a expressão $a + b$, o interpretador executará os seguintes passos (veja também a Figura 1):

1. Se a implementa `__add__`, Python invoca `a.__add__(b)` e devolve o resultado, a menos que seja `NotImplemented`.
2. Se a não implementa `__add__`, ou a chamada `a.__add__(b)` devolve `NotImplemented`, Python verifica se b implementa `__radd__`, e então invoca `b.__radd__(a)` e devolve o resultado, a menos que seja `NotImplemented`.
3. Se b não implementa `__radd__`, ou a chamada `b.__radd__(a)` devolve `NotImplemented`, Python gera um `TypeError` com a mensagem "unsupported operand types" (tipos de operandos não suportados).

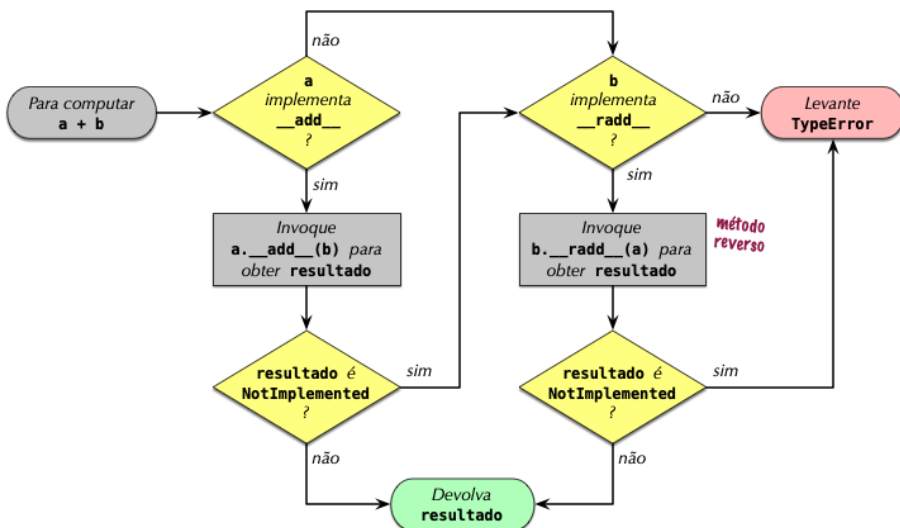


Figura 1. Fluxograma para computar $a + b$ com `__add__` e `__radd__`.



O método `__radd__` é chamado de variante "reversa" ou "refletida" de `__add__`. Adotei o termo geral "métodos especiais reversos". A documentação de Python usa os dois termos. O capítulo «Modelo de Dados» [fpy.li/dtmodel] usa "refletido", mas em «Implementando operações aritméticas» [fpy.li/16-7], a documentação menciona métodos "adiante" (*forward*) e "reverso" (*reverse*), uma terminologia que considero melhor, pois "adiante" e "reverso" descrevem sentidos opostos, enquanto o oposto de "refletido" não é tão evidente.

Assim, para fazer as somas de tipos diferentes no Exemplo 6 funcionarem, precisamos implementar o método `Vector.__radd__`, que Python vai invocar como alternativa, se o operando à esquerda não implementar `__add__`, ou se implementar mas devolver `NotImplemented`, indicando que não sabe como tratar o operando à direita.



Não confunda `NotImplemented` com `NotImplementedError`. O primeiro é um valor *singleton* especial, que um método especial de operador infixado deve devolver para informar o interpretador que não consegue tratar um dado operando.

Por sua vez, `NotImplementedError` é uma exceção que um método abstrato pode levantar para avisar que subclasses devem sobrescrever este método. Esta exceção é antiga no Python; atualmente a melhor forma de marcar um método abstrato é usar o decorador `@abc.abstractmethod`.

A implementação viável mais simples de `__radd__` aparece no Exemplo 7.

Exemplo 7. Os métodos `__add__` e `__radd__` de `Vector`

```
# dentro da classe vetor
def __add__(self, other): ①
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
    return Vector(a + b for a, b in pairs)

def __radd__(self, other): ②
    return self + other
```

- ① Nenhuma mudança no `__add__` do Exemplo 4; repeti o código aqui porque é invocado por `__radd__`.
- ② `__radd__` apenas delega para `__add__`.

Muitas vezes, `__radd__` pode ser simples assim: apenas a invocação do operador apropriado, delegando para `__add__` neste caso. Isso se aplica para qualquer operador comutativo. O `+` é comutativo quando lida com números ou com nossos vetores, mas não é comutativo ao concatenar sequências no Python.

Se `__radd__` apenas invoca `__add__`, aqui está uma forma mais eficiente de obter o mesmo efeito:

```
def __add__(self, other):
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
    return Vector(a + b for a, b in pairs)

__radd__ = __add__
```

Os métodos no Exemplo 7 funcionam com objetos `Vector` ou com qualquer iterável com itens numéricos, tal como um `Vector2d`, uma tupla de inteiros ou um array de números de ponto flutuante. Mas se alimentado com um objeto não-iterável, `__add__` gera uma exceção com uma mensagem não muito útil, como no Exemplo 8.

Exemplo 8. O método `Vector.__add__` precisa de operandos iteráveis

```
>>> v1 + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 328, in __add__
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
TypeError: zip_longest argument #2 must support iteration
```

E pior ainda, recebemos uma mensagem enganosa se um operando for iterável, mas seus itens não puderem ser somados aos itens `float` no `Vector`. Veja o Exemplo 9.

Exemplo 9. O método `Vector.__add__` exige um iterável com itens numéricos

```
>>> v1 + 'ABC'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 329, in __add__
    return Vector(a + b for a, b in pairs)
  File "vector_v6.py", line 243, in __init__
    self._components = array(self.typecode, components)
  File "vector_v6.py", line 329, in <genexpr>
    return Vector(a + b for a, b in pairs)
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Tentei somar um `Vector` a uma `str`, mas a mensagem reclama de `float` e `str`.

Na verdade, os problemas no Exemplo 8 e no Exemplo 9 são mais profundos que meras mensagens de erro obscuras: se um método especial de operando não é capaz de devolver um resultado válido por incompatibilidade de tipos, ele tem que devolver `NotImplemented` e não gerar um `TypeError`. Ao devolver `NotImplemented`, a porta fica aberta para o outro operando executar a operação, quando Python tentar invocar o método reverso em sua classe.

No espírito da tipagem pato, não vamos testar o tipo do operando `other` ou o tipo de seus elementos. Vamos capturar as exceções e devolver `NotImplemented`. Se o interpretador ainda não tiver invertido os operandos, tentará isso em seguida. Se a invocação do método reverso devolver `NotImplemented`, então Python irá gerar um `TypeError` com uma mensagem de erro padrão "unsupported operand type(s) for +: 'Vector' and 'str'" (*tipos de operandos não suportados para +: Vector e 'str'*)

Veja a implementação final dos métodos especiais de adição de `Vector`.

Exemplo 10. `vector_v6.py`: métodos do operador + adicionados a `vector_v5.py` (Exemplo 16 do Capítulo 12)

```
def __add__(self, other):
    try:
        pairs = itertools.zip_longest(self, other, fillvalue=0.0)
        return Vector(a + b for a, b in pairs)
    except TypeError:
        return NotImplemented
```

```
def __radd__(self, other):  
    return self + other
```

Observe que agora `__add__` captura um `TypeError` e devolve `NotImplemented`.



Se um método de operador infixo gera uma exceção, ele interrompe o algoritmo de despacho do operador. No caso específico de `TypeError`, geralmente é melhor capturar esta exceção e devolver `NotImplemented`. Isto permite que o interpretador tente chamar o método reverso do segundo operando.

Agora que já sobrecarregamos o operador `+` com segurança, implementando `__add__` e `__radd__`, vamos enfrentar outro operador infixo: `*`.

16.5. Sobrecarregando `*` para multiplicação por escalar

O que significa `Vector([1, 2, 3]) * x`? Se `x` é um número escalar, isto é uma "multiplicação por escalar", e o resultado deve ser um novo `Vector` com cada componente multiplicado por `x`—também conhecida como multiplicação elemento a elemento (*elementwise multiplication*):

```
>>> v1 = Vector([1, 2, 3])  
>>> v1 * 10  
Vector([10.0, 20.0, 30.0])  
>>> 11 * v1  
Vector([11.0, 22.0, 33.0])
```



Outro tipo de multiplicação envolvendo vetores é o produto escalar (*dot product*). Os operandos de um produto escalar são dois vetores, e o resultado é um número escalar (não um vetor). É como uma multiplicação de matrizes, considerando um vetor como uma matriz de $1 \times N$ e o outro como uma matriz de $N \times 1$. Implementaremos o produto escalar em `Vector` na Seção 16.6.

Voltando à nossa multiplicação por escalar, começamos novamente com os métodos `__mul__` e `__rmul__` mais simples possíveis que possam funcionar:

```
# dentro da classe Vector

def __mul__(self, scalar):
    return Vector(n * scalar for n in self)

def __rmul__(self, scalar):
    return self * scalar
```

Estes métodos funcionam, exceto quando recebem operandos incompatíveis. O argumento `scalar` precisa ser um número que, quando multiplicado por um `float`, produz outro `float` (porque nossa classe `Vector` armazena um array de números de ponto flutuante). Então um `complex` não serve, mas pode ser um `int`, um `bool` (`bool` é subclasse de `int`) ou até uma instância de `fractions.Fraction`. No Exemplo 11, o método `__mul__` não faz nenhuma checagem de tipos explícita com `scalar`. Em vez disso, o converte para `float`, e devolve `NotImplemented` se a conversão falhar. É mais um exemplo prático de tipagem pato.

Exemplo 11. `vector_v7.py`: métodos do operador `` adicionados*

```
class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    # métodos omitidos; código completo em https://fpy.li/code

    def __mul__(self, scalar):
        try:
            factor = float(scalar)
        except TypeError: ①
            return NotImplemented ②
        return Vector(n * factor for n in self)

    def __rmul__(self, scalar):
        return self * scalar ③
```

- ① Se scalar não pode ser convertido para float...
- ② ...não temos como lidar com ele, então devolvemos `NotImplemented`, para permitir ao Python tentar `__rmul__` no operando scalar.
- ③ Neste exemplo, `__rmul__` funciona bem apenas executando `self * scalar`, que delega a operação para o método `__mul__`.

Com o Exemplo 11, é possível multiplicar um `Vector` por valores escalares de tipos numéricos comuns e não tão comuns:

```
>>> v1 = Vector([1.0, 2.0, 3.0])
>>> 14 * v1
Vector([14.0, 28.0, 42.0])
>>> v1 * True
Vector([1.0, 2.0, 3.0])
>>> from fractions import Fraction
>>> v1 * Fraction(1, 3)
Vector([0.3333333333333333, 0.6666666666666666, 1.0])
```

Agora que podemos multiplicar `Vector` por valores escalares, vamos ver como implementar o produto de um `Vector` por outro `Vector`.



Na primeira edição de *Python Fluente*, usei tipagem ganso no Exemplo 11: checava o argumento `scalar` de `__mul__` com `isinstance(scalar, numbers.Real)`. Agora evito usar as ABCs de `numbers`, por não serem suportadas pelas anotações de tipo introduzidas na PEP 484. Usar durante a execução tipos que não podem ser também checados de forma estática me parece uma má ideia.

Outra alternativa seria checar com o protocolo `typing.SupportsFloat`, que vimos na Seção 13.6.2. Escolhi usar tipagem pato naquele exemplo por considerar que pythonistas fluentes devem se sentir confortáveis com esse padrão de programação.

Mas `__matmul__`, no Exemplo 12, que é novo e foi escrito para essa segunda edição, é um bom exemplo de tipagem ganso.

16.6. Usando @ como operador infix

O símbolo @ é o prefixo de decoradores de função, mas desde 2015 também pode ser usado como um operador infix.

Por muitos anos, o produto escalar (*dot product*) era escrito como `numpy.dot(a, b)` na biblioteca NumPy. A notação de invocação de função faz com que fórmulas mais longas sejam difíceis de traduzir da notação matemática para Python,^[4] então a comunidade de computação numérica fez campanha pela *PEP 465*—*A dedicated infix operator for matrix multiplication* [fpy.li/pep465] (Um operador infix dedicado para multiplicação de matrizes), que foi implementada no Python 3.5. Hoje podemos escrever `a @ b` para computar o produto de dois arrays da NumPy.

O operador @ é suportado pelos métodos especiais `__matmul__`, `__rmatmul__` e `__imatmul__`, cujos nomes derivam de "matrix multiplication". Até o Python 3.10, estes métodos não são usados em lugar algum na biblioteca padrão, mas são reconhecidos pelo interpretador desde o Python 3.5, então nós e os desenvolvedores da NumPy podemos implementar o operador @ em nossas classes. O analisador sintático de Python foi modificado para aceitar o novo operador, pois `a @ b` era um erro de sintaxe até o Python 3.4.

Estes testes simples mostram como @ deve funcionar com instâncias de Vector:

```
>>> va = Vector([1, 2, 3])
>>> vz = Vector([5, 6, 7])
>>> va @ vz == 38.0 # 1*5 + 2*6 + 3*7
True
>>> [10, 20, 30] @ vz
380.0
>>> va @ 3
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for @: 'Vector' and 'int'
```

O resultado de `va @ vz` no exemplo acima é o mesmo que obtemos no NumPy fazendo o produto escalar de arrays com os mesmos valores:

```
>>> import numpy as np
>>> np.array([1, 2, 3]) @ np.array([5, 6, 7])
38
```

O Exemplo 12 mostra o código dos métodos especiais relevantes na classe `Vector`.

Exemplo 12. `vector_v7.py`: métodos para o operador `@`

```
class Vector:
    # vários métodos omitidos nesta listagem

    def __matmul__(self, other):
        if (isinstance(other, abc.Sized) and ①
            isinstance(other, abc.Iterable)):
            if len(self) == len(other): ②
                return sum(a * b for a, b in zip(self, other)) ③
            else:
                raise ValueError('@ requires vectors of equal length.')
        else:
            return NotImplemented

    def __rmatmul__(self, other):
        return self @ other
```

① Ambos os operandos precisam implementar `__len__` e `__iter__`...

② ...e ter o mesmo tamanho, para permitir...

③ ...uma linda aplicação de `sum`, `zip` e uma expressão geradora.



Desde o Python 3.10, a função `zip` aceita um argumento opcional apenas nomeado, `strict`. Quando `strict=True`, a função gera um `ValueError` se um iterável termina antes de outro. O default é `False`. Este comportamento se alinha à filosofia de «falhar rápido» [fpy.li/16-8] de Python. No Exemplo 12, poderíamos trocar o `if` interno por um `try/except` `ValueError` e acrescentar `strict=True` à invocação de `zip`. Neste caso específico, como `self` e `other` suportam `__len__`, prefiro o teste explícito com `if`, por clareza. O `strict` é mais útil quando o `zip` vai lidar com iteradores, que não têm `__len__`.

O Exemplo 12 é um bom exemplo prático de tipagem ganso. Não usamos `isinstance(other, Vector)`, porque queremos oferecer mais flexibilidade para os usuários. Suportamos operandos que sejam instâncias de `abc.Sized` e `abc.Iterable`. Estas duas ABCs implementam o `__subclasshook__`, portanto qualquer objeto que forneça `__len__` e `__iter__` satisfaz nosso teste—não há necessidade de criar subclasses concretas dessas ABCs ou sequer registrar-se com elas, como explicado na Seção 13.5.8. Em particular, nossa classe `Vector` não é subclasse nem de `abc.Sized` nem de `abc.Iterable`, mas passa os testes de `isinstance` contra aquelas ABCs, pois implementa os métodos necessários.

Vamos revisar os operadores aritméticos suportados pelo Python antes de mergulhar na categoria especial dos operadores de comparação rica (Seção 16.8).

16.7. Resumindo os operadores aritméticos

Ao implementar `+`, `*`, e `@`, vimos os padrões de programação mais comuns para operadores infixos. As técnicas descritas são aplicáveis a todos os operadores listados na Tabela 2 (os operadores de atribuição aritmética serão tratados na Seção 16.9).

Tabela 2. Nomes dos métodos de operadores infixos (os operadores internos são usados para atribuição aumentada; operadores de comparação estão na Tabela 3)

op	direto	reverso	interno	descrição
<code>+</code>	<code>__add__</code>	<code>__radd__</code>	<code>__iadd__</code>	Adição ou concatenação
<code>-</code>	<code>__sub__</code>	<code>__rsub__</code>	<code>__isub__</code>	Subtração
<code>*</code>	<code>__mul__</code>	<code>__rmul__</code>	<code>__imul__</code>	Multiplicação ou repetição
<code>/</code>	<code>__truediv__</code>	<code>__rtruediv__</code>	<code>__itruediv__</code>	Divisão exata
<code>//</code>	<code>__floordiv__</code>	<code>__rfloordiv__</code>	<code>__ifloordiv__</code>	Divisão inteira
<code>%</code>	<code>__mod__</code>	<code>__rmod__</code>	<code>__imod__</code>	Módulo (resto)
<code>divmod()</code>	<code>__divmod__</code>	<code>__rdivmod__</code>	<code>__idivmod__</code>	Devolve uma tupla com o quociente da divisão inteira e o módulo
<code>**</code> , <code>pow()</code>	<code>__pow__</code>	<code>__rpow__</code>	<code>__ipow__</code>	Exponenciação ^[5]
<code>@</code>	<code>__matmul__</code>	<code>__rmatmul__</code>	<code>__imatmul__</code>	Multiplicação de matrizes

op	direto	reverso	interno	descrição
&	<code>__and__</code>	<code>__rand__</code>	<code>__iand__</code>	E binário (bit a bit)
	<code>__or__</code>	<code>__ror__</code>	<code>__ior__</code>	OU binário (bit a bit)
^	<code>__xor__</code>	<code>__rxor__</code>	<code>__ixor__</code>	XOR binário (bit a bit)
<<	<code>__lshift__</code>	<code>__rlshift__</code>	<code>__ilshift__</code>	Deslocamento de bits para a esquerda
>>	<code>__rshift__</code>	<code>__rrshift__</code>	<code>__irshift__</code>	Deslocamento de bits para a direita

Operadores de comparação rica usam regras diferentes.

16.8. Operadores de comparação rica

O tratamento dos operadores de comparação rica `==`, `!=`, `>`, `<`, `>=` e `<=` pelo interpretador Python é similar ao que já vimos, com uma importante diferença: não existem métodos reversos com o prefixo `__r...__`. Os mesmos métodos são usados para invocações diretas ou reversas do operador. As regras estão resumidas na Tabela 3.

Tabela 3. Comparação rica: a última coluna mostra o resultado quando as tentativas devolvem `NotImplemented` ou o operando não implementa o método.

grupo	op	invocação direta	invocação reversa	quando não implementado
igualdade	<code>a == b</code>	<code>a.__eq__(b)</code>	<code>b.__eq__(a)</code>	Devolve <code>id(a) == id(b)</code>
	<code>a != b</code>	<code>a.__ne__(b)</code>	<code>b.__ne__(a)</code>	Devolve <code>not (a == b)</code>
ordenação	<code>a > b</code>	<code>a.__gt__(b)</code>	<code>b.__lt__(a)</code>	Levanta <code>TypeError</code>
	<code>a < b</code>	<code>a.__lt__(b)</code>	<code>b.__gt__(a)</code>	Levanta <code>TypeError</code>
	<code>a >= b</code>	<code>a.__ge__(b)</code>	<code>b.__le__(a)</code>	Levanta <code>TypeError</code>
	<code>a <= b</code>	<code>a.__le__(b)</code>	<code>b.__ge__(a)</code>	Levanta <code>TypeError</code>

Por exemplo, no caso de `==`, tanto a chamada direta quanto a reversa invocam `__eq__`, apenas permutando os argumentos. Uma chamada direta a `__gt__` pode ser seguida de uma chamada reversa a `__lt__`, com os argumentos permutados.

Nos casos de `==` e `!=`, se o método não existe no segundo operando, ou devolve `NotImplemented`, os métodos correspondentes `__eq__` e `__ne__` herdados da classe `object` comparam os IDs dos objetos, então não ocorre `TypeError`.

Considerando estas regras, vamos revisar e aperfeiçoar o comportamento do método `Vector.__eq__`, escrito assim no `vector_v5.py` (Exemplo 16 do Capítulo 12):

```
class Vector:
    # várias linhas omitidas

    def __eq__(self, other):
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))
```

Este método produz os resultados do Exemplo 13.

Exemplo 13. Comparando um `Vector` a um `Vector`, a um `Vector2d`, e a uma tupla

```
>>> va = Vector([1.0, 2.0, 3.0])
>>> vb = Vector(range(1, 4))
>>> va == vb ①
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> vc == v2d ②
True
>>> t3 = (1, 2, 3)
>>> va == t3 ③
True
```

- ① Duas instâncias de `Vector` com componentes numéricos iguais são iguais.
- ② Um `Vector` e um `Vector2d` também são iguais se seus componentes são iguais.
- ③ Um `Vector` também é considerado igual a uma tupla ou qualquer sequência com itens escalares de valor igual.

O último resultado no Exemplo 13 pode ser indesejável. Queremos mesmo que um `Vector` seja considerado igual a uma tupla contendo os mesmos números?

Não tenho uma regra fixa sobre isso; depende do contexto da aplicação. O "Zen of Python" diz:

Em face da ambiguidade, rejeite a tentação de adivinhar.

Liberalidade excessiva na avaliação de operandos pode levar a resultados surpreendentes, e programadores odeiam surpresas.

Buscando inspiração no próprio Python, vemos que `[1, 2] == (1, 2)` é `False`. Então, seremos conservadores e faremos checagem de tipos. Se o segundo operando for uma instância de `Vector` (ou uma instância de uma subclasse de `Vector`), então usaremos a mesma lógica do `__eq__` atual. Caso contrário, devolvemos `NotImplemented` e deixamos Python cuidar do caso. Veja o Exemplo 14.

Exemplo 14. `vector_v8.py`: `__eq__` aperfeiçoado na classe `Vector`

```
def __eq__(self, other):
    if isinstance(other, Vector): ①
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))
    else:
        return NotImplemented ②
```

- ① Se o operando `other` é uma instância de `Vector` (ou de uma subclasse de `Vector`), executa a comparação como antes.
- ② Caso contrário, devolve `NotImplemented`.

Rodando os testes do Exemplo 13 com o novo `Vector.__eq__` do Exemplo 14, obtemos os resultados do Exemplo 15.

Exemplo 15. Mesmas comparações do Exemplo 13: o último resultado mudou

```
>>> va = Vector([1.0, 2.0, 3.0])
>>> vb = Vector(range(1, 4))
>>> va == vb ①
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> vc == v2d ②
```

```
True
>>> t3 = (1, 2, 3)
>>> va == t3 ③
False
```

- ① Mesmo resultado de antes, como esperado.
- ② Mesmo resultado de antes, mas por quê? Explicação a seguir.
- ③ Resultado diferente; era o que queríamos. Mas por que isso funciona?
Continue lendo...

Dos três resultados no Exemplo 15, o primeiro não é novidade, mas os dois últimos foram causados por `__eq__` devolver `NotImplemented` no Exemplo 14. Eis o que acontece no exemplo com um `Vector` e um `Vector2d`, `vc == v2d`, passo a passo:

1. Para avaliar `vc == v2d`, Python invoca `Vector.eq(vc, v2d)`.
2. `Vector.__eq__(vc, v2d)` verifica que `v2d` não é um `Vector` e devolve `NotImplemented`.
3. Diante do resultado `NotImplemented`, Python tenta `Vector2d.__eq__(v2d, vc)`.
4. `Vector2d.__eq__(v2d, vc)` transforma os dois operandos em tuplas e os compara: o resultado é `True` (o código de `Vector2d.__eq__` está no Exemplo 11 do Capítulo 11).

Já para a comparação `va == t3`, entre `Vector` e `tuple` no Exemplo 15, os passos são:

1. Para avaliar `va == t3`, Python invoca `Vector.__eq__(va, t3)`.
2. `Vector.__eq__(va, t3)` verifica que `t3` não é um `Vector` e devolve `NotImplemented`.
3. Diante do resultado `NotImplemented`, Python tenta `tuple.__eq__(t3, va)`.
4. `tuple.__eq__(t3, va)` não tem a menor ideia do que seja um `Vector`, então devolve `NotImplemented`.
5. No caso especial de `==`, se a chamada reversa devolve `NotImplemented`, Python compara os IDs dos objetos, como último recurso.

Não precisamos implementar `__ne__` para `!=`, pois o comportamento alternativo do `__ne__` herdado de `object` nos serve: quando `__eq__` é definido e não devolve `NotImplemented`, `__ne__` devolve a negação booleana do resultado de `__eq__`.

Em outras palavras, dados os mesmos objetos que usamos no Exemplo 15, os resultados de `!=` são consistentes:

```
>>> va != vb
False
>>> vc != v2d
False
>>> va != (1, 2, 3)
True
```

O `__ne__` herdado de `object` funciona como o código abaixo (mas o original é escrito em C):^[6]

```
def __ne__(self, other):
    eq_result = self == other
    if eq_result is NotImplemented:
        return NotImplemented
    else:
        return not eq_result
```

Vimos o básico da sobrecarga de operadores infixos. Agora veremos uma categoria diferente: os operadores de atribuição aumentada.

16.9. Operadores de atribuição aumentada

Nossa classe `Vector` já suporta os operadores de atribuição aumentada `+=` e `*=`. Isso acontece porque a atribuição aumentada trabalha com sequências imutáveis criando novas instâncias e re-vinculando a variável à esquerda do operador.

O Exemplo 16 os mostra em ação.

*Exemplo 16. Usando += e *= com instâncias de Vector*

```
>>> v1 = Vector([1, 2, 3])
>>> v1_alias = v1 ①
>>> id(v1) ②
4302860128
>>> v1 += Vector([4, 5, 6]) ③
>>> v1 ④
Vector([5.0, 7.0, 9.0])
>>> id(v1) ⑤
4302859904
>>> v1_alias ⑥
Vector([1.0, 2.0, 3.0])
>>> v1 *= 11 ⑦
>>> v1 ⑧
Vector([55.0, 77.0, 99.0])
>>> id(v1)
4302858336
```

- ① Cria um alias, para podermos inspecionar o objeto `Vector([1, 2, 3])` mais tarde.
- ② Verifica o `id` do `Vector` inicial, vinculado a `v1`.
- ③ Executa a adição aumentada.
- ④ O resultado esperado...
- ⑤ ...mas foi criado um novo `Vector`.
- ⑥ Inspeciona `v1_alias` para confirmar que o `Vector` original não foi alterado.
- ⑦ Executa a multiplicação aumentada.
- ⑧ Novamente, o resultado é o esperado, mas um novo `Vector` foi criado.

Se uma classe não implementa os métodos internos listados na Tabela 2, os operadores de atribuição aumentada funcionam como açúcar sintático: `a += b` é avaliado exatamente como `a = a + b`. Este é o comportamento esperado para tipos imutáveis, e se você fornecer `__add__`, então `+=` funcionará sem qualquer código adicional.

Entretanto, se você implementar um método interno tal como `__iadd__`, aquele método será chamado para computar o resultado de `a += b`. Como indica seu nome, espera-se que esses operadores modifiquem internamente o operando da esquerda^[7], e não criem um novo objeto como resultado.



Nunca devemos implementar métodos internos para atribuição aumentada em tipos imutáveis como nossa classe `Vector`. Pode ser óbvio, mas vale a pena enfatizar. Por este motivo, deixaremos de lado o tema dos vetores nos próximos exemplos.

Para mostrar o código de um método interno de atribuição aumentada, vamos estender a classe `BingoCage` do Exemplo 10 do Capítulo 13 para implementar `__add__` e `__iadd__`.

Vamos chamar a subclasse de `AddableBingoCage`. Os doctests da classe (Exemplo 17) mostram o comportamento esperado do operador `+`.

Exemplo 17. O operador `+` cria uma nova instância de `AddableBingoCage`

```
>>> vowels = 'AEIOU'
>>> globe = AddableBingoCage(vowels) ①
>>> globe.inspect()
('A', 'E', 'I', 'O', 'U')
>>> globe.pick() in vowels ②
True
>>> len(globe.inspect()) ③
4
>>> globe2 = AddableBingoCage('XYZ') ④
>>> globe3 = globe + globe2
>>> len(globe3.inspect()) ⑤
7
>>> void = globe + [10, 20] ⑥
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'AddableBingoCage' and
'list'
```

① Cria uma instância de `globe` com cinco itens (cada uma das `vowels`).

② Extrai um dos itens, e verifica que é uma das `vowels`.

- ③ Confirma que globe tem agora quatro itens.
- ④ Cria uma segunda instância, com três itens.
- ⑤ Cria uma terceira instância pela soma das duas anteriores. Esta instância tem sete itens.
- ⑥ Tentar adicionar uma AddableBingoCage a uma list falha com um TypeError. A mensagem de erro é produzida pelo interpretador de Python quando nosso método `__add__` devolve `NotImplemented`.

Como uma `AddableBingoCage` é mutável, o Exemplo 18 mostra como ela funcionará quando implementarmos `__iadd__`.

Exemplo 18. Uma `AddableBingoCage` existente pode ser carregada com += (continuando do Exemplo 17)

```
>>> globe_orig = globe ①
>>> len(globe.inspect()) ②
4
>>> globe += globe2 ③
>>> len(globe.inspect())
7
>>> globe += ['M', 'N'] ④
>>> len(globe.inspect())
9
>>> globe is globe_orig ⑤
True
>>> globe += 1 ⑥
Traceback (most recent call last):
...
TypeError: right operand in += must be 'Tombola' or an iterable
```

- ① Cria um alias para podermos checar a identidade do objeto mais tarde.
- ② globe tem quatro itens aqui.
- ③ Uma instância de `AddableBingoCage` pode receber itens de outra instância da mesma classe.
- ④ O operador à direita de `+=` também pode ser qualquer iterável.
- ⑤ Durante todo esse exemplo, `globe` sempre se refere ao mesmo objeto que `globe_orig`.

- ⑥ Tentar adicionar um não-iterável a uma `AddableBingoCage` falha com uma mensagem de erro apropriada.

Observe que o operador `+=` é mais liberal que `+` quanto ao segundo operando. Com `+`, queremos que ambos os operandos sejam do mesmo tipo (neste caso, `AddableBingoCage`), pois se aceitássemos tipos diferentes, isso poderia causar confusão quanto ao tipo do resultado, violando a propriedade comutativa da adição. Com o `+=`, a situação é mais clara: o objeto à esquerda do operador é atualizado internamente, então não há dúvida quanto ao tipo do resultado.



Validei os comportamentos diversos de `+` e `+=` observando como funciona o tipo embutido `list`. Ao escrever `my_list + x`, você só pode concatenar uma `list` a outra `list`, mas se você escrever `my_list += x`, você pode estender a lista da esquerda com itens de qualquer iterável `x` à direita do operador. É assim que o método `list.extend()` funciona: ele aceita qualquer argumento iterável.

Agora que vimos o comportamento desejado para `AddableBingoCage`, podemos estudar sua implementação no Exemplo 19. Lembre-se de que `BingoCage`, (Exemplo 10 do Capítulo 13), é uma subclasse concreta da `ABC Tombola` do Exemplo 7 do Capítulo 13.

Exemplo 19. `bingoaddable.py`: `AddableBingoCage` é subclasse de `BingoCage` com suporte aos operadores `+` e `+=`

```
from tombola import Tombola
from bingo import BingoCage

class AddableBingoCage(BingoCage): ①

    def __add__(self, other):
        if isinstance(other, Tombola): ②
            return AddableBingoCage(self.inspect() + other.inspect())
        else:
            return NotImplemented

    def __iadd__(self, other):
        if isinstance(other, Tombola):
```



```

        other_iterable = other.inspect() ③
    else:
        try:
            other_iterable = iter(other) ④
        except TypeError: ⑤
            msg = ('right operand in += must be '
                  "'Tombola' or an iterable")
            raise TypeError(msg)
    self.load(other_iterable) ⑥
    return self ⑦

```

- ① AddableBingoCage estende BingoCage.
- ② Nosso `__add__` só vai funcionar se o segundo operando for uma instância de `Tombola`.
- ③ Em `__iadd__`, obtém os itens de `other`, se for uma instância de `Tombola`.
- ④ Caso contrário, tenta obter um iterador sobre `other` (estudaremos a função embutida `iter` no «Capítulo 17» [fpy.li/17] (vol.3)).
- ⑤ Se aquilo falhar, gera uma exceção explicando o que o usuário deve fazer. Sempre que possível, mensagens de erro devem orientar o usuário para a solução.
- ⑥ Se chegamos até aqui, podemos carregar o `other_iterable` em `self`.
- ⑦ Muito importante: os métodos especiais de atribuição aumentada de objetos mutáveis devem devolver `self`. É o que os usuários esperam.

Podemos resumir toda a ideia dos operadores de atribuição interna comparando as instruções `return` que devolvem os resultados em `__add__` e em `__iadd__` no Exemplo 19:

`__add__`: O resultado é computado chamando o construtor `AddableBingoCage` para criar uma nova instância.

`__iadd__`: O resultado é `self`, após ele ter sido modificado.

Uma última observação sobre o Exemplo 19: não implementei `__radd__` em `AddableBingoCage`, porque não há necessidade. O método direto `__add__` só vai lidar com operandos do mesmo tipo à direita, então se Python tentar computar `a + b`, onde `a` é uma `AddableBingoCage` e `b` não, devolvemos `NotImplemented`—talvez a

classe de `b` possa fazer isso funcionar. Mas se a expressão for `b + a` e `b` não for uma `AddableBingoCage`, e devolver `NotImplemented`, então é melhor deixar Python desistir e gerar um `TypeError`, pois não temos como tratar `b`.



Se um método de operador infixo direto (por exemplo `__mul__`) é projetado para funcionar apenas com operandos do mesmo tipo de `self`, é inútil implementar o método reverso correspondente (por exemplo, `__rmul__`) pois, por definição, esse método só será invocado quando estivermos lidando com um operando de um tipo diferente.

Assim terminamos nossa exploração de sobrecarga de operadores no Python.

16.10. Resumo do capítulo

Começamos o capítulo revisando algumas restrições impostas pelo Python à sobrecarga de operadores: é impossível redefinir operadores nos tipos embutidos, a sobrecarga está limitada aos operadores existentes, e alguns operadores não podem ser sobrecarregados (`is`, `and`, `or`, `not`).

Colocamos a mão na massa com os operadores unários, implementando `__neg__` e `__pos__`. A seguir vieram os operadores infixos, começando por `+`, suportado pelo método `__add__`. Vimos que operadores unários e infixos devem produzir resultados criando novos objetos, sem nunca modificar seus operandos. Para suportar operações com outros tipos, devolvemos o valor especial `NotImplemented` (não uma exceção) permitindo ao interpretador tentar novamente chamando o método especial reverso do segundo operando (por exemplo, `__radd__`). O algoritmo usado pelo Python para tratar operadores infixos está resumido no fluxograma da Figura 1.

Misturar operandos de mais de um tipo exige detectar os operandos que não podemos tratar. Neste capítulo fizemos isso de duas maneiras: ao modo da tipagem pato, apenas fomos em frente e tentamos a operação, capturando uma exceção de `TypeError` se ela acontecesse; mais tarde, em `__mul__` e `__matmul__`, usamos um teste `isinstance` explícito. Há prós e contras nas duas abordagens: tipagem pato é mais flexível, mas a checagem explícita de tipo é mais previsível.

De modo geral, bibliotecas devem aproveitar a tipagem *pato* para lidar objetos de diferentes tipos, desde que eles suportem as operações necessárias. Entretanto, o algoritmo de despacho de operadores de Python pode produzir mensagens de erro enganosas ou resultados inesperados quando combinado com a tipagem *pato*. Por essa razão, a disciplina da checagem de tipos com invocações de `isinstance` contra ABCs é muitas vezes útil quando escrevemos métodos especiais para sobrecarga de operadores. Esta é a técnica batizada de tipagem *ganso* (*goose typing*) por Alex Martelli—como vimos na Seção 13.5. A tipagem *ganso* é um compromisso entre a flexibilidade e a segurança, porque os tipos definidos pelo usuário, existentes ou futuros, podem ser declarados como subclasses reais ou virtuais de uma ABC. Além disso, se uma ABC implementa o `__subclasshook__`, objetos podem então passar por checagens com `isinstance` contra aquela ABC apenas fornecendo os métodos exigidos—sem necessidade de ser uma subclasse ou de se registrar com a ABC.

O próximo tópico tratado foram os operadores de comparação rica. Implementamos `==` com `__eq__` e descobrimos que Python oferece uma implementação conveniente de `!=` no `__ne__` herdado da classe base `object`. A forma como Python avalia esses operadores, bem como `>`, `<`, `>=`, e `<=`, é um pouco diferente, com uma lógica especial para a escolha do método reverso, e um tratamento alternativo para `==` e `!=` que nunca gera erros, pois a classe `object` já implementa os métodos necessários.

Na última seção, nos concentramos nos operadores de atribuição aumentada. Vimos que Python os trata, por default, como uma combinação do operador simples seguido de uma atribuição: `a += b` é avaliado exatamente como `a = a + b`. Isto sempre cria um novo objeto, então funciona para tipos mutáveis ou imutáveis.

Para objetos mutáveis, podemos implementar métodos especiais de atualização interna, tal como `__iadd__` para `+=`, e alterar o valor do operando à esquerda. Para demonstrar isto na prática, implementamos uma subclasse de `BingoCage`, suportando `+=` para adicionar itens ao reservatório de itens para sorteio, de modo similar à forma como o tipo embutido `list` suporta `+=` como um atalho para o método `list.extend()`. Vimos que `+` tende a ser mais estrito que `+=` em relação aos tipos aceitos. Em sequências, `+` normalmente exige que ambos os operandos sejam do mesmo tipo, enquanto `+=` muitas vezes aceita qualquer iterável como o operando à direita do operador.

16.11. Para saber mais

Guido van Rossum escreveu uma boa apologia da sobrecarga de operadores em *Why operators are useful* [fpy.li/16-10] (Porque operadores são úteis). Trey Hunner postou *Tuple ordering and deep comparisons in Python* [fpy.li/16-11] (Ordenação de tuplas e comparações profundas em Python), argumentando que os operadores de comparação rica de Python são mais flexíveis e poderosos do que os programadores vindos de outras linguagens costumam pensar.

A sobrecarga de operadores é uma área da programação em Python onde testes com `isinstance` são comuns. A melhor prática relacionada a tais testes é a tipagem ganso, tratada na Seção 13.5. Se você pulou essa parte, assegure-se de voltar lá e ler aquela seção.

A principal referência para os métodos especiais de operadores é o capítulo Modelo de Dados [fpy.li/2j] na documentação de Python. Outra leitura relevante é Implementando as operações aritméticas [fpy.li/7r] no módulo `numbers` da biblioteca padrão de Python.

Um exemplo brilhante de sobrecarga de operadores apareceu no pacote `pathlib` [fpy.li/16-13], a partir do Python 3.4. Sua classe `Path` sobrecarrega o operador `/` para construir caminhos do sistema de arquivos a partir de strings, como mostra o exemplo abaixo, da documentação:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
```

Outro exemplo não aritmético de sobrecarga de operadores está na biblioteca `Scapy` [fpy.li/16-14], usada para "enviar, farejar, dissecar e forjar pacotes de rede". Na `Scapy`, o operador `/` cria pacotes empilhando campos de diferentes camadas da rede. Veja *Stacking layers* [fpy.li/16-15] (Empilhando camadas) para mais detalhes.

Se você está prestes a implementar operadores de comparação, estude `functools.total_ordering`. Esse é um decorador de classes que gera automaticamente os métodos para todos os operadores de comparação rica em qualquer classe que defina ao menos alguns deles. Veja a documentação do módulo `functools` [fpy.li/7q].

Se tiver curiosidade sobre o despacho de métodos de operadores em linguagens com tipagem dinâmica, duas leituras fundamentais são *A Simple Technique for Handling Multiple Polymorphism* [fpy.li/16-17] (Uma técnica simples para tratar polimorfismo múltiplo), de Dan Ingalls (membro da equipe original de Smalltalk), e *Arithmetic and Double Dispatching in Smalltalk-80* [fpy.li/16-18] (Aritmética e despacho duplo no Smalltalk-80), de Kurt J. Hebel e Ralph Johnson (Johnson ficou famoso como um dos autores do livro *Padrões de Projetos* original).

Os dois artigos discutem em profundidade o poder do polimorfismo em linguagens com tipagem dinâmica, como Smalltalk, Python e Ruby. Python não implementa despacho duplo exatamente como descrito naqueles artigos. O algoritmo de despacho duplo em Python, usando operadores diretos e reversos, é mais fácil de suportar em classes definidas pelo usuário que o despacho duplo clássico, mas exige tratamento especial pelo interpretador. Por outro lado, o despacho duplo clássico é uma técnica geral, que pode ser usada no Python ou em qualquer linguagem orientada a objetos, para além do contexto específico de operadores infixos. E, de fato, Ingalls, Hebel e Johnson usam exemplos muito diferentes para descrever essa técnica.

O texto *The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling* [fpy.li/16-1] (A Família de Linguagens C: entrevista com Dennis Ritchie, Bjarne Stroustrup, e James Gosling), de onde tirei a epígrafe deste capítulo, apareceu na *Java Report*, 5(7), julho de 2000, e na *C++ Report*, 12(7), julho/agosto de 2000, juntamente com outros trechos que usei no Ponto de Vista deste capítulo (logo adiante). Se você se interessa pelo design de linguagens de programação, faça um favor a si mesmo e leia aquela entrevista.

Ponto de Vista

Sobrecarga de operadores: prós e contras

James Gosling, citado no início deste capítulo, tomou a decisão consciente de excluir a sobrecarga de operadores quando projetou o Java. Na entrevista *The C Family of Languages* [fpy.li/16-1] ele diz:

Talvez uns 20 a 30% da população acha que sobrecarga de operadores é obra do demônio; alguém fez algo com sobrecarga de operadores que realmente os tirou do sério, porque usaram algo como + para inserção em listas, e isso torna a vida muito, muito confusa. Muito do problema vem do fato de existirem apenas uma meia dúzia de operadores que podem ser sobrecarregados de forma razoável, mas existem milhares ou milhões de operadores que as pessoas gostariam de definir—então é preciso escolher, e muitas vezes as escolhas entram em conflito com a sua intuição.

Guido van Rossum escolheu o caminho do meio no suporte à sobrecarga de operadores: ele não deixou a porta aberta para que os usuários criassem novos operadores arbitrários como `<=>` ou `: -`), evitando uma Torre de Babel de operadores customizados, e que o analisador sintático de Python continue simples. Python também não permite a sobrecarga dos operadores dos tipos embutidos, outra limitação que promove a legibilidade e o desempenho previsível.

Gosling continua:

E então há uma comunidade de aproximadamente 10% que havia de fato usado a sobrecarga de operadores de forma apropriada, e que realmente gostavam disso, e para quem isso era realmente importante; essas são quase exclusivamente pessoas que fazem trabalho numérico, onde a notação é muito importante para avivar a intuição [das pessoas], porque elas vêm com uma intuição sobre o que + significa, e a poder dizer $a + b$, onde a e b são números complexos ou matrizes ou alguma outra coisa, realmente faz sentido.

Também há benefícios em não permitir a sobrecarga de operadores em uma linguagem. Já ouvi o argumento de que C é melhor que C++; para programação de sistemas, porque a sobrecarga de operadores em C++ pode fazer com que operações dispendiosas pareçam triviais. Duas linguagens modernas bem sucedidas, que compilam para executáveis binários, fizeram escolhas opostas: Go não tem sobrecarga de operadores, Rust tem [fpy.li/16-21].

Mas operadores sobrecarregados, quando usados de forma sensata, tornam o código mais fácil de ler e escrever. É um ótimo recurso em uma linguagem de alto nível moderna.

Um exemplo de avaliação preguiçosa

Se você olhar de perto o *traceback* no Exemplo 9, vai encontrar evidências da avaliação preguiçosa [fpy.li/16-22] de expressões geradoras. O Exemplo 20 é o mesmo *traceback*, agora com explicações.

Exemplo 20. Mesmo que o Exemplo 9

```
>>> v1 + 'ABC'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 329, in __add__
    return Vector(a + b for a, b in pairs) ①
  File "vector_v6.py", line 243, in __init__
    self._components = array(self.typecode, components) ②
  File "vector_v6.py", line 329, in <genexpr>
    return Vector(a + b for a, b in pairs) ③
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

- ① A chamada a `Vector` recebe uma expressão geradora como seu argumento `components`. Nenhum problema nesse estágio.
- ② A `genexp` `components` é passada para o construtor de `array`. Dentro do construtor de `array`, Python tenta iterar sobre a `genexp`, causando a avaliação do primeiro item `a + b`. É quando ocorre o `TypeError`.
- ③ A exceção se propaga para a chamada ao construtor de `Vector`, onde é relatada.

Isso mostra como a expressão geradora é avaliada no último instante possível, e não onde é definida no código-fonte.

Se, por outro lado, o construtor de `Vector` fosse invocado como `Vector([a + b for a, b in pairs])`, então a exceção ocorreria bem ali, porque a compreensão de lista tentou criar uma `list` para ser passada como argumento para a chamada a `Vector()`. O corpo de `Vector.__init__` nunca seria alcançado.

O «Capítulo 17» [fpy.li/17] (vol.3) vai tratar das expressões geradoras em detalhes, mas eu não queria deixar essa demonstração acidental de sua natureza preguiçosa passar despercebida.

[1] Fonte: *The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling* [fpy.li/16-1] (A Família de Linguagens C: entrevista com Dennis Ritchie, Bjarne Stroustrup, e James Gosling).

[2] As demais ABCs na biblioteca padrão de Python funcionam bem para tipagem ganso e tipagem estática. O problema com as ABCs `numbers` é explicado na Seção 13.6.8.

[3] Em matemática, um "escalar" é um número que pode ser representado por um ponto em uma linha, ou "escala". Em Python, instâncias de `int`, `float`, `decimal.Decimal` e `fraction.Fraction` são escalares, mas um `complex` não é um escalar.

[4] Veja o Ponto de Vista para uma discussão deste problema.

[5] `pow` pode receber um terceiro argumento opcional, `modulo`: `pow(a, b, modulo)`, também suportado pelos métodos especiais quando invocados diretamente (por exemplo, `a.__pow__(b, modulo)`).

[6] A lógica de `object.__eq__` e `object.__ne__` está na função `object_richcompare` em *Objects/typeobject.c* [fpy.li/16-9], no código-fonte do CPython.

[7] NT: O prefixo "i" nos nomes destes métodos se refere a *in-place*, traduzido como "interno" na documentação brasileira oficial de Python.